



ISP DDK

Imagination Technologies

Strictly Confidential

Release 2.8.4- External- August 05, 2016- CL: 4173926

Copyright © 2016 Imagination Technologies Limited. All Rights Reserved.
This document is confidential. Neither the whole nor any part of the information contained in,
nor the product described in, this document may be adapted or reproduced in any material
form except with the written permission of Imagination Technologies Limited. Imagination
Technologies, the Imagination logo, PowerVR, MIPS, Meta, Ensigma and Codescape are
trademarks or registered trademarks of Imagination Technologies Limited. All other logos,
products, trademarks and registered trademarks are the property of their respective owners.
This document can only be distributed subject to the terms of a Non-Disclosure Agreement or
Licence with Imagination Technologies Limited.

Contents

1	Dictionary	1
1.1	Acronyms	1
1.2	V2500 modules	2
2	Getting Started Guide	6
2.1	V2500 Software	6
2.2	V2500 Hardware	7
2.3	C Simulator (CSIM)	9
2.4	System Requirements	10
2.5	Deliverables	10
2.6	Driver build steps	12
2.7	IMGVideo Support	19
2.8	HDR Libraries support	21
2.9	GUI build steps	21
2.10	Build System details	22
2.11	Doxygen documentation	25
3	Platform Integration Guide	27
3.1	Kernel Module changes	27
3.2	Device Access	28
3.3	Memory management	30
4	Sensor driver	34
4.1	Platform Integration	34
4.2	Implementing a new sensor	36
4.3	Example sensor drivers	38
5	Test Tools	40
5.1	Running modes	40
5.2	Register override	41
5.3	Common Output	42
5.4	Common Input	44
5.5	Command line application user guide	47
5.6	GUI tools applications user guide	86
6	User Tuning Guide	130
6.1	Tuning work-flow	130
6.2	Sensor BLC tuning	130
6.3	White Balance Correction Tuning	131
6.4	Sensor Lens Shading Tuning	131
6.5	Lateral Chromatic Aberration Tuning	132
6.6	Primary Denoiser Tuning	132

6.7	Defective Pixels Tuning	132
6.8	Sharpening and Secondary denoiser Tuning	134
6.9	Tone-mapping Tuning	135
6.10	Image Enhancer Tuning	139
7	Capture Interface	140
7.1	Design Choices	140
7.2	Limitations	144
7.3	HW Modules and resources	145
7.4	User/Kernel interactions	165
7.5	Interrupt Management	171
7.6	Memory Mapping Unit (MMU)	174
7.7	Shot and Buffer lifecycle	179
8	ISP Control Library	190
8.1	ISPC Architecture	190
8.2	Camera Class	193
8.3	Pipeline and SetupModule Classes	200
8.4	Control and ControlModule Classes	241
8.5	Sensor Class	278
8.6	Several Camera sharing a Sensor	280
8.7	High level parameters	283
8.8	Performance measurement	285
9	Android Camera HAL	287
9.1	Context of Camera HAL in media framework	288
9.2	Library architecture	289
9.3	Implementation of Camera HAL v3.x	305
9.4	Gralloc	315
9.5	Library configuration	318
9.6	Testing Camera HAL	322
10	Android Build Instructions	325
10.1	Build instructions	325
10.2	Building Android Camera HAL	327
10.3	Example: building V2500 software for android-x86	328
10.4	Example: Building Android Camera HAL for AOSP Lollipop	332
11	Vision Libraries Usage Instructions	336
11.1	Code structure and usage	336
11.2	Build instructions	336
12	Frequently Asked Questions	339
13	Appendix: Errors troubleshooting	340
13.1	Building GUI tools: troubleshooting	340
13.2	Running the GUI: troubleshooting	341
13.3	Kernel module insertion errors	342
14	CI Appendix: Adding a new DebugFS counter	344
15	CI Appendix: Frame Size Computation	345
15.1	Bayer output (RGGB)	346

15.2	Bayer TIFF output	346
15.3	RGB input/output	347
15.4	YUV output	347
15.5	Tiled output	348
15.6	Output sizes examples	349
16	CI Appendix: System calls mapping	353
17	CI Appendix: IOCTL tables	355
17.1	Driver commands	355
17.2	Configuration commands	357
17.3	Capture commands	361
17.4	Gasket commands	362
17.5	Internal Data Generator commands	363
18	CI Appendix: Debugging a Page Fault	366
18.1	Initial Information	366
18.2	MMU fault analysis	367
18.3	We proved that the mapping structure is correct...	369
18.4	We proved that the mapping structure was wrong...	369
19	CI Appendix: Debugging a Failed to acquire frame	370
19.1	Frames not received by hardware	371
19.2	Partial frame	371
19.3	Software Interrupt delay	372
20	ISPC Appendix: From ISPC to kernel module	373
21	Android Appendix: Debugging	382
21.1	Dumping Jpeg captures	382
22	Android Appendix: Live Tuning	383
22.1	Enabling ISPC_tcp in Android	383
22.2	Running ISPC_tcp	383

Chapter 1

Dictionary

This section contains the abbreviations used for the software and hardware components as well as the common acronyms used in the documentation.

1.1 Acronyms

C

CI: Capture Interface (DDK library).

CSIM: C Simulator.

Bit accurate software version of the hardware used for verification and early software development.

D

DDK: Driver Development Kit.

By extension used as the software package delivered.

H

HAL: Hardware Abstraction Layer.

In Android layer to be implemented when adding devices support (e.g. Camera HAL).

HW: Hardware

I

I²C: Inter-Integrated Circuit.

A bus standard, pronounced I-square-C or I-two-C.

ION: Shared memory allocator. Used in Android.

ISP: Image Signal Processor.

ISPC: ISP Control (DDK library).

M

MIPI: Mobile Industry Processor Interface.

In this document this usually refers to a CSI-2 specification. See TRM for more details.

N

NUMA: Non-Uniform Memory Access.

A computer memory design, where the memory access time depends on the memory location relative to the processor.

P

PDP: Pixel Display Pipeline.

Imagination IP to display images on screen.

PHY: Physical layer IP for an interconnection interface (e.g image input/output).

S

SCB: Serial Communication Block. Imagination's IP to handle I2C.

SW: Software

T

TAL: DDK library to access HW registers.

TRM: Technical Reference Manual.

By extension the delivered V2500 hardware document (not provided as part of software package).

1.2 V2500 modules

This section provides the list of HW and SW modules abbreviations as used in the documentation and code.

In Pipeline order:

1.2.1 Modules

GAS: Gasket.

[Data from there on is Bayer RGGB].

IIF: Imager Interface (ISPC: *Imager Interface (IIF)* (page 211)).

BLC: Black Level Correction (ISPC: *Black Level Correction (BLC)* (page 203)).

RLT: Raw Look Up Table (**HW v2 only**, register space only reserved in HW v1). (ISPC: *Raw Look-up Table (RLT)* (page 225)).

LSH: Lens Shading, also contains register for SW module: WBC (ISPC: *Lens Shading (LSH)* (page 213)).

WBC: *SW only* White Balance Correction, part of LSH module in HW (ISPC: *White Balance Correction (WBC)* (page 231)).

FLS: Filter Line Store (no registers - linestores are configured at top-level by SW).

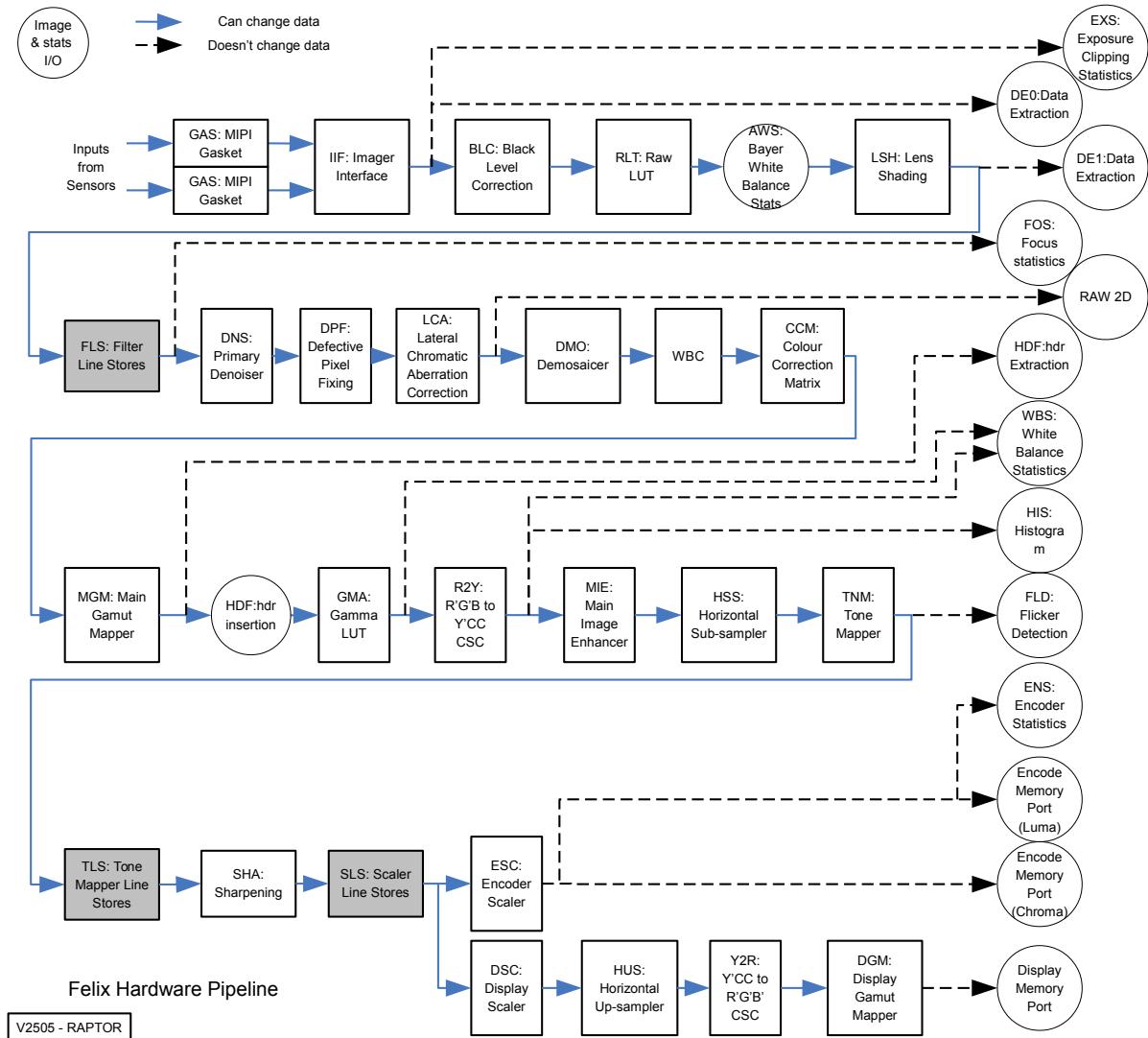


Fig. 1.1: HW Pipeline modules (adapted from HW TRM diagram).

DNS: Primary Denoiser (ISPC: *Denoiser (DNS/SHA_DN)* (page 206)).

DPF: Defective Pixel Fixing, also considered as output and input (*Defective Pixels Fixing (DPF)* (page 207)).

LCA: Lateral Chromatic Aberration (ISPC: *Lateral Chromatic Aberration (LCA)* (page 212)).

DMO: Demosaicer (no registers - DMO configured using IIF information)

[Data is converted from Bayer to RGB here].

CCM: Colour Correction Matrix (ISPC: *Colour Correction Matrix (CCM)* (page 204)).

MGM: Main Gamut Mapper (ISPC: *Main Gamut Mapper (MGM)* (page 216)).

GMA: Gamma Look Up Table - only to enable or not the usage of values stored into the GMA_LUT register bank.

R2Y: RGB to YUV converter (ISPC: *RGB to YUV (R2Y)* (page 224)).

[Data is converted from RGB to YUV 444 here].

MIE: Main Image Enhancer, SW uses MIE for the Memory Colour part of the HW block - also contains VIB module (ISPC: *Memory Image Enhancer (MIE)* (page 216)).

VIB: *SW only* Vibrancy, part of MIE module in HW (ISPC: *Vibrancy (VIB)* (page 230)).

HSS: Horizontal sub-sampler (no registers).

[Data is converted from YUV 444 to YUV 422 here].

TNM: Tone Mapper (ISPC: *Tone Mapper (TNM)* (page 226)).

FLD: Flicker Detection, also considered an output (*Flicker Detection (FLD)* (page 235)).

TLS: Tone Mapper Line Store (no registers - linestores are configured at top-level by SW).

SHA: Sharpening (ISPC: *Sharpening (SHA)* (page 225) and *Denoiser (DNS/SHA_DN)* (page 206)).

SLS: Scaler Line Stores (no registers - linestores are configured at top-level by SW).

ESC: Encoder Scaler (ISPC: *Encoder Pipeline Scaler (ESC)* (page 209)).

[Data is converted from YUV 422 to YUV 420 optionally here (or in VSS)].

VSS: Vertical Sub-sampler (no registers - information is from ESC).

[Data is converted from YUV 422 to YUV 420 optionally here (or in ESC)].

DSC: Display Scaler (ISPC: *Display Pipeline Scaler (DSC)* (page 208)).

HUS: Horizontal Up-sampler (no registers).

[Data is converted from YUV 422 to YUV 444 here].

Y2R: YUV to RGB converter (ISPC: *YUV to RGB (Y2R)* (page 232)).

[Data is converted from YUV to RGB here].

DGM: Display Gamut Mapper (ISPC: *Display Gamut Mapper (DGM)* (page 204)).

1.2.2 Outputs

All high level setups are in the ISPC *Output formats (OUT)* (page 210) section.

DEX1: (*not in diagram*) - Image data Extraction just before BLC module (Bayer).

DEX2: (*not in diagram*) - Image data Extraction just before FLS module (Bayer).

Encode: (Luma and Chroma) - YUV output at the end pipeline.

Display: RGB output at the end pipeline.

RAW 2D Extraction: Bayer extraction just before DMO module (Bayer in TIFF format).

HDR Extraction: RGB extraction just before HDF module.

1.2.3 Statistics outputs

EXS: Exposure Clipping Statistics (ISPC: *Exposure Statistics (EXS)* (page 234)).

FOS: Focus statistics (ISPC: *Focus Statistics (FOS)* (page 236)).

DPF: Defective Pixel Fixing, optional output of fixed map input (*Defective Pixels Fixing (DPF)* (page 207)).

AWS: Auto White Balance Statistics providing data useful for AWB Planckian Locus algorithm. (ISPC: *Auto White Balance Statistics (AWS)* (page 239)).

WBS: White Balance statistics (ISPC: *White Balance Statistics (WBS)* (page 238)).

HIS: Histogram (ISPC: *Luma Histogram Statistics (HIS)* (page 237)).

FLD: Flicker Detection (ISPC: *Flicker Detection (FLD)* (page 235)).

ENS: Encoder Statistics (ISPC: *Encoder Statistics (ENS)* (page 233)).

1.2.4 Image Inputs

DG, ExtDG: External Data Generator, piece of HW connected to the gasket on Imagination's FPGA system and CSIM (no available on production chips).

IIFDG or intDG: Internal Data Generator, similar to DG but available on production chips (Parallel Gasket only).

HDF: HDR insertion, insert a high dynamic range image (designed to be input of the merging of several images extracted using HDR Extraction).

Chapter 2

Getting Started Guide

This chapter aims to describe the steps to follow to compile and run a simple test using the V2500 DDK sources.

This guide is intended for people who have basic knowledge about what is an ISP. It is also assumed that the reader will have read the V2500 Product Overview and have some early understanding of what the IP is providing in Hardware and Software. The reader without such knowledge may refer to the Product Overview delivered with the Hardware package.

The customer reading this chapter is expected to know which platform they are targetting and details about such platform (e.g. IP location, is the memory shared with other IPs, etc). Knowledge about using the Linux kernel module and CMake would also be of great help to understand all the details of the steps. More details about how to run the test tools can be found in the [Test Tools](#) (page 40) section. More details about how to use the GUIs specifically can be found in the [GUI tools applications user guide](#) (page 86) section.

Building the Android package is detailed in the [Android Build Instructions](#) (page 325) section and the [Platform Integration Guide](#) (page 27) details modification the customer are expected to on the DDK to support their system.

For advanced users the [Capture Interface](#) (page 140) section has additional information about the low-level driver used. The [ISP Control Library](#) (page 190) provide more information about the ISP Control layer and the [Implemented Setup Modules](#) (page 203), [Implemented Statistics Modules](#) (page 233) and [Implemented Control Modules](#) (page 245) provide descriptions of the ISPC Modules.

2.1 V2500 Software

The V2500 Software libraries organisation is described in Figure [V2500 Software APIs](#) (page 7) and is composed of the following libraries:

- Android Camera HAL - responsible of HW abstraction layer for Android porting.
- ISP Control - responsible for setting up the pipeline to produce a good looking picture and handle AAA algorithms (Auto exposure, Auto white balance, Auto focus)
- Sensor API - responsible to implement sensor controls
- Capture Interface (CI) - responsible for managing the ISP HW and providing user-space abstraction for all other elements to use.

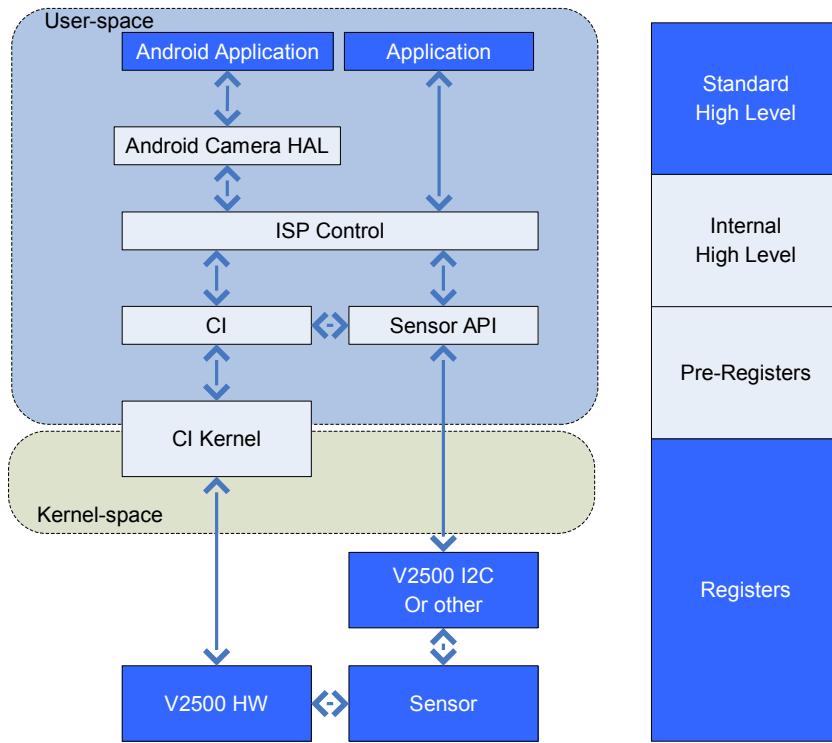


Fig. 2.1: V2500 Software APIs

2.2 V2500 Hardware

Information about the V2500 hardware (HW) should be available in the Technical Reference Manual (TRM) of the HW delivery. However this section highlights some basic information and definitions about the HW.

The HW Pipeline modules order and their acronyms is available in the *V2500 modules* (page 2) section.

To understand the basis of the HW we need to describe 3 elements of the ISP:

- The HW context (a unit that processes the image - can have multiple instances and run in parallel)
- The HW imager (not provided by Imagination - it is the actual “camera” part of the ISP)
- The HW gasket (converts the imager’s data to a common format that the HW context can understand)

The Figure *V2500 HW Organisation* (page 8) illustrates how those elements interact in the V2500 IP. The number of Gaskets and Contexts are configured and have to be chosen when selecting the HW capabilities.

Note: The HW also contains a single IIF Datagen (Imager Interface Data-generator) which can replace a gasket’s data to insert frames from memory instead.

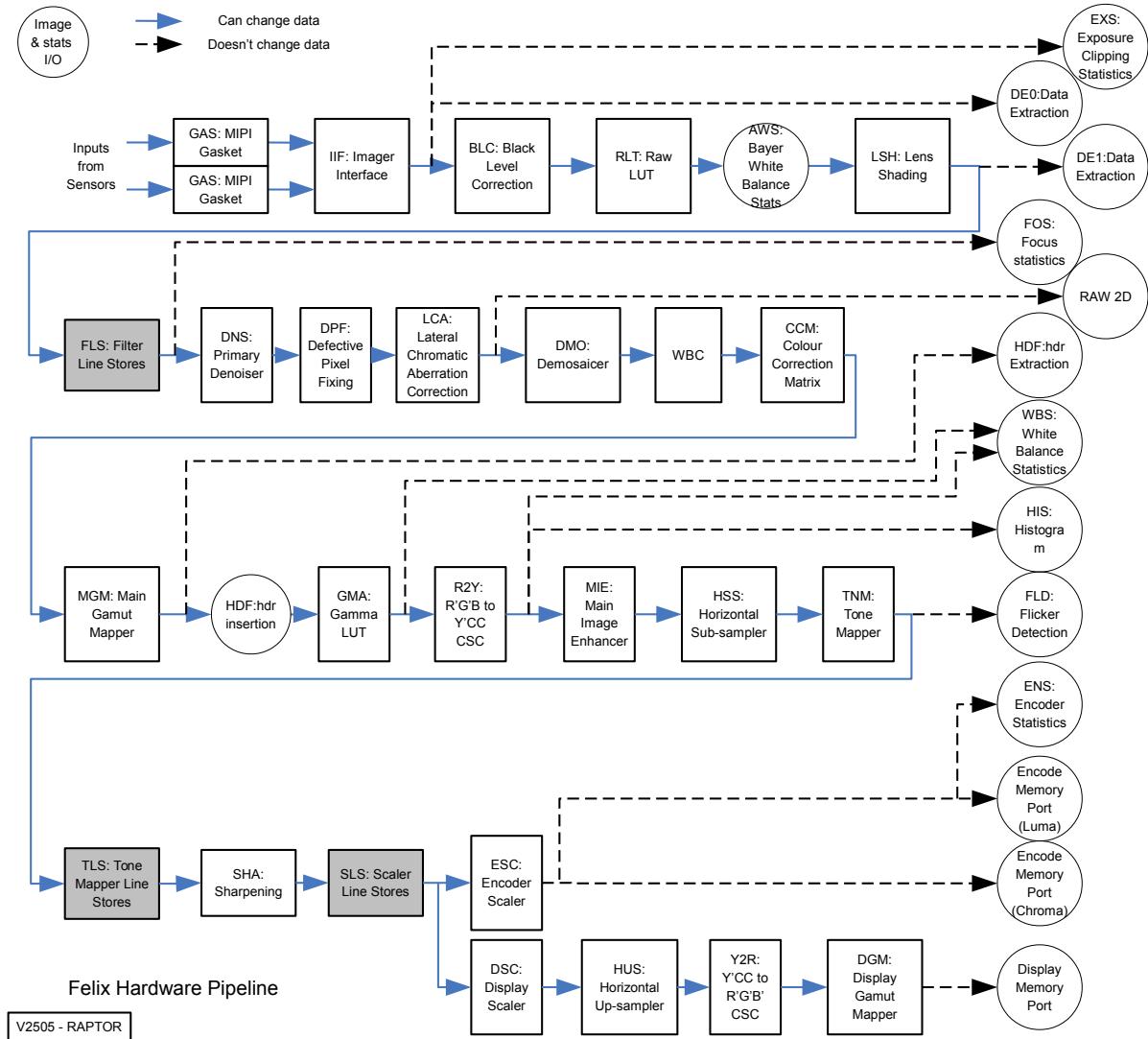


Fig. 2.2: HW Pipeline modules (source: HW Technical Reference Manual).

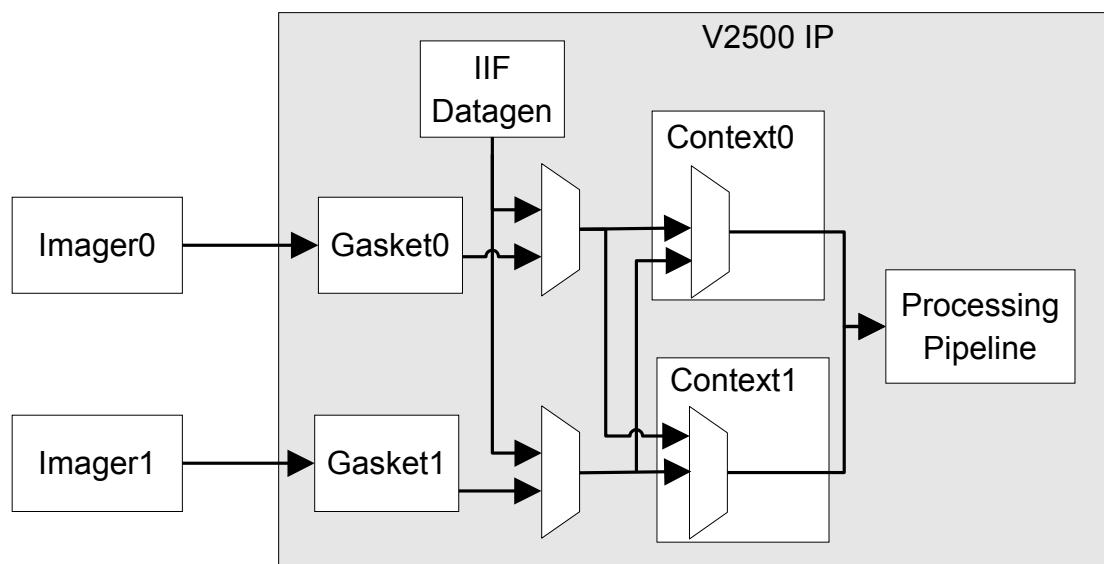


Fig. 2.3: V2500 HW Organisation

2.2.1 The Context

The Context is the main block of the ISP. It is the block that the driver configures the most as it applies image modifications. The input data is always Raw in Bayer format and its output can be of several types (see TRM). The IP can contain several contexts which can process images in parallel from a different gasket or the same gasket.

The HW implement the multiple context using a shared processing pipeline which is shared between all the capturing HW contexts. Each HW context therefore has 2 sets of registers:

- unique set per context (non-swappable)
- swappable set for the processing pipeline

As explained in the TRM the non-swappable registers usually contain information about the frame's sizes.

2.2.2 The Gasket

The Gasket is responsible for transforming the Imager's data into an Imagination format that the context can understand (similar to parallel format). A gasket can either be of MIPI CSI-2 or Parallel format (see TRM for details) and is defined by customer's choices. Each gasket is attached to a single imager but broadcasts the data it receives to all contexts. Each context can choose which gasket they acquire data from.

Internal Data Generator special case

The internal data generator is a block that can replace a gasket's data using frame information from memory. It is therefore useful to either re-process data or at calibration time.

2.2.3 The Imager

The Imager is whatever is providing data to the gasket. A single imager is usually attached to a single gasket. Imagination does not provide imagers (or sensors) therefore it is the customer's responsibility to follow the HW Core Integration Guide document to integrate their Imager.

The External Data Generator special case

When providing the C Simulator package, or using the internal Imagination Technologies FPGA image, an external Data Generator is also available. This block can be considered as an imager - outside of the IP - that can generate either MIPI or Parallel data for a gasket.

2.3 C Simulator (CSIM)

The C Simulator is designed to be a bit accurate simulator of the V2500 HW. It can be provided as an evaluation kit that is used internally for early validation. The driver can be built to communicate with the simulator using the "fake" mode. It will not produce any kernel-side code and have a TCP/IP connection with the simulator. As it is 100% software the CSIM is very slow and does not contain real imagers (but it contains external data generators to replace them).

Information about the “fake” driver is available in *Fake Implementation* (page 165).

2.4 System Requirements

Non-provided software tools required:

- CMake ≥ 3.2
- GNU Compiler Collection $\geq 4.X$ (gcc, g++)
- GNU/Linux kernel sources ≥ 3.4
- Qt ≥ 5.4 if building GUIs. Does not need Qt Multimedia
- OpenCV 2.4.9 if building GUIs - EXACT VERSION NEEDED
- If using IMG SCB block: i2c-driver should be available in the GNU/Linux kernel

Provided external libraries:

- Gtest 1.6.0 - used for unit-testing
- Qwt 6.1.0 - used with Qt when building GUIs
- Mxml 2.7 - used for internal libraries when using the fake build to run against C simulator
- Zlib 1.2.7 - used for internal libraries when using the fake build to run against C simulator

Optional software tools:

- Doxygen - used to generate the code’s documentation (version ≥ 1.8 recommended)

2.5 Deliverables

This section details what is produced when building the DDK. The referred names are available when calling `make help`, and target names in CMake. For each deliverable the installation location is relative to the `CMAKE_INSTALL_PREFIX` given at CMake time. Libraries and executables not described in this section are internal and not considered deliverables.

2.5.1 Main Libraries

The kernel-module is treated as if it is a library by CMake.

Note: The libraries are not installed by the `make install` step unless specified.

CI_User: Capture Interface - user side library Low Level HW driver communicating with kernel-side.

CI_Kernel_KO, CI_Kernel: The kernel-module for the low level driver when using the real driver (output is Felix.ko) Or the library of the low level HW driver when using the fake interface (C Simulator). Installed in km/ when building kernel module.

DG_User: Low level External Data Generator driver - user side library Optional, only if trying to use external data generator.

DG_Kernel: Library of the low level Data Generator driver when using the fake interface.
It is included in CI_Kernel_KO when building the real driver (e.g. for the Imagination FPGA image which contains the external datagen).

ISPC: Image Signal Processor Control (uses CI_User and sensor API) High level driver library.
Control of the image.

sensorapi: Sensor driver API Library use to control the sensor using the IMG SCB block.
Should be modified by the client to supports its own sensors.

FelixCommon: Library that contains information about pixel formats and allocation sizes.
Used by CI_User and kernel module. Should also be used by external applications that intent to import buffers (to compute the sizes of the buffers in the shared allocator).

2.5.2 Command line binaries

This list does not include the unit test executables that can be found using the information in *Unit tests location* (page 25). These executable behaviour and user-guide are available in the *Command line application user guide* (page 47) section.

driver_test: Low level driver application.

A different version is compiled when using the fake interface or the real driver (to fake insmod).

CI_User library is used.

Installed in CI/

Uses either External or Internal Data Generator.

dpf_conv: Utility tool to convert output DPF binary files (HW format) to input DPF binary files (HW format).

Installed in utils/

ISPC_test: High level driver test application to run with data generator for a specified amount of frames.

A different version is compiled when using the fake interface or the real driver (to fake insmod).

ISPC library is used.

Installed in ISPC/

Uses either External or Internal Data Generator.

ISPC_loop: High level driver application running with the sensor continuously. Can be used with data-generator as well.

ISPC library is used.

Installed in ISPC/

ISPC_tcp: High level application similar to ISPC_loop but used to run against VisionLive GUI.

ISPC library is used.

Installed in ISPC/

Note: Information on how to use ISPC_tcp is available in the *GUI tools applications user guide* (page 86) document.

ISPC_capture: Simple application that can be used to capture videos. Accumulates a given number of frames to memory before saving them to disk (designed to avoid memory access latency that may exist on some test systems). Cannot be used with data-generators.

ISPC library is used.

Installed in `ISPC/`

sensor_test: Simple application that allow us to run an implementation of sensor only enabling the gasket and the selected sensor.

CI and `sensorapi` libraries are used.

Installed in `utils/`

2.5.3 GUI binaries

These executables are only built when using the GUI options. Detailed information and user-guide are available in *GUI tools applications user guide* (page 86).

VisionTuning: Offline calibration tool (does not need to connect to device).

Installed in `GUI/VisionTuning/`

VisionLive: Live calibration tool (connects to device using `ISPC_tcp`)

Installed in `GUI/VisionLive/`

Information on how to run the GUI tools is available in the *GUI tools applications user guide* (page 86) section.

2.6 Driver build steps

This section details how to build the drivers on a GNU/Linux machine. In the following steps it is assumed that the DDK sources are present in a folder similar to:

```
~/workspace/ISPDDK_1.2.3/
  \- Android - source directory for Android
  \- DDKSource - source directory of the DDK
  |
  \- cmake - directory used for out of source building, can be deleted
  \- install - directory used to store the install files, can be deleted
```

When building the user is expected to already be in the `cmake` folder. The advantage of CMake is that the actual build directory can be anywhere as long as the path to the sources is known. It is expected that the `Unix Makefile` generator is used. If it is not the case one can compile using the command `cmake --build .` from the build directory.

Any files generated in the `cmake` or `install` directory can be safely removed but may have to be regenerated running the `cmake` or `build` step. CMake generates a list of dependencies between targets therefore the build result described in this section may be different when ran from other computers. Obviously before anything has run both folders are empty.

More options can be given to the build system than the one presented here, refer to the *GUI build steps* (page 21) for more details.

In this whole Section a command line will always be prefixed by a \$ sign, other text in blocks are results from commands or informative.

2.6.1 Choosing a build variant

The driver comes with several build variants, which have to be chosen at CMake time:

- Building the driver to run against the CSIM in user mode (fake interface): see *Fake interface (Simulator/unit tests)* (page 14)
- Building the driver to run against real HW in kernel space and user space. There are several variations of this *Kernel-module build (GNU/Linux)* (page 15) or *Kernel module build for Android* (page 16).

As the build is done using CMake it is possible to have 1 CMake folder per configuration, for example `cmake_fake`, `cmake_linux` that use the same source folder but have different build parameters. In this document we will always refer to them as the cmake folder.

Note: The external Data Generator option is available for all platforms but the External Data Generator's HW is not present on real chip (but it is on Imagination FPGA images and in the C simulator).

Platform choice

The kernel is constructed in a way so that the *platform* it is built for is chosen at compilation time. A platform is a combination of:

- A device implementation responsible for discovering the device and storing information about it
- A memory allocator implementation responsible for allocating the physical memory for the device

The following examples of devices integration are provided in the *Platform Integration Guide* (page 27):

- “PCI” device to discover and use on Imagination FPGA systems
- “Memory Mapped device” as an untested example

and memory allocators:

- “Page allocator” which allocates system memory pages directly
- “Carved-out memory” (where memory is reserved for a device, for example PCI memory on the chip or reserved band of system memory at Linux boot)
- “ION allocator” for Android which is a special case (it implements a special version of carved-out or page allocator) which is turned on when compiling for Android.

For more details on the CMake options please see *the list of available options for the Linux build* (page 15).

Note: IMG Video is an additional platform using the *imgvideo* module used by other teams. The ISP DDK can be compiled following the documentation provided with IMG Video and the minimal steps to run cmake are provided in the section *IMGVideo Support* (page 19).

2.6.2 Generic steps

Regardless of the build options a successful build will always have results similar to the following.

```
$ cmake <path to source> [options]
```

A successful CMake run should finish with an output similar to:

```
-- Configuring done
-- Generating done
-- Build files have been written to: ~/workspace/cmake
```

Build step

```
$ make
```

A successful build should finish with a similar output to:

```
Linking CXX executable driver_test
[100%] Built target driver_test
```

Install step

```
$ make install
```

This should result in files being copied over in `~/workspace/install`. The output should look similar to:

```
Install the project...
-- Install configuration: "Release"
-- Installing: ~/workspace/install/km/Felix.ko
```

2.6.3 Fake interface (Simulator/unit tests)

To run against the simulator we need to enable the Fake interface. This mainly concerns the CI component. The `BUILD_UNIT_TESTS` flag can be added to generate the unit test executables (it is optional).

CMake step

```
$ cmake ../DDKSource -DCI_BUILD_KERNEL_MODULE=OFF \
-DCI_EXT_DATA_GENERATOR=ON -DBUILD_UNIT_TESTS=ON \
-DCMAKE_INSTALL_PREFIX=~/workspace/install -DCMAKE_BUILD_TYPE=Release
```

The *generic steps* (page 14) can now be followed to build the DDK.

Running against the simulator

Even if it is possible to run against the simulator using the Internal Data Generator it would be very limiting (only 1 input possible - testing of parallel computing would not be possible). It is therefore recommended to use the External Data generator support if running against CSIM.

Note: Information on how to run test applications, such as driver_test or ISPC_test, can be found in the *Test Tools* (page 40) document.

Running unit tests

CMake provides an easy way to run all unit tests:

```
$ ctest
```

A successful run produces an output similar to:

```
100% tests passed, 0 tests failed out of 9
Total test time (real) = 2.37 sec
```

2.6.4 Kernel-module build (GNU/Linux)

The kernel build creates a kernel module and mainly concerns the CI component. We will assume that the Linux kernel to use is located in `/usr/src/linux-kernel`. If the build should use the current kernel then the `LINUX_KERNEL_BUILD_DIR` option does not have to be used. The kernel folder used for building should contain the kernel `Makefile` used to build modules (`cmake/CI/felix/felix_lib/km`).

Choosing what platform to use

The CMake options to choose the platform are `-DCI_DEVICE=<device>` `-DCI_ALLOC=<allocator>` The details can be found in:

```
DDKSource/CI/felix/felix_lib/kernel/build_kernel.cmake
```

The device options are:

`-DCI_DEVICE=MEMMAPPED`: Compiles a Memory Mapped device. This is the default if no options are given.

`-DCI_DEVICE=PCI`: Compiles the PCI device for IMG FPGA

`-DCI_DEVICE=ANDROID_EMULATOR`: Test environment for android (use our CSIM as a library in a QEMU virtual machine)

The allocator options are:

`-DCI_ALLOC=CARVEOUT`: Compiles an allocator that assumes that the memory offset given by the device is a reserved band of memory to be used (e.g. memory on a PCI chip). This is the default if no options are given.

-DCI_ALLOC=PAGEALLOC: Compiles an allocator that will allocate system memory pages for the device. This is not supported by the driver yet.

[OBSOLETE] -DCI_ALLOC=ION: Now obsolete, the ION allocator is used if the build is an Android build and the ION implementation will have a different configuration depending of the CI_ALLOC choice.

CMake step

```
$ cmake .. /DDKSource \
-DLINUX_KERNEL_BUILD_DIR=/usr/src/linux-headers \
-DCI_DEVICE=<device> -DCI_ALLOC=<allocator> -DCMAKE_BUILD_TYPE=Release
```

The *the build steps* (page 14) can now be used to build the DDK. Information on how to instantiate the kernel module is available in *run the drivers* (page 16). If the path to the kernel headers is not specified the currently running kernel is used.

2.6.5 Kernel module build for Android

Building the kernel module for Android is described in detail in the *Android Build Instructions* (page 325). The basic steps are to use the right tool-chain for the platform you are building on and to set the Linux kernel root to be the Android Linux kernel root.

2.6.6 Run a kernel-module

This Section details how to run the kernel-module driver. If the driver was compiled with the Fake interface to run against the simulator these operations are not needed.

IMG SCB driver (I2C driver)

If using IMG SCB block, or any standard i2c driver, the i2c-driver should be present in the Linux kernel. If compiled as a module it should be inserted for the i2c device to be detected by the driver. The can be inserted as:

```
$ insmod /lib/modules/`uname -r`/kernel/drivers/i2c/i2c-dev.ko
```

After both the V2500 and i2c-dev modules are inserted a new device should appear. To be able to read/write I2C device without root privileges, below command should be executed:

```
$ chmod 0666 /dev/i2c-detected_id
```

It's possible to give read/write permissions to other users during system startup automatically:

1. Android - edit init.rc file and add below line in “on init” section:

```
$ chmod 0666 /dev/i2c-*
```

2. Linux - add udev rule, creating new file 70-i2c.rules in /etc/udev/rules.d/ with below content

```
KERNEL=="i2c-[1-9]*", GROUP="i2c", MODE="0666"
```

Insert V2500 module

Warning: If using IMG SCB the i2c driver, it should be inserted before the steps stated above!

Before running the user application the Felix driver must be inserted in the kernel using `insmod` (as root):

```
$ insmod install/km/Felix.ko
```

A valid insertion should display no message on the console. But one can also check the system log using `dmesg` to get more information (and have a result similar to):

```
$ dmesg
[16735.653265] FELIX driver initialisation...
[16735.661450] FELIX Main MMU pre-configured (ext address range 1)
[16735.678301] FELIX DG MMU pre-configured (ext address range 1)
```

A successful insertion should result in the module being listed in the list of active modules:

```
$ lsmod | grep Felix
Felix      227325  0
```

Devices should also have been created:

```
$ ls /dev/img*
/dev/imgfelix0
```

The driver will also have some registered elements in the system:

```
$ ls /sys/class/misc/img*
/sys/class/misc/imgfelix0
```

If the driver was compiled with `CI_DEBUG_FUNCTIONS` then some debugFS files are created as well (as root, assuming debugFS is mounted ¹):

```
$ ls /sys/kernel/debug/imgfelix
DriverCTX0Active DriverCTX0Int_DoneAll DriverCTX0Triggered DriverNConnections ...
```

A list of the debugFS files is available in the *Capture Interface* (page 140) document. All parameters given to `insmod` should be accessible in :

```
$ ls /sys/module/Felix/parameters/
frametimeout mmucontrol tilingScheme
```

See the *Kernel module insertion errors* (page 342) for helps on potential insertion errors.

V2500 Insertion options

The driver is compiled with a few `insmod` options:

`tilingScheme=<mode>`: mode is 256 or 512. If your device supports tiling changes the MMU tiling scheme for 256Bx16 or 512Bx8 schemes.

¹ <http://www.linuxforu.com/2010/10/debugging-linux-kernel-with-debugfs/> (accessible on 22/08/2013)

tilingStride=<stride>: If non-0 used as common stride when tiled buffers are used. This has to be a power of 2 and has to be big enough for all enabled output (even non-tiled ones that could be tiled).

frametimeout=<time in ms>: Changes the amount of time the driver will wait when acquiring a captured frame. Once that time has passed the kernel module will return an error and the user-side has to choose what to do (e.g. try again or restart the sensor)

mmucontrol=<mode>: Controls the way the IMG MMU is used. It is not recommended to use this parameter (default is mode=2 use MMU HW in 40b mode).

Warning: If using mmucontrol=0 the MMU HW will be setup to not use virtual memory. The used memory allocator **HAS TO SUPPORT** contiguous memory allocation.

gammaCurve=<curve>: Choose which Gamma Look Up curve to use when running the driver. It is not recommended to use this parameter (it is better to change the default to the desired standard). See the *Capture Interface* (page 148) on details for default value and how to add other curves.

extDGPLL=<mult,div> When using the external data-generator can be used to change the speed of processing. Provide the multiplier and divider to the test IO bank (**not part of the ISP IP**) - both values need to be specified.

If 0 is given (default) then the data-generator driver computes the default PLL $\frac{\text{parallesim} \times 10}{\text{parallesim} \times 10}$

Warning: If incorrect values are provided the PLL may not lock up and the FPGA will need to be reprogrammed.

If given values are too high the test may not generate frames because the input rate will be too high for the ISP to handle.

ciLogLevel=<level>: Define the runtime value for the log level of the kernel module. Values are from 0 to 4, each level includes the lower values: 0 none, 1 `CI_FATAL()`, 2 `CI_WARNING()`, 3 `CI_INFO()`, 4 `CI_DEBUG()` (very verbose). Default is computed by cmake and given at build time as pre-processor value `CI_LOG_LEVEL` and is 1 for Release and 3 for Debug.

Details are available in:

```
DDKSource/CI/felix/felix_lib/kernel/kernel_src/ci_init_km.c
```

Example, change timeout to be 2 seconds:

```
$ insmod Felix.ko frametimeout=2000
```

Removing module

Simply run (as root):

```
$ rmmod Felix
```

If an application is currently running using the Felix kernel module this operation may fail. Stop or kill the application to be able to remove the module. A valid removal should produce no message on the console but dmseg should show something similar to:

```
$ dmseg
[17008.502019] FELIX shutting down HW
```

2.7 IMGVideo Support

IMG Video is the common kernel libraries used by various Imagination IP. The ISP DDK being a more independent IP the usage of the common module is optional. It is however possible for a customer to compile the device memory abstraction implementation if they possess a compatible IMGVideo source code.

Warning: Continuing on this section it is assumed that the customer has:

- ISP DDK package
- a compatible IMGVideo package (not provided by ISP DDK)
- documentation for the IMGVideo package
- read IMGVideo documentation to know how to port to their memory interface

Note: IMGVideo is delivered with every release of the Encoder or Decoder. Compatibility with the ISP is not ensured unless delivered as the same package. Customers that are integrating both ISP and Encoder/Decoder should ensure a compatible IMGVideo package is delivered by asking their contact at Imagination.

The IMGVideo supports allows a central memory management to be used but the platform discovery and handling is done following the usual way for the ISP (as described in the *Platform Integration Guide* (page 27)).

The ISP DDK integrates with IMGVideo cmake structure:

```
video
|
\--imgvideo
|
\--vdec (optional)
\--encode (optional)
|
\--isp (ISP-DDK folder)
|
\--DDKSource
```

The cmake can be run the same way than any other IMGVideo cmake by adding the relevant CI options (see *Standard CMake options* (page 22)).

```
$ mkdir build
$ cd build
$ cmake ../video -DPLATFORM=<platform> <other imgvideo options> \
-DCI_DEVICE=MEMMAPPED -DCI_ALLOC=IMGVIDEO <other CI options if needed> \
<other options - e.g. -DENABLE_3_9_CONFIG_ALL=TRUE for encoder> \
-DCMAKE_INSTALL_PREFIX=../install
$ make
```

The other possible IMGVideo options can be:

- If using carveout memory:

```
-DPORTFWRK_SYSMEM_CARVEOUT=TRUE
```

- If using page allocation:

```
-DPORTFWRK_SYSMEM_UNIFIED=TRUE -DPORTFWRK_MEMALLOC_UNIFIED_VMALLOC=TRUE
```

- If using ION allocation (untested as the ISP has its own ion support):

```
-DION_BUFFERS=TRUE
```

- Support for allocation and import of dma-buf : carveout

Enables build of `dmabuf_exporter.ko` kernel module and adds a imgvideo heap suitable for importing externally allocated dma-buf buffers.

```
-DDMABUF_EXPORTER=TRUE  
-DPORTFWRK_SYSMEM_DMABUF=TRUE  
-DSYSSMEM_DMABUF_IMPORT=TRUE
```

Note: The `dmabuf_exporter.ko` module supports following load time parameters:

- **carveout_phys_base** - base physical address. if 0 (default), then use pci vendor/product) (ulong)
- **carveout_size** - carveout size in bytes. if 0 (default) and phys_base is also 0, then use all PCI memory (ulong)
- **carveout_offset** - offset from carveout_phys_base. default: 0 (ulong)
- **carveout_pci_vendor** - carveout PCI vendor id (uint)
- **carveout_pci_product** - carveout PCI product id (uint)
- **carveout_pci_bar** - carveout PCI BAR (uint)
- **use_carveout** - use carveout memory (default : 0, to use coherent) (int)

By using these parameters, the user can limit the available memory for dmabuf allocations.

- Support for allocation and import of dma-buf : coherent

Same as former config but `dmabuf_exporter.ko` should be insmodded with `use_carveout=0` parameter in order of using coherent allocator.

```
-DDMABUF_EXPORTER=TRUE  
-DPORTFWRK_SYSMEM_DMABUF=TRUE  
-DSYSSMEM_DMABUF_IMPORT=TRUE
```

Note: Felix source tree contains an implementation of `dmabuf` userspace library required for user side image buffer allocations using `dmabuf_exporter.ko`. For testing purposes, `ISPC_loop` can be configured to use this library for allocations of external image buffers. It is then possible to import the buffers to ISP MMU using `CI_PipelineImportBuffer()` (for more info refer *Creation of Buffers* (page 183) in *Capture Interface* (page 140)).

To facilitate this functionality, the test application CMake should check if `SYSSMEM_DMABUF_IMPORT` is defined and will allow the use of the run time `-importBuffers` parameter to be used if it is present.

More on building and execution of `ISPC_loop` see *ISPC loop application: ISPC_loop* (page 69).

2.7.1 Example: building IMGVideo for Imagination's FPGA

For this example we chose to present building CI with the external data generator (which is not present on the IP or outside IMG) and the DebugFS support merely as example of CI options.

Using carveout memory (typical FPGA usage):

```
$ cmake .. /video -DPLATFORM=img_fpga -DPORTFWRK_SYSMEM_CARVEOUT=TRUE \
-DCI_DEVICE=PCI -DCI_ALLOC=IMGVIDEO \
-DCI_EXT_DATA_GENERATOR=ON -DCI_DEBUG_FUNCTIONS=ON \
-DCMAKE_INSTALL_PREFIX=../install
```

Building to use page-alloc (direct access of memory through PCI also called bus-mastering):

```
$ cmake .. /video -DPLATFORM=img_fpga -DPORTFWRK_SYSMEM_UNIFIED=TRUE \
-DPORTFWRK_MEMALLOC_UNIFIED_VMALLOC=TRUE -DFPGA_BUS_MASTERING=ON \
-DCI_DEVICE=PCI -DCI_ALLOC=IMGVIDEO \
-DCI_EXT_DATA_GENERATOR=ON -DCI_DEBUG_FUNCTIONS=ON \
-DCMAKE_INSTALL_PREFIX=../install
```

Using Android ion allocator, FPGA on PCI, built against Android kernel located in \$ANDROID_BUILD_TOP/kernel/common:

```
$ cmake .. /video -DPLATFORM=img_fpga -DANDROID=ON -DION_BUFFERS=ON \
-DPORTFWRK_MEMALLOC_UNIFIED_VMALLOC=TRUE -DFPGA_BUS_MASTERING=ON \
-DCI_DEVICE=PCI -DCI_ALLOC=IMGVIDEO \
-DLINUX_KERNEL_BUILD_DIR=$ANDROID_BUILD_TOP/kernel/common \
-DCMAKE_INSTALL_PREFIX=../install
$ make modules
```

2.7.2 IMGVideo recovering results

When building with IMGVideo support some of the source is compiled into the source tree. The kernel modules are available in:

- video/imgvideo/imgvideo.ko
- video/isp_km/Felix.ko

Or if CMAKE_INSTALL_PREFIX was given and `make install` run the ISP DDK is available normally. IMGVideo should install its kernel modules under the top level `km` folder.

2.8 HDR Libraries support

The package delivered does not contain the HDRLibs archive and will not build the HDR libraries support. These libraries are internal to Imagination and the applications making use of them detail how to replace them if necessary.

2.9 GUI build steps

The tuning tools GUIs are delivered as part of the DDK sources but can be built separately (e.g. build driver for an android phone but build tuning tools for GNU/Linux system). There

are 2 tools used for tuning:

- **VisionTuning** used for offline calibrations (does not need an actual device to run). For example this is used to generate the deshading matrix.
- **VisionLive** used for dynamic calibrations (connects to the device and see the result of calibration live). For example this is used to calibration the tone mapping curve.

2.9.1 Build steps

As for the driver part, CMake needs to be run. It does not have to be in a separated folder.

```
$ cmake ../DDKSource -DBUILD_GUI=ON
```

See *Building GUI tools: troubleshooting* (page 340) and *Running the GUI: troubleshooting* (page 341) for the help on errors.

Note: The DDK does not support windows in any other mode than the Fake driver to communicate with CSIM. The GUI tools can however be built on windows but **ONLY** support Visual Studio.

It is usually needed to specify both *QT_BINARY_DIR* and *OpenCV_DIR* cmake variable to compile the GUI on windows (see *Standard CMake options* (page 22)).

Warning: It is imperative to use the same compiler for the GUI tools than was used to compile the Qt library.

The GUI tools only support Visual Studio, ensure the Qt library are installed for the appropriate VS version.

The run steps are detailed in the *GUI tools applications user guide* (page 86) document.

2.10 Build System details

This section aims to list all of the CMake options that an advanced user may consider when customising the build of the V2500 driver.

The V2500 DDK is built using CMake. CMake generates Makefiles (or project files for editors such as Eclipse or Code::Blocks) that are then used to build the various libraries and executables.

2.10.1 Standard CMake options

One might consider changing the following CMake options (that are standard to any CMake script):

CMAKE_INSTALL_PREFIX (Path): Location where package is installed if the install target is run (e.g. when doing make install).

QT_BINARY_DIR (Path): Path to the installation of the Qt library. May be needed when building GUIs and the Qt library cannot be found on the system. We also added **QT_LIBRARY_CMAKE_DIR** (which is not standard) to help find the cmake configuration file of Qt5. If the following Path are not defined **QT_LIBRARY_CMAKE_DIR** will be used to guess them, otherwise they can be manually defined one at a time:

- Qt5Widgets_DIR
- Qt5Test_DIR
- Qt5PrintSupport_DIR
- Qt5Concurrent_DIR
- Qt5Help_DIR

Some more may be needed depending on compilation options.

OpenCV_DIR (Path): Path to the installation of the OpenCV library. May be needed when building GUIs and the OpenCV library cannot be found on the system.

CMAKE_BUILD_TYPE (String): To control the build of Debug or Release. Useful when building against libraries that have different behaviour in release/debug. E.g. GUIs against Qt and OpenCV

More variables can be found in CMake documentation ².

2.10.2 DDK options

These options are global to the Felix DDK libraries. They define the behaviour of the build system and often what flavour of the driver to build.

Components choice

Choosing which components are built, from low level to higher level. By default all components except the GUI are built. This information is stored in the top-level `CMakeLists.txt`.

BUILD_CAPTUREINTERFACE [ON]: Build the Capture Interface (CI) - Felix low level HW driver

BUILD_ISPCONTROL [ON]: Build the ISP Control (ISPC) - Imager parameter control. Depends on the Capture Interface

BUILD_SENSORAPI [ON]: Build the Sensor API - control of the camera sensors Depends on the Capture Interface

BUILD_GUI [OFF]: Build the provided GUIs for parameter tuning and demonstration

BUILD_SCB_DRIVER [ON]: Build the IMG SCB driver as part of the CI (kernel side). May not be available on all HW versions.

BUILD_UNIT_TESTS [OFF]: Will build all libraries unit-tests. Note that some unit tests may be disabled if building the kernel module.

BUILD_TEST_APPS [ON]: If disabled the test applications are not built.

ANDROID_BUILD [OFF]: Choose to build kernel-side for android.

Global options

ENABLE_GPROF [OFF]: Enable the GProf support when using GCC. May need special toolchain adjustments to locate library at linking time.

² http://www.cmake.org/cmake/help/v2.8.8/cmake.html#section_VariablenThatChangeBehavior (accessible on 14/07/2014)

QT_LIBRARY_CMAKE_DIR [QT_BINARY_DIR]/..../lib/cmake/] Location of the library folder in Qt5. The folder should contain the several modules that can be searched for when using Qt. Used as a root (see other Qt variables in standard CMake variable section).

DEBUG_MODULES [OFF]: Display CMake debug information

CI options

Configure some of the low level driver options. This information is taken from CI/felix/project.cmake.

CI_BUILD_KERNEL_MODULE [ON]: Chose to build the kernel module or the fake interface (to use with the simulator).

On windows this is forced to OFF.

LINUX_KERNEL_BUILD_DIR: Path to the Linux kernel to use. If none is provided uses the current kernel.

LINUX_KERNEL_SOURCE_DIR: Path to the Linux kernel source directory. Used by Android builds when some headers are not available in the build directory otherwise do not use. (see *Configure V2500 Module using CMake* (page 326)).

CI_DEVICE: *Choose device* (page 15) to build for real driver.

CI_ALLOC: *Choose allocator* (page 15) to build for real driver.

FORCE_32BIT_BUILD [OFF]: Force 32b build even if system is 64b. The system needs the 32b libraries installed.

CI_EXT_DATA_GENERATOR [ON]: External Data Generator low-level driver

CI_DEBUG_FUNCTIONS [OFF]: Build the CI debug capabilities. Using fake device it includes debug functions. Using kernel module it produces debugfs files and read/write to registers from user-space CI.

Warning: Because this option allows user-space to read/write any registers it is recommended to not compile production drivers with the debug functions enabled or to manually comment the kernel-space function that allows access to registers. See the *Driver commands* (page 355) section for details about the register access through IOCTL.

CI_HW_REF_CLK (Int): Reference HW clock in MHz - for the moment only stored in CI_HWINFO Defined in CI/felix/felix_lib/kernel/CMakeLists.txt.

CI_MEMSET_ALLOCATED_MEMORY [OFF]: Enable the memset of device memory to 0x0 (may slow down allocation).

CI_MEMSET_ADDITIONAL_MEMORY [ON]: Enables the memset of additional device memory to 0xAA (allocations to nearest page size).

Turning off may reduce the out2.prm size when generating pdumps with Fake driver and speed up allocation when using real driver.

CROSS_COMPILE: Can be defined to specify a cross compiler root that will be used by both the top level CMakeList to define the C and C++ compiler and by the used in build_kernel.cmake

to generate the kernel module makefile (using the CROSS_COMPILE variable used by the kernel Makefile).

The top level CMakeLists can be modified to specify additional C and C++ flags too but GenKernel will not necessarily take them in account.

E.g. use /home/felix/uclibc/bin/arm-buildroot-linux-uclibcgnueabihf- to use arm-buildroot-linux-uclibcgnueabihf-gcc and arm-buildroot-linux-uclibcgnueabihf-g++ as compilers.

ISPC options

ISPC_PERFLOG: [OFF] Enable the usage of the performance measurement code in ISPC. See *Performance measurement* (page 285) for details.

2.10.3 Unit tests location

Several of the libraries in the DDK include unit tests. In the unlikely event of a test failing one can run the test independently in order to produce more detailed results (or read the xml file generated). This section describes where to locate the executable or output file using the same assumptions than used in *CMake step* (page 14). Refer to the same section to know how to build the unit tests. All of the filenames generated when running CTest should be formatted like xunit_<testname>.xml. Most of the provided unit tests are using Google Test framework. The Google Test documentation ³ may be an interesting read when considering running a sub-set of tests.

CaptureInterface: CI/felix/felix_lib/test

DynamicCommandLine: common/dyncmd/test

FelixCommon: common/felixcommon/tests

GZipFileIO: CI/imglib/libraries/gzip_fileio/test

IMGIncludes: common/img_includes/test

ISPControl: ISP_Control/ISPC_lib/test

LinkedList: common/linkedlist/test

MMULib: CI/imgmmu/test

sensorapi: sensorapi/tests

SimImage: common/sim_image/test

TAL_Test: CI/target/test

VisionTuning: GUI/tuning/test

2.11 Doxygen documentation

The source code is commented in such a way to allow the generation of structured documentation of the code. This includes information on how to create and include a new camera sensor driver.

³ <http://code.google.com/p/googletest/wiki/Documentation> (accessible on 14/07/2014)

To generate the documentation make sure you have Doxygen installed on your system. If you have to install it then you must re-run CMake as described in the steps above. Then when you do your make step you can also do

```
make doc
```

This will generate html documentation under `doc/V2500` of your build folder. Open the file `index.html` to start using the documentation.

What if the doc target does not exists:

It is possible that your doxygen installation is not picked up by CMake when generating the target. The doxyfile will be generated regardless of the presence of Doxygen on the system as `doc/V2500_API.Doxyfile` in your build folder.

What if different outputs are needed (e.g. LaTeX):

The documentation is optimised to be generated as HTML but is possible to modify the desired outputs. The doxyfile template can be found under `DDKSource/build/DoxyfileHTMLTemplate.cmakein`. This file is used as in input to the `configure_file()`⁴ CMake function.

⁴ http://www.cmake.org/cmake/help/v2.8.8/cmake.html#command:configure_file (accessible on 14/07/2014)

Chapter 3

Platform Integration Guide

This document assumes that the driver has already been successfully built on an x86 system following the [Getting Started Guide](#) (page 6). The reference drivers run on a PCI based FPGA platform, as such the steps required to register a device may differ to a system which has the device mapped to a system virtual address.

This guide is intended to describe how and where to make the necessary changes to allow the driver to run on another platform. The porting falls into two main categories, device access and device memory management.

This guide does NOT covers the sensor porting which is an expected state for every customers described in [Sensor driver](#) (page 34).

Understanding the low level driver (see the [Capture Interface](#) (page 140) section) will help for the integration. All source code mentioned here is present in the kernel-side driver:

```
DDKSource/CI/felix/felix_lib/kernel/
```

3.1 Kernel Module changes

The following sections contain possible changes that customers should consider to the provided DDK.

3.1.1 Kernel Module initialisation

The kernel module initialisation is implemented in:

```
DDKSource/CI/felix/felix_lib/kernel/kernel_src/ci_init_km.c
```

This file contains the insmod options (see [V2500 Insertion options](#) (page 17)) and the DebugFS initialisation (if compiled).

The kernel module registers a *misc device* to the OS but the code contains *char device* entries as example (legacy implementation).

The device name can be changed there (see macro *DEV_NAME*). The device is registered using the Device access abstraction *SYS_DevRegister()* and on a successful discover *probeSuccess()* will be called to create the CI driver. It is recommended to modify the device class to handle clocks speed and other device specific operations rather than the initialise function.

3.1.2 Change to ioctl

When using IOCTL on a Linux environment each driver has to define a “magic” character that will be used for the base of the IOCTL number to avoid confusion. This character is defined as CI_MAGIC in :src‘ci_ioctl.h‘ and can be changed by the customer if need be.

A list of implemented ioctl is available in the section: *CI Appendix: IOCTL tables* (page 355).

3.2 Device Access

The device access is abstracted in:

```
DDKSource/CI/felix/felix_lib/kernel/include/ci_internal/sys_device.h
```

It contains the definition of the SYS_DEVICE structure which contains:

- The associated interrupt handler for the device
- The file operation structure for the IO (i.e. the ioctl, mmap and other system call hooks)
- The suspend and resume call-backs for when the system becomes idle
- The board registers the CPU address when using a PCI card (which may not be useful for other platforms)
- The V2500 chip register base CPU addresses to be able to access them
- The V2500 chip memory base CPU address to be able to access them

It should also define a few operations which are generic for all devices:

- Register and de-register a device (e.g. for PCI registers the driver)
- Remove a device (operations to do when the kernel module is removed)
- Clear interrupts: if additional actions need to be performed after the interrupts are handled for V2500 (e.g. clear a common line on PCI)
- Request and free interrupt: register an interrupt handle for that device
- Power control management which allows implementation of turning system clocks on/off

Note: The choice of which device is used by the driver is done at compilation time therefore some options may have to be added to your cmake build to include the correct files. Currently the choice is made in:

```
DDKSource/CI/felix/felix_lib/kernel/build_kernel.cmake
```

Note: The device access and memory management are closely coupled because the device access dictates the way memory is available to the system. For device type selection CI_DEVICE cmake variable is used (*Choosing what platform to use* (page 15)). Following devices are available:

- PCI: PCI based device
- MEMMAPPED: device which registers are available in CPUs address space.
- ANDROID_EMULATOR: build for Android emulator which connects with V2500 C simulator via transif interface (library loaded at run-time).

3.2.1 Device clock speed

The CI_HWINFO structure contains a field with the clock speed of the device that should be modified: CI_HWINFO::ui32RefClockMhz. The default value is written in HW_CI_DriverLoadInfo() using the macro *CI_HW_REF_CLK* defined from the CMake variable of the same name. The customer can also add a field into SYS_DEVICE to contain the clock and the value could also be retrieved from the attached device (*pDriver->pDevice*).

Warning: This mechanism assumes that the ISP does not have a variable clock speed once initialised. If it is the case the customer may want to remove that entry from the CI_HWINFO and add an additional IOCTL to query the clock speed.

3.2.2 PCI Based Driver

The current FPGA system Imagination uses for testing is PCI based. If the target system is also PCI based it should be easy to modify the existing file to point to correct PCI device. The implementation is available in:

`/DDKSource/CI/felix/felix_lib/kernel/platform_src/sys_device_pci_kernel.c`

The top of the file defines the vendor and device ID that will be searched by the probe function; these will need adjustment for each specific system (including register definitions).

When the device is registered the PCI driver is added to the Linux kernel with a probe function. The current implementation will not cope with several PCI driver (e.g. one per device). When the device is detected the SYS_DEVICE information is filled in using the information from the `pci_dev` structure from the kernel. Registers and memory are memory mapped to the system, which may prove to be expensive for memory on real systems (using carved out memory of 256 MB out of the `vmalloc` pool of the Linux kernel).

The parts about PDP and additional interrupts cleaning may be ignored when porting to another PCI device as they are specific to Imagination's FPGA board.

3.2.3 Memory-mapped Driver

A memory mapped example is provided in:

`/DDKSource/CI/felix/felix_lib/kernel/internal_src/sys_device_memmapped.c`

Warning: Imagination does not possess any platform that can be used as memory mapped therefore this device implementation is provided *as is*.

To reflect platform specific parameters, `ci_device_resources[]` array should be modified with following values:

- `reg_base_addr`: register base address
- `mem_base_addr`: memory base address in case device is using built-in carved-out memory
- `irq`: interrupt number

3.2.4 Device Tree based driver

The device tree is a popular Linux driver model. In this instance the OS has information about the location of a device which the device can query for based upon its name. In this instance the device tree implementation only needs to know what name the device has in the *.dts* device tree map, and from this it can locate itself and then map in the device as required. An example implementation may be provided in the future and will most likely be an extension of the memory mapped driver.

3.2.5 I2C interface driver

In cases where the customer is using the IMG SCB block for I2C communication with camera sensor, I2C driver is provided by Imagination. There are two versions of SCB drivers:

- PCI based
- Memory mapped based

Note: There are no separate device ids for devices programmed into the Imagination FPGA, which means that the ISP and SCB block are visible on the PCI bus as one device. In such a case there is one kernel module which is using PCI registering functions to control both ISP and SCB block at once.

Memory mapped driver is a regular Linux kernel driver which is loaded separately from the ISP. Parameters for the SCB driver can be passed either by device tree or platform specific board file. More details about I2C device interaction in user space can be found in *PHY control (sensor_phys.c)* (page 35).

3.3 Memory management

As for the device access memory management is described in a unique header:

`DDKSource/CI/felix/felix_lib/kernel/include/ci_internal/sys_mem.h`

The generic functions (implemented in *sys_mem.c* regardless of the platform) are designed to cope with the V2500 device MMU and are:

- Allocating and freeing device memory (high level function), which will use the platform functions and reserve device virtual addresses if the MMU is used
- Get first device address (device virtual address when using MMU, physical address when not using MMU)
- Mapping/Unmapping to the device MMU
- Importing and releasing memory, which will use platform functions to do so

The platform specific functions start with prefix `Platform_` and are:

- Allocating non-contiguous, allocating contiguous and freeing device memory
- Importing and releasing memory for platform that support such mechanisms
- Access CPU mapped memory

- Update host and device memory for NUMA systems (e.g. to flush the caches)
- Read and write words or addresses to a location in memory
- Get a list of physical pages to map into the device MMU
- Allocating/freeing a page in the device physical memory for the MMU and also read and write access to such pages
- Map the memory to user-space

Note: Allocating contiguous memory is optional and only used when the device MMU is disabled. It is not recommended to disable the device MMU.

Note: The current implementation does not expose the physical address used to the user-space as it would only be possible when allocating contiguous memory. This could be the case on carved-out memory systems.

In an SOC the device needs to have a common memory allocator which the device needs to be able to communicate with in order to allocate/import memory (e.g. *gralloc* in Android).

For memory allocation types `CI_ALLOC` CMake option is used (see *Choosing what platform to use* (page 15)). Following memory allocation types are available:

- **CARVEOUT:** memory is allocated from special carved-out memory pool. This memory pool can be either reserved during Linux kernel start using `mem=` kernel parameter (Linux system won't use this memory and will treat it as reserved area) or it can be physical memory which is solely available for given device.
- **PAGEALLOC:** memory is allocated from system memory. Virtual addresses are continuous but physical addresses are not.

When Android build is enabled (CMake `ANDROID_BUILD=ON`), ION allocator is used. `CI_ALLOC` define still needs to be defined as it instructs ION how memory should be allocated.

3.3.1 Carved out memory

As explained in *PCI Based Driver* (page 29) our PCI implementation maps the memory of the PCI board as carved out memory (256 MB of it) which may prove expensive on real systems (it is reserving some CPU virtual memory on the vmalloc pool). The implementation is done using that knowledge and may need modification if another approach for memory allocation is used (for example allocating pages manually). The source code is available in:

```
DDKSource/CI/felix/felix_lib/kernel/platform_src/sys_mem_carveout.c
```

3.3.2 Page allocated memory

The page allocator assumes that the ISP can access all memory available to the CPU and will allocate memory block 1 CPU page at a time. The implementation is available in:

```
DDKSource/CI/felix/felix_lib/kernel/platform_src/sys_mem_pagealloc.c
```

3.3.3 ION implementation

The ION implementation for the memory is available in:

`DDKSource/CI/felix/felix_lib/kernel/platform_src/sys_mem_ion.c`

It is designed to handle memory management when using Android. Currently it supports two types of allocations: carved-out (reserved memory) and page-allocated (system-wide memory).

Note: The current ION implementation assumes that the ION allocator module creates the heaps.

3.3.4 Device view of the memory VS CPU view of the memory

In a SoC, or a bus based system such as PCI, it is possible that the ISP device can only access a small band of the system memory. For a SoC it is down to the customer to know what this actual band by setting the correct values while porting `SYS_DEVICE` as explained in [Memory-mapped Driver](#) (page 29) (`mem_base_addr` and `SYS_DEVICE::uiDevMemoryPhysical`). For a bus based access the kernel should give us a physical address that works for the CPU to access it while the device will most likely assume its own physical address starts at `0x0` (forcing `SYS_DEVICE::uiDevMemoryPhysical` to `0`).

For such a scenario it is impossible for the device to use the provided `sys_mem_pagealloc.c` and it is recommended that the customer studies and modifies `sys_mem_carveout.c` if needed (if using an external allocator such as ION the allocator `gralloc` most likely needs to know that information as well).

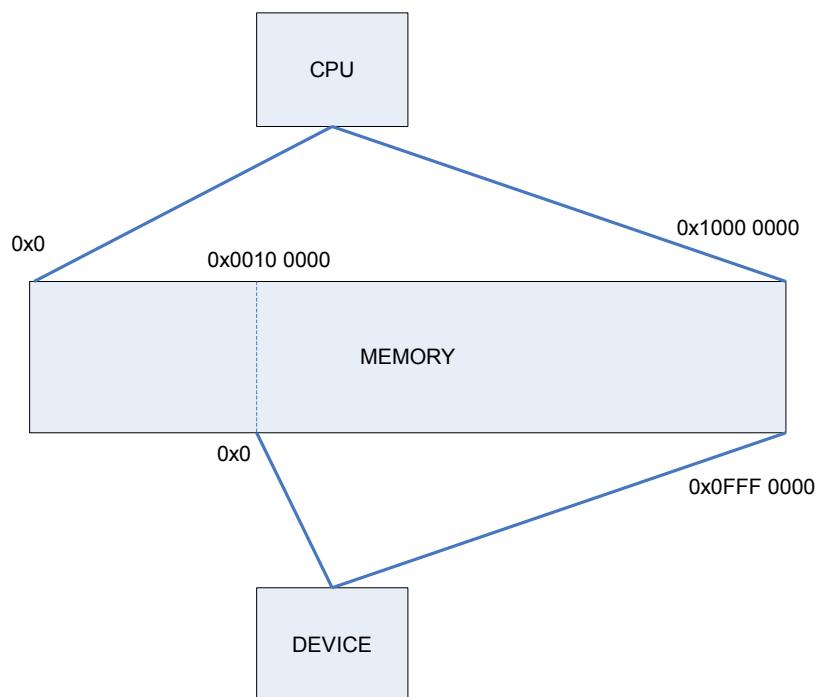


Fig. 3.1: Memory view from CPU and Device example

For example the CPU may see the device memory from `0x0000 0000` to `0x1000 0000`. But the device has an offset in its memory view, address `0x0000 0000` from the device perspective is

0x0010 0000 for the CPU.

1. Memory mapped device: CPU related addresses are put into the `struct resource` given to the kernel, and device accessible are put in the `SYS_DEVICE` object given to the CI driver.

```
static struct resource ci_device_resources[] = {
...
{
    .name = "mem_base_addr",
    .start = 0x00100000,
    .end = 0x10000000,
    .flags = IORESOURCE_MEM,
},
...
};

IMG_RESULT SYS_DevRegister(SYS_DEVICE *sysDevice)
{
...
/* Memory space setup */
sysDevice->uiDevMemoryPhysical = 0; // instead of res->start
sysDevice->uiMemorySize = res->end - res->start;
sysDevice->uiMemoryPhysical = res->start;
sysDevice->uiMemoryCPUVirtual = (IMG_UINTPTR)ci->mem_base;
...
}
```

2. Bus based device: CPU related addresses are not hard coded. Calls to kernel such like `pci_resource_start()` give the original physical address. `SYS_DEVICE` contains the device accessible addresses.

```
sysDevice->uiMemorySize = pci_resource_len(dev, PCI_ATLAS_MEMORY_BAR);

sysDevice->uiMemoryPhysical = pci_resource_start(dev, PCI_ATLAS_MEMORY_BAR);
sysDevice->uiMemoryCPUVirtual =
(IMG_UINTPTR)ioremap(
    pci_resource_start(dev, PCI_ATLAS_MEMORY_BAR),
    g_PCIDriver.pSysDevice->uiMemorySize);
sysDevice->uiDevMemoryPhysical = 0; // instead of g_PCIDriver.pSysDevice->uiMemoryPhysical
```

Chapter 4

Sensor driver

Warning: It is the customer responsibility to modify the PHY control to cope with their PHY. Customers are also expected to modify the sensor API to support their own sensor.

This section will explain how to handle sensor porting using the Imagination PHY and sensor drivers as examples. The Figure *Imagination FPGA PHY/Gasket links* (page 34) shows how the Imagination FPGA uses the IMG PHY to select the source of data for the V2500 IP. Any block of HW not provided by IMG that has to be configured for a sensor to be able to transfer data to the V2500 Gasket should be considered as a PHY. If none is present then the PHY control can be removed.

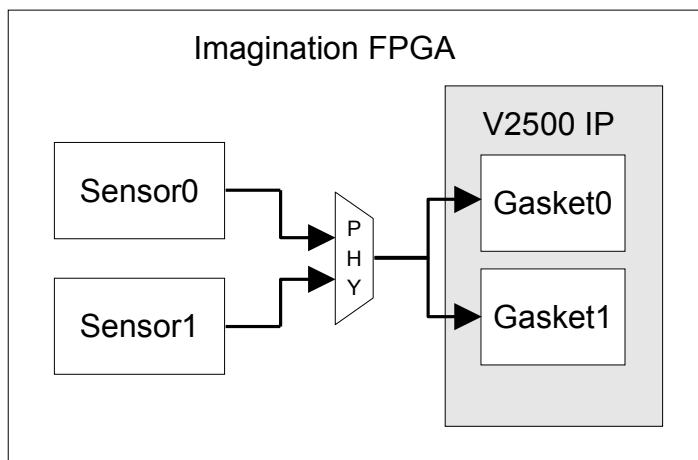


Fig. 4.1: Imagination FPGA PHY/Gasket links

4.1 Platform Integration

The V2500 drivers will require customisation to work within a SoC and to work with different camera sensors. The example sensor drivers demonstrate the interface a sensor driver needs to implement to allow the other parts of the driver stack to interact with a camera sensor (see *Example sensor drivers* (page 38)). The PHY and SCB connection are managed by the `sensor_phy.h` functions (implemented in `sensor_phy.c`). The Sensor drivers are implemented in the various sensor files of the `sensors` folder.

4.1.1 PHY control (`sensor_phy.c`)

Note: This implementation is specific to IMG PHY, an internal block of HW available only on IMG FPGA test systems.

This also assumes that the IMG I2C bus is used. This is not available on all HW versions.

The sensor driver is written as a user mode driver which does its own mapping of the registers it requires for accessing the IMG PHY. User-mode was chosen because the PHY may be completely different on customer systems or even not present at all on a SoC.

The sensor driver also requires the V2500 driver to be installed to allow the sensor driver access to the ISP gaskets. The communication to the PHY can be changed to an ioctl based or any other mechanism that allows the customer to configure their PHY. The customer could even choose to integrate their PHY setup in the kernel-side gasket setup.

The SCB connection assumes that either the Imagination I2C device is used or that a similar one implementing the standard i2c-device for GNU/Linux is present. It is also likely that `sensor_phy.h` should be modified to update the `SENSOR_PHY` structure with the correct fields.

To port the sensor drivers to a new platform a customer is expected to:

- Modify `SensorPhyInit()` to interact with the PHY as required. This means replacing the `UserGetMapping()` to use whatever means the customer choose to interact with their own PHY. The part initialising the Gasket object is required as reservation of Gaskets is done through the CI driver.
- Any operation done in `SensorPhyInit()` should be reversed in `SensorPhyDeinit()`.
- `SensorPhyCtrl()` should be modified to replace the register access to the PHY with the correct operation to enable a particular sensor to communicate with the V2500 Gasket correctly. For instance the IMG PHY has to setup the number of MIPI lanes when using a MIPI sensor.

Warning: The order of operations Gasket **then** PHY should be maintained.

It is expected that the sensor starts streaming **after** both PHY and Gasket are configured. And obviously that the sensor stops streaming **before** PHY and gasket are disabled.

- Modify `find_i2c_dev()` to search for the correct I2C device if not using the Imagination SCB block.

This function does not have to be called by all sensors implementation. Typically on a known platform it is expected to know which I2C device to use and not have to search for it by name.

4.1.2 I2C interface

Some of the camera sensors are controlled using I2C interface. Every platform usually has different I2C controller, so the best solution is to use some generic I2C device to interact with I2C bus.

Such I2C generalization under Linux/Android systems is brought by `i2c-dev` module. It detects all registered I2C adapters and creates user space devices to interact with them. In case that a platform has more than one I2C controller, there is a need to select the one which is actually

connected to the camera sensor. As explained before the detection which I2C controller is done in `sensor_phy.c` file, which opens all the I2C devices using: `/sys/class/i2c-dev/i2c-id/name` and checks name of the controller. In delivered sensor code, it looks for *ImgTec SCB I2C* name which identifies I2C controller from IMG. In case customer is using different I2C controller this code should be changed or even not called at all by the sensor's implementation.

To load `i2c-dev` module refer to the *IMG SCB driver (I2C driver)* (page 16).

4.2 Implementing a new sensor

This section assumes that the PHY and SCB are already implemented:

1. Implement the sensor functions
2. Make it available in sensor API

Note: CMake has several options to choose which sensors to compile, the default values are available in `FindSensorAPI.cmake`. The options can be modified to fit the sensor available on the customer's platform to avoid adding sensor drivers to unavailable sensors.

4.2.1 Sensor implementation

- Create a sensor implementation file (e.g. `ar330.c`)
- This file needs to implement as many of the Sensor API functions defined in the `SENSOR_FUNCS` as possible.
- This file must also implement the initialisation function for the sensor which populates the `SENSOR_FUNCS` structure.
- This should initialise a PHY structure using `SensorPhyInit()`

Use one of the samples which most closely resemble the configuration of the selected sensor as an example (e.g. AR330 for MIPI, P401 for Parallel, Internal DG for data generators).

Un-implemented functions can be left as NULL in the `SENSOR_FUNCS` if they are described as optional. Any function not marked as optional **HAS TO BE IMPLEMENTED** even if they do not do a thing (e.g. Internal Data Generator cannot change exposure but implements the function).

The `sensorapi.c` file must have access to both the new sensor name and initialise function. Create a header containing the declaration of the initialise function:

- Modify the `InitialiseSensors[]` array to have the new initialise function in its list
- Modify the `Sensors[]` to have the sensor name in the same offset than the initialise function was inserted.

If the sensor supports extended parameters they can be listed in that header too for the applications to have an easy access to it as well.

Warning: It is important that mode setting and enabling are handled separately. The mode configures the output and frame-rate for a sensor before acquiring information about the sensor. The enable function starts sending data to the ISP. The separation is important because the V2500 HW recommends that the sensor is started once the gasket is enabled (but the SW needs to know what the sensor configuration is before starting it).

The enable function is also responsible for enabling the ISP gasket which is done using the PHY structure created in the sensor's initialised function. To enable the gasket sensors pGasket structure of the PHY should be filled in indicating

- The gasket number to be enabled. This is expected to be static (expected 1 to 1 mapping of sensors to gaskets). If the customised PHY can modify which gasket receives the data it should be taken into account.

Customers are **expected** to set the correct value for their SoC for each sensor.

- If this is a parallel or mihi sensor (this is **IMPORTANT** as V2500 has 2 different gasket types that are part of customer configurations!)
- The width, height, bit-depth of the mode
- For a parallel sensor it is **IMPERATIVE** that it also contain the Gasket width, height and the sync polarity of H sync and VSync. Examples of these are in the sample P401 drivers.

Having those parameters set for MIPI sensors have no effects.

The enable function should then call the SensorPhyCtrl() function with the parameters indicating enable/disable number of mihi lanes (for mihi sensors) and the sensor number. If there is any additional configuration for an external PHY it should be done in the SensorPhyCtrl() function.

Note: The IMG PHY also asks for a `ui8SensorNum` because our sensor board can have several sensors attached. If the customer's PHY does not support such selection then this parameter could be ignored.

4.2.2 Additional information

The sensor API is expected to provide information about the selected sensor (stored into the SENSOR_INFO structure). The needed information is (not limited to):

- Aperture f/# of the attached lens
- Focal length of the lens in millimeters
- Usable full well depth (number of electrons the sensor wells can hold before clipping)
- Bit depth of the sensor data
- Read noise of the sensor in electrons

Additional functions may provide more information about the selected running mode:

- GetGainRange() as possible gain multipliers for the sensor

- GetExposureRange() as exposure lengths in microseconds
- GetFocusRange() as distances in millimeters

The SENSOR_MODE structure contains information about a running mode for a sensor (a sensor can have several modes to run faster/slower etc). That contains:

- Width of the sensor output in pixels
- Height of the sensor output in pixels
- Frame rate of the sensor in Hz
- The total number of lines including blanking time to be used by the flicker detection

4.3 Example sensor drivers

The three real sensor drivers provide examples on how to interface to different types of sensor (parallel or MIPI) with different possible configurations.

4.3.1 Omnivision OV4688

Implementation is available in ov4688.h and ov4688.c. The OV4688 driver communicates with a MIPI sensor which supports exposure, gains and focus control.

The mode setting registers are stored in a table which is written in a loop at enable time (not configured like other sensors). The sensor uses 16 bit addresses and 8 bit data. The focus is controlled by a voice coil motor which is controlled by an I2C device on another address. The sensor driver demonstrates the setup of multiple I2C devices on the same bus.

Note: That driver had to be modified to have the whole setup done in enable instead of when selecting the mode as the sensor would otherwise start sending frames even if it was not enabled yet.

The data sheet for the VCM provides register values for 4 focus distances, the driver interpolates between these to provide a continuous range. The interpolation is done using the reciprocal of the focus distance as this is much more linear than the focus distance as shown in Figure *Distance (mm) over register values* (page 38) and *Distance (mm) over register values* (page 38).

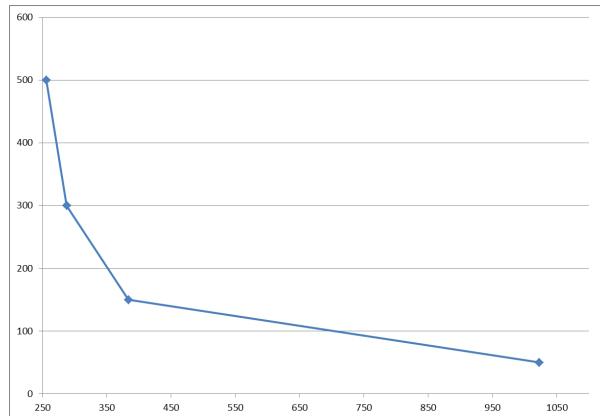


Fig. 4.2: Distance (mm) over register values

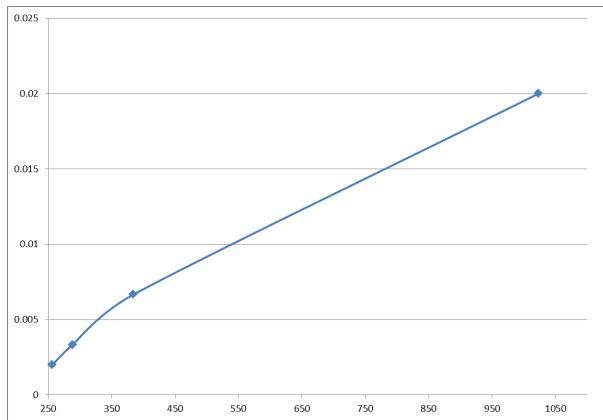


Fig. 4.3: Distance (mm-1) over register values

4.3.2 Aptina AR330

Implementation is available in `ar330.h` and `ar330.c`. The AR330 driver is a MIPI sensor that supports exposure and gains controls as well as modes using different number of MIPI lanes.

This driver demonstrates extracting of information from the mode configuration register in order to calculate the number of MIPI lanes, and minimum exposure time. Also provided in the driver is an example for controlling flash.

Note: Flash control has not been tested but has been done following the AR330 documentation.

4.3.3 Aptina P401

Implementation is available in `p401.h` and `p401.c`. This is a very simple driver showing the usage of a parallel gasket that supports exposure and gains controls.

Note: The `P401_Create()` has a special feature searching for the 1st parallel gasket. The customers are expected to know which one is their first parallel gasket and therefore this function can be replaced by directly writing the number of the gasket.

4.3.4 Internal Data Generator

Implementation is available in `iifdatagen.h` and `iifdatagen.c`. The Internal Data Generator Sensor driver is provided as a way of passing image to the internal data generator that is part of some ISP versions.

This sensor is available in HW and can be used both with CSIM and silicon to insert images. However it only supports FLX as an input format but could be modified to get the image from other sources (the customer can implement a different frame converter to generate the data the IIF Datagen needs).

The Internal Data Generator demonstrates a simple implementation of an insertion capable sensor. It does not handle gains or exposure and has a simpler implementation than the external data generator. It also can be used as an example of using extended parameters.

Chapter 5

Test Tools

This chapter aims to detail how the several command line test tools can be used to run tests on the V2500 DDK. Please refer to the [Getting Started Guide](#) (page 6) for details on how to build the drivers and insert the kernel module. This document also covers the GUI tools.

The target reader is assumed to know the V2500 IP software and hardware, its capabilities and deliverables. This chapter introduces the tools and parameter files that can be used to run the driver test applications in different conditions.

5.1 Running modes

Regardless of the tool used the running mode is either using the Fake interface (connection to the simulator) or the kernel-module. Some tools will only run against the kernel-module (e.g. [ISPC_capture](#) (page 76)).

5.1.1 Running against the simulator

This section assumes that a compiled version of the V2500 C Simulator is available (compile it using the CSIM documentation or use the provided binary package). The CI should be compiled using the Fake Interface. The internal library that manages register access (TAL) handles different protocols to access the device. The protocol used for connecting to the simulator using TCP/IP is called *devif*. Before running the application you will need to start the simulator:

```
$ ./testbench -testbench devif_slave -devifportnum 2345 <other options>
```

Please refer to the *User Guide* which accompanies the simulator package to find more details about other options. A potentially important option is the **-hwcfg** this should be set to match the hardware configuration you are using.

The simulator can be compiled as a library and executables linked against it in a virtual platform system (e.g. android emulator uses this technique). This interface is called *transif*.

5.1.2 Running against the kernel-module

The only requirement is to insert the kernel module prior to run the test (as root):

```
$ insmod Felix.ko
```

Refer to the *V2500 Insertion options* (page 17) for more details on the kernel-module options.

If using sensors with IMG SCB device the i2c-driver should be inserted (or enabled in the kernel build):

```
$ insmod /lib/modules/`uname -r`/kernel/drivers/i2c/i2c-dev.ko
```

5.2 Register override

If the driver was compiled using the Fake interface **and** the debug functions (see the *Getting Started Guide* (page 6)) the register override functionality is available to *driver_test* (page 49) and *ISPC_test* (page 61). This allows some specific registers to be overridden prior to submitting a configuration. This should only be considered by advanced users who know the pipeline well.

The test application is looking for a file following the format `drv_regovr_ctxX_frmD.txt` where X is the context number and D the frame number if `-CTX_RegOverride` option is set to 1. Only the registers loaded from the HW load structure are available to be overridden and the file format is:

```
<bank name> <field name> <0x prefixed hex value>\n
```

The file is read line by line and any other format is potentially treated as a comment (this is a debugging option and is not aiming to be resilient). More information is available in *Register override* (page 168).

5.2.1 Register override file example

```
// SHA_PARAMS_0
REG_FELIX_CONTEXT_0 SHA_THRESH 0x3B // 59
REG_FELIX_CONTEXT_0 SHA_STRENGTH 0x14 // 20
REG_FELIX_CONTEXT_0 SHA_DETAIL 0x16 // 22
REG_FELIX_CONTEXT_0 SHA_DENOISE_BYPASS 0x0 // 0

// SHA_PARAMS_1
REG_FELIX_CONTEXT_0 SHA_GWEIGHT_0 0xB // 11
REG_FELIX_CONTEXT_0 SHA_GWEIGHT_1 0xE // 14
REG_FELIX_CONTEXT_0 SHA_GWEIGHT_2 0x6 // 6

// SHA_ELW_SCALEOFF
REG_FELIX_CONTEXT_0 SHA_ELW_SCALE 0x5D // 93
REG_FELIX_CONTEXT_0 SHA_ELW_OFFSETS 0x06 // 6

// SHA_DN_EDGE_SIMIL_COMP_PTS_0
REG_FELIX_CONTEXT_0 SHA_DN_EDGE_SIMIL_COMP_PTS_0_TO_3[0] 0xA1 // 161
REG_FELIX_CONTEXT_0 SHA_DN_EDGE_SIMIL_COMP_PTS_0_TO_3[1] 0x41 // 65
REG_FELIX_CONTEXT_0 SHA_DN_EDGE_SIMIL_COMP_PTS_0_TO_3[2] 0x3D // 61
REG_FELIX_CONTEXT_0 SHA_DN_EDGE_SIMIL_COMP_PTS_0_TO_3[3] 0x6E // 110

// SHA_DN_EDGE_SIMIL_COMP_PTS_1
REG_FELIX_CONTEXT_0 SHA_DN_EDGE_SIMIL_COMP_PTS_4_TO_6[0] 0x89 // 137
REG_FELIX_CONTEXT_0 SHA_DN_EDGE_SIMIL_COMP_PTS_4_TO_6[1] 0x10 // 16
```

```

REG_FELIX_CONTEXT_0 SHA_DN_EDGE_SIMIL_COMP PTS_4_TO_6[2] 0x49 // 73

// SHA_DN_EDGE_AVOID_COMP PTS_0
REG_FELIX_CONTEXT_0 SHA_DN_EDGE_AVOID_COMP PTS_0_TO_3[0] 0x95 // 149
REG_FELIX_CONTEXT_0 SHA_DN_EDGE_AVOID_COMP PTS_0_TO_3[1] 0xC1 // 193
REG_FELIX_CONTEXT_0 SHA_DN_EDGE_AVOID_COMP PTS_0_TO_3[2] 0xD4 // 212
REG_FELIX_CONTEXT_0 SHA_DN_EDGE_AVOID_COMP PTS_0_TO_3[3] 0x2F // 47

// SHA_DN_EDGE_AVOID_COMP PTS_1
REG_FELIX_CONTEXT_0 SHA_DN_EDGE_AVOID_COMP PTS_4_TO_6[0] 0x7C // 124
REG_FELIX_CONTEXT_0 SHA_DN_EDGE_AVOID_COMP PTS_4_TO_6[1] 0xFB // 251
REG_FELIX_CONTEXT_0 SHA_DN_EDGE_AVOID_COMP PTS_4_TO_6[2] 0xF5 // 245

```

5.3 Common Output

Most of the test applications share the same output rules.

5.3.1 Format strings

The YUV and RGB format strings are generated in the code from the pixel format:

```
DDKSource/common/felixcommon/source/pixel_format.c FormatString()
```

More information about each format is available in the HW TRM documentation.

YUV formats

NV21: For 8b 420 Y/**VU**, chroma is interleaved (NV12 is **UV** order).

NV61: For 8b 422 Y/**VU**, chroma is interleaved (NV16 is **UV** order).

NV12-10bit and NV21-10bit: Are 10b equivalent of NV12 and NV21. See HW documentation about the packing of the data.

NV16-10bit and NV61-10bit: Are 10b equivalent of NV16 and NV61. See HW documentation about the packing of the data.

RGB formats

BI_RGB24: For 8,8,8 RGB in 24b (B in LSB).

Also available as BGR24 with R in LSB.

BI_RGB32: For 8,8,8 RGB in 32b (alpha is not used, B in LSB).

Also available as BGR32 with R in LSB.

BI_RGB32-10bit: For 10,10,10 RGB in 32b (B in LSB).

Aslo available as BGR32-10bit with R in LSB.

BI_BGR32-10bit: For 10,10,10 RGB in 32b (R in LSB) - HDR Extraction in V2500 v2 only.

BI_BGR64-16bit: For 16,16,16 RGB in 64b (R in LSB) - HDR Insertion in V2500 v2 only.

Bayer formats

RGGB8: For 8b Bayer data (see HW documentation about the packing of the data).

RGGB10: For 10b Bayer data (see HW documentation about the packing of the data).

RGGB12: For 12b Bayer data (see HW documentation about the packing of the data).

TIFF10: For 10b Bayer data with TIFF byte aligned packing (see HW documentation - RAW 2D Extraction in V2500 v2 only).

TIFF12: For 12b Bayer data with TIFF byte aligned packing (see HW documentation - RAW 2D Extraction in V2500 v2 only).

5.3.2 Statistics Format

The output statistics are mostly raw statistics from the HW (binary - see Save Structure details in the HW TRM) with an additional header and footer:

```
#stats-<size>B#
<binary data from HW>
#stats-done#
```

The size being rounded up for page allocation in the driver will most likely always be 8192B. The header is therefore 13 Bytes and the footer 12 Bytes.

Note: When using 16kB pages the save structure is 16384B the header becomes therefore 14 Bytes.

Let's assume a 4kB CPU page and a 2 frame run. The statistic file may look like (depending on the test application used):

```
#stats-8192B#
<8192 Bytes in HW format>
#stats-done#
#stats-8192B#
<8192 Bytes in HW format>
#stats-done#
```

Each frame uses:

$$\text{stats size (Bytes)} = \text{header size} + \text{stats size} + \text{footer size}$$

In our example the header is 13 Bytes (1 per character) and footer 12 Bytes (12), each frame therefore uses 8217 Bytes ($13 + 8192 + 12 = 8217$).

Let's imagine we want to access the information in the statistics at offset 0x6C (decimal offset: 108 Bytes):

- the information for frame 1 should therefore be at header size + offset = $13 + 108 = 121 = 0x79$
- the information for frame 2 should be at stats size+header size+offset = $8217 + 13 + 108 = 8338 = 0x2092$

By extension the offset for frame n (from 1) is

$$(n - 1) * \text{stats size} + \text{header size} + \text{offset}$$

Note: The size reported in the header vary when using ISPC or CI based applications. CI based application will report the full page size (4096 or 16384 depending on CPU page size) and save it all to disk. ISPC based application will report only the effective statistics size (e.g. 5188 Bytes).

Text statistics format

The ISPC::Save class allows the test application such as ISPC_test or ISPC_loop to also save the statistics as text. The format follows the syntax:

```
frame:
  Title:
    config:
      # comment
      MC_field = (value value value)
  content:
    MC_field[0][1] = value
```

Some of the field contain description to help the analysis of the statistics but it is expected that the reader knows how to interpret those statistics.

The text format also contains the DPF output map and the ENS output as plain text.

Warning: The DPF input map is currently not supported and will always show as null even if the number of entries is correct.

5.3.3 Defective Pixel output map

The defective pixel map generated by all test applications is currently the raw HW format described in the TRM:

5.3.4 Encoder statistics output

The Encoder statistics map generated by all test applications is currently the raw HW format described in the TRM:

5.4 Common Input

5.4.1 Deshading Grid file format (LSH)

The deshading grid format used to load a Lens Shading matrix is not the HW format. The matrix is saved as floating points values and encoded to the HW format to allow compatibility between several versions of the HW.

The ISPC::ModuleLSH stores the grid in the same way the MC level does. More information about how the grid is stored in the low-level driver is available in the *Deshading Grid (Lens shading)* (page 149) section.

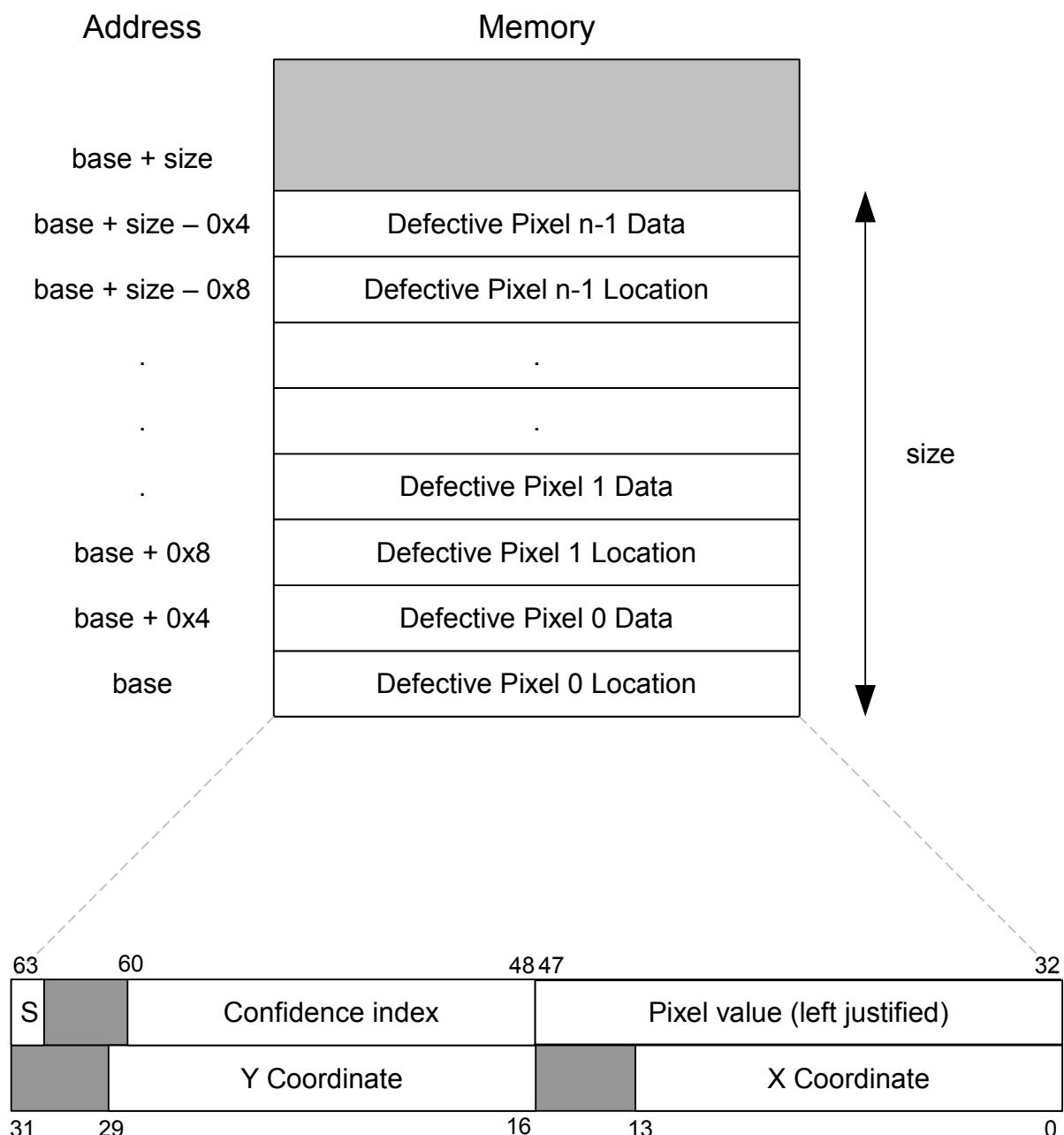


Fig. 5.1: Output Defective Pixel Map Format

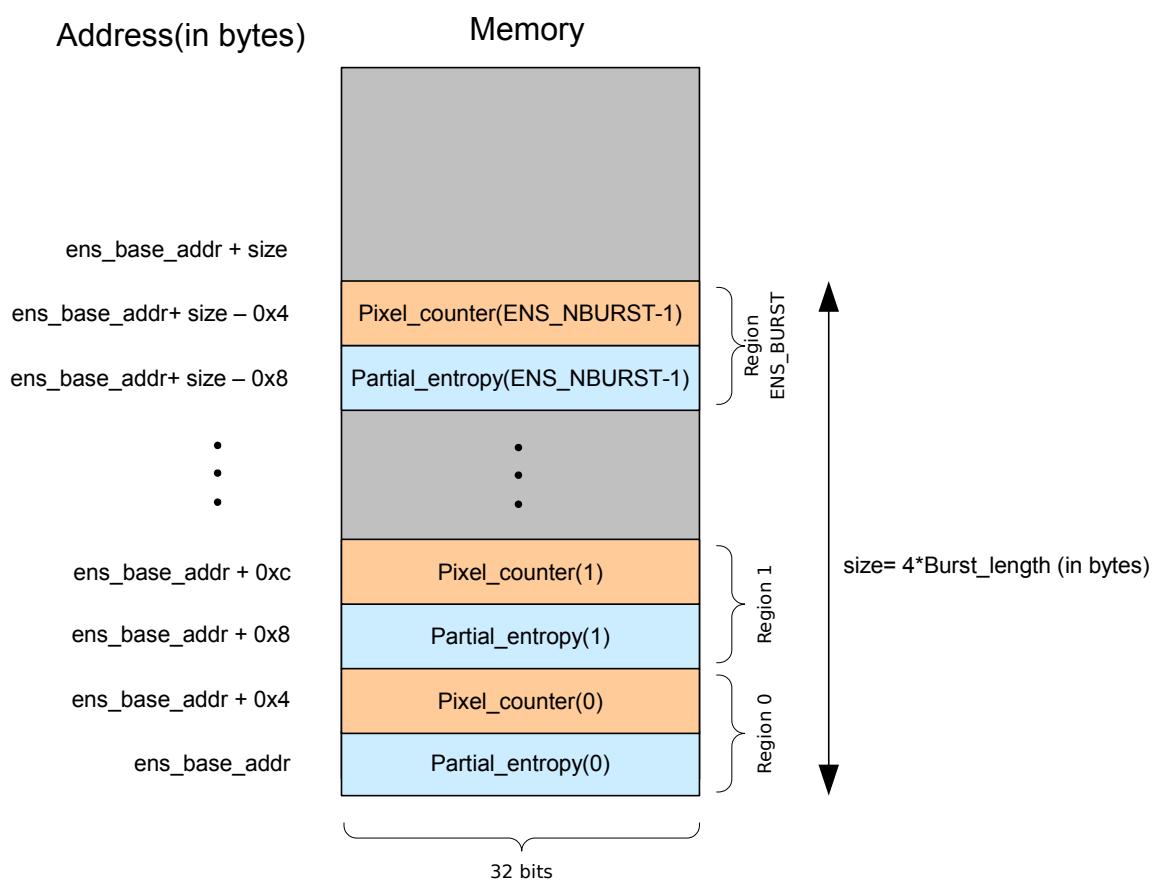


Fig. 5.2: Encoder statistics output Format

Common functions are available to load and save the grid in lshgrid.h.

The format is binary, each channel is stored after the other in raster order.

```
<char*4 == "LSH\0"><i32 version>
<i32 width><i32 height><i32 tile-size>
<float32 channel0>*width*height
<float32 channel1>*width*height
<float32 channel2>*width*height
<float32 channel3>*width*height
```

An example was generated using the tuning tools and is available in the DDK package for the AR330 telescopic lens (`sensorConfig/ar330/ar330/ar330_telescope_16.lsh`).

5.4.2 Defective Pixels Fixing read map (DPF)

Implementation of the loading is done in dpfmap.h. Currently the drivers loads a defective map following the pure HW format described in the TRM as in *Defective Pixel Map Format* (page 48).

The defect map consists of a list of defective pixels given as pixel coordinates that correspond to the full dimensions and resolution of the imager, excluding any optical black border as this is not ‘seen’ by the Defective Pixel Fixing module. The list of pixels is always specified in **raster scan order**.

However, not all defective pixels in the imager need to be present in the map, but only the subset of them which is required for processing of the current image.

The defect map should be a contiguous list of defective pixels, specified in the format shown in Figure *Defective Pixel Map Format* (page 48).

More information about the DPF module is available in the *Defective Pixels Fixing (DPF)* (page 207) section.

5.5 Command line application user guide

The available command line applications are:

- *Capture Interface test: driver_test* (page 49)
- *ISP Control test: ISPC_test* (page 61)
- *ISPC loop application: ISPC_loop* (page 69)
- *ISP Control video capture: ISPC_capture* (page 76)
- *ISP Control HDR: ISPC_hdr* (page 77)
- *Defective Pixel output converter: dpf_conv* (page 82)
- *Sensor Test application sensor_test* (page 85)
- *Console Raw Mode testing* (page 86)

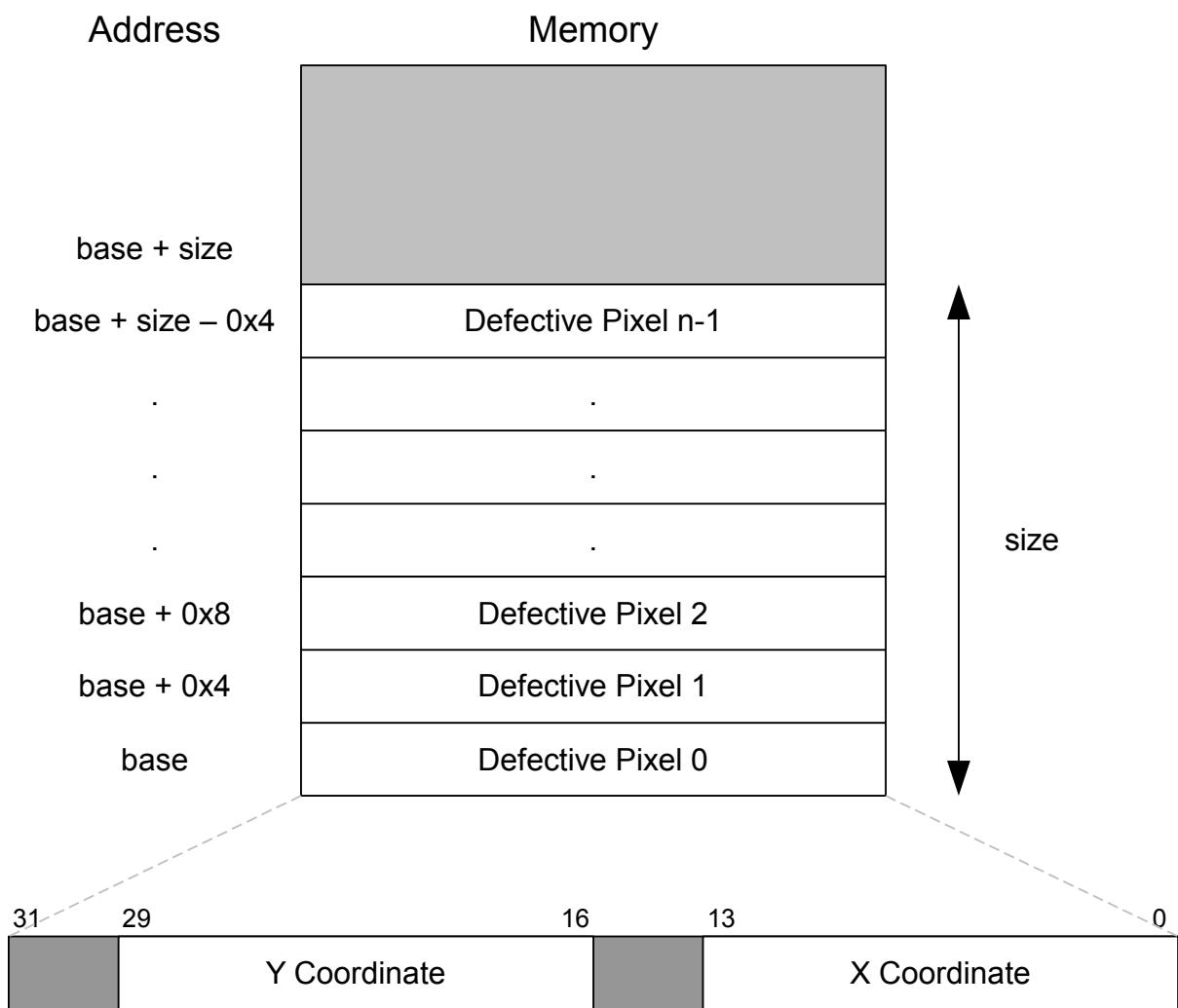


Fig. 5.3: Defective Pixel Map Format

5.5.1 Capture Interface test: driver_test

This application is part of the Capture Interface (CI) low level driver. It is designed to run very simple test with the driver therefore it does not include any image quality related parameters. The aim is to produce different *scenarii* that can be used to test the HW and SW at low level.

This tool can use the external or internal data-generator if they are present (compilation time choice, see *Getting Started Guide* (page 6)) but **cannot** use a real imager.

This tool can be used with the real driver or the fake interface (simulator). The options are not chosen at run-time for the simple reason that running against the simulator needs to fake the insmod operation. When running with the real driver the kernel-module should be inserted before using this tool.

The executable is located in the CI library. The sources are in `CI/felix/driver_test` and are not as simple as they could be because they incorporate a lot of the logic that the higher software layers have.

The tool is expecting an FLX format image file to use for the data-generator and is going to output one or more FLX (containing RGB or Bayer) and/or YUV files as output. Each HW context can be programmed individually and the tool should be using most if not all of the available CI functionalities.

The Data Generators are used as imagers (imager 0 refers to Data Generator 0 and *vice-versa*).

The executable should be available in the installed folder `CI/` after make install.

Parameters

The parameters can be given by command line or in a configuration file.

The parameter types are:

- *Bool*: a Boolean value of either 0 or 1
- *Int / UInt*: an integer (signed or unsigned is parameter specific)
- *Cmd*: command that act as a Boolean of value 1 if present, of value 0 if not found
- *Str*: string or path
- *Float*: floating point number

-source: Provide a file to parse for command line parameters. The latest given parameter is used as the value for the option.

-h: Display the help and exit.

Generic Parameters

These parameters affect the application as a whole.

Similar to insmod (fake driver only) These parameters all have `insmod` equivalent with the same name.

-mmucontrol: Int <0, 1, 2> Use the device MMU. If enabled Device Virtual Addresses are used.

- 0 – bypass MMU (virtual=physical)
- 1 enables MMU but does not enable extended address range (32b MMU).
- 2 – using extended address range (40b MMU)

Only affects runs with Fake interface.

Warning: The value 1 is supported by the IMG BUS4 MMU. However the V2500 HW was not tested with this option as CSIM does not support it, use at own risks.

-tilingScheme: Int <256, 512> Choose the Device MMU tiling scheme. Either 256 for 256x16 or 512 for 512x8.

Only affects runs with Fake interface with MMU enabled and may not be available on all HW versions.

-tilingStride: Int If non-0 used as common stride when tiled buffers are used. This has to be a power of 2 and has to be big enough for all enabled output (even non-tiled ones that could be tiled).

Only affects runs with Fake interface with MMU enabled and may not be available on all HW versions.

-gammaCurve: Int Change the default gamma curve used. More details about Gamma Curve selection available in the *Getting Started Guide* (page 17).

-extDGPLL: UInt [x2] When using the external data-generator can be used to change the speed of processing. Provide the multiplier and divider to the test IO bank (**not part of the ISP IP**) - both values need to be specified.

If 0 is given (default) then the data-generator driver computes the default PLL $\frac{\text{parallesim}}{\text{parallesim}} \times 10$

Warning: If incorrect values are provided the PLL may not lock up and the FPGA will need to be reprogrammed.

If given values are too high the test may not generate frames because the input rate will be too high for the ISP to handle.

-cpuPageSize: UInt CPU page size in kB. Interesting values are 4kB and 16kB.

Note: This is **not** an insmod parameter as it is defined by the OS.

-no_pdump: Cmd Does not generate pdumps when running.

Note: This is *not** an insmod parameter as real driver does not generate pdumps.

Global Parameters

-data-extraction: Int The data extraction point to use in HW. Value should be a valid CI_INOUT_POINTS value (see CI header)

-encOutCtx: UInt [x2] Which context to use for the Encoder pipeline statistics output: 1st value is context, second value is the pulse width in clock (see FELIX_CORE:ENC_OUT_CTRL).

In HW the direct output sends PARALLELISM signals every PARALLELISM lines.

- dump_params: **Cmd** Generate a text file with all the used parameters
- parallel: **Cmd** Use the parallel capture mode (see *Behaviour* (page 59))
- no_write: **Cmd** Do not write output to file (including statistics).
 - Is interesting when running on system with high memory latency.
- ignoreDPF: **Bool** Override DPF results in statistics and skip saving of the DPF output file.
 - Use this flag for verification of output between CSIM and HW as both may not produce the same DPF output.
- hw_info: **Cmd** Prints the HW information from the Connection object.

Note: When using CSIM a connection to the CSIM is needed to read the CORE registers.

- writeDGFrame: **Bool** Write converted DG frame to text file.

Fake interface only These parameters are related to the CSIM or functionalities available to the fake driver only.

- simIP: **Str** Simulator IP address when using the Fake interface with TCP/IP.
 - Default is localhost.
- simPort: **UInt** Simulator port when using Fake interface with TCP/IP.
 - Default is 2345.
- simTransif: **Str** Path the simulator transif library when using the Fake interface with transif.
 - Also needs -simconfig.
 - If not specified TCP/IP fake interface is used.
- simConfig: **Str** Configuration file to give to the simulator when using the transif Fake interface.
- simTransifTimed: **Bool** Use timed mode for transif.
- simTransifDebug: **Bool** Debug transif operations
- pdumpEnableRDW: **Bool** Enable generation of RDW commands in pdump output (not useful for verification). Only affect Fake driver. Default is 0.
- register_test: **Int <0, 1, 2>** Run the register test, internal HW test used to generate pdumps to verify the FPGA.
 - 0 – no test
 - 1 – read/write all registers instead of running test
 - 2 – read/poll all registers in context at the end of the test (expected addresses)

Context Parameters

The context parameters are defined per HW context. The X should be replaced by the context number starting from 0 (e.g. -CT0_encoder).

Ouput format options

-CTX_encoder: **Str** Enables the encoder output (YUV) and defines the output filename. If the filename does not finish by .yuv the size and format will be added to it (e.g. `encoder0` becomes `encoder0-360x280-NV21.yuv`).

The output format is by default NV21 (420 8b) but can be changed with other options.

-CTX_YUV422: **Bool** The encoder output should be YUV 422 instead of 420.

-CTX_YUV10b: **Bool** The encoder output should be 10b (packed) instead of 8b.

-CTX_rawYUV: **Bool** Enable output of the raw YUV output from HW (i.e. do not remove the strides - adds an additional output file).

-CTX_display: **Str** Enables the display output (RGB) and defines the output filename. Cannot be enabled with data extraction (HW design).

The output format is by default RGB8 (888 32b) but can be changed with other options.

-CTX_RGB24: **Bool** The display output should be RGB24 (888 24b) instead of RGB8 (888 32b).

-CTX_RGB10: **Bool** The display output should be RGB10 (101010 32b) instead of RGB8 (888 32b) – takes precedence over RGB24 option.

-CTX_dataExtraction: **Str** Enables the data extraction using the Generic parameter to define which DE point to use. Cannot be enabled with display (HW design).

The output format is using the bitdepth of the pipeline by default.

-CTX_DEBitDepth: **UInt <8, 10, 12>** Data Extraction Bayer output bitdepth, 0 for pipeline bitdepth (default), <10 for 8b, >10 for 12b and 10 for 10b.

-CTX_rawDisp: **Bool** Enable the saving of raw display or data-extraction outputs (no stride removal).

Additional outputs (V2500 v2 only) These outputs may not be available for all HW configurations. If the HW does not support them these options will be ignored.

-CTX_raw2DExtraction: **Str** Enable the Raw 2D Extraction for this context and provides filename to use for output. The output is transformed to be displayed as an RGGB FLX file.

-CTX_raw2DFormat: **UInt <10, 12>** Raw 2D format bit-depth to use (10b or 12b).

-CTX_rawTiff: **Bool** Additional output of the raw TIFF format from RAW 2D output.

-CTX_HDRExtraction: **Str** The filename to use as the output of the HDR Extraction. Format is always RGB10 in 32b.

-CTX_rawHDR: **Bool** Additional output of the raw HDR Extraction buffer that HW provided.

-CTX_nEncTiled: **UInt** Give a number of Encoder buffers to have tiled (if same than **-CTX_nBuffers** then all buffers are tiled).

`mmucontrol` must be ≥ 1 and `tilingScheme` must be legitimate.

-CTX_nDispTiled: **UInt** Give a number of Display buffers to have tiled (if same than **-CTX_nBuffers** then all are tiled).

`mmucontrol` must be ≥ 1 and `tilingScheme` must be legitimate.

-CTX_nHDRExtTiled: **UInt** Give a number of HDR Extraction buffers to have tiled (if same than -CTX_nBuffers then all are tiled).

mmucontrol must be ≥ 1 and **tilingScheme** must be legitimate.

Note: Raw 2D output cannot be tiled by HW (byte alignment while tiling needs page alignment).

Additional input (V2500 v2 only) These inputs may not be available for all HW configurations. If the HW does not support them these options will be ignored.

-CTX_HDRInsertion: **Str** The filename to use as the input of the HDR Insertion if none do not use HDR Insertion. The image has to be an FLX in RGB of the same size than the one provided to DG but does not have to be of the correct bitdepth (max supported bitdepth is 16b per channel).

This will replace the image in the pipeline and Encoder and Display output will use that image as source.

-CTX_HDRInsRescale: **Bool** Rescale provided HDR Insertion file to 16b per pixels. Does not downscale.

Run options Configures the number of frames and buffers to run with and some other test options.

-CTX_nBuffers: **UInt (>0)** Number of image buffer to have ready for this context.

-CTX_nFrames: **Int** Number of frames to capture using that context. Negative value will use the number of image available from the associated DG (see imager option). Default: -1.

-CTX_imager: **UInt** The imager to use. By default it is the same than this context (e.g. CT1 uses imager 1).

-CTX_checkCRC: **Bool:** Add additional POL in pdump for HW testing that check CRC at the end of frame. Does not validate that CRCs are correct.

Note: Available only when running Fake Interface to generate pdumps for the HW testing.

Allocation size options These options can be used to change the allocation size, offset and stride of each output. These options are for advanced users who are familiar with the pixel formats requirements.

Note: The driver side will check that each allocation respect the rule:

<code>alloc >= offset + (stride * height)</code>

Warning: The offset parameters are ignored when using tiled buffers because the DDK does not allow offsets to be used when using tiling (because of virtual memory alignment explained in MMU Tiling information (page 179)).
--

- CTX_YUVAlloc:** **UInt** Size to allocate for the YUV buffer in bytes (multiple of 64 Bytes required). Use 0 (default) to let driver compute the size.
- CTX_YUVStrides:** **UInt [x2]** Strides for the Y and CbCr (multiple of 64 Bytes required). Use 0 (default) to let the driver compute the stride. If only one is given it is assumed to be luma stride and is duplicated to the chroma stride.
- CTX_YUVOffsets:** **UInt [x2]** Offset for Y and CbCr buffers in bytes (multiple of 64 Bytes required). A CbCr offset of 0 means computed by the driver (default).
 - If only one is given it is assumed to be the luma offset, and chroma offset will be set to 0.
 - Ignored if buffer is tiled.
- CTX_DisAlloc:** **UInt** Display output (or DE output) allocation size in bytes (multiple of 64 Bytes required). Use 0 (default) to let driver compute the size.
- CTX_DisStride:** **UInt** Display stride (or DE stride) in bytes (multiple of 64 Bytes required). Use 0 (default) to let the driver compute the stride.
- CTX_DisOffset:** **UInt** Display offset (or DE) in bytes (multiple of 64 Bytes required). Default is 0.
 - Ignored if buffer is tiled.
- CTX_raw2DAlloc:** **UInt** Raw 2D allocation size in bytes (multiple of 64 Bytes required). Use 0 (default) to let driver compute the size.
- CTX_raw2DStride:** **UInt** Raw 2D stride in bytes (does **NOT** need to be a multiple of 64 Bytes). Use 0 (default) to let driver compute the stride.
- CTX_raw2DOffset:** **UInt** Raw2D offset in bytes (does **NOT** need to be a multiple of 64 Bytes). Default is 0.
- CTX_HDRExtAlloc:** **UInt** HDR Extraction allocation size in bytes (multiple of 64 Bytes required). Use 0 (default) to let the driver compute the size.
- CTX_HDRExtStride:** **UInt** HDR Extraction stride in bytes (multiple of 64 Bytes required). Use 0 (default) to let driver compute the stride.
- CTX_HDRExtOffset:** **UInt** HDR Extraction offset in bytes (multiple of 64 Bytes required). Default is 0.
 - Ignored if buffer is tiled.
- CTX_updateMaxSize:** **Bool** Update maxEncOutWidth maxEncOutHeight maxDispOutWidth and maxDispOutHeight using pitch computed output size. In other words will allocate the minimum size buffers if no allocation size is provided. Default is off and the allocation will be big enough to cover the whole input image resolution.

Configuration options Additional output configuration for the pipeline such as statistics and output sizes.

-**CTX_stats:** **Int** If -1 given all the statistics are enabled for this context or use the same value as the save flags in linked list to enable stats one by one.

Statistics are stored into `ctx_X_stats.dat` binary file. See also DPF option for extra statistics.

See *Statistics Format* (page 43) for details about the saved format.

-CTX_escPitch: Float [x2] Encoder output vertical and horizontal scale down pitch (input/output ratio). By default it is 1.0 (no downscaling).

If only 1 value is given the same scaling factor is applied vertically and horizontally.

Only relevant if the encoder output is enabled.

-CTX_dscPitch: Float [x2] Similar to -CTX_escPitch for the display output.

-CTX_DPF: UInt <0, 1, 2> Enable defective pixel detection:

- 0 disabled,
- 1 detect and correct,
- >1 detect+write result into statistics file (see stats option).

A separate advance option is available to enable the read-map.

Advanced configuration options These options are targeting advanced users who know a bit more about the pipeline and its modules.

-CTX_regOverride: Bool Allow register override if a file is present (see *Register override* (page 41)).

Note: Available only with Fake Interface.

Warning: This option should be avoided as it requires a complete knowledge about the registers to choose the correct value and does not work with real drivers.

-virtHeapOffset: UInt [x7] Virtual address heaps offsets in bytes - will be rounded up to MMU page size.

Warning: This option is only available with the Fake interface and should be used only to generate very specific tests to prevent virtual memory from being used. Refer to the *Virtual Address management and heaps* (page 176) section to know about virtual heaps.

-CTX_enableTS: Bool: Enable timestamps generation. By default with real driver timestamps are enabled but they can be disabled when doing HW output verification against results generated with CSIM (e.g. statistics may not be binary equivalent as they contain timestamp).

Note: No effect when running Fake driver: CSIM does not generate timestamps therefore the driver does not enable them.

-CTX_useLSH: UInt <0, 1, 2> Use the deshading grid module (LSH):

- 0 disables,
- 1 generate linear grid from 4 corners,
- 2 generate grid from 5 corners (4 corners + center). See -CTX_LSHCorners option.

-CTX_LSHtile: **UInt** The deshading tile size in CFA (between LSH_TILE_MIN and LSH_TILE_MAX and has to be a power of 2) - used to compute deshading grid size using input FLX size and corner values.

-CTX_LSHCorners: **Float [x20]** The corner values used to generate the matrix (TL, TR, BL, BR, CE) per channel. Center is only used if -CTX_useLSH is 2 but has to be specified.

-CTX_fixLSH: **Bool** Fix the LSH grid if the differences are too big; may not generate a nice grid but should run.

-CTX_LSHGrads: **Float [x8]** The gradients values {X,Y} per channel to be written to registers after precision is applied.

-CTX_black: **UInt** Aimed REGISTER black level value for the context (default is equivalent to 64).

-CTX_ENSNLines: **UInt** The Encoder Statistics number of lines per region (changes the output size of the ENS block). See the register definitions (must be a power of 2). Used to be wrongly called -CTX_ENSN Couples.

-CTX_DPFReadMap: **Str** Path to a DPF input map file (following DPF read map format for HW).

-CTX_RLTMode: **UInt** Raw Look Up Table mode selection:

- 0 disabled
- 1 linear correction (i.e. 1 curve applied on all channels)
- 2 cubic correction (i.e. 1 curve per channel)

Changes meaning of -CTX_RLTSlopes

-CTX_RLTSlopes: **Float [x4]** Raw Look Up Table slopes values.

If mode is linear, slope values for each knee section of the curve (slope[0] is between 0 and knee[1], slope[1] between knee[0] and knee[1], slope[2] between knee[2] and knee[3] and slope[3] between knee[3] and 65535. See -CTX_RLTKnee to define knee positions.

If mode is cubic, slope value per channel in R, G1, G2, B order.

-CTX_RLTKnee: **UInt [x3]** Raw Lookup Table knee point values for linear mode. Knee points for the curve (0 is implicitly the 1st value) are between 1 and 65535:

0 - knee[0] - knee[1] - knee[2] - 65536

-CTX_AWSCoeffsX: **Float [x5]** If HW supports it (from 2.7) change the AWS line segment horizontal coefficients. 1 value per segment.

-CTX_AWSCoeffsY: **Float [x5]** If HW supports it (from 2.7) change the AWS line segment vertical coefficients. 1 value per segment.

-CTX_AWSOffset: **Float [x5]** If HW supports it (from 2.7) change the AWS line segment offsets. 1 value per segment.

-CTX_AWSBoundary: **Float [x5]** If HW supports it (from 2.7) change the AWS line segment boundaries. First value is before segment 0, second value between segment 0 and segment 1. Last boundary is expected to be ‘infinity’.

Boundaries are expected to be decreasing. Very small values (e.g. -16) can be used to represent segments never hit (to run with less than 5 segments).

-CTX_AllocateLate: Bool If false allocate the buffer just after configuring the pipeline, if enabled will allocate the buffers after starting.

Can be used in multi-context to try to simulate allocations while another context is running.

Internal Data Generator parameters

The same parameter format than the context parameters is used for the internal data generator ones.

-IntDGX_inputFLX: Str Path to the input FLX file in RGGB.

-IntDGX_inputFrames: UInt (>0) Number of frames to use from the input file.

E.g. To load from a file with 10 frames but limit the used input to 4 first.

-IntDGX_gasket: UInt Gasket to replace data from with internal data generator.

Related to the **-CTX_imager** option used.

-IntDGX_blanking: UInt [x2] Horizontal and vertical blanking to use. If 2nd value (vertical blanking) is not specified default is used.

-DGX_reload: Bool Enable the frame preloading of the data-generator (HW feature).

External Data Generator parameters

The same parameter format than the context parameters is used for the external data generator ones.

Unlike the internal data-generator, the external data-generator use the gasket associated to their number (i.e. DG0 is associated to gasket 0 and DG1 to gasket 1).

Warning: The external data-generator is not available on silicon but only when using CSIM.

-DGX_inputFLX: Str Path to the input FLX file in RGGB.

-DGX_inputFrames: UInt (>0) Number of frames to use from the input file.

E.g. To load from a file with 10 frames but limit the used input to 4 first.

-DGX_blanking: UInt [x2] Horizontal and vertical blanking to use. If 2nd value (vertical blanking) is not specified default is used.

-DGX_reload: Bool Enable the frame preloading of the data-generator (HW feature).

-DGX_MIPILF: Bool If the associated gasket is MIPI will use the long format (additional line header).

-DGX_mipiLanes: UInt <1, 2, 4> If the associated gasket is MIPI will set the number of MIPI lanes to use. Valid number of lanes are 1, 2 and 4.

Parameter Example

An example of parameter file should be available in the `driver_test` folder (`DDKSource/CI/felix/driver_test/driver_args.txt`). The one here may be a bit different and uses 2 Contexts with a single Internal Data Generator:

```
#-----
# generic
#-----
-mmcontrol 2 # 0 = no MMU - 2 = 40b MMU
-data_extraction 1 # data extraction point in the pipeline
-parallel # run the shoots in "parallel" - i.e. submit them at the same time

#-----
# Pipeline
#-----
-CT0_encoder encoder0 # will be extended with format and size - NV21 (420 8b)
#-CT0_YUV422 0
#-CT0_YUV10b 0
#-CT0_display display0.flx # no RGB output
-CT0_dataExtraction dataExtraction0.flx
-CT0_DEBitDepth 12 # force Bayer bitdepth to 12 regardless of pipeline bitdepth
-CT0_nBuffers 5 # allocates 5 buffers
-CT0_nFrames 10 # run for 10 frames
-CT0_imager 0 # using DG0

#-CT1_encoder encoder1 # no YUV output
-CT1_display display0.flx # RGB8 32b
#-CT1_RGB24 0
#-CT1_RGB10 0
#-CT1_dataExtraction dataExtraction1.flx # no Bayer output
-CT1_nBuffers 1 # allocates 1 buffer
-CT1_nFrames 10 # run for 10 frames
-CT1_imager 0 # use DG0, like CT0 - will process the images independently

#-----
# data generators
#-----
-IntDG0_inputFLX fight_960x540-12b-91.flx # this file has 91 frames
-IntDG0_inputFrames 10 # but load only 10 frames out of the 91
-IntDG0_gasket 0 # replace data on gasket 0
-IntDG0_blanking 960 40 # apply blanking of 960H 40V to limit input bandwidth
```

Alteration of parameters

It is possible to change one of the file parameters without changing the file by giving a value after the `-source` switch. For example the given path to Data Generator 0 expects the file to be in the running folder, which is not necessarily the best place to store images, one can modify it by repeating the parameter after the source command (the last specified parameter should be used):

```
$ ./driver_test -source driver_args.txt -IntDG0_inputFLX \
~isp/FLX/fight_960x540-12b-91.flx
```

Behaviour

This paragraph explains the behaviour of the `driver_test` application in terms of configuring the HW and running captures. The first step is to read the parameters then the driver can be initialised.

Driver initialisation

When running against the simulator the first step is to perform a fake insertion of the kernel module. Remember that in that mode the kernel-module is compiled as a user-side library. The insertion is therefore just an initialisation call of this library.

The connection to the kernel-module is then done using the open function and the configuration of the pipeline can start.

Configuration of the Pipeline

The HW is configured through the CI Pipeline object. One object is created for every enabled HW context. For the data generator a Camera object is created for each Data Generator context that is enabled. The Data Generator is configured first, using the given FLX file to retrieve the image size and information. From this information and the parameters the Pipeline objects are configured. Once configured the capture can start.

Capture Mode

There are 2 capture modes in the test application. The default one is serial: each context is running all its captures one after another. This means only 1 Pipeline object is ever registered as started at one time to the kernel module. For every pair Context-Data generator the capture is run as followed:

- Start context and its associated data generator
- Trigger capture on CT and trigger shot on DG (repeated for each frame)
- Stop context and data-generator

The other mode is enabled with the `-parallel` option. All the Context and Data Generator are run at the same time, interleaving their triggers. A Data Generator is associated with an owner context (the context with the highest number so that the result of the DG is received by all the CTs connected to it) and the triggers are run from CT0 to the last:

- Trigger capture on CT
- If this context has an owned DG, trigger shoot

The parallel mode only makes sense if several contexts are available and enabled.

Expected Output

According to the parameters provided some files are generated from the test application (where X is the context number):

- One encoder output file per context (may have additional raw output)

- One display or one data-extraction file per context
- One statistics file per context `ctx_X_stats.dat` in working directory
- Some deshading grid files, text and binary `lsh_matrix_ctxX.dat` in working directory
- One DPF output file `ctxX-F-dpf_write_S.dat` where F is the frame and S is the number of defective pixels.
- One ENS output file (raw HW binary format) `ctxX-F-ens_out-S.dat` where F is the frame and S is the number of defects
- If used one RLT output file in text `ctxX_rlt_curve.txt`

The encoder and display output may be smaller than the given input file but their image quality should remain similar to the input.

If tiling is supported additional files may be written in the working directory:

- One encoder output file per context `tiledF_WxH_(TwxTh-S).yuv` where F is the frame number, W is the width in pixels, H is the height in pixels, Tw is the tile width used, Th the tile height used and S the stride
- One display output file per context `tiledF_WxH_(TwxTh-S).rgb` following the same format that the encoder output
- One HDR output file per context `tiledF_WxH_(TwxTh-S)-hdr.rgb` following the same format that the encoder output

Some Data Extraction points were placed on the pipeline to be used in the GPU or other HW elements and therefore the test application also provides raw HW binary files that could be used to validate them:

- If enabled one raw HDR extraction file `hdrextF_WxH-str_S-lines_L.rgb` where F is the frame number, W is the width in pixels, H is the height in pixels, S is the stride and L is the number of allocated lines
- If enabled one raw TIFF from the RAW 2D extraction point `raw2dF_WxH-str_S-lines_L_tiffT.rgb` where F is the frame number, W the width in pixels, H the height in pixels, S the stride, L the number of allocated lines and T the bit-depth of the TIFF format (e.g. `raw2D30_960x540-str_1200-lines_540_tiff10.rgb`)

Example of *scenarii*

This paragraph list some of the *scenarii* that could be interesting to test using `driver_test`:

- data extraction (point 0) output, 1 frame
- YUV 420 encoder output, display output, 3 frames
- 2 Contexts, data extraction (point 1) output for both, 3 frames serial
- CT0 encoder output (420 down-scaled 1:2.0) without display, CT1 with data-extraction (point 0), DG0 for both context

5.5.2 ISP Control test: ISPC_test

ISPC_test is designed to provide the same flexibility as the **driver_test** application but using high level parameters that are used to configure the pipeline. It uses the ISPC library layer to configure the pipeline and can only run with data generators (external or internal).

This tool is useful to test slow frame-rate systems but provides limited support over the AAA algorithms.

The source is available in **ISPControl/test_apps/** and the binary should be installed in **ISPC/** after make install.

Parameters

The ISP Control test application uses similar parameters as the driver test and an additional Felix High level configuration file (usually named **FelixSetupArgs.txt**). The file contains information about the setup of every module in V2500 ISP. More information about this file can be found in the *High level parameters* (page 283) documentation.

The parameters can be given by command line or in a configuration file.

The parameter types are:

- *Bool*: a Boolean value of either 0 or 1
- *Int / UInt*: an integer (signed or unsigned is parameter specific)
- *Cmd*: command that act as a Boolean of value 1 if present, of value 0 if not found
- *Str*: string or path
- *Float*: floating point number

-source: Provide a file to parse for command line parameters. The latest given parameter is used as the value for the option.

-h: Display the help and exit.

Generic Parameters

These parameters affect the application as a whole.

Similar to insmod (fake driver only) These parameters all have **insmod** equivalent with the same name.

-mmucontrol: Int <0, 1, 2> Use the device MMU. If enabled Device Virtual Addresses are used.

- 0 – bypass MMU (virtual=physical)
- 1 enables MMU but does not enable extended address range (32b MMU).
- 2 – using extended address range (40b MMU)

Only affects runs with Fake interface.

Warning: The value 1 is supported by the IMG BUS4 MMU. However the V2500 HW was not tested with this option as CSIM does not support it, use at own risks.

-tilingScheme: **Int <256, 512>** Choose the Device MMU tiling scheme. Either 256 for 256x16 or 512 for 512x8.

Only affects runs with Fake interface with MMU enabled and may not be available on all HW versions.

-tilingStride: **Int** If non-0 used as common stride when tiled buffers are used. This has to be a power of 2 and has to be big enough for all enabled output (even non-tiled ones that could be tiled).

Only affects runs with Fake interface with MMU enabled and may not be available on all HW versions.

-gammaCurve: **Int** Change the default gamma curve used. More details about Gamma Curve selection available in the *Getting Started Guide* (page 17).

-extDGPLL: **UInt [x2]** When using the external data-generator can be used to change the speed of processing. Provide the multiplier and divider to the test IO bank (**not part of the ISP IP**) - both values need to be specified.

If 0 is given (default) then the data-generator driver computes the default PLL $\frac{\text{parallesim} \times 10}{\text{parallesim} \times 10}$

Warning: If incorrect values are provided the PLL may not lock up and the FPGA will need to be reprogrammed.

If given values are too high the test may not generate frames because the input rate will be too high for the ISP to handle.

-cpuPageSize: **UInt** CPU page size in kB. Interesting values are 4kB and 16kB.

Note: This is **not** an insmod parameter as it is defined by the OS.

-no_pdump: **Cmd** Does not generate pdumps when running.

Note: This is **not** an insmod parameter as real driver does not generate pdumps.

Global Parameters

-parallel: **Cmd** Use the parallel capture mode (see *Behaviour* (page 59))

-no_write: **Cmd** Do not write output to file (including statistics).

Is interesting when running on system with high memory latency.

-wait: **Cmd** Wait for a key-press at the end of execution. Useful when running with batch systems on windows.

-DumpConfigFiles: **Cmd** Output configuration files for default, minimum and maximum values for all High Level Parameters into:

- *FelixSetupArgs2_def.txt*
- *FelixSetupArgs2_min.txt*

- `FelixSetupArgs2_max.txt`
- `FelixSetupArgs_reST.txt`

-encOutCtx: UInt [x2] Which context to use for the Encoder pipeline statistics output: 1st value is context, second value is the pulse width in clock (see `FE-LIX_CORE:ENC_OUT_CTRL`).

In HW the direct output sends PARALLELISM signals every PARALLELISM lines.

-hw_info: Cmd Prints the HW information from the Connection object.

Note: When using CSIM a connection to the CSIM is needed to read the CORE registers.

-ignoreDPF: Bool Override DPF results in statistics and skip saving of the DPF output file.

Use this flag for verification of output between CSIM and HW as both may not produce the same DPF output.

Fake interface only These parameters are related to the CSIM or functionalities available to the fake driver only.

-simIP: Str Simulator IP address when using the Fake interface with TCP/IP.

Default is localhost.

-simPort: UInt Simulator port when using Fake interface with TCP/IP.

Default is 2345.

-simTransif: Str Path the simulator transfif library when using the Fake interface with transfif.
Also needs `-simconfig`.

If not specified TCP/IP fake interface is used.

-simConfig: Str Configuration file to give to the simulator when using the transfif Fake interface.

-pdumpEnableRDW: Bool Enable generation of RDW commands in pdump output (not useful for verification). Only affect Fake driver. Default is 0.

Context Parameters

The context parameters are defined per HW context. The X should be replaced by the context number starting from 0 (e.g. `-CT0_setupFile`).

Run options

-CTX_setupFile: Str Path to the `FelixSetupArgs.txt` to load

-CTX_imager: UInt Imager that the context should use.

-CTX_nBuffers: UInt Number of buffers to allocate for the run

-CTX_nEncTiled: UInt Number of buffers to allocate as tiled encoder output (maxed to the number of allocated buffers).

- CTX_nDispTiled: **UInt** Number of buffers to allocate as tiled encoder output (maxed to the number of allocated buffers).
- CTX_nHDRExtTiled: **UInt** Number of buffers to allocate as tiled encoder output (maxed to the number of allocated buffers).
- CTX_nFrames: **UInt** Number of frames to run for on this context
- CTX_re-use: **Bool** Enable the re-use of the previously used Camera object when running in serial mode (the previously used context needs to be configured to use the same DG as the new one!)
- CTX_updateMaxSize: **Bool** Update output sizes using ISPC::Camera::setEncoderDimensions() and ISPC::Camera::setDisplayDimensions() using output size after the scalers. In other words will allocate the minimum size buffers. Default is off and the allocation will be big enough to cover the whole input image resolution.

Additional input (V2500 v2 only) These inputs may not be available for all HW configurations. If the HW does not support them these options will be ignored.

- CTX_HDRInsertion: **Str** The filename to use as the input of the HDR Insertion if enabled in given setup file - if none do not use HDR Insertion. The image has to be an FLX in RGB of the same size than the one provided to DG but does not have to be of the correct bitdepth (max supported bitdepth is 16b per channel).

This will replace the image in the pipeline and Encoder and Display output will use that image as source.

Warning: The setup parameter *OUT_DI_HDF* has to be set to enable HDR insertion!

- CTX_HDRInsRescale: **Bool** Rescale provided HDR Insertion file to 16b per pixels. Does not downscale.

Configuration options

- CTX_saveStats: **Bool** Save statistics generated by HW in the same way than `driver_test` does (default is 1 – disabled if `-no_write` enabled).

See [Statistics Format](#) (page 43).

- CTX_saveTxtStats: **Bool** Save statistics generated by HW in text format. This will exclude CRCs but contain the module configuration when available (default is 1 - disabled if `-no_write` enabled).

See [Statistics Format](#) (page 43).

- CTX_LSHGrid: **Str** Path to the LSH grid file to load. Overrides the one provided in setup file.

- CTX_changeLSHPerFrame: **Bool** If more than 1 LSH grid is loaded will try to change the grid based on the frame number. This may be disabled if using some control algorithms that provide this functionality as well.

This allow the testing of several matrices but because the capture is not restarted they have to share the same configuration (tile size, bits per diff, etc). To use matrices with different configuration several run of ISPC _ test should be made.

This should be used with only 1 buffer or not all matrices will be used.

-CTX_saveRaw: Bool Enable saving of the raw image data for images output (except YUV which is always RAW image from HW). Stride is removed.

-CTX_saveLSH: Bool Save the LSH matrices loaded in the test folder. If disabled still save the current matrixId used (default is enabled). The matrixId associated to the loaded file can be found in the log output.

-CTX_regOverride: Bool Allow register override if a file is present (see *Register override* (page 41)).

Note: Available only with Fake Interface.

Warning: This option should be avoided as it requires a complete knowledge about the registers to choose the correct value and does not work with real drivers.

-CTX_enableTS: Bool: Enable timestamps generation. By default with real driver timestamps are enabled but they can be disabled when doing HW output verification against results generated with CSIM (e.g. statistics may not be binary equivalent as they contain timestamp).

Note: No effect when running Fake driver: CSIM does not generate timestamps therefore the driver does not enable them.

Algorithms controls These parameters give some input over the ISPC Control algorithms.

Information about algorithms is available in the *Implemented Control Modules* (page 245) document.

-CTX_TNM: Bool Enable usage Global Tone Mapper control algorithm.

-CTX_TNM-local: Bool Enable usage Local Tone Mapper control algorithm.

-CTX_TNM-adapt: Bool Enable usage Adaptive Tone Mapper control algorithm.

-CTX_LBC: Bool Enable usage Light Based Controls algorithm (and loading of the LBC parameters).

-CTX_WBC: Int Enable usage of White Balance Control algorithm; values are:

- 0 disabled
- 1 AC
- 2 WP
- 3 HLW
- 4 COMBINED
- 5 Planckian Locus

Please note that Planckian Locus method is available for HW versions starting from 2.6, where it's the default value, otherwise Combined statistics is chosen.

-CTX_AE: Bool Not available! Data Generator does not support different exposures/gains.

-CTX_AF: Bool Not available! Data Generator does not support different focus.

Internal Data Generator parameters

The same parameters than *driver_test* (page 57) are used for the internal data-generator.

-IntDGX_inputFLX: Str Path to the input FLX file in RGGB.

-IntDGX_inputFrames: UInt (>0) Number of frames to use from the input file.

E.g. To load from a file with 10 frames but limit the used input to 4 first.

-IntDGX_gasket: UInt Gasket to replace data from with internal data generator.

Related to the **-CTX_imager** option used.

-IntDGX_blanking: UInt [x2] Horizontal and vertical blanking to use. If 2nd value (vertical blanking) is not specified default is used.

-IntDGX_reload: Bool Enable the frame preloading of the internal data-generator (HW feature).

External Data Generator parameters

The same parameters than *driver_test* (page 57) are used for the external data-generator.

Warning: The external data-generator is not available on silicon but only when using CSIM.

-DGX_inputFLX: Str Path to the input FLX file in RGGB.

-DGX_inputFrames: UInt (>0) Number of frames to use from the input file.

E.g. To load from a file with 10 frames but limit the used input to 4 first.

-DGX_blanking: UInt [x2] Horizontal and vertical blanking to use. If 2nd value (vertical blanking) is not specified default is used.

-DGX_reload: Bool Enable the frame preloading of the data-generator (HW feature).

-DGX_MIPILF: Bool If the associated gasket is MIPI will use the long format (additional line header).

-DGX_mipiLanes: UInt <1, 2, 4> If the associated gasket is MIPI will set the number of MIPI lanes to use. Valid number of lanes are 1, 2 and 4.

Behaviour

The behaviour of this test application is similar to the one explained for the *driver_test* (page 59) application. The only difference is that the configuration of the pipeline is done by loading the *FelixSetupArgs.txt* and configuring the HW through the ISP Control layer. It has the same option for parallel or sequential running of multi-context tests.

Parameter Example

This section provides a basic test which duplicates the behaviour of the one given in *driver_test* (page 58).

It is possible to run it using

```
$ ./ISPC_test -source ispc_params.txt
```

And in the same way that it is possible to modify parameters in *driver_test* additional values on the command line will modify the file that was used as a reference, for example using another image for DG0:

```
$ ./ISPC_test -source ispc_params.txt -DG0_inputFLX ~felix/FLX/DE_A_rggb.flx
```

ispc_params.txt:

```
#-----
# global setup
#-----
-mmcontrol 2 # 0 for no MMU, 2 for MMU 40b
-parallel 0   # run in serial mode to be able to do re-use

#-----
# pipeline setup
#-----
-CT0_setupFile FelixSetupArgs_CTO.txt # configuration to load for this context
-CT0_nBuffers 5 # nb of buffers to allocate
-CT0_nFrames 10 # nb of frames to run for
-CT0_imager 0   # using DG 0
-CT0_WBC 4      # use combined White Balance controls - use 5 for Plankian
-CT0_LBC 0      # do not enable Light Based controls
-CT0_TNM-adapt 1 # enable adaptive tone mapper curve

-CT1_setupFile FelixSetupArgs_CT1.txt
-CT1_nBuffers 1
-CT1_nFrames 10
-CT1_imager 0
-CT1_re-use 1    # run with the same object from CT0 - possible because same DG is used

#-----
# data generators
#-----
-DG0_inputFLX fight_960x540-12b-91.flx # this file has 91 frames
-DG0_inputFrames 10 # but load only 10 frames out of the 91

-DG1_inputFLX DE_A_rggb.flx # not used for that test
-DG1_inputFrames 1 # number of frames to load from input file
```

FelixSetupArgs_CTO.txt:

```
// Global output parameters
OUT_DE BAYER12
OUT_DE_POINT 1
OUT_DISP NONE
OUT_ENC NV21

// WB white balance multiple CCM
```

```

WB_CCM_0 1.1981 -0.0346 -0.2413 -0.3086 1.1716 0.0749 -0.0681 -0.999 1.98
WB_CCM_1 1.0463 0.0366 -0.1541 -0.3571 1.2661 0.0371 -0.1288 -0.6301 1.6843
WB_CCM_2 1.0017 0.0818 -0.1479 -0.2517 1.1521 0.0406 -0.0583 -0.9271 1.9070
WB_CCM_3 1.2889 -0.2184 -0.1441 -0.2686 1.2324 -0.0183 -0.0131 -0.6337 1.5793
WB_CCM_4 1.2939 -0.2349 -0.1441 -0.2410 1.2197 -0.0399 0.0183 -0.5901 1.5035
WB_CORRECTIONS 5
WB_GAINS_0 1 1.21216 1.19692 3.22325
WB_GAINS_1 1 1.10695 1.09254 2.7292
WB_GAINS_2 1.2724 1.01386 1 3.05519
WB_GAINS_3 1.21401 1.01559 1 1.88554
WB_GAINS_4 1.37693 1.01983 1 1.57071
WB_OFFSETS_0 -115.957 -118.808 -117.855
WB_OFFSETS_1 -116.65 -119.668 -120.553
WB_OFFSETS_2 -118.114 -118.879 -119.727
WB_OFFSETS_3 -117.457 -119.671 -119.148
WB_OFFSETS_4 -114.749 -118.905 -119.947
WB_TEMPERATURE_0 2800
WB_TEMPERATURE_1 3300
WB_TEMPERATURE_2 4000
WB_TEMPERATURE_3 5000
WB_TEMPERATURE_4 6500

```

FelixSetupArgs_CT1.txt:

```

// Global output parameters
OUT_DE NONE
OUT_DE_POINT 1
OUT_DISP RGB888_32
OUT_ENC NONE

```

Expected Output

All outputs are written in the directory where to application running.

According to the parameters provided some image files are generated from the test application (where X is the context number):

- One encoder output file per context `encoderX_WxH_FMT_alignA.yuv` where W is the width in pixels, H the height in pixels, FMT the format string and A the alignment of the allocation in Bytes (e.g. `encoder0-1020x720-NV21-align1.yuv`).
- One display or one data-extraction file per context `displayX.flx` or `dataExtractionX.flx`.
- One HDR extraction file `hdrExtractionX.flx` (V2500 v2 only).
- One Raw2D extraction file `raw2DExtractionX.flx` (V2500 v2 only).

Statistics files can be generated too:

- If enabled one binary statistics file per context `ctx_X_stats.dat` (see *Statistics Format* (page 43)).
- If enabled one DPF output file `ctxX-F-dpf_write_S.dat` where F is the frame and S the number of defects (see *Defective Pixel output map* (page 44)).
- If enabled one statistics file per context and frame `ctxX-F-statistics.txt` (see *Statistics Format* (page 43)).

- If enabled the loaded lsh are saved as `ctxX-lshgridY.lsh` where Y is the matrixId (see [Deshading Grid file format \(LSH\)](#) (page 44)).

If the save RAW option is enabled some additional image files can be saved:

- If the encoder output buffer is tiled an additional file is saved `encoderX_WxH_tiled_strS_FMT-alignA.yuv` (with S the tiled buffer stride).
- `dataExtractionX_F_WxH_strS_FMT.flx` or
- `displayX_F_WxH_strS_FMT.flx` or `displayX_F_WxH_tiled_strS_FMT.flx` if the buffer is tiled
- `hdrExtractionX_F_WxH_strS_FMT.flx` or `hdrExtractionX_F_WxH_tiled_strS_FMT.flx` if the buffer is tiled
- `raw2dExtractionX_F_WxH_strS_FMT.flx` or `raw2dExtractionX_F_WxH_tiled_strS_FMT.flx` if the buffer is tiled

See the [driver test format string](#) (page 42) for naming convention.

5.5.3 ISPC loop application: ISPC_loop

The loop application is designed to run basic ISPC setup, like ISP Control test, continuously against a real sensor or a data-generator. It was tested on our FPGA platform and incorporates some AAA support.

This tool can use a data generator but does not have to. However it cannot run against the simulator.

Sources are available in `ISPCControl/test_apps` and it is installed under `ISPC/`.

Note: If pressing the key does not work on your platform try to use the [Console Raw Mode testing](#) (page 86) application and modify it to make it work there.

Parameters

The parameters can be given by command line or in a configuration file.

The parameter types are:

- *Bool*: a Boolean value of either 0 or 1
- *Int/UInt*: an integer (signed or unsigned is parameter specific)
- *Cmd*: command that act as a Boolean of value 1 if present, of value 0 if not found
- *Str*: string or path
- *Float*: floating point number

-source: Provide a file to parse for command line parameters. The latest given parameter is used as the value for the option.

-h: Display the help and exit.

Sensor control

-sensor: **Str** Sensor to use. Available sensors are:

- “External Datagen”
- “IIF Datagen”
- AR330
- P401
- OV4688

Obviously this needs the actual sensor to be present. The *Platform Integration Guide* (page 27) has more details about adding support for more sensors.

If using either data-generator the **-DG_inputFLX** should specified the source file.

-sensorMode: **UInt** The mode to run for the selected sensor. Each sensor can have several modes (e.g. to have different resolution or speed). Refer to the Sensor API documentation for more details.

Default 0.

-sensorExposure: **UInt** Initial exposure (μ s) to configure the sensor with (optional).

If auto expose is not disabled the exposure value will change while running.

-sensorGain: **Float** Initial gain to configure the sensor with (optional).

If auto exposure is not disabled the gain value will change while running.

-sensorFlip: **Bool [x2]** Flip the sensor horizontally and vertically. Not all sensor and modes support flipping.

If only 1 value is given it is assumed to be the horizontal flipping.

Data Generator Parameters If using either data generator these parameters can be used.

The internal or external data-generator choice is made with the **-sensor** value.

-DG_inputFLX: **Str** If using either Datagen sensors it is the path to the FLX input file to use. Similar to **-DGX_inputFLX** and **-IntDGX_inputFLX** in *ISPC_test* (page 61).

-DG_gasket: **Str** Gasket to use with the data-generator. Similar to **-IntDGX_gasket** or DG number in *ISPC_test* (page 61).

-DG_blank: **UInt [x2]** Horizontal and vertical blanking in pixels for the data-generator. Similar to **-DGX_blank** or **-IntDGX_blank** in *ISPC_test* (page 61).

If only 1 value is specified it is assumed to be the horizontal blanking.

V2500 Control

-setupFile: **Str** Path to the Felix Setup Args file to load. Similar to **-CTX_setupFile** in *ISPC_test* (page 61).

-delay: **UInt** Fixed length delay to be inserted between calls to enqueue buffers, measured in microseconds.

- randomDelayLow:** **UInt** Lower bound of random delay to be inserted between calls to enqueue buffers, measured in microseconds.
- randomDelayHigh:** **UInt** Upper bound of random delay to be inserted between calls to enqueue buffers, measured in microseconds.
- updateASAP:** **Cmd** Enable the update ASAP in the ISPC::Camera. Can reduce latency when using a lot of buffers but will also not provide a reliable way of knowing when parameters were applied.

Control Modules

- disableAE:** **Cmd** Does not create the Auto Exposure control object when creating the Camera. This functionality will not be available at all for this run.
- disableAWB:** **Cmd** Does not create the Auto White Balance control object when creating the Camera. This functionality will not be available at all for this run.
- chooseAWB:** **UInt** Choose the White Balance Control algorithm; values have to be a legitimate ISPC::Correction_Types enum:
 - 0 disabled
 - 1 AC
 - 2 WP
 - 3 HLW
 - 4 COMBINED
 - 5 Planckian Locus

Please note that Planckian Locus method is available for HW versions starting from 2.6, where it's the default value, otherwise Combined statistics is chosen.

Similar to **-CTX_WBC** in *ISPC_test* (page 61).

- useWBTS:** **Bool** Enable White Balance Temporal Smoothing algorithm;
 - 0 disabled
 - 1 enabled
- Default is disabled.
- wbtsTemporalStretch:Int** Time to settle awb after change (milliseconds). Value outside range will be clipped.
- min = 200 max = 5000
- Default 300
- wbtsWeightBase:Float** Set the base for temporal smoothing weights generation. Lower base - output values closer to average of n previous samples. Higher base - output values closer to non smoothed (calculated for current) Value outside range will be clipped.
- min = 1.00 max = 10.00
- Default 2.00

-featuresWBTS: **UInt** Choose the White Balance Temporal Smoothing features; values have to be a legitimate `ISPC::Smoothing_Features` enum:

- 0 disabled
- 1 FF (Flash Filtering)

Default is disabled.

-disableAF: **Cmd** Does not create the Auto Focus control object when creating the Camera. This functionality will not be available at all for this run.

-disableTNMC: **Cmd** Does not create the Tone Mapper control object when creating the Camera. This functionality will not be available at all for this run.

-disableLBC: **Cmd** Does not create the Light Based Control object when creating the Camera. This functionality will not be available at all for this run.

-disableDNS: **Cmd** Does not create the Denoise Control object when creating the Camera. This functionality will not be available at all for this run.

-disableAutoDisplay: **Cmd** Disable the always enabling of display output (unless taking DE).

Performance measurement

If the ISPC library was compiled with the performance measurement logs (see *Performance measurement* (page 285)) the following command line parameters are available:

-monitorXXX: **Cmd** Monitor and log efficiency of XXX control algorithm. Number of frames to converge is provided in logs marked with PERF. Replace XXX with AWB, AF, or AE. E.g. -monitorAWB.

-monitorAAA: **Cmd** Monitor and log efficiency of all AAA algorithms (AWB, AF, and AE).

-measureXXX: **Cmd** Measure performance of control module. Replace XXX with 2/3 letter acronym of the module/control.

E.g. -measureAF, -measureDNS.

Average time is also calculated and logged.

-measureAAA: **Cmd** Measure performance of all controls i.e. algorithms AWB, AF, and AE.

-measureMODULES: **Cmd** Measure performance of all modules. Only those equipped with performance measurement markers will be measured.

-measureVERBOSE: **Cmd** Verbose all measurement info. With measureVERBOSE defined each performance log will verbose the about current call.

Image buffer management

-nBuffers: **UInt** Number of pre-allocated buffers to run with. Also used as the number of data-generator buffers. Similar to `-CTX_nBuffers` in *ISPC_test* (page 61).

Default is 2.

-importBuffers: **Cmd** Allocate and import image buffers from user space instead of doing internal allocation in kernel space. This option is enabled only if `ISPC_loop` has been configured with cmake variable (see *IMGVideo Support* (page 19)):

```
-DISPC_LOOP_USES_DMABUF=TRUE
```

Note: Current implementation supports buffer allocations using `dmabuf` userspace library dependent on `dmabuf_exporter.ko` kernel module, and only if Felix device driver has been configured properly with *IMGVideo Support* (page 19) enabled.

Behaviour

The tool was designed to demonstrate V2500 possibilities on IMG FPGA platform. It will force the output to be RGB32 8b (to use IMG FPGA PDP output) but run-time key press can enable saving of other formats. This default can be altered by running with `-disableAutoDisplay`.

De-shading grid limitation

If a LSH grid is provided into the given setup file it has to have a global path or be a relative to where the tool is running.

Runtime key press

The following keys can be pressed while the tool is running.

Control Modules Keybinding that enable/disable control algorithms.

Key	Behaviour
W	Toggle AWB on/off
E	Toggle Defective Pixel correction
R	Toggle usage of LSH matrix if available
T	Toggle adaptive TNMC on/off (needs TNMC on)
Y	Toggle local TNMC on/off (needs TNMC on)
U	Toggle TNMC on/off
I	Toggle DNS gain update from Sensor's gain
O	Toggle LBC on/off
+/-	<p>Change target brightness for AE.</p> <p>Change current exposure if AE is disabled.</p>
P	<p>Loop through flicker rejection modes used in AE</p> <p>Available modes are : OFF, AUTO, 50Hz, 60Hz</p>
,/.	Focus close/further if available
M	Trigger focus sweep
0	Force CCM to identity matrix if AWB is disabled
1-9	Force one of the loaded multi-CCM if AWB is disabled

Other controls Keybinding that allow other operations such as saving

Key	Behaviour
X	Exit the application
A	<p>Save all file formats as enabled in given setup file.</p> <p>The same limitations apply to the selected output than for a normal run (i.e. RGB and Bayer are mutually exclusive).</p>
S	Save RGB output as RGB FLX file
D	Save YUV output as YUV file
F	Save DE output as RGGB FLX file
G	Save Raw 2D Extraction output if available in HW in FLX file
H	Save HDR Extraction output if available in HW in FLX file

Parameter Example

Running the loop is similar to running any other test application:

```
$ ./ISPC_loop -sensor OV4688 -setupFile ISPC2_OV4688.txt -sensorFlip 1 1
```

Equivalent to

```
$ echo "-sensor OV4688" > loop_params.txt
$ echo "-setupFile ISPC2_OV4688.txt" >> loop_params.txt
$ echo "-sensorFlip 1 1" >> loop_params.txt
$
$ ./ISPC_loop -source loop_params.txt
```

Warning: When using the `-sensor` with a name containing space (such as "IIF Datagen" or "External Datagen") the parameter cannot be part of the sourced file (parsing of the file cuts the space and only "IIF" is considered).

Using internal Data-generator

This example is to run the loop using a previously captured image using our internal data-generator.

```
$ ./ISPC_loop -source loop_dgparams.txt -sensor "IIF Datagen"
```

`loop_dgparams.txt`:

```
-DG_inputFLX /home/felix/FLX/DE_A_rggb.flx # input FLX needs to be RGGB
-DG_blanking 1080 720 # optional, blanking to limit input throughput to ISP

# ISP settings
-setupFile /home/felix/config/ISPC2_AR330.txt
```

Expected Output

When saving a frame, regardless of the format, the following files will be available (X being the frame number):

- `FelixSetupArgs_X.txt` containing the setup used to capture this output
- `statisticsX.dat` containing the statistics (save_structure) following the *Statistics Format* (page 43)
- `encoder_statsX.dat` containing the ENS output if enabled
- `defective_pixelsX.dat` containing the DPF output if enabled

Depending on which key is pressed some other files may be saved:

- `encoderX-widthxheight-format-alignA.yuv` for the YUV output
- `displayX.flx` for the RGB output
- `dataExtractionX.flx` for the Bayer output
- `hdrExtractionX.flx` for the HDR output

- `raw2DExtractionX.flx` for the Raw 2D output

5.5.4 ISP Control video capture: ISPC_capture

This test application was designed to capture RGGB videos using Imagination's FPGA system, the limitation being the small throughput when accessing the device memory (limited to ~4MB/s). This application overcomes this problem by allocating 1 buffer per required frame and not attempting to access the memory until all the frames are captured.

The number of shots used to feed the HW is set to 2 by default but can be configured.

Obviously this application needs to use a real sensor.

The sources are available in `ISPControl/capture` and the executable is installed in `ISPC/`.

Parameters

The parameters can be given by command line or in a configuration file.

The parameter types are:

- *Bool*: a Boolean value of either 0 or 1
- *Int/UInt*: an integer (signed or unsigned is parameter specific)
- *Cmd*: command that act as a Boolean of value 1 if present, of value 0 if not found
- *Str*: string or path
- *Float*: floating point number

-source: Provide a file to parse for command line parameters. The latest given parameter is used as the value for the option.

-h: Display the help and exit.

Required Parameters

-sensor: **Str** Sensor name to use (one of the supported ones in sensor API).

-sensorMode: **UInt** Sensor running mode (chooses size and FPS for a given sensor).

-nFrames: **UInt** Number of frames to run for. The limit is the amount of memory available to the device.

-nBuffers: **UInt** Number of buffers to be pushed into the HW linked list. Minimum of 1, maximum of 16.

Optional Parameters

-sensorExposure: **UInt** Forces the sensor exposure, unit is μ s.

If Auto Exposure is enabled used as initial exposure.

-sensorGain: **Float** Forces the sensor gain.

If Auto Exposure is enabled used as initial gain.

-sensorFlip: Bool [x2] Flip the sensor horizontally and vertically. Not all sensor and modes support that.

If only 1 value is given it is assumed to be the horizontal flipping.

-DEBitDepth: UInt Data extraction bit depth to use (8b, 10b or 12b).

Default is 10b.

-output: Str Name of the output file.

Default is dataExtraction.flx

-concurrent: Cmd Save the frames while they are captured with double buffering.

Warning: This may be slow if memory latency is high (can be used to test it).

-enableAE: Float Enable the auto exposure algorithm and sets its target brightness.

Behaviour

A Camera object will be configured to extract RGGB data from the 1st Data Extraction point and the given number of frames will be allocated. If the system does not have enough memory the application will fail at that point.

For example Imagination's FPGA system uses 256MB of PCI carved-out memory. This amount of memory allows us to capture around 190 720p frames at once (258.375.680 Bytes are used). On our particular FPGA system where the memory access is slow the capture took a couple dozen seconds while the saving took a couple hundred.

The amount of buffers and shots allocated is always the number of frames as they are accumulated until the end (unless using concurrent mode). However the number of shots pushed to the HW linked list is configurable.

If the concurrent mode is used it will simply buffer multiple times in an attempt to capture those frames. This may allow more frames to be captured but could result in slower frame rate if the memory latency is high.

Expected Output

The output file should be an FLX file with either a default file name or the one given. If auto exposure was enabled it will also save `sensor_info.txt` with the exposure/gain used per frame.

5.5.5 ISP Control HDR: ISPC_hdr

ISPC_hdr is designed to provide an initial test platform for the sequential HDR algorithm. It uses a CPU version of the HDR merging, called HDRLibs, which is **not** provided to customers. However customers can implement their own merging function and use it in this application (see [Implement another HDR merge](#) (page 80)).

This tool can be used only with a real sensor and a HW that supports HDR extraction and insertion.

The source code is available in `ISPControl/ISPC_hdr/` and is installed in `ISPC/` after make install.

Parameters

The parameters can be given by command line or in a configuration file.

The parameter types are:

- *Bool*: a Boolean value of either 0 or 1
- *Int / UInt*: an integer (signed or unsigned is parameter specific)
- *Cmd*: command that act as a Boolean of value 1 if present, of value 0 if not found
- *Str*: string or path
- *Float*: floating point number

-source: Provide a file to parse for command line parameters. The latest given parameter is used as the value for the option.

-h: Display the help and exit.

Sensor Parameters

These parameters affect the selection of the sensor mode and initial state.

-sensor: **Str** Sensor to use. Available sensors are:

- AR330
- P401
- OV4688

Obviously this needs the actual sensor to be present. The *Platform Integration Guide* (page 27) has more details about adding support for more sensors.

-sensorMode: **UInt** [optional] The mode to run for the selected sensor. Each sensor can have several modes (e.g. to have different resolution or speed). Refer to the Sensor API documentation for more details.

-sensorFlip: **Bool [x2]** [optional] Flip the sensor horizontally and vertically. Not all sensor and modes support flipping.

-sensorFocus: **UInt** [optional] Initial focus position of the sensor in **mm**. May not work on all sensors. Will change if Auto Focus (AF) is enabled.

Capture parameters

-HDR0_exposure: **UInt** Value in **μs** for 1st frame.

[optional] If using AE as offset of computed exposure.

[mandatory] If not using AE actual exposure value.

-HDR0_gain: **Float** [optional] Gain values for the captures.

If using AE overrides the computed gain.

If not using AE initial gain for the capture.

-HDR1_ratio: **Int** Ratio divider to compute exposure of the second frame such as $E1 = \frac{E0}{ratio}$.

- setupFile: String FelixSetupArgs file to use to configure the Pipeline.
- saveIntermediates: Bool If enable will save the intermediate images during the processing:
 - HDR extraction 1st frame as frame0_‘exposure’_x‘gain’.flx
 - HDR extraction 2nd frame as frame1_‘exposure’_x‘gain’.flx
 - result of the frame merging in hdr_input.flx

Algorithm parameters

These parameters affects which algorithms are used before the capture of the HDR extraction images.

- AE: Bool [optional] Enable the Auto Exposure (AE) algorithm. Can be ignored if -HDR0_exposure is provided.
- AWB: Bool [optional] Enable Auto White Balance (AWB) algorithm. Can be ignored if provided setup file contains correct CCM or -AWB_temperature is provided instead.
- AWB_temperature: Float [optional] if AWB is not enabled loads the correction for specified temperature into the CCM and WBC modules.
- AF: Bool [optional] Enable Auto Focus (AF) algorithm if the sensor supports it. Can be ignored if -sensorFocus is provided.
- settleFrames: UInt [optional] Number of frames for the AE, AWB and AF to settle before the capture is done. Used only if one of the algorithm is enabled. Default is 30.
- nShots: UInt [optional] Number of shots to capture for each frame. Should be at least 1, may need to be more to allow exposure/gain to settle.

Behaviour

The ISPC_hdr application goes through several stages:

- running the algorithms if any are enabled (optional).
- running the capture of the 2 HDR extraction
- merging the 2 HDR images together
- running the merged image as HDR input

Running the algorithms

It is possible to run the algorithms before the capture of HDR images to simulate the preview a user would normally do before pressing the trigger. However it is possible to skip this step by providing the correct information for each algorithm:

Auto Exposure: Provide an initial exposure and gain to avoid using auto-exposure

Auto White Balance: Either provide a configuration file with the desired values for the CCM/WBC modules

or provide a setup file with multiple CCM and provide a temperature to extract the correction from them.

Auto Focus: Provide the desired focus position.

Auto Tone Mapper: Is not run during the initial capture but the result of the auto tone mapper can be provided as part of a setup file

Note: It is a valid test scenario to run the ISPC_loop or ISPC_tcp/VisionLive as a first step to gather the algorithms values before running the ISPC_hdr. Be sure to provide an output setup file (for example save an image with ISPC_loop).

Running the capture

The capture of the 2 HDR extraction frames is done over several frames because the exposure and gain may take several frames to apply on the sensor. Refine the given number to match what the selected sensor implementation provides.

Note: A bigger number may increase the ghosts on the image (i.e. moving elements).

Merging images

The merging of the 2 frames is done with HDR libraries if available. Those libraries are not provided to customers as they are internal research libraries used to specify the GPU merging algorithms. The customer can however implement a merging algorithm to be able to use the ISPC_hdr tool.

Implement another HDR merge The provided HDR frame merge is available in the ispc2test library. The default one is a simple average merge (does average of 2 frames).

Note: It is important to understand the HW formats to be able to do an efficient HDR merge. Check the HW TRM for packed details of the formats.

It is possible for the customer to implement another function to merge two frames following the scheme of the one provided, here is a pseudo-code version

```
IMG_RESULT AverageHDRMerge(const ISPC::Buffer &frame0,
const ISPC::Buffer &frame1, CI_BUFFER *pHDRBuffer) {

    unsigned i = 0, j = 0;
    unsigned inStride = 0, outStride = 0;

    // insertion buffer bitdepth = 16b - HDR output is 10b
    // merging the two frame0 and frame1 needs a rescaling of the values
    const int input_pixel_bit = 10;
    const int input_mask = 0x3FF;
    const int r = 2, g = 1, b = 0; // channel order
    const int shift = 16 - input_pixel_bit;
    const int mask = (1 << (shift + 1))-1;

    // do verifications here, correct sizes and correct formats

    outStride = sizeInfo.ui32Stride / sizeof(IMG_UINT16);
```

```

inStride = frame0.stride/sizeof(IMG_UINT32);

const IMG_UINT32 *pFrame0 = (IMG_UINT32*)frame0.data;
const IMG_UINT32 *pFrame1 = (IMG_UINT32*)frame1.data;
const IMG_UINT32 maxPix = (1<<16) -1;

for (j = 0; j < frame0.height; j++)
{
    for (i = 0; i < frame0.width; i++)
    {
        int out = j*outStride + i*4,
            in = j*inStride + i;

        // low exposure
        int r0 = (pFrame0[in]>>(r*input_pixel_bit))&input_mask,
            g0 = (pFrame0[in]>>(g*input_pixel_bit))&input_mask,
            b0 = (pFrame0[in]>>(b*input_pixel_bit))&input_mask;

        // high exposure
        int r1 = (pFrame1[in]>>(r*input_pixel_bit))&input_mask,
            g1 = (pFrame1[in]>>(g*input_pixel_bit))&input_mask,
            b1 = (pFrame1[in]>>(b*input_pixel_bit))&input_mask;

        // red
        IMG_UINT32 pix = (r0+r1)/2;
        pix = pix<<shift |
              (pix>>(input_pixel_bit - shift))&mask;
        pBuffData[out + r] = IMG_MIN_INT(pix, maxPix);

        // green
        pix = (g0+g1)/2;
        pix = pix<<shift |
              (pix>>(input_pixel_bit - shift))&mask;
        pBuffData[out + g] = IMG_MIN_INT(pix, maxPix);

        // blue
        pix = (b0+b1)/2;
        pix = pix<<shift |
              (pix>>(input_pixel_bit - shift))&mask;
        pBuffData[out + b] = IMG_MIN_INT(pix, maxPix);

        pBuffData[out + 3] = 0; // blank space
    }
}
}

```

Inserting the merged result

Once merged the result is inserted in the pipeline again. The same setup file is used to program the Pipeline as the one that was given for the extraction of frames.

Parameter Example

For example here is how to run with the AR330 sensor and $\frac{1}{8}$ exposure ratio, the application will display similar information as the one presented here:

```
$ ./ISPC_hdr -sensor AR330 -AE 1 -HDR1_ratio 8 \
-setupFile ~/configs/ISPC2_AR330_outputs.txt
(...)

INFO [ISPC_hdr]: runExtraction() Running algorithms for 30 frames
INFO [ISPC_hdr]: runExtraction() AE provided 53616us x2.696173
INFO [ISPC_hdr]: runExtraction() Frame0: 53616us x2.696173 - Frame1
6702us 2.696173 (ratio=1/8) - 3 shots each
```

As *-saveIntermediates* does not specify the output format configured in the file given *-setupFile* will be present: *hdr_encoder0_1280x720_NV21_align1.yuv*

Expected Output

Any output format after the HDF block that is enabled in the given *-setupFile* will produce an output file in the last run. The acquisition step will not produce any output files.

- *hdr_display.flx* if display output is enabled
- *hdr_encoder0_WxH_F_alignA.yuv* if the encoder output is enabled (Width, Height, Format, Alignment in bytes).

Any other enabled output is disabled at configuration time and tiling is not available.

If *-saveIntermediates* is enabled then additional files will be present so that the merging can be done by another program or the HDR input reproduced:

- *frameX_Eus_xG.flx* as the extracted HDR images (X frame, Exposure in μ s, Gain multiplier)
- *hdr_input.flx* as the result of the merge

5.5.6 Defective Pixel output converter: dpf_conv

The DPF converter application can be used to extract some information of DPF write map (HW format) to a DPF read map (HW format). Both HW formats are described in the TRM and are loaded or written as binary.

The sources are located into `CI/felix_lib/utils` and the binary should be installed in `utils/`.

To run the tool a DPF write map should be generated using either the *Capture Interface test: driver_test* (page 49) or the *ISP Control test: ISPC_test* (page 61) applications.

Parameters

The parameters can be given by command line or in a configuration file.

The parameter types are:

- *Bool*: a Boolean value of either 0 or 1
- *Int/UInt*: an integer (signed or unsigned is parameter specific)

- *Cmd*: command that act as a Boolean of value 1 if present, of value 0 if not found
- *Str*: string or path
- *Float*: floating point number

-source: Provide a file to parse for command line parameters. The latest given parameter is used as the value for the option.

-h: Display the help and exit.

-input: **Str** Input DPF map (write map HW format).

-output: **Str** Output DPF map (read map HW format). If no output is given a file name `dpf_read_N.dat`. The number of corrected elements is always appended at the end (`_N.dat`).

-falsePositive: **Bool** If false no false positive are used in the output map.

-falseNegative: **Bool** If false no false negative are used in the output map.

-confidence: **UInt [x2]** The minimum and maximum confidence to use in the output map (allows filtering of the result).

-analyse: **UInt [x2]** Prints analyses of the generated output map (DPF read format). Needs the imager height (1st parameter) and the Felix Parallelism (2nd parameter).

If no parallelism is given 1 is assumed.

-info: **Cmd** Information about the DPF formats.

-v: **Cmd** Verbose information about the dropped values (when filtering false positive/negative or by confidence).

Generating DPF write map

The HW generates a DPF write map according to the defective pixels found. The threshold available in the setup control determines whether a pixel is defective or not (register `DPF_SENSITIVITY` or setup parameter `DPF_THRESHOLD` and `DPF_WEIGHT`).

Using `driver_test`

Using the information provided in the *driver_test parameters section* (page 49) we can create a configuration file to run DPF extraction

```
$ ./driver_test -source driver_params.txt
```

`driver_params.txt`:

```
#-----
# Pipeline
#-----
-CT0_nBuffers 1 # allocates 1 buffers
-CT0_nFrames 1 # run for 1 frames
-CT0_imager 0 # using DG0
-CT0_DPF 2 # 0=disabled, 1=detect, 2=write
#-----
```

```
# data generators
#-----
-DG0_inputFLX DE_A_rggb.flx # taken with AR330 camera
```

The **driver_test** application does not have any way to set the thresholds but the default should be set to the maximum. At the time of writing, running the test generates a DPF write file containing more than 9000 defects.

Note: The DE_A_rggb.flx file should be provided in the CSIM evaluation package examples. If using a different file the number of defects may vary.

Using ISPC_test

The **ISPC_test** application is closer to reality. To enable the DPF map the Setup file should only contain the following parameter:

```
$ ./ISPC_test -source ispc_params.txt
```

FelixSetupArgs.txt:

```
// all DPF parameters
DPF_DETECT_ENABLE      0 // will be forced on if DPF_WRITE_MAP_ENABLE is on
DPF_READ_MAP_ENABLE    0 // enable loading a DPF read map
DPF_WRITE_MAP_ENABLE   1 // will force DPF_DETECT_ENABLE to on if on
//DPF_READ_MAP_FILE // provide an input DPF read file
DPF_THRESHOLD          0
DPF_WEIGHT              16
```

ispc_params.txt:

```
-CT0_setupFile FelixSetupArgs.txt # or use FelixSetupArgs_full.txt
-CT0_nBuffers 1
-CT0_nFrames 1
-CT0_imager 0

-DG0_inputFLX DE_A_rggb.flx # taken with AR330 camera
```

At the time of writing running with these parameters generates more than 9000 defects (as the threshold/weight parameters are not configured correctly).

Note that the Setup file described here does not output anything else other than the DPF write map, add the correct output parameters to generate more output if needed.

Using the tool

This section assumes a DPF write map has been generated using the one test application. In order to use it as a Read-map for another run the DPF Converter can be used. But the Thresholds for the DPF module were badly configured and the generated map has several thousand entries.

First run the tool to get information about the map:

```
$ ./dpf_conv -input ctx0_dpf_write_0.dat
```

```
INFO: -output not specified - default used
Convert DPF Write format to DPF read format (confidence 0-255, false
negative 1, false positive 1)
Input: dpf_write0-0.dat
Output: dpf_read
Reading done: 0 corrections dropped - 9095 kept - 9095 found
Confidence (excluding drops): min 1, max 146, avg 3.94
Status (excluding drops): 0 false positive - 9095 false negative
conversion written to dpf_read_09095.dat
```

The confidence value in the map is between 1 and 146 and the average is pretty low (as expected because the thresholds were not setup correctly). Therefore we choose to only take the upper half of the values:

```
$ ./dpf_conv -input ctx0_dpf_write_0.dat -confidence 73 146
```

```
INFO: -output not specified - default used
Convert DPF Write format to DPF read format (confidence 73-146,
false negative 1, false positive 1)
Input: dpf_write0-0.dat
Output: dpf_read
Reading done: 9087 corrections dropped - 8 kept - 9095 found
Confidence (excluding drops): min 79, max 146, avg 103.13
Status (excluding drops): 0 false positive - 8 false negative
conversion written to dpf_read_00008.dat
```

We successfully managed to reduce the output map to a more realistic number of defects.

5.5.7 Sensor Test application sensor_test

This test application is designed to run a sensor with a particular mode without using the ISP. It will use only the Sensor API and CI libraries (the latter only to read information on the gasket). The expected usage is to validate sensor modes on a particular platform. It could also be used to load mode registers from files (the AR330 driver has an example of such feature).

The application will run forever and print the gasket status regularly.

The source is available in `sensorapi/apps/sensor_test.c` and once installed it is available in `utils/`.

Parameters

The parameters can be given by command line or in a configuration file.

The parameter types are:

- *Bool*: a Boolean value of either 0 or 1
- *Int / UInt*: an integer (signed or unsigned is parameter specific)
- *Cmd*: command that act as a Boolean of value 1 if present, of value 0 if not found
- *Str*: string or path

- **Float:** floating point number

-source: Provide a file to parse for command line parameters. The latest given parameter is used as the value for the option.

-h: Display the help and exit.

-sensor: **Str** Sensor name to use (one of the supported ones in sensor API).

-sensorMode: **UInt** Sensor running mode (chooses size and FPS for a given sensor).

-usleepTime: **UInt** [optional] Time to wait in μ s between two printing of the gasket status.

-ar330_registers: **Str** [optional] If using AR330 and specified will override the mode to be the special mode and load the registers from this file.

Note: Press X key to stop the application.

If pressing the key does not work on your platform try to use the *Console Raw Mode testing* (page 86) application and modify it to make it work there.

Using the tool

This example is using the AR330 sensor with a new set of registers. Before compiling the set of registers as a new mode it can be tested agasint the gasket on a test chip to ensure it does not produces errors.

```
$ ./sensor_test -sensor AR330 -ar330_registers ~/test_registers/ar330_newmode.txt
```

But the application could also be use simply to test existing mode 0 of sensor OV4688:

```
$ ./sensor_test -sensor OV4688 -sensorMode 0
```

Or simply to list all the available modes of the sensors:

```
$ ./sensor_test -h
```

5.5.8 Console Raw Mode testing

This is a very small test application to ensure the raw-mode functions used in *sensor_test* (page 85) and *ISPC_loop* (page 69) work.

The source code is available in `ISP_Control/test_apps/src/rawemode_test.c` and will be installed in `utils/`.

This tool only waits for the X key to pressed to quit.

The expected usage is to ensure raw-mode can work on a platform. And to experiment until it works otherwise.

5.6 GUI tools applications user guide

The available GUI applications are:

- *VisionLive* and *ISPC_tcp* (page 87)

- *Vision Tuning* (page 122)

5.6.1 VisionLive and ISPC_tcp

The live tuning tool is designed to allow the ISP to be fully embedded in the target device whilst allowing the tuning tool to run on a system with a large screen and a rich GUI environment. For this reason the tools are split into the ISPC_tcp component, which runs on the target device and the VisionLive component which runs on a PC. The communication occurs on the network through TCP/IP.

The VisionLive application starts with default parameters as well as ISPC_tcp application so there is no need for preparing a parameter file to run the application. After the application has been run the parameter file can always be loaded using “File/Load Configuration” option. The parameters adjusted in VisionLive are transmitted to the ISPC_tcp application which applies these to the ISP. The resulting image can be observed in the “*LiveFeedView* (page 92)”. Frames can also be captured using the “Record” button (see *ImageViewControls* (page 91) details). The captured frames can be used for offline analysis/calibration or viewing. The user is expected to use VisionLive to capture the frames for tuning, using “Tune” buttons in specific modules.

Note: Due to limitations of the network bandwidth FPS ratio may be very low. To increase FPS ratio one can try switching from RGB to YUV format and/or decreasing image size.

The ISPC_tcp application is very similar to the ISPC_loop application detailed in the *ISPC loop application: ISPC_loop* (page 69) section. At any point in time the current configuration can be saved as a Felix Setup Args file (see the *High level parameters* (page 283) documents for details) using the “File/Save Configuration” option.

The version of the tool used is available in the “Help/About VisionLive” menu.

VisionLive contains four main components as displayed in Figure *MainWindow* (page 87):

- *ImageView* (page 90)
- *ModuleView* (page 99)
- *LogView* (page 119)
- *CapturePreview* (page 121)

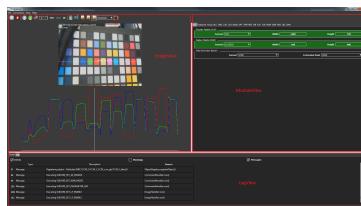


Fig. 5.4: MainWindow

ImageView, *ModuleView* and *LogView* components are separated from each other by splitters so the size of each component can be adjusted without the hidden id. By default splitters are set so only the *ImageView* component is visible and *ModuleView* and *LogView* are hidden. To show hidden components or adjust their size just press and hold the splitter and drag it across the window.

Note: ModuleView component will automatically appear when selecting module from

“View/Module Access” menu.

Warning: Be aware that splitters look different on different operating systems. For example on Windows it’s represented as wide white line, while on Linux it’s represented as three white dots.

Running steps

The first step is to launch the GUI application. This provides the settings that will be used by the ISPC_tcp application. There are no parameters required so simply run VisionLive using “Connection/Connect” option. The GUI has a number of tabs which allow a user to tune various aspects of the ISP, these should be familiar to an experienced ISP engineer.

Once the GUI starts up a connection should be open, so that the ISPC_tcp can connect to the GUI to receive the configuration. Use the “Connection/Connect” option to start one (see Figure *Start connection dialog* (page 88)) and select the desired port. Be aware that VisionLive uses 2 separate ports for connection. One for sending commands to ISPC_tcp and second (selected port + 1) to receive frames for “*LiveFeedView* (page 92)”. If there is problem in connecting be sure that both port numbers are free. After accepting connection settings awaiting connection from ISPC_tcp window will popup (see Figure *Awaiting connection dialog* (page 88)).

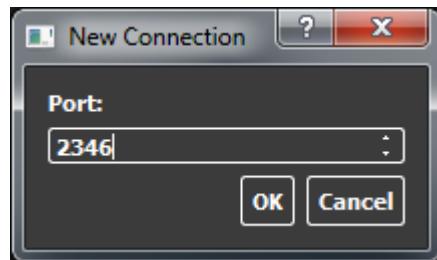


Fig. 5.5: Start connection dialog

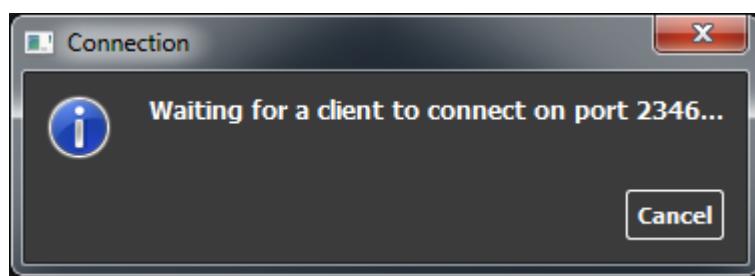


Fig. 5.6: Awaiting connection dialog

Once the GUI is waiting for a connection the ISPC_tcp should be started (see section *ISPC_tcp options* (page 89) for details). For example:

```
$ ./ISPC_tcp -sensor AR330 -setupIP localhost -setupPort 2346
```

ISPC_tcp options

The ISPC_tcp options are very similar to the ones described in the *Test Tools* (page 40) document for the *ISPC loop application: ISPC_loop* (page 69) with the addition of the connection information. It is important that the sensor has been ported in the sensor API to be used with that tool.

Refer to the Sensor API doxygen documentation in the DDK and the *Platform Integration Guide* (page 27) for more information on how to add a sensor. It is possible to run the ISPC_tcp using the Data Generator (external or internal) if the platform supports it, but it isn't possible to run that tool against the simulator for latency reasons.

The parameters can be given by command line or in a configuration file.

The parameter types are:

- *Bool*: a Boolean value of either 0 or 1
- *Int/UInt*: an integer (signed or unsigned is parameter specific)
- *Cmd*: command that act as a Boolean of value 1 if present, of value 0 if not found
- *Str*: string or path
- *Float*: floating point number

-source: Provide a file to parse for command line parameters. The latest given parameter is used as the value for the option.

-h: Display the help and exit.

Sensor controls

-sensor: **Str** Sensor to use. Available sensors are:

- “External Datagen”
- “IIF Datagen”
- AR330
- P401
- OV4688

Obviously this needs the actual sensor to be present. The *Platform Integration Guide* (page 27) has more details about adding support for more sensors.

If using either data-generator the **-DG_inputFLX** should specified the source file.

-sensorMode: **UInt** The mode to run for the selected sensor. Each sensor can have several modes (e.g. to have different resolution or speed). Refer to the Sensor API documentation for more details.

Default 0.

-sensorFlip: **Bool [x2]** Flip the sensor horizontally and vertically. Not all sensors and modes support flipping.

If only 1 value is given it is assumed to be a horizontal flip.

Data Generator Parameters

If using either data generator these parameters can be used. The internal or external data-generator choice is made with the **-sensor** value.

-DG_inputFLX: Str If using either Datagen sensors it is the path to the FLX input file to use.

Similar to **-DGX_inputFLX** and **-IntDGX_inputFLX** in *ISPC_test* (page 61).

-DG_gasket: Str Gasket to use with the data-generator. Similar to **-IntDGX_gasket** or DG number in *ISPC_test* (page 61).

-DG_blank: UInt [x2] Horizontal and vertical blanking in pixels for the data-generator.

Similar to **-DGX_blank** or **-IntDGX_blank** in *ISPC_test* (page 61).

If only 1 value is specified it is assumed to be the horizontal blanking.

Fake interface only

These parameters are related to the CSIM or functionalities available to the fake driver only.

-simIP: Str Simulator IP address when using the Fake interface with TCP/IP.

Default is localhost.

-simPort: UInt Simulator port when using Fake interface with TCP/IP.

Default is 2345.

V2500 controls

These controls are the same as the *ISPC_loop* (page 69) V2500 controls with the exception of the setupFile which will be replaced by connection controls.

-nBuffers: UInt Number of pre-allocated buffers to run with. Also used as the number of data-generator buffers. Similar to **-CTX_nBuffers** in *ISPC_test* (page 61).

Default is 2.

Connection controls

-setupIP: Str IP of the computer running the VisionLive tool.

-setupPort: UInt Port open by the VisionLive when starting the connection.

ImageView

ImageView contains three main components as displayed in Figure *ImageView* (page 91):

- *ImageViewControls* (page 91)
- *LiveFeedView* (page 92)
- *ShowView* (page 93)

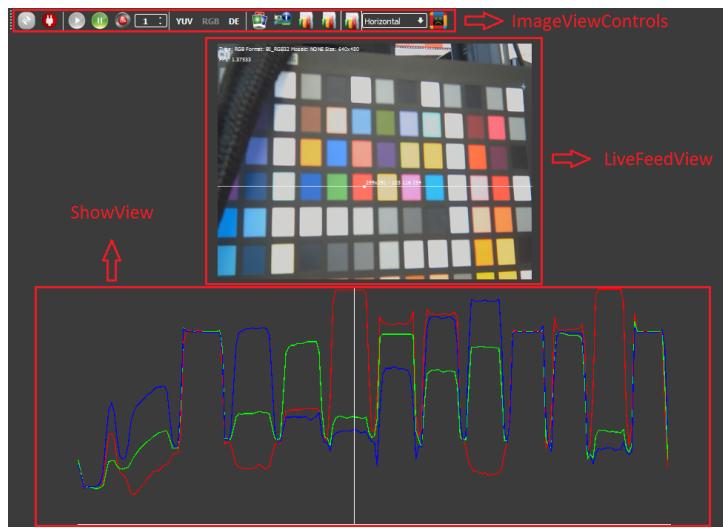


Fig. 5.7: ImageView

ImageViewControls

ImageViewControls is a toolbar which offers faster access to menu options. It contains 16 buttons divided in 4 groups which allows a user to handle connection, livefeed control, image format control and ShowView widget selection as shown in Figure *ImageViewControls* (page 91).



Fig. 5.8: ImageViewControls

ImageViewControls buttons description as displayed in Figure *ImageViewControls* (page 91):

1. Connect - Starts new connection (accessible also from “Connection” menu)
2. Disconnect - Stops connection (accessible also from “Connection” menu)
3. Resume - Resumes receiving images captured by ISP and updating *LiveFeedView* (page 92) frames
4. Pause - Stops receiving images captured by ISP (*LiveFeedView* (page 92) frames are not updated but ISP is still capturing frames and all control algorithms are still processing frames)
5. Record - Captures specified number of frames which can then be viewed in *CapturePreview* (page 121) by selecting a specific snapshot in *CaptureGallery* (page 93)
6. Specifies the number of frames that will be recorded by the “Record” button.
7. YUV - Select YUV format of *LiveFeedView* (page 92) image
8. RGB - Select RGB format of *LiveFeedView* (page 92) image

9. DE - Select Data Extraction format of *LiveFeedView* (page 92) image
10. CaptureGallery - Selects *CaptureGallery* (page 93) to be viewed in *ShowView* (page 93) component
11. HardwareInfo - Selects *HardwareInfo* (page 93) to be viewed in *ShowView* (page 93) component
12. Histogram - Selects *Histogram* (page 93) to be viewed in *ShowView* (page 93) component
13. Vectorscope - Selects *Vectorscope* (page 93) to be viewed in *ShowView* (page 93) component
14. LineView - Selects *LineView* (page 93) to be viewed in *ShowView* (page 93) component
15. Switches between “Horizontal” and “Vertical” mode of *LineView* (page 93)
16. DPF - Selects *DPF* (page 99) to be viewed in *ShowView* (page 93) component

Note: Selecting DPF without properly setting *DPF* (page 108) module will result in no action (no DPF point will be marked on *LiveFeedView* (page 92) image).

Warning: Be aware that selecting image format that is not supported in *OUT* (page 100) module will result in not receiving any image (*LiveFeedView* (page 92) frames will not be updated) and an appropriate warning will be put into the log for every frame.

LiveFeedView

This component shows live feed from camera sensor and on top of it displays information about the frame (format, mosaic and size), fps ratio (calculated based on 10 frames), pixel color values and signal to noise ratio (calculated from selected region). A preview of the window is displayed in Figure *LiveFeedView* (page 92).



Fig. 5.9: LiveFeedView

To display pixel information simply click on the image (left mouse button). If the selection is inside of the frame the selected pixel will be marked and information will appear containing pixel coordinates and colour values.

Warning: Be aware that pixel information depends on the selected image type. For RGB the values are the following: blue, green, and red; for YUV: Y, Cb, Cr and for DE: red, green1, green2, blue.

To display SNR ratio click and hold (left mouse button) on the image. Then move across the image to select region. The selected region will be marked with white rectangle and SNR information will be displayed in the top left corner of the image, below FPS information. Noise information consists of AVG (average pixel values) and SNR (signal to noise ratio in [dB] (SNR = $20\log_{10}(\text{AVG}/\text{SDEV})$)), where SDEV is standard deviation of pixel values. AVG, SDEV and SNR are given per channel. To remove SNR information and selected region marking simply click on the image (right mouse button).

ShowView

ShowView is a component that displays widgets selected with *ImageViewControls* (page 91) with exception of DPF. DPF points markings are displayed directly on the image. As default ShowView component is hidden when no widget is selected.

CaptureGallery CaptureGallery widget holds all recorded captures and displays them and miniatures. After recording, the CaptureGallery widget opens automatically without directly selecting it from *ImageViewControls* (page 91) as displayed in Figure *ShowView - CaptureGallery* (page 94).

Every CaptureGallery item contains a context menu that allows the user to open a *CapturePreview* (page 121) window of specified recording, save recorded frames to disc in JPEG or FLX format, save configuration of the recording and removing selected item. It is also possible to open item simply by double clicking it.

Warning: Be aware that saving an item containing multiple frames in JPEG format will result in creating multiple files on disc, each containing one frame. Saving the same item in FLX format will result in creating one file containing all captured frames.

HardwareInfo This widget displays many hardware information taken from ISP just after connection as displayed in Figure *ShowView - HardwareInfo* (page 95).

Histogram This widget displays histogram of the current frame and is updated every frame as in Figure *ShowView - Histogram* (page 96).

Vectorscope This widget displays pixel colours in HSV format and is updated every frame as in Figure *ShowView - Vectorscope* (page 97).

The Vectorscope is generated from every 10th pixel from each line of the image. The Vectorscope contains only one line per hue (H) value (max 360 lines) of the highest saturation (S) value.

LineView This widget displays a pixel color graph of a selected line and is updated every frame. Selected line is marked with white line on the *LiveFeedView* (page 92) image as displayed in Figure *ShowView - LineView* (page 98).

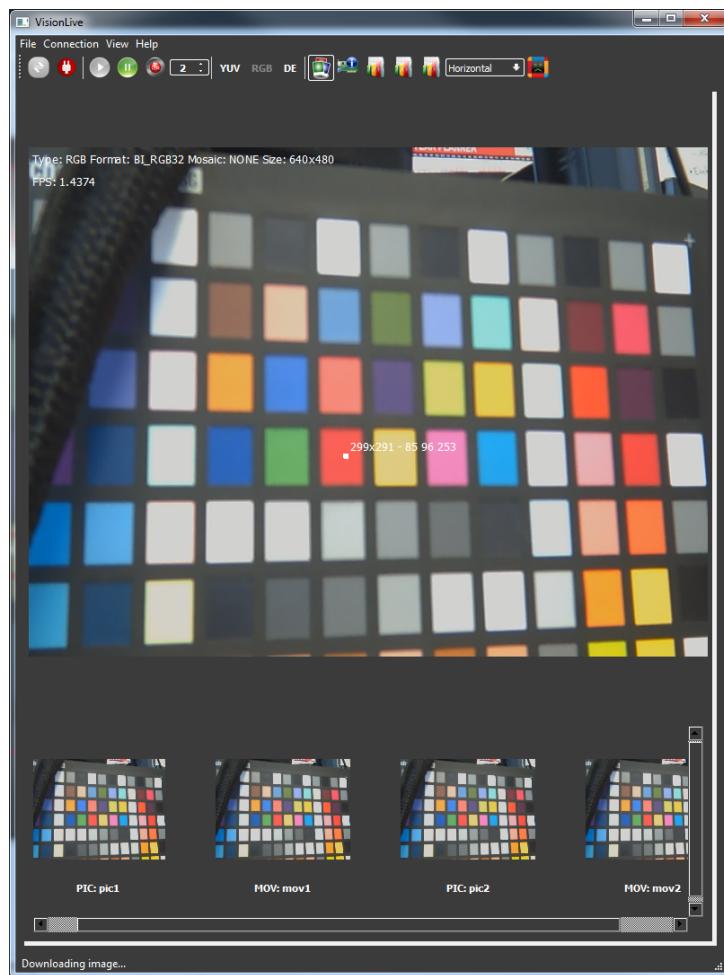


Fig. 5.10: ShowView - CaptureGallery



Fig. 5.11: ShowView - HardwareInfo

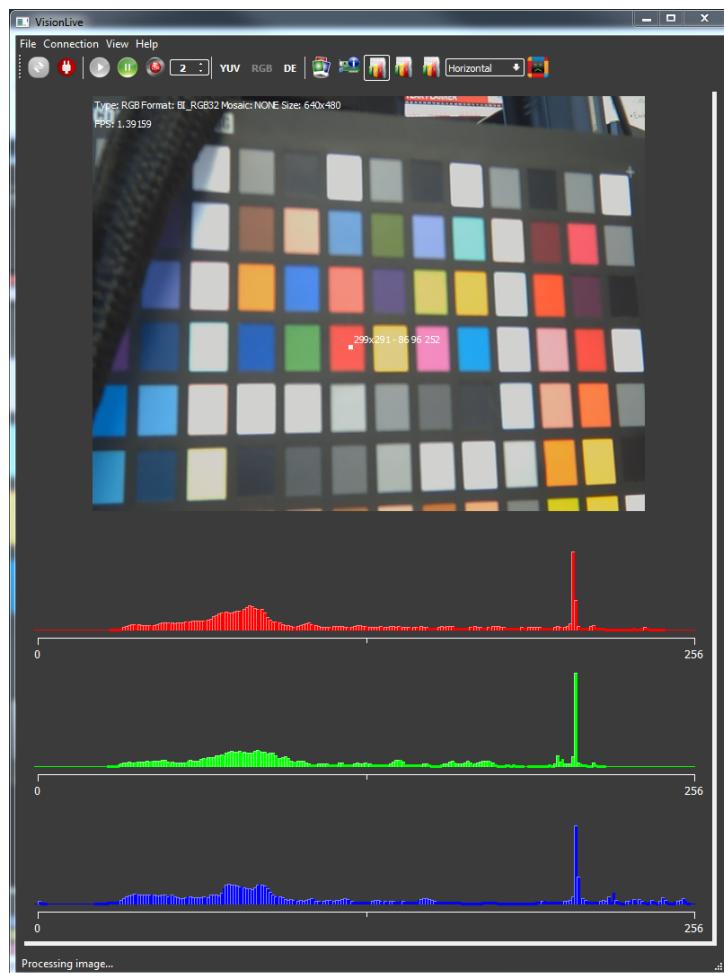


Fig. 5.12: ShowView - Histogram

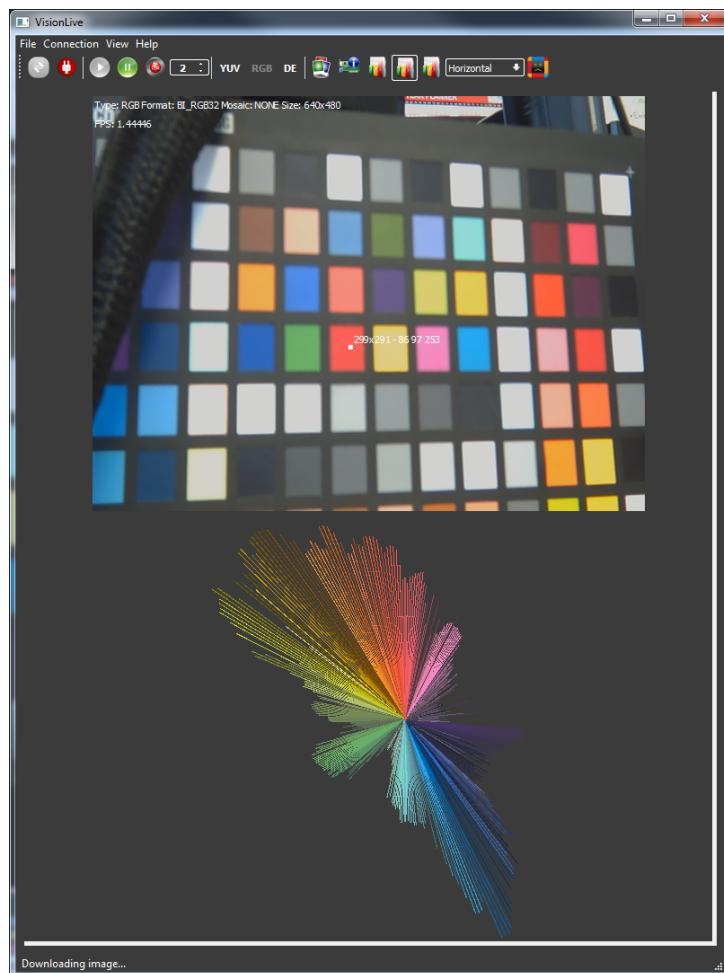


Fig. 5.13: ShowView - Vectorscope

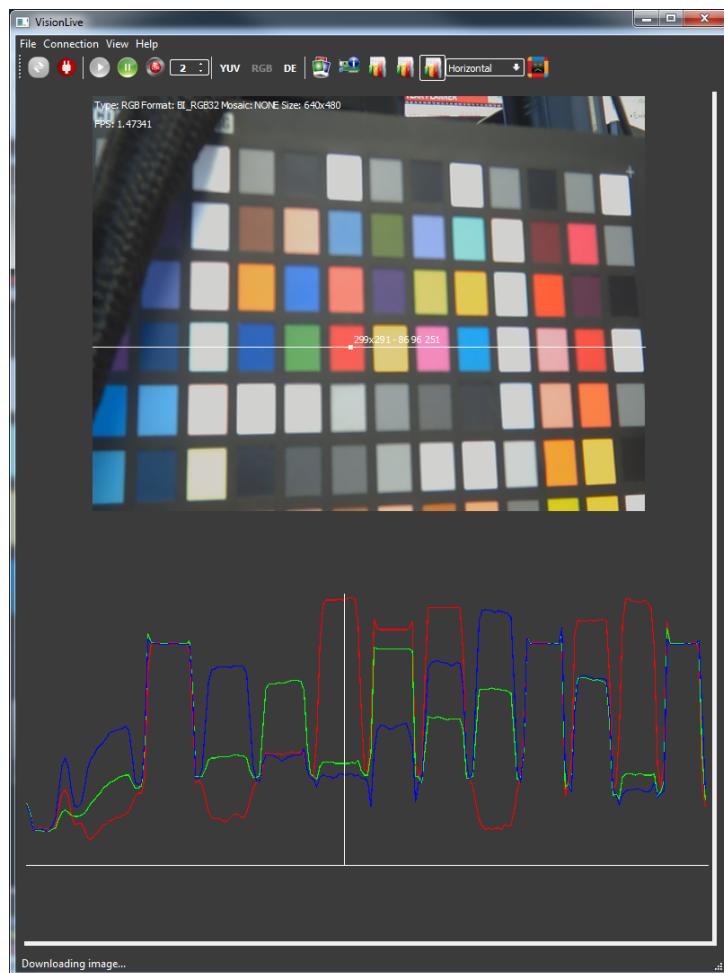


Fig. 5.14: ShowView - LineView

Line selection is done simply by clicking on the image. The LineView can be presented in “Horizontal” and “Vertical” modes by selecting mode in *ImageViewControls* (page 91). The bottom horizontal white line of the LineView widget marks minimum colour value (0) of the pixel while the top of the vertical white line marks maximum colour value (255) of the pixel. The vertical line also marks selected pixel location on the graph.

DPF When the DPF option is selected defective pixels will be marked on the image by red dots and will be updated for every frame as seen in Figure *ShowView - DPF* (page 99).

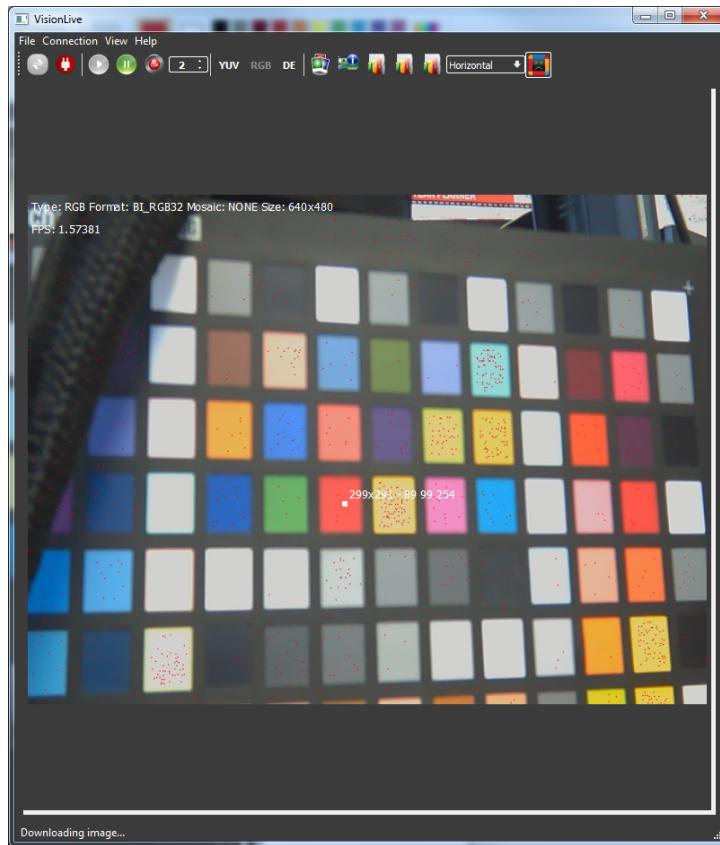


Fig. 5.15: ShowView - DPF

Note: Selecting DPF without properly setting *DPF* (page 108) module will result in no action (no DPF point will be marked on *LiveFeedView* (page 92) image).

ModuleView

ModuleView contains following modules:

- *OUT* (page 100)
- *Exposure* (page 100)
- *Focus* (page 101)
- *BLC* (page 102)
- *WBC* (page 103)

- *LSH* (page 105)
- *LCA* (page 106)
- *Noise* (page 107)
- *DPF* (page 108)
- *TNM* (page 110)
- *MIE* (page 111)
- *VIB* (page 113)
- *R2Y* (page 114)
- *Y2R* (page 115)
- *MGM* (page 115)
- *DGM* (page 115)
- *ENS* (page 117)
- *LBC* (page 117)
- *GMA* (page 118)

OUT

This module lets a user to control size and format of the output images of three different pipeline points. The displayed options are similar to the ones available in *OUT High Level Parameters* (page 211). A preview of the window is available in Figure *OUT Module* (page 101).

More importantly it allows the selection of the Bayer extraction (data extraction) format which will need to be “DE1: before BLC” to capture images for calibration.

Note: Only image formats enabled here can be displayed after in *LiveFeedView* (page 92).

Warning: Be aware that RGB and Bayer formats can't be selected simultaneously (the DDK is enforcing the HW limitations).

Exposure

This module gives a user control over sensor parameters like “Exposure” and “Gain” which can be set manually as displayed in Figure *ModuleExposure* (page 102).

VisionLive provides several controls to tune AE control module which are associated with *AE High Level Parameters* (page 249).

Auto Enable/disable the automatic exposure.

Flicker Rejection Enables/disables flicker rejection option and frequency to reject. If enabled the exposure should move in step with the relevant flicker frequency.

Brightness The **target brightness** sets a target that the AE needs to achieve in [-1,+1]. A bigger value increases the exposure while reducing it should reduce the output brightness. It is expected that a brightness of 0 is balanced but the target brightness is scene dependent.

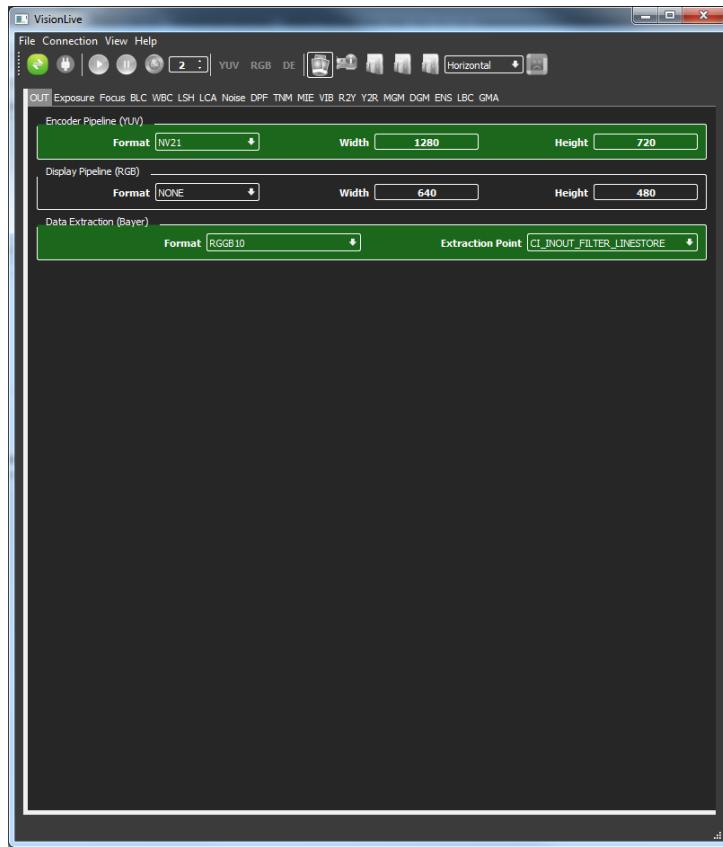


Fig. 5.16: OUT Module

The **measured brightness** is reported by the AE using statistics from the ISP.

Bracket Size Margin around target brightness which we will consider as proper exposure.

Update Speed Update speed for the new exposure to be applied on the sensor in range of [0,1].

Bigger values make the algorithm apply the changes faster while values closer to 0 ensure a smoother transition between the parameters and the application.

Sensor exposure The amount of sensor exposure time the AE is using.

Sensor gain The amount of sensor gain the AE is using as a multiplier.

Set exposure When the AE is turned off the user can program the sensor exposure independently, this is necessary for taking calibration images and when debugging.

Set gain When the AE is turned off the user can program the sensor gain independently, this is necessary for taking calibration images and when debugging.

Warning: Be aware that only the **Auto** option is applied automatically. The rest of the controls have to be applied using “Connection->Apply configuration” option or by using “Ctrl+Shift+A” shortcut.

Focus

This module gives the user control over the sensor focus distance which can be set manually. The user can also set the “Auto” option to enable AF control module and press the “Search”



Fig. 5.17: ModuleExposure

button to run through the whole sensor distance range and find the best focus distance. The window preview is available in Figure *ModuleFocus* (page 103).

Auto Enable the auto-focus algorithm.

Warning: Does NOT trigger the focus search.

Search Triggers the search for the sharpest focus position.

State and Scan State Feedback on progress of AF.

Distance Current distance displays the current focus as reported by the sensor. If auto-focus is disabled the distance can be set manually.

Warning: Be aware that only the “Auto” option and “Search” action are applied automatically. The rest of the controls have to be applied using “Connection->Apply configuration” option or by using “Ctrl+Shift+A” shortcut.

BLC

This module allows the BLC configuration to be modified while running as displayed in Figure *ModuleBLC* (page 104).

The result can be previewed using Bayer capture (with selected DE point after LSH). The associated parameters are described in *BLC High Level Parameters* (page 203).

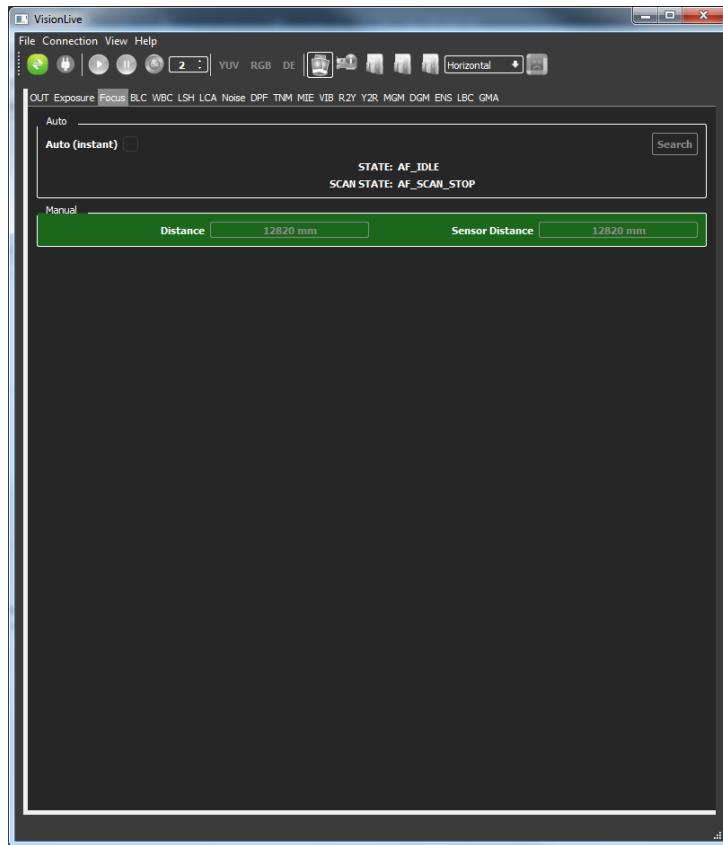


Fig. 5.18: ModuleFocus

Calibration of the BLC is explained in *Sensor BLC tuning* (page 130).

Sensor black level This is the amount of black level that need subtracted from the sensor signal. This would be one of the first operations to be done in the ISP.

System black level This is the value that the system black level is set to.

Warning: The value is 64 and should not be changed (expected when requested for tuning purposes).

Warning: Be aware that all the controls have to be applied using “Connection->Apply configuration” option or by using the “Ctrl+Shift+A” shortcut.

WBC

This module allows the user to apply different CCM configurations. The user can also set one of the “Auto Mode” options to let AWB control module do automatic adjustments to CCM configuration. The window preview is available in Figure *ModuleWBC* (page 105).

This is using the high level setup described in *Automatic White Balance (AWB)* (page 252).

Description on how to tune the CCM for white balance correction is available in the *White Balance Correction Tuning* (page 131).

VisionLive provides some information from the AWB control module:

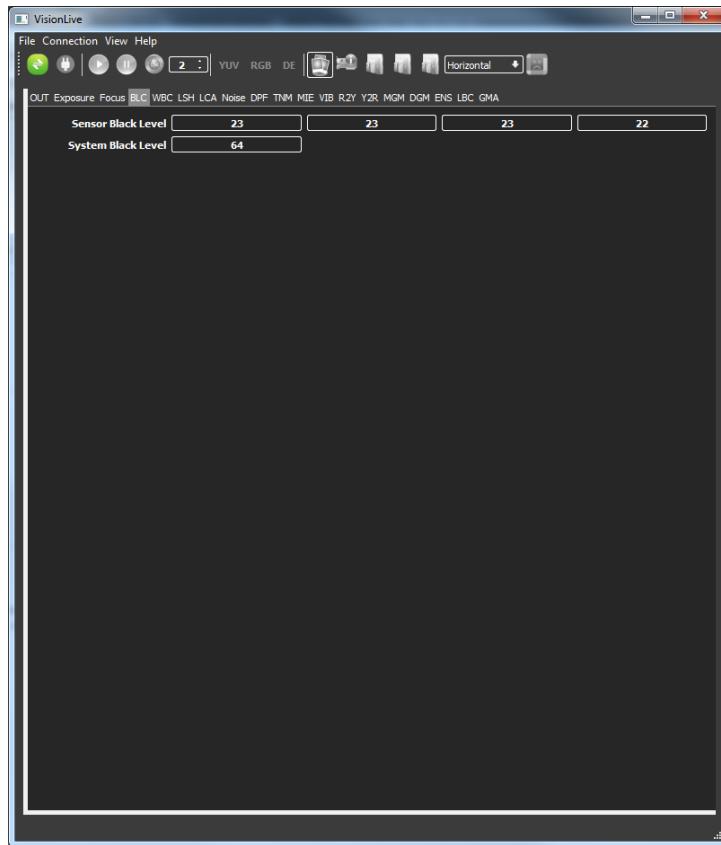


Fig. 5.19: ModuleBLC

Auto mode This is a choice between disabled (AWB off), Average Colour, White Point, High illuminant white and combined.

The current recommendation is combined, however it is advised that the customer does relevant testing and make their own choice.

Measured Temperature Currently measured image temperature

Gains, Clips, CCM Current state of white balance and colour corrections

From hardware version 2.6 Auto White Balance algorithm is using White Balance Statistics (AWS) module to determine which pixels to use in white balance correction. WBC module also provides controls needed to tune AWS module.

From hardware version 2.6 WBC module also enables controls for tuning Temporal Smoothing.

Debug mode Turns on AWB debug mode which marks on live feed image (with red dots) pixels that have been taken into calculations on white balance correction.

Warning: For “debug mode” to display accurate data DataExtraction format should be selected.

This module also provides option of using multiple CCM configurations. The first tab CCM is the default CCM used (using *CCM High Level Parameters* (page 204)) while the others are the ones used by the AWB algorithm.

Promote to Multi CCM Adds a CCM configuration to the Multi CCM set.

Return to Identity Turns a CCM configuration to its default state.

The CCM used by the AWB have additional features:

Temperature Is the temperature of light they are correcting. This will be used by the AWB to know which CCM to interpolate from when running.

Use when applying If un-ticked this CCM will be ignored when the configuration is applied to the ISP.

Set as default To override the default CCM with the values of the current CCM.

Remove To delete this CCM from the list.

Warning: Be aware that only the “Auto Mode” option is applied automatically. The rest of the controls have to be applied using “Connection->Apply configuration” option or by using “Ctrl+Shift+A” shortcut. Know that only CCM configurations with “Use when Applying” option checked will be applied.



Fig. 5.20: ModuleWBC

LSH

This module provides the option to provide multiple LSH grids that can be used later by AWB control module or manual selection of one of them.

The associated high level parameters are described in the *LSH High Level Parameters* (page 215) section.

The LSH grid files have to be previously created using the “Tune” option.

Tuning for the LSH is explained in section *Sensor Lens Shading Tuning* (page 131).

Add Sends selected LSH grids to the ISP and loads them.

Apply Apply one of the grids to be used. This option is disabled when AWB control module is running because then AWB applies grid that matches best current calculated image temperature.

Remove

Removes selected grids from ISP.

Current LSH grid Displays file name of the grid currently applied by AWB control module.

Warning: Be aware that all actions are applied automatically.

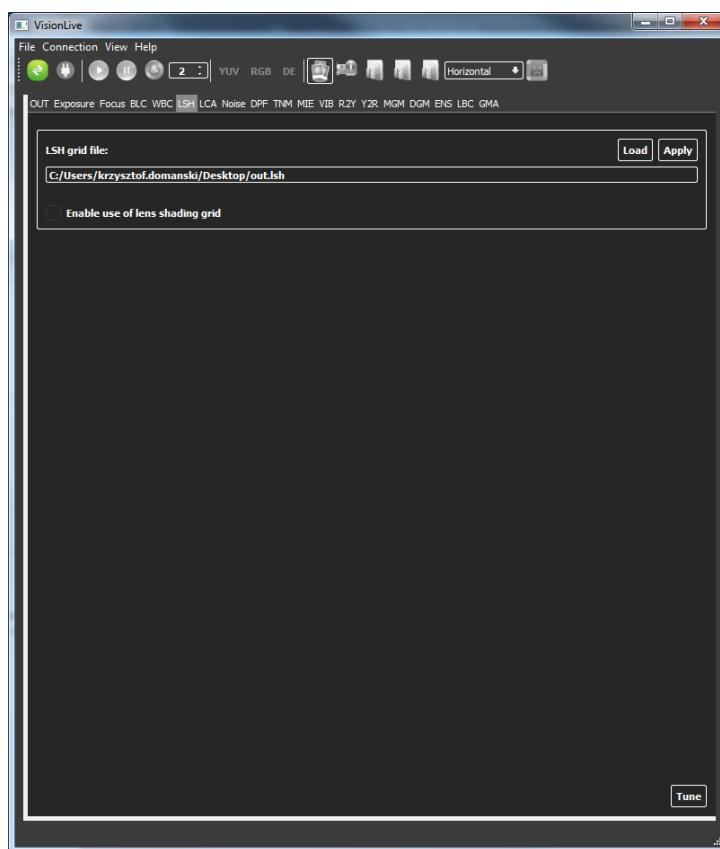


Fig. 5.21: ModuleLSH

LCA

The window preview is available in Figure *ModuleLCA* (page 107).

This module's parameters should be tuned in the way described in *Lateral chromatic aberration (LCA)* (page 128).

Use the "Tune" button to tune LCA module.

Warning: Be aware that all controls have to be applied using “Connection->Apply configuration” option or by using “Ctrl+Shift+A” shortcut.

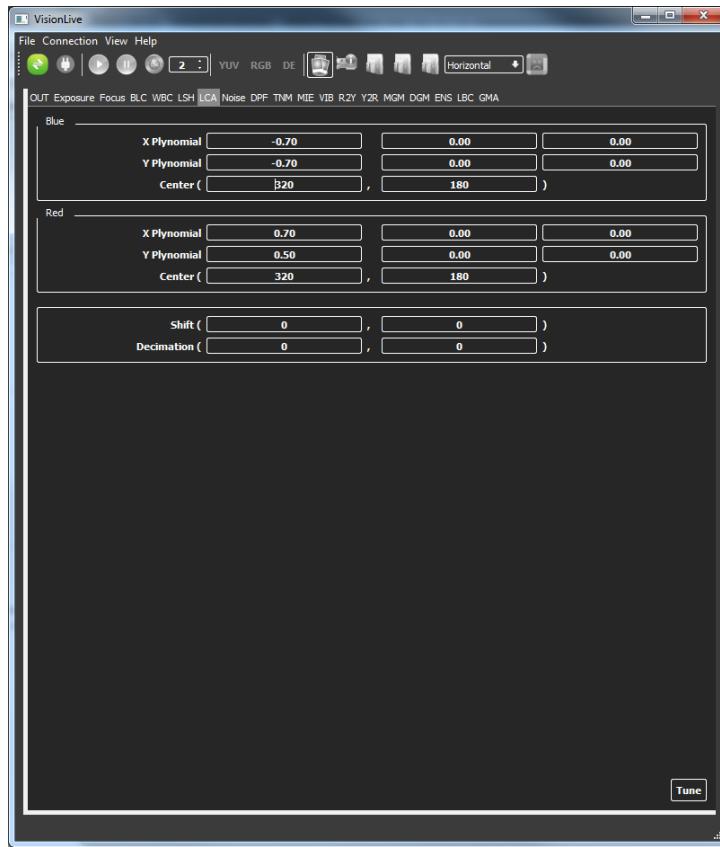


Fig. 5.22: ModuleLCA

Noise

This module allows the user to modify the high level parameters of the V2500 denoise and sharpening modules as displayed in Figure [ModuleNoise](#) (page 109).

Information on how to tune the noise is available in the [Primary Denoiser Tuning](#) (page 132) and [Sharpening and Secondary denoiser Tuning](#) (page 134).

It contains the control for the *auto denoiser* which is the simplest algorithm in ISPC (gathers gain from the sensor and adapts the associated DNS parameter).

Auto Control the automatic gathering of the DNS sensor gain parameter.

Controls for the Primary denoiser are done using high level parameters described in [Primary denoiser \(DNS\)](#) (page 206).

Strength Is how much correction the primary denoiser should apply to the image.

Note: The parameters detailed in [Primary denoiser \(DNS\)](#) (page 206) contain several other parameters which are mostly provided by the sensor (e.g. sensor bitdepth). Therefore it is important when porting a sensor to provide accurate information for the denoiser to perform correctly. information for the denoiser to perform correctly.

The Sharpening module is situated after the Tone Mapper module and prior to the Encode and Display Scalers. It consists of a sharpening sub-block and a denoising sub-block which operate in parallel, sharing an edge detection pre-filter. The final output of the module is the sum of the edge enhancement signal produced by the sharpening logic, and the denoised pixel generated by the denoising logic.

The module has a number of controls available in *SHA High Level Parameters* (page 225), only a subset will need tuned.

Radius This is the radius in pixels that sharpening will be applied to.

Any value over 5 would be meaningless, if blur is spread over 5 pixels then it will only have a small impact. Typical values for modern small pixel sensors would be in the range of [1.5, 2.5],

Strength Correction strength.

Threshold Should stay at 0.

Detail This parameter will determine what gets sharpened (0 for only edges, 1 for everything).

Finally the Secondary denoiser allows some further refinement of the noise after sharpening was applied (using secondary denoiser parameters from *Secondary Denoiser (part of SHA in HW)* (page 206)):

Edge avoidance As this is increased more edge avoidance will occur. This is called Tau multiplier in the parameters.

Strength As this is increased more noise will be suppressed. This is called Sigma multiplier in the parameters.

Warning: Be aware that only the “Auto” option is applied automatically. The rest of the controls have to be applied using “Connection->Apply configuration” option or by using “Ctrl+Shift+A” shortcut.

DPF

The DPF module does live defect fixing by detecting pixels that have values that appear incorrect in comparison to the pixels immediately around them, and correcting the values using interpolation of the local pixels. The window preview is available in Figure *ModuleDPF* (page 110).

The GUI is using the *DPF High Level Parameters* (page 208) to control the module.

Information about tuning this module is available in the *Defective Pixels Tuning* (page 132) section.

If a defect map is available, this can be used instead of, or together with, the live detection.

Live defect detection is applied to the current frame without reference to historical data. As the DPF follows the main denoiser, the random sensor noise will have been substantially reduced, and the defective pixels are either stuck pixels that don't have a normal photometric response, or normal pixels carrying random noise that exceeded the denoiser threshold (e.g. 4 sigma).

The DPF is located after the primary denoiser, before the LCA correction and the demosaicer, in white balanced Bayer space.

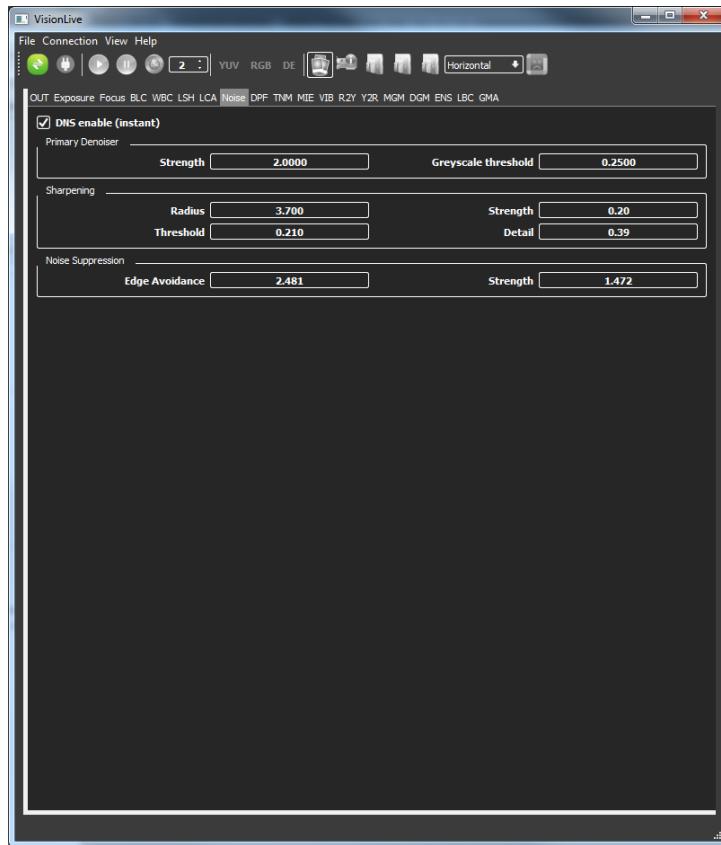


Fig. 5.23: ModuleNoise

The GUI is split into controls and statistics information:

HW Detection Will control the live detection. When enabled the *fixed pixel* should be updated.

Threshold The detection threshold.

Weight The detection MAD (Median of Absolute Deviation) weight.

Output Map Allows the detected pixels to be saved to memory. When enabled *written to map* and *dropping from writing* should be enabled.

Input Map Allows usage of previously stored DPF map to be taken into account when correcting defective pixels.

Load Use to load previously saved DPF map.

Select Confidence Range and **Select S** These parameters are used to filter loaded DPF map.

Apply Applys loaded DPF map.

The statistics display basic information about the detected pixels:

Fixed pixels Is the number of defects the HW detected with live detection.

Written to map Is the number of defects the HW wrote to the defect map. This can be less than the number of detected pixels as the map has a finite size and a limit per line.

Dropped from writing The number of pixels that were detected but not written to the output map. This can happen because the output map is too small to store all detected pixels or

because the number of defects was to high for the HW internal defect buffer as explained in the HW TRM.

Warning: Be aware that all the controls have to be applied using “Connection->Apply configuration” option or by using “Ctrl+Shift+A” shortcut.

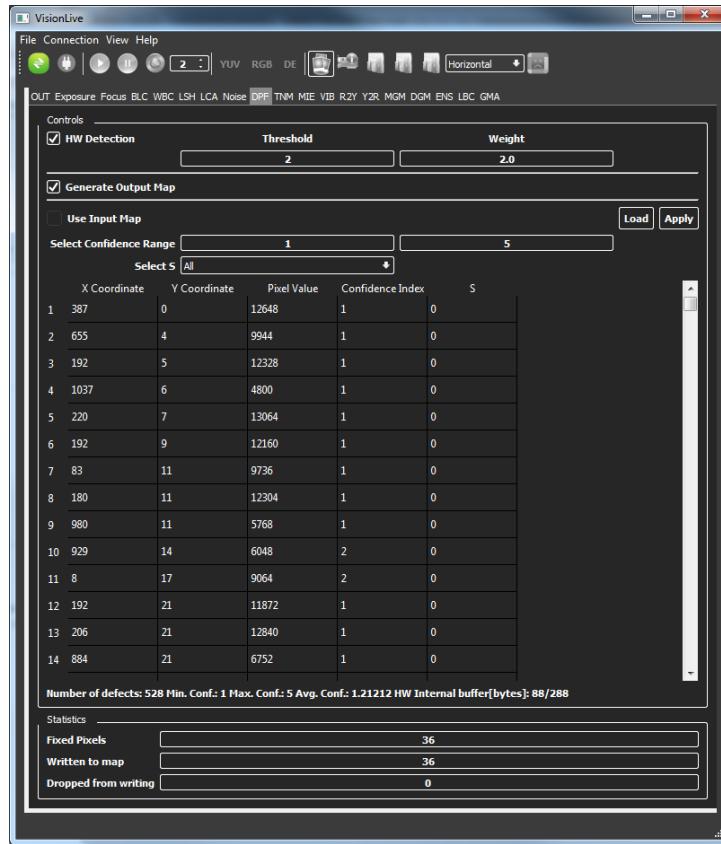


Fig. 5.24: ModuleDPF

TNM

This module allows control over the auto tone mapping curve computation and the preview of the applied result. It also gives control over the local tone mapping. The user can also set the “Auto” option to let TNM control module do automatic adjustments of those parameters. The window preview is available in Figure *ModuleTNM* (page 112).

Information about the algorithm is available in *Algorithm overview* (page 226) and tuning in *Tone-mapping Tuning* (page 135).

VisionLive provides several controls to tune Auto TNM control module using the *Automatic Tone Mapping Control (TNMC)* (page 272) parameters:

Auto Enable/disable the control module.

Adaptive Enable/disable adaptive tone mapping strength control

Hist. Min, Hist. Max Histogram clipping range for global tone mapping curve generation

Tempering Tempering value for global tone mapping curve generation, more tempering produces more gentle mapping curves.

Smoothing Smoothing value for global tone mapping curve generation.

Local TNM Enable/disable local tone mapping.

Local Strength local tone mapping strength

Update Speed update speed for application of newly generated global curves

It also provides controls to the normal tone mapping parameters *TNM High Level Parameters* (page 229):

Luma range Luma range that the tone mapper considers as input.

It is recommended to never change this value from -64, 64.

Bypass TNM Disables the tone mapping. The luma range is still applied.

The other parameters are explained in the tuning section *Tone-mapping Tuning* (page 135).

The GUI also allows the live preview of the applied curve and allow to modify the curve manually.

Editable Make the curve preview points movable vertically.

Warning: When editing the curve manually it is important to know that the curve must be increasing (i.e. every point has to be higher than the previous one). If the curve becomes invalid it will turn red.

Interpolate forward All points after the moving ones are interpolated to create a linear curve to 1.

Interpolate backward All points before the moving ones are interpolated to create a linear curve from the 0.

Reset Resets to the last applied curve.

To identity Regenerate the default curve for the TNM (identity from 0 to 1).

Warning: Be aware that only the “Auto” option is applied automatically. The rest of controls has to be applied using “Connection->Apply configuration” option or by using “Ctrl+Shift+A” shortcut.

MIE

This module gives control over the Memory Colour part of the MIE block of the V2500 HW. The MIE block has 3 memory colours that can be enhanced. The preview window is available in Figure *ModuleMIE* (page 113).

The import button can be used to import a set of YUV values to automatically set the Y slices and CbCr centre but the user is expected to have to calibrate the other options.

The details about the MIE block is available in the *Algorithm overview* (page 217) section. Tuning information is available in the *Image Enhancer Tuning* (page 139) section.

Enable MC Tick to enable the associated memory colour parameters. If disabled the parameters are ignored.

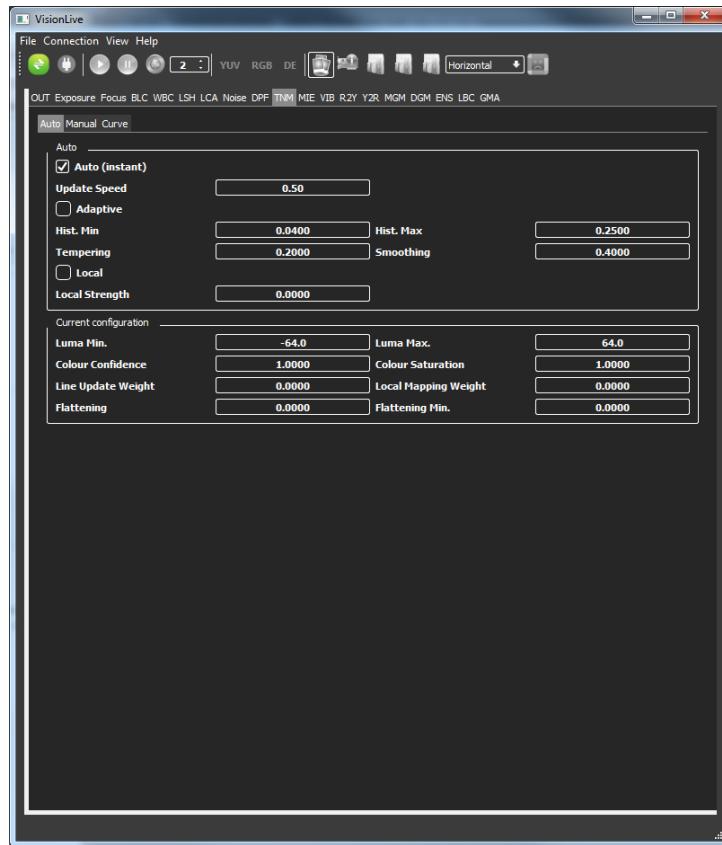


Fig. 5.25: ModuleTNM

Import value Can be used to import the values to the luma slice and CbCr center for the current memory colour tab. This option automatically imports pixel colour info of pixel selected in *LiveFeedView* (page 92).

For each memory colour tab:

Slice selection radio button Allows the user to highlight the selected slice in the luma slices and the diamond Chroma view. This is only visual.

Luma slide min/max Luma values computed for the colour being selected.

Luma gain controls Strength of the modification per slice. Should add up to 1.

Cb/Cr center The value in Cb/Cr space of the colour that should be changed.

Note: This is the *input* of the MIE block therefore to see the value in the preview MIE and TNM should be turned off.

Aspect Changes the shape of the Chroma diamond.

Rotation Rotation of the colour diamond.

Extent control How big the colour selection diamond will be for each slice. The smaller the selection the more focussed the colour space, if the value is too big then it turns into a global operator.

Show selected colours If ticked the brightness is forced to -1 to display selected colours as black pixels on the preview.

Output HSBC The *hue*, *saturation*, *brightness* and *contrast* modifiers to apply on the selected pixels.

Warning: Be aware that all the controls have to be applied using “Connection->Apply configuration” option or by using “Ctrl+Shift+A” shortcut.

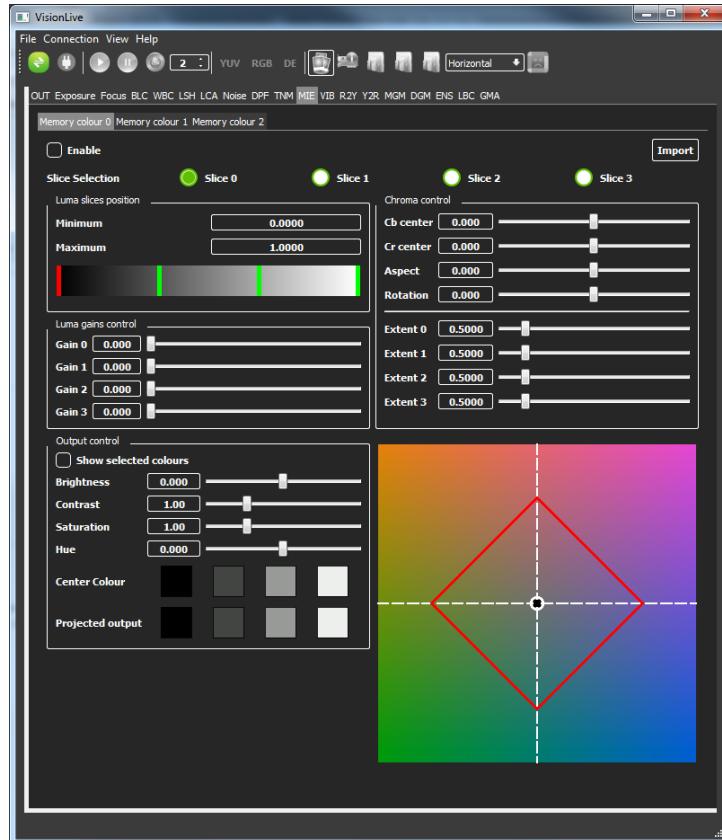


Fig. 5.26: ModuleMIE

VIB

This module enables manipulation on a saturation curve used by the MIE HW block. The saturation curve is converted to vibrancy gains for specific colour by the driver. The window preview is available in Figure *ModuleVIB* (page 114).

The setup is performed using the *VIB High Level Parameters* (page 230). Details about the algorithm is also available in the section *Vibrancy (VIB)* (page 230).

The user can either drag and drop the two points of the curve or set the position with the spinboxes for more precision.

Warning: Be aware that all the controls have to be applied using “Connection->Apply configuration” option or by using “Ctrl+Shift+A” shortcut.

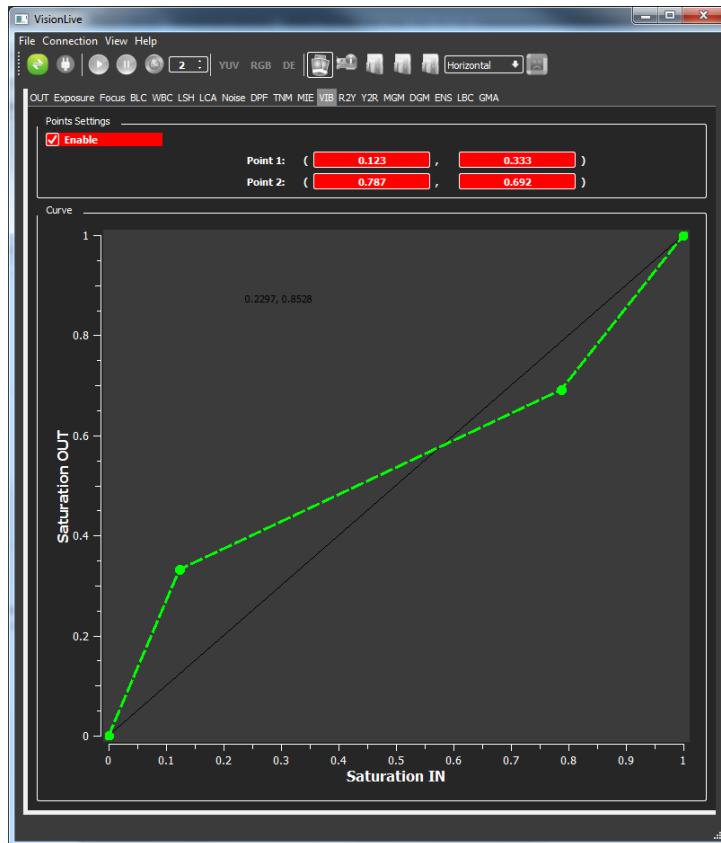


Fig. 5.27: ModuleVIB

R2Y

This module allows some control over the R2Y module of the pipeline. It is not expected that a customer will have to modify these values but it can be used to saturate/desaturate images before the image enhancement modules (MIE and TNM). The window preview is available in Figure [ModuleR2Y](#) (page 115).

The R2Y block is a CSC that will convert RGB to YCbCr after the Gamma was applied.

Brightness Global brightness offset (-0.5, 0.5)

Contrast Global contrast (luma gain) control (0.1, 10.0)

Saturation Global saturation (chroma gain) control (0.1, 10.0)

Hue Global hue control (rotation in degrees) (-30, 30)

Conversion matrix Choice conversion matrix format (BT709 or BT601).

Range multiplier Are gain multipliers for each YUV channels.

Warning: Be aware that all the controls have to be applied using “Connection->Apply configuration” option or by using “Ctrl+Shift+A” shortcut.

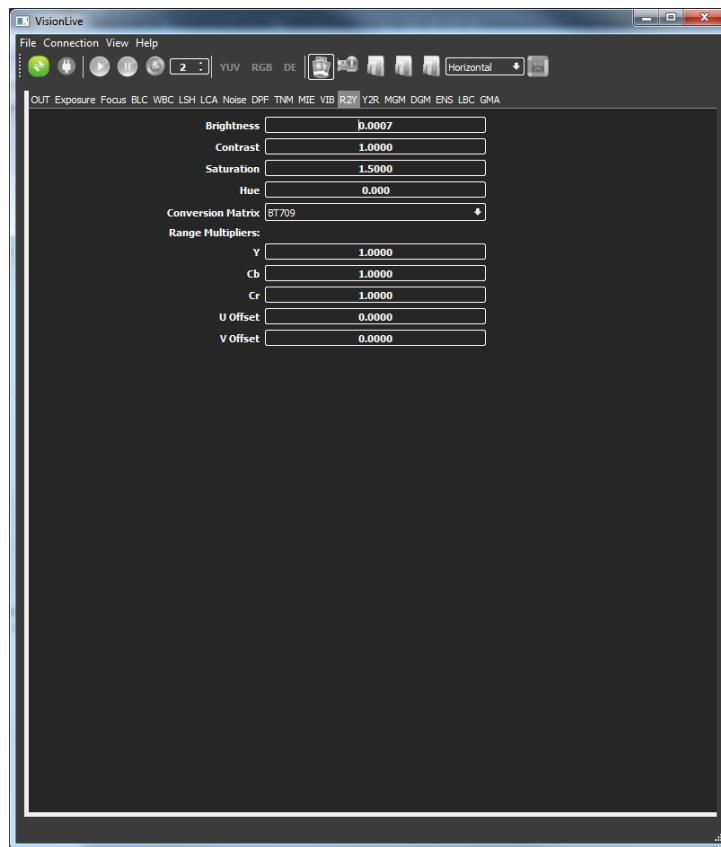


Fig. 5.28: ModuleR2Y

Y2R

This module allows some control over the Y2R module of the pipeline. It is not expected that a customer will have to modify these values but it can be used to saturate/desaturate images just before the display gamut mapper (DGM). The preview window is available in Figure *ModuleY2R* (page 116).

Warning: Be aware that all the controls have to be applied using “Connection->Apply configuration” option or by using “Ctrl+Shift+A” shortcut.

MGM

This module allows users to tune Main Gamut Mapper module parameters as displayed in Figure *ModuleMGM* (page 116).

Warning: Be aware that all of the controls have to be applied using “Connection->Apply configuration” option or by using “Ctrl+Shift+A” shortcut.

DGM

This module allows users to tune Display Gamut Mapper module parameters as displayed in Figure *ModuleDGM* (page 117).

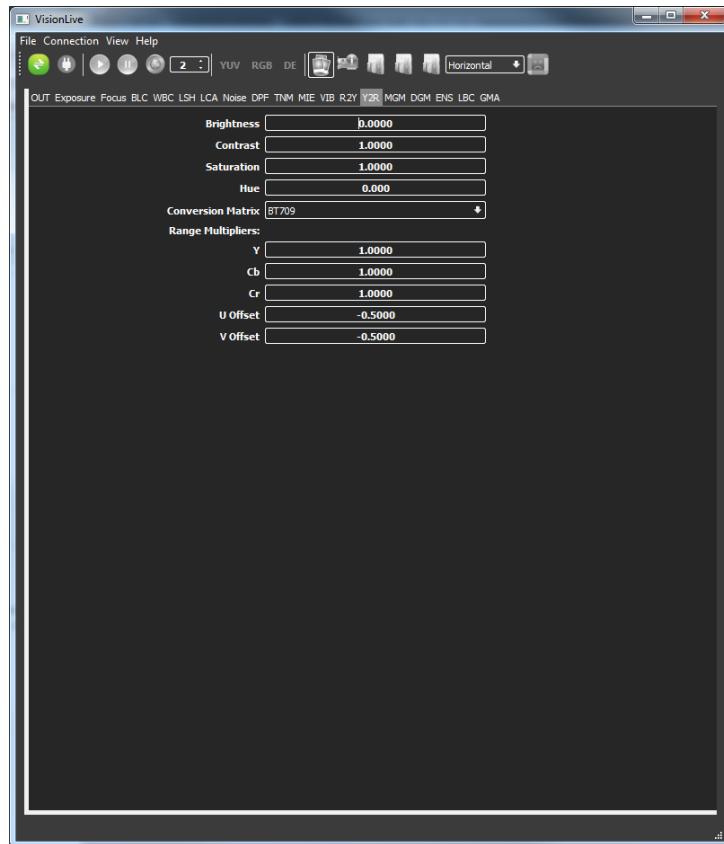


Fig. 5.29: ModuleY2R

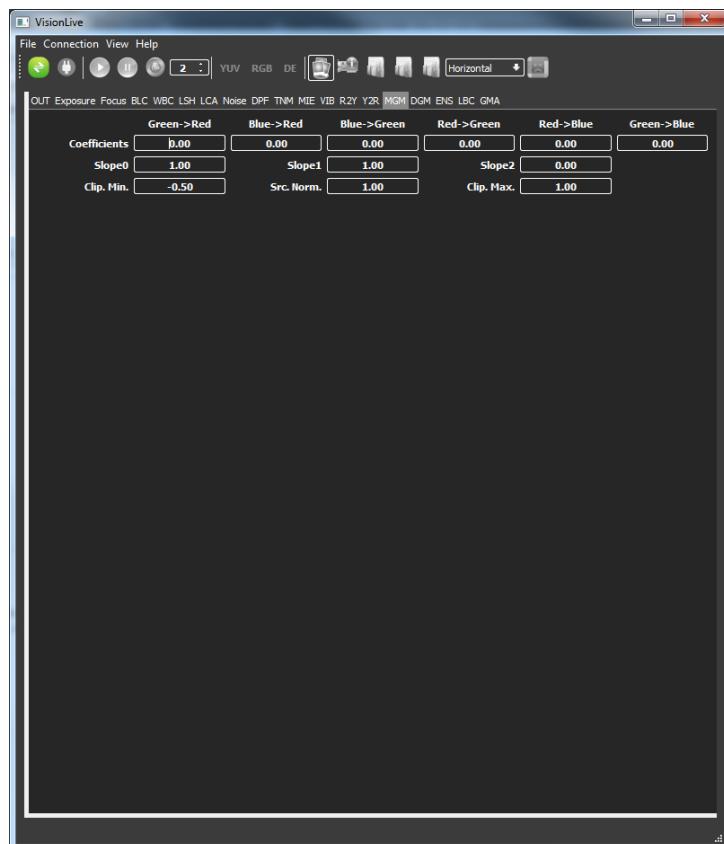


Fig. 5.30: ModuleMGM

Warning: Be aware that all of the controls have to be applied using “Connection->Apply configuration” option or by using “Ctrl+Shift+A” shortcut.

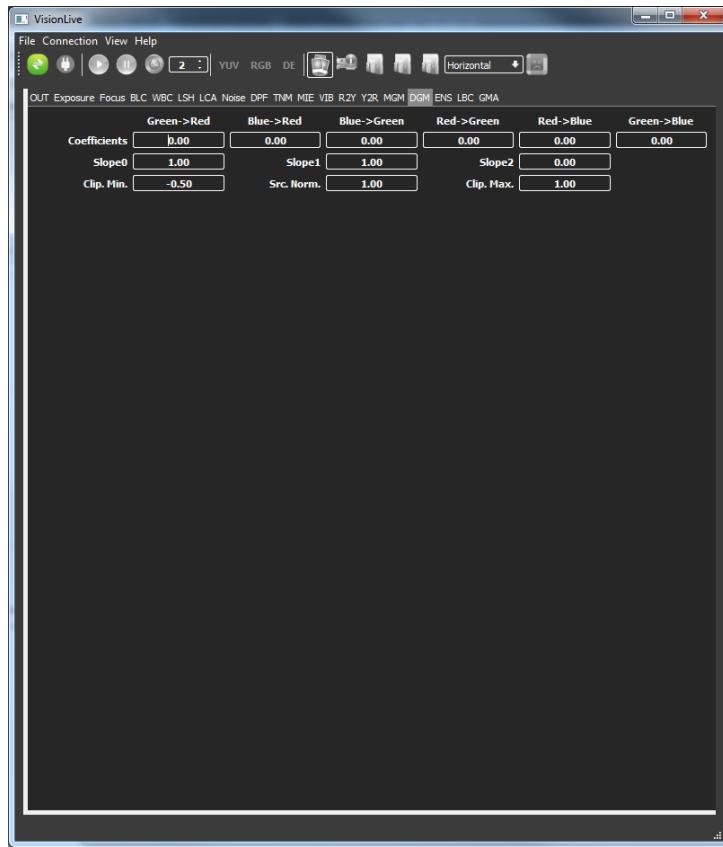


Fig. 5.31: ModuleDGM

ENS

This module allows users to enable generation of encoder statistics output. The module also gives information about expected statistics size as displayed in Figure *ModuleENS* (page 118).

Warning: Be aware that all of the controls have to be applied using “Connection->Apply configuration” option or by using “Ctrl+Shift+A” shortcut.

LBC

This module allows a user to create light based configurations for specific image light levels for use by LBC control module.

Users can enable/disable LBC control module using “Auto” option. When LBC is enabled it will extrapolate between provided configurations to find values (brightness, contrast, saturation, sharpness) for currently detected light level. The preview window is available in Figure *ModuleLBC* (page 119).

VisionLive provides several controls to tune the LBC control module:

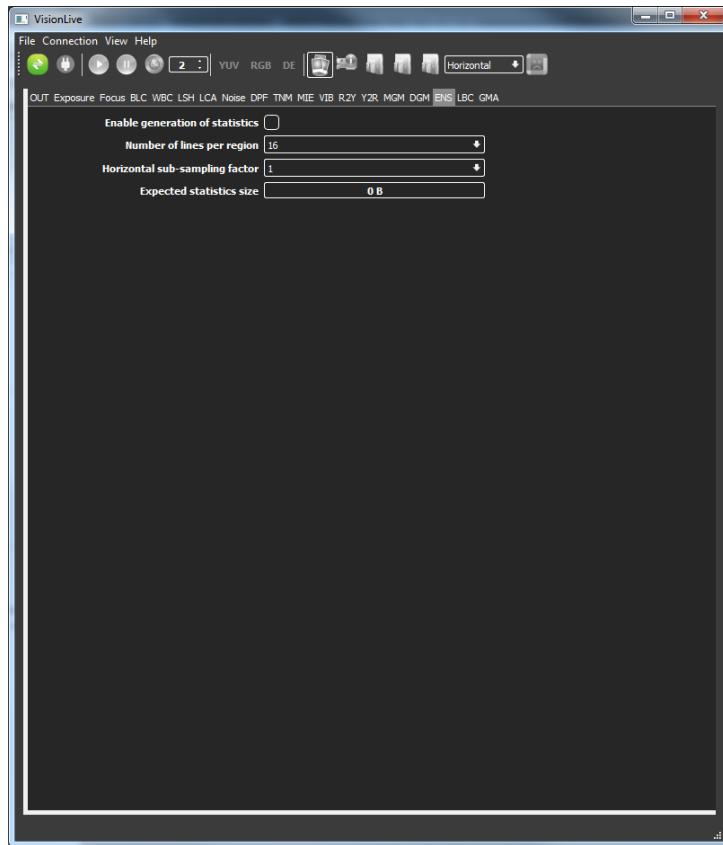


Fig. 5.32: ModuleENS

Update Speed speed at which light metering is done so there is smooth transition between configurations

VisionLive provides some information from LBC control module:

Measured Light Level currently calculated image light level (combination of exposure, gain and brightness)

Current Configuration currently used configuration (extrapolated based on provided configurations and Measured Light Level)

Warning: Be aware that only “Auto” option is applied automatically. The rest of the controls have to be applied using “Connection->Apply configuration” option or by using “Ctrl+Shift+A” shortcut.

GMA

The GMA module gives users the ability to view and create new gamma look up tables to perform gamma correction as displayed in Figure *ModuleGMA* (page 120).

On start up there are two loaded gamma LUT's (BT709 and sRGB). These standard gamma curves can be exported to create new (editable) gamma LUT's. All created gamma LUT's can be exported as pseudo code to a .txt file to be used later (compiled in code as explained in the *Gamma Look-Up table customisation* (page 148) section). Gamma LUT's can be presented (and exported) as register values (integers of range <0-4095>) or preview values (doubles of range

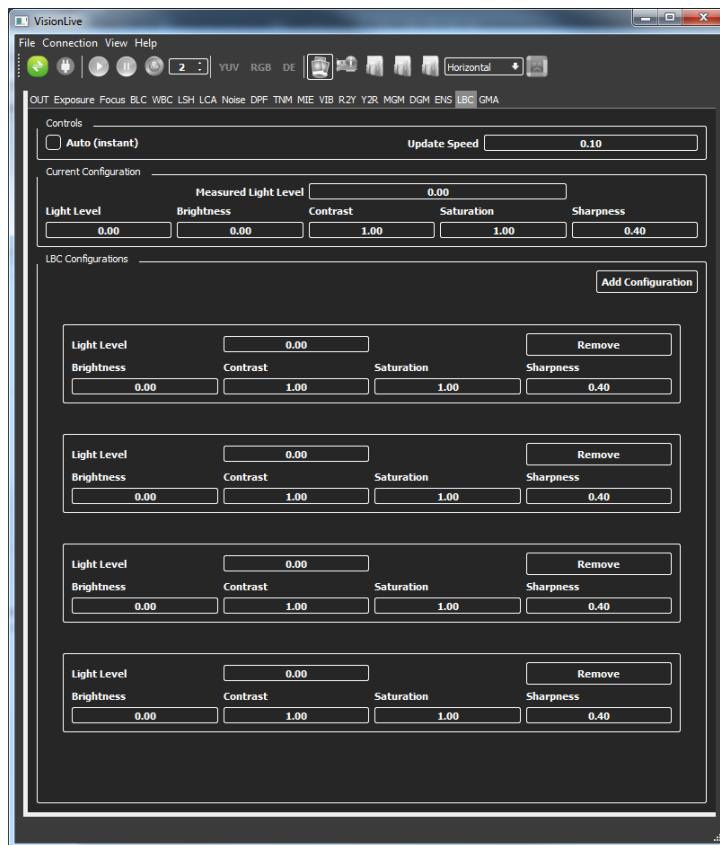


Fig. 5.33: ModuleLBC

<0-255>). User has also ability to change Gamma LUT while running VisionLive observe image change online.

Warning: Be aware that “Change Gamma LUT” option is applied automatically.

LogView

The LogView component displays application logs and actions as shown in Figure *LogView* (page 120).

Each log contains type, description and source. There are three types of logs: Error, Warning and Message. Description contains log information (what happened, what is wrong) and source tells where did described situation occurred. Source information is very useful for debugging proposes but rather irrelevant to the user.

Logs can also be filtered by using three checkboxes at the top of “Log” tab. Only logs of selected type will be displayed. As default only error type logs are displayed. Logs can be exported to “.txt” file using “File/Export Log” option. The “Action” tab of the LogView contains information about actions done by the user while running application. This data can later be used to solve issues the user might experience while running the application. Action can be exported to “.txt” file using “File/Export Action Log” option.

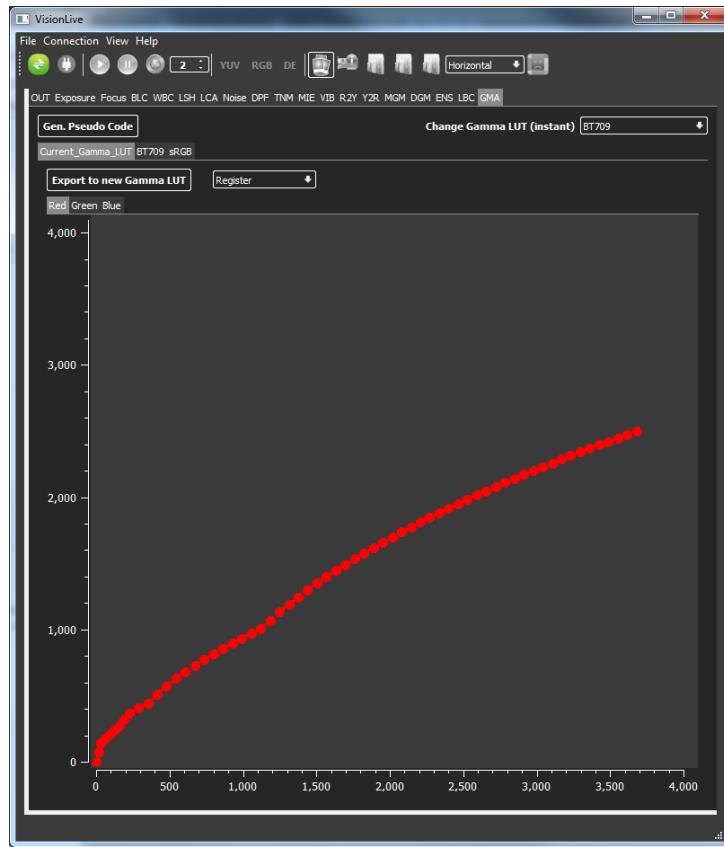


Fig. 5.34: ModuleGMA

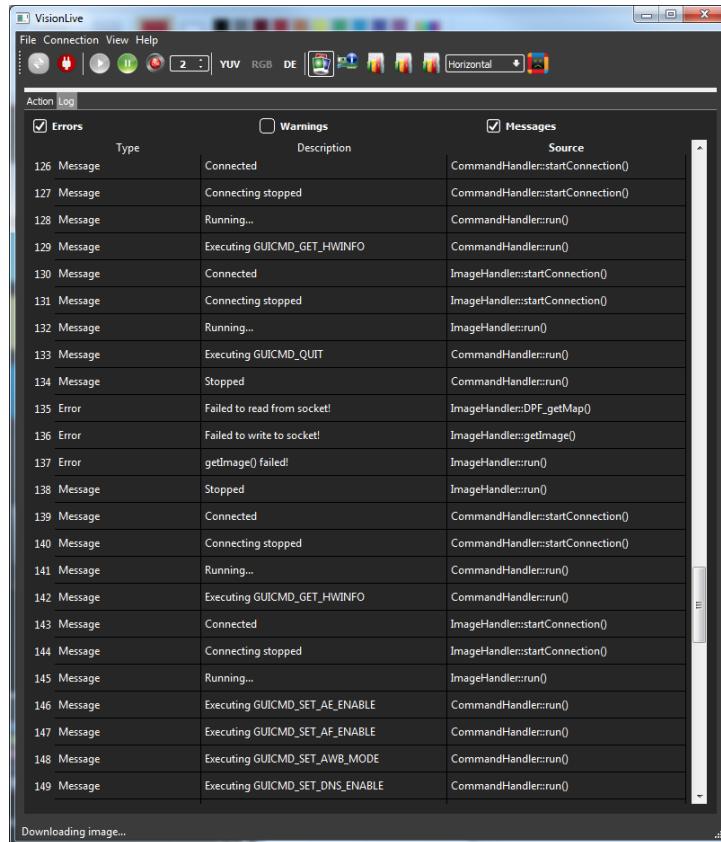


Fig. 5.35: LogView

CapturePreview

This component allows the user to view and analyse recorded frames. It can display *Histogram* (page 93), *Vectorscope* (page 93), *LineView* (page 93) and *DPF* (page 99) in the same manner as *ShowView* (page 93). Also this component displays DPF output map (if DPF module has been set up correctly) and configuration with which capture was taken as shown in Figure *CapturePreview* (page 121).

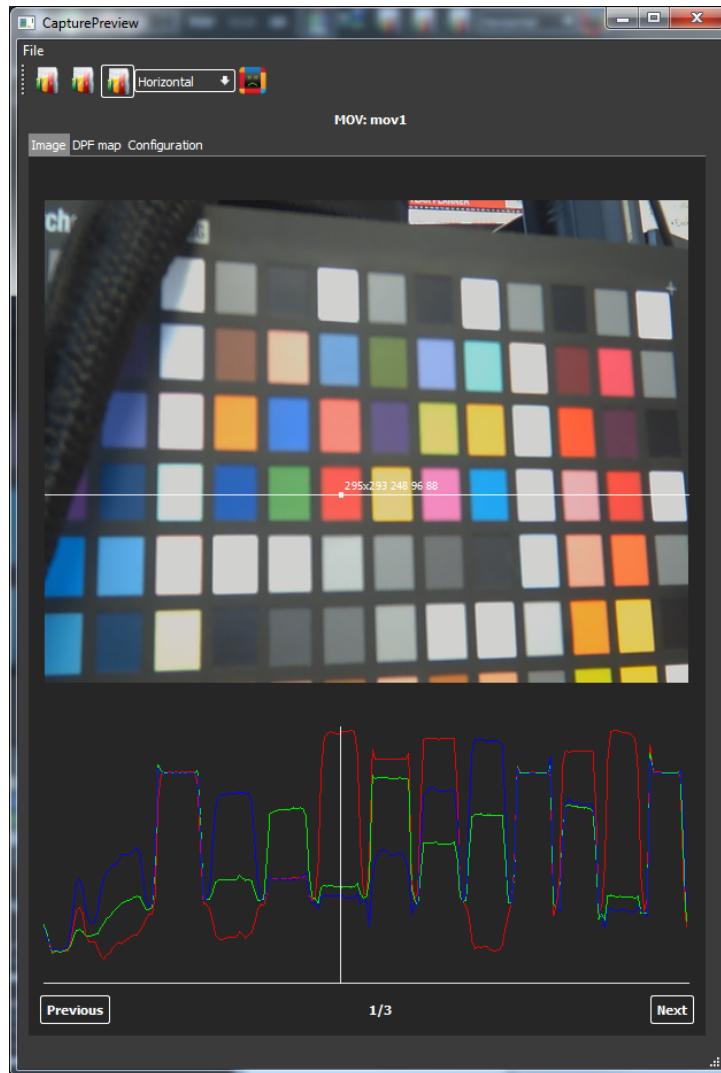


Fig. 5.36: CapturePreview

If capture contains many frames, “Previous” and “Next” buttons will appear at the bottom of “Image” tab. Those tabs allow switching between recorded frames. When displaying Histogram, Vectorscope, LineView or DPF while switching frames, displayed widgets will be automatically updated.

CapturePreview frames can be saved to disc using one of “File/Save Image” options.

Warning: Be aware that saving item containing multiple frames in JPEG format will result in creating multiple files on disc, each containing one frame. Saving the same item in FLX format will result in creating one file containing all captured frames.

DPF maps can be saved to disc using “File/Save DPF Map” option and then selecting which dpf map to save. Configuration can be saved to disc using “File/Save Configuration” option.

5.6.2 VisionTuning

The VisionTuning tool is used for offline calibration of the ISP for a given lens/sensor. It uses images of standard calibration charts, in the FLX format. The tool analyses the captures to determine the colour, lens shading and lateral chromatic aberration characteristics of the lens/sensor and calculates the required corrections. These are output in a format which can then be used by the *VisionLive and ISPC_tcp v1 (obsolete)* (page ??) tool (see *High level parameters* (page 283) document for details about this format).

Running the tool

The application requires no command line parameters. VisionTuning is composed of a window containing 4 tabs as below when starting. The File/ menu can be used to save the overall configuration. The View/ menu will allow the user to hide/show additional windows. The Action/ menu will allow you to perform actions on the opened tabs (e.g. add more CCM Tuning tabs). The Help/ About Vision Tuning option will let a user know the particular version of the tool.

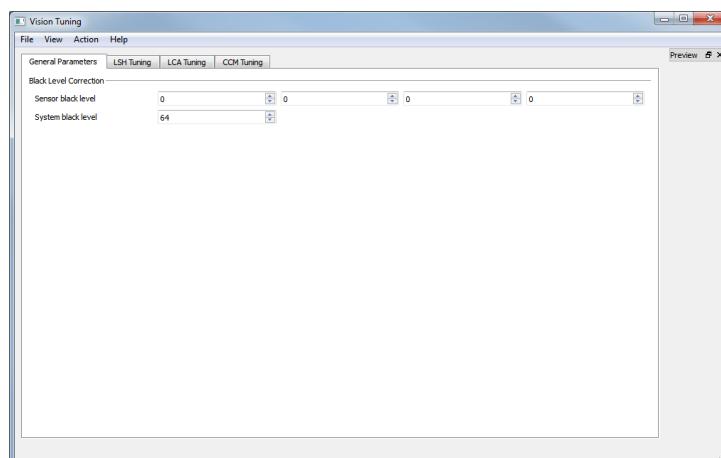


Fig. 5.37: General Parameters Tab

Running steps

This section describes steps to run the calibration process. To run the basic calibration it is expected to have:

- Information about the sensor black level per channel
- The desired system black level of the system (unified black level after correction of the sensor one)
- 1 FLX image per Colour Correction Matrix needed to be generated (Bayer image from sensor). It is possible to use multiple CCM.

- 1 FLX image per deshading matrix needed to be generated (Bayer image from sensor). This has to be a flat field illuminated as constantly as possible. It is expected to have only 1 LSH matrix generated per sensor.
- 1 FLX image to calibrate the Lateral Chromatic Aberration (Bayer image from sensor). This should contain detectable patterns (such as crosses).
- Measurement of desired light level parameters

Black level (BLC)

The first step will be to setup a sensor and system black level in the General Parameters tab (see [General Parameters Tab](#) (page 122)). The 4 sensor black levels are in CFA order.

Colour correction matrix (CCM)

The next step is the CCM Tuning. In the CCM Tuning tab click the **Setup** button and you will be given a dialog box like the one in [CCM Configuration setup dialog \(default\)](#) (page 123). Note that it is possible to add several CCM tabs using the Action menu or the **add CCM** button. It is recommended to have at least 3 different temperatures but many is possible.

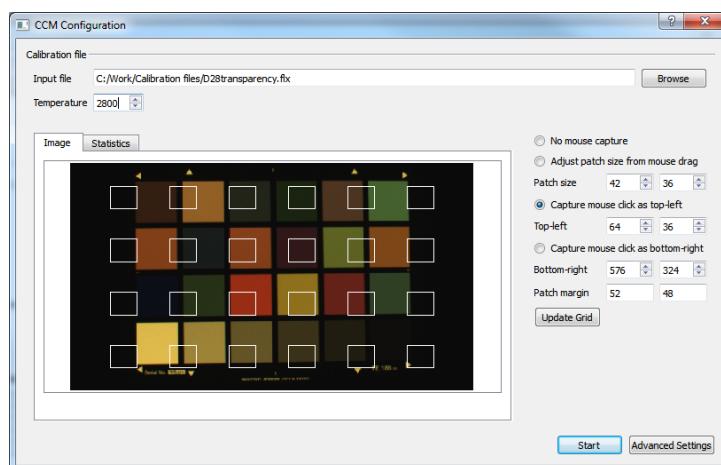


Fig. 5.38: CCM Configuration setup dialog (default)

Use the **Browse** button to select the FLX file for the desired colour temperature (it should contain a standard colour checker chart). The **temperature** value can be set after computation as it is not used for the computation.

The customer is expected to take the calibration images for the selected sensor following the [White Balance Correction Tuning](#) (page 131) section.

Use the right panel to define the behaviour of the mouse on the image preview to try to match the displayed squares with the selected chart. Values can be typed in but in such a case the grid will only update when the button is clicked. The easiest way to match the checker is to play with the “capture as top-left” and “capture as bottom-right” to try to match the grid (adjusting the patch size may be needed). See a desired result in [CCM Configuration setup dialog \(after selection\)](#) (page 124).

Once Start is pressed the computation will begin. After computation has finished CCM Tuning will be completed (see [CCM Tuning Tab](#) (page 124)).

The output tab will then show the corrected images, the CCM Matrix will have tabs to view the CIE94 Error for each colour patch. The algorithm attempt to minimise the average colour error.

The algorithm is a modified version of iterative refinement algorithm. This means the same parameters may produce different results. One can re-run the algorithm if the error seems too high or run it with longer iterations from the advance setups. It is not guaranteed that changing the advance setup will produce a better result.

The output preview also allows the viewing of results at different steps of the correction (only with white balance applied in the WBC output or only with the black correction in BLC output).

It is possible to choose to save just these parameters, or you can wait until you have setup other modules and save all at once.

Warning: Modifying the BLC after calibration CCM will invalidate the computed CCMs.

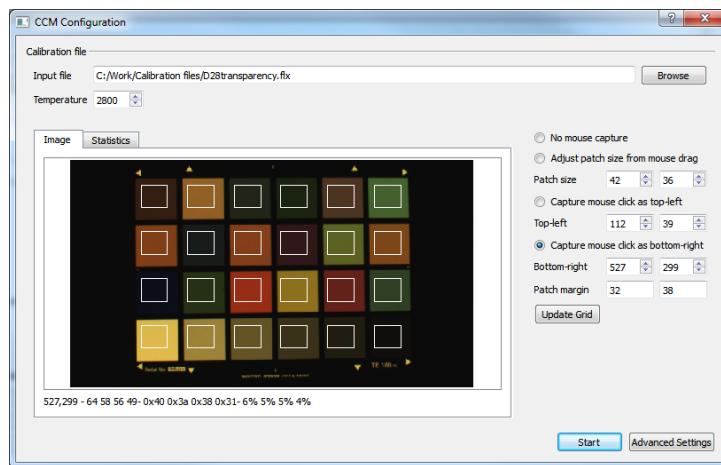


Fig. 5.39: CCM Configuration setup dialog (after selection)

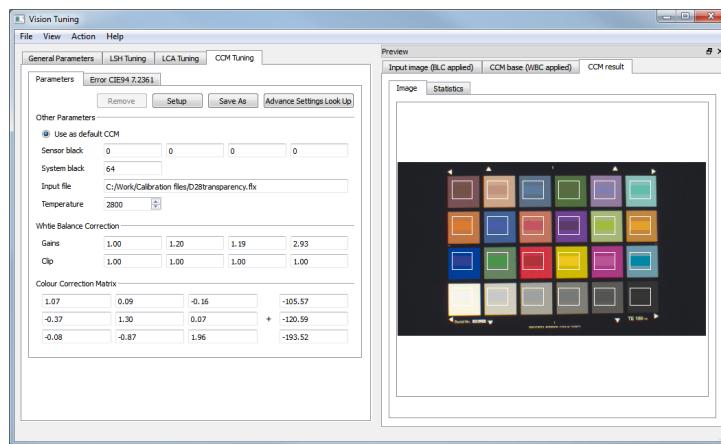


Fig. 5.40: CCM Tuning Tab

Advanced settings Here are some of the advanced settings:

Nb iterations Number of iterations to run the algorithm for.

Nb warm iterations Number of iterations at the beginning where the algorithm allows large changes. May impact the output a lot as these iterations produce CCM that can be very different from the best found.

Verbose logging Logging messages. No impact on image quality.

Display error curve Shows progress on improving accuracy, default settings will give low error. This means that more iterations won't help.

Normalise to 1 CCM matrix normalised to 1.

Use CI94 error metric If un-ticked will use the direct different with target as a metric.

Apply gamma correction to the displayed images No impact on the generated but affects the displayed image.

Temperature strength Strength of the changes between iterations.

Change probability Change the probability of modifying the best CCM to produce the next one.

The **WBC limits** tab limits gains applied to white balance and CCM, would only be used in extreme situations where trading off colour balance for high gain.

The **Target patches** allow the user to modify the information about the chart used and the value for each patch. The weights can be used to give more importance to some patches.

Lens shading (LSH)

The Lens Shading calibration has a setup dialog which, like the CCM tab, will ask for a source FLX files. Click on the **Setup** button to show it. Clicking on the setup button will replace the current calibration.

It is expected that the customer will produce their own calibration images following the *Sensor Lens Shading Tuning* (page 131) section.

The setup window is shown in Figure *LSH Configuration setup dialog* (page 126). The user is expected to first select an input images and then configure the other parameters:

Add Files Adds input FLX files for tuning LSH grids.

Add Temps. Adds input temperatures for tuning LSG grids for which FLX files were not provided. For those temps. grids are going to be interpolated based on provided FLX files.

Output filename prefix Prefix that output file names.

Note: File names of created grids are going to be generated automatically.

Tile size This is the size of tile (or points between interpolation) used by LSH algorithm. The default value is 8 CFA which is the most accurate, but will provide the largest file to be saved into memory. This can be changed to 128 CFA which is least accurate, but gives smallest file.

Warning:	This cannot be changed once processed. To generate another file the process has to be done another time.
-----------------	--

Algorithm Can be fitting or direct curve, a fitting curve will have been smoothed. Whilst the direct curve will not.

It is recommended to use fitting curve algorithm.

When all parameters have been provided the application will analyse the image and produce a lens shading configuration to flatten the image. The result tab should display slices of the computed grid per channel (see *LSH Tuning Tab* (page 126)).

Warning: Changing the BLC will invalidate the computed LSH grid.

Use the Save As button to save only the computed grid and its associated setup file. (This option is not active when LSH configuration is accessed from VisionLive).

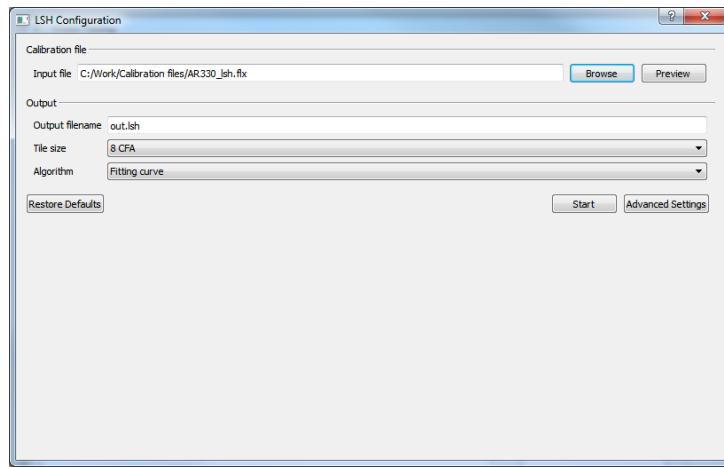


Fig. 5.41: LSH Configuration setup dialog

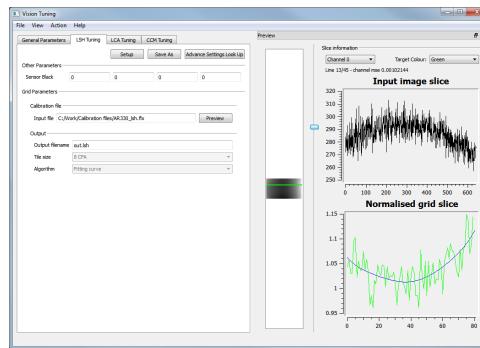


Fig. 5.42: LSH Tuning Tab

Advanced settings The LSH grid is generated in floating point format. The HW uses a differential encoding on fixed point values to load the matrix and it is the responsibility of the driver to convert the grid the tool generate to the HW format. However the HW has several limitations that the tool should take into account when generating a grid as exposed in the advanced setting:

Margins A margin per side of the image can be used to remove black pixels. The margin will reduce the size of the image used by the algorithm as a result the output grid may cover a smaller resolution.

The margin can be specified for Left, Top, Right and Bottom sides separately.

Note: The preview image can be used to see the cropped image (the colour of the rectangle can be changed) but it is expected that the customer will know how many black pixels are configured in their calibration image.

Input smoothing This is a smoothing factor that can be used to remove noise from the input image.

Maximum gain The HW has a maximum gain that can be used at any point in the matrix. The default value should be the appropriate number of the HW 2.

This value should not be modified and the box should be left enabled.

Maximum line size This can be used to limit the maximum size in Bytes a line of the grid can fit in (encoded in the HW differential format).

This value cannot be modified and the box should be left enabled.

Note: This limitation **could** be turned off in some HW versions if the LSH matrix limitation is not present.

Warning: This is an estimate of the size following the same steps the driver will to convert the grid to the HW format.

Enforce max bits in HW format This can be used to limit the maximum number of bits (unsigned) that the differential encoding will use.

Warning: This is an estimate of the size following the same steps the driver will to convert the grid to the HW format.

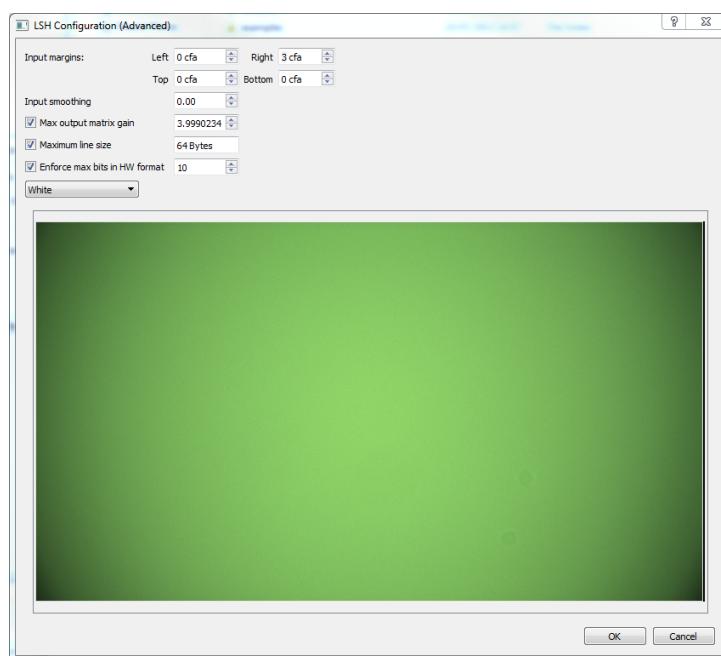


Fig. 5.43: LSH Advanced Parameters dialog

Lateral chromatic aberration (LCA)

The LCA tab requires an input image for Lateral Chromatic Aberration auto-calibration. It is expected that the customer will take such image following the *Lateral Chromatic Aberration Tuning* (page 132) section.

Clicking the **setup** button will display the menu to configure the algorithm as shown in Figure *LCA Configuration setup dialog* (page 128).

Select an appropriate image and validate. The setup dialog will automatically locate the crosses in the image provided for LCA calibration. The feature size option can help to detect the features that may not have been found. The tool requires at least 20 features to be discovered to proceed.

The centre of the lens (in CFA) should also be selected at this stage. By default it is the centre of the image.

Use the start button to generate the LCA correction. The preview tab will display the detected aberration per channel in comparison with one of the Green channels. Use the display amplifier value to increase/decrease the size of the displayed arrows.

The user can save just the LCA polynomial use the **Save As** button.

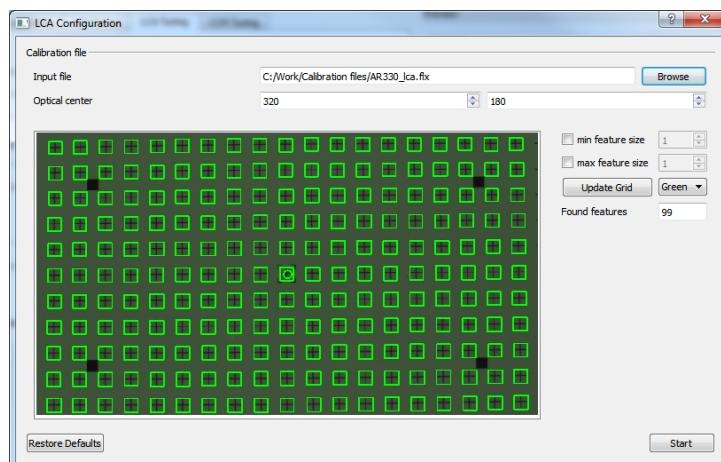


Fig. 5.44: LCA Configuration setup dialog

Saving results

Use the File/Save All to save the result as a single Felix Setup Args file (and potentially an LSH grid if applicable) that can be used by other tools (e.g. *VisionLive* and *ISPC_tcp v1 (obsolete)* (page ??) or *ISP Control test: ISPC_test* (page 61)).

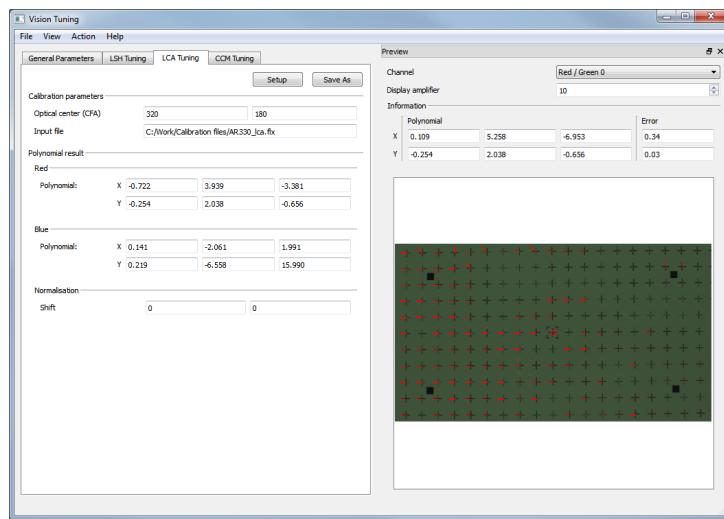


Fig. 5.45: LCA Tuning Tab

Chapter 6

User Tuning Guide

This section shows how to use the *VisionLive* and *ISPC_tcp* (page 87) and *VisionTuning* (page 122) tools to tune the V2500 ISP.

It is assumed that the reader is familiar with the VisionLive and VisionTuning tools.

6.1 Tuning work-flow

The tuning work flow would be the following:

- Black level calibration
- AWB/CCM calibration
- LSH calibration
- LCA calibration
- Primary Denoiser tuning
- DPF tuning
- Secondary denoiser/Sharpening tuning
- Tonemapping tuning
- MIE (optional)

It would be expected that the customer would perform a wide suite of objective/subjective tests that may require tweaks to the original settings, this is expected in the normal tuning flow for an ISP.

6.2 Sensor BLC tuning

This method will be done by capturing a black frame from the image sensor.

1. Ensure no light is getting into the sensor (cover with cloth or lens cap).
2. Turn off AAA, LSH and ensure white balance gains are 1 then manually set the exposure time to 30ms/1x gain.

3. Setup to take DE image CI_INOUT_FILTER_LINESTORE and set the format to match the sensor bit depth in the *Output tab* (page 100)
4. In *BLC tab* (page 102) set *sensor black* and *system black* to 0.
5. Apply all these settings and start to take images using Liveview, it should be a black image. View the histogram of the image, it should be a histogram which is a positive value. A histogram for each colour channel should be seen.
6. Start by adjusting *sensor black*, the aim is to get all four “black histograms” centred on 0. When this is achieved *sensor black* has been set

Note:

It might be that different colour channels have different values for *sensor black*.

Warning: Once tuned the *system black* should be set back to 64.

6.3 White Balance Correction Tuning

The white balance and CCM correction is calibrated using the VisionTuning *Colour correction matrix (CCM)* (page 123) tool. This is the recommended process for taking CCM calibration image. The camera should be setup so that it is illuminated with the Macbeth chart, the white patch of the chart should be ~70-80% LSB.

1. Select Bayer extracted output mode at the CI_INOUT_BLACK_LEVEL point, the bit-depth should match the image sensor.
2. Turn off AAA and LSH.
3. Manually change gain to 1x.
4. Select a colour temperature to calibrate for.
5. Manually change exposure time until the middle of the image is ~70-80% LSB.
6. Capture and save image.
7. Change the colour temperature and repeat from the exposure step.

The saved images can then be used with the VisionTuning tool.

6.4 Sensor Lens Shading Tuning

Tuning for the LSH is done using VisionTuning *Lens shading (LSH)* (page 125). This is the recommended process for taking a LSH calibration image. The camera should be setup so that it is illuminated with a uniform light source (>95% uniform).

1. Select Bayer extracted output mode at the CI_INOUT_BLACK_LEVEL point, the bit-depth should match the image sensor.
2. Turn off AAA
3. Manually change gain to 1x.

4. Manually change exposure time until the middle of the image is ~70-80% LSB.
5. Capture and save image.

The saved image can then be used with VisionTuning to generate the deshading matrix.

6.5 Lateral Chromatic Aberration Tuning

The LCA calibration is done using the VisionTuning *Lateral chromatic aberration (LCA)* (page 128) tool. This is the recommended process for taking LCA calibration images. The camera should be setup so that it is illuminated by the TE-251 chart from image engineering, the white of the chart should be ~70-80% LSB.

1. Select Bayer extracted output mode at the CI_INOUT_BLACK_LEVEL point, the bit-depth should match the image sensor.
2. Turn off AAA
3. Manually change gain to 1x
4. Manually change exposure time until the middle of the image is ~70-80% LSB.
5. Capture and save image.

The saved image can then be used in the VisionTuning tool.

6.6 Primary Denoiser Tuning

The tuning for the denoiser can be done using the VisionLive *Noise tab* (page 107).

The *primary denoiser strength* is considered a preference parameter for the user to tune to their personal preference. This could be done using objective and subjective methods:

- Objective: Using a chart that can be used for SNR and detail (sharpness) and sweeping the value until the best compromise for both SNR and detail.
- Subjective: This would be taking a number of images in a range of real scenes with different strength values until the preferred value was obtained.

Note: The secondary denoiser and sharpening modules are setup later in the process and information is available in the *Sharpening and Secondary denoiser Tuning* (page 134) section.

6.7 Defective Pixels Tuning

The DPF tuning can be done using the VisionLive *DPF tab* (page 108). The tuning is a multi-stage process. This section will focus on the live detection tuning, the process is similar for map creation.

The calibration should be in the following order:

1. Dark calibration
2. Mid-light calibration

3. Lab validation

6.7.1 Dark calibration (stage 1)

This calibration stage is looking specifically for bright or hot pixels.

1. The camera should be covered using a lens cap or cloth (it is critical that no light reaches the sensor).
2. Ensure that all AAA and Auto TNM are turned off
3. Manually expose the image to the limit of the sensor mode (e.g. 30 or 60 ms) and set the gain to max value.
4. Turn on *HW detection*.
5. Set *threshold* and *weight* both to 0 (this should now be correcting lots of defects, it should actually be over-correcting).
6. Increase the threshold in steps of 1 to about 10 and determine the best value (checking image/number of pixels corrected).
7. Increase the weight in steps of 0.5 to ~ 2 and determine the best value (checking image/number of pixels corrected).

6.7.2 Mid-light calibration (stage 2)

This calibration stage will check to see if any dark or cold pixels exist and determine if the tuning in stage 1 can resolve them.

1. The camera should be in a scene that is a flat field (similar setup used for lens shading).
2. Ensure that all AAA and Auto TNM are turned off
3. Manually expose the image to get $\sim 70\%$ limit of the output and keep the gain at 1x.
4. Use settings from stage 1, if no dark cold/pixels seen then this stage is complete.
5. If dark/pixels are seen then change weight and threshold to removed them.

6.7.3 Lab validation (stage 3)

This stage has AAA running and uses an image with fine detail (resolution chart/text from book) and checks that over correction does not occur.

1. The camera should be in a scene with a resolution chart or text.
2. Turn AAA and Auto TNM on
3. Use DPF settings from stage 2
4. Check that fine detail/resolution chart does not show signs of over-correction (false correction).
5. If it does then change DPF settings to remove it.

Note: Final validation would include running the settings in real life scenarios to ensure that satisfactory performance is achieved.

6.8 Sharpening and Secondary denoiser Tuning

This section will detail how to setup the Sharpening and Secondary denoiser. The parameters are available in the VisionLive *Noise tab* (page 107).

6.8.1 Sharpening tuning

The starting parameters for sharpening tuning would be:

```
SHA_DENOISE_BYPASS 0      # not in GUI - is default
SHA_DETAIL 1.0            # in Sharpening
SHA_DN_SIGMA_MULTIPLIER 0.1 # secondary denoiser strength
SHA_DN_TAU_MULTIPLIER 1    # secondary denoiser edge avoidance
SHA_EDGE_OFFSET 0          # not in GUI - is default
SHA_EDGE_SCALE 0           # not in GUI - is default
SHA_RADIUS 1.5             # in Sharpening
SHA_STRENGTH 0.5           # in Sharpening
SHA_THRESH 0                # in Sharpening
```

Warning: For all sharpness tuning it is assumed that the lens has been optimally focussed.

1. The scene for tuning sharpness would typically be a lab scene with charts and real objects. A chart like ISO12233 is a good choice and real objects like dolls, wool (items that have fine texture).
2. Change the *sharpening radius* in 0.5 increments until it gets to 2.5 and determine optimal value.
3. Change *sharpening strength* in 0.1 increment until reasonable results at edges are achieved.
4. Change *sharpening details* in 0.1 decrement (if necessary) to reduce noise in flat regions of the image without removing the sharpening at edges.

Note: After tuning these values it should be validated over a number of scenes/light levels to understand if further fine tuning is required.

6.8.2 Secondary denoiser tuning

The secondary denoiser has got two controls and assuming that sharpening has been done before the process would be as below:

1. Increase the *secondary denoiser strength* (Sigma multiplier) to increase noise reduction.
2. Increase the *edge avoidance* (Tau multiplier) to perform more edge avoidance.

6.9 Tone-mapping Tuning

This section illustrates how to tune the TNM block using the VisionLive *TNM tab* (page 110). Information about the algorithm is available in the *Algorithm overview* (page 226) section and *Automatic Tone Mapping Control (TNMC)* (page 272) section.

6.9.1 Local tone mapping

For illustration purposes we will explain how to setup the local tone mapping starting with an image with the local tone mapping settings at maximum strength. The Figure *Global tone mapping only (LHS) vs Global + full strength tone mapping (RHS)* (page 135) shows the pipeline output when applying just a global tone map and maximum local tone mapping strength.

Setting the local tone mapping values at maximum strength will produce unpleasant looking pictures, enhanced noise, haloing and other artefacts. In the following subsections we will explain how to set it up so those undesired effects are avoided.



Fig. 6.1: Global tone mapping only (LHS) vs Global + full strength tone mapping (RHS)

The setup process is as follow:

1. Have Tone Mapper and Auto Tone Mapper Curve items enabled.
2. Put camera in a static scene (something like a test chart in light booth).
3. Set Local Tone Mapper parameters to maximum values:

```

TNM_WEIGHT_LOCAL 0.999 # local mapping weight
TNM_WEIGHT_LINE 0.999 # line update weight
TNM_FLAT_FACTOR 0.999 # flattening
TNM_FLAT_MIN     0.999 # flattening minimum

```

The result should be similar to the Figure *Full strength tone mapping* (page 136).

4. Reduce the *line update weight* value until the problems caused by vertical abrupt changes in the image are smoothed. It is not necessary to make them disappear completely at this step as that will be done by tuning rest of the parameters.

The Figure *Line update reduced example (0.075)* (page 136) shows an example.

5. Increase the amount of flattening in the image by reducing the *flattening minimum* value. With this parameter the minimum flattening is applied to the whole image regardless of the local statistics. This will reduce exaggerated local enhancement and also the artefacts caused by abrupt vertical changes in the image.



Fig. 6.2: Full strength tone mapping



Fig. 6.3: Line update reduced example (0.075)

The Figure *Flattening minimum reduced example (0.35)* (page 137) shows an example.



Fig. 6.4: Flattening minimum reduced example (0.35)

6. Reduce the *flattening* will address local contrast in the flatter regions of the image. Reducing this value also helps to avoid noise enhancement.

The Figure *Flattening reduced (0.1)*. (page 137) shows a result of flattened image. Note that a small reduction in the local contrast enhancement in the grey background, the wooden board and some of the checkers is visible.



Fig. 6.5: Flattening reduced (0.1).

7. The final step would be to get the desired blending value for the local and global mapping results using *local mapping weight*.

The Figure *Final result with local mapping weight reduced (0.35)* (page 138) shows a final result example.

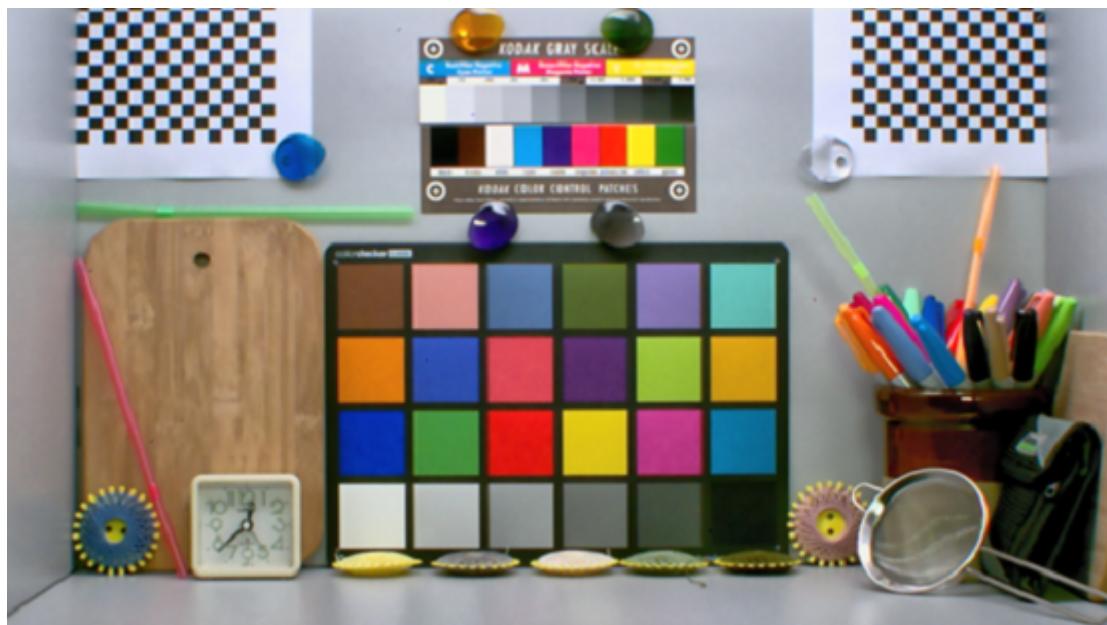


Fig. 6.6: Final result with local mapping weight reduced (0.35)

At the end of the tuning for this particular image the settings are:

```
TNM_WEIGHT_LOCAL 0.35 # local mapping weight
TNM_WEIGHT_LINE 0.075 # line update weight
TNM_FLAT_FACTOR 0.1 # flattening
TNM_FLAT_MIN     0.35 # flattening minimum
```

Figure *Final result showing improvements*. (page 138) shows the improvements.

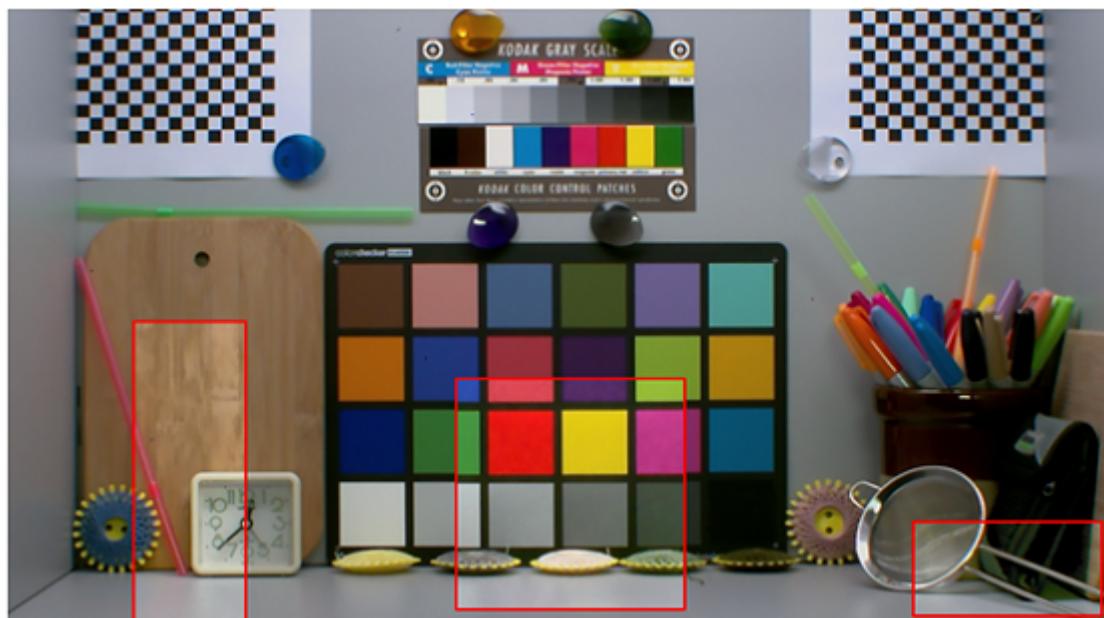


Fig. 6.7: Final result showing improvements.

6.9.2 Colour

The setup process is detailed as an explanation in the algorithm section about the *Colour saturation* (page 227).

1. As an starting point for setup both *colour saturation* and *colour confidence* can be set to 1.0.
1. Assess if the output image is more/less saturated than the original (adjust *colour saturation* in accordance).
2. Check if there are colour artefacts in very dark or light regions of the image (and adjust *colour confidence* to remove them).

6.10 Image Enhancer Tuning

The MIE module allows the configuration of some colours to be selected to stand out as explained in the *Algorithm overview* (page 217). This section will explain how to use the *MIE tab* (page 111) to tune the memory colour.

It is highly recommended to read the algorithm information before proceeding to the tuning steps.

Note: This is an optional step of the tuning.

1. Ensure MIE and TNM is turned off.
2. When scene is set as required, ensure live view is on YUV and capture an image (or pause live view).
3. Pick a pixel which is the colour to be changed, this should give three values in the bottom of the image preview.
4. In the MIE tab select import value, this will put values into Luma slice min/max and Chroma centre Cb/Cr.
5. Set all *luma gains* to 1
6. Change the value of extent to a smaller value, suggest to start with 0.1 (this may need to change further).
7. Turn on selected colour and turn on relevant MC. The user should now see selected pixels turn black.
8. If less pixels are required then reduced extent or if more are required increase extent.
9. Once happy then turn off selected colour and change the values in output HSBC.
10. Repeat for other colours.
11. When process is complete turn on TNM.

Chapter 7

Capture Interface

This section is stating some additional information about the Capture Interface (CI). The CI is the lowest level of the V2500 Software Pipeline (namely the user and kernel side driver) and deals with the Hardware (HW) and registers.

The CI folder also contains the external Data Generator driver (DG) that is the equivalent of the CI used to fake a sensor when running tests on Imagination's FPGA or against the C Simulator (CSIM).

The user-side CI driver also defines the Module Configuration (MC) that simplifies the configuration of the HW by abstracting some information (such as the registers precision when they represent real numbers instead of integers).

Note that V2500 may be referred as *felix* in this section (its internal name).

The reader is expected to know about the V2500 architecture and to search more detail information about the driver in this chapter. The initial section can also help learning about some of the design objectives of the CI layer. However the behaviour of the HW should not be explained in that document and the interaction between CI and other SW elements will not be detailed.

7.1 Design Choices

7.1.1 Naming convention

The Capture Interface library is composed of several sub-libraries. A single naming convention is nonetheless used (using capital letters for the library and camel-case for the object-operation block):

```
<library namespace>_<object name><operation>
```

e.g.: CI_PipelineCreate()

library CI, object Pipeline, operation Create

7.1.2 Folder organisation

It is not obvious how the CI folder is organised. The driver depends on several libraries, some that are shared among several Imagination Technologies projects and some that are not. An

attempt of separating what is shared and what is not was made to try to make cross-project building easier in the future.

Multi-project libraries are directly under `DDKSource/CI`:

- Target Abstraction Layer (TAL – `target` folder) is used to access registers and device memory (it has many debugging features)
- IMG libraries (`imglib` folder) are several internal libraries mostly used by the TAL or other shared components
- OSA is an abstraction layer for the operating system used in the TAL for synchronisation in debug mode
- `imgmmu` is a shared library in IMG for the management of the page tables of the MMU HW
- `transif` is a shared library in IMG used to abstract the connect to the simulator when using development drivers

Multi-project libraries that are not directly under CI but used by the low-level driver (in `DDKSource/common` folder):

- IMG includes (`DDKSource/common/img_includes`) are several include files used in internal libraries to define OS independent data-types and definitions (like memory allocation). This copes with user and kernel difference for GNU/Linux as well.
- Felix common (`DDKSource/common/felixcommon`) that contains pixel format transformations
- Linked list (`DDKSource/common/linkedlist`) that contains a linked list implementation that works in both kernel and user-space

The driver related code is under the `DDKSource/CI/felix` folder:

- `regdefs` contains the register definitions files specific to V2500 HW
- `felix_lib` is the actual driver (user and kernel side)
- `driver_test` is the test application for `felix_lib` see *Capture Interface test: driver_test* (page 49)

Finally the actual driver code is in `DDKSource/CI/felix/felix_lib` folder:

- `data_generator` contains the user and kernel side of the external DG driver
- `kernel` contains the kernel side of the CI
- `user` contains the user side (CI and MC)
- `testdata` has a few images used for unit-testing
- `test` contains the unit tests for the library when using it fully in user-side
- `utils` contains the code used for the simple DPF write map to read map converter (see *Defective Pixel output converter: dpf_conv* (page 82)) and an IOCTL printer.

7.1.3 Libraries and namespaces

The CI layer is in fact composed of several “libraries” that interact with each-others as shown in *The Capture Interface eco-system* (page 142).

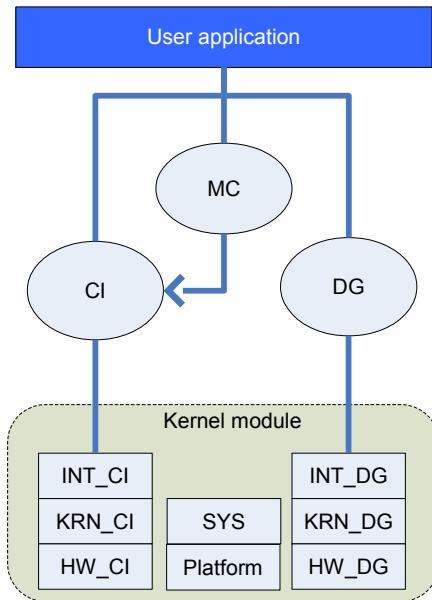


Fig. 7.1: The Capture Interface eco-system

In user space:

- Capture Interface (CI) is the library to access the V2500 driver
- Module Configuration (MC) is the very thin library on top of CI to help configuring the CI modules
- Data Generator (DG) is the library to access the V2500 external Data Generator driver

In kernel space both CI and DG are separated in several namespaces but built in the same kernel module

- Interaction (INT_CI, INT_DG) functions called from user-space through system calls (`ioctl()`, `memmap()`, ...)
- Kernel (KRN_CI, KRN_DG) functions called from kernel-space and that handle the logic of the drivers
- Hardware (HW_CI, HW_DG) functions that deal with the register or device memory
- Device (DEV_CI, DEV_DG) functions that are called by the OS to handle system calls on kernel-side
- System (SYS) is a mini library that abstracts some of the operating system behaviour. Its goal is to mimic the GNU/Linux kernel interface that can be implemented for other OS in the future (and also used for debug purposes in user-space using posix). For example locks are implemented in that layer.
- Platform (Platform) is used to represent platform specific functionalities for the device management and allocation of memory. More information about these functions is available in the *Platform Integration Guide* (page 27) chapter.
- It is also possible that some internal local function will be prefixed with `IMG`, showing that they should not be called outside of their file.

7.1.4 Fake and Real driver

The driver is developed while running against the C Simulator (bit accurate HW simulator) and most of the time this is done fully in user-space (to allow easier debugging). This mode is called fake because the device does not really exist. Some mechanisms were created to replace what is normally provided by the Linux Kernel or by the HW.

- Fake IOCTL calls (that directly map to kernel side – see *User/Kernel interactions* (page 165))
 - Fake interrupt handling (that does not stop other threads from running)

The driver is therefore considered *real* if it is running as a kernel module or compiled as part of the GNU/Linux kernel.

This is implemented in the `SYS_IO` namespace (in user-space), composed one function per system call:

- SYS_IO_Open() abstracts `open()`
 - SYS_IO_Close() abstracts `close()`
 - SYS_IO_Control() abstracts `ioctl()`
 - SYS_IO_MemMap2() abstracts `mmap2()` or `mmap()`
 - SYS_IO_MemUnmap() abstracts `munmap()`

When running the Fake implementation a structure is provided when calling `SYS_IO_Open()` with one pointer to function per system call. These functions will be called instead of the real system calls and their prototypes are expected on the kernel side. The interaction is described in [Fake System-call](#) (page 143) and [Real System-call \(GNU/Linux\)](#) (page 143).

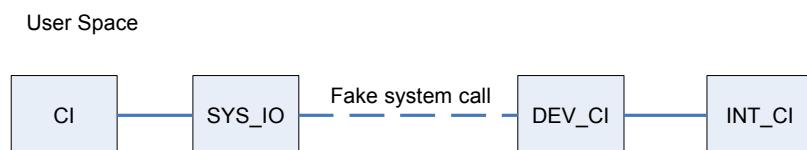


Fig. 7.2: Fake System-call

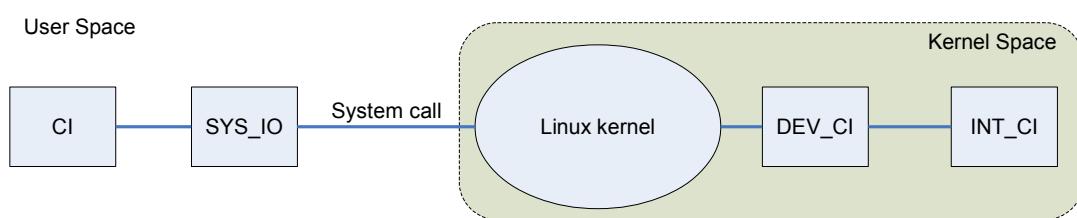


Fig. 7.3: Real System-call (GNU/Linux)

7.2 Limitations

7.2.1 Identifiers

Each “connection” to the driver (i.e. each call to `open()` through `CI_DriverInit()`) can only have a limited number of “elements”:

- Each created Pipeline object has a unique identifier relative to its parent Connection as an `int`.

This identifier relates the `CI_PIPELINE` given to the user to the `KRN_CI_PIPELINE` one stored in the kernel. Identifiers are stored in a user-space private structure: `INT_PIPELINE`.

- Each buffer mapped from kernel-side to user-side has a unique identifier relative to its parent Connection as an `int`.

This identifier allows the user-space to associate the `KRN_CI_BUFFER` and `KRN_CI_SHOT` to a `CI_SHOT` and the memory it needs through some user-space internal structures: `INT_BUFFER` and `INT_SHOT`.

Signed integers are used for both unique IDs, this means that if more than `INT_MAX` objects are created in a single connection the kernel driver may confuse two elements. Overflow of signed integer ID is not allowed since it is very unlikely to happen (the driver will fail if it happens).

Typically `off_t` is used when doing sharing memory from kernel-side to user-side as it is the parameter to `mmap()`. However to support Android versions which may have a 64b kernel and 32b user-space we chose to use that offset as an `int`.

For example the maximum values for a GNU/Linux system (`PAGESIZE` is 4096):

System	<code>INT_MAX</code>
32b	2,147,483,647
64b	2,147,483,647

Therefore the maximum number of Pipeline object to be created per call to `open()` is a bit more than 2 billion. More than one buffer is allocated for a single frame capture (e.g. encoder, display, DPF, HDR Extraction, statistics) but even if we assume we need 10 “mappable” elements per frame to capture it is still more than 200 million unique buffers per call to `open()` (in a world where a unique buffer would be allocated for every frame of a video at 60 FPS it is a bit less than 1,000 h of video).

7.2.2 Linux Kernel Trouble

Known trouble with some Linux kernel libraries used by the driver.

Memory band management (`genalloc`)

Linux kernel 3.2.0 contains defects in `genalloc.c` triggered when using the MMU Heap implementation.

- Allocating a bloc using `gen_pool_alloc()` returns 0 on error, which is problematic when the 1st valid address is 0.

This is solved by shifting the address range by a page when the pool is created or by not using the 1st band of memory.

- In kernel versions prior to 3.12 gen_pool_add_virt() was computing as:

```
chunk->end_addr = virt + size; // (size_t size, phys_addr_t phys)
```

On 32b machines this could result in an overflow if using the last page of the system.

The V2500 driver's virtual heaps therefore avoid the 1st and last page of the device MMU.

7.3 HW Modules and resources

This section highlights how the driver handles the HW resources and modules.

7.3.1 HW Resources

The driver considers that the HW can provide several “resources” which are not shareable between several processes. These resources are:

- HW Context (the HW has CI_HWINFO::config_ui8NContexts of them)
- HW Gaskets (the HW has CI_HWINFO::config_ui8NIImagers of them)
- If present HW Internal Data Generator (the HW has CI_HWINFO::config_ui8NIIFDataGenerator of them)
- If present HW External Data Generator (the HW has DG_HWINFO::config_ui8NDatagen of them)
- The linestore is shared between HW Context and therefore handled globally
- The GMA look-up table is shared between HW Context and therefore handled globally

Note: CI_HWINFO is accessible in user-space as part of CI_CONNECTION::sHWInfo.

7.3.2 Line-store

The line-store configuration is used by some HW module to know the number of lines to store for the computation of the module. The driver is responsible for setting up the size of the stored lines. The HW expects 1 line-store start position for each context. The driver stores additional information such as the size (that is either until another line-store start is found or to the maximum possible for this context).

It is likely that the line-store should not be modified often as its configuration certainly depends on the size of the sensors.

HW Knowledge

- The maximum size of a context is defined in a register (accessible in CI_HWINFO structure)

This maximum size is the maximum size from the start point (it can be anywhere in the line-store buffer).

- The total size of the line-store buffer is the maximum size across all context's sizes (usually the context 0 is the biggest)
- The line-store **cannot** be changed whilst a context is capturing
- The line-store configuration is stored globally; this means that any user connected to the device can ask to change it.
- Modifying the line-store in the driver is expensive (all contexts are locked in the driver).

If modifications have to be made on the line-store it is recommended to update it from the driver first (to get modification other could have done).

- Updating a local copy of the line-store is relatively cheap but still involves locking (each context is locked one at a time just to copy the configuration values).
- Initially the line-store is set-up to be shared equally among contexts.
- The `LS_BUFFER_ALLOCATION` register is written to when the capture is started (before setting up the `CONTEXT_CONTROL`). At this point the value is converted from pixels to “pairs of pixels” as the register expects.
- The driver can refuse a capture start if there is not enough room in the HW linestore to let it start or if the context position given by user-space overlaps with one an already started context.

Note: Locking is software locking only.

HW Context and CI_PIPELINE object

In user-space several `CI_PIPELINE` objects can be created. The number is not limited by any HW features. However when registered a Pipeline object selects on which HW context it will require to run (`CI_PIPELINE::ui8Context`). When starting the capture the kernel-module will try to associate a Pipeline object to its desired HW context. Only 1 Pipeline object can be running on a HW Context so the capture may fail to start if it is already used by another object (from the same process or a different process).

HW Gasket and CI_GASKET object

The user-space CI library provides a `CI_GASKET` object that is used to configure the gasket when starting the sensor. It has to be reserved and only 1 connection (call to `open()`) can hold a gasket at any given time.

HW Internal Data Generator and CI_DATAGEN object

The user-space CI library provides a `CI_DATAGEN` object that can be used to configure the internal Data Generator if it is available (available if `CI_HWINFO::eFunctionalities` has the `CI_INFO_SUPPORTED_IIF_DATAGEN` bit set).

As per the Pipeline objects several can be created for the available HW but only 1 can run per HW resource.

HW External Data Generator and DG_CAMERA object

The DG_CAMERA objects are similar to the CI_PIPELINE. There is not limitation on how many can be created by user-space but they can only be associated to 1 running HW data generator. They are used to convert a given file to the memory format expected by the HW module.

7.3.3 Gamma Correction

This section is about the Gamma Correction Look-up table (also called Gamma curve). It is recommended to read the TRM section about Gamma correction to fully understand this HW block.

HW Knowledge

The Gamma Look Up table register bank the V2500 is shared among all the contexts. Therefore it is not recommended that the user changes the values while any context is running (one may use an incomplete curve). Each context can however choose to enable the usage of this table individually for each frame (part of the load structure).

By design the GMA LUT is 63 points to represent the whole Gamut range. Because the HW can support extended Gammut the signal range is [-0.5 ; 1.5]. The HW block handles 11 bits, therefore the whole range is [0 ; 3071].

The GMA LUT registers are representing the whole input on a non-evenly spaced subsampling of the curve:

Input Range		Knee points	Points
0	31	2	0; 15
32	63	4	32; 39; 47; 55
64	127	4	64; 79; 95; 111
128	255	4	128; 159; 191; 223
256	511	8	256; 287; 319; 351; 383; 415; 447; 479
512	1023	8	512; 575; 639; 703; 767; 831; 895; 959
1024	3071	32	1024; 1087; 1151; 1215; 1279; 1343; 1407; 1471 1535; 1599; 1663; 1727; 1791; 1855; 1919; 1983 2047; 2111; 2175; 2239; 2303; 2367; 2431; 2495 2559; 2623; 2687; 2751; 2815; 2879; 2943; 3007

Driver Limitations

The driver was designed with the assumption that the GMA curve is representing a standard and, because it is shared amongst all HW contexts, is not intended to be changed unless trying calibration for different standards.

It would have been technically possible to change the GMA curve every-time a context requests a capture to start but because of the assumption that the standard curve does not change it would have involved unnecessary checking.

The driver initialises the default curve at the creation of the KRN_CI_DRIVER object when `insmod` is invoked. It is possible to change which default curve is used at that time (see [Getting](#)

Started Guide (page 17)). See *Gamma Look-Up table customisation* (page 148) to know how to change the default and values.

The CI interface in user space allows the retrieving of the current curve but also the override of the curve. The current value of the Gamma LUT can be read in the given CI_CONNECTION. The value can be updated using CI_DriverGetGammaLUT() (get updated version from the kernel-side) or CI_DriverSetGammaLUT() (propose LUT from the user-side).

Note: It is not recommended that the user changes the Gamma curve using that method (changing the default one is safer).

Warning: The Gamma LUT **cannot** be changed while any context is capturing, therefore proposing a new table may not work all the time.

The driver should have a curve for the BT709 and the sRGB standards. These curves were generated from the standard *formulae* and modified to limit the error of the approximation when converted to our register format (see *Default Gamma Look-Up tables (BT709 and sRGB)* (page 148)). The values are available in KRN_CI_DriverDefaultsGammaLUT(), in `kernel_src/ci_internal.c`.

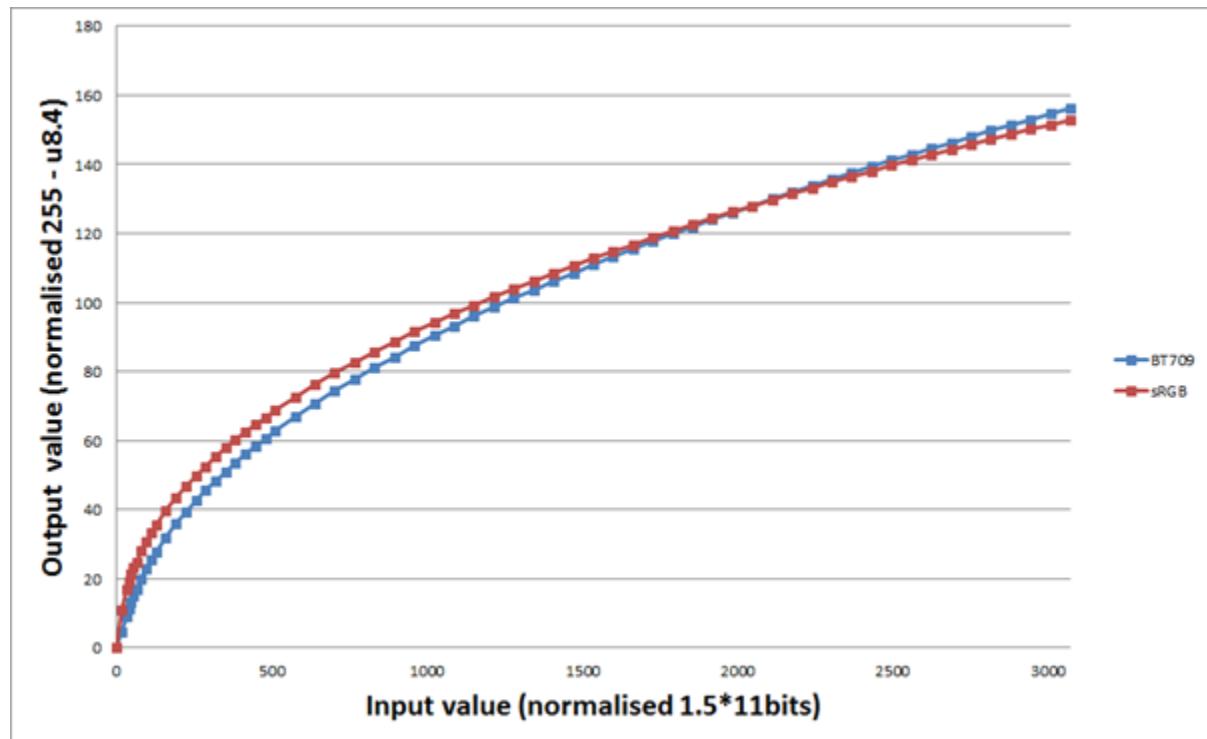


Fig. 7.4: Default Gamma Look-Up tables (BT709 and sRGB)

Gamma Look-Up table customisation

This section contains information about the GMA LUT registers and how to change the default curve or add other curves to the driver.

Change the default curve

The default used curve is specified in ci_kernel.h with the macro CI_DEF_GMACURVE. If this value is change make sure KRN_CI_DriverDefaultsGammaLUT() can support the new value or every `insmod` without the gammaCurve parameter will fail.

Add other curves

Using the register values from the HW documentation it should be possible to generate alternative curves and add them to KRN_CI_DriverDefaultsGammaLUT() in `kernel_src/ci_internal.c`. The current implementation uses the same values for the Red, Green and Blue channels but the values could be different.

The *Vision Live tool* (page 118) provided can help generate register values.

7.3.4 Deshading Grid (Lens shading)

The deshading grid (also called matrix) is a part of the Lens Shading module (LSH) that is loaded from device memory. It is stored in a specific format described in the TRM.

Note: It is important to read and understand the TRM about the LSH block memory format to fully comprehend the following section.

The details about are covered in the TRM:

- relationship between sensor resolution, tile-size and grid size
- memory format used by the HW (with bits per difference, line size and stride)
- skip and offset values to handle decimation and cropping from the IIF

The module also has size limitations that are verified in CI_ModuleLSH_verif().

The understanding HW format can be simplified to the following (per line):

- one 16b initial fix point coefficient, followed by
- several differential values encoded on [LSH_DELTA_BITS_MIN ; LSH_DELTA_BITS_MAX] bits (usually 4 to 10). The actual number of bits can be chosen by SW.

Warning: It is important to remember that the LSH HW block has several responsibilities:

- correct the de-shading effect (configured in SW by LSH module)
- apply White Balance gains and clip (configured in SW by the WBC module)
- apply the system black level (configured in SW by the BLC module)

The TRM will detail the order of the operations and which registers are related to which operation.

LSH grid from user-space

From user-space the following functions are used the control the de-shading grid:

- CI_PipelineAllocateLSHMatrix() Allocate a deshading grid with a given size in Bytes. The computation of the required size for a grid is explained later.

This returns a matrix identifier that should be used for all other functions. The matrix ID 0 is invalid.

- CI_PipelineDeregisterLSHMatrix() Free a deshading grid.

Can only be performed if the selected grid is not in use.

- CI_PipelineAcquireLSHMatrix() gives access to an already allocated deshading grid. The grid should then be released using CI_PipelineReleaseLSHMatrix().

The matrix currently used cannot be acquired.

- CI_PipelineUpdateLSHMatrix() to change the matrix currently in use. If the given matrix identifier is 0 then disables the usage of the LSH grid.

- CI_PipelineHasLSHBuffers() can be used to count the number of allocated matrices.

All these functions expect to deal with the HW memory format as it is stored in a CI_LSHMAT structure:

```
typedef struct CI_LSHMAT
{
    /** @brief matrix identifier - do not modify */
    IMG_UINT32 id;
    /** @brief pointer to user-side accessible data in HW format */
    void *data;
    /** @brief Size in Bytes */
    IMG_UINT32 ui32Size;
    /** @brief Associated configuration */
    CI_MODULE_LSH_MAT config;
} CI_LSHMAT;

typedef struct CI_MODULE_LSH_MAT
{
    /** @note Load Structure: LSH_ALIGNMENT:LSH_SKIP_X */
    IMG_UINT16 ui16SkipX;
    /** @note Register: LSH_OFFSET:LSH_SKIP_Y */
    IMG_UINT16 ui16SkipY;

    /** @note Load Structure: LSH_ALIGNMENT:LSH_OFFSET_X */
    IMG_UINT16 ui16OffsetX;
    /** @note Register: LSH_OFFSET:LSH_OFFSET_Y */
    IMG_UINT16 ui16OffsetY;

    /**
     * @brief The mesh size - a power of 2 in range of
     * [LSH_DELTA_BITS_MIN ; LSH_DELTA_BITS_MAX]
     *
     * @note Register: LSH_GRID_TILE:TILE_SIZE_LOG2
     */
    IMG_UINT8 ui8TileSizeLog2;

    /**
     * @brief Number of bits to use to store the differences
     *
     * @note Load Structure: LSH_GRID:LSH_VERTEX_DIFF_BITS
     */
}
```

```

IMG_UINT8 ui8BitsPerDiff;

/**
 * @brief Number of elements in a line of deshading grid
 * (including the 1st full 16b element)
 *
 * @note Not written to registers, @see ui16LineSize
 */
IMG_UINT16 ui16Width;

/**
 * @brief Number of lines in the deshading grid.
 *
 * @note Not written to registers, the HW reads as many lines as needed
 * according to the IIF output
 */
IMG_UINT16 ui16Height;

/**
 * @brief Number of elements in the matrix line in units of 16 bytes
 * minus 1
 * - related to ui16TileSize and the image's size
 *
 * @warning IN UNITS OF 16 BYTES
 *
 * @note Register: LSH_GRID_LINE_SIZE:LSH_GRID_LINE_SIZE
 */
IMG_UINT16 ui16LineSize;

/**
 * @brief The matrix stride (per channel) in bytes
 *
 * @note Register: LSH_GRID_STRIDE:LSH_GRID_STRIDE
 */
IMG_UINT32 ui32Stride;

} CI_MODULE_LSH_MAT;

```

Populating the field of that structure from user space can be done manually but it is recommended to use the helper functions from the MC level that take advantage of the FelixCommon LSH_GRID structure that simplifies the access to a grid:

```

typedef struct LSH_GRID
{
    /**
     * @brief tiles are square. width >= imager_width/tiles_size and
     * height >= imager_height/tile_size
     */
    IMG_UINT16 ui16TileSize;
    /** @brief in "tiles" */
    IMG_UINT16 ui16Width;
    /** @brief in rows of tiles */
    IMG_UINT16 ui16Height;

    /** @brief a ui16Width*ui16Height matrix for each channel */
    LSH_FLOAT *apMatrix[LSH_MAT_NO];
} LSH_GRID;

```

FelixCommon also provides several functions to handle the structure.

- LSH_CreateMatrix() or LSH_AllocateMatrix(). Respectively to create a matrix when the sensor's resolution is known or allocate a matrix when the size of the matrix is actually known.

Either way the matrix should be deleted using LSH_Free().

- For testing purposes LSH_FillLinear() or LSH_FillBowl() can be used to fill different channels with data.
- Or matrices can be loaded or saved from files using LSH_Load_bin() and LSH_Save_bin(). The details about the file format are available in the *Deshading Grid file format (LSH)* (page 44) section.

The MC layer can also be used to get information about LSH_GRID and transform it to relevant information to fill the CI_LSHMAT properly. The main one being MC_LSHConvertGrid() that converts a LSH_GRID to an equivalent CI_LSHMAT using the following functions:

- MC_LSHComputeMinBitdiff() can be used to compute the minimum number of bits to use to encode the matrix into the HW format.
- MC_LSHGetSizes() get the stride and line size for a matrix using a specific bits per diff for the encoding.

For testing purposes the MC layer also has MC_LSHPreCheckTest() that can be used to ensure that matrices generated using LSH_FillLinear() or LSH_FillBowl() are following the maximum gain and correct bits per difference selected.

An example of usage of the FelixCommon and MC functions could be (simplified version of felixtest.c used for the driver_test test application):

```
IMG_UINT32 uiAllocation = 0;
IMG_UINT32 matId = 0;
IMG_UINT32 uiStride = 0;
IMG_UINT32 uiLineSize = 0;
IMG_UINT8 ui8BitsPerDiff = 0;
CI_LSHMAT sMatrix;
LSH_GRID sGrid;
IMG_RESULT ret;
const char *filename = "deshading_t32.lsh";

// ensure that structure is initialised
IMG_MEMSET(&sGrid, 0, sizeof(LSH_GRID));

// allocates and populate the LSH_GRID structure from disk
ret = LSH_Load_bin(&LSH_GRID, filename);
if (IMG_SUCCESS != ret)
{
    LOG_ERROR("Failed to load '%s'\n", filename);
    return IMG_ERROR_FATAL;
}

ui8BitsPerDiff = MC_LSHComputeMinBitdiff(&sGrid, NULL);

LOG_INFO("LSH grid %s uses %u bits per difference\n",
        filename, ui8BitsPerDiff);

uiAllocation = MC_LSHGetSizes(&sGrid, ui8BitsPerDiff,
```

```

    &uiLineSize, &uiStride);

ret = CI_PipelineAllocateLSHMatrix(pPipeline, uiAllocation,
    &matId);
if (IMG_SUCCESS != ret)
{
    LOG_ERROR("Failed to allocate LSH matrix buffer\n");
    LSH_Free(&sGrid);
    return IMG_ERROR_FATAL;
}

// first get the memory
ret = CI_PipelineAcquireLSHMatrix(pPipeline, matId, &sMatrix);
if (IMG_SUCCESS != ret)
{
    LOG_ERROR("Failed to acquire the LSH matrix %d\n", matId);
    LSH_Free(&sGrid);
    return IMG_ERROR_FATAL;
}

// then convert the grid to HW format
ret = MC_LSHConvertGrid(&sGrid, ui8BitsPerDiff, &sMatrix);
if (IMG_SUCCESS != ret)
{
    LOG_ERROR("Failed to convert the LSH matrix %d\n", matId);
    LSH_Free(&sGrid);
    return IMG_ERROR_FATAL;
}

// apply the converted matrix to the device memory
ret = CI_PipelineReleaseLSHMatrix(pPipeline, &sMatrix);
if (IMG_SUCCESS != ret)
{
    LOG_ERROR("Failed to release the LSH matrix %d\n", matId);
    LSH_Free(&sGrid);
    return IMG_ERROR_FATAL;
}

// configure the new matrix to be the one the HW uses
ret = CI_PipelineUpdateLSHMatrix(pPipeline, matId);
if (IMG_SUCCESS != ret)
{
    LOG_ERROR("Failed to update the LSH matrix %d\n", matId);
    LSH_Free(&sGrid);
    return IMG_ERROR_FATAL;
}

// the matrix has been converted - we don't need the initial object
LSH_Free(&sGrid);

```

The *CI Appendix: IOCTL tables* (page 355) contains the details of the user-kernel communication.

LSH grid from kernel-space

The handling of the grid in the kernel space is a bit more complex than the user-space because it has to handle a very strong HW limitation: the LSH module has *not* been designed to allow the grid to change every frames.

The HW needs several registers AND load structure elements to be written for a complete LSH grid to be configured properly. This means that configuring a new grid can fail if the configuration is different from the currently applied one.

Note: This limitation only applies to a Pipeline object that:

- started the capture
- has elements in the pending list

And applies to *any* change including configuring the first grid.

LSH grid: buffer creation

The creation of a buffer for the LSH grid is similar to a normal output buffer. It uses the same IOCTL and enters the kernel modules in INT_CI_PipelineCreateBuffer(). The creation is delegated to KRN_CI_PipelineCreateLSHBuffer() which creates a KRN_CI_LSH_MATRIX that contains a normal KRN_CI_BUFFER used for the memory. The KRN_CI_BUFFER uses a special type (CI_BUFF_LSH_IN) to be recognised when mapping to user space.

Note: LSH buffers cannot be tiled.

The mapping to user space is handled in a similar way that for any buffer (to the extent that LSH buffers are accessible for writing). The KRN_CI_BUFFER is added to the non-mapped list using KRN_CI_PipelineAddBuffer(). When *mmap()* is called by user-space it will trigger KRN_CI_PipelineBufferMapped() which will add it the new matrix to the KRN_CI_PIPELINE::sList_matrixBuffers instead of KRN_CI_PIPELINE::sList_availableBuffers.

Which matrix is currently in use

The kernel side Pipeline knows which LSH matrix is in use by using the KRN_CI_PIPELINE::pMatrixUsed pointer which can be NULL.

The currently used matrix is changed using KRN_CI_PipelineUpdateMatrix(). This is done when starting the capture (KRN_CI_PipelineStartCapture()) or when updating the matrix in INT_CI_PipelineChangeLSHMatrix()

Warning: INT_CI_PipelineChangeLSHMatrix() updates the matrix only if:

- the capture is not started or
- for a started capture: the Pipeline has no pending buffers or
- the configuration for the new matrix is the same than the old one

This ensures that if the HW is computing frames only the LSH address can change.

The function KRN_CI_DriverCheckDeshading() is used to ensure that the matrix size respects the HW limitation (especially the CI_HWINFO::ui32LSHRamSizeBits).

7.3.5 Output Formats and sizes

This section describes how to choose the output format using the registers values and the MC/CI setup associated with these formats. More advanced output formats may be possible using the high level setup parameters to configure the ISPC library. (see *Output formats (OUT)* (page 210)).

The last part of this section explains how the buffer sizes are computed and how the user can allocate or import different sizes. It also explains how to trigger frames to be captured with different memory layout.

Encoder pipeline formats (YUV)

The encoder pipeline (after ESC block) can produce 4 YUV formats (see *TRM* for more details):

- NV21 8 bits (420PL12YVU8) and its 10 bits equivalent
- NV61 8 bits (422PL12YVU8) and its 10 bits equivalents

The size used by default for allocation is using the sensor's size from the IIF configuration but can be changed using CI_PIPELINE::ui16MaxEncOutWidth and CI_PIPELINE::ui16MaxEncOutHeight. These variables are updated if the capture is not started and no image buffers were allocated yet.

Register setup

The control of the output format is done using a few registers:

- **SAVE_CONFIG_FLAGS:**
 - ENC_FORMAT to select the bit-depth (0 for 8 bits, 1 for 10 bits).
 - ENC_ENABLE should be 1 (enabled).
- ENC_SCAL_V_SETUP::422_NOT_420
- ENC_422_TO_420_CTRL::ENC_422_TO_420_ENABLE is a legacy register and should always be 0.

Format	ENC_FORMAT	422_NOT_420	ENC_422_TO_420_ENABLE
NV21	0 for 8b 1 for 10b	0	0
NV61	0 for 8b 1 for 10b	1	0

Warning: When subsampling to 420 using the scaler the Vertical Chroma taps should be computed as if the vertical pitch was doubled.

CI setup

To enable YUV output CI_PIPELINE should be modified with:

- eEncType should have the correct YUV format that can be setup using the PixelTransformYUV() function with the correct ePxlFormat value:
 - YUV_420_PL12_8
 - YUV_422_PL12_8
 - YUV_420_PL12_10
 - or YUV_422_PL12_10
- ui16MaxEncOutWidth and ui16MaxEncOutHeight should be the imager size after decimation (or the output will need cropping).
- CI_MODULE_SCALER::bOutput422 should be configured correctly according to the selected output in CI_PIPELINE::sEncoderScaler.

MC setup

The ePxlFormat enum is stored in MC_PIPELINE::eEncOutput and the size is derived from the IIF setup. To change the maximum size ensure it is over-written after the conversion is done.

E.g. in `mc_convert.c`:

```
PixelTransformYUV(&(pCIPipeline->eEncType), pMCPipeline->eEncOutput);
pCIPipeline->ui16MaxEncOutWidth = pMCPipeline->sIIF.ui16ImagerSize[0]*CI_CFA_WIDTH;
pCIPipeline->ui16MaxEncOutHeight = pMCPipeline->sIIF.ui16ImagerSize[1]*CI_CFA_HEIGHT;
/* ... */
pCIPipeline->sEncoderScaler.bOutput422 =
    pCIPipeline->eEncType.ui8HSubsampling == 1 ? IMG_TRUE : IMG_FALSE; // 422
```

Enabling Tiling

The YUV output can be enabled as tiled. In that case the fields ENC_L_TILE_EN and ENC_C_TILE_EN in the TILING_CONTROL register of the linked list should be updated.

When allocating the buffer with CI the tiling option will change the output size and the result buffer from the HW will need de-tiling to be displayable.

Additional YUV formats

It is possible to configure the HW such that the chroma component order could be different (HW outputs VU order as NV21 supports). Modifying the values for the R2Y block and any subsequent block affected by the Chroma order would allow the output of NV12 (MIE, R2Y). It is possible that R2Y may not be affected if similar options are available for RGB ordering (output of BGR instead of standard RGB, see *Display pipeline formats (RGB)* (page 157)).

This configuration tricks should however be avoided at the CI level as the High level libraries provide an easier way to support such formats.

Display pipeline formats (RGB)

The display pipeline (after DSC block) can produce 3 RGB formats:

- RGB8 24b (3 channels, 8b per channel).
- RGB8 32b (3 channels, 8b per channel packed in the LSB of 32b) – similar to RGBA without alpha channel.
- RGB10 32b (3 channels, 10b per channel packed in the LSB of 32b).

Warning: It is not possible to have data extraction and RGB output enabled at the same time (HW design).

The size used by default for allocation is using the sensor's size from the IIF configuration but can be changed using `CI_PIPELINE::ui16MaxDispOutWidth` and `CI_PIPELINE::ui16MaxDispOutHeight`. These variables are updated if the capture is not started and no image buffers were allocated yet.

Register setup

To enable RGB output 3 fields have to be setup in the `SAVE_CONFIG_FLAGS` register:

- `DISP_DE_ENABLE` should be 1 (enabled)
- `DE_NO_DISP` should be written to 0 (enable display pipeline rather than data extraction)
- `DISP_DE_FORMAT` should have the correct value

Format	DE_NO_DISP	DISP_DE_FORMAT
RGB8 24b	0	0
RGB8 32b	0	3
RGB10 32b	0	1

CI setup

To enable RGB output `CI_PIPELINE` should be modified with:

- `eDispType` should have the correct RGB format that can be setup using the `PixelTransformRGB()` function with the correct `ePxlFormat` value:
 - `RGB_888_24`
 - `RGB_888_32`
 - `RGB_101010_32`
- `ui16MaxDispOutWidth` and `ui16MaxDispOutHeight` should be the imager size after decimation (or the output will need cropping).

MC setup

The `ePxlFormat` enum is the one stored in `MC_PIPELINE::eDispOutput`. The size is derived from the IIF setup.

E.g. (from `mc_convert.c`)

```
PixelTransformRGB(&(pCIPipeline->eDispType), pMCPipeline->eDispOutput);
pCIPipeline->ui16MaxDispOutWidth = pMCPipeline->sIIF.ui16ImagerSize[0]*CI_CFA_WIDTH;
pCIPipeline->ui16MaxDispOutHeight = pMCPipeline->sIIF.ui16ImagerSize[1]*CI_CFA_HEIGHT;
```

Enabling Tiling

The RGB output can be enabled as tiled. In that case the field `DISP_DE_TILE_EN` in the `TILING_CONTROL` register of the linked list should be updated.

Additional RGB formats

It is possible to configure the HW modules to output RGB formats with different components orders. The V2500 HW outputs standard RGB (B in LSB) but by swapping the Y2R matrix it is possible to configure the HW to output BGR (R in LSB). Note that this may affect other modules as well (DGM needs some of its register values to be swapped as well).

As for the YUV swapping this configuration tricks should however be avoided at the CI level as the High level libraries provide an easier way to support such formats.

Data Extraction formats (Bayer)

The V2500 hardware has several data extraction points where the image can be retrieved before further corrections are applied to it. The *TRM* should detail them. The point of extraction is global between contexts and for the moment only supports Bayer format (it is at the beginning of the HW pipeline). Bayer format with 8b, 10b or 12b RGGB patterns are available.

The size used by default for allocation is using the sensor's size from the IIF configuration

Warning: It is not possible to have data extraction and RGB output enabled at the same time (HW design).

Register setup

To enable DE output 3 fields have to be setup in the `SAVE_CONFIG_FLAGS` register:

- `DISP_DE_ENABLE` should be 1 (enabled)
- `DE_NO_DISP` should be written to 1 (enable data extraction instead of display pipeline)
- `DISP_DE_FORMAT` should have the correct value

Format	DE_NO_DISP	DISP_DE_FORMAT
RGGB 8b	1	0
RGGB 10b	1	1
RGGB 12b	1	2

CI setup

The CI setup to enable Data Extraction is slightly more complicated, CI_PIPELINE should be modified with:

- eDispType should have the correct Bayer format for the selected DE point.
- ui16MaxDispOutWidth and ui16MaxDispOutHeight should be the imager size after decimation (or the output will need cropping).
- eDataExtraction should be the DE point in the Pipeline

E.g.

```
PixelTransformBayer(&(pCIPipeline->eDispType), BAYER_RGGB_10, MOSAIC_RGGB);
pCIPipeline->ui16MaxDispOutWidth = ui16ImagerSize[0]*CI_CFA_WIDTH;
pCIPipeline->ui16MaxDispOutHeight = ui16ImagerSize[1]*CI_CFA_HEIGHT;
pCIPipeline->eDataExtraction = eSelectedDEPoint;
```

MC setup

To setup the MC layer with Data Extraction enabled a few parameters of the MC_PIPELINE should be modified:

- eDEPoint should be set to the wanted DE point
- eDispOutput should be disabled (PXL_NONE)
- the maximum size is computed using the IIF setup

E.g.

```
sMCPipelineConfig.eDispOutput = PXL_NONE;
sMCPipelineConfig.eDEPoint = 0;
```

Enabling Tiling

Enabling Bayer tiling is possible using the same mechanism than the Display output for the HW. But the driver does not allow tiling of the data-extraction buffer.

HDR Extraction

HDR Extraction is a special point in the pipeline where RGB image can be extracted at a higher bit-depth than display output. The images are then intended to be merged by an algorithm and inserted back in the HDR insertion point.

The V2500 HW allows extraction of a high bit-depth RGB format that can be used by Imagination's GPU to perform HDR processing over several frames. The extraction is done in between the Main Gamut Mapper and the Gamma LUT blocks in the HW. A single format is supported by the HW that is very similar to RGB32 10b. The intended usage is to extract a few frames using HDR Extraction and push them back into the pipeline after the GPU merged them using the HDR Insertion.

The size used by default for allocation is using the sensor's size from the IIF configuration

Register setup

To enable HDR Extraction (called HDF in registers) 2 fields have to be set into the **SAVE_CONFIG_FLAGS** register:

- **HDF_WR_ENABLE** should be 1
- **HDF_WR_FORMAT** should be the relevant format

Format	HDF_WR_FORMAT
BGR10 32b	1

CI setup

To enable the HDR Extraction the **CI_PIPELINE** should be modified with:

- **eHDRExtType** should be the correct format (**BGR_101010_32** transformed with **PixelTransformRGB()**).
- The size of the allocated buffer should be whole image processed by the IIF (when allocating the buffer with CI it is computed using the IIF register values).

MC setup

To enable the HDR Extraction the **MC_PIPELINE::eHDRExtOutput** should have the correct enum value (**BGR_101010_32**). The size limitations are the same than in CI.

Enabling Tiling

The HDR Extraction output can be tiled and the **HDF_WE_TILE_EN** field of the **TILING_CONTROL** register should be updated accordingly.

Raw 2D Extraction (TIFF)

The Raw 2D extraction is a special point in the pipeline that allow to extract specially formatted Bayer images just before the demosaicer.

Because the format is Byte aligned in memory (not using the alignment other output buffers use) it cannot be tiled (HW design). The HW supports 2 formats: 10b and 12b TIFF.

The size used by default for allocation is using the sensor's size from the IIF configuration

Register setup

To enable Raw 2D Extraction 2 fields of the **SAVE_CONFIG_FLAGS** register have to be setup:

- **RAW_2D_ENABLE** set to 1
- **RAW_2D_FORMAT** set to the correct value.

Format	RAW_2D_FORMAT
TIFF 10b	0
TIFF 12b	1

CI setup

The CI_PIPELINE element that should be modified to enable RAW2D extraction are:

- eRaw2DExtraction derived from a TIFF format using PixelTransformBayer().
- The size of the allocated buffer should be whole image processed by the IIF (when allocating the buffer with CI it is computed using the IIF register values).

MC setup

To enable RAW 2D extraction the MC_PIPELINE::eRaw2DExtOutput enum should be modified to be either BAYER_TIFF_10 or BAYER_TIFF_12. The same size limitations apply than in CI.

Enabling Tiling

Warning: Tiling cannot be enabled for RAW 2D output (HW design).

Different Output Size

The user-space library provides several way to allocate output buffers, both are using CI_PipelineAllocateBuffer():

- give a size of 0 and let the kernel-space compute the correct output size
- give a defined size for the output.

The kernel-space KRN_CI_PipelineCreateBuffer() will verify the size of the buffer using KRN_CI_BufferFromConfig(). This function delegates the size computation to several functions that are part of the FelixCommon library:

- CI_ALLOC_RGBSizeInfo() to compute RGB output sizes (HDR input and output are also RGB)
- CI_ALLOC_RGBSizeInfo() is also used to compute Bayer sizes.
- CI_ALLOC_Raw2DSInfo() to compute Bayer TIFF sizes as the format is packed.
- CI_ALLOC_YUVSizeInfo() is used to compute the YUV output sizes.
- All of the above use CI_ALLOC_GetTileInfo() to get tiling information.

If the provided size is 0 then the size from KRN_CI_BufferFromConfig() will be used. Otherwise the size will be checked to be big enough.

Note: The maximum output sizes are the sizes used for the Encoder and Display output. It is important that those sizes are correct when allocating buffers.

The images before the scaler (Bayer or HDR) are using the size configured in the IIF block.

The CI_PipelineAllocateBuffer() also returns an optional buffer identifier. This identifier is very important if using several sizes of buffers.

The default memory layout assumed by CI_PipelineAllocateBuffer() does not include special offsets between planes. It is quite common that the consumer of a buffer will require some special output sizes or offset to be respected. The size given to the allocation function should reflect that. However the actual offset of each plane is provided for each enqueued frame.

More information about how each format size is computed is available in *CI Appendix: Frame Size Computation* (page 345).

Different memory layout

The memory layout is specified when triggering frames. The user has a choice:

- trigger the first available buffer from the list with default memory layout.
- trigger specific frame with a changed memory layout.

The first solution is covered by using CI_PipelineTriggerShoot() or CI_PipelineTriggerShootNB() and is triggering captures with planes usually back to back.

The second solution is using CI_PipelineTriggerSpecifiedShoot() or CI_PipelineTriggerSpecifiedShootNB() and requires a CI_BUFFID structure to be populated. This structure will contain a buffer identifier, stride and offset for all planes of all desired outputs.

Note: If the identifier given for a particular buffer is 0 it will also use the first available buffer but use a changed memory layout.

For example, at time of writing, the structure CI_BUFFID is contains the following fields for the YUV output:

```
typedef struct CI_BUFFID
{
    /** @brief Encoder buffer Identifier */
    IMG_UINT32 encId;
    /**
     * @brief Encoder Y buffer stride in bytes - if 0 ignored
     *
     * @note Needs to be a multiple of SYSMEM_ALIGNMENT
     */
    IMG_UINT32 encStrideY;
    /**
     * @brief Encoder CbCr buffer stride in bytes - if 0 ignored
     *
     * @note Needs to be a multiple of SYSMEM_ALIGNMENT
     */
    IMG_UINT32 encStrideC;
    /**
     * @brief start of luma plane offset in bytes
     *
     * @note Needs to be a multiple of SYSMEM_ALIGNMENT
     */
}
```

```

IMG_UINT32 encOffsetY;
< /**
 * @brief chroma offset in bytes - if 0 computed by kernel-side to be
 * just after luma plane
 *
 * @note Needs to be a multiple of SYSMEM_ALIGNMENT
 */
IMG_UINT32 encOffsetC;

/* ... */
};

```

Using the example computation from *2 Frames example* (page 349) we can therefore use the buffer as following (see *Populating a specific Buffer when enqueueing shots* (page 197) for ISPC equivalent):

```

unsigned int size = 1567808;
unsigned int stride = 1088;
unsigned int a_off_y = 0, a_off_cbcr = 783360,
             b_off_y = 1088, b_off_cbcr = 784448;

/*
 * assumes pPipeline is connected, configured but not started yet
 * the call could also be to CI_PipelineImportBuffer()
 */
int buffId = 0;

ret = CI_PipelineAllocateBuffer(pPipeline, CI_TYPE_ENCODER, size,
                               IMG_FALSE, &buffId);

/*
 * the capture should now be started so that we can configure the
 * trigger of a frame buffId could be found from a list after
 * allocation or left to 0 and let the 1st available buffer be used
 * (in that case the assumption is that all the YUV buffers were
 * allocated with the correct size
 */

CI_BUFFID toTrigger;
// memset to ensure other fields are disabled
memset(&toTrigger, 0, sizeof(CI_BUFFID));

toTrigger.encId = buffId;
toTrigger.encStrideY = stride;
toTrigger.encStrideC = stride;
if (is_a)
{
    toTrigger.encOffsetY = a_off_y;
    toTrigger.encOffsetC = a_off_cbcr;
}
else
{
    toTrigger.encOffsetY = b_off_y;
    toTrigger.encOffsetC = b_off_cbcr;
}

ret = CI_PipelineTriggerSpecifiedShoot(pPipeline, &toTrigger);

```

Notes on Tiling

Tiling is supported as part of the IMG MMU IP. The V2500 chip has been modified slightly to enable better pre-fetching when tiling is enabled. The intended usage of tiling is to tile output buffers that will be used by the GPU, that will de-tile them at the same time processing is done on the buffer.

Please refer to the *Video Bus4 MMU Functional Specification* HW documentation for more details on tiling.

Obviously when allocating the buffer with CI the tiling option will change the output size and the result buffer from the HW will need de-tiling to be usable as normal outputs.

To enable tiling output support the user is expected to enable tiling with `CI_PIPELINE::bSupportTiling` and also allocate tiled buffers with `CI_PipelineAllocateBuffer()`. The CI driver will figure out if a buffer is tiled or not and configure the stride accordingly. See the [MMU Tiling information](#) (page 179) section for details on the SW limitations.

The shared tiling stride is computed in `KRN_CI_PipelineInit()`. The CI library does not let the user-side compute the size of tiled buffers, it relies on `ci_alloc_info.h` to provide the correct sizes for all outputs. Because each buffers could be triggered as tiled when the `CI_PIPELINE::bSupportTiling` is ON then all outputs that can be tiled will be used to compute the shared tiled stride. This tiled stride can be changed at insmod time (but if given parameter is too small for allocation then the setup of the Pipeline will fail) as explained in [V2500 Insertion options](#) (page 17).

7.3.6 V2500 Cache behaviour

The Felix HW is connected to a memory interface (E.g. AXI on arm architectures) that has to control the way the memory is read/written. The cache policy controls the behaviour of several HW blocks and is always the same:

- Bypass the cache
- Or write through (direct write to cache and memory)
- Or write combine (write to the cache and flush memory later)

Additionally some blocks have a configurable “fence” behaviour, which is usually done at the end of writing.

Compilation choice

The driver can be changed so that other options than the defaults are used. The available values for the cache related registers are available in `felix_hw_info.h` and can be changed by the customer. But it is the customer’s responsibility to insure that the caches are dealt with correctly according to the system they integrate with. The current values are generated from the defaults stored in our register definitions.

The cache policies are written when accessing the registers in `kernel_src/ci_hwstruct.c`.

Example of cache policy:

```
#define USE_WBS_CACHE_POLICY (0) //< @brief Value for WBS_MISC:WBS_CACHE_POLICY
```

Example of writing a cache policy:

```
tmp = 0;
REGIO_WRITE_FIELD(tmp, FELIX_LOAD_STRUCTURE, WBS_MISC, WBS_ROI_ACT,
    pWhiteBalance->ui8ActiveROI);
REGIO_WRITE_FIELD(tmp, FELIX_LOAD_STRUCTURE, WBS_MISC, WBS_CACHE_POLICY,
    USE_WBS_CACHE_POLICY);
REGIO_WRITE_FIELD(tmp, FELIX_LOAD_STRUCTURE, WBS_MISC, WBS_RGB_OFFSET,
    pWhiteBalance->ui16RGBOffset);
REGIO_WRITE_FIELD(tmp, FELIX_LOAD_STRUCTURE, WBS_MISC, WBS_Y_OFFSET,
    pWhiteBalance->ui16YOffset);
WriteMem((IMG_UINT32*)pMemory, FELIX_LOAD_STRUCTURE_WBS_MISC_OFFSET, tmp);
```

7.3.7 Suspend and Resume calls

The *suspend* and *resume* calls can be made from the Linux OS to the drivers to inform that the system is going to a halt. The expected behaviour is to save the current “HW context” (as registers are lost when resume is called) and restart to work when resume is triggered. Suspension is transparent to user-side and it is assumed that user-side is already frozen when suspend is called.

The driver does not expect a capture to be “resumable”, as the time of suspension is unknown it is hard to relate with previously captured statistics and images. Therefore when suspend is called the driver stops all running captures and turns the HW off.

When the *resume* action is called the HW is restarted and user-space has to resynchronise its “started” status (when IMG_ERROR_UNEXPECTED_STATE is received). Once that is done the application can choose to restart the capture or not.

7.4 User/Kernel interactions

The driver is designed to use IOCTL to communicate information from user-side to kernel-side. This is fairly standard for a Linux driver.

Read and write operations are not supported.

The image buffers are mapped from kernel space to user space using standard memory mapping to avoid memory copying. But some internal buffers are only available in the kernel-space.

7.4.1 Fake Implementation

The Fake mode was designed to have a “fake” ioctl() call when running the driver in full user-mode (full user-mode with simulator as HW). This was implemented by calling the kernel’s library ioctl switch routine directly. In order to have ioctl macros working on windows the header was copied in a local linux/ioctl.h.

In order to make the Fake mode work correctly the open() and close() functions were faked too. Some kernel structures are defined partially in user-mode so that the functions used to manage the device are the same.

On the kernel-side the `copy_from_user()` and `copy_to_user()` are implemented as simple alias for `memcpy()`.

The fake `memmap()` and `memunmap()` implementation is as simple as the ioctl implementation. It does not have to prepare memory. But an extra flag was added to the struct vma to allow the caller to receive the address of the required object. The real mode is just calling the standard functions.

7.4.2 Interactions

An interaction is *direct* when initiated directly by the user (e.g. ask for a frame to be captured) or *indirect* when done by the CI layer to provide easier functionalities to the above layer.

Direct interaction

- Update line-store information (CI_IOCTL_LINE_GET).
- Propose new line-store setup (CI_IOCTL_LINE_SET).
- Update gamma table information (CI_IOCTL_GMAL_GET).
- Propose new gamma table (CI_IOCTL_GMAL_SET).

Indirect interaction

- Register a connection to kernel side (`open()`)
- Update driver information (CI_IOCTL_INFO)
- Manage Pipeline structures connection (CI_IOCTL_PIPE_REG and CI_IOCTL_PIPE_DEL)
- Access to image buffers without copy (`memmap()` and `memunmap()`)

More details about the *system calls mapping* (page 353) and *ioctl tables* (page 355) is available in the appendices.

7.4.3 IMG Errors and Errno values

Having an interface with the Linux kernel the driver is expected to return values based from `errno`. But the standard IMG way is to use `IMG_RESULT` values for that. In order to provide this behaviour a mapping of `errno` values to `IMG_RESULT` is made (see `ci_errors.h` in the kernel sources):

IMG_SUCCESS: 0 Success.

IMG_ERROR_MALLOC_FAILED: -ENOMEM Allocation failed. Used whenever an allocation returns NULL.

IMG_ERROR_FATAL: An unexpected failure occurred.

Default value when converting errors.

IMG_ERROR_NOT_SUPPORTED: -ENOTSUP The action is not supported. Used either because the system is not ready for such action or because the action does not make sense. This is most likely happening only on development code.

IMG_ERROR_INVALID_PARAMETERS: -EINVAL Parameters are not legitimate. This should only happen in development code.

IMG_ERROR_INTERRUPTED: -EINTR An action has been interrupted. The user-side should try again.

IMG_ERROR_MEMORY_IN_USE: -EADDRINUSE The memory is already in use at creation or still in used at destruction time. Should only happen in development code.

IMG_ERROR_ALREADY_INITIALISED: -EEXIST The structure is already initialised. Should only happen in development code.

IMG_ERROR_MINIMUM_LIMIT_NOT_MET: -E2BIG Some requested size is too big to be supported.

IMG_ERROR_COULD_NOT_OBTAIN_RESOURCE: -EALREADY When failures to access resources (e.g. down a semaphore failed).

IMG_ERROR_TIMEOUT: -ETIME Timeout occurred (e.g. when waiting on semaphore down).

IMG_ERROR_UNEXPECTED_STATE: -ECANCELED An operation was cancelled due to unexpected state.

7.4.4 Debug Functions (fake driver)

When build the CI libraries using the Fake device some debug functionalities are optionally available. They are enabled at CMake time, and should be called directly from “user-space” (they are implemented in the “kernel-space” but only make sense when running a fake device):

- Dump register values – automatically done when triggering a shoot
- Over-write register values – done when a register over-write file is given
- Read CRC values from memory to help HW verification

All the functions are available from the ci_debug.h header and are using the GZip FileIO library to read or write to file.

Register Dumping

Register dumping is simply writing all the registers value to a file.

- KRN_CI_DebugDumpCore() dump the core registers
- KRN_CI_DebugDumpContext() dump the registers of a given context
- KRN_CI_DebugDumpRegisters() dump all registers (core and context)

The memory structures that contain registers information can also be written to file using:

- KRN_CI_DebugDumpPointersStruct() dump the linked list structure from memory
- KRN_CI_DebugDumpSaveStruct() dump the save structure from memory

- KRN_CI_DebugDumpLoadStruct() dump the load structure from memory

The mechanism used to dump registers is to use some REGIO structures that contain all fields' information. The first step is to sort registers by addresses so that the structure can be explored from start to end and its information used to read registers.

Register dumping file format

This format was agreed on with the V2500 Simulator to be easily comparable.

For register:

```
"[offs 0x%08X] %-4s = 0x%08X\n", regOffset, regName, regVal
```

If the detailed field option is ticked:

```
"[mask 0x%08X] .%-4s = %u (0x%08X)\n", fieldMask, fieldName, fieldVal, fieldVal
```

Register dumping options

Register dumping can be enabled or disabled at running time using KRN_CI_DebugEnableRegisterDump() and KRN_CI_DebugRegisterDumpEnabled().

Register dumping can be compressed using GZip FileIO library using the compile time option FELIX_DUMP_COMPRESSED_REGISTERS (as an `IMG_BOOL8`). By default the dumping is not compressed.

Register override

The register over-write functionality allows the context registers to be over-written just after configuration to a specific value. The driver provides this functionality as a file to load that will be used to overwrite the load structure submitted every time a capture is triggered (until reset or changed). From user side the only operation to do is to specify a file to load for a specific context using CI_DebugSetRegisterOverride().

On kernel side the initialisation is done when the driver is created and finalisation when it is destroyed. The over-write is called for every submitted capture using KRN_CI_DebugRegisterOverride(). An internal list for fields IDs (position into the sorted field list that is used for register dumping) and value is maintained per HW context and used to over-write some memory location.

This format was agreed on with the V2500 Simulator to be easily used during testing:

```
"%s %s 0x%x\n", bankName, fieldName, value
```

Examples of usage are available in *Register override* (page 41).

7.4.5 DebugFS (real driver)

DebugFS is a GNU/Linux feature that allows some variables to be accessible as files in a specific location of the kernel. This is only available when building the real driver with `CI_DEBUG_FUNCTIONS` enabled.

DebugFS is implemented as part of the module initialisation in `kernel_src/ci_init_km.c`.

Several files may be available and will vary from versions to version but the following lists can be used as a reference of what to expect in the CI debugFS variables. The files should be located in `/sys/kernel/debug/imgfelix/` and root privileges are needed to access them (DebugFS needs to be supported by the chosen kernel and the location mounted).

See also [CI Appendix: Adding a new DebugFS counter](#) (page 344).

General entries

DevMemMaxUsed: Maximum amount of device memory used since insmod.

DevMemUsed: Current amount of device memory used.

DriverNConnections: Number of connection to the driver (i.e. different calls to `open()`).

DriverNServicedHardInt: Number of timed the top-half of the interrupt handler has been called.

DriverNServicedThreadInt: Number of timed the bottom-half of the interrupt handler has been called. This should be the same than the number of time the top-half has been called.

DriverLongestHardIntUS: Longest time in μs (microseconds) (10^{-6}) spent in top-half interrupt handler.

DriverLongestThreadIntUS: Longest time in μs (microseconds) (10^{-6}) spent in bottom-half interrupt handler.

Context entries (# is context number)

DriverCTX#Active: Is the context active?

DriverCTX#Int: Number of interrupts received.

DriverCTX#Int_DoneAll: Number of interrupt frame done all received.

DriverCTX#Int_Ignore: Number of interrupt error ignore received.

DriverCTX#Int_Start: Number of interrupts frame start received.

DriverCTX#TriggeredHW: Number of frames triggered in HW (pushed to be captured).

DriverCTX#TriggeredSW: Number of frames triggered in SW (pushed to be captured). May not have been pushed in HW yet.

External Data-generator entries (# is datagen number)

DriverDG#Int: Number of interrupts received.

DriverDG#SubmittedSW: Number of frames triggered in SW (may not have been pushed to HW yet).

DriverDG#TriggeredHW: Number of frames triggered in HW.

DriverDGNServicedHardInt: If supporting external DG the number of time the top-half of the external DG interrupt handler has been called. Otherwise does not exists. This

should be the same than the number of time the top-half has been called (DriverNServicedHardInt).

DriverSGNServicedThreadInt: If supporting external DG the number of time the bottom-half of the external DG interrupt handler has been called. Otherwise does not exists. This does not have to be the same than the number of calls to the top-half (see *Bottom Half - Thread context* (page 173)).

DriverDGLongestHardIntUS: Longest time in μs (microseconds) (10^{-6}) spent in top-half interrupt handler.

DriverDGLongestThreadIntUS: Longest time in μs (microseconds) (10^{-6}) spent in bottom-half interrupt handler.

Internal Data-generator entries (# is internal DG number)

DriverIntDG#Int_EndOfFrame: Number of end of frame interrupt received.

DriverIntDG#Int_Error: Number of interrupt error received.

7.4.6 Register access for debugging purposes

Some of following functions **will fail** if the kernel module is not compiled with the debug functions (the associated IOCTL will be ignored).

It is always possible for the user-space to query the following states:

- the current HW timestamps using CI_DriverGetTimestamp()
- the current RTM info status using CI_DriverGetRTMInfo(). The RTM info is a HW debugging feature, please refer to the HW TRM for more details.
- the current status of a gasket using CI_GasketGetInfo()
- the HW read only registers are also available in the CI_CONNECTION::sHWInfo object (see CI_HWINFO structure for details of what is available).

If the kernel module was compiled with debug functions then the following functions can also be used for register access:

- read any register or memory location using CI_DriverDebugRegRead()
- write any register or memory location using CI_DriverDebugRegWrite()

Note: When using the register or memory access the user is expected to give an integer that identifies the memory bank. The enum present in `ci_ioctl.h` CI_DEBUG_BANKS contains the values. This is debug feature therefore the header may not be accessible on every include context. The user can use integer instead of the enum values.

For example at time of writing:

```
enum CI_DEBUG_BANKS {
    CI_BANK_CORE = 0,
    CI_BANK_GMA = 1,
    /* from CI_BANK_CTX to (CI_BANK_CTX + CI_N_CONTEXT - 1) */
    CI_BANK_CTX = 2,
    /* from CI_BANK_GASKET to (CI_BANK_GASKET + CI_N_IMAGERS - 1) */
    CI_BANK_GASKET = CI_BANK_CTX_MAX,
```

```

/** from CI_BANK_IFF to (CI_BANK_IFF + CI_N_IIF_DATAGEN - 1) */
CI_BANK_IIFDG = CI_BANK_GASKET_MAX,
CI_BANK_MEM = CI_BANK_IIF_MAX,

/** not an actual value, if >= then bank enum is invalid */
CI_BANK_N,
};

```

For example to access the all context status register (example offset 0x10 *not an actual value*) the user is expected to do:

```

IMG_RESULT ret;
IMG_UINT32 status; // context 1 status
int ctx;
const int ctx_base = CI_BANK_CTX; // or 2 if ci_ioctl.h is not available in this scope
const int reg_ctx_status = 0x10;

/* assuming pConnection is an existing CI_CONNECTION opened with success */

for (ctx = 0; ctx < pConnection->sHWInfo.nctx; ctx++)
{
    ret = CI_DriverDebugRegRead(pConnection, ctx_base+ctx, reg_ctx_status, &status);
    if (IMG_SUCCESS != ret)
    {
        fprintf(stderr, "failed to read registers\n");
    }
    else
    {
        printf("context[%d] status 0x%08x\n", ctx, status);
    }
}

```

7.5 Interrupt Management

The scope of this section is the different actions the kernel-side driver does to manage HW interrupts. Most of device driver for GNU/Linux the interrupts are handled using the Top/Bottom halves mechanism.

The Top Half is triggered when the HW interrupt is received. As it is in the kernel interrupt context it has to be completed as fast as possible. Its behaviour is detailed in the Top Half paragraph.

The Bottom Half is implemented using a standard work-queue that will perform some operations later in time. Its behaviour is detailed in the Bottom Half paragraph.

The V2500 driver does not have much work to do once an interrupt is received but it implements its top/bottom half approach by taking advantage of a GNU/Linux function: `request_threaded_irq()`. This interface allows 2 interrupt handler functions (one is the top half, the threaded one is the bottom half).

7.5.1 Request and release of the interrupt line

The CI driver request the interrupt line when the 1st call to `open()` is performed - precisely when creating a connection object and realising that no other connection object exist. This can fail in that case the system call will fail to connect to the device.

The interrupt line is released at the last call to `close()` - precisely when destroying the connection object and realising that it is the last one.

The request to the IRQ is implemented in the Platform Device function: `SYS_DevRequestIRQ()`.

7.5.2 Configured interrupts

The CI driver uses 2 interrupts from the V2500 HW:

- `INT_FRAME_DONE_ALL`: this interrupt signals that the capture of a frame is finished and that device memory is updated.
- `INT_START_OF_FRAME RECEIVED`: this interrupt is used to read the gasket frame counter and verify if frames we missed (expecting the gasket frame count to increment by one from last read).

The other possible interrupts from the HW that could be interesting are:

- `INT_CONFIGURATION_LOADED`: that signals a configuration has been loaded from the HW pending list. This could be used to have a closer-grained management of the semaphore that allows the driver to submit elements to the HW. Currently this semaphore is incremented when the completed frame is processed.
- `INT_ERROR_IGNORE`: this interrupt is generated to signal that the context could not retrieve a frame. This interrupt may become interesting to try to realise that the bandwidth limitation of the HW is not respected.

7.5.3 Top Half - Interrupt context

This runs in interrupt context in the kernel. It cannot access user-space memory and should complete as fast as possible. It is implemented in `HW_CI_DriverHardHandleInterrupt()`.

All read statuses are stored into a structure allocated on the stack. If interrupts were detected a status structure is allocated using the `GFP_ATOMIC` heap of kmalloc and enqueued into the driver's `KRN_CI_DRIVER::sWorkqueue`.

Contexts

When the interrupt is signalled the driver loops through all contexts, for each context it reads:

- `INTERRUPT_STATUS` register
- `INTERRUPT_ENABLE` register to know if interrupts were received while disabled
- `LAST_FRAME_INFO` to know if we missed interrupts

If the interrupt status is different than 0 the status is stored and will be passed to the bottom-half.

In all cases it resets the interrupts using the INTERRUPT_CLEAR register.

Internal Data Generator

The DG_INTER_STATUS is read as well as the frame count. It is stored and sent to the bottom-half regardless of the value.

Note: The frame count is accessed through `HW_CI_DataGenFrameCount()` function.

Gasket

The gasket can have interrupts but the driver does not enable them. The frame count is however read for each gasket as it will be used by the bottom-half.

Note: The frame count is accessed through `HW_CI_GasketFrameCount()` function.

MMU

The driver also verifies interrupt status for the MMU unit. But if an interrupt is sent from the MMU it means an erroneous configuration was set and the only way to recover is to start the capture again. See *CI Appendix: Debugging a Page Fault* (page 366) to know what to do in case of MMU issues.

External Data Generator

If the driver was compiled with External Data Generator support then the external data generator interrupt handler function `KRN_DG_DriverHardHandleInterrupt()` is called. The return code from that handler is stored and will be used in the bottom-half to know if the threaded equivalent should be called or not.

System Device clear

The last operation is to call the `SYS_DevClearInterrupt()` that can be implemented to clear additional registers on the interrupt lines.

For example IMG FPGA platform needs to clear an additional register to put the line down.

7.5.4 Bottom Half - Thread context

This should be automatically called by the kernel when `IRQ_WAKE_THREAD` is returned by the Top half. This can sleep and deals with the work provided by the top half.

Contexts

For INT_START_OF_FRAME RECEIVED it reads the gasket frame counter and compares it with the last read counter. This value may not have changed for the first frame.

Note: If the Internal Data Generator is replacing that gasket then the frame counter is read from the internal Data Generator instead.

For INT_FRAME_DONE_ALL it does:

1. Reads the LAST_FRAME_INFO status and get the last used frame ID from it (LAST_CONTEXT_TAG field).
2. Get the Pipeline structure associated to that HW context from the driver object.
3. Get the 1st frame on the Pipeline's pending list - the element is *removed* from the list.
4. Add the frame to the processed list and increment the associated semaphore
5. If this frame's ID is different from the last used one go back to 3.

Internal Data Generator

The driver also checks the Internal Data Generator interrupts:

1. Read DG_INTER_STATUS from the status - if it signals no interrupt no more processing is done for that datagen.
2. It moves the head of the Busy list in the Processed list.
3. If the Busy list still has elements in it trigger them in the HW.

It is possible that the Internal Data Generator cannot process a frame if the bandwidth limitation is not respected. In that case the HW cannot recover therefore the capture has to be stopped and started again. It is impossible to continue triggering frames on an erroneous internal data generator.

External Data Generator

If the driver was compiled with External Data Generator support then the external data generator interrupt handler function KRN_DG_DriverThreadHandleInterrupt() is called according to the return code from that was stored in the top-half.

7.6 Memory Mapping Unit (MMU)

Documentation about the MMU HW should be delivered as part of the HW package. The reference document is named *Video Bus 4 MMU Functional Specification*.

The MMU HW has 4 directory entries but only 2 can be used at once¹. The V2500 IP has 1 internal MMU that manages memory access for the Contexts and internal data generator.

¹ The MMU HW is designed for the decoder that has a front end and back end, resulting in only 2 of the 4 MMU directories to be used at once (and the decoder driver has to choose which one at given points of the decoding). This limitation may change from a version of the MMU HW to another.

The FPGA build also has an external MMU that manages memory access for the external Data Generators.

The CI Driver (kernel side) manages the internal MMU. Each Pipeline object creates its own software heap (i.e. manages its range of device virtual addresses, see [Virtual Address management and heaps](#) (page 176)) and the device memory is mapped to the MMU when the capture is started (and unmapped when it is stopped).

The internal data generator shares the same requestor as the 1st context (see [MMU Requestors](#) (page 175)) therefore they have to share the same directory entry. However the virtual addresses heaps do not overlap between Context memory and internal data generator (see [Virtual Address management and heaps](#) (page 176)). The sharing of a requestor may create trouble as the device MMU may need cache flushing (see [MMU Cache management and flushing](#) (page 178)).

The Data Generators part of the driver manages the external MMU. Each DG has its own software heap and memory is mapped when starting shooting.

It is possible to disable the memory translation to directly give physical address to the HW (virtual address, heaps or tiling is not supported when bypassing the MMU) when creating the kernel structures. This requires the memory to be allocated in continuous blocks. The register to configure this behaviour is `MMU_ADDRESS_CONTROL::MMU_BYPASS`.

When enabled the physical address of the page used as directories should be written into `MMU_BANK_INDEX` register.

The MMU can be stopped from processing memory requests using the `MMU_CONTROL1::MMU_PAUSE_SET` register. The pause is removed using `MMU_CONTROL1::MMU_PAUSE_CLEAR`.

7.6.1 MMU size

The MMU manages virtual addresses that are 32b wide but can support different physical address size. In V2500 the typical size is 40b physical. The supported page size is currently 4kB (as supported by Linux for x86 or arm). The MMU HW cannot support other sizes than 4kB but it is possible to modify the driver's allocator to support such configurations.

To support the wider physical range the MMU can support shifting the physical addresses (configurable in `MMU_ADDRESS_CONTROL::MMU_ENABLE_EXT_ADDRESSING`). Note that when the extended addressing is enabled not only the entries in the page entries but also the directory page address should be shifted using that mechanism.

The size of the physical memory supported by the current MMU is available in the register `MMU_CONFIG0::EXTENDED_ADDR_RANGE`.

7.6.2 MMU Requestors

The V2500 HW interacts with the MMU through *requestors*. A requestor is an identifier that is used by the MMU as a reference to choose which directory entry to use. The driver must configure the requestor to use a specified directory. For V2500 the following values are used:

HW Element	MMU Requestor
Felix Context 0	0
Felix Context 1	1
Felix Context N	N
Internal Data Generator	0

HW Element	External MMU Requestor
External Data Generator 0	0
External Data Generator 1	1
External Data Generator N	N

In the MMU HW the requestors are configured using the `MMU_BANK_INDEX` register.

7.6.3 MMU Interrupts

Interrupts from the MMU are generated when a page fault is signalled. The MMU does not have an interrupt management of its own but requires its attached IP to signal it with its own flags and line.

The `MMU_STATUS0` and `MMU_STATUS1` registers can be used to figure out the source of the fault:

- `MMU_STATUS0::MMU_PF_N_RW` signals if the fault is a page fault (if 0 it is read/write flag error).
- `MMU_STATUS1::MMU_FAULT_RNW` signals if the fault was during a read (if 0 a write) operation.
- `MMU_STATUS1::MMU_FAULT_INDEX` specifies which directory generated the fault
- `MMU_STATUS0::MMU_FAULT_ADDR` is the virtual address that generated the fault

The V2500 driver does not handle such interrupt besides printing the error but it uses the information to crawl the page table entries and print the stored values to help debugging.

The `MMU_CONTROL1::MMU_FAULT_CLEAR` register should clear the fault and send the memory request back to the MMU to try to translate again.

Note: MMU interrupt can only occur if the system was badly configured.

See *CI Appendix: Debugging a Page Fault* (page 366) to try debugging page faults.

7.6.4 Virtual Address management and heaps

In this section heap means “a range of virtual address that is supposed to be used for one defined purpose” (e.g. virtual address range from `0x0` to `0x1FFFF` is for data buffer, the range from `0x20000` to `0xFFFFFFF` is for image buffers).

Hardware heap management

The MMU HW does not manage “heaps” as such. But the MMU HW supports different area of memory to be used as tiled buffers. The configuration for the tiling is done through `MMU_TILE_MIN_ADDR_T`, `MMU_TILE_MAX_ADDR_T` and `MMU_TILE_CFG` table registers.

V2500 MMU HW supports tiling and the driver can be configured at insmod time to choose between the 2 possible tiling schemes (*Getting Started Guide* (page 17)). The choice of tiling a buffer should then be done at allocation/import time.

The 2 tiling schemes are:

- 256 Bytes x 16 lines
- 512 Bytes x 8 lines

Software heap management

The virtual address management is done by heaps from the IMGMMU library. The driver configures those heaps at driver creation in `kernel_src/ci_internal.c` using an array of heaps per Pipeline object. The number of heaps is defined as `CI_N_HEAPS` in `ci_kernel_structs.h` and has 3 types of heaps as described later. As mentioned previously each context uses a directory therefore they can use the same virtual addresses. The “software” heaps are:

1. The Data Heap (`CI_DATA_HEAP`) contains the V2500 data structures. It has to be big enough to hold the Lens Shading matrix, the DPF read map and all the data structures that could be needed to fill up the Linked List register (HW Load structure, HW Save structure and HW Link list structure and DPF output map – rounded up to `PAGE_SIZE` which is usually 4kBytes). The usual needed size is around 20 MB for 16 buffers. The heap is configured to use the first 256MB of virtual memory.
2. The internal data generator heap (`CI_INTDG_HEAP`) contains the internal data generator frames.
3. The Image Heap (`CI_IMAGE_HEAP`) contains the image buffers that are not tiled. It is configured to be half of the remaining memory after the Data Heap.
4. The Tiled Image Heaps (`CI_TILED_IMAGE_HEAP0` to `CI_TILED_IMAGE_HEAP3`) are similar to the image heap but for tiled images. However because of the HW MMU limitation about the tiling stride each HW context has its own heap (i.e. if the Pipeline object is attached to HW context 1 it should use `CI_TILED_IMAGE_HEAP1` for its tiled images).

Heap	V2500 Virtual address range
Data Heap	0x00001000 – 0x0FFFFFFF 65,535 pages, ~256 MB (missing the 1st page) Entries 0:1 to 63:1023
Image Heap	0x10000000 – 0x7FFFFFFF 458,752 pages, ~1.75 GB Entries 64:0 to 511:1023
Tiled Image Heaps Shared equally amongst each context	0x80000000 – 0xEFFFFFFF 458,752 pages, ~1.75 GB Entries 512:0 to 959:1023
Internal DG Heap	0xF0000000 – 0xFFFFFEFFF 65,535 pages, ~256 MB (missing the last page) Entries 960:0 to 1023:1022

Note: If the usage of the Raptor IP needs more virtual memory it is recommended to simply change the computation of the sizes by forcing the unused heaps to 0 when in `kernel_src/ci_internal.c` `KRN_CI_DriverCreate()`.

The heaps for the External Data Generator are similar but do not support tiled images. Moreover the Data Heap is not used currently (the data generator is driven manually rather than using its linked list) therefore the Data Heap is put as the last 256MB of memory.

Heap	V2500 Virtual address range (External)
Data Heap	0xE0000000 – 0xFFFFEFFF 65,535 pages, ~256 MB (missing the last page) Entries 896:0 to 1023:1022
Image Heap	0x00001000 – 0xDFFFFFFF 917,503 pages, ~3.5GB (missing the 1st page) Entries 0:1 to 895:1023

Note: Neither the V2500 nor External Data Generator heaps uses the first or last page of the system. This serves a double purpose:

- Virtual address 0 (in fact <0x1000) are invalid
- Avoid trouble with genalloc.c when using the GNU/Linux kernel (see *Memory band management (genalloc)* (page 144)) (rounding errors when reaching 4GB on 32b systems).

Example of heap address

Example for tiled heaps with a HW configuration that has 2 contexts:

Heap	V2500 Virtual address range
Tiled Image Heap 0	0x80000000 – 0xB7FFEFFF 229,375 pages, ~895 MB Entries 512:0 to 735:1022
Tiled Image Heap 1	0xB7FFF000 – 0xEFFFDFFF 229,375 pages, ~895 MB Entries 735:1023 to 959:1021
Tiled Image Heap 2	None
Tiled Image Heap 3	None

7.6.5 MMU Cache management and flushing

The MMU HW supports two cache management operations: invalidate and flush. The HW has 2 caches: one for the directory entries and one for the page table entries. Both cache are unique per directory and are invalidated using a repeated field.

Any update related to the directory page (either the update of the base address in `MMU_BANK_INDEX` register or the addition of a page into the directory) should invalidate the 2 caches using the correct `MMU_CONTROL1::MMU_INVALIDC`.

Any update in the page tables entries should invalidate the corresponding “table cache” for that directory using `MMU_CONTROL1::MMU_FLUSH`.

It is expected that the memory managed by the MMU unit is un-cached in the CPU MMU.

Note: The `MMU_INVALIDC` action invalidates both caches while `MMU_FLUSH` only invalidates the table cache.

Warning: Internal Data Generator and Context 0 potential issues

The internal Data Generator and the 1st Context share the same directory entry in the MMU. Therefore flushing the cache for directory 0, when manipulating the mapping of one, may affect the memory latency of the other. This can cause frames to be dropped if the system is used close to its output bandwidth limit.

7.6.6 MMU Tiling information

The tiling of the memory is handled by the device MMU entirely. The process is transparent to the ISP HW. However enabling such output has effects on the SW as the MMU has several expectations.

When enabling tiling in the MMU expects a region of the virtual memory to be reserved (called heap). This region is shared amongst ALL MMU contexts, therefore the driver made the decision to allow only 1 tiled region per MMU context (as explained in the *Virtual Address management and heaps* (page 176)).

For each tiled heap the SW has to define one tiling stride. Therefore any tiled buffer has to use the **same** tiling stride in a given Pipeline object. For example if the ISP outputs both YUV and RGB it is expected that the YUV stride will be smaller: 420 8b YUV needs 1 Byte per pixel while RGB usually needs 3 or 4 Bytes per pixels as R/G/B are interleaved. If both YUV and RGB are tiled then they have to share the same tiling stride: the biggest is therefore chosen, the YUV buffer will be a lot bigger than it needs to be.

When using tiling it is also important that the 1st virtual address used for a buffer is at the start of a tile (so that the transformation using the tiling scheme does not go backward in virtual space). To do so the driver reserves additional virtual memory. The size of the additional memory depends on the selected tiled stride.

The maximum and minimum tiling strides supported by the SW are defined in ci_kernel.h as MMU_MAX_TILING_STR and MMU_MIN_TILING_STR.

7.7 Shot and Buffer lifecycle

This section describes the several operations that are done on a Shot, from the creation (allocation or importation) to the destruction. A Shot contains information about several Buffers (e.g. encoder buffer and display buffer) but also some internal data. Both kernel and user-space store information about Shots and Buffers in different structures.

The distinction between SHOT and BUFFER is very important: a Buffer represent only 1 output of the pipeline while a Shot aggregates all the outputs. Not all the memory accessible from a Shot is a Buffer, for instance internal memory is only part of the Shot (i.e. the user cannot choose which internal buffer to use when triggering a shot).

Internal memory buffers accessible to the user are:

- Statistics output
- Defective pixels output map
- Encoder statistics output

Internal memory buffers accessible only in kernel-space are:

- Linked list structure
- Load structure
- Other input buffers (Defective read map, LSH grid) that are transferred from user to kernel before being written

Importable memory buffers are all the output images:

- Encoder pipeline output
- Display pipeline output
- Data Extraction output
- HDR extraction output if the HW version allows this format
- RAW 2D extraction output if the HW version allows this format

As shown in *Shot elemets (simplified)* (page 180) the internal only structures are part of the Shot while the external Buffers can be dynamically attached to it ².

The following sections will explain the Shots and Buffer lifecycle from the user-side perspective as an overview of the cycle and from the kernel-side driver perspective as more in depth view of the system.

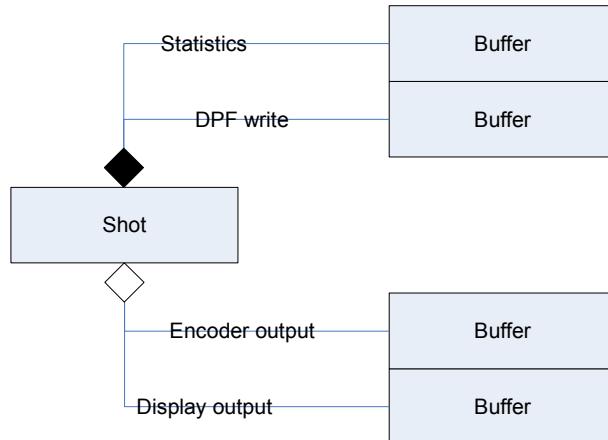


Fig. 7.5: Shot elemets (simplified)

7.7.1 Shot pool size and number of buffers

The Shots represent the pool of simultaneous request that can be pending in the system. That includes the number of request to the HW to keep busy capturing while user-space processes the result. The number of buffers per output is the number of image data circulating in the whole system.

The CI library allows the triggering of a capture with specific Buffers but will always take the 1st available Shot (because the user may have preferences on which buffer to fill but should not care if a Shot is used or not).

For example let's consider a system where a user needs 1 YUV image to give to an encoder. Let's imagine that the encoder requires 2 images to be able to encode 1 image (the previous

² In C the distinction is a pointer for dynamically attached objects while the other Buffers are part of the Shot structure.

one for reference). Let's also assume that encoding 1 frame requires the same time as capture a frame. We also assume that the encoder does not need any other information from the V2500 HW.

We need at least 4 Buffers to keep the V2500 busy at all time. However we can reuse the Shots (we only need 2 to keep HW busy all the time):

- Initial stage: 2 Shots (0 and 1) and 4 Buffers (0 to 3) (*Capture initial steps (t=0, t=1)*. (page 181) t=0)
- 1st capture uses Shot 0 with Buffer 0 (*Capture initial steps (t=0, t=1)*. (page 181) t=1)
- 2nd capture uses Shot 1 with Buffer 1; give result Buffer 0 to the encoder; Shot 0 becomes available (*Capture running steps (t=2, t=3)*. (page 182) t=2).
- 3rd capture uses Shot 0 with Buffer 2; give result Buffer 1 to encoder (reference Buffer 0); Shot 1 becomes available (*Capture running steps (t=2, t=3)*. (page 182) t=3).
- 4th capture uses Shot 1 with Buffer 3; give result Buffer 2 to encoder (reference Buffer 1); Buffer 0 becomes available; Shot 0 becomes available (*Capture running steps (t=4, t=5)*. (page 182) t=4).
- 5th capture uses Shot 0 with Buffer 0; give result Buffer 3 to encoder (reference Buffer 2); Buffer 1 becomes available; Shot 1 becomes available (*Capture running steps (t=4, t=5)*. (page 182) t=5).
- Continue until the end of sequence.

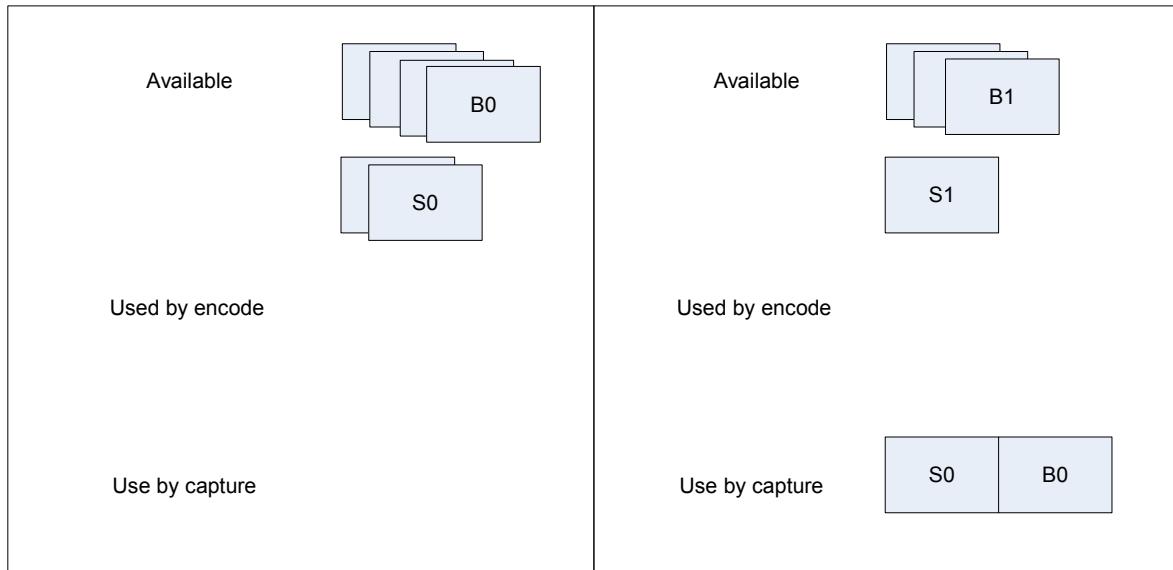
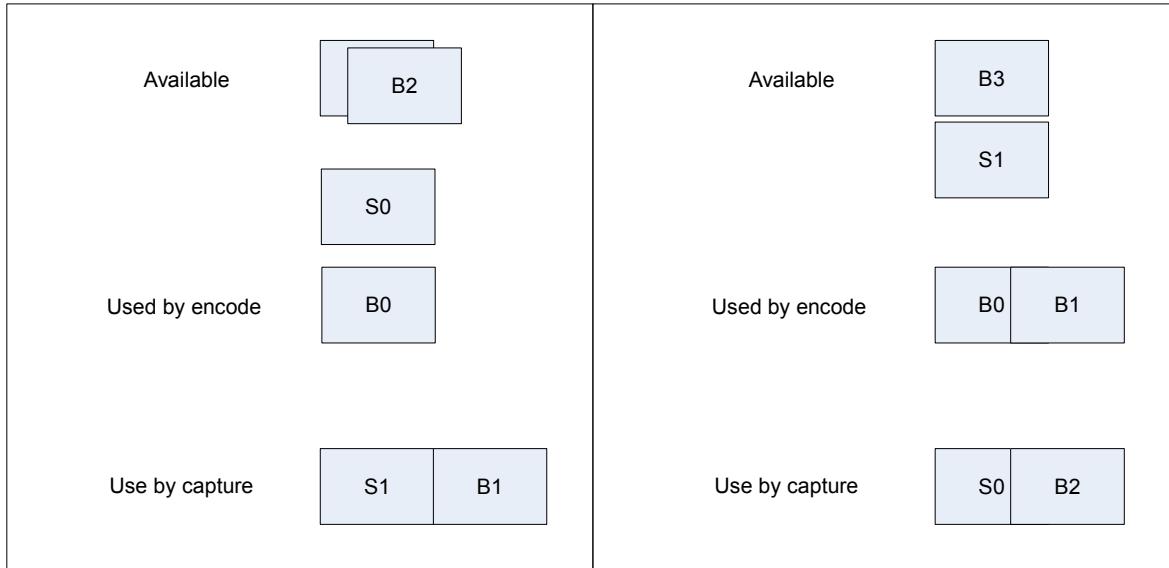
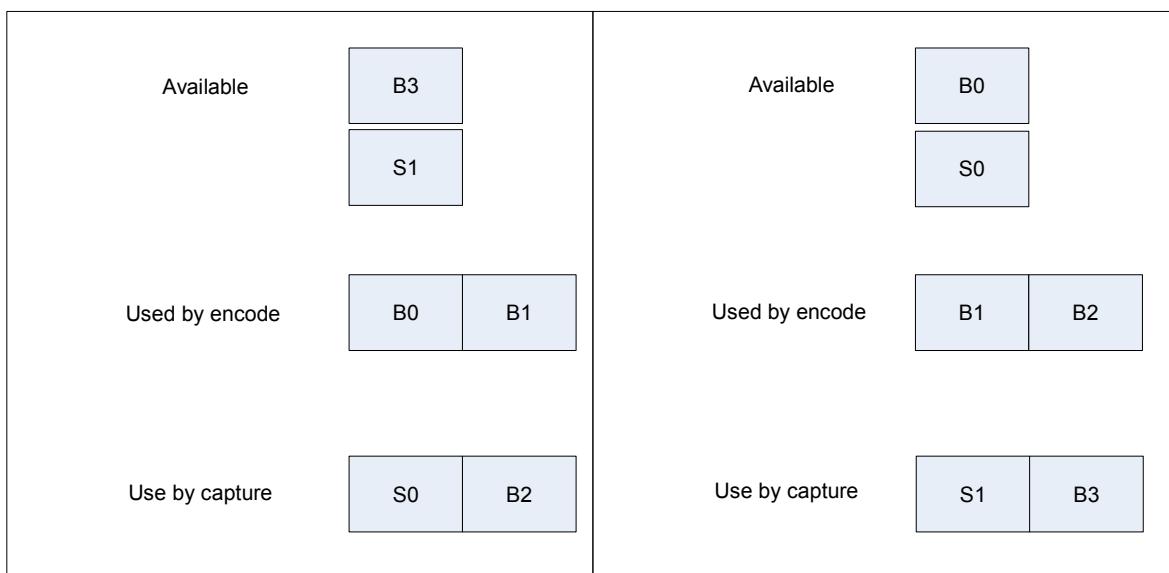


Fig. 7.6: Capture initial steps (t=0, t=1).

7.7.2 User-side perspective

The user-side management of Shots and Buffers is relatively basic. The user-side driver considers Shots and Buffers as consumer would: waiting for them to be available.

Fig. 7.7: Capture running steps ($t=2, t=3$).Fig. 7.8: Capture running steps ($t=4, t=5$).

Storage

In user-side the Shots are stored as an internal structure INT_SHOT (i.e. not accessible by the user of the CI library) that exposes a public CI_SHOT containing the information that the CI user may need.

The Shots are stored in a list of the internal pipeline object (INT_PIPELINE) that the user cannot see (created internally when creating CI_PIPELINE): INT_PIPELINE::sList_shots.

Because the user needs the possibility to specify which Buffer is used for a capture the Buffers are stored in a different list, INT_PIPELINE::sList_buffers, as an internal structure INT_BUFFER. This list is searched when acquiring a captured frame for all specified Buffers and the result is aggregated into an available Shot.

Creation of Buffers

The creation is initialised from the user-side, the application either import information using ion (e.g. CI_PipelineImportBuffer()) or using buffer allocation CI_PipelineAllocateBuffer(). The CI_PIPELINE object has to be registered to kernel-side for either of these functions to work.

The buffer for each output has to be created separately.

For the creation of a buffer the user-space does the following steps:

- Contact the kernel-side with CI_IOCTL_CREATE_BUFF to receive information about the Buffer memory. This includes the Buffer ID and size. The result is stored in a new INT_BUFFER.
- Map the buffer to user space using `mmap()`³
- Add the buffer to INT_PIPELINE::sList_buffers as an available Buffer. The INT_BUFFER has the same ID than the one in the kernel.

Buffers can be de-allocated at any-time as long as the capture is stopped.

Creation of Shots

Shots are allocated by pools. The size of the pool represent the number of simultaneous frames the user wants to request at once (e.g. to use double buffering at least 2 shots should be created).

For each created Shot the following steps are performed:

- Contact the kernel-side with CI_IOCTL_PIPE_ADD and receive information about the newly created Shot's ID
- The Shot ID is stored into a new INT_SHOT and statistics and other accessible information are mapped using `mmap()`.
- The INT_BUFFER is pushed into INT_PIPELINE::sList_shots as available. This Shot has the same shot ID than one in the kernel-side.

Warning: Once allocated a Shot cannot be destroyed until the destruction of its parent Pipeline object.

³ For the moment `mmap()` is used even when using ION allocator, which may be changed to use `gralloc` to retrieve the user-side memory in the future.

Start the capture

This event has no impact on the user-side storage of the Shots and Buffers.

Triggering a Shot

The Shots are queued into the HW capture list when the user side enables it. The triggering can be blocking or non-blocking (which will be explained in the kernel-side section). The CI_PIPELINE object has to be started to perform that operation.

The CI_PIPELINE object allows the user to specify which Buffer should be used for a particular shot (CI_PipelineTriggerSpecifiedShoot()), or simply use the first available ones (CI_PipelineTriggerShoot()). It is not recommended to mix the calls (use either) as the Pipeline object verifies if the Buffer is available or not for capture.

The operation from the user-side is very simple:

- Contact the kernel-side with CI_IOCTL_CAPT_TRG with Buffer ID information for the shot.

The triggering can fail for several reasons, including the fact that the specified buffers are not available.

The triggering is a FIFO, and available Shots have to be acquired to access the image they contain.

Acquiring a Shot

When the user-side needs a Shot it has to query for an available one. The user can choose for the acquisition to be waiting for an available shot or returning with an error when none are found. Note that the kernel-side only waits for a limited amount of time before returning an error even in the former case to avoid infinite wait if the HW is stuck. The process is:

- Contact the kernel side with CI_IOCTL_CAPT_BAQ which will return a Shot ID and Buffer IDs to an available capture.
- The Shot ID is searched in the list of Shots (INT_PIPELINE::sList_shots). The Shot is tagged as acquired and cannot be used by a trigger call until released.
- The Buffer IDs are searched in the list of Buffers (INT_PIPELINE::sList_buffers) and attached to the found Shot. They are tagged as acquired and cannot be used by a trigger call until released.
- The public part of the Shot (CI_SHOT) is returned to the user.

The user can now use the memory and must release the shot when it is no longer required to free up its associated Buffers.

Releasing an acquired Shot

The user-side acknowledges that a Shot is now available for another capture following the operations:

- Contact the kernel-side with CI_IOCTL_CAPT_BRE to signal which Shot and Buffer should now be considered available again

- INT_BUFFER and the INT_SHOT are tagged as available again

Destruction

The Shots and Buffers are destroyed when the CI_PIPELINE object that owns them is destroyed. They cannot be passed to another object. Imported Buffers can of course be re-imported in other objects. At destruction time both Shot and Buffer lists are cleared.

- Each Buffer is unmapped
- Each Shot has its internal memory unmapped
- Contact the kernel-side with CI_IOCTL_PIPE_DEL to signal that the equivalent object should be destroyed

Note: It is also possible to de-register a Buffer using CI_PipelineDeregisterBuffer() but the capture cannot be running for that to occur.

7.7.3 Kernel-side perspective

The kernel-side is contacted by the user-side in regards of Shot and Buffer management. It has to deal with several duties that the user-side does not have to know about:

- Allocation size
- Device MMU handling
- HW waiting list

Storage

The Shots are stored in the kernel-side as KRN_CI_SHOT, which contains user side information (CI_SHOT), internal memory available to user-space (e.g. statistics) and internal memory needed for HW management. Some Buffers can be attached as KRN_CI_BUFFER. Each Shot is stored in a unique KRN_CI_PIPELINE object. A Shot also has a state which specifies the potential usage of the memory.

The Buffers that can be specified by the user are stored as pointers of KRN_CI_BUFFER in a KRN_CI_SHOT.

A KRN_CI_BUFFER contains the information needed to handle the memory with the HW.

The KRN_CI_PIPELINE has several lists where Shots are stored (see *KRN_CI_SHOT states (kernel-side)*. (page 186) or KRN_CI_SHOT_eSTATUS):

- KRN_CI_PIPELINE::sList_available where Shots have the CI_SHOT_AVAILABLE state.
The Shots stored in there can be pushed into the HW queue for capture. They will be marked as pending when a capture is triggered.
- KRN_CI_PIPELINE::sList_pending where Shots have the CI_SHOT_PENDING state.
The Shots stored there are waiting for the HW to process them. They are added when a capture is triggered. The interrupt-handler removes the shot from the list and marks them as processed when the HW completed the capture.

- KRN_CI_PIPELINE::sList_processed where Shots have the CI_SHOT_PROCESSED state.
The Shots stored there were processed by the HW and are waiting to be acquired by user-space.
- KRN_CI_PIPELINE::sList_sent where Shots have the CI_SHOT_SENT state.
The Shots stored there were acquired by the user-side and will be pushed back into the available list when released.

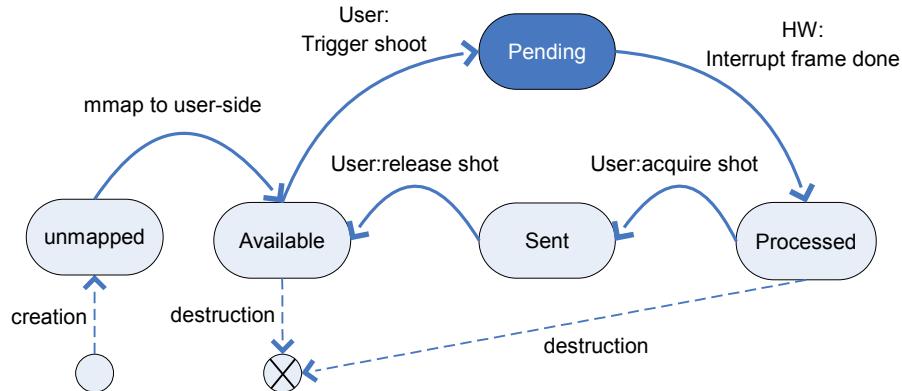


Fig. 7.9: KRN_CI_SHOT states (kernel-side).

Creation of Shots

In the kernel-side the creation is done in 2 actions from the user-side

1. creation of the Shot (CI_IOCTL_PIPE_ADD)
2. mapping of the memory to user-side (`mmap()` called)

CI_IOCTL_PIPE_ADD event received

The kernel-side considers that a Shot should be created when the user-side notifies it. The user-side may provide buffers to import later on. The CI_PIPELINE from user-side has to be registered in the kernel-space for the addition to occur.

- The various buffer sizes of the internal elements are determined from the configuration.
- The physical memory is allocated and virtual memory reserved for the internal buffers
- A unique identifier is created for each internal buffer accessible in user-space. This is returned to user-space to allow them to map them.
- A new KRN_CI_SHOT object is created and added to the KRN_CI_CONNECTION::sList_unmappedShots list until it is mapped in user-space.

`mmap()` called

The `mmap()` system call should be called for every Shot's internal buffer that the user-space has an ID for. For each call the kernel-side will:

- Search for the Shot that contains a buffer with the given ID in KRN_CI_CONNECTION::sList_unmappedShots
- Map the memory to user-side of the given internal buffer (e.g. statistics)
- If all the internal elements for the Shot have been mapped the Shot will be removed from the KRN_CI_CONNECTION::sList_unmappedShots list and added to the KRN_CI_PIPELINE::sList_available one.

Creation of Buffers

In the kernel-side the creation is done in 2 actions from the user-side

1. creation of the Shot (CI_IOCTL_CREATE_BUFF)
2. mapping of the memory to user-side (`mmap()` called)

CI_IOCTL_CREATE_BUFF event received

The kernel-side has 2 use cases when this even is received: either the buffer should be allocated or imported. For both cases the CI_PIPELINE from user-side has to be registered in the kernel-space for the addition to occur.

- Compute the estimated size of the buffer using ci_alloc_info.h functions and the known size from the IIF setup.
- Reserve the virtual memory if using the MMU (tiling may affect that and make it bigger than the actual physical memory allocation is for the 1st address alignment it needs).
- Allocate or import the physical memory for the buffer (and verify that it is big enough when importing).
- Compute a unique identifier for the Buffer that will be returned to user-space for mapping
- Push the Buffer into the KRN_CI_CONNECTION::sList_unmappedBuffers until user-space calls `mmap()`.

`mmap()` called

When `mmap()` is called the Buffer is considered available for captures and added to KRN_CI_PIPELINE::sList_availableBuffers.

Start the capture

When the CI_IOCTL_CAPT_STA event is received the kernel-side has to reserve the HW and configure the device MMU. Each allocated Buffer will be mapped to the device MMU at this point.

Triggering a Shot

When the capture is started the kernel-side will use the CI_IOCTL_CAPT_TRG event to add elements into the HW processing FIFO using the following steps:

- Get the 1st KRN_CI_SHOT from the KRN_CI_PIPELINE::sList_available. This may be a blocking action.
- Searching the list of Buffers KRN_CI_PIPELINE::sList_availableBuffers for the specified buffers and remove them.
- Attach the found Buffers to the retrieved Shot
- Prepare the internal memory with the correct memory. This means updating the linked list addresses and load structure from the configuration.
- Add the Shot to the back of the pending list KRN_CI_PIPELINE::sList_pending
- Submit the capture the HW processing FIFO

HW interrupt signal: capture is over

When the frame is processed the HW emits a FRAME_DONE_ALL interrupt which the kernel-side will catch and do:

- Remove the 1st KRN_CI_SHOT from the KRN_CI_PIPELINE::sList_pending.
- Add the Shot to the KRN_CI_PIPELINE::sList_processed.

Acquiring a Shot

On reception of CI_IOCTL_CAPT_BAQ the kernel-side will:

- Get the 1st KRN_CI_SHOT from the KRN_CI_PIPELINE::sList_processed. This may be a blocking action.
- Push the acquired Shot to the back of the KRN_CI_PIPELINE::sList_sent list.
- Update the host memory (CPU side) for all device-memory to ensure user-space has up-to-date memory
- Return the Shot ID and Buffer IDs to user-side

Releasing an acquired Shot

Once the user-side is done with the memory it should emit a CI_IOCTL_CAPT_BRE event. The kernel-side will process it as:

- Searching for the given Shot ID into the KRN_CI_PIPELINE::sList_sent list.
- Push back all the available Buffers into the KRN_CI_PIPELINE::sList_availableBuffers
- Remove the Shot from the sent list and add it into the KRN_CI_PIPELINE::sList_available one.

Deregister a Buffer

It is possible to de-register a Buffer if the Pipeline is not doing a capture. In that the Buffer will:

- Be removed from the KRN_CI_PIPELINE::sList_availableBuffers.

- Free the allocated device and virtual memory (or release the device memory if imported).

Destruction of the Shots

On destruction of the KRN_CI PIPELINE object the various Shots associated with it will be cleared, that simply means that each Shot internal buffer:

- Free the allocated device and virtual memory

Blocking and non-blocking calls

The 2 lists that the driver has to wait on in the kernel-side

- KRN_CI PIPELINE::sList_available when triggering a shoot (waiting for an available Shot and its internal structures).
- KRN_CI PIPELINE::sList_processed when acquiring a captured Shot (waiting for HW to signal a shot is available).

To allow waiting to occur without having to busily check if any element was added to the processed list the kernel-side driver has a semaphore incremented every-time an interrupt is handled successfully (KRN_CI PIPELINE::sProcessedSem).

The available list does not have an associate semaphore and the triggering will fail if no available shots are pending (because they are released by the user therefore waiting would result in a lockup). However the driver has to ensure that the HW FIFO list is not full and a semaphore is used for that (so the user-side can be kept waiting pushing a frame for capture until the HW FIFO has an available slot). This is implemented with the KRN_CI DRIVER::aListQueue semaphore, incremented when handling interrupts as well.

Waiting on the semaphore is never for an infinite amount of time: a compilation time value defines the maximum amount of time to wait for (CI_SEM_WAIT in `kernel_src/ci_init_km.c`). At the time of writing the value is 5000ms, this value should be more than enough for a frame to be processed by the HW using the external data generator on limited bandwidth system. A timeout on acquiring a frame can be considered as a HW or sensor lockup or failure. The actual waiting time can be changed at insmod time (see *Getting Started Guide* (page 17)).

Chapter 8

ISP Control Library

The ISP Control library (or ISPC/ISPC2) is a high level API for controlling the V2500 pipeline, sensor and includes an implementation of the basic loop control algorithms. This document summarizes the functionalities and architecture of the ISP Control library.

ISPC2 is a C++ implementation designed to be extensible and customizable in terms of pipeline setup modules and control algorithms. It represents an alternative to the historic ISPC1 library implemented in C.

The ISPC library makes use of the MC and CI driver layers which serve as an interface to program the pipeline and program/retrieve shots. The complexity of MC and CI is encapsulated within the library calls and it is not required to directly invoke their functions (unless very specific functionalities are required).

ISPC overview (page 191) depicts the application architecture with the usage of the ISPC library. In this case the sensor API and control loops are integrated within the library. The ISPC objects are aimed to provide all required methods to interact with the pipeline, sensor and control loops in a unified way, therefore simplifying the implementation of applications making use of the system.

In the rest of this document the architecture and implementation of the ISPC library will be described as well as the implemented functionalities and potential extensibility.

As a complementary material the description of the implemented algorithms and HW modules as well as notes on their implementation and configuration can be found in the *Implemented Control Modules* (page 245) section.

8.1 ISPC Architecture

The main class in the ISPC architecture is the Camera class. The Camera class is integrated by several other classes implementing different functionalities for controlling the capture of images. The main classes within the Camera class are the following:

- Pipeline: Encapsulates the functionality for interacting with the CI pipeline in terms of module configuration, buffer allocations, shot capture, etc. Also contains the list of SetupModule list that contain the high-level setups to configure the CI pipeline.
- Sensor: Encapsulates the Sensor API allowing initializing, starting and configuring the physical sensor present in the system.

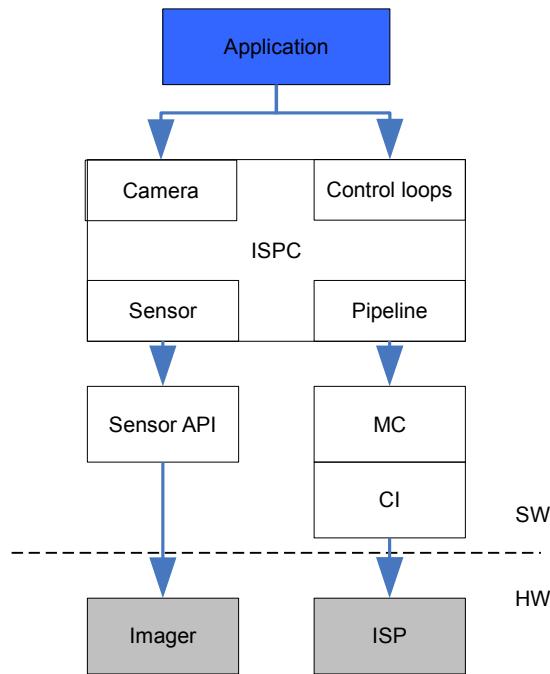


Fig. 8.1: ISPC overview

- Control: The control class is a register that allows to dynamically add loop control algorithms to be run periodically during the capture process and that allow to update the pipeline and sensor configuration while capturing according to the scene characteristics (most typically will include auto exposure, auto focus and auto white balance algorithms).

The ISPC::Camera relations figure shows a class diagram of the Camera organization. The Camera instance is composed by an instance of the Pipeline, one instance of the Sensor and one instance of the Control classes with the functionalities described above.

The Pipeline instance can have registered an arbitrary number of SetupModule instances which serve for the configuration of the different modules of the pipeline.

On the other hand, the Control object can have registered an arbitrary number of ControlModule instances which typically would implement the control loop algorithms or any other algorithms requiring a periodic execution.

Finally the Sensor object is typically owned by the Camera object but can be shared between several Camera objects. In case of sharing the Camera object uses the Sensor as a passive object (only to retrieve information) and will not destroy it when destroyed.

The Figure *ISPC classes communication* (page 192) shows the internal interaction between the Pipeline, Sensor and Control classes which integrate the Camera class. The Pipeline instance is in charge of the pipeline and each SetupModule object stored in it provides a number of configuration parameters and a setup function in charge of translating those high level parameters into lower level parameters that are passed to the MC and CI layers to be programmed in the pipeline hardware (see more details *Pipeline and SetupModule Classes* (page 200)).

The Sensor instance serves for encapsulating the “Sensor API” and provides functionality for setting and querying about the sensor capture parameters (most importantly the exposure time and gain) and can be easily extended to provide additional functionality for setting up advanced functionalities included in the Sensor API.

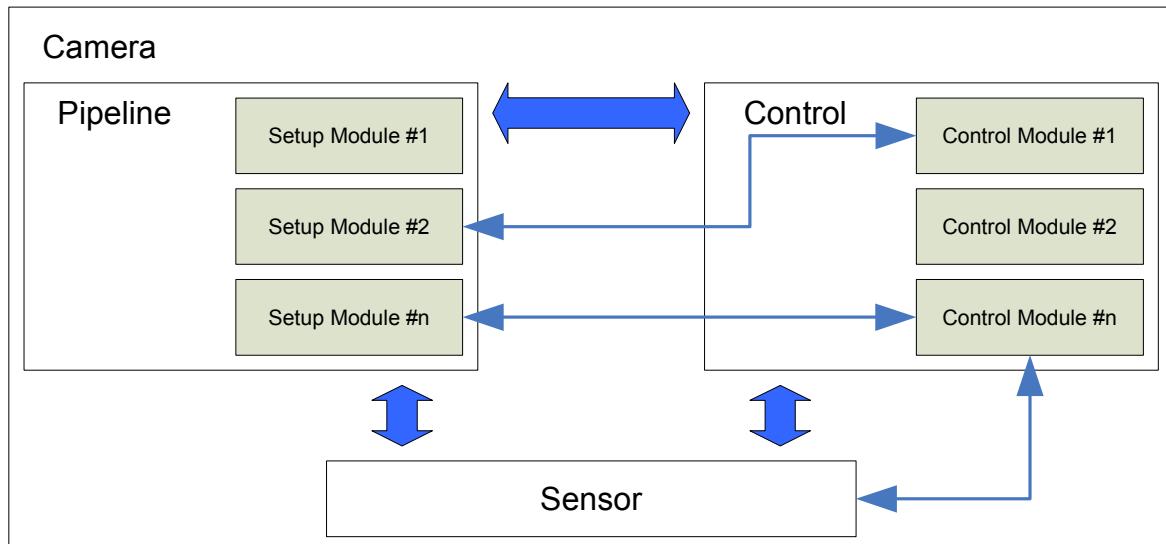
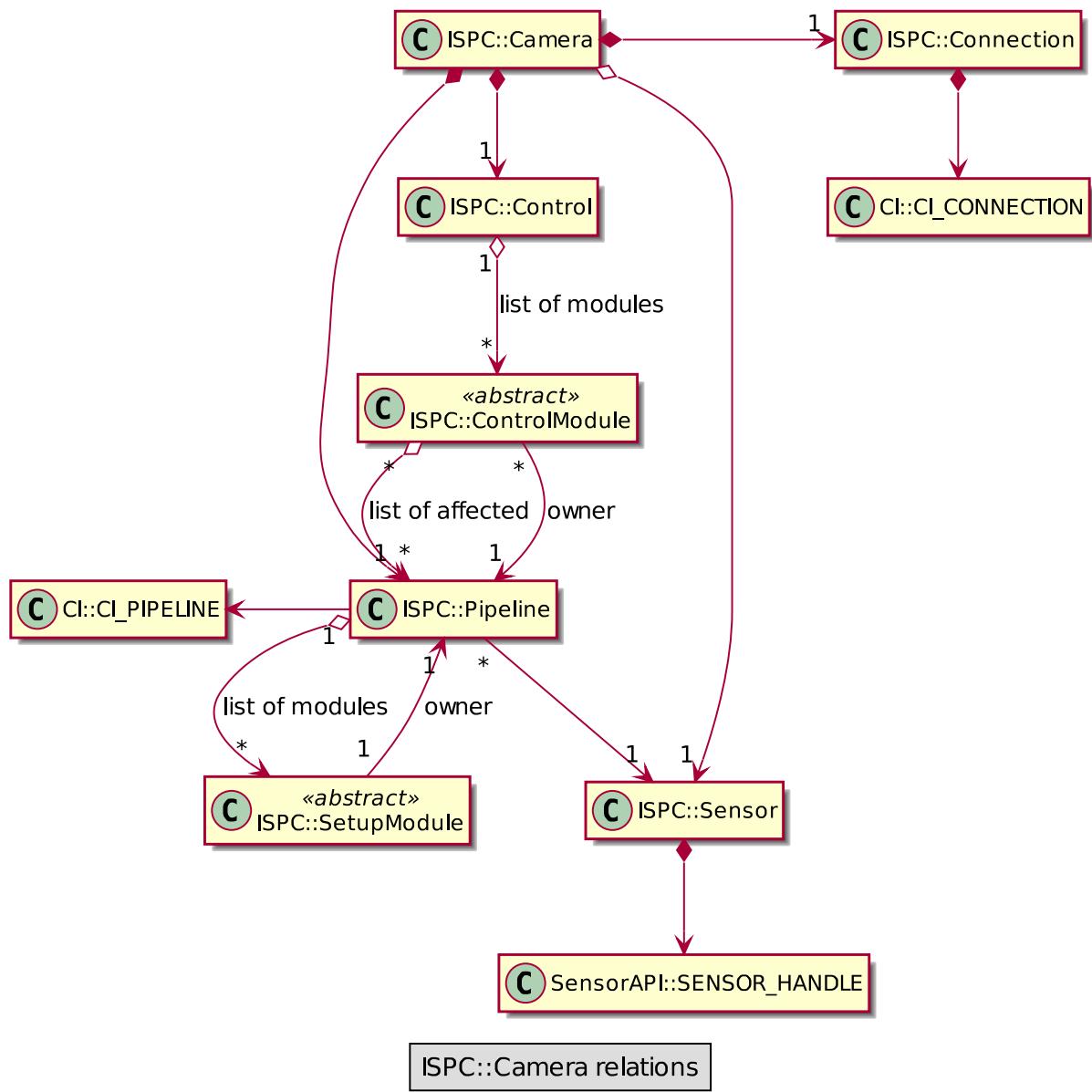


Fig. 8.2: ISPC classes communication

The Control instance is a registry where a number of ControlModule instances can be registered. A ControlModule is intended to correspond with a module which must operate in a continuous or periodical basis to control different aspects of the pipeline configuration in an automatic way. For example the automatic exposure, auto-focus, auto white balance and other loop control algorithms are implemented in the ISPC library as ControlModule instances (actually inheriting from the ControlModule class and adding specialized functionality). The Control class is in charge of requesting updates from the registered ControlModule instances when required.

The Figure *ISPC classes communication* (page 192) also shows some examples of communications between the different modules. Both the Module instances in the Pipeline class and the ControlModule instances in the Control instance can have access to the Sensor instance through their Pipeline “owner”. Such access is available for several reasons. For example, in order to set up different elements of the HW pipeline, it may be required to get information about the sensor size, modes, etc. On the other hand it is also possible that a requirement is to set up different capture settings from one of the ControlModule instances, for example the one implementing the auto exposure algorithm.

Modules in the Control registry also have access to the Pipeline modules as it is required many times to change configuration parameters of the pipeline from one of the ControlModule instances and that is normally carried out through the corresponding SetupModule in the Pipeline class.

8.2 Camera Class

This section gives more details about the Camera class and its usage.

8.2.1 Initialisation

The Camera class is the principal interface for controlling the pipeline and sensor in an application making use of the ISPC library. Some details of the composition of the Camera class have been provided in the previous section (see Figure in *ISPC Architecture* (page 190)) and this section will be focused on its usage.

The Figure *Camera Instance states* (page 194) shows the set of different internal states a Camera instance goes through in its lifetime. Creating an instance of the Camera class is straightforward just providing a context number to use and the sensor identifier (which passes through the Sensor class to the Sensor API).

The Camera constructor takes care to the connection to the CI driver but does not register modules. The CameraFactory class can be used to populate a Camera object with the relevant modules according to the discovered HW (CameraFactory::populateCameraFromHWVersion()). The modules can also be added manually as needed.

Note: The factory provides a list of Control modules but does not add them to the Camera.

As depicted in Figure *Camera Instance states* (page 194), once the setup modules have been registered the Camera instance will be in CAM_REGISTERED state which means we have a set of setup functions to be used as an interface with the HW pipeline and will permit its configuration. From such state we are ready to setup up the pipeline modules by calling setupModules(), which calls all the setup functions of every SetupModule instance registered in the Pipeline, or loadParameters() which allows each registered Module instance to load the parameters from a

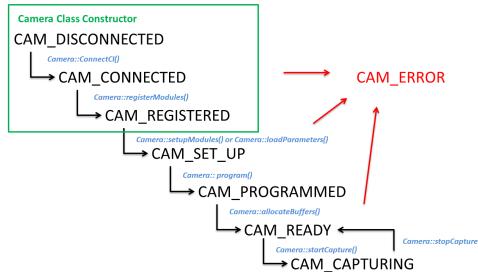


Fig. 8.3: Camera Instance states

ParameterList object (see *Several Camera sharing a Sensor* (page 280)) followed by a call to its setup function. The Camera will be in **CAM_SET_UP** state at that point.

The next step (see Figure *Camera Instance states* (page 194)) would be the step in which the HW pipeline is actually programmed with the configuration obtained after calling to each Module setup function. Once the HW is programmed it is possible (and required prior to capture images) to request the allocation of buffers. The HW must be configured beforehand to allow a proper size calculation for the allocated buffers.

The allocation can be done using the `allocateBufferPool()` method which allocates the amount of required buffers and shots. The Camera state will change to **CAM_READY** which indicates that the Camera is ready to start a capture (see *Buffer allocation vs buffer importation* (page 196) for more details about allocation or even importation of image data).

The last step, as shown in the Figure *Camera Instance states* (page 194) would be the call to `startCapture()` which will reserve the HW for its usage by the application. The hardware can be released with a call to `stopCapture()`. The following pseudo-code shows an example of the camera initialization process (adapted from the `ISPC_capture` application) which enables 1 control algorithm (auto exposure):

```

ISPC::Camera camera(0, pszSensor, uiSensorMode);
ISPC::ParameterList parameters;
ISPC::ControlAE *pAE = new ISPC::ControlAE();
IMG_RESULT ret;

ret = ISPC::CameraFactory::populateCameraFromHWVersion(camera,
    camera.getSensor());
if (IMG_SUCCESS != ret) {
    fprintf(stderr, "Error: failed to populate camera modules\n");
    goto exit_point;
}

ret = camera.registerControlModule(pAE);
if (IMG_SUCCESS != ret) {
    fprintf(stderr, "Error: failed to register AE to camera\n");
    goto exit_point;
}

// loads defaults if the file is empty
ret = camera.loadParameters(parameters);
if (IMG_SUCCESS != ret) {
    fprintf(stderr, "Error: failed to load camera configuration\n");
    goto exit_point;
}

```

```

ret = camera.setupModules();
if (IMG_SUCCESS != ret) {
    fprintf(stderr, "Error: failed to setup camera\n");
    goto exit_point;
}

ret = camera.program();
if (IMG_SUCCESS != ret) {
    fprintf(stderr, "Error: failed to program camera\n");
    goto exit_point;
}

ret = camera.allocateBufferPool(uiNBuffers);
if (IMG_SUCCESS != ret) {
    fprintf(stderr, "Error: failed to allocate %d buffers!\n", uiNBuffers);
    goto exit_point;
}

// preparation of the output files should be done here

ret = camera.startCapture();
if (IMG_SUCCESS != ret) {
    fprintf(stderr, "Error: failed to start the capture!\n");
    goto exit_point;
}

```

8.2.2 Capture

Shots can be captured if the Camera instance is in CAM_CAPTURING state. First a shot must be en-queued in the pipeline by calling enqueueShot(). The captured shot is returned as a Shot which is populated when getShot() method is called. The Shot structure contains the captured image(s) and metadata.

Once we are done with the captured shot we must return it to the pool by releasing it, so it can be reused for further captures. This is done with the call to the releaseShot() method. This is an example of capture and release sequence (adapted from ISPC_capture application):

```

ISPC::Shot frame;
IMG_RESULT ret;

// add one element to the capture queue
// will be filled when HW receives a frame from the Sensor
ret = camera.enqueueShot();
if (IMG_SUCCESS != ret) {
    fprintf(stderr, "Error: failed to enqueue shot\n");
    goto exit_point;
}

// may be blocking until the frame is received
ret = camera.acquireShot(frame);
if (IMG_SUCCESS != ret) {
    fprintf(stderr, "Error: failed to get shot\n");
    goto exit_point;
}

// use the frame data as needed (e.g. save to disk)

```

```
// no need to call Control with the metadata information (it is done when
// acquiring a Shot)

ret = camera.releaseShot(frame);
if (IMG_SUCCESS != ret) {
    fprintf(stderr, "Error: failed to release shot\n");
    goto exit_point;
}
```

Where we can carry out any desired operations with the captured shot after the `getShot()` call and before the shot is released with the call to `releaseShot()`.

8.2.3 Stopping the capture

Once we are done with the capture process the HW must be released with the call to `stopCapture()`. All pending captures will be cancelled and already non-acquired already captured shots will be lost but the acquired ones are still valid until released:

```
ret = camera.stopCapture();
```

The allocated shots and buffers will be destroyed when the Camera object is destroyed but it is possible to remove image buffers one by one.

Warning: Important: It must be noted that the registered SetupModule and ControlModule instances (more details in the following sections) are assumed to be freed by the Camera class once they are registered. Therefore they shouldn't be freed outside the Camera class and they should be dynamically allocated to avoid errors when deleting the allocated SetupModule and ControlModule instances.

8.2.4 Buffer allocation vs buffer importation

The above example on how to use the Camera object is allocating the memory. But it is possible that on some systems (e.g. Android) the image buffers should be pre-allocated by another component and this memory should be “shared” with the ISPC library. To use such buffer an “importation” mechanism is available in the ISPC library using the mechanisms provided by the low level libraries (CI).

Buffers VS Shots

The section *Shot and Buffer lifecycle* (page 179) details the difference a `CI_SHOT` and `CI_BUFFER` (not an actual structure in user-space). The same differentiation applies in ISPC:

- An ISPC::Buffer is considered as an image buffer; it can be either allocated by the V2500 driver or imported from an external driver. Every Buffer is identified by a unique number. Buffers can be added and removed at will as long as the Camera is not in the `CAM_CAPTURING` state.

Note: When importing a buffer it is assumed that the `SYS_Mem` implementation of the ISP kernel module can recognise the given buffer identifier.

- A ISPC::Shot is the glue between several Buffer. It also contains the additional internal information that cannot be imported (i.e. the meta-data).

Both Shot and Buffer should be pre-allocated as they are used in the capture queues but can be allocated after the capture is started at the cost of performance.

Different output sizes

The default allocation size for all outputs is the sensor's resolution. However because the Encoder and Display output can be scaled down it is possible to modify the maximum size that either will be configured for using Camera::setEncoderDimensions() and Camera::setDisplayDimensions().

The allocation process is described in the *Different Output Size* (page 161) section. In ISPC the procedure is the same using Camera::allocateBuffer() to specify the buffer type and size.

Allocating a pool or importing image buffers

The example in *Initialisation* (page 193) uses the Camera::allocateBufferPool(n) method which allocates n Shots and Buffers with default size.

It is possible to allocate only Shots using Camera::addShots(n). However Buffers have to be allocated one by one for each output using the Camera::allocateBuffer() method.

The alternative to the Buffer allocation is importing it. When importing a Buffer it is assumed that the memory has already been allocated using a common allocator (in the Android case gralloc and ION) supported by the CI kernel-side driver (Platform_MemImport() function). This common allocator will produce an identifier that can be given the Camera::importBuffer() to retrieve that memory.

Populating a specific Buffer when enqueueing shots

When creating a Buffer a unique buffer identifier is provided as a return value. This identifier can be used to trigger frame captures that will populate specific Buffers using Camera::enqueueSpecifiedShot().

Using the Camera::enqueueShot() function will enqueue the 1st available Buffer and Shot therefore it is not recommended to call both functions on the same object (it should be known from the start of day if specific shots need to be populated or if any will do).

When specifying a Buffer for the capture it is possible to change the memory layout of the output (change the stride or offsets) using a similar mechanism than in CI (see *Different memory layout* (page 162)). In ISPC the Camera::enqueueSpecifiedShot() function is used and takes a CI_BUFFID structure as input.

Using the exmaple computation from *2 Frames example* (page 349) we can therefore use the buffer as following:

```
unsigned int size = 1567808;
unsigned int stride = 1088;
unsigned int a_off_y = 0, a_off_cbcr = 783360,
             b_off_y = 1088, b_off_cbcr = 784448;

/*
 * assumes camera is connected, configured but not started yet
```

```

* the call could also be to Camera::importBuffer()
*/
int buffId = 0;

ret = camera.allocateBuffer(CI_TYPE_ENCODER, size,
    false, &buffId);

/*
* the capture should now be started so that we can configure the
* trigger of a frame buffId could be found from a list after
* allocation or left to 0 and let the 1st available buffer be used
* (in that case the assumption is that all the YUV buffers were
* allocated with the correct size
*/

CI_BUFFID toTrigger = CI_BUFFID();
// C struct constructor that should be equivalent than a memset to 0

toTrigger.encId = buffId;
toTrigger.encStrideY = stride;
toTrigger.encStrideC = stride;
if (is_a)
{
    toTrigger.encOffsetY = a_off_y;
    toTrigger.encOffsetC = a_off_cbcr;
}
else
{
    toTrigger.encOffsetY = b_off_y;
    toTrigger.encOffsetC = b_off_cbcr;
}

ret = camera.enqueueSpecifiedShot(toTrigger);

```

Lifetime of images

In a system that supports image buffer importation it is very likely that the Camera object should be initialised beforehand and configured with a given amount of Shot (to cope with the latency and double buffering as explained in the *Shot pool size and number of buffers* (page 180) section). When the system requires a capture to be done it will provide a handful of image buffers to be imported just before the capture is started. As soon as the capture terminates however the system should be able to reclaim the given buffers (to liberate the memory) and therefore they should be “de-registered” from the Camera object. This is possible with the Camera::deregisterBuffer() function.

8.2.5 Sensor ownership

According to the constructor used for the Camera the object is assumed to own or not the Sensor. The ownership is assumed when the sensor is created by the Camera object but can always be modified later using setter functions.

```

// creates the sensor therefore owns it
Camera(unsigned int contextNumber, int sensorId, int sensorMode=0);

```

```
// is given a sensor therefore does not own it
Camera(unsigned int contextNumber, Sensor *pSensor);

// to known if the sensor is owned by this object
bool ownsSensor() const;
// to change the fact that the sensor is owned
void setOwnSensor(bool b);
```

Once a Sensor is owned by a Camera it is assumed that the Camera has the right to modify it:

- Configure the sensor
- Start and stop the sensor
- Destroy the sensor object when destroyed

Warning: The control modules will not verify if the Camera they are part of owns the Sensor and will always modify it. Ensure that control modules that modify sensor settings are registered in Camera instances that owns their sensor.

This does not apply to setup modules in general as they usually only need information about the sensor.

Ownership usages

It is possible to play with Sensor ownership to share a sensor with multiple Cameras to handle more outputs or apply different configurations at once as explained in the *Several Camera sharing a Sensor* (page 280) section. It is also possible to use the sensor owner as a way to create and manage the sensor from an external point than the Camera without having to re-implement all the Control and Pipeline handling.

Warning: In the second case remember that the sensor has to be started **AFTER** the ISP pipeline and configured before programming it.

```
ISPC::Sensor appSensor;
ISPC::Camera camera(0, &appSensor);
ISPC::ParameterList parameters;
IMG_RESULT ret;

// configure sensor - the size and information will now be correct
// HAS TO BE DONE BEFORE CONFIGURING THE CAMERA
// sensor will start sending frames when enabled
appSensor.configure(sensorMode);

ret = ISPC::CameraFactory::populateCameraFromHWVersion(camera, &appSensor);
if (IMG_SUCCESS != ret) {
    fprintf(stderr, "Error: failed to populate camera modules\n");
    goto exit_point;
}

// loads defaults if the file is empty
ret = camera.loadParameters(parameters);
if (IMG_SUCCESS != ret) {
    fprintf(stderr, "Error: failed to load camera configuration\n");
    goto exit_point;
```

```

}

ret = camera.setupModules();
if (IMG_SUCCESS != ret) {
    fprintf(stderr, "Error: failed to setup camera\n");
    goto exit_point;
}

ret = camera.program();
if (IMG_SUCCESS != ret) {
    fprintf(stderr, "Error: failed to program camera\n");
    goto exit_point;
}

ret = camera.allocateBufferPool(uiNBuffers);
if (IMG_SUCCESS != ret) {
    fprintf(stderr, "Error: failed to allocate %d buffers!\n", uiNBuffers);
    goto exit_point;
}

// preparation of the output files should be done here

// start ISP pipeline - sensor is not enabled therefore it does not send
// frames yet
ret = camera.startCapture();
if (IMG_SUCCESS != ret) {
    fprintf(stderr, "Error: failed to start the capture!\n");
    goto exit_point;
}

ret = appSensor.enable();
if (IMG_SUCCESS != ret) {
    fprintf(stderr, "Error: failed to enable the sensor!\n");
    camera.stopCapture();
    goto exit_point;
}

// do the capture here

appSensor.disable();

camera.stopCapture();

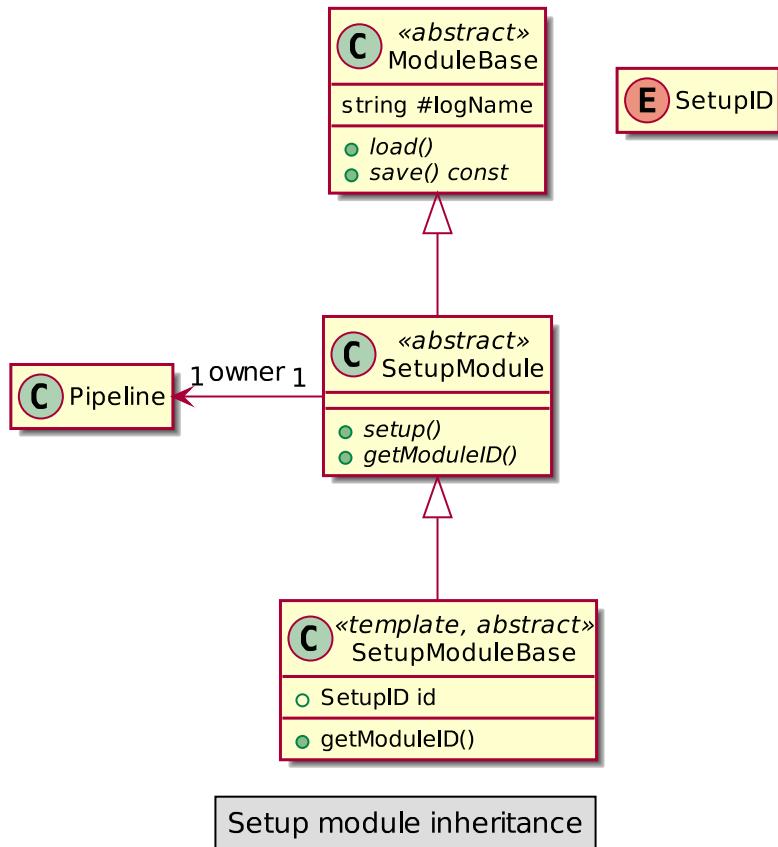
// this is the end of the C++ context - both camera and appSensor will
// be destroyed

```

8.3 Pipeline and SetupModule Classes

Within the Camera object the Pipeline is in charge of managing the configuration of the HW ISP. The Pipeline is able to register an arbitrary number of SetupModule instances.

The SetupModule class is actually an interface intended to be extended so each actual HW module has a corresponding class specialised on its configuration as detailed in the Setup Module inheritance figure.



Each implemented “Setup Module” class will declare a number of public attributes which will represent the HW module setup parameters.

The ModuleBase abstract class is the root for every module. It defines the abstract **load()** and **save()** function that allow the manipulation of high-level parameters using a ParameterList object. This class is used by both Setup and Control modules.

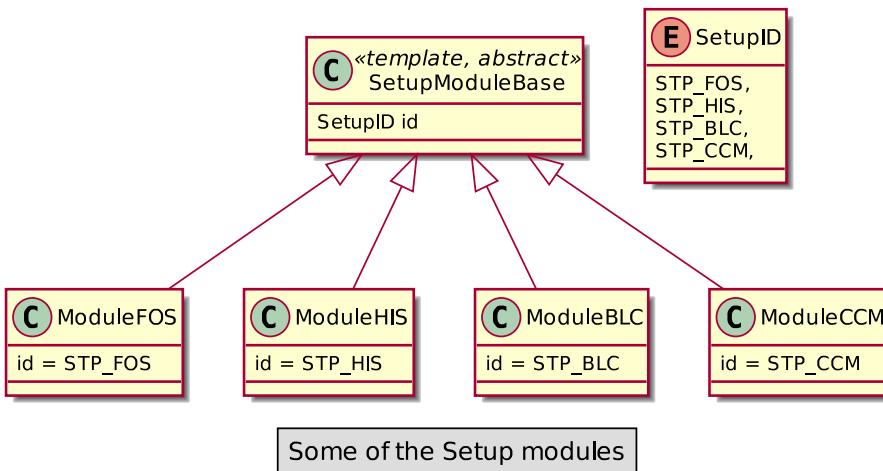
The SetupModule class defines the abstract **setup()** method that will convert the parameters into a set of lower level parameters to be programmed in the HW.

An intermediate class is used in between the SetupModule and actual module implementation to encapsulate some template declarations. Each implemented module uses a unique SetupID value that can be used to retrieve the module using a simple template function in the Pipeline as explained in the section [Module Setup and Update](#) (page 202).

The ISPC library already implements a setup module for each existing HW module which is registered in the Camera when it is created; some of the HW v1 modules are shown here as an example.

When the Pipeline::registerModule() method is invoked the setup modules is registered in the Pipeline instance, the order of registration should not matter. If it was required to create a Pipeline with a different set of modules the CameraFactory class could be extended to generate a different list of modules. Alternatively the Pipeline instance created within the Camera object has public access, as does its elements; this includes methods for clearing the SetupModule registry and manually registering other setup modules (see the doxygen documentation for more details).

Every module is registered with a unique id from the list in ISPC::SetupID (see Module.h) so it is not possible to register two different modules with the same ID. The ID's usually correspond



to actual HW modules present in the pipeline but some additional ones exist, for example the additional ID named `STP_OUT` corresponds to the *Output formats (OUT)* (page 210) which control the output pixel format of the pipeline.

8.3.1 Module Setup and Update

As explained in the previous section each module registered in the camera Pipeline instance is associated with a unique ID. Such ID can be used to retrieve the associated setup module registered in the Pipeline. There are two types of getter functions declared:

- `SetupModule* Pipeline::getModule(moduleID id)` method returns a `SetupModule` base class type pointer to the instance of module associated with a given ID
- `T* Pipeline::getModule<T>()` returns a pointer to the object of type `T` derived from `SetupModule` class.

Such pointers can be used to modify setup parameters as in the following examples:

```

// non template
static_cast<ModuleWBC*>(pipeline->getModule(ISPC::WBC))->WBGAIN[0] = 15.6;
// template
pipeline->getModule<ModuleWBC>()->requestUpdate();

```

Usage of this template method avoids the need of casting and thus helps to have cleaner code and fewer bugs. However direct access as in the example is discouraged (the module could not exist and `getModule()` would return 0).

This is a common way in which the control loop algorithms can access the setup parameters of any module in the pipeline (a `ControlModule` instance keeps a list of `Pipeline` instance to update). The second line in the example is a call to the `ISPC::SetupModule::requestUpdate()` method which marks a given module for being updated. The changes in a given module parameters are not applied until the corresponding module `SetupModule::setup()` function is called.

There are several methods to invoke the registered module's setup functions from a Camera or a Pipeline instance, here some of the Camera ones:

- `Camera::setupModules()` update all modules by calling all `setup()` functions
- `Camera::setupModule()` update a specific module by running its `setup()` function

Refer to the doxygen documentation for more details on the other functions.

Note: The Pipeline class has a more detailed update function (e.g. update only the modules that requested an update). To understand the previous example it must be noted that every time a new shot is queued for a new capture, the Camera class calls the Pipeline::setupRequested() method which runs the setup function for every setup module that has previously received a requestUpdate() call. Therefore it is not needed to update the configuration of each module but only of those with changes and with a request for update.

Alternatively it is also possible to directly call the SetupModule::setup() function of any retrieved module instead of marking it for later update or delegating its update to the Camera.

- *Implemented Setup Modules* (page 203)
- *Implemented Statistics Modules* (page 233)

Implemented Setup Modules

The following sections will detail what the modules provide as corrections. It is mainly composed of the high level parameters loaded for the SW modules. More information about the modules order is available in the *V2500 modules* (page 2) section.

Black Level Correction (BLC)

Correct the black level of the sensor by a fixed offset per channel.

Note: The HW also supports a black average mode but current implementation does not support it.

Warning: The BLC module in HW only applies the black correction. The system black is actually applied in the LSH module in HW but configured in the BLC in SW for simplicity

This module is part of the tuning procedure: *Sensor BLC tuning* (page 130).

BLC High Level Parameters

BLC_SENSOR_BLACK: Format: int[4] range [-128,127]

Defaults: 127 127 127 127

Sensor black level in R, G, G, B order (regardless of sensor mosaic).

Consider as HW fixed point [s7.1] scaled to 8-bit image dynamic range in floating point terms. The parameter does not cover full dynamic range of the image.

BLC_SYS_BLACK: Format: int range [0,32767]

Defaults: 64

Desired internal black level. Consider as HW fixed point [u8.2].

Colour Correction Matrix (CCM)

Colour Correction matrix used for white balance in RGB domain.

This module is part of the tuning procedure: *White Balance Correction Tuning* (page 131).

CCM High Level Parameters

CCM_MATRIX: Format: double[9] range [-3,3]

Defaults: 1 0 0 0 1 0 0 0 1 (identity)

3x3 row-wise colour correction matrix. First three values convert input R,G,B into output R.

CCM_OFFSETS: Format: double[3] range [-32768,32767]

Defaults: 0 0 0

Offset vector added as HW fixed point [s9.3].

Display Gamut Mapper (DGM)

Gamut modifier after CCM and before Gamma is applied. It is similar to the *Main Gamut Mapper (MGM)* (page 216) but at a later point in the pipeline.

Gamut Algorithm overview The Gamut algorithm is the same for the MGM and DGM modules. The HW block is composed of several parameters that allow it to be set up to do any conversion from clipping to scaling. The mapping operation uses a simple two-slope scaling depicted in Figure *Gamut transfer and gains curve* (page 205). The gamut mapper processes a single pixel at a time without any consideration of the locality.

The components of pixels input are individually clipped to an extended gamut range (normally equivalent to xvYCC) set by *clip min* and *clip max*.

```
for ch = 1:3
    pixel[ch] = max(clip_min, pixel[ch])
    pixel[ch] = min(clip_max, pixel[ch])
end for
```

In the first stage of the mapping, two variables are created to control the transfer of negative components to the complementary colour. Negative components are multiplied by *slope 0*.

```
for ch = 1:3
    if (pixel[ch] < 0) then
        var_0[ch] = pixel[ch] * slope[0]
        var_1[ch] = pixel[ch]
    else
        var_0[ch] = pixel[ch]
        var_1[ch] = 0
    end if
end for
```

For tuning the strength of the effect, each transfer (negative red to green, negative red to blue, etc. for all combinations) is weighted by a coefficient that is nominally -0.5 but can range from 0 to -2.0.

```

pixel[0] = var_0[0] + coeff[0] * var_1[1] + coeff[1] * var_1[2]
pixel[1] = var_0[1] + coeff[2] * var_1[2] + coeff[3] * var_1[0]
pixel[2] = var_0[2] + coeff[4] * var_1[0] + coeff[5] * var_1[1]

```

In the next stage, all three components are mapped through a two segment gain curve in such a way that the hue is preserved by maintaining the component ratios. The largest component is used to lookup a gain factor which is applied to all three components.

```

if (largest < point[1]) then
    gain = slope[1]
else
    gain = slope[1] - slope[2] * (largest - point[1])
end if

```

Finally, the pixel is multiplied by the gain.

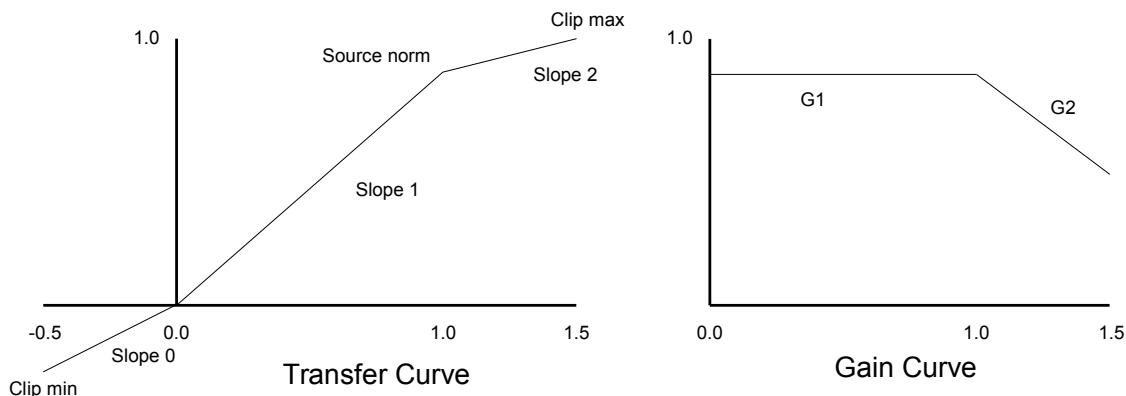


Fig. 8.4: Gamut transfer and gains curve

DGM High Level Parameters

Note: Similar to [Main Gamut Mapper \(MGM\)](#) (page 216) but in different point in the pipeline.

DGM_CLIP_MAX: Format: double range [0,2]

Defaults: 1.5

The upper clip limit common to all three input channels.

DGM_SRC_NORM: Format: double range [0,2]

Defaults: 1

Defines the knee point on the input side of the transfer curve. Corresponds to the upper limit of normal gamut at the source.

DGM_CLIP_MIN: Format: double range [-1,0]

Defaults: -0.5

This is the lower clip limit common to all three input channels.

DGM_COEFF: Format: double[6] range [-2,2]

Defaults: 0 0 0 0 0 0

Scaling factors for converting negative extended range to increased positive range.

0: green to red, 1: blue to red, 2: blue to green, 3: red to green, 4: red to blue, 5: green to blue

DGM_SLOPE: Format: double[3] range [0,2]

Defaults: 1 1 1

0: The conversion gain for the negative sector of the input range. Applied after the input DGM_CLIP_MIN.

1: The conversion gain for the input range from 0 to DGM_SRC_NORM.

2: The conversion gain for the input range from DGM_SRC_NORM to DGM_CLIP_MAX.

Denoiser (DNS/SHA_DN)

This Module configures several HW blocks: the primary denoiser (DNS) and the denoiser part of the SHA module.

The primary denoiser reduces noise before the defective pixels are corrected while the secondary denoiser is used to avoid artefacts after the MIE and TNM modified the image.

This module is part of the tuning procedures: *Primary Denoiser Tuning* (page 132) and *Sharpening and Secondary denoiser Tuning* (page 134).

Denoiser High Level Parameters

Primary denoiser (DNS)

DNS_COMBINE_ENABLE: Format: bool range {0,1}

Defaults: 1

Combine green channels during denoising. This should be enabled and never changed.

DNS_STRENGTH: Format: double range [0,6]

Defaults: 0

Denoising strength.

DNS_GREYSC_PIXTHRESH_MULT: Format: double range [0.07,14.5]

Defaults: 0.25

Pixel threshold multiplier used to adjust the sensitivity of colour differences for the greyscale weights. This value is typically around 0.25.

Secondary Denoiser (part of SHA in HW)

SHA_DENOISE_BYPASS: Format: bool range {0,1}

Defaults: 0

Enable (0) or Disable (1) the secondary denoiser in the HW SHA block.

SHA_DN_SIGMA_MULTIPLIER: Format: double range [0,16]**Defaults:** 1

Multiplier to apply to sharpening denoiser standard deviation (sigma curve) (1.0=no effect, <1.0=less denoising, >1.0=more denoising).

SHA_DN_TAU_MULTIPLIER: Format: double range [0,16]**Defaults:** 1

Multiplier to apply to the driver-determined sharpening denoiser edge avoid effort (tau curve) (1.0=no effect, <1.0=less edge avoidance, >1.0=more edge avoidance).

Sensor information for denoiser This is information read from the sensor object but can be saved to record the value that was used:

Gain: Multiplier used for the capture of the frame.

Read Noise: Standard deviation of noise when reading pixel value off a sensor in electrons.

Sensor bit depth: Size of the data from the sensor.

Sensor well depth: Maximum number of electrons a sensor pixel can collect before clipping.

Values ported to the sensor driver should be provided by the company providing the sensor.

Warning: The information used to be loaded from following parameters which are now deprecated. The information is loaded directly from the Sensor object attached to the Pipeline that owns the module.

DNS_ISO_GAIN: Deprecated, saved as SENSOR_GAIN.

DNS_READ_NOISE: Deprecated, see SENSOR_READ_NOISE.

DNS_SENSOR_BITDEPTH: Deprecated, see SENSOR_BITDEPTH.

DNS_SENSOR_WELLDEPTH: Deprecated, see SENSOR_WELL_DEPTH.

See *Sensor High-level Parameters* (page 279) for details on the sensor high level parameters.

Defective Pixels Fixing (DPF)

The Defective Pixel Fixing module is responsible for detecting and correcting potential defective pixels from the sensor. It is also considered as a statistics module as it can output a list of detected defects.

The defective pixels are detected by analysing the surrounding pixels colours. The neighbouring pixels also provide a candidate replacement value if the difference between the current pixel predicted value exceeds the configured threshold.

Additionally the map of defects can be loaded from a customised defect map (see section *Defective Pixels Fixing read map (DPF)* (page 47) for details about the format). The loading of defects is done using a **shared** HW buffer amongst all contexts therefore the map size is limited when using several contexts at once. The size of this shared context is configuration dependent and the value reported by the HW is available in the CI_HWINFO::config_ui32DPFInternalSize attribute.

Finally all discovered defects (excluding the one loaded) can be saved as statistics.

This module is part of the tuning procedure: *Defective Pixels Tuning* (page 132).

DPF High Level Parameters

DPF_DETECT_ENABLE: Format: bool range {0,1}

Defaults: 0

Enable HW detection of defective pixels (uses threshold and weight).

DPF_READ_MAP_ENABLE: Format: bool range {0,1}

Defaults: 0

Enable the usage of the read map defined by DPF_READ_MAP_FILE.

DPF_WRITE_MAP_ENABLE: Format: bool range {0,1}

Defaults: 0

Enable writing of the HW detection to the write map output.

Application can use that output to save it to disk or analyse it.

DPF_READ_MAP_FILE: The path to the file to load as a DPF input list in HW format
(see *Defective Pixels Fixing read map (DPF)* (page 47)).

DPF_THRESHOLD: Format: int range [0,63]

Defaults: 0

Detection threshold

DPF_WEIGHT: Format: double range [0,255]

Defaults: 16

Detection MAD weight.

Differences with CSIM internal driver

- DPF_WRITE_MAP_SIZE is ignored in drivers! The size of the write map is computed from the size of the imager in MC_DPFInit() and stored in MC_DPF::ui32OutputMapSize. ISPC::ModuleDPF::setup() could be modified to load the size from a parameter or override the computed value.
- PF_WRITE_MAP_POS_THR and DPF_WRITE_MAP_NEG_THR are ignored in drivers! The registers are forced to 0 to ensure the whole DPF write map is written to file as recommended by HW TRM.

Display Pipeline Scaler (DSC)

Scaler in YUV domain before the image is converted to RGB for the display pipeline.

Note: Similar to *Encoder Pipeline Scaler (ESC)* (page 209) but data processed is going to the display pipeline output while ESC goes to encoder pipeline output.

The resolution of the DSC is also smaller (less filtering taps).

DSC High Level Parameters

DSC_ADJUST_CUTOFF_FREQ: Format: bool range {0,1}

Defaults: 0

Adjust the cut-off frequency when the taps are computed in MC.

DSC_PITCH: Format: double[2] range [0,16]

Defaults: 1 1

Scaling factor for horizontal and vertical direction (input/output).

Can be set to 0 and automatically computed using the imager's size and the DSC_RECT.

DSC_RECT: Format: int[4] range [0,8192]

Defaults: 0 0 0 0

Cropping rectangle coordinates (see DSC_RECT_TYPE).

DSC_RECT_TYPE: Format: {cliprect, croprec, outsize}

Defaults: croprec

Clip rectangle: X/Y of Top Left point, X/Y of Bottom Right point.

Note: The points are included so a value of:

0 0 1280 720

Will be an image of **1281** by **721** pixels.

Crop rectangle: margins from Left, Top, Right, Bottom.

Out size: X/Y of Top Left point and width/height of input (before scaling)

Encoder Pipeline Scaler (ESC)

Scaler in YUV domain for the encoder pipeline output.

Note: Similar to *Display Pipeline Scaler (DSC)* (page 208) but data processed is going to the encoder pipeline output while DSC goes to display pipeline output.

The ESC has a higher resolution scaling than the DSC (more filtering taps).

ESC High Level Parameters

ESC_ADJUST_CUTOFF_FREQ: Format: bool range {0,1}

Defaults: 0

Adjust the cut-off frequency when the taps are computed in MC.

ESC_PITCH: Format: double[2] range [0,16]

Defaults: 1 1

Scaling factor for horizontal and vertical direction (input/output).

Can be set to 0 and automatically computed using the imager's size and the ESC_RECT.

ESC_RECT: Format: int[4] range [0,8192]

Defaults: 0 0 0 0

Cropping rectangle coordinates (see ESC_RECT_TYPE).

ESC_RECT_TYPE: Format: {cliprect, croprect, outsize}

Defaults: croprect

Clip rectangle: X/Y of Top Left point, X/Y of Bottom Right point.

Note: The points are included so a value of:

0 0 1280 720

Will be an image of **1281** by **721** pixels.

Crop rectangle: margins from Left, Top, Right, Bottom.

Out size: X/Y of Top Left point and width/height of input (before scaling)

ESC_CHROMA_MODE: Format: {inter, co-sited}

Defaults: inter

Horizontal chroma placement relative to luma (interstitial or co-sited).

Gamma Look-Up table (GMA)

The driver allows to change the Gamma Look Up table at insmod time as explained in the *V2500 Insertion options* (page 17) section. It is also possible to add additional GMA LUTs following the *Gamma Look-Up table customisation* (page 148) section. This module only controls if the gamma curve is applied or not.

GMA High Level Parameters

GMA_BYPASS: Format: bool range {0,1}

Defaults: 0

Disable (1) or enabled (0) the loaded Gamma look up table.

Output formats (OUT)

Not technically a single HW module but regroups all output formats for the pipeline.

Note: Some output formats are version dependent and may not be available in all HW configurations. Check the relevant CI_INFO_eSUPPORTED in the CI driver connection.

OUT High Level Parameters

OUT_DE: Format: {NONE, BAYER8, BAYER10, BAYER12}

Defaults: NONE

Data Extraction output format (if OUT_DE_POINT is enabled).

Cannot be enabled if OUT_DISP is enabled.

OUT_DE_POINT: Format: int range [1,3]

Defaults: 1

Data Extraction output position (shared between all contexts):

1 = after IIF, before BLC blocks

2 = after LSH, before DNS blocks

3 = disabled

OUT_DISP: Format: {NONE, RGB888_24, RGB888_32, RGB101010_32, BGR888_24, BGR888_32, BGR101010_32}

Defaults: NONE

Display output format (will disable DE format if enabled as HW can only support 1).

OUT_ENC: Format: {NONE, NV21, NV12, NV61, NV16, NV21-10bit, NV12-10bit, NV61-10bit, NV16-10bit}

Defaults: NONE

Encoder output format.

OUT_DE_HDF: Format: {NONE, BGR101010_32}

Defaults: NONE

HDR Extraction output format. Only available from V2500 HW v2 and may not be available on all HW configurations.

OUT_DI_HDF: Format: {NONE, BGR161616_64}

Defaults: NONE

HDR Insertion format. Only available from V2500 HW v2 and may not be available on all HW configurations.

OUT_DE_RAW2D: Format: {NONE, TIFF10, TIFF12}

Defaults: NONE

Raw 2D Extraction output format. Only available from V2500 HW v2 and may not be available on all HW configurations.

Imager Interface (IIF)

Point of entry of the image in the ISP pipeline. Can decimate and crop the image.

IIF High Level Parameters

IIF_BAYER_FORMAT: Format: {RGGB, GRBG, GBRG, BGGR}

Defaults: RGGB

Sensor bayer mosaic used.

Loaded from the sensor but saved to know which one was used.

Note: When running with data-generator IIF_BAYER_FORMAT information is given by the FLX file meta-data.

IIF_CAP_RECT_TL: Format: int[2] range [0,8192]

Defaults: 0 0

Top left coordinate of capture rectangle in bayer image pixels.

IIF_CAP_RECT_BR: Format: int[2] range [0,8192]

Defaults: 0 0

Bottom right coordinate of capture rectangle (included).

This means values of:

```
IIF_CAP_RECT_TL 0 0
IIF_CAP_RECT_BR 1920 720
```

Will capture an image of **1921** by **721** pixels!

IIF_DECIMATION: Format: int[2] range [1,16]

Defaults: 1 1

Horizontal and vertical decimation factor (1; 1 means no decimation, 2; 1 means half horizontal, same vertical, etc).

Can be used to reduce output size drastically but quality will suffer greatly from it (no interpolation).

IIF Size information When cropping or decimation is enabled the setup module computes the size of the IIF will process as (knowing that CFA_SIZE is 2 pixels):

$$\text{sizeCFA}[x] = \frac{\text{cropBR}[x] - \text{cropTL}[x] + \text{CFA_SIZE}}{\text{CFA_SIZE} * \text{decimation}[x]}$$

Differences with CSIM internal driver IIF_BAYER_FORMAT is ignored by drivers. This value is loaded from the camera information.

Lateral Chromatic Aberration (LCA)

Chromatic aberration is caused by the lens having a different focal length for different wavelengths of light (i.e. colours). This fringing is corrected by applying a differential scaling factor to the red and blue channels relative to the green one.

This module is part of the tuning procedure: *Lateral Chromatic Aberration Tuning* (page 132).

LCA High Level Parameters

LCA_BLUEPOLY_X: Format: double[3] range [-16,15.9]

Defaults: 0 0 0

Coefficients of horizontal correction polynomial for blue channel (x,x^2,x^3).

LCA_BLUEPOLY_Y: Format: double[3] range [-16,15.9]

Defaults: 0 0 0

Coefficients of vertical correction polynomial for blue channel (x,x^2,x^3).

LCA_BLUE_CENTER: Format: int[2] range [-4095,4095]

Defaults: 0 0

Position of blue channel LCA correction centre, relative to left edge of frame cropped & decimated in IIF.

LCA_REDPOLY_X: Format: double[3] range [-16,15.9]

Defaults: 0 0 0

Coefficients of horizontal correction polynomial for red channel (x,x^2,x^3).

LCA_REDPOLY_Y: Format: double[3] range [-16,15.9]

Defaults: 0 0 0

Coefficients of vertical correction polynomial for red channel (x,x^2,x^3).

LCA_RED_CENTER: Format: int[2] range [-4095,4095]

Defaults: 0 0

Position of red channel LCA correction centre, relative to left edge of frame cropped & decimated in IIF.

LCA_DEC: Format: int[2] range [0,15]

Defaults: 0 0

Decimation compensation multiplier used with LCA_SHIFT. Should convert image size from post-IIF (decimated) to pre-IIF (sensor).

LCA_SHIFT: Format: int[2] range [0,3]

Defaults: 0 0

Right-shift (division) to normalise pre-IIF image width/2 in CFAs to 512.

Lens Shading (LSH)

Image vignetting caused by lens shading decreases brightness of the image around its edges. Lens Shading HW module present in ISP fixes this effect by applying additional gain to colour channels. The gain value depends on location of image tile - closer the image edge, larger the gain value (see [Lens Shading \(LSH\)](#) (page 213)). Because lens shading depends on illuminant temperature a set of de-shading grids can be loaded by ISP driver. Each grid should be generated during ISP calibration process ([Sensor Lens Shading Tuning](#) (page 131)) and should corresponds

to given illuminant temperature. The *Lens Shading Grid Control* (page 271) module is responsible for choosing which grid should be applied based on the temperature reported by the AWB module.

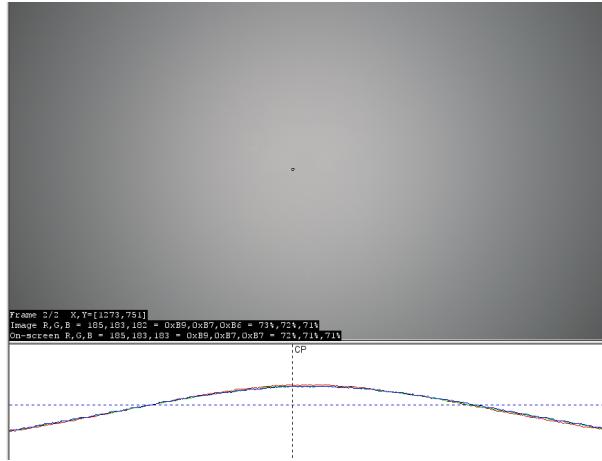


Fig. 8.5: Image before LSH correction

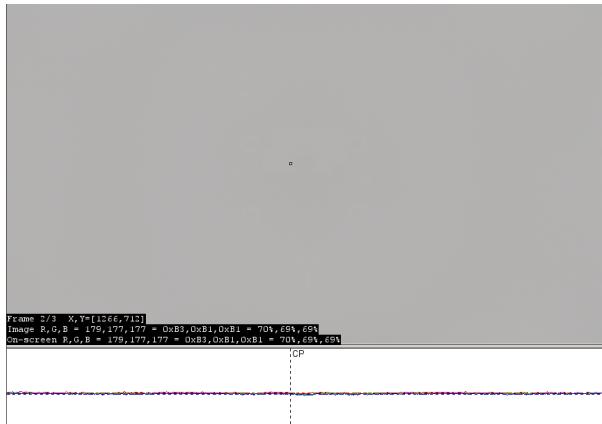


Fig. 8.6: Image after LSH correction

Lens shading gradients and deshading matrix loading from the HW LSH module.

The channel gradients are applied evenly across the entire image to correct vertical and horizontal gradual shading.

The de-shading grid can be used to correct vignetting effect caused by the lens. Typically there is less light captured in the corners of the image and the lens shading grid is a 2D matrix of gains to apply to compensate for that. The sampling step is referred as a *tile* (in CFA) and needs to be a power of 2. Smaller tile size yields better approximation of the ideal surface at the expense of using more memory.

This module is part of the tuning procedure: *Sensor Lens Shading Tuning* (page 131).

The ModuleLSH can load several deshading grid and the user can either choose which one to use at run time or instantiate the ControlLSH control Module to choose from the colour temperature provided by the Auto White Balance algorithm.

Note: In `ModuleLSH::load()` the matrix grids are not loaded. The function Mod-

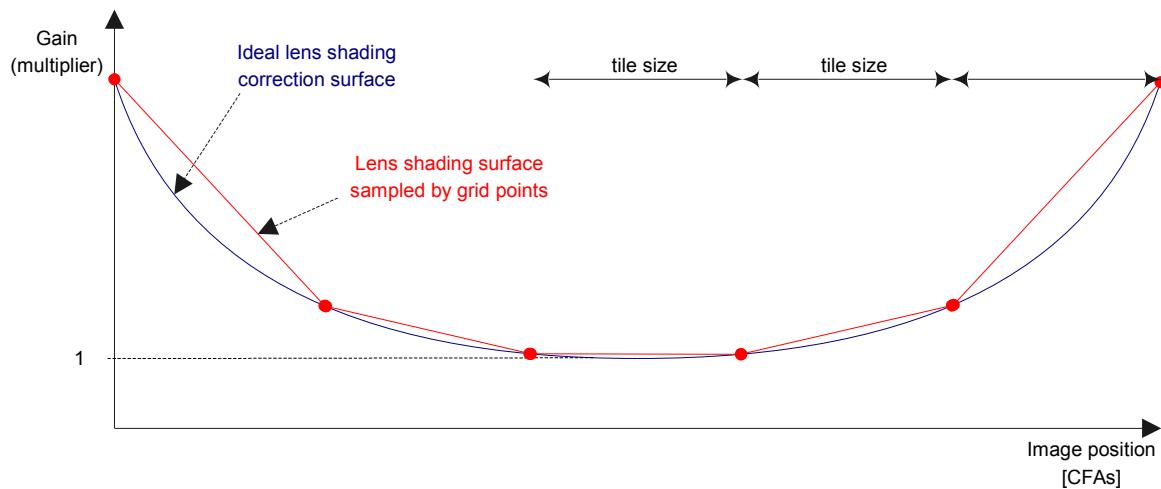


Fig. 8.7: LSH grid sampling and tile size (source: TRM)

`uleLSH::loadMatrices()` can be used to load all the available matrices from the parameter file using the [Lens Shading Grid Control](#) (page 271) method.

If the ControlLSH module is instantiated then loading the matrices is the responsibility of the ControlLSH object.

LSH High Level Parameters

LSH_GRADIENTX: Format: double[4] range [-4,4]

Defaults: 0 0 0 0

Horizontal linear shading gradient per channel (R, G, B, B).

LSH_GRADIENTY: Format: double[4] range [-4,4]

Defaults: 0 0 0 0

Vertical linear shading gradient per channel (R, G, B, B).

LSH_MATRIX_ENABLE: Format: bool range {0,1}

Defaults: 0

Enable/disable use of lens shading matrix.

Differences with CSIM internal driver

- `LSH_MATRIX_FILE` is ignored by ISPC! The matrices are loaded using the [Lens Shading Grid Control](#) (page 271) module.
- `LSH_MATRIX_TILE_SIZE` is ignored by ISPC! The tile size is loaded from the file.
- `LSH_MATRIX_DELTA_BITS` is ignored by ISPC! The number of bits used in the matrix are computed when translating the matrix from Floats to HW format in `MC_LSHConvert()`.

Main Gamut Mapper (MGM)

Similar to DGM but in different point in the pipeline. Algorithm is detailed in section *Display Gamut Mapper (DGM)* (page 204).

MGM High Level Parameters

MGM_CLIP_MAX: Format: double range [0,2]

Defaults: 1.5

The upper clip limit common to all three input channels.

MGM_SRC_NORM: Format: double range [0,2]

Defaults: 1

Defines the knee point on the input side of the transfer curve. Corresponds to the upper limit of normal gamut at the source.

MGM_CLIP_MIN: Format: double range [-1,0]

Defaults: -0.5

This is the lower clip limit common to all three input channels.

MGM_COEFF: Format: double[6] range [-2,2]

Defaults: 0 0 0 0 0 0

Scaling factors for converting negative extended range to increased positive range.

0: green to red, 1: blue to red, 2: blue to green, 3: red to green, 4: red to blue, 5: green to blue

MGM_SLOPE: Format: double[3] range [0,2]

Defaults: 1 1 1

0: The conversion gain for the negative sector of the input range. Applied after the input MGM_CLIP_MIN.

1: The conversion gain for the input range from 0 to MGM_SRC_NORM.

2: The conversion gain for the input range from MGM_SRC_NORM to MGM_CLIP_MAX.

Memory Image Enhancer (MIE)

This configures the Image Enhancer Memory Colour part of the HW module. The vibrancy part is configured with the *Vibrancy (VIB)* (page 230) parameters.

Information about the tuning process for this block is available in *Image Enhancer Tuning* (page 139).

Algorithm overview In human perception, people strongly associate certain colours with common objects and features. The term “Memory Colour” (MC) is used for these colours.

Example of memory colours include:

1. The pink/brown tone associated with skin tones.
2. The blue tones associated with sky and sea.
3. The green/brown tones associated with foliage and outdoor nature shots.

These colours often require specialized colour correction, because observers are particularly sensitive to a correct reproduction of those colours.

The Memory Image Enhancement (MIE) block supports 3 Memory Colours (MC). For each MC:

- Calculates the likelihood L_X of the current pixel i.e. its similarity to the current MC.
- Modify the pixel YCC value according to MC-specific transformation T_X .

Image Enhancement mixes the 3 MC transformations and the original YCbCr value (non-memory colour path) into a single one using the likelihood as mixing weights. Figure [Memory Colour Mixture \(source: TRM\)](#) (page 217) shows the mixing scheme, including the bypass branch.

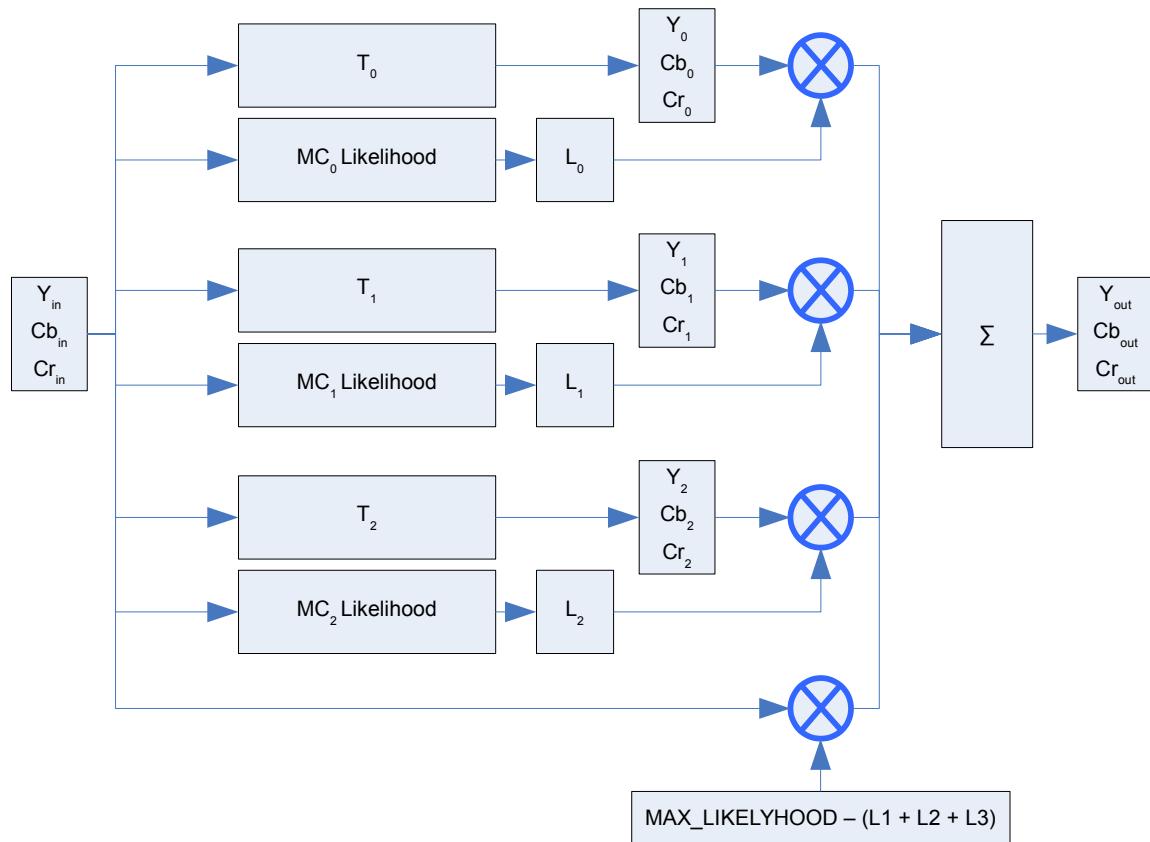


Fig. 8.8: Memory Colour Mixture (source: TRM)

Likelihood estimate The statistical distribution of a memory colour in YCbCr is modelled as piecewise linear gain function $G(y)$ on the Y-axis and a diamond-shaped Gaussian distribution

$W_y(Cb, Cr)$ centred around a CbCr centroid $Cb0, Cr0$ and fading to zero, representing the prototype memory colour (e.g. a blue tint for sky): $L(y, Cb, Cr) = G(y) \cdot W_y(Cb, Cr)$

Figure *Memory Colour Likelihood Model* (source: TRM) (page 219) graphically illustrates how the memory colour is modelled in YCbCr space. Four equally spaced, diamond-shaped CbCr layers (slices) are stacked on the Y-axis. The Likelihood is interpolated (dotted line in inset A and B) for pixels yielding Y value between two slices.

The diamond shape in Figure *Memory Colour Likelihood Model* (source: TRM) (page 219) inset C represents the iso value curve in CbCr where the distribution is 50% of its peak value in $Cb0, Cr0$. In fact the distribution in CbCr is theoretically unbound and fades to zero as the CbCr value moves away from $Cb0, Cr0$, following a Gaussian decay (see inset B). The fading control parameters are and corresponding to the axes of the diamond shape. Note how can be rotated in CbCr by an angle.

The Extent (dashed line in Figure *Memory Colour Likelihood Model* (source: TRM) (page 219) inset C,) is the equivalent circular area with radius equal to the geometric average of the diamond shape axis. The Variance parameters σ_b, σ_r change with the slice number.

MIE High Level parameters

MIE Generic Parameters These parameters affect the loading of all the memory colours.

MIE_BLACK_LEVEL: Format: double range [0,1]

Defaults: 0.0625

Black level at input of MIE, specified in normalized 0.0 to 1.0.

E.g. black level = $16/256 = 0.0625$

Warning: Also used by VIB HW module.

MIE_MEMORY_COLOURS: Format: unsigned range [0,9]

Defaults: 3

Number of memory colours configurations to load from the parameters.

For each of the memory colours parameters the _X is the memory colour it is applied to (e.g. *MIE_ENABLED_0* 1 enabled memory colours 0, *MIE_OUT_BRIGHTNESS_2* 0.5 sets the brightness to 0.5 for memory colours 2).

Note: The HW supports up to 3 enabled memory colours but the parameters and the drivers can have more memory colours defined.

If more than 3 memory colours are enabled the driver will only use the first 3 and discard the others.

Differences with CSIM internal driver The CSIM internal driver uses the deprecated parameters. Please ensure the setup parameters used for the module were updated properly.

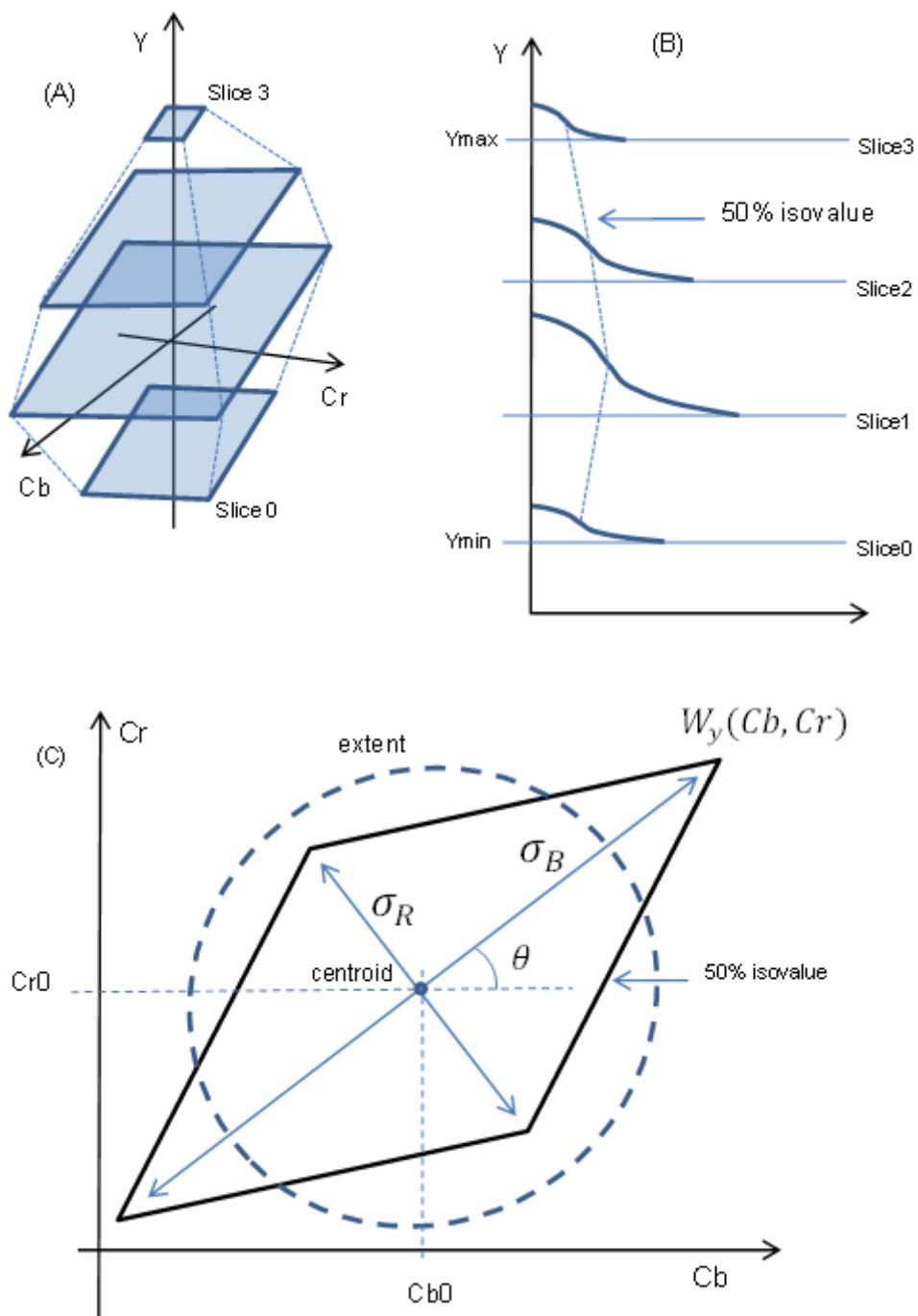


Fig. 8.9: Memory Colour Likelihood Model (source: TRM)

Luma input and modification

MIE_ENABLED_X: Format: bool range {0,1}

Defaults: 0

Memory Colour enabled.

MIE_YMIN_X: Format: double range [0,1]

Defaults: 0

Minimum Y value selection (1st slice of the 4 evenly distributed between YMin and YMax).

MIE_YMAX_X: Format: double range [0,2]

Defaults: 1

Maximum Y value selection (4th slice of the 4 evenly distributed between YMin and YMax).

MIE_YGAINS_X: Format: double[4] range [0,1]

Defaults: 0 0 0 0

4 values, 1 per slice.

Gain per slice of the likelihood function.

Chroma input

MIE_CCENTER_X: Format: double[2] range [-0.5,0.5]

Defaults: 0 0

Chroma likelihood center point: Cb and Cr values.

MIE_CEXTENT_X: Format: double[4] range [0.0039,4]

Defaults: 0.5 0.5 0.5 0.5

4 values, 1 per slice.

Extension of the Chroma likelihood (the bigger the likelier).

MIE_CASPECT_X: Format: double range [-2,2]

Defaults: 0

Ellipsoid aspect ratio (log2) in CbCr plane.

MIE_CROTATION_X: Format: double range [-1,1]

Defaults: 0

Ellipsoid rotation in CbCr plane.

Chroma output effects

MIE_OUT_HUE_X: Format: double range [-1,1]

Defaults: 0

Output hue rotation (degrees).

MIE_OUT_SATURATION_X: Format: double range [0,4]

Defaults: 1

Output saturation.

MIE_OUT_BRIGHTNESS_X: Format: double range [-1,1]

Defaults: 0

Output brightness.

MIE_OUT_CONTRAST_X: Format: double range [0,4]

Defaults: 1

Output contrast.

Deprecated parameters These parameters are now deprecated and will not be loaded by the setup module. They are provided as references to allow the update of previous parameters.

Note: The provided example values are not tuning parameters and only provided to easily differentiate between memory colours and slices.

MIE_VERIFY_PARAMS: This was not used by the drivers.

This parameter modifies gains if they get too strong. The Setup tools do it for the driver.

MIE_MC_ON: Is now enabled per memory colour.

MIE_MC_YMIN: Used to be a vector of 1 value per memory colour. Now is specified per memory colour using MIE_YMIN_X.

Deprecated:

```
MIE_MC_YMIN 0.2422 0.1 0.8
```

Updated:

```
MIE_YMIN_0 0.2422
MIE_YMIN_1 0.1
MIE_YMIN_2 0.8
```

MIE_MC_YMAX: Used to be a vector of 1 value per memory colour. Is now specified per memory colour using MIE_YMAX_X.

Deperacted:

```
MIE_MC_YMAX 0.4 0.3 1.0
```

Updated:

```
MIE_YMAX_0 0.4
MIE_YMAX_1 0.3
MIE_YMAX_2 1.0
```

MIE_MC_YGAIN_1: Used to be a vector of 1 value per memory colour for each parameter. Is now specified as a vector of 4 values per memory colour using MIE_YGAINS_X

Deprecated:

```
MIE_MC_YGAIN_1 0.4 0.2 0.4
MIE_MC_YGAIN_2 0.5 0.4 0.1
MIE_MC_YGAIN_3 0.68 0.2 0.1
MIE_MC_YGAIN_4 0.75 0.1 0
```

Updated:

```
MIE_YGAINS_0 0.4 0.5 0.68 0.75
MIE_YGAINS_1 0.2 0.4 0.2 0.1
MIE_YGAINS_2 0.4 0.1 0.1 0
```

MIE_MC_YGAIN_2: See MIE_MC_YGAIN_1.

MIE_MC_YGAIN_3: See MIE_MC_YGAIN_1.

MIE_MC_YGAIN_4: See MIE_MC_YGAIN_1.

MIE_MC_CB0: Used to be 1 value per memory colour representing the centre of the chroma slices. Now specified as a centre per memory colour using MIE_CCENTER_X.

Deprecated:

```
MIE_MC_CB0 0.038 -0.038 0.222
MIE_MC_CRO 0.044 -0.044 0.33
```

Updated:

```
MIE_CCENTER_0 0.038 0.044
MIE_CCENTER_1 -0.0388 -0.044
MIE_CCENTER_2 0.222 0.33
```

MIE_MC_CRO: See MIE_MC_CB0.

MIE_MC_CEXTENT_1: Used to be 1 value per memory colour for each parameter. Is now specified as a vector of 4 values per memory colour using MIE_CEXTENT_X.

Deprecated:

```
MIE_MC_CEXTENT_1 0.08 0.05 0.15
MIE_MC_CEXTENT_2 0.09 0.25 0.01
MIE_MC_CEXTENT_3 0.1 0.5 0.05
MIE_MC_CEXTENT_4 0.11 0.2 0.3
```

Updated:

```
MIE_CEXTENT_0 0.08 0.09 0.1 0.11
MIE_CEXTENT_1 0.05 0.25 0.5 0.2
MIE_CEXTENT_2 0.15 0.01 0.05 0.3
```

MIE_MC_CEXTENT_2: See MIE_MC_CEXTENT_1.

MIE_MC_CEXTENT_3: See MIE_MC_CEXTENT_1.

MIE_MC_CEXTENT_4: See MIE_MC_CEXTENT_1.

MIE_MC_CASPECT: Used to be 1 value per memory colour. Is now specified per memory colour using MIE_CASPECT_X.

Deprecated:

```
MIE_MC_CASPECT -0.444 0.444 0.666
```

Updated:

```
MIE_CASPECT_0 -0.444
MIE_CASPECT_1 0.444
MIE_CASPECT_2 0.666
```

MIE_MC_CROTATION: Used to be 1 value per memory colour. Is now specified per memory colour using MIE_CROTATION_X.

Deprecated:

```
MIE_MC_CROTATION -0.259 0.259 0.5
```

Updated:

```
MIE_CROTATION_0 -0.259
MIE_CROTATION_1 0.259
MIE_CROTATION_2 0.5
```

MIE_MC_HUE: Used to be 1 value per memory colour. Is now specified per memory colour using MIE_OUT_HUE_X.

Deprecated:

```
MIE_MC_HUE 0.889 0 0.5
```

Updated:

```
MIE_OUT_HUE_0 0.889
MIE_OUT_HUE_1 0
MIE_OUT_HUE_2 0.5
```

MIE_MC_SATURATION: Used to be 1 value per memory colour. Is now specified per memory colour using MIE_OUT_SATURATION_X.

Deprecated:

```
MIE_MC_SATURATION 1.84 0.2 0.5
```

Updated:

```
MIE_OUT_SATURATION_0 1.84
MIE_OUT_SATURATION_1 0.2
MIE_OUT_SATURATION_2 0.5
```

MIE_MC_BRIGHTNESS: Used to be 1 value per memory colour. Is now specified per memory colour using MIE_OUT_BRIGHTNESS_X.

Deprecated:

```
MIE_MC_BRIGHTNESS 0 0.5 -0.4
```

Updated:

```
MIE_OUT_BRIGHTNESS_0 0
MIE_OUT_BRIGHTNESS_1 0.5
MIE_OUT_BRIGHTNESS_2 -0.4
```

MIE_MC_CONTRAST: Used to be 1 value per memory colour. Is now specified per memory colour using MIE_OUT_CONTRAST_X.

Deprecated:

```
MIE_MC_CONTRAST 1 0.2 0
```

Updated:

```
MIE_OUT CONTRAST_0 1
MIE_OUT CONTRAST_1 0.2
MIE_OUT CONTRAST_2 0
```

RGB to YUV (R2Y)

Similar to *YUV to RGB (Y2R)* (page 232) but converts RGB to YUV while Y2R converts YUV to RGB at a different point in the pipeline.

R2Y High Level Parameters

R2Y_MATRIX: Format: {BT601, BT709, JFIF}

Defaults: BT709

Standard matrix to use as base.

R2Y_BRIGHTNESS: Format: double range [-0.5,0.5]

Defaults: 0

Output brightness (offset).

R2Y_CONTRAST: Format: double range [0,2]

Defaults: 1

Output contrast (luma gain).

R2Y_HUE: Format: double range [-30,30]

Defaults: 0

Hue rotation in degrees.

R2Y_SATURATION: Format: double range [0.1,10]

Defaults: 1

Output saturation (chroma gain).

R2Y_OFFSETU: Format: double range [-0.5,0.5]

Defaults: 0

Normalised offset on the U plane (Cb).

R2Y_OFFSETV: Format: double range [-0.5,0.5]

Defaults: 0

Normalised offset on the V plane (Cr).

R2Y_RANGE_MUL: Format: double[3] range [0,2]

Defaults: 1 1 1

Gain multipliers for Y, Cb, Cr (before output offset).

Raw Look-up Table (RLT)

RLT High Level Parameters

RLT_CORRECTION_MODE: Format: {DISABLED, LINEAR, CUBIC}

Defaults: DISABLED

RLT correction mode to use. Changes the meaning of the RLT_POINTS_X parameters.

RLT_POINTS_X: Format: unsigned[16] range [0,65536]

Defaults: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

RLT_POINTS_0, RLT_POINTS_1, RLT_POINTS_2 and RLT_POINTS_3 are the RLT correction points.

If mode is linear the curve is 64 points loaded in order (0 is 1st 16, 1 is 2nd 16 etc).

If mode is cubic each entry used for 1 colour channel (0=R, 1=G1, 2=G2, 3=B).

Sharpening (SHA)

This module is part of the tuning procedure: *Sharpening and Secondary denoiser Tuning* (page 134).

SHA High Level Parameters

SHA_DETAIL: Format: double range [0,1]

Defaults: 1

Sharpening texture vs. edges detail parameter.

0 is only edges while 1 is everything.

SHA_EDGE_OFFSET: Format: double range [-1,1]

Defaults: 0

It is recommended to keep this parameter at 0.

SHA_EDGE_SCALE: Format: double range [0,1]

Defaults: 0.25

Sharpening edge likelihood scale parameter.

High values produce abrupt changes in edge/non-edge likelihood map.

It is recommended to keep this parameter at 0.

SHA_RADIUS: Format: double range [0.5,10]

Defaults: 2.5

Sharpening radius in pixels that sharpening will be applied to.

Any value over 5 would be meaningless, if blur is spread over 5 pixels then it will only have a small impact. Typical values for modern small pixel sensors would be in the range of [1.5, 2.5]

SHA_STRENGTH: Format: double range [0,1]

Defaults: 0.4

Sharpening strength.

SHA_THRESH: Format: double range [0,1]

Defaults: 0

Sharpening mask threshold.

It is recommended to keep this parameter at 0.

Tone Mapper (TNM)

The tone mapper module allows the modification of the global and local tone as well as modifying the luma range. Information about how to tune this module is available in [Tone-mapping Tuning](#) (page 135).

This module is part of the tuning procedure: [Tone-mapping Tuning](#) (page 135).

Algorithm overview The tone mapping module operates with YCC data and the complete tone mapping process can be separated into 5 distinct phases (see [Tone Mapper block diagram \(source: TRM\)](#) (page 226)). The first and last phases are simply to scale the input and output ranges of the luma and chroma signals according to the desired gamut. After input range correction, the luma component undergoes a global tone map transfer using a predetermined tone curve that is programmed through the host interface. Then, the globally tone mapped luma values undergo a localised tone map transfer, using statistics that are collected from previously processed pixels of the current frame. After tone mapping of the luma signal, the chroma values are adjusted according to the change in the corresponding luma component.

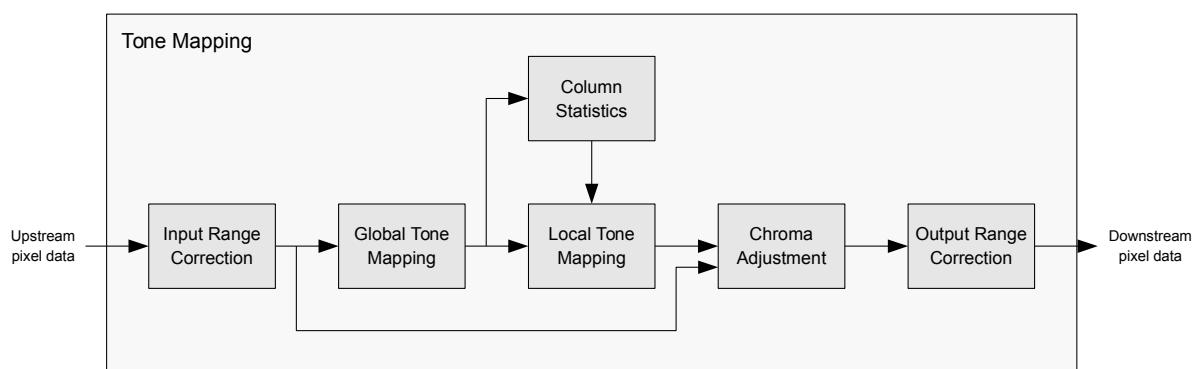


Fig. 8.10: Tone Mapper block diagram (source: TRM)

Luma range The values specified in the TNM_IN_Y setup parameter define the valid range for the application of tone mapping. This range will be used for both global and local tone mapping and any values below or above the defined range will be clipped.

The global tone mapping curve is scaled to cover the defined range so the same global mapping curve applied with different TNM_IN_Y values will produce different effects in the output of the tone mapper. That implies as well that if the global tone mapping curve is derived from the frame to be processed (or previous frames) the gathering of statistics and curve generation must take into consideration the tone mapper TNM_IN_Y for a proper alignment of the image data and the global curve.

Warning: It is expected that the range for the tone map collection is -64 to 64 and it should not need to be changed by the user.

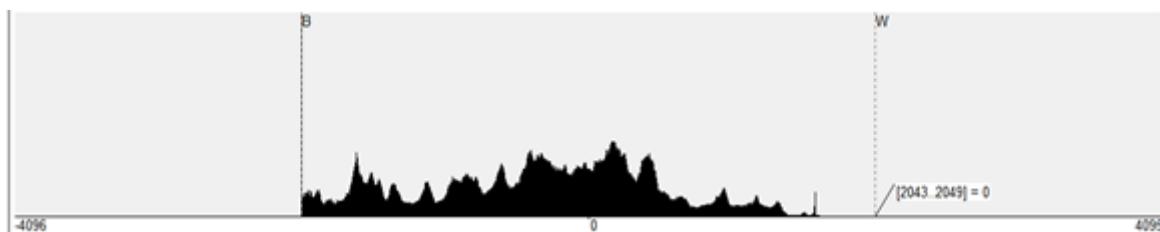


Fig. 8.11: Tone Mapper Luma range

Colour saturation Apart from the global and local tone mapping controls there are two setup parameters devoted to control the colour appearance after tone mapping: *colour confidence* and *colour saturation*.

The saturation controls are used because, as result of the luminance changes carried out by the tone mapper, the saturation of the output colours may vary. In order to deal with that the tone mapper internally scales the chroma values of the pixels to keep the same levels of saturation in the input and the output.

Nevertheless, depending on the global and local configuration, there are cases in which the output perceived colour saturation might vary with respect to the input. The *colour saturation* is a scaling factor for the saturation aimed to correct any perceived bias in the output saturation. Typically we would like to set it up so the input and output saturation look the same but it is basically a matter of preference.

The Figure *Effect of different values of colour saturation* (page 227) show an example of two different *colour saturation* values and how they affect the output.



Fig. 8.12: Effect of different values of *colour saturation*

The *colour confidence* parameter serves as a safeguard for undesired colour effects that might occur when the original luma values are very close to the black or white level. For example, very dark regions in the original image might contain some colour cast not perceivable in the original image. If such pixels luminance values are heavily increased by the tone mapper such colour cast will be amplified resulting in unnatural colours in certain regions of the image.

Figure *Effect of different values of colour confidence* (page 228), generated with a bit excessive local enhancement and *colour confidence* of 4, shows an example of those colour casts in the right bottom corner (see augmented region).

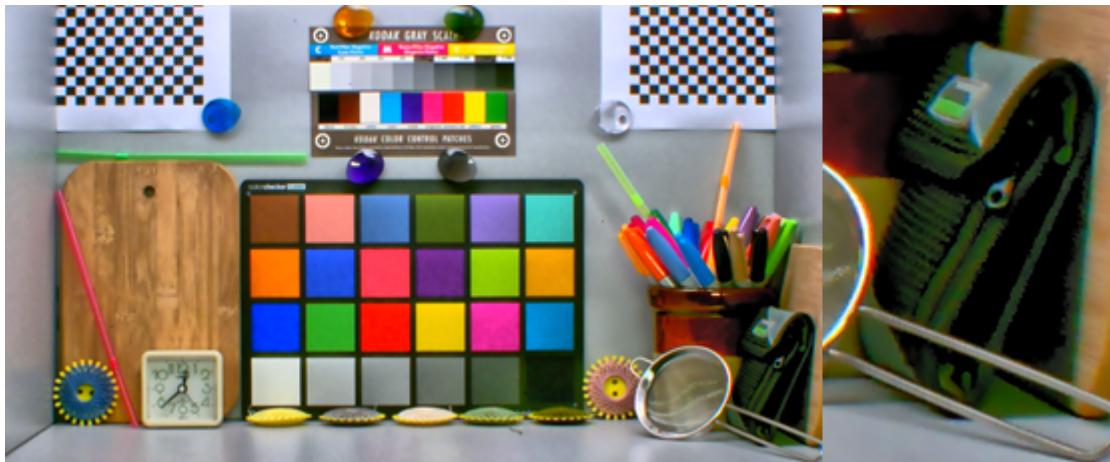


Fig. 8.13: Effect of different values of *colour confidence*

In order to deal with such problem the *colour confidence* reduces the saturation correction in the output image, specially for values close to the black/white levels. The tone mapper calculates a confidence measure for chroma correction according to the input luma value as shown in Figure *Colour confidence scaling calculations* (page 228) (A). The *colour confidence* value is used to scale up or down the confidence value. As the confidence measure will be always between 0 and 1 obtained values above 1 will be clipped as shown in the figure below (B).

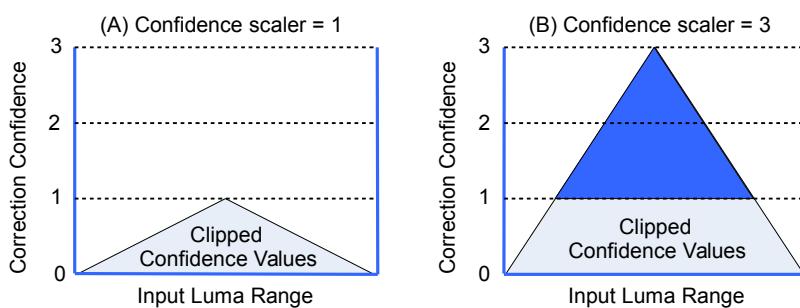


Fig. 8.14: Colour confidence scaling calculations

This mechanism allows smoothing the chrominance correction near the black or white points, where the saturation ratio between chrominance and luminance is less reliable.

Figure *Colour confidence = 4 (LHS) vs Colour confidence = 1 (RHS)* (page 229) shows the effect of a colour confidence value reduction applied to remove unwanted colour casts when low luminance regions of the image are highlighted by the tone mapper. It must be noted that depending on the colour confidence value applied the saturation in pixels with luminance values not that close to black or white could be affected as well.



Fig. 8.15: Colour confidence = 4 (LHS) vs Colour confidence = 1 (RHS)

TNM High Level Parameters

TNM Generic parameters

TNM_BYPASS: Format: bool range {0,1}

Defaults: 0

Enable (0) or disable (1) tone mapping. If disabled the Luma is still rescaled.

TNM_IN_Y: Format: double[2] range [-127,127]

Defaults: -64 64

Valid range of input luma values. Values outside get clipped.

It is recommended to always be -64, 64.

TNM Tone Mapping parameters

TNM_FLAT_FACTOR: Format: double range [0,1]

Defaults: 0

Strength of histogram flattening for local tone mapping (0=none, 1=full).

TNM_FLAT_MIN: Format: double range [0,1]

Defaults: 0

Minimum histogram flattening applied for local tone mapping (0=none, 1=full).

TNM_WEIGHT_LINE: Format: double range [0,1]

Defaults: 0

Rate at which local tone mapping curves adapt (cca per line, 0=none, 1=full).

TNM_WEIGHT_LOCAL: Format: double range [0,1]

Defaults: 0

Strength of local tone mapping (0=none, 1=full).

TNM_CURVE: Format: double[63] range [0,1]

Defaults: 0.0153846 0.0307692 0.0461538 0.0615385 0.0769231 0.0923077 0.107692
0.123077 0.138462 0.153846 0.169231 0.184615 0.2 0.215385 0.230769 0.246154 0.261538
0.276923 0.292308 0.307692 0.323077 0.338462 0.353846 0.369231 0.384615 0.4 0.415385
0.430769 0.446154 0.461538 0.476923 0.492308 0.507692 0.523077 0.538462 0.553846
0.569231 0.584615 0.6 0.615385 0.630769 0.646154 0.661538 0.676923 0.692308 0.707692
0.723077 0.738462 0.753846 0.769231 0.784615 0.8 0.815385 0.830769 0.846154 0.861538
0.876923 0.892308 0.907692 0.923077 0.938462 0.953846 0.969231 (linear identity)

Tone mapping curve normalised Y=[0, 1].

The HW uses a 65 point curve and assumes that 1st point is 0 and last point 1.

TNM Colour parameters

TNM_COLOUR_CONFIDENCE: Format: double range [0,64]

Defaults: 1

Colour confidence factor. Applies always unless module is bypassed.

TNM_COLOUR_SATURATION: Format: double range [0,16]

Defaults: 1

Colour saturation factor. Applies always unless module is bypassed.

Vibrancy (VIB)

This block applies an input/output transfer curve to the Chroma signal Cb, Cr, according to an estimate of the pixel saturation. The goal is to increase saturation of low and mid-saturated pixels without affecting those pixels whose saturation value is already high, therefore providing vibrancy (colourfulness) without amplifying colours that are already “colourful”. This provides superior performance compared to simple saturation.

Vibrancy curve is specified with a 32 point piecewise linear curve, which in principle can be arbitrary (within the register bit depth limitations). However a procedure is used in the GUI tools to generate a 32-point curve that can be controlled with two control points and provides an intuitive way to add vibrancy to the frame.

The Figure *Vibrancy curve example* (page 231) shows an exemplary vibrancy curve (green - using similar 2 points as in *VisionLive VIB tab* (page 113)) compared against an equivalent saturation curve (red, clipped to 1). The green curve allows more control.

VIB High Level Parameters This configures the Vibrancy part of the MIE HW module.

MIE_VIB_ON: Format: bool range {0,1}

Defaults: 0

Enable vibrancy module.

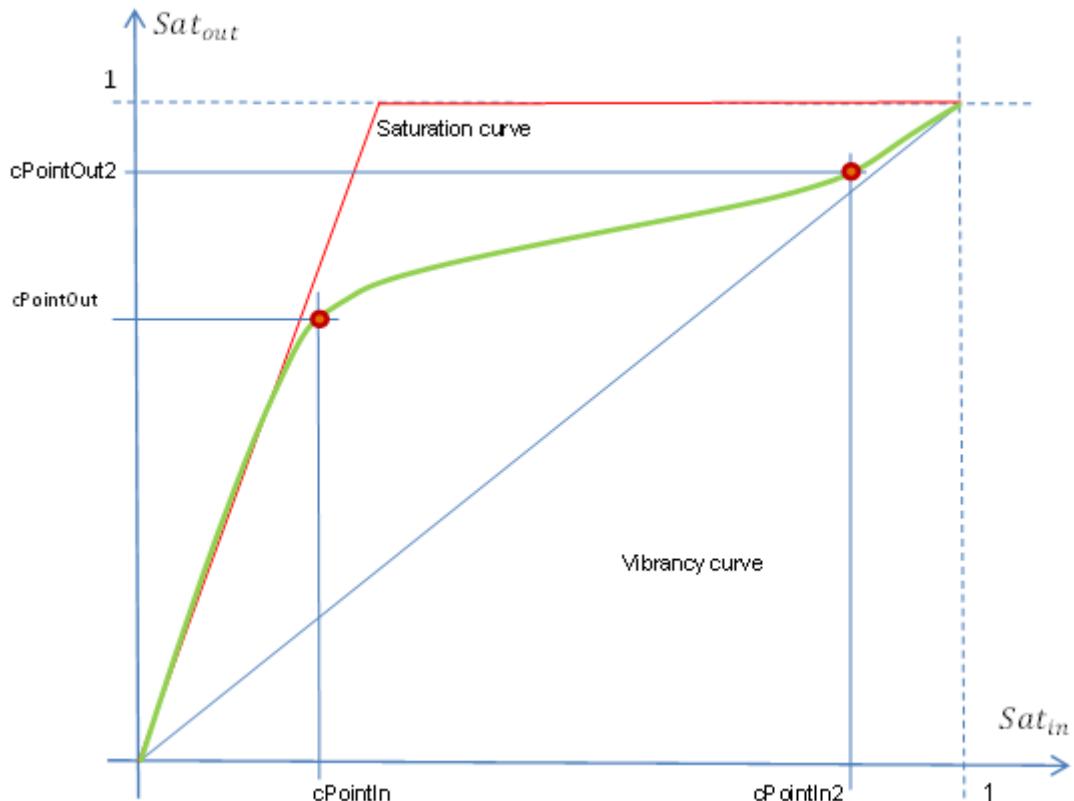


Fig. 8.16: Vibrancy curve example

MIE_VIB_SAT_CURVE: Format: double[32] range [0,1]

Defaults: 0 0.0322581 0.0645161 0.0967742 0.129032 0.16129 0.193548 0.225806 0.258065
0.290323 0.322581 0.354839 0.387097 0.419355 0.451613 0.483871 0.516129 0.548387
0.580645 0.612903 0.645161 0.677419 0.709677 0.741935 0.774194 0.806452 0.83871
0.870968 0.903226 0.935484 0.967742 1

Output Saturation curve normalised [0, 1]. Input points are evenly distributed in [0, 1] using 32 samples.

White Balance Correction (WBC)

This is the White Balance part of the LSH HW module that applies a gain value (a multiplier) to each Bayer colour channel.

This module is part of the tuning procedure: *White Balance Correction Tuning* (page 131).

WBC High Level Parameters

LSH_WBCLIP: Format: double[4] range [0.5,2]

Defaults: 1 1 1 1

Output pixel white clip level per channel. One per Bayer channel using R, G, G, B order regardless of sensor's mosaic.

LSH_WBGAIN: Format: double[4] range [0.5,8]

Defaults: 1 1 1 1

White balance gains. One per Bayer channel using R, G, G, B order regardless of sensor's mosaic.

YUV to RGB (Y2R)

Similar to [RGB to YUV \(R2Y\)](#) (page 224) but converts YUV to RGB while R2Y converts RGB to YUV at a different point in the pipeline.

Y2R High Level Parameters

Y2R_MATRIX: Format: {BT601, BT709, JFIF}

Defaults: BT709

Standard matrix to use as base.

Y2R_BRIGHTNESS: Format: double range [-0.5,0.5]

Defaults: 0

Output brightness (offset).

Y2R_CONTRAST: Format: double range [0.1,10]

Defaults: 1

Output contrast (luma gain).

Y2R_HUE: Format: double range [-30,30]

Defaults: 0

Hue rotation in degrees.

Y2R_SATURATION: Format: double range [0.1,10]

Defaults: 1

Output saturation (chroma gain).

Y2R_OFFSETU: Format: double range [-0.5,0.5]

Defaults: -0.5

Normalised offset on the U plane (Cb).

Y2R_OFFSETV: Format: double range [-0.5,0.5]

Defaults: -0.5

Normalised offset on the V plane (Cr).

Y2R_RANGE_MUL: Format: double[3] range [0,2]

Defaults: 1 1 1

Gain multipliers for Y, Cb, Cr (before output offset).

Implemented Statistics Modules

The following sections will detail the modules that gather statistics. It will also list the high level parameters for the SW modules. More information about the modules order is available in the *V2500 modules* (page 2) section.

Defective Pixel Fixing (DPF)

Detailed in *Defective Pixels Fixing (DPF)* (page 207).

Encoder Statistics (ENS)

The ENS module does a partial estimation of the entropy of regions of an image. The image is partitioned into regions that each accumulate their own entropy value. The calculation is done over a 2x2 kernel (line pairs) and measures the difference among the cluster of pixel. The result generated by the HW can be used as a histogram of differences across the image.

ENS High Level Parameters

ENS_ENABLE: Format: bool range {0,1}

Defaults: 0

Enable/disable encoder statistics.

If enabled then the ENS buffer is saved and its size computed from the image's size and the other parameters. The saving format is application dependent but usually follows the HW format (see TRM).

ENS_REGION_NUMLINES: Format: int range [8,256]

Defaults: 16

Number of lines in each region. **Must be a power of 2.**

Similar to *driver_test* (page 49) -CTX_ENSNLines.

ENS_SUBS_FACTOR: Format: int range [1,32]

Defaults: 1

Horizontal sub-sampling (ratio of 1:sub-sampling) for the region. **Must be a power of 2.**

Encoder statistics output encoder signal limitation There is no implementation enabling the *out_enc* HW signal to propagate statistics from ISP to Encoder (see core register *ENC_OUT_CTRL*) in the ISPC layer. The low level driver allows a selection using *CI_PIPELINE::bUseEncOutLine* and *CI_PIPELINE::ui8EncOutPulse*.

Selecting the context enabling the signal can be controlled with *-encOutCtx* in *driver_test* (page 49).

Exposure Statistics (EXS)

The EXS module in HW collects 2 sets of information on a column of pixels (or a single pixel if HW parallelism is 1):

- A number of pixels that are over the *pixel max* threshold for each channel over the whole image
- A number of pixels that are over the *pixel max* threshold for each channel on a region

The regional statistics contains a grid of tiles, all of the same dimension. The grid is 7 tiles wide and 7 tiles high. Each tile has to be at least 8 pixels wide.

It is valid for the ROI to be partially located off the frame at the bottom or right edges. However the statistics may not be accurate unless the whole tile is located within the image (see *Tile update when off frame (source: TRM)* (page 234)).

- If a tile is completely off the right edge of the picture then 0 will be stored for that tile.
- If a tile is partially calculated then that value will be stored.
- If a tile is completely off the bottom edge of the picture they will not be updated and may contain memory from previous frames.

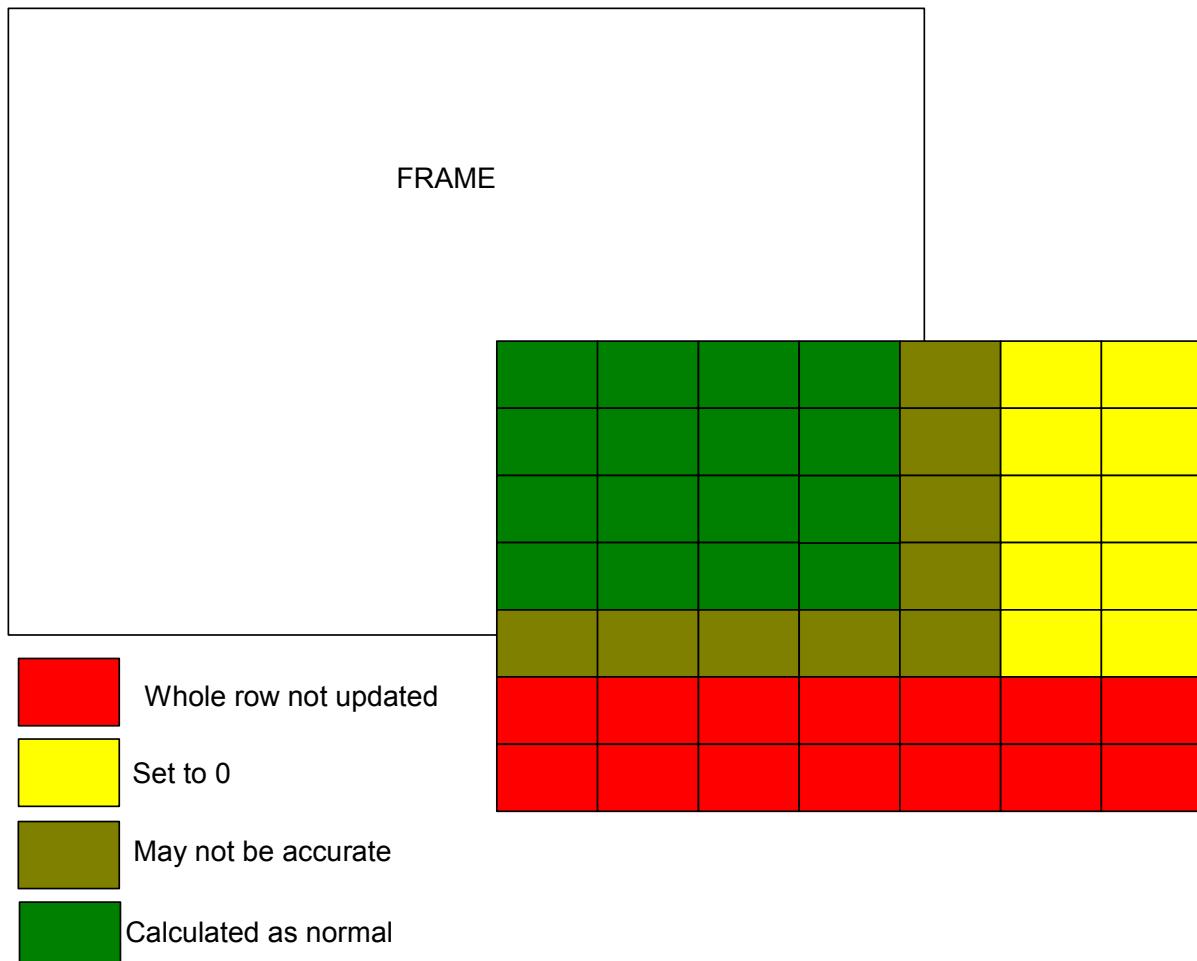


Fig. 8.17: Tile update when off frame (source: TRM)

Note: Currently not used by any algorithms.

EXS High Level Parameters

EXS_GLOBAL_ENABLE: Format: bool range {0,1}

Defaults: 0

EXS_GRID_START_COORDS: Format: int[2] range [0,8192]

Defaults: 0 0

EXS_GRID_TILE_DIMENSIONS: Format: int[2] range [8,8191]

Defaults: 9 9

EXS_PIXEL_MAX: Format: int range [0,4095]

Defaults: 4095

EXS_REGIONAL_ENABLE: Format: bool range {0,1}

Defaults: 0

Flicker Detection (FLD)

Used in *Automatic Exposure (AE)* (page 245) module as the optional source of flicker frequency.

FLD High Level Parameters

FLD_ENABLE: Format: bool range {0,1}

Defaults: 0

Enable flicker detection.

FLD_COEF_DIFF_TH: Format: int range [0,32767]

Defaults: 50

Difference between 50 and 60hz coefficients.

FLD_MIN_PN: Format: int range [0,63]

Defaults: 4

If scene change is detected, vote for final result unless this number of frames has been processed.

FLD_NF_TH: Format: int range [0,32767]

Defaults: 15000

“No flicker” threshold.

FLD_PN: Format: int range [0,63]

Defaults: 16

If this number of frames has been processed, vote for final result.

FLD_RSHIFT: Format: int range [0,63]

Defaults: 10

Number of bits to right-shift the cos/sin value based on the amplitude of difference signal.

FLD_SCENE_CHANGE_TH: Format: int range [0,1048575]

Defaults: 300000

Scene change threshold.

FLD_RESET: Format: bool range {0,1}

Defaults: 0

Reset internal variables of the detection algorithm, restart detection.

Sensor information for flicker detection This is information read from the sensor object but can be saved to record the value that was used:

Vertical total: Total size in lines of the sensor (including black borders).

Frame rate: Expected frames per seconds.

Warning: The information used to be loaded from following parameters which are now deprecated. The information is loaded directly from the Sensor object attached to the Pipeline that owns the module.

FLD_VTOT: Deprecated, saved as SENSOR_VTOT.

FLD_FRAME_RATE: Deprecated, see SENSOR_FRAME_RATE.

DNS_SENSOR_BITDEPTH: Deprecated, see SENSOR_BITDEPTH.

DNS_SENSOR_WELLDEPTH: Deprecated, see SENSOR_WELL_DEPTH.

See *Sensor High-level Parameters* (page 279) for details on the sensor high level parameters.

Focus Statistics (FOS)

The FOS module snoops a column of pixels from the pipeline and collects statistics on a 5x5 kernel from the selected regions (1 regions of interest and 1 global grid) to detect edges.

The minimum ROI size is 8 pixels in width or height. The global statistics is composed of 7x7 tiles each of the same dimension.

The global grid must be encapsulated within the frame. All the pixels inside the grid will be captured and cannot be within 2 pixels of the picture's edges.

It is valid for the ROI to be partially located off the frame at the bottom or right edges. However the statistics may not be accurate unless the whole tile is located within the image (similar to the EXS grid illustrated in *Tile update when off frame (source: TRM)* (page 234)).

- If a tile is completely off the right edge of the picture then 0 will be stored for that tile.
- If a tile is partially calculated then that value will be stored.
- If a tile is completely off the bottom edge of the picture they will not be updated and may contain memory from previous frames.

Used by the *Automatic focus (AF)* (page 269) if the sensor supports focus controls.

FOS High Level Parameters

FOS_GRID_ENABLE: Format: bool range {0,1}

Defaults: 0

Enable the Focus Statistics grid.

FOS_GRID_START_COORDS: Format: int[2] range [0,32767]

Defaults: 0 0

Sets the position of the top-left corner of the grid in pixels.

A grid is composed of 7 tiles.

FOS_GRID_TILE_SIZE: Format: int[2] range [0,32767]

Defaults: 0 0

Sets the size of the grid tiles in pixels.

FOS_ROI_ENABLE: Format: bool range {0,1}

Defaults: 0

Enable the Region Of Interest.

FOS_ROI_START_COORDS: Format: int[2] range [0,32767]

Defaults: 0 0

Sets the position of the top-left corner of the region of interest in pixels.

FOS_ROI_END_COORDS: Format: int[2] range [0,32767]

Defaults: 0 0

Sets the position of the bottom-right corner of the region of interest in pixels (inclusive). Sets the position of the bottom-right corner of the region of interest in pixels (inclusive).

FOS Coordinates

The coordinates are inclusive. Therefore for a region covering the whole of a 720p image it would be:

```
FOS_ROI_START_COORDS 0 0
FOS_ROI_END_COORDS 1279 719
```

Luma Histogram Statistics (HIS)

The HIS module collects statistics from a column of luma pixels (or a single pixel for parallelism 1) on the distribution of values for the picture and a selected region of the picture. The module only records the top pixel of the column of pixels that is transferred at the snoop point. The collected values are scaled to fit the complete input pixel bit range before being ordered to form a histogram at either global or regional level.

The regional statistics are composed of a 7x7 grid over the whole image with each tile being the same size. The tiles should be at least 10 pixel wide and 8 pixels high.

Warning: For HW configurations with parallelism >1 only the top of the column of pixels are used for statistics generating a sparse histogram.

Used by *Automatic Exposure (AE)* (page 245), *Automatic Tone Mapping Control (TNMC)* (page 272) and *Light Based Control (LBC)* (page 275).

HIS High Level Parameters These parameters may be ignored and overridden by some control algorithms!

HIS_GLOBAL_ENABLE: Format: bool range {0,1}

Defaults: 0

Enable/disable global histogram statistics.

HIS_REGIONAL_ENABLE: Format: bool range {0,1}

Defaults: 0

Enable/disable regional histogram statistics.

HIS_GRID_START_COORDS: Format: int[2] range [0,8191]

Defaults: 0 0

The position of the upper-left corner of the region in pixels.

HIS_GRID_TILE_DIMENSIONS: Format: int[2] range [8,4095]

Defaults: 10 8

The dimensions of the region in pixels.

HIS_INPUT_OFFSET: Format: int range [0,1023]

Defaults: 256

The offset to be subtracted from the input luma value in order to remove signal foot room.

HIS_INPUT_SCALE: Format: int range [0,65535]

Defaults: 32767

The scaling factor used to scale the input luma value according to the desired gamut.

White Balance Statistics (WBS)

The WBS module uses a column of pixels to collect statistics at the R2Y point in the pipeline (it accumulates both RGB and YUV statistics). It has several ROI and 1 global region.

Used by *Automatic White Balance (AWB)* (page 252).

WBS High Level Parameters

WBS_ROI_ENABLE: Format: int range [0,2]

Defaults: 0

Number of enabled regions of interest.

WBS_RMAX_TH: Format: double[2] range [0,1]

Defaults: 0.75 0.75

1 value per ROI.

White patch Red channel threshold.

WBS_GMAX_TH: Format: double[2] range [0,1]

Defaults: 0.75 0.75

1 value per ROI.

White patch Green channel threshold.

WBS_BMAX_TH: Format: double[2] range [0,1]

Defaults: 0.75 0.75

1 value per ROI.

White patch Blue channel threshold.

WBS_YHLW_TH: Format: double[2] range [0,1]

Defaults: 0.75 0.75

1 value per ROI.

High luminance white luma channel threshold.

WBS_ROI_START_COORDS: Format: int[4] range [0,32767]

Defaults: 0 0 0 0

2 values per ROI.

Sets the position of the left/top corner of the regions of interest in pixels.

WBS_ROI_END_COORDS: Format: int[4] range [0,32767]

Defaults: 0 0 0 0

2 values per ROI.

Sets the position of the right/bottom corner of the regions of interest in pixels.

WBS Coordinates The coordinates are inclusive. Therefore for a region covering the whole of a 720p image it would be and another covering half it would be:

```
WBS_ROI_START_COORDS 0 0 0 0
WBS_ROI_END_COORDS 1279 719 639 359
```

Auto White Balance Statistics (AWS)

The AWS module uses a tile based method of collecting statistics of CFA cells which contain information about all grey points (that is closest to Planckian Locus curve located in R/G - B/G ratio domain) found in current capture.

For more information on AWS module please refer to TRM.

Used by *Automatic White Balance (AWB)* (page 252).

AWS High Level Parameters

AWS_ENABLED: Format: bool range {0,1}

Defaults: 0

Enable AWS statistics module in ISP

AWS_DEBUG_MODE: Format: bool range {0,1}

Defaults: 0

Enable statistics debug bitmap mode.

AWS_TILE_START_COORDS: Format: int[2] range [0,32767]

Defaults: 0 0

Left top coordinate of tile [0,0] in statistics grid in pixels.

AWS_TILE_SIZE: Format: int[2] range [0,32767]

Defaults: 16 16

Width and height in pixels of single tile.

AWS_LOG2_R_QEFF: Format: double range [0,256]

Defaults: 1

Value of log2() of Red channel quantum efficiency.

AWS_LOG2_B_QEFF: Format: double range [0,256]

Defaults: 1

Value of log2() of blue channel quantum efficiency.

AWS_R_DARK_THRESH: Format: double range [0,256]

Defaults: 8

Dark threshold for red channel.

AWS_G_DARK_THRESH: Format: double range [0,256]

Defaults: 8

Dark threshold of green channel.

AWS_B_DARK_THRESH: Format: double range [0,256]

Defaults: 8

Dark threshold of blue channel.

AWS_R_CLIP_THRESH: Format: double range [0,256]

Defaults: 128

Clip threshold of red channel.

AWS_G_CLIP_THRESH: Format: double range [0,256]

Defaults: 128

Clip threshold of green channel.

AWS_B_CLIP_THRESH: Format: double range [0,256]

Defaults: 128

Clip threshold of blue channel.

AWS_BB_DIST: Format: double range [0,128]

Defaults: 0.2

Maximum distance from Planckian Locus curve for a CFA being considered to be white and collected to statistics.

AWS_CURVES_NUM: Format: int range [0,5]

Defaults: 0

Number of Planckian Locus curve approximation lines defined

AWS_CURVE_X_COEFFS: Format: double[5] range [-16,16]

Defaults: 0.886719 0.966797 0.998047 0 0

List of X coefficients of Planckian Locus curve approximation lines.
AWS_CURVES_NUM items.

AWS_CURVE_Y_COEFFS: Format: double[5] range [-16,16]

Defaults: 0.460938 0.254883 0.0644531 0 0

List of Y coefficients of Planckian Locus curve approximation lines.
AWS_CURVES_NUM items.

AWS_CURVE_OFFSETS: Format: double[5] range [-16,16]

Defaults: 0.0390625 0.255859 0.636719 0 0

List of offsets of Planckian Locus curve approximation lines. AWS_CURVES_NUM items.

AWS_CURVE_BOUNDARIES: Format: double[5] range [-16,16]

Defaults: 0.25 -0.84375 -1.90625 16 16

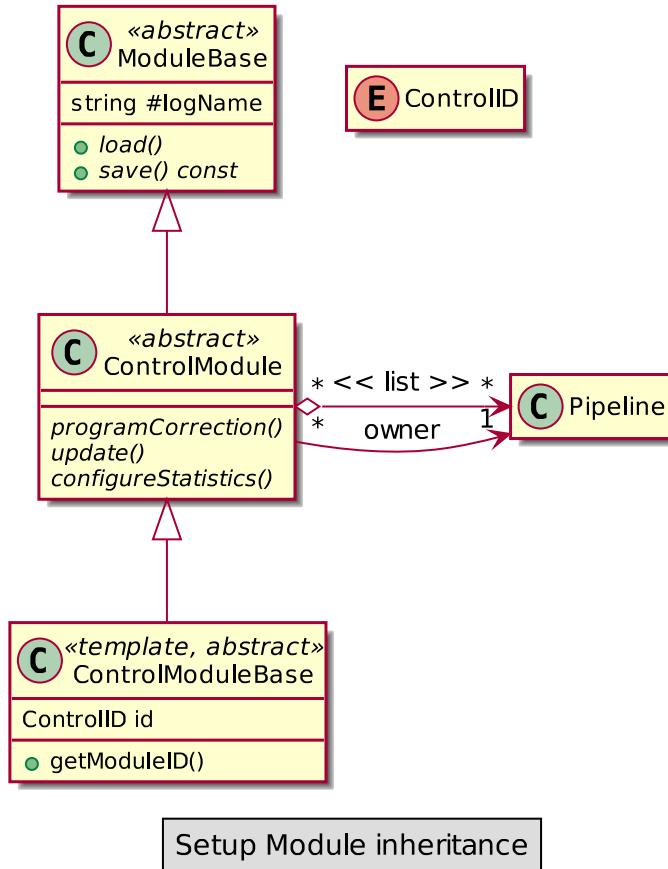
List of Planckian Locus curve approximation lines ‘Y’ boundaries. AWS_CURVES_NUM items.

8.4 Control and ControlModule Classes

The Control and ControlModule classes are targeted to be able to easily “attach” or “detach” control loop algorithms to the Camera. As described in previous sections the Camera object maintains an instance of the Control class which is used to register an arbitrary number of ControlModule instances. Each ControlModule would typically represent a different algorithm so different versions or new functionalities could be easily integrated, substituted or tested. However it would be perfectly possible to implement all the loop algorithms in a single ControlModule.

The ModuleBase abstract class is the root for every module. It defines the abstract **load()** and **save()** function that allow the manipulation of high-level parameters using a ParameterList object. This class is used by both Setup and Control modules.

Any new control algorithm must extend the ControlModule class which serves as an interface. It is mandatory to implement some methods: the **load()**, **save()** and **update()** functions.



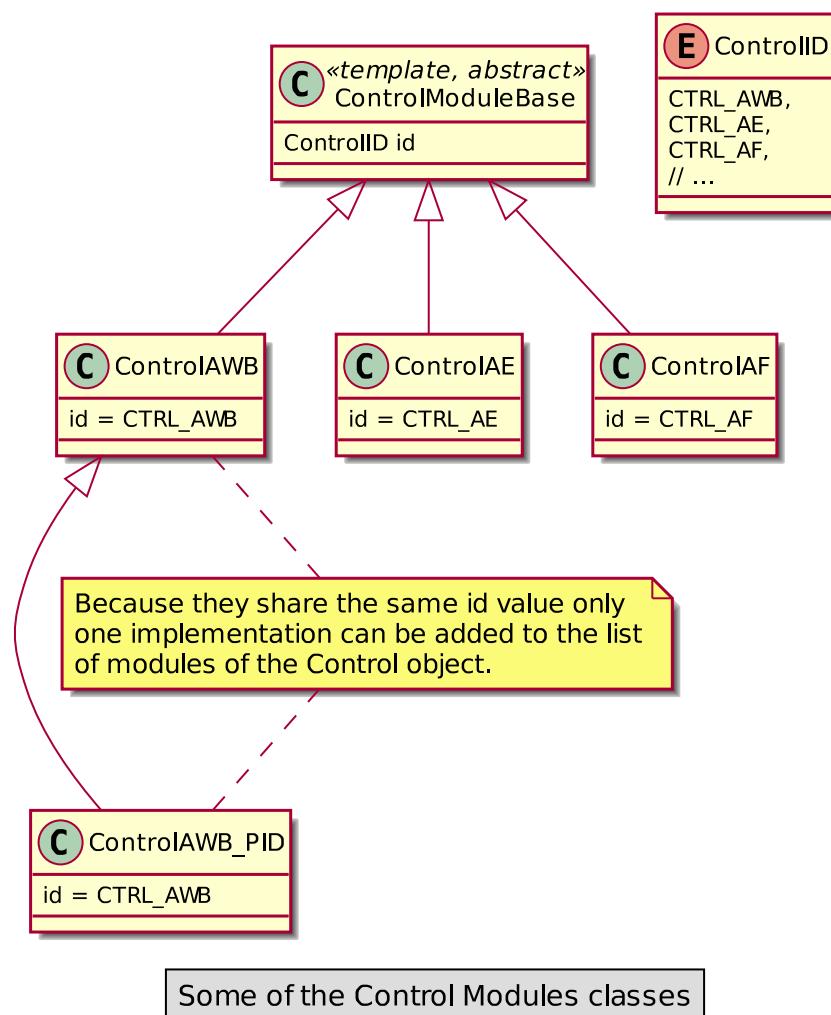
Similar to a setup module, the load() function is in charge of loading any configuration for the control algorithm from a ParameterList (if the algorithm requires so). The save() function saves the current configuration of the control algorithm.

The update() function will implement the control algorithm's logic. The update() function will be called every time a new shot is captured with the camera if the ControlModule update mode has been set to UPDATE_SHOT module. If the update mode is set to UPDATE_REQUEST the control module will not be called every capture but only on demand. By default the registered ControlModule instances are set to UPDATE_SHOT mode. Other functions are available in the doxygen documentation.

In the same manner as the setup modules, an intermediate class is used in between the ControlModule and actual module implementation to encapsulate the template declarations. Each implemented module uses a unique ControlID value that can be used to retrieve the module using a simple template function in the Pipeline as explained in the section [Module Setup and Update](#) (page 202).

The Control Module classes figure shows the set of already implemented control modules:

- ControlAE Implements the automatic exposure algorithm.
- ControlAWB Implements automatic white balance algorithm using interpolated gain/ccm control points.
- ControlAWB_PID Implements automatic white balance algorithm using PID controlled gains and interpolated CCM control points. It is an alternative to the original AWB and behaves better non-laboratory conditions.
- ControlDNS allows the update of some of the Denoiser parameters (sensor gain).



- ControlLBC Light Based Control, different sharpness, contrast, saturation and brightness settings for different captured light levels.
- ControlTNM Dynamic generation of global tone mapping curves and dynamic strength settings.

Further details on each control module included in the ISPC implementation can be found in the *Implemented Control Modules* (page 245) section.

8.4.1 Registering and Retrieving a ControlModule

As opposed to the case of the Pipeline instance in the Camera object, which on creation is populated with the default set of pipeline SetupModule instances, these are not ControlModule instances registered by default in the Control object. That's because it is required to configure the pipeline modules for a proper operation of the camera while it is not necessary to have any loop control algorithm at all.

The Camera::registerControlModule() can be used for registering new ControlModule instances through an available Camera instance. In order to register a new control module in the camera a pointer to a ControlModule instance and an ID must be provided. It must be noted the Camera will take care of the destruction of the registered ControlModule and therefore the allocated instances should not be deleted ‘manually’ once registered in the camera.

Each control module must be registered with a unique ID from a set of predefined ones (see ControlID enum in Module.h). The ID is used for retrieving a particular control module from the registered ones. There is a set of predefined IDs available (subject to be extended when new types of control algorithms become available) which also include a couple of generic IDs (CTRL_AUX1, CTRL_AUX2) for registering miscellaneous control modules. The following example shows how to register a new control module:

```
ret = camera.registerControlModule(new ControlAWB());
```

As in the example above, it is necessary to dynamically allocate a ControlModule as registered control modules will be automatically destroyed when the Camera instance is destroyed. In order to retrieve one of the registered ControlModule instances the Camera::getControlModule() method is available and the ID used for registering the control module is used:

```
static_cast<ControlAWB*>(
    camera.getControlModule(CTRL_TNM))->setAllowHISConfig(true);
```

As with setup modules, the template method has been provided to help avoid use of casting:

```
camera.getControlModule<ControlTNM>()->setAllowHISConfig(true);
```

The example shows how a ControlModule pointer can be retrieved and used for calling any specialized function implemented in the module. It will be common that different implementations of control modules inheriting from ControlModule will have implemented additional functionality and methods to be configured, etc.

In the above example the ControlTNM needs to setup the Histogram statistics modules. If a ControlAE module was not registered to the Camera the HIS will not be configured and therefore the ControlTNM can be allowed to configure the statistics as it needs them to operate.

As previously mentioned a ControlModule is registered by default so its ControlModule::update() function is called after each captured shot. In order to change this behaviour

the ISPC::ControlModule::setUpdateType() function can be used to set a different update policy and the *ControlModule::update()* function can be used to ‘manually’ run an iteration of a control module at any time.

8.4.2 Additional Functionalities

The Camera class only provides a limited number of functions to interact with the Control and register new ControlModule. There is some additional methods to manage the registry of control modules (clear it, run a set or a particular control module) available in the Control class. For accessing these functionalities the Control class instance of the Camera class has public access. See the doxygen documentation for more details and a complete set of available functions).

8.4.3 Implemented Control Modules

This section describes the currently implemented algorithms in the ISPC library (version 2).

A brief description of the algorithms is provided followed by a longer description (if necessary) as well as notes about the implementation, configuration and dependencies with statistics modules, etc.

The control modules in the ISPC library are implemented by extending the ControlModule class and require fulfilling certain conditions. More details about how the ControlModule classes are implemented and how they interact with the rest of the ISPC library are provided in the *ISP Control Library* (page 190) document and it is recommended to read that document before this one.

The working algorithms include:

- *Automatic Exposure (AE)* (page 245)
- *Automatic White Balance (AWB)* (page 252)
- *Automatic focus (AF)* (page 269)
- *Automatic Tone Mapping Control (TNMC)* (page 272)
- *Lens Shading Grid Control* (page 271)
- *Light Based Control (LBC)* (page 275)
- *Denoiser Control* (page 278)

Automatic Exposure (AE)

The automatic exposure algorithm is designed to control the programmed sensor exposure time and gain aiming to reach a target ‘brightness’ value for the image. The ‘brightness’ concept, as used in this context, does not represent an actual average of the image intensity but a value related to the image brightness but computed in a more elaborate way (for example taking into account the location information, under/over exposed pixels, etc.) The auto exposure algorithm employed also includes a flicker rejection mechanism by calculating the exposure time and gain combination to be programmed in the sensor in a way in which the light flickering effect is avoided or minimized (when possible). Note that the target sensors are for mobile market and therefore it is assumed that we do not have control over the aperture.

In this document and in the available code there are two differentiated concepts than can lead to confusion: “exposure value” and “exposure time”. The “exposure time” concept corresponds to the amount of exposure time programmed in the sensor. On the other hand “exposure value” represents a combination of the exposure time and the gain:

$$\text{exposure value} = \text{exposure time} \cdot \text{gain}$$

Therefore the exposure value is a combination of the programmed exposure time and gain and equivalent exposure values can be obtained with different combinations of exposure times and gains. Ideally, assuming the camera capturing the same scene, different captures with different exposure times and gains but the same combined exposure should produce images with approximately the same brightness measurement.

Note: Both exposure time and exposure value are sometimes shortened in the comments and code to ‘exposure’ only, in which cases the context should help to specify what of the two concepts we are talking about.

Code organization

The AE control is implemented in the ControlAE class in the ISPControl 2 library. Such class, as the rest of the control algorithms for the capture, inherits from the ControlModule class and the main entry point is the ControlAE::update() function. The ControlAE::update() function of every control module registered in a given Camera instance is called once per captured shot (unless it is set up in a different way), being able to update whatever parameters are required regularly. The ControlAE::update() function receives an ISPC::Metadata structure containing the metadata corresponding to a shot as generated by the pipeline. Within the update function in ControlAE the code is split in three steps:

1. Calculate a brightness metering from the shot metadata: Carried out in the call to ControlAE::getBrightnessMetering() function.
2. Calculate update values for gain and exposure time to be programmed in the sensor. Values are based on the previous ones programmed in the sensor, maximum and minimum exposure time and gain for the sensor, light flicker frequency, etc.
3. Program the updated exposure time and gain in the sensor with the setExposure() and setGain()

Statistics Configuration

The AE algorithm computes the brightness measure making use of V2500 statistics modules. Therefore such modules must be configured in a proper manner in order for the AE to operate properly.

The used statistics come from the image histograms (*HIS* module) and potentially from the exposure clipping statistics as well (*EXS*) although the latter ones have not been used in the current AE implementation and are not taken into consideration.

The HIS statistics are in principle configurable using the Felix setup file but whatever configuration is specified there is overridden by the ControlAE module once it starts operating. The ControlAE::configureStatistics() function is in charge of the configuration and is called in the first call to the module ControlAE::update() function.

The histogram extraction configuration must be configured so the histogram range covers the whole image pixels possible range when the image reaches the HIS module. That is, the beginning of the histogram range should correspond to pixels which receive no (or a negligible) amount of light. The end of the histogram range must correspond with the pixels maximum clipping value. If the range is set up incorrectly the AE algorithm won't work properly. Note that the tone mapper relies also on the image histogram and the same requirements for the histogram range configuration.

By default the `configureStatistics()` function sets up the histogram range between the -64 and 64 value range (expressed in a s8.x precision for the input values in the HIS module). Therefore, instead of expecting to change the histogram configuration it is expected that the previous modules in the pipeline are configured in a way which yields to such valid pixel values range. Also the histogram 7x7 grid is configured to cover as much area as possible from the input image (using the image size obtained from the sensor mode) while centring the grid in the image.

Note: For more information on the configuration of the HIS module check the TRM and the rest of the module related documentation.

Algorithm overview

As described above the code for the AE is split in two parts: First getting a brightness measure from the image and then using the calculated brightness, the target one and other parameters to compute the next exposure time and gain values to be programmed in the sensor.

Brightness measurement The brightness measurement is calculated making use mainly of the 7x7 histogram grid placed centred in the image and covering the largest possible area (without going out of bounds). The measurement is weighted so larger weight is assigned to the central regions of the image and therefore a proper exposure in the centre of the image will have more priority than in the areas close to the image borders. There are two 7x7 weights matrices defined in the code which are combined:

- **WEIGHT_7x7_A** matrix represents a very spread centred priority weights array (so the difference between the central grid tiles and the rest of tiles in the grid is not very high).
- **WEIGHT_7x7_B** weight matrix is also centre weighted but with much larger weights in the central area compared with the surrounding tiles.

Some of the functions in the code (like `getWeightedStatsBlended()`) apply a mixture of both matrices' weights combined with a given combination weight so the algorithms can be tuned to be more or less centre-based.

All the brightness metering calculations are encapsulated in the `getBrightnessMetering()` function and from the 7x7 histogram grid several 7x7 arrays of statistics are gathered: average brightness per tile and overexposed/underexposed measurements per tile. The under/over exposed values are just a measurement of a large amount of pixels in the tile being under/over exposed (or close to being under/over exposed). A brightness metering value is calculated by combining the previously enumerated measurements.

There's also a simple backlight detection mechanism: a 7x7 grid defining foreground/background tiles (see **BACKLIGHT_7x7** matrix in the code) is used to compute the average brightness of the image in areas considered as background/foreground (no changes of orientation in the image are taken into account). When there is large difference between background and foreground a

‘backlight accumulator’ value is increased. Such value is used as weight to apply two different brightness calculation approaches. Summarizing, if a big difference between background and foreground is present for a prolonged amount of time bigger weight will be given to the foreground brightness metering so the AE will try to adapt to that area of the image.

The `getBrightnessMetering()` function returns a value between -1 and 1 (-1 would correspond with a totally black frame while 1 while correspond with a totally overexposed, white, frame). For more details check the `getBrightnessMetering()` function and code comments on auxiliary functions.

Calculation of exposure time and gain For each iteration the exposure time and gain to be programmed in the sensor for the following capture are calculated based on the previous shot brightness measurement, the previously programmed sensor exposure time and gain and the brightness target. The algorithm estimates what would be exposure value required to get the desired brightness and then calculates the combination of exposure time and gain applied according to the light flickering frequency and sensor configuration constraints.

It must be noted that the estimated exposure value is not just applied straight away as a too abrupt change in the sensor settings could yield to AE oscillations and also it must be considered that the exposure value is estimated. Therefore only incremental changes are made in the exposure time and gain until the desired brightness measurement is obtained smoothly. In the `ControlAE` class the update speed is controlled with `setUpdateSpeed()` function which allows controlling the speed of the sensor settings updates.

All the calculations for the AE settings are carried out in the `getAutoExposure()` function in the following steps:

- The new target exposure value is calculated in the `autoExposureCtrl()` function already including the speed control.
- If one of the gain or exposure time values are fixed to a pre-set value the other one is calculated accordingly.
- If exposure or/and gain are not fixed, the transfer function is applied for given target exposure value, providing exposure time and gain components.
- Once exposure time and gain are calculated, if the flicker rejection is activated the exposure time and gain values are corrected to avoid flickering with the call to the `getFlickerExposure()` function.
- Finally if the sensor has a discreet set of possible values for the exposure time the gain and exposure time are corrected in the call to the `adjustExposureStep()` function.

The behavior of transfer function calculating sensor exposure and sensor gain for given target exposure value has been described below:

- For lower range of exposure time, the function maintains *theoretical* gain at constant value 1.0 with the exposure time function growing continuously up to some specified level below or equal to sensor frame duration, called **target exposure**. This guarantees no reduction of capture FPS.

Note: Because the real sensor value is corrected using `adjustExposureStep()`, the gain must be modified accordingly and will differ from theoretical value.

This can be seen as sawtooth-like line on the *Diagrams* (page 249) on Zone 0 blue gain line.

For the case of disabled flicker rejection, this knee point of exposure time is equal to the frame duration for current sensor mode (light blue dashed line on first diagram below).

If flicker rejection is being applied, the **target exposure** is the runtime calculated duration equal to integer number of flicker periods less or equal to the sensor frame duration. It is represented as light green dashed line on second diagram below.

- After exposure time reaches the level of **target exposure**, it's is being maintained at constant level by increasing gain value up to **target gain**, which is represented by dark green dashed line.
- After the gain reaches **target gain**, the function cannot maintain constant FPS anymore. From this point, the value of gain is maintained below or equal to target gain, and the exposure time is allowed to increase up to the **max AE exposure**, which is the absolute maximum exposure AE algorithm can reach. It should be noted that this value is software limit only and should be set below the maximum exposure the sensor hardware can handle.

Note: If flicker reduction is enabled, **max AE exposure** value is rounded down to nearest integer number of flicker periods. This level is represented as light green dashed line on the second diagram below.

- After exposure reaches **max AE exposure**, it is maintained at this maximum level, and the target exposure is being tracked by increasing gain up to the level of **max AE gain**. The final gain applied in sensor is clipped to this value.
- Everything above **max AE gain** causes the final capture to be underexposed. This state can be read by calling `isUnderexposed()` method.

For flicker reduction cases, the exposure time grows in steps of flicker period. This means 20ms step for 50Hz and 16.(6)ms for 60Hz AC power supplies. The output gain is corrected accordingly to maintain target exposure.

Diagrams The diagrams below visualize the transfer function converting target exposure to (exposure time, gain) pairs, as implemented in *ControlAE* module.

The sensor exposure time is represented on left vertical axis, and sensor gain on the right. The horizontal axis represents the samples for increasing target exposure (not shown on the diagram).

Both diagrams were captured for sensor FPS equal to 18 frames per second.

Flicker rejection can be enabled or disabled in the *ControlAE* class with the `enableFlickerRejection()` function.

AE High Level Parameters

This Control Module configures the sensor's exposure and gain to use according the HIS statistics gathered.

AE_BRACKET_SIZE: Format: double range [0,1]

Defaults: 0.05

Margin around the target brightness which we will consider as a proper exposure (no changes in the exposure settings will be carried out).

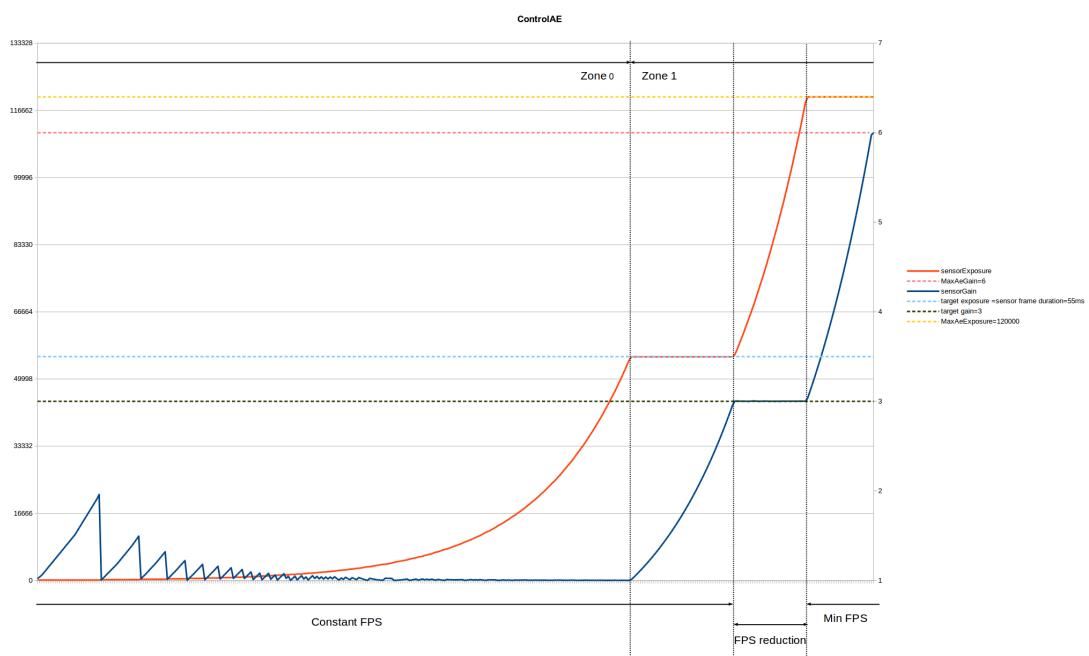


Fig. 8.18: Visualization of auto exposure algorithm with disabled flicker reduction

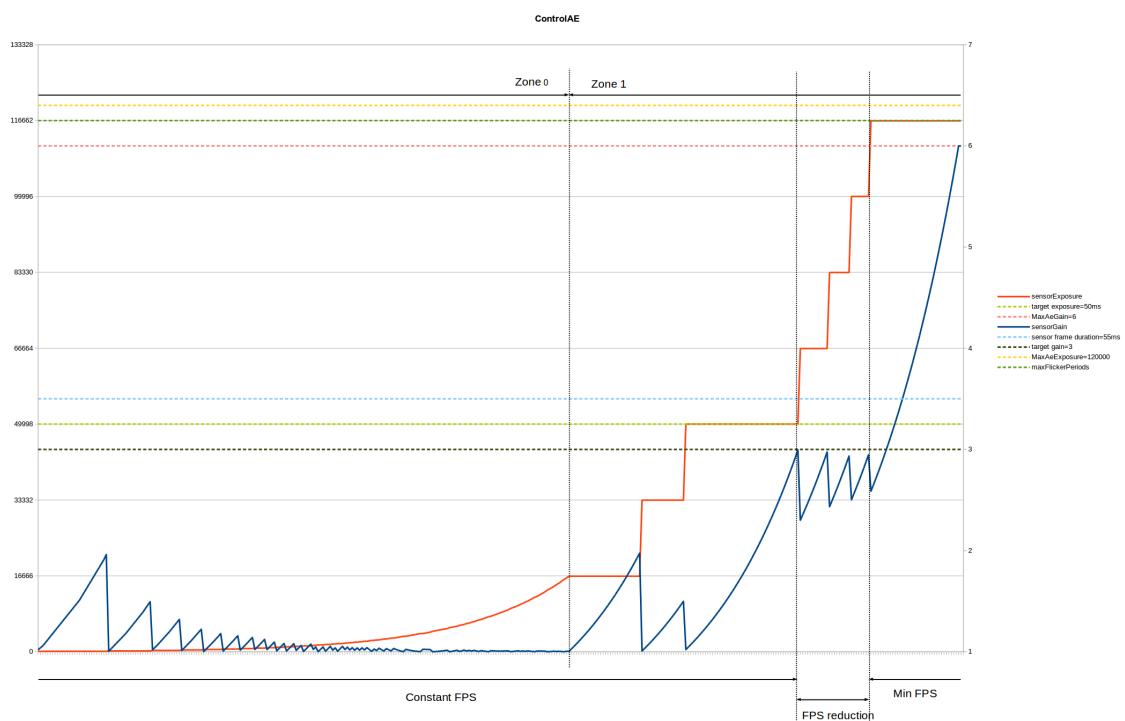


Fig. 8.19: Visualization of auto exposure algorithm with flicker reduction set to 60Hz

AE_FLICKER_REJECTION: Format: bool range {0,1}**Defaults:** 0

Take flicker rejection into account when computed desired exposure.

AE_FLICKER_AUTODETECT: Format: bool range {0,1}**Defaults:** 1Enable use of autodetected flicker frequency read from *Flicker Detection (FLD)* (page 235) module statistics.**AE_FLICKER_FREQ:** Format: double range [1,200]**Defaults:** 50**AE_TARGET_BRIGHTNESS:** Format: double range [-1,1]**Defaults:** 0

Desired brightness.

AE_UPDATE_SPEED: Format: double range [0,1]**Defaults:** 1

Update speed for the exposure. Increase if AE works slowly, reduce if AE doesn't converge (oscillates).

AE_TARGET_GAIN: Format: double range [1,MAX_DOUBLE]**Defaults:** MAX_DOUBLE

Knee point above which AE allows decrease of current FPS rate by increasing sensor exposure above current sensor frame duration.

AE_MAX_GAIN: Format: double range [1,MAX_DOUBLE]**Defaults:** MAX_DOUBLE

Maximum gain AE module can set in sensor. This is software only limit, purposed to be set below maximum gain supported by sensor.

AE_MAX_EXPOSURE: Format: long int range [1,MAX_LONG_INT]**Defaults:** MAX_LONG_INT

Maximum exposure AE module can set in sensor in microseconds. This is software only limit, purposed to be set below maximum exposure supported by sensor. Practical values are less than 200ms.

Automatic White Balance (AWB)

The automatic white balance algorithm is in charge of estimating the captured scene illuminant temperature and set the colour correction matrix, gains and offsets accordingly to compensate for the colour cast produced by the illuminant.

V2500 hardware provides various types of statistics to support AWB mechanism.

AC/HLW/WP statistics

Estimation is based on statistics gathered by the *White Balance Statistics (WBS)* (page 238) module in the pipeline and will provide accumulated pixel values for the R,G,B channels using three different approaches:

- AC - Average Colour: Added up value per channel (RGB) of all the pixels in the image.
- HLW – High Luminance White: Added up value per channel (RGB) of all the pixels in the image whose luminance value is above certain threshold.
- WP – White Patch: Added up value per channel (RGB) of all the pixels in the image whose channel value is above certain threshold. A different threshold per RGB channel is applied.

The gathered statistics are post-processed in the AWB algorithm implementation to convert them from accumulated pixel values to averages later used for the illuminant estimation. The AWB algorithm is also in charge of dynamically calculating and programming the thresholds for the HLW and WP statistics in the pipeline WBS module.

Code Organization The automatic WB control has a couple of implementations these are implemented in the ControlAWB and ControlAWB_PID the classes in the ISPControl 2 library. Such class, as the rest of the control algorithms for the capture, inherits from the ControlModule class and the main entry point is the update() function. The ControlAWB::update() function of every control module registered in a given Camera instance is called once per captured shot (unless it is intentionally set up in a different way), being able to update whatever pipeline parameters are required regularly.

The ControlAWB::update() function receives an ISPC::Metadata structure containing the metadata corresponding to a previously captured shot as generated by the pipeline. Such structure is where the statistics for AC, HLW and WP are gathered from. In the case of the automatic white balance algorithms, the ControlAWB::update() function is used just for encapsulation of the ControlAWB::programCorrection() function which contains all the logic for threshold update, illuminant estimation and colour correction calculation. The two implementations differ in their approach. The ControlAWB::programCorrection() function performs all the tasks in the following steps:

- Calculation of the adequate thresholds for the WB statistics: The optimum thresholds to be set up are those which generate statistics using about 7.5% of the pixels only. The estimateWPThresholds() and estimateHLWThreshold() functions deal with this calculation. Both functions make use of the estimateThreshold() function which implements a PID controller for the dynamic adjustment of the thresholds.
- Thresholds are programmed in the pipeline with the `programStatistics()` function.
- A proportional controller adjusts a temperature estimate control variable, the controller attempts get the ratio of R/G and B/G average values of the statistics being used to be 1:1
- Using the temperature control variable an interpolated colour correction matrix, gains and offsets are calculated from the set of predefined corrections for different colour temperatures (see setup parameters section below) in the call to the TemperatureCorrection::getCorrection() method from the TemperatureCorrection class.

- The colour correction matrix gains and offsets are programmed in the pipeline in the call to the `programCorrection()` function.

The `ControlAWB::programAWB_PID()` function performs all the tasks in the following steps:

- Calculation of the adequate thresholds for the WB statistics: The optimum thresholds to be set up are those which generate statistics using about 7.5% of the pixels only. The `estimateWPThresholds()` and `estimateHLWThreshold()` functions deal with this calculation. Both functions make use of the `estimateThreshold()` function which implements a PID controller for the dynamic adjustment of the thresholds.
- Thresholds are programmed in the pipeline with the `programStatistics()` function.
- A pair of PID control loops are used to adjust the red and blue gain factors. The setpoint of the control loops are to reduce the ratios of R/G and B/G to be 1. Note that this balances the green channel as well as the red/blue channels.
- Using the computed red, green and blue gains the temperature is estimated using the `getCorrelatedTemperature()` function.
- The CCM is interpolated from the predefined CCMs for different colour temperatures
- The computed gain factors get the inverse quantum efficiency applied, this is taken from the precomputed gains for the 6500 colour temperature.
- The colour correction matrix gains and offsets are programmed in the pipeline in the call to the `programCorrection` function.

Nearest to Planckian Locus statistics

This estimation algorithm is implemented in `ControlAWB_Planckian` class and uses statistics data collected by *Auto White Balance Statistics (AWS)* (page 239) module. This class inherits from `ControlAWB` and uses `TemperatureCorrection` and `ColorCorrection` auxiliary classes.

Module operation The AWS statistics for each tile contain `log2()` sums of every R, G and B channel accumulated across all analyzed CFA cells and a total counter of CFA's which have met ALL of the following conditions:

- all pixels in CFA are above respective dark thresholds (`AWS_X_DARK_THRESH`)
- all pixels in CFA are below clip threshold (`AWS_X_CLIP_THRESH`)
- CFA distance to Planckian Locus is below `AWS_BB_DIST`
- counters for a tile are not saturated

For every tile i , AWS statistics module produces $COLLECTED_R_i$, $COLLECTED_G_i$, $COLLECTED_B_i$ and CFA_NUM_i registers. These values allow control module to calculate R/G and B/G ratios for every tile i with non zero CFA_NUM_i . The calculation formulas for

R and B channels ratios are presented below:

$$\begin{aligned} \log_2(\text{ratio R}_i) &= \frac{\text{COLLECTED_R}_i + \text{CFA_NUM}_i \cdot \log_2(\text{QEr}) - \text{COLLECTED_G}_i}{\text{CFA_NUM}_i} \\ \log_2(\text{ratio B}_i) &= \frac{\text{COLLECTED_B}_i + \text{CFA_NUM}_i \cdot \log_2(\text{QEb}) - \text{COLLECTED_G}_i}{\text{CFA_NUM}_i} \\ \text{ratio R}_i &= 2^{\log_2(\text{ratio R}_i)} \\ \text{ratio B}_i &= 2^{\log_2(\text{ratio B}_i)} \end{aligned}$$

Ratio R and B formulas for tile i

Where QEr and QEb are sensor quantum efficiencies for R and B channel respectively.

Having R and B ratios for every tile i in a set of N valid tiles, control module processes them in the following sequence:

1. Preprocess the valid tiles by filtering out those which occur below low range bounding box.

This software defined boundary is defined by quantum efficiency of the sensor calibration point captured for the lowest available temperature ($R/G_{low}, B/G_{low}$) with a AWS_BB_DIST threshold margin applied (see *AWS High Level Parameters* (page 240)).

Effectively, the pass condition for each tile is:

$$\begin{aligned} \text{ratio R}_i &< \text{R}/\text{G}_{low} + \text{AWS_BB_DIST} \\ &\quad \text{and} \\ \text{ratio B}_i &> \text{B}/\text{G}_{low} - \text{AWS_BB_DIST} \end{aligned}$$

This step discards all tiles, for which the estimated tile temperature lies outside the lowest temperature point of calibration (please refer to the image below).

2. For a set of N tiles which have passed the previous step, calculate the *main* centroid point using weighted average formula

The centroid represents the heaviest (regarding the number of collected CFA's) center point of a group of tiles in ratio R and B coordinate space (please refer to *the image* (page 255)).

$$\begin{aligned} \text{centroid R} &= \frac{1}{\sum_{i=1}^N \text{CFA_NUM}_i} \cdot \sum_{i=1}^N (\text{ratio R}_i \cdot \text{CFA_NUM}_i) \\ \text{centroid B} &= \frac{1}{\sum_{i=1}^N \text{CFA_NUM}_i} \cdot \sum_{i=1}^N (\text{ratio B}_i \cdot \text{CFA_NUM}_i) \end{aligned}$$

3. Determine whether the tiles form single (close enough) or multiple (distant) *clouds* on R/B plane.

This step is required to properly handle images containing areas with colors close to Planckian Locus but interpreted as different light temperatures.

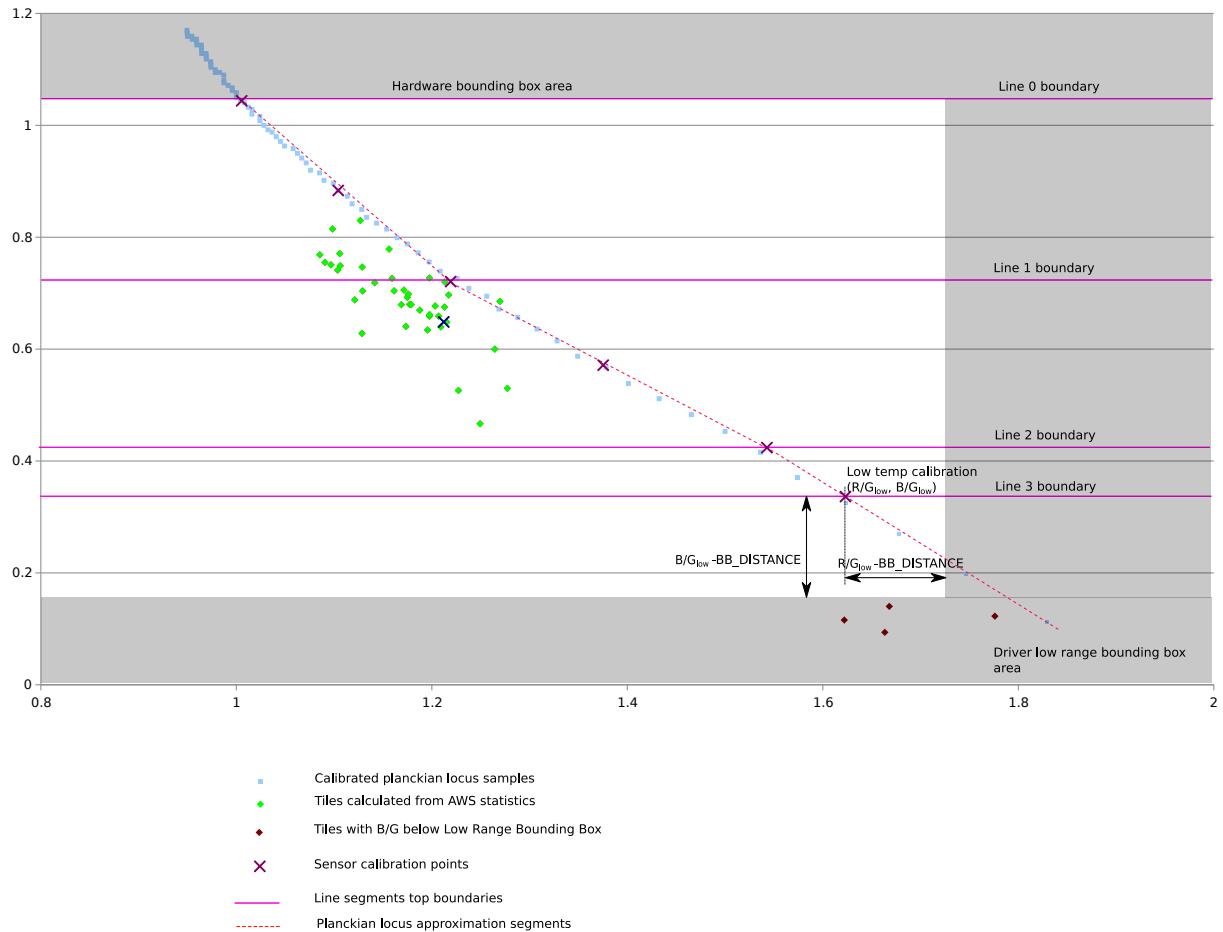


Fig. 8.20: Visualization of R/G (horizontal) and B/G (vertical) ratios, Planckian Locus approximation and tile boundaries.

Note: If, for example, the scene contains substantial areas of white and light blue, and is lit using 2800K warm light source, the white would be categorized as white in 2800K but light blue may have been most probably placed near cold range of planckian locus, 7000K and even higher. In such case main centroid could be calculated in between, near 5000K. If only this point were used as the center for white balance correction, the result of correction would be most probably incorrect.

3.1. Division of statistics to two distinct ‘clouds’ of tiles.

The main centroid point (centroid R, centroid B) is projected on the Planckian Locus, which divides the curve into two parts, called *above* and *below*. For each tile (R_i, B_i) , a point of projection on the Planckian Locus is calculated. If $B_i >$ centroid B, the tile belongs to *above*, otherwise to *below*.

3.2. Then two side centroids, using tiles belonging to *above* or *below* respectively, are calculated.

The same formula as described in [this step](#) (page 255) is used.

3.3. Distance between main centroid point and each side centroid is calculated.

If only one is greater than $1.5 \cdot \text{MaxDistance}$ (on MaxDistance please refer [the paragraph](#) (page 257)), the centroid is treated as ‘weak’ and all tiles composing respective centroid are discarded in further processing. This case is visualized in the picture below, where *above* centroid is strong and *below* is weak.

4. Next step is to filter out the tiles which are too distant from derived centroid point.

The distance is defined as the radius of the circle with the center placed in the point defined by (centroid R, centroid B) coordinates (called *MaxDistance*).

The pass condition for every tile i from a set of N tiles is:

$$\sqrt{(\text{ratio } R_i - \text{centroid R})^2 + (\text{ratio } B_i - \text{centroid B})^2} < \text{MaxDistance}$$

The figure below illustrates the process for an example capture. The horizontal axis is R/G, vertical is B/G. The example MaxDistance has picked radius of 0.2, which is shown as a dark blue circle around centroid cross.

This diagram shows that more distant tiles (that is the tiles with farthest estimated temperature, bright red) won’t take part in final estimation of illuminator.

5. After filtering, the set of M closest tiles is used to calculate output ratios and estimate illuminator temperature and final CCM correction coefficients.

Firstly, AWB driver calculates the sums of COLLECTED_R_j , COLLECTED_G_j , COLLECTED_B_j and CFA_NUM_j fields separately

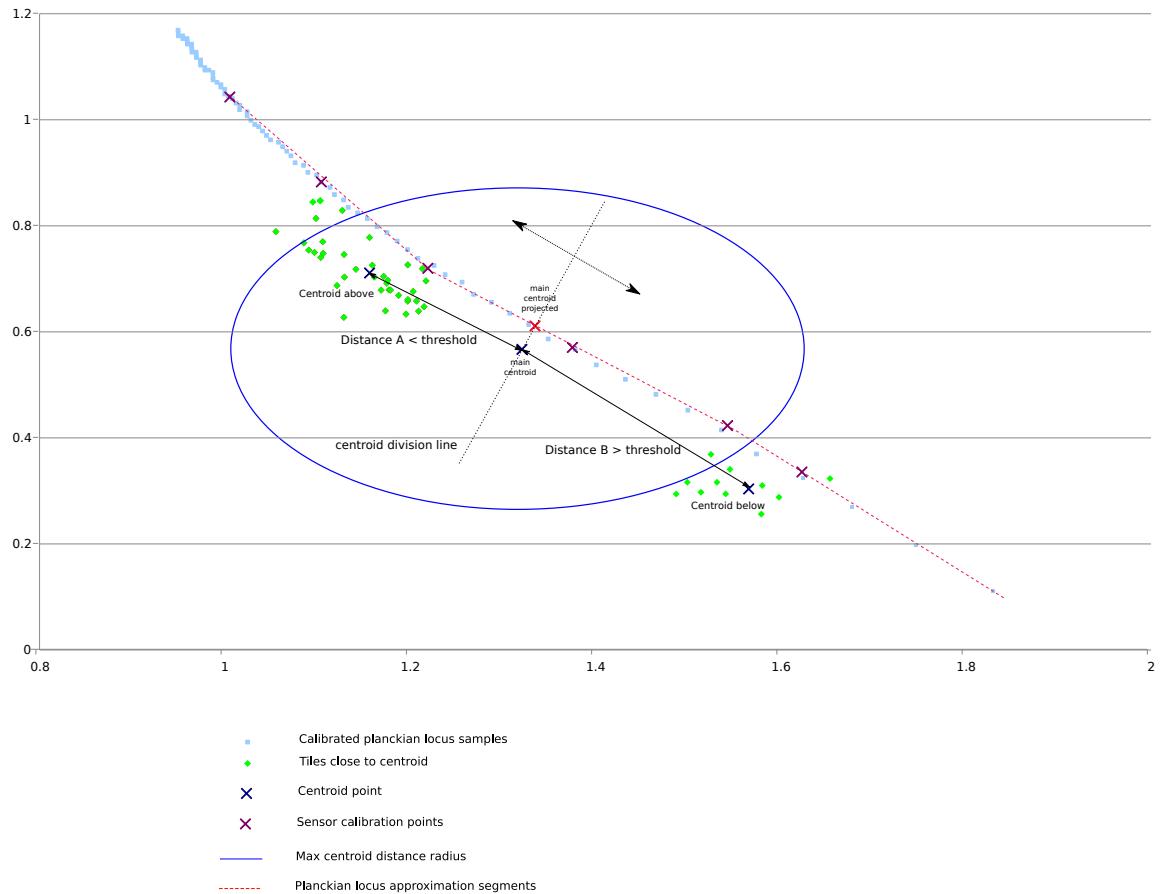


Fig. 8.21: Example distribution of $\log_2(R/G)$ (horizontal) and $\log_2(B/G)$ (vertical) ratios. Tiles form two clouds with below side centroid being too distant from main centroid point.

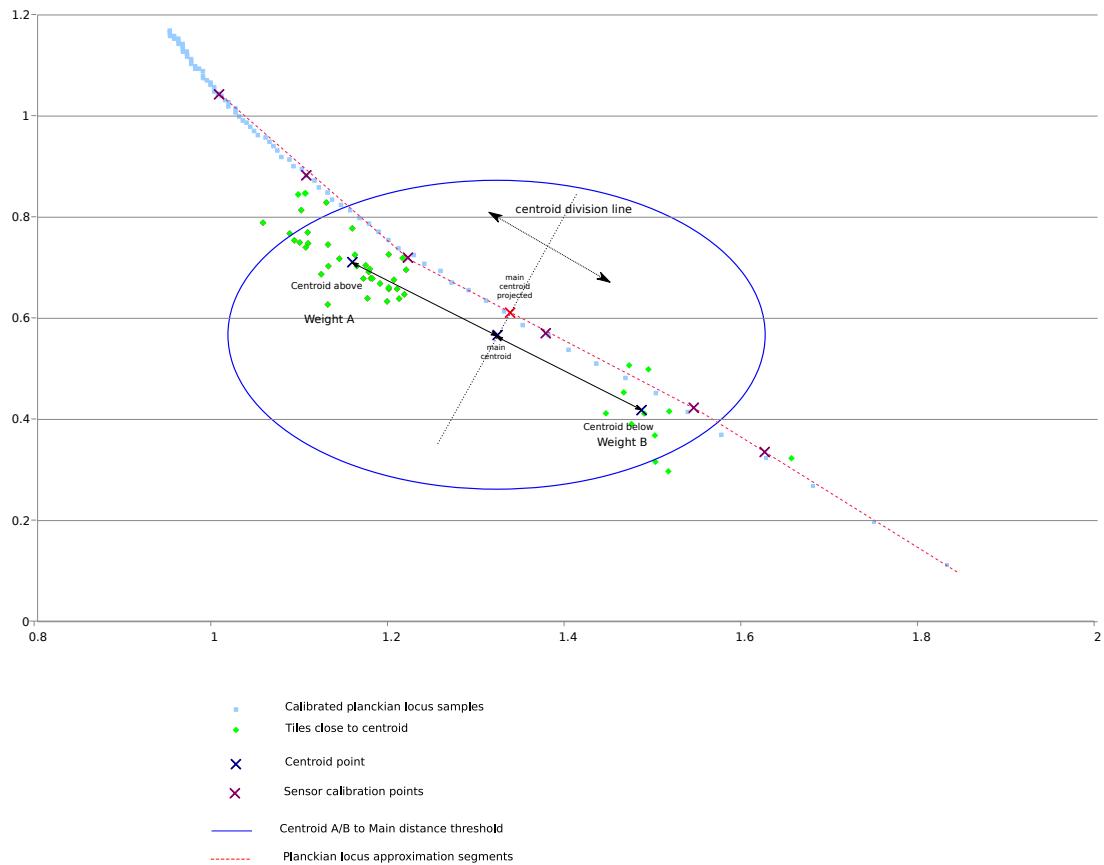


Fig. 8.22: Example distribution of $\log_2(R/G)$ (horizontal) and $\log_2(B/G)$ (vertical) ratios. Tiles form two clouds with both side centroids being close to main centroid point.

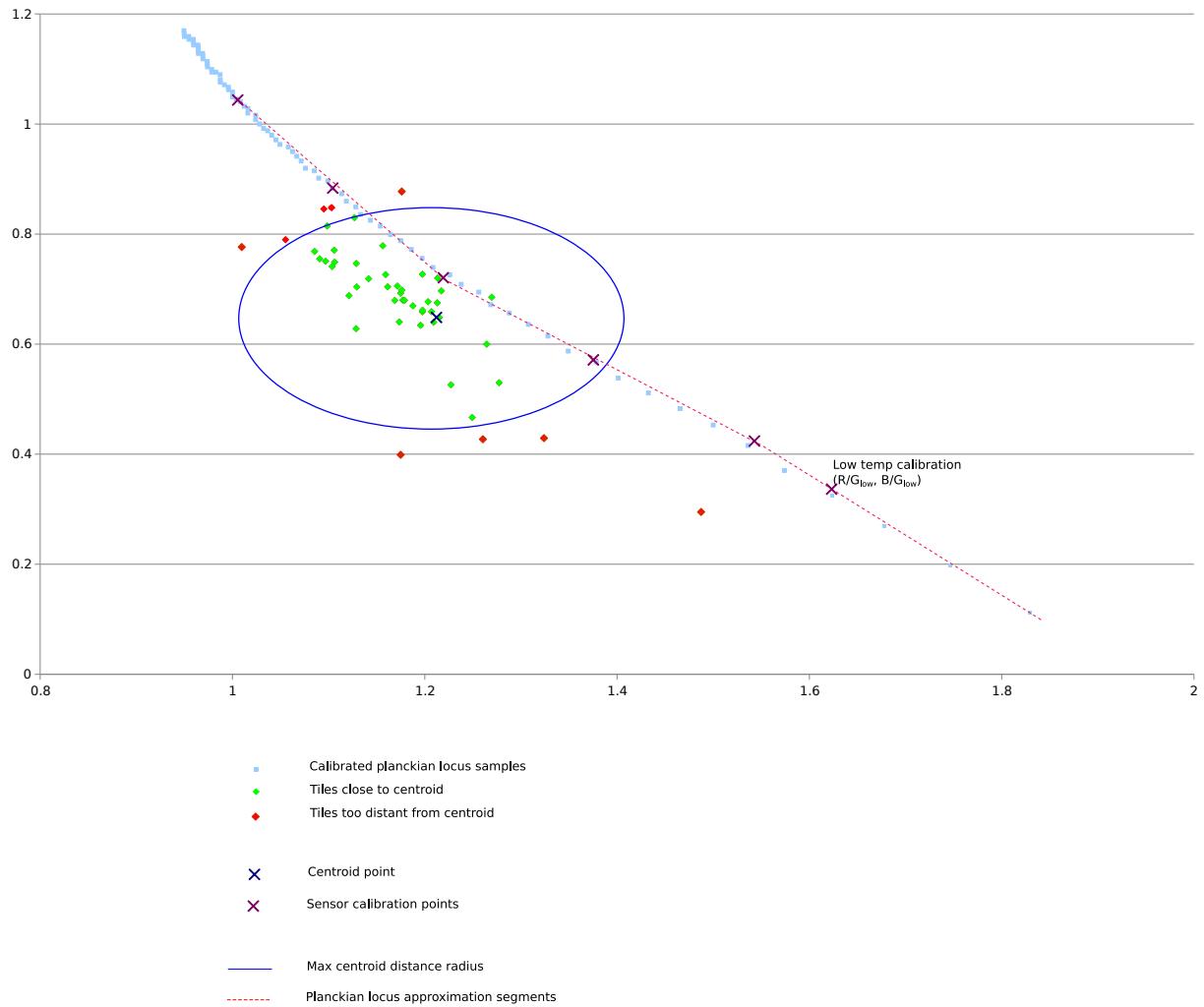


Fig. 8.23: All tiles placed at most MaxDistance to the centroid are marked green and used for final correction.

for all M tiles, getting:

$$\begin{aligned} \text{COLLECTED_R} &= \sum_{j=1}^M \text{COLLECTED_R}_j \\ \text{COLLECTED_G} &= \sum_{j=1}^M \text{COLLECTED_G}_j \\ \text{COLLECTED_B} &= \sum_{j=1}^M \text{COLLECTED_B}_j \\ \text{NUM_CFA} &= \sum_{j=1}^M \text{NUM_CFA}_j \end{aligned}$$

Now the R and B ratios can be calculated using *the formula* (page 254)

Now, just like in *AC/HW/WP statistics* (page 253), the following steps are taken to have the color correction applied for the consecutive captures:

- Using the computed red, green and blue gains the temperature is estimated using the `getCorrelatedTemperature()` function.
- The CCM is interpolated from the predefined CCMs for different colour temperatures
- The computed gain factors get the inverse quantum efficiency applied, this is taken from the precomputed gains for the 6500 colour temperature.
- The colour correction matrix gains and offsets are programmed in the pipeline in the call to the `programCorrection` function.

Auxiliary Classes

All ControlAWB and all it's child classes depend on two auxiliary classes implemented to encapsulate additional functionality, structures and operations.

Colour Correction The `ColourCorrection` class encapsulates colour correction transform information: colour correction matrix coefficients, offsets and white balance gains. It also overloads addition and product operators to ease the interpolation of colour correction transforms.

Temperature Correction The `TemperatureCorrection` class provides functionality for estimation of colour correlated temperature from RGB values as well as a mechanism for storing an arbitrary number of colour transforms associated to specific colour temperatures. It provides the colour correction interpolation when transformed for specific colour temperatures not included in the list are requested.

The class implements three methods of Correlated Color Temperature estimation:

- Planckian Locus linear approximation from sensor calibration points

Used as the default temperature correlation algorithm to obtain most precise temperature correlation for given calibrated imager sensor.

The paragraphs *Line Segment* (page 262) and *Line Segments* (page 262) provide some details of math behind this method.

- McCamy's formula

Used as fallback in case no calibration points are available.

Note: Please refer to McCamy, Calvin S. (April 1992). "Correlated color temperature as an explicit function of chromaticity coordinates". Color Research & Application 17 (2): 142–144

- sRGB colorspace table lookup with interpolation

Provided for reference, not used in current image processing.

Line Segment Each line segment, which approximates the sensor calibration curve between two calibration points (x_1, y_1, T_1) and (x_2, y_2, T_2) , is represented by LineSegment class. This class provides all required functions to:

- Calculate coordinates (X_{R_1}, Y_{B_1}) of point (R_1, B_1) projected on the line segment
- Calculate the distance between given point and a line segment (blue dashed lines).
- Interpolate the temperature represented by given point $(RB_1$ to T_{RB_1})
- Handle the cases where the line segment is not bound to one or both sides. This gives the ability to either:
 - extrapolate the temperature if the coordinates of the point projected on the line are placed outside of the segment (RB_3 to T_{RB_3})

or

- clip the temperature to the maximum defined by line segment (RB_2 to T_{RB_2})

The diagram below visualizes the described functionality of LineSegment class.

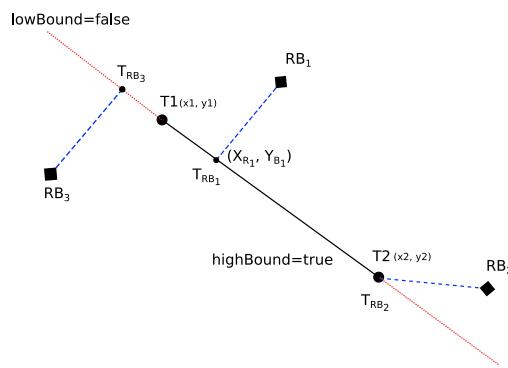


Fig. 8.24: Functionality of LineSegment object

Note: This class is used by TemperatureCorrection and ControlAWB_Planckian classes.

Line Segments This class is a container of a set of LineSegment objects, and abstracts line approximation of the full Planckian Locus curve. The object of LineSegment is a field of TemperatureCorrection and provides the following functionality:

- Return an iterator to LineSegment which is the closest to the given point (*ratioR*, *ratioB*).
- Calculate the (*Xp*, *Yp*) coordinates of point (*ratioR*, *ratioB*) projected on the linear approximation of the curve.
- Estimate the temperature represented by given point (*ratioR*, *ratioB*).

Note: This container object is initialized by TemperatureCorrection object.

AWB High Level Parameters

This Control module replaces the WBC and CCM correction based on temperature computation using the WBS statistics. It interpolates between several corrections defined for a specific temperature at tuning time.

$$\text{Temperature estimation} = (\text{temperature estimation}^{\circ} \\ WB_ESTIMATION_SCALE) \cdot WB_ESTIMATION_SCALE$$

Note: The `WB_*` parameters are not used in `ControlAWB_Planckian` module.

AWB_USE_AWS_CONFIG: Format: bool range {0,1}

Defaults: 0

If 1 then certain AWS settings are not overwritten by sensor runtime parameters in `ControlAWB_Planckian::configureStatistics()`. Usable when debugging.

Applies to AWS_LOG2_R_QEFF, AWS_LOG2_B_QEFF,
AWS_TILE_START_COORDS and AWS_TILE_SIZE.

AWB_MAX_RATIO_DISTANCE: Format: double range [0,1]

Defaults: 0.2

Max distance of tile R and B ratios from centroid point.

Temperature CCM

WB_CORRECTIONS: Format: int range [0,20]

Defaults: 0

Number of corrections to load from the file. Will load the following parameters from 0 to `WB_CORRECTIONS-1`

WB_TEMPERATURE_X: Format: double range [0,100000]

Defaults: 0

E.g. `WB_TEMPERATURE_0`

Temperature (in Kelvin) of that correction.

WB_CCM_X: Format: double[9] range [-3,3]

Defaults: 1 0 0 0 1 0 0 0 1

E.g. WB_CCM_0

3x3 matrix that will replace the value in CCM module (see *Colour Correction Matrix (CCM)* (page 204)).

WB_OFFSETS_X: Format: double[3] range [-32768,32767]

Defaults: 0 0 0

E.g. WB_OFFSETS_0

Offsets for the CCM. Will replace the value loaded in CCM module (see *Colour Correction Matrix (CCM)* (page 204)).

WB_GAINS_X: Format: double[4] range [0.5,8]

Defaults: 1 1 1 1

E.g. WB_GAINS_0

1 value per Bayer channel in R, G, G, B order regardless of the sensor mosaic.

Will replace the value loaded in WBC module (see *White Balance Correction (WBC)* (page 231)).

Note: The LSH_WBCLIP is not replaced

Parameters modified When a ControlAWB (and all it's derivatives) module is enabled in the pipeline, the colour correction configuration parameters (*Colour Correction Matrix (CCM)* (page 204)) are ignored and taken care of by the control module.

The white balance statistics (*Auto White Balance Statistics (AWS)* (page 239) in case of ControlAWB_Planckian and *White Balance Statistics (WBS)* (page 238) for other modules) module configuration is also overridden (refer to *Statistics Configuration* (page 264)).

Statistics Configuration

Note: This paragraph applies only to AWB modules which use *White Balance Statistics (WBS)* (page 238) statistics.

As previously explained, the ControlAWB modules takes care of automatically configuring the white balance statistics so the number of pixels counted for the statistics is close to the desired ratios. Both implementations handle the statistics identically.

The WBS statistics allow defining two different ROIs (region of interests) to extract the statistics from. The ControlAWB configures both of them to use the whole image but only one of them uses dynamic thresholds while the other is kept to keep count of (near) clipped pixels. For the clipping count YHLW_TH is set to 0.9 and RED_MAX_TH, GREEN_MAX_TH and BLUE_MAX_TH are set to 0.45. Those fixed values correspond to a threshold set to a value 90% of the maximum brightness of the pixels received in the WBS module. It is assumed that the pipeline is configured in such a way than the luminances of the pixels are ranged between -64 and 64 (in a s8.x precision). If for whatever reason the configuration of the pipeline changes the fixed thresholds must be adjusted accordingly.

The dynamic thresholds are calculated automatically by the WB control module and set appropriately. By default the target of pixels to be considered in the statistics is a 7.5% and the thresholds vary according to the image statistics to keep close to such ratio.

Correction Configuration

As mentioned in the setup parameters section below, it is possible to redefine the correction configurations applied by the WB control algorithm for different illuminant temperatures. The module will automatically interpolate between the registered configurations with temperature immediately above or below the requested temperature. In case the requested temperature is below the lowest or above the highest defined temperatures no interpolation is carried out and only the closest temperature correction will be selected.

White Balance Flash Filtering (WBFF)

The purpose of Flash Filtering algorithm is to identify single frame illuminated with changed light and to prevent from using statistics for this frame. This applies to any changed light but most common example is flash because this is (almost) guaranteed to last for no more than one frame. Other changed light conditions usually occupy more than one frame. The following figures show how the flash filtering works.



Fig. 8.25: Light change for single frame. Input to AWB.

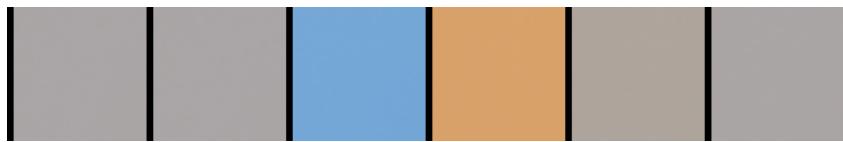


Fig. 8.26: Light change for single frame. Flash filtering disabled.



Fig. 8.27: Light change for single frame. Flash filtering enabled.

WBFF High Level Parameters

WBT_USE_FLASH_FILTERING: Format: bool range {0,1}

Defaults: 0

Flash filtering enabled/disabled.

Code Organization Flash filtering is implemented as function called in ControlAWB_Planckian::update(). Intercepts calculated Red to Green and B blue Green ratios, identifies changed lighting condition, and prevents from using changed ratio statistics if they last for less than 2 frames. The positive effect of flash filtering effect is experienced when only one frame is illuminated with different light. The negative effect of flash filtering is when scene changes because the algorithm causes two frames to be incorrect instead of one. However, this effect is only important when white balance temporal smoothing is disabled. The output from flash filtering is passed to white balance temporal smoothing algorithm.

White Balance Temporal Smoothing (WBTS)

The purpose of White Balance Temporal Smoothing (WBTS or WBT) is to improve user experience when temperature of light in the scene changes rapidly. Because statistics for the picture arrive to driver when this picture is already on his way for display it is not possible to react on this picture. Therefore some colour corrections are inevitable. The role of temporal smoothing is to improve user experience.

The pictures below shows the series of grey background images.

Second picture shows corresponding output images after AWB corrections had been applied. First image is not corrected because there are no statistics yet.



Fig. 8.28: The images without any white balance corrections i.e. the images as received from the sensor (lsh already applied). Input to AWB.

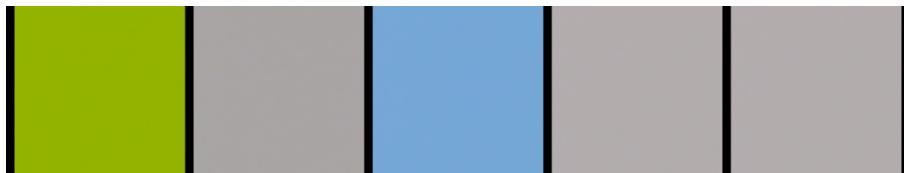


Fig. 8.29: AWB applied.

Third image is incorrect because the statistics applied by hardware to process this input are taken from previous image. There is no way to avoid that. The hardware block uses “incorrect” data. This generates incorrect frame. White balance temporal smoothing task is to make change from incorrect to correct more gradual as this improves user experience.

The WBTS algorithm has two calibration parameters:

- temporal stretch
- smoothing strength

Temporal Stretch The temporal stretch defines the maximum time needed to stabilize output after illumination of the scene changes. Smoothing algorithm operates on gain ratios. Red to

Green (R/G) and Blue to Green (B/G). Note smoothing strength is related in sw to base of the algorithm. The base takes values 1-10. Outside range values are clipped.

$$\text{strength} = 11 - \text{base}$$

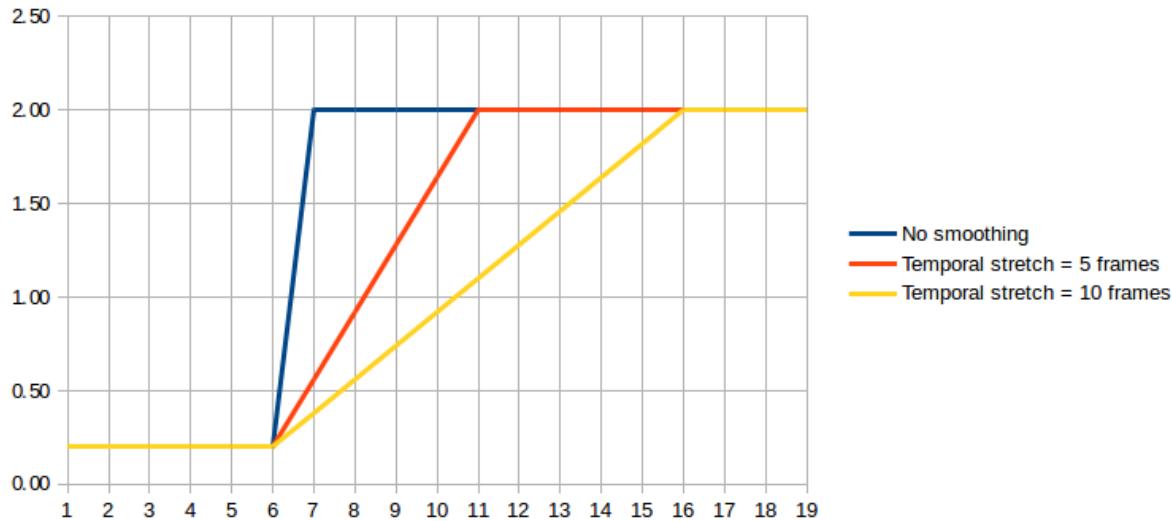


Fig. 8.30: WBTS output depending on temporal stretch. Gain ratio graph.



Fig. 8.31: WBTS temporal stretch input.



Fig. 8.32: WBTS output depending on temporal stretch. 7 frames stretch.

Smoothing Strength The smoothing strength defines how aggressive is the smoothing algorithm. The less aggressive algorithm the closer the output is to not smoothed.

The figure above shows that smoothing strength parameter can affect temporal stretch. Temporal stretch is maintained for smoothing strength values above 9.5.



Fig. 8.33: WBTS output depending on temporal stretch. 13 frames stretch.

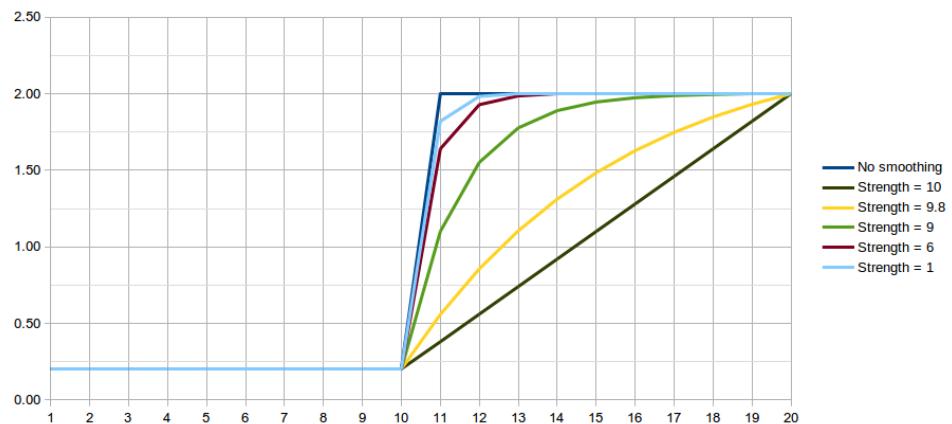


Fig. 8.34: WBTS output depending on smoothing strength. Gain ratio graph.



Fig. 8.35: WBTS input.



Fig. 8.36: WBTS output depending on smoothing strength. Smoothing strength large.



Fig. 8.37: WBTS output depending on smoothing strength. Smoothing strength medium.



Fig. 8.38: WBTS output depending on smoothing strength. Smoothing strength small.

WBTS High Level Parameters

WBT_USE_SMOOTHING: Format: bool range {0,1}

Defaults: 0

Temporal smoothing enabled/disabled.

WBT_TEMPORAL_STRETCH: Format: int range [200,5000]

Defaults: 300

Temporal smoothing enabled/disabled.

WBT_WEIGHT_BASE: Format: float range [1.001,10]

Defaults: 2

Base for temporal smoothing algorithm. Should be regarded as smoothing strength were $\text{smoothing_strength} = 11 - \text{WEIGHT_BASE}$. The stronger the smoothing strength is the more smooth are the changes.

Code Organization White balance temporal smoothing is implemented as function called in ControlAWB_Planckian::update(). Intercepts calculated (and flash filtered) Red to Green and B blue Green ratios, and calculate weighted average of the values for this and previous frames. The number of previous frames taken into account is determined by “temporal stretch” parameter. The weights for paritcular values are detemined by “smoothing strength” paramter. The lower smoothing strenght is the higher weight is applied to latest frame statistics leading to values closer to calculated for the most recent frame.

Automatic focus (AF)

The autofocus algorithm is in charge of automatically selecting the focus distance which provides an (nearly) optimal sharpness in a particular region of the image. The autofocus algorithm relies on the focus sharpness statistics provided by the pipeline (FOS module). The mechanism for focusing consists in a double sweep along different focus ranges: in a first sweep the whole focus range is covered with a larger step. The sweeping range and step is reduced in a second stage for a more precise search in a range of focus distances closer to the positions yielding to higher sharpness.

Current implementation triggers the autofocus loop under demand and then the focus distance is kept fixed until a new autofocus loop is triggered. Alternatively it is possible to send commands to the AF control to manually set a further or closer focus distance. (An alternative way to set up the focus without the AF module would be to program it through the Sensor object within the instantiated Camera class).

Code Organization

The autofocus control is implemented in the ControlAF class which, as the rest of the control modules, inherits from the ControlModule class. The virtual function ControlAF::update() is call after the capture of each new shot and it is in that function where the updates in the configuration must be carried out.

The autofocus loop is implemented as a state machine. It can be triggered or the focus can be set to a further or closer distance. The ControlAF::update() function is regularly called.

After the first iteration, used for initialization, the runAF() method will be called on call to the ControlAF::update() function. The runAF() method implements the actual state machine logic for controlling the autofocus search. Current state of the state machine can be retrieved with the getState() method. The principal states in which the AF algorithm can be are:

- **AF_IDLE**: AF is not actively searching for a focus position.
- **AF_SCANNING**: AF scanning has been triggered and the algorithm is searching for an optimal position.
- **AF_FOCUSED**: An in-focus position has been found.
- **AF_OUT**: Out of focus, no proper in-focus position has been found.

There are a number of *sub states* within the **AF_SCANNING** state which represent different stages in the search. These states are not accessible externally in current implementation (states are **AF_SCAN_STOP**, **AF_SCAN_INIT**, **AF_SCAN_ROUGH**, **AF_SCAN_FINE**, **AF_SCAN_POSITIONING**).

The ControlAF process can be managed externally with the setCommand() method. Such methods receives one of the available commands at any time during execution and behaves accordingly (under some circumstances one of the commands can be ignored). The list of available commands is the following:

- **AF_TRIGGER**: Trigger an autofocus loop, i.e. search for a proper focus position.
- **AF_STOP**: Stop an autofocus loop.
- **AF_FOCUS_CLOSER**: Position the focus in a closer position.
- **AF_FOCUS_FURTHER**: Position the focus in a further position.
- **AF_NONE**: No command (== no action).

The last command received is stored internally and cleared once a call of the runAF() method has been completed (in such call the command should be have dealt with).

The search for a focus position relies in the sharpnessGridMetering() function. Such function computes a single sharpness measure by combining the focus statistics provided by the HW pipeline (FOS) module. Current implementation makes use of the focus sharpness grid which is configured to cover the whole captured image if possible (or the largest possible area). The sharpness value of each tile in the grid is combined with a centred approach using a combinations of the weights defined in the **WEIGHT_7X7_SPREAD** and **WEIGHT_7X7_CENTRAL** matrices. The sharpnessGridMetering() function receives as one of the parameters a value to blend both matrices. Therefore, the ControlAF module uses a central region of the image to find the focus position. For using other regions it should be easy to modify the sharpnessGridMetering() to make it use a different set of weights or the sharpness ROI that can be defined to be used in the FOS module or substitute the call sharpnessGridMetering() by a new function.

Statistics Configuration

The ControlAF relies on the focus sharpness statistics (FOS) gathered from the pipeline. In the current implementation, the statistics are configured to override those defined in the setup file. Both ROI and grid statistics are activated and the grid is configured to cover the whole image while the ROI is configured to use 1/3 width and height central region of the image (note that actually the ROI is not used in current AF implementation).

Lens Shading Grid Control

Main task of lens shading correction control is choice of de-shading grid relevant to detected illuminant temperature reported by the AWB module. Current implementation uses linear mapping between detected temperature and available deshading grids. If deshading matrix for detected temperature doesn't exist the one with temperature closest to the detected temperature is used.

User can decide not to use ControlLSH. In that case de-shading matrices can be loaded by ModuleLSH::loadMatrices() and applied by ModuleLSH::configureMatrix().

Warning: The HW design does not allow the update of all the LSH related registers in one operation. Therefore when using multiple LSH it is required to ensure all matrices have the same configuration and size. This also includes the fact that a LSH matrix must be loaded before Pipeline::startCapture() otherwise multiple registers need to be written.

Code Organization

The lens de-shading control is implemented in the ControlLSH class which, as the rest of the control modules, inherits from the ControlModule class. The virtual function ControlLSH::update() is called after the capture of each new shot and it is in that function where the updates in the configuration must be carried out. From such function the method ControlLSH::programCorrection() is called to update the pipeline de-shading grid. ControlLSH polls ControlAWB for calculated illuminant temperature. Because of that ControlAWB must be registered in ISPC::Camera before ControlLSH. It must be also registered in ControlLSH using ControlLSH::registerCtrlAWB(). De-shading matrices are loaded by ControlLSH during ControlLSH::load() call.

Control LSH High Level Parameters

LSH_CTRL_BITS_DIFF: Format: int range [0,10]

Defaults: 0

How many bits are needed to differentially encode all LSH matrices. Matrix with biggest value determines the common value for the others. If parameter is omitted ControlLSH will calculate it, but it is strongly recommended to define parameter, as auto-calculation slows down the ISP start and doubles number of LSH matrix file reads.

LSH_CTRL_CORRECTIONS: Format: int range [0,200]

Defaults: 0

How many correction matrices are available. Will determine how many of LSH_CTRL_TEMPERATURE_X, LSH_CTRL_FILE_X and LSH_CTRL_SCALE_WB_X are loaded.

LSH_CTRL_TEMPERATURE_X: Format: unsigned range [0,100000]

Defaults: 0

Illuminant temperature correlated LSH group X.

E.g. LSH_CTRL_TEMPERATURE_0 goes with LSH_CTRL_FILE_0 and LSH_CTRL_SCALE_WB_0.

LSH_CTRL_FILE_X: Format: string**Defaults:** <matrix file>

File with deshading matrix correlated to LSH group X.

E.g. LSH_CTRL_FILE_0 goes with LSH_CTRL_TEMPERATURE_0 and LSH_CTRL_SCALE_WB_0.

LSH_CTRL_SCALE_WB_X: Format: double range [0,100]**Defaults:** 1

Factor to scale the WB gains when using this matrix to compensate for the scaling to cope with HW limitations for LSH group X.

E.g. LSH_CTRL_SCALE_WB_0 goes with LSH_CTRL_TEMPERATURE_0 and LSH_CTRL_FILE_0.

Automatic Tone Mapping Control (TNMC)

The automatic tone mapping control is in charge of:

- Dynamically generate a global mapping curve based on previous captures' statistics.
- Control the strength of the tone mapped module (TNM) so tone mapping strength is reduced when the sensor is setup in a way which may produce noisy images (basically when Sensor's gain value is high).

The global curve generation is based in the histogram generated in the pipeline corresponding to a previously captured shot. The algorithm generates a mapping curve from such histogram and programs it to be applied in subsequent captures. The mapping curve is updated by merging the new curve with the previous one, in order to provide smooth transitions. There are a number of parameters to guide the curve generation which are basically analogous to the ones used in the tone mapping curve generation standalone tool. See *Global curve generation* (page 272) below for more details.

There is another aspect of the tone mapping control which is to modify the strength of the tone mapping applied according to the sensor settings. For more details see *Dynamic Tone Mapping Strength* (page 273).

Global curve generation

The curve generation is based on the accumulation of the histogram obtained from a previous shot. This is a common technique for histogram equalization which increases the contrast in the output image by maximizing the usage of the output range and ‘assigning’ a larger range to more frequent pixel values in the original image.

However the histogram accumulation technique can easily produce undesired effects and artefacts in the output image so it is necessary to post-process the input image histogram to avoid such effects. The controls in the global tone mapping generation drive such process. The curve generation process is as follows:

- Histogram is clipped between the min and max histogram clipping values. To clip high histogram values helps to prevent too steep mapping curves, which might produce exaggerated contrast enhancement in the output. The min clip value assures a minimum value

in all the histogram bins to avoid that too large ranges of the input image values are ‘removed’ to allow the expansion of other more frequent value ranges.

- Histogram is smoothed.
- Histogram is tempered, flattening it, so the closer it gets to a flat histogram (a histogram with the same value in all bins) the closer the output mapping curve will get to an identity function (which would make the output = input).
- Histogram normalization.
- Mapping curve generation.

Dynamic Tone Mapping Strength

The Automatic Tone mapping control module includes a mechanism to prevent the tone mapping module to apply strong tone mapping when the capture settings are likely to produce noisy images. There is an `adaptiveStrength` attribute which is only maximum (1.0) when the ISO value set up in the sensor is 100 or below. For higher ISO values the `adaptiveStrength` value is proportionally reduced:

$$\text{adaptiveStrength} = \frac{100}{\text{ISO}}$$

$$\text{adaptiveStrength} = \frac{1.0}{\text{sensor gain}}$$

The `adaptiveStrength` value will control both the local tone mapping and global tone mapping. A 1.0 value will represent full strength while a value of 0.0 implies no local tone mapping and identity global mapping curve.

$$\text{localStrength} = \text{localStrength} \cdot \text{adaptiveStrength}$$

$$\text{outCurve}[i] = \text{globalCurve}[i + 1] \cdot \left(\frac{i + 1}{\text{TNMC_N_CURVE}} \right) \cdot (1.0 - \text{adaptiveStrength})$$

Note: `TNMC_N_CURVE` is the number of elements in the curve (65).

Code Organization

The tone mapping automatic control is implemented in the `ControlTNM` class which, as the rest of the control modules, inherits from the `ControlModule` class. The entry point is the `ControlTNM::update()` function. Under normal running conditions, the `ControlTNM::update()` function for every control module registered in the pipeline is called once for every captured shot.

The `ControlTNM::update()` function in `ControlTNM` first calculates the value for the `adaptiveStrength` and then loads the histogram from the metadata in the call to the `loadHistogram()` function. After that the global tone mapping curve is generated in the `generateMappingCurve()` function (as described in section above) which uses the following set of parameters:

- `histMin, histMax`: For histogram clipping use the `TNMC_HIST_CLIP_MIN` and `TNMC_HIST_CLIP_MAX` parameters.
- `smoothing`: Smoothing histogram value configurable using `TNMC_SMOOTHING` parameter.

- **tempering:** tempering histogram value configurable using TNMC_TEMPERING parameter.
- **updateSpeed:** Parameter to control the speed of the global tone mapping update. Use TNMC_UPDATE_SPEED parameter to control the value.

The local tone mapping parameters are not modified by the tone mapping control module with the exception of the local tone mapping strength value, which is used to enable/disable and dynamically control local tone mapping strength. The rest of the local tone mapping settings are left with whatever values they previously had. To control the local tone mapping the function enableLocalTNM() as well as the TNMC_LOCALSTRENGTH parameter can be used to control the characteristics of the local tone mapping.

Regarding the automatic tone mapping strength control it can be enabled or disabled with the enableAdaptiveTNM() function.

TNMC High Level Parameters

This Control Module computes the TNM global curve and the strength of the local curve based on the global HIS statistics and the strength of the sensor's gain.

More information about the algorithm is available in the *Automatic Tone Mapping Control (TNMC)* (page 272) section.

TNMC Generic parameters

TNMC_UPDATE_SPEED: Format: float range [0,1]

Defaults: 0.5

Update speed for application of the newly generated global curves. (0=no update, 1.0=instant update).

TNMC_ADAPTIVE: Format: bool range {0,1}

Defaults: 0

Uses 1/(sensor gain) information to limit the strength of the tone mapping (both local and global) when sensor has high gains.

TNMC Global Curve parameters

TNMC_HIST_CLIP_MIN: Format: float range [0,1]

Defaults: 0.035

Histogram clipping min value for global tone mapping curve generation.

TNMC_HIST_CLIP_MAX: Format: float range [0,1]

Defaults: 0.25

Histogram clipping maximum value for global tone mapping curve generation.

TNMC_SMOOTHING: Format: float range [0,1]

Defaults: 0.4

Smoothing value from 0.0 (no smoothing) to 1.0 (max smoothing) for the global mapping curve generation.

TNMC_TEMPERING: Format: float range [0,1]

Defaults: 0.2

Tempering value between 0.0 (no tempering) to 1.0 (max tempering) for the global mapping curve generation. More tempering produces more gentle mapping curves.

TNMC Local Tone Mapping parameters

TNMC_LOCAL: Format: bool range {0,1}

Defaults: 0

Enable the local tone mapping.

TNMC_LOCALSTRENGTH: Format: float range [0,1]

Defaults: 0

Replaces the local weight parameter of the TNM Module (see *Tone Mapper (TNM)* (page 226)) (affected by the sensor's gain too).

TNMC Parameters modified The ControlTNM module loading those parameters will use them to modify only: TNM_CURVE and TNM_WEIGHT_LOCAL. The other parameters described in *Tone Mapper (TNM)* (page 226) should not be altered.

Warning: It is important to point out that the tone mapper range defined by the TNM_IN_Y parameter must be properly aligned with the histogram generated in the HIS statistics module. Otherwise the global curve generated will not match properly the image characteristics.

Statistics Configuration

The Auto TNM module also has the ability to configure the HIS it uses if another module did not configure it already (the choice has to be done when instantiating the control algorithms).

The ControlTNM module relies in the HIS statistics extraction for proper behaviour. The configuration of the HIS statistics is exactly the same as required by the AE module (see *Statistics Configuration* (page 246)). It is expected that the histogram range goes from the black level to the white level in the pipeline. By default those values would range between -64 and 64 (assuming a s8.x precision pipeline). The TNM range must be configured in such a way that it covers exactly the same range as the extracted histogram.

Light Based Control (LBC)

The Light Based Control (LBC) is a control module in charge of changing settings of the pipeline according to the amount of light that the sensor is receiving. It is possible to define an arbitrary number of configurations corresponding to different light levels and define the values for a pre-defined set of pipeline parameters for each one of the configuration. The module is in charge of estimating the amount of light and interpolating configurations from the defined set of available ones. Currently the module supports control for the following set of parameters:

- **Sharpness:** Making use of the SHA_STRENGTH setup parameter.

- **Brightness:** Making use of the R2Y_BRIGHTNESS setup parameter.
- **Contrast:** Making use of the R2Y_CONTRAST setup parameter.
- **Saturation:** Making use of the R2Y_SATURATION setup parameter.

It must be noted that the LBC relies on the HIS statistics and those are gathered **after** the R2Y module. Therefore changes in the brightness and contrast in particular can affect the statistics and hence the light level estimation in the LBC module. For this reason it is recommended to be careful if using the LBC module to control the brightness and saturation values as this could yield to unstable configurations, oscillations, etc.

Code Organization

The Light Based Control is implemented in the ControlLBC class which, as the rest of the control modules, inherits from the ControlModule class. The virtual function ControlLBC::update() is called after the capture of each new shot and it is in that function where the updates in the configuration must be carried out.

The algorithm is implemented in the following stages:

- First the light level is estimated by calling the calculateBrightness() method to calculate the captured image average brightness and then applying the following estimation formula:

$$\text{lightLevel} = \text{exposure} \cdot \frac{\text{gain}}{\text{averageBrightness}}$$

- The configuration to be applied is interpolated using the auxiliary class LBCConfiguration (see description in section below).
- The interpolated configuration is applied in the call to the ControlLBC::programCorrection() method.

Auxiliary classes

Within the ControlLBC.h header file it is included the definition for the LBCConfiguration class. The LBCConfiguration class is used for storing a set of configuration parameters and associated light level. It also defines operators and methods in order to ease the interpolation between couples of configurations. The ControlLBC class maintains a collection of LBCConfiguration instances as defined in the setup parameters to interpolate from. The set of interpolated configurations is retrieved from loaded from a ParameterList as any other setup parameter.

LBC High Level Parameters

This Control Module modifies the parameters of the R2Y and SHA modules according to computed light level from global HIS statistics. It loads several configurations defined at tuning time.

More information about the module is available in *Light Based Control (LBC)* (page 275).

LBC Generic parameters**LBC_UPDATE_SPEED:** Format: double range [0,1]**Defaults:** 0.1

Speed to update the light metering so we have a smooth transition between configurations.

LBC Multi high-level based values**LBC_CONFIGURATIONS:** Format: int range [0,20]**Defaults:** 0

Number of Light Level Configurations to load from the file.

Will load all the following parameters from 0 to LBC_CONFIGURATIONS-1.

LBC_LIGHT_LEVEL_X: Format: double range [0,1e+08]**Defaults:** 0

E.g. LBC_LIGHT_LEVEL_0

Light level of that particular configuration.

LBC_BRIGHTNESS_X: Format: double range [-0.5,0.5]**Defaults:** 0

E.g. LBC_BRIGHTNESS_0

Brightness to replace the one loaded in R2Y module (see *RGB to YUV (R2Y)* (page 224)).**LBC_CONTRAST_X:** Format: double range [0,2]**Defaults:** 1

E.g. LBC_CONTRAST_0

Contrast to replace the one loaded in R2Y module (see *RGB to YUV (R2Y)* (page 224)).**LBC_SATURATION_X:** Format: double range [0.1,10]**Defaults:** 1

E.g. LBC_SATURATION_0

Saturation to replace the one loaded in R2Y module (see *RGB to YUV (R2Y)* (page 224)).**LBC_SHARPNESS_X:** Format: double range [0,1]**Defaults:** 0.4

E.g. LBC_SHARPNESS_0

Sharpness to replace the one loaded in SHA module (see *Sharpening (SHA)* (page 225)).

Parameters modified The corresponding *Sharpening (SHA)* (page 225) and *RGB to YUV (R2Y)* (page 224) parameters are modified.

Statistics Configuration

The LBC module also has the ability to configure the HIS it uses if another module did not configure it already (the choice has to be done when instanciating the control algorithms).

The LBC control module makes use of the histogram statistics (HIS) which must be configured as described in the AE module (see *Statistics Configuration* (page 246)).

Denoiser Control

The denoiser control is in charge of dynamically configuring the denoiser module (DNS) according to the capture settings (and potentially other criteria like light levels, etc.). Current implementation is a simple approach where only the ISO_GAIN parameter of the DNS is configured with the value used for the sensor capture configuration. The denoiser strength is not modified in that control module.

Code Organization

The denoiser control is implemented in the ControlDNS class which, as the rest of the control modules, inherits from the ControlModule class. The virtual function ControlDNS::update() is called after the capture of each new shot and it is in that function where the updates in the configuration must be carried out. From such function the method ControlDNS::programCorrection() is called to update the pipeline configuration.

DNSC High Level Parameters

The denoiser control doesn't require additional setup parameters. If activated the DNS_ISO_GAIN corresponding to the DNS module is ignored and the control module programs the corresponding values in the pipeline.

8.5 Sensor Class

The Sensor class encapsulates the Sensor API and provides functionality for sensor initialisation, starting as well as basic controls (such as gain, exposure, focus distance, etc.). There is a Sensor class instance as part of the Camera object which gives a pointer to its Pipeline object. The registered SetupModule and ControlModule instances have access to the Sensor object through their Pipeline or Control owner.

The control loop algorithms will very commonly require access to the sensor (for example the auto exposure algorithms) but also the Pipeline setup modules as some aspect of the configuration may require knowledge about the sensor characteristics such as size, frame rate, etc.

The Sensor class provides a number of public methods for setting/getting the gain and exposure settings as well as for querying for the maximum or minimum gain and exposure. In addition there are a number of public members in the class to allow access to the sensor width, height, sensor format, etc.

See the code documentation for a complete listing and details of the Sensor class methods and members.

An overloaded version of the Sensor class is available to cope with the internal and external data-generator: DGSensor. This object allows the loading of FLX images as input through the Sensor API interface. It also makes it easier to use a data-generator by triggering the shot on the data-generator when triggering a frame capture.

8.5.1 Sensor High-level Parameters

These parameters are loaded and saved using a Sensor object and several modules use this information in their setup process by querying the sensor object directly.

SENSOR_EXPOSURE_MS: Format: double range [0,5000]

Defaults: 35

Exposure used for the capture in miliseconds.

Only saved as information.

When using a data-generator sensor this parameter is ignored.

SENSOR_GAIN: Format: double range [0,128]

Defaults: 1

Gain applied by the sensor (analogue & digital).

Only saved as information.

When using a data-generator sensor this parameter is loaded to allow the correct configuration of the denoiser module.

SENSOR_READ_NOISE: Format: double range [0,100]

Defaults: 0

Standard deviation of noise when reading pixel value off a sensor in electrons.

Only saved as information is gathered from the sensor driver directly.

When using a data-generator sensor this parameter is loaded to allow the correct configuration of the denoiser module.

SENSOR_BITDEPTH: Format: unsigned range [8,16]

Defaults: 10

Bitdepth of data from the sensor.

Only saved as information is gathered from the sensor driver directly.

When using a data-generator this value comes from the given input file bitdepth.

SENSOR_WELL_DEPTH: Format: unsigned range [0,65535]

Defaults: 5000

Maximum number of electrons a sensor pixel can collect before clipping.

Only saved as information is gathered from the sensor driver directly.

When using a data-generator sensor this parameter is loaded to allow the correct configuration of the denoiser module.

SENSOR_FRAME_RATE: Format: double range [1,255]

Defaults: 30

Expected frame-rate of the sensor configuration in frame per seconds.

Only saved as information.

When using a data-generator this parameter is loaded to allow the correct configuration of the flicker detection module.

SENSOR_ACTIVE_SIZE: Format: unsigned[2] range [0,16384]

Defaults: 16384 16384

Active size of the sensor configuration (width and height).

Only saved as information.

When using a data-generator this value comes from the given input file resolution.

SENSOR_VTOT: Format: unsigned range [0,16383]

Defaults: 525

Total number of lines the sensor is capturing (including blanking).

Only saved as information.

When using a data-generator this values comes from the given input file resolution and the configured vertical blanking.

8.6 Several Camera sharing a Sensor

This is also known as “multi-context” processing. The underlying idea is to use a single Sensor object but have several Camera objects to handle more output formats than a single Camera can produce or use different configuration for the outputs.

8.6.1 Creation of the Camera

When calling the Camera constructor it is possible to specify if the Camera object owns the Sensor. In the case of sharing a sensor **only ONE** Camera object should own the sensor. The owner will be responsible for the Sensor object; it will configure the Sensor, start the Sensor, stop the Sensor etc. The sharer will always assume the Sensor is in the correct state. The sharer will however use the Sensor to retrieve information (such as size and current gain).

Note: Therefore it is the responsibility of the user to ensure the Camera that owns the sensor is used to manage the sensor before the sharer tries to access it.

The creation should therefore look like:

```
ISPC::Camera *owner = new ISPC::Camera(0, sensorID, sensorMode);
ISPC::Sensor *sensor = owner->getSensor();
ISPC::Camera *sharer = new ISPC::Camera(1, sensor);

ISPC::CameraFactory::populateCameraFromHWVersion(owner, sensor);
ISPC::CameraFactory::populateCameraFromHWVersion(sharer, sensor);
```

Note: The 1st parameter is the HW context number and it cannot be the same for the 2 Cameras or they will not be able to run at the same time. The maximum number of context for the HW is available in `CI_CONNECTION::sHWInfo::config_ui8NContexts`.

8.6.2 Control Modules ownership

When sharing a Sensor thoughts have to be made on which Camera will own the control modules. The choice has to be made with the following information in mind:

- If the control module tries to modify the Sensor's parameters (e.g. gain) **ONE AND ONLY ONE** should ever have access to the sensor.
- A control module only tries to modify the statistics it uses on the Pipeline that owns it.
- Several control modules can try to access the same statistics and may have to be configured to do so!
- A control module can modify several Pipelines objects with their results.
- Several control modules of the same type can coexists **if and only if** they are in different Camera objects.

Example (ControlAE and ControlTNM)

The Auto Exposure control module accesses the Sensor object. **ONE AND ONLY ONE** can exist for a Sensor. It should be added to the owner as it handles all there is to handle about the Sensor.

The Tone Mapper control module does not access the Sensor object. Therefore it is possible to have different objects in the `owner` and `sharer` (to apply different TNM curves). However the ControlTNM uses the histogram statistics (HIS), and so does the ControlAE.

We have two possible choices:

- create several ControlTNM objects to allow different curves or
- create a single ControlTNM object and the same curve will be applied to both Pipelines

Multiple ControlTNM

We have several ControlTNM objects therefore the `sharer`'s ControlTNM has to be configured to enable the histogram statistics (because the ControlAE enables the HIS in the owner).

```
// owner control modules
ISPC::ControlAE *pAE = new ISPC::ControlAE();
ISPC::ControlTNM *pTNM_owner = new ISPC::ControlTNM();

owner->registerControlModule(pAE);
owner->registerControlModule(pTNM_owner);

// sharer control modules
ISPC::ControlTNM *pTNM_sharer = new ISPC::ControlTNM();
```

```
// enable configuration of HIS because AE does not exists on this pipeline
pTNM_sharer->setAllowHISConfig(true);
sharer->registerControlModule(pTNM_sharer);
```

Single ControlTNM

In this case a single ControlTNM object can be added to either the owner or the sharer object. We choose to add it to the owner so that it will be the only Camera with control modules. If it was added to the sharer then the HIS should be configured as in the Multiple ControlTNM example.

```
ISPC::ControlAE *pAE = new ISPC::ControlAE();
ISPC::ControlTNM *pTNM = new ISPC::ControlTNM();

owner->registerControlModule(pAE);
owner->registerControlModule(pTNM);

// allow the control module to modify more than 1 Pipeline
pTNM->addPipeline(sharer->getPipeline());
```

8.6.3 Camera and Sensor management

It is important to understand that the owner object handles all the sensor operations (see *Sensor ownership* (page 198)) therefore it has to be started last and stopped first. Triggering frame can be done in any order.

```
ISPC::ParameterList ownerSetup; // assumes it is ready to be loaded
ISPC::ParameterLsit sharerSetup; // assumes it is ready to be loaded

owner->load(ownerSetup);
sharer->load(sharerSetup);

owner->setupModules();
sharer->setupModules();

owner->program();
sharer->program();

sharer->startCapture(); // does not affect the sensor - only starts ISP pipeline
owner->startCapture(); // start ISP pipeline and then starts sensor

loop() {
    Shot ownerShot;
    Shot sharerShot;

    owner->enqueueShot();
    sharer->enqueueShot();

    owner->acquireShot(ownerShot);
    sharer->acquireShot(ownerShot);
}

owner->stop(); // stops the sensor and then the ISP pipeline
sharer->stop(); // stops the ISP pipeline
```

In the case of triggering 2 frames for the same Sensor's image the low level driver does not ensure that both of them will be from the same capture. Therefore it is possible that a new frame arrives in between the `owner->enqueueShot()` and `sharer->enqueueShot()`. However once several shots have been enqueued they should be removed from each pipeline waiting list at the same rate.

Note: The same number of frames does not have to be triggered on both sides when running against real HW. However when running against the CSIM the same number of frames **HAVE TO** be triggered on both sides (the simulator does not handle a frame start signal on an empty linked list).

Note: Timestamps from the `Shot::_src:metadata<Metadata>::_src:timestamps<MC_STATS_TIMESTAMP>` can be used to ensure the frames were captured at the same time. But the values may be different as the TS values are when the ISP started processing the frames (and there is only 1 processing pipe in the ISP HW). However knowing the frame rate of the sensor it should be possible to determine if 2 timestamps, even if different, are likely to be the same sensor frame.

8.7 High level parameters

Many of the classes implemented in the ISPC library make use of the ParameterList class to load parameters generated externally. All the ModuleBase instances must implement a `load()` function that receives a ParameterList and makes use of it at its own convenience (there is no obligation for any given module to actually make use of external parameters).

A ParameterList is just a collection of Parameter objects each one being composed by a tag and a set of associated values. Internally both the tags and values are stored as strings. It is very common in our test applications for a ParameterList object to have been generated from an external configuration file but the source of the tags/values could be another program, a socket, etc. The ParameterList allows adding and retrieving tags and associated values in a flexible way and also ‘interpret’ the values associated to a tag as different data types: integer, float, boolean or string. Once a populated ParameterList has been received it is simple to retrieve values using one of the “get parameter” functions. Each SetupModule and ControlModule contains a set of public ParamDef object which lists the parameter they will try to access in the parameter list (but they may also try to access parameters from other modules!).

The get parameter functions, receive as parameters, the tag we are looking for (as a string) and the index of the value we want to retrieve. There are also versions of the same functions to specify minimum, maximum and default values if the parameters are not found and functions to check if a tag is defined or not, etc. (see the doxygen documentation for more details).

8.7.1 Accessing parameters

The following example shows how to check if a parameter exists in the ParameterList and to retrieve its first value (index 0) if so:

```
if(parameters.exists("WB_ESTIMATION_SCALE"))
    estimationScale = parameters.getParameterFloat("WB_ESTIMATION_SCALE", 0);
```

The following example depicts how to retrieve the second value associated to a given tag (index 1) and also defining minimum, maximum and default values:

```
aInY[1] = parameters.getParameterFloat("TNM_IN_Y", 1, -127, 127, 127);
```

It is also possible to retrieve parameters using directly the ParamDef objects:

```
estimationScale = parameters.getParameter(ISPC::ControlAWB::AWB_ESTIMATION_SCALE);
```

8.7.2 Parameter File Parsing

The ParameterFileParser has been implemented for being able to parse generic high level parameters files in which a list of tags (one per line) followed by an arbitrary number of associated values can be specified. The format is compatible with the format for Felix setup parameter files used by CSIM and the ParameterFileParser is used for its parsing and conversion into a ParameterList. Its usage is straightforward as in this example:

```
ParameterList parameters = ISPC::ParameterFileParser::parseFile(filename);
```

Once a file has been parsed into a ParameterList it can be easily used to set up the ISPC modules (see doxygen documentation for more details about the ParameterList and ParameterFileParser functionalities).

8.7.3 Setup Parameters (a.k.a. Felix Setup Args)

The Modules are the input to the HW Pipeline. The high level values are transformed to register values within the drivers MC functions.

The defaults, minimums and maximums can be generated using the *ISP Control test: ISPC_test* (page 61) application.

The C Simulator delivered as a HW preview contains an internal driver, developed with the aim of testing the HW. The High level parameters used in the simulator are mostly compatible with the one in the driver. However when differences exists this section will try to highlight them in the modules sections.

Life-cycle of the information

The information loaded as setup parameters usually comes into a text file and end up as a register value. The transformation to the register value occurs in several stages within the driver libraries.

This cycle is repeated every time the high level parameters are changed (i.e. potentially every frame).

Loading with ISPC

The loading can be done from a text file (ISPC::ParameterFileParser) but it is possible to implement other ways that populate an ISPC::ParameterList object.

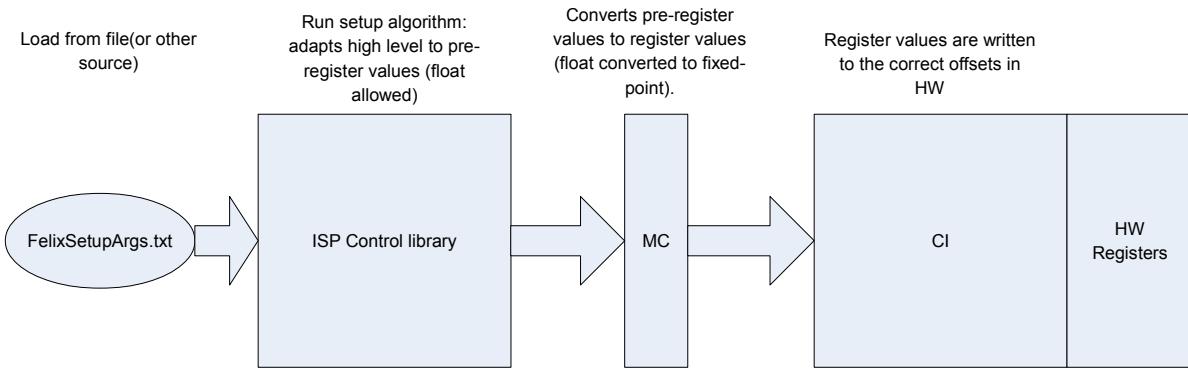


Fig. 8.39: Life-cycle of the parameters

Run setup algorithms

The setup algorithms convert the setup modules values into pre-register values. These values can contain floating point information.

The Algorithms are run just before that stage (e.g. auto white balance will try to modify the CCM module before it is converted to MC values).

This is usually done when setup functions are run for a module (e.g. Camera::setupModules()).

Convert pre-register to registers

This operation is done in 2 steps.

The first step is to convert the pre-register values to register values. That means converting the floating point values to the fixed point the HW is expecting. At this stage the values that can be computed from others are also done (e.g. the IIF buffer threshold). This is all done in user-space MC library and transferred to kernel-space through the CI layer.

The second step is to write the register values into the correct register offsets. This is done by the CI layer in the kernel module.

Both steps are done when Camera::program() is called.

8.8 Performance measurement

The ISPC library contains an auxiliary function to log performance measurements. This option is disabled by default and the library should be compiled with the appropriate CMake flags (see *DDK options* (page 23)).

The header `ispc/PerfTime.h` allows the creation of a performance stack. An element should be registered to the performance stack using one of the `ISPC::LOG_Perf_Register()` function. Elements can be added to the stack as an input using `ISPC::LOG_Perf_In()` (i.e. entry into a function) or as output using `ISPC::LOG_Perf_Out()` (i.e. exit of a function). The average time spent in the function, the last time and the number of calls will be monitored and printed when using the `ISPC::LOG_Perf_Summary()`.

The macros `LOG_PERF_IN()` and `LOG_PERF_OUT()` can also be used to help with the gathering of functions/file/lines.

When a module is registered as verbose it prints the time when IN/OUT was called.

Chapter 9

Android Camera HAL

The section following contains detailed information about architecture, configuration and possible extensions of Android HAL library for V2500 image processors.

As the mobile market was the primary target of V2500 ISP processors, the Hardware Abstraction Layer (HAL) library has been developed to enable support for V2500 IP in Android versions 4.4, 5.0 and later (code-named Kitkat and Lollipop respectively).

The primary responsibility of camera HAL is to expose the ISP functionality via well documented `android.hardware.camera` and `android.hardware.camera2` interfaces to the user applications. In details, to be able to support more advanced camera applications, Android Camera HAL library shall support Camera HAL interface version 3.x defined in `camera3.h` header of Android sources. This C interface is used by the ‘media’ android service (implemented in mediaserver daemon) to communicate directly with HAL library. More on this architecture can be read on Android camera stack online documentation page ¹.

The functionality of camera HAL library depends on the current version of ISPC+CI software pipeline stack. Therefore the Android build scripts provided will compile most of the needed dependencies automatically. The exception of this is the kernel Felix.ko module, which has to be built separately. For full building procedure, please refer to *Android Build Instructions* (page 325).

Note: V2500 may be referred as `Felix` in this section (its internal name).

Document structure

The chapter *Context of Camera HAL in media framework* (page 288) depicts the native environment where Camera HAL shared library is being used. The sub-chapter *Main use cases* (page 288) lists the main *scenarii* and the module may hit during runtime of various camera applications. The chapter *Library architecture* (page 289) describes the high level structure of the library. Specifically, *Class hierarchy* (page 289) and *Class responsibilities* (page 291) describe the class dependencies and responsibility areas. Later on, the structure of the source code and build system variables has been described in *Structure of source code* (page 302) and *Build system* (page 304) respectively. The next chapter, *Implementation of Camera HAL v3.x* (page 305), details Camera HAL internals, such as description of capture request processing pipeline, specifics of jpeg processing and level of support for request metadata. Finally, the testing methodologies can be referred in *Testing Camera HAL* (page 322).

¹ <https://source.android.com/devices/camera/index.html>

9.1 Context of Camera HAL in media framework

Before reading the Camera HAL documentation, the reader is encouraged to become familiar with the architecture of Android media framework architecture, especially in part directly related with camera. The reference documentation is available under ¹.

9.1.1 Main use cases

The high level modes the camera can operate in a common Android based device are defined by `enum camera3_request_template_t` defined in `camera3.h` header. These are:

- `CAMERA3_TEMPLATE_PREVIEW` Standard camera preview operation with 3A on auto.
- `CAMERA3_TEMPLATE_STILL_CAPTURE` Standard camera high-quality still capture with 3A and flash on auto.
- `CAMERA3_TEMPLATE_VIDEO_RECORD` Standard video recording plus preview with 3A on auto, torch off.
- `CAMERA3_TEMPLATE_VIDEO_SNAPSHOT` High-quality still capture while recording video. Application will include preview, video record, and full-resolution YUV or JPEG streams in request. Must not cause stuttering on video stream. 3A on auto.
- `CAMERA3_TEMPLATE_ZERO_SHUTTER_LAG` Zero-shutter-lag mode. Application will request preview and full-resolution data for each frame, and reprocess it to JPEG when a still image is requested by user. Settings should provide highest-quality full-resolution images without compromising preview frame rate. 3A on auto.
- `CAMERA3_TEMPLATE_MANUAL` A basic template for direct application control of capture parameters. All automatic control is disabled (auto-exposure, auto-white balance, auto-focus), and post-processing parameters are set to preview quality. The manual capture parameters (exposure, sensitivity, etc.) are set to reasonable defaults, but should be overridden by the application depending on the intended use case.

These operation modes define different configurations HAL must be able to handle. Each mode can be translated into a set of output configurations the framework can request from the Camera HAL when it is in active state.

Note: Each output stream requested can be configured to different pixel type and resolution. In practice, output buffers which are used in encoder, as application callbacks or Zero Shutter Lag purpose, are always internally allocated as YUV color space. Those used as display preview, are allocated as `IMPLEMENTATION_DEFINED` type.

On PC+FPGA based development platforms, preview buffers are internally allocated in gralloc as RGB32 due to no support for GPU hardware on such setups (sw fallback pixelflinger uses RGB). For GPU enabled platforms, YUV pixels may be chosen as the format for preview surfaces, so gralloc should internally allocate these buffers as YUV.

Below is the list of common low level stream combinations which can be requested as an effect of choosing one of camera templates:

	Output streams				Input stream	
	YUV	RGB	JPEG	RAW	YUV	RAW
preview video recording application callback RAW (optional)	0-3	0-1	N/A	0-1	N/A	N/A
preview video recording non ZSL still capture (JPEG) application callback RAW (optional)	0-3	0-1	1	0-1	N/A	N/A
ZSL YUV to JPEG reprocessing	N/A	N/A	1	N/A	1	N/A
ZSL RAW to JPEG reprocessing	N/A	N/A	1	N/A	N/A	1
ZSL RAW to YUV reprocessing	1	N/A	N/A	N/A	N/A	1

This table defines the requirements imposed on the ISP hardware, in order of minimum software processing needed for regular camera operation. More on this in *Hardware requirements* (page 305).

Note: Current implementation of HAL does not support RAW captures.

9.1.2 Lifetime of Camera HAL

The camera library is loaded by the multimedia framework (`mediaserver`) directly after initialization of `media` service.

Note: In debugging process, the ability to manual control `mediaserver` state proves to be very useful. For this purpose, the `media` service can be manually stopped and started using the following adb shell commands:

```
$ adb shell stop media
$ adb shell start media
```

Then, Camera HAL interface is used to instantiate all existing cameras, read capabilities, and execute the sequence of calls which configure and enqueue capture requests in ISP processor.

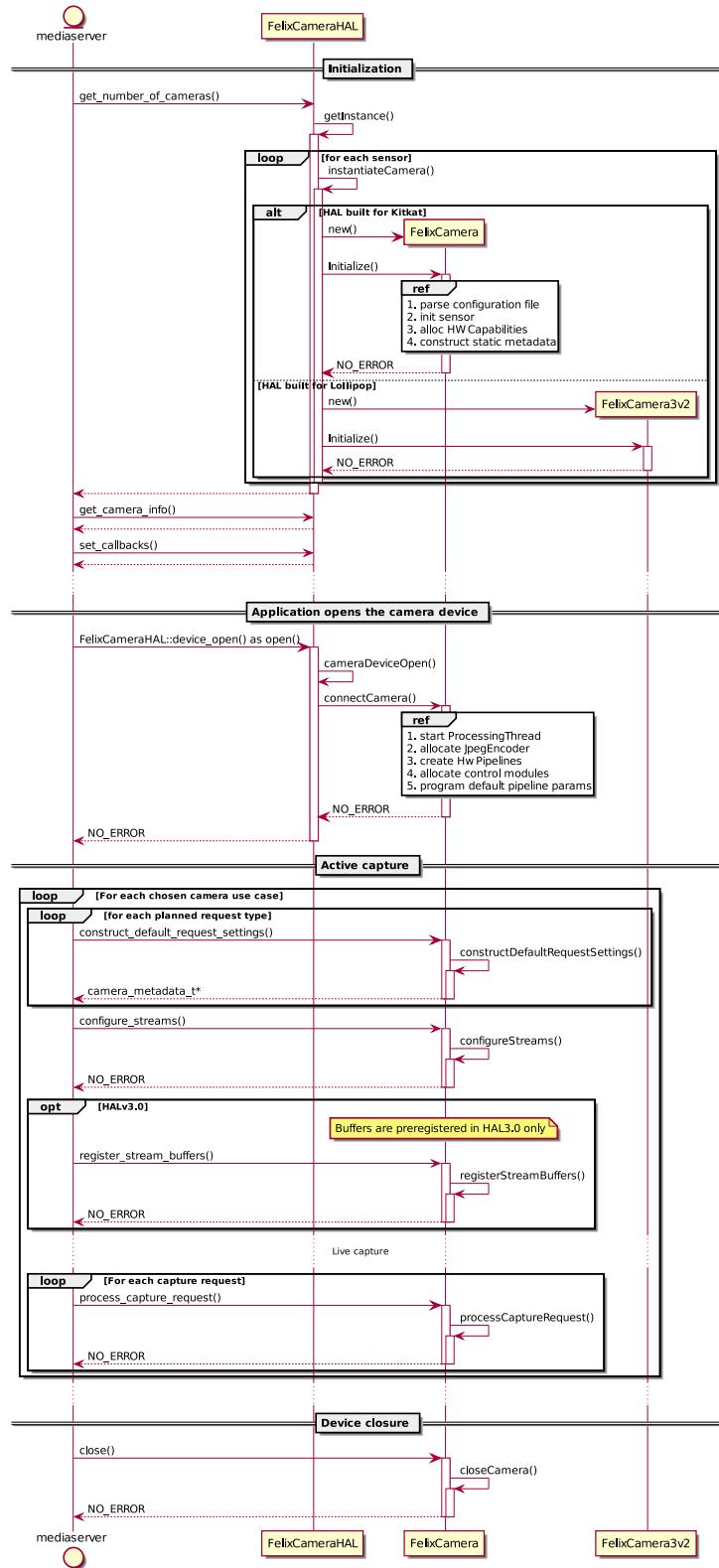
The following diagram depicts the common lifetime of Camera HAL in Android. The exact sequence of calls to HAL v3 interface has been visualized for better overview of the library interface flow.

9.2 Library architecture

9.2.1 Class hierarchy

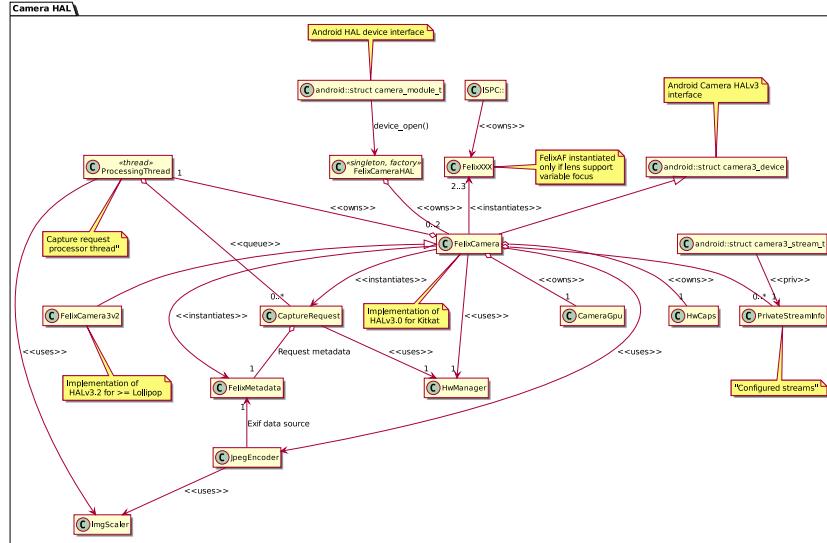
The HAL library is structured into following main functional parts:

- Android Camera HAL interface modules
- Asynchronous capture request processing modules
- Hardware abstraction (capabilities and ISPC control)



- AAA handling modules
 - Helper classes abstracting image processing and JPEG compression, buffer management

The Camera HAL diagram shows the high level FelixCamera HAL hierarchy.



The main owner class is `FelixCamera`, which represents an instance of a camera sensor connected to ISP pipeline, while exposing HAL 3.x interface to Android camera framework. The instance ‘owns’ and ‘uses’ `FelixProcessing` thread for the purpose of buffering capture requests, asynchronous request processing and sending back capture results to the framework.

9.2.2 Class responsibilities

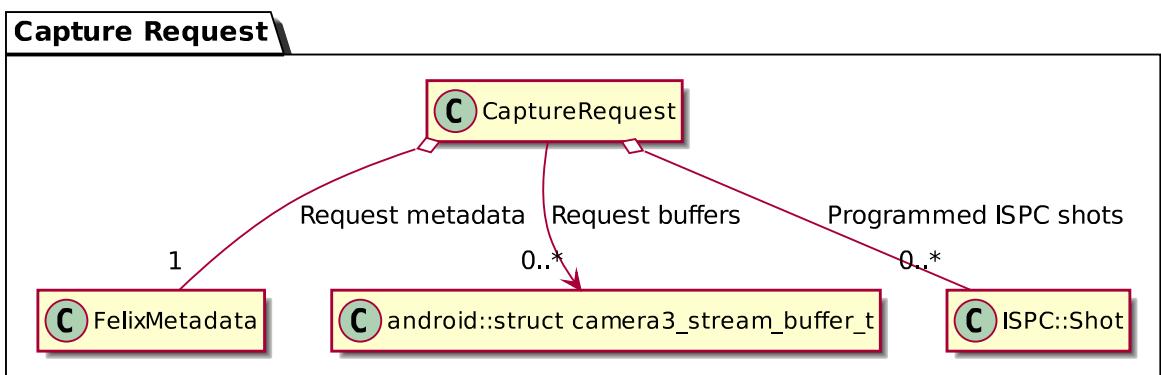
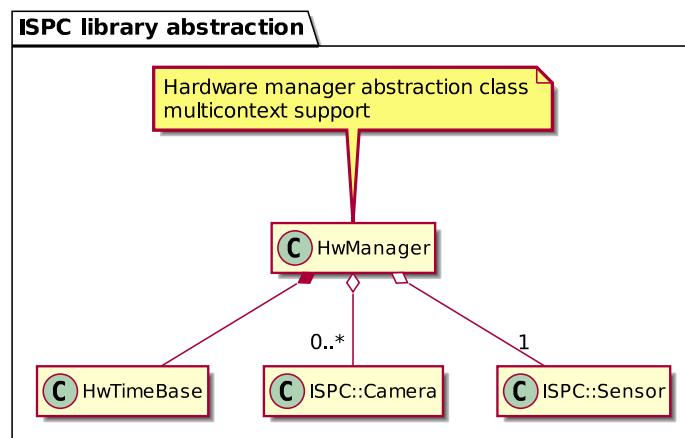
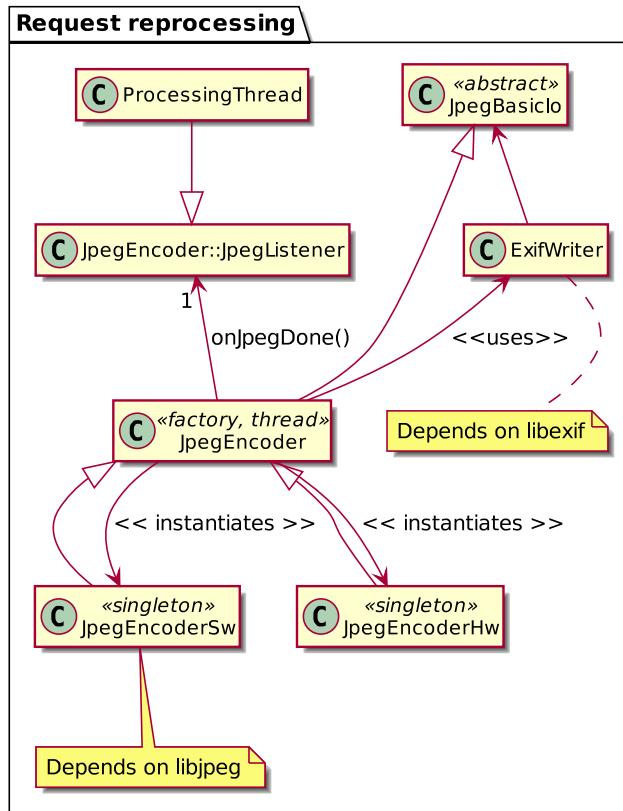
The library objects can be grouped into 7 different functionalities. These are:

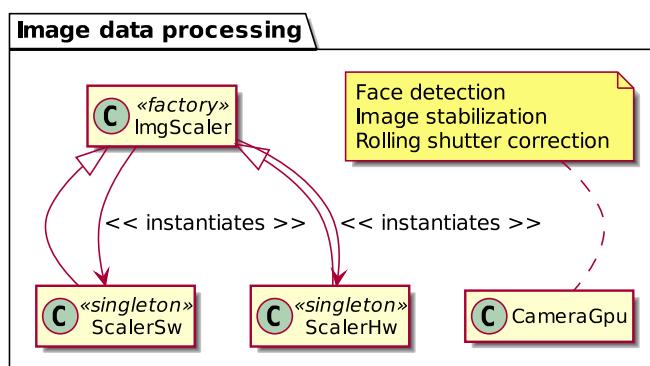
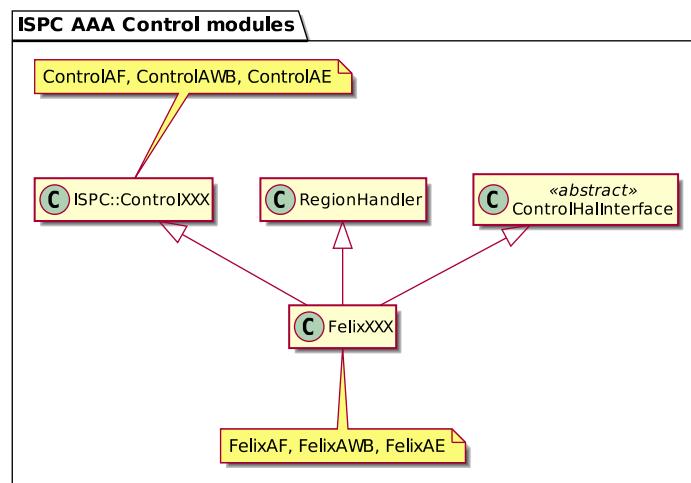
1. *Camera HAL v3.0 interface* (page 291)
 2. *Hardware abstraction* (page 295)
 3. *Capture request processing* (page 297)
 4. *AAA handling* (page 297)
 5. *JPEG compressor* (page 298)
 6. *Image processing* (page 300)
 7. *Helper classes* (page 301)

The following paragraphs contain high level description, design pattern used and data flow of each class.

Camera HAL v3.0 interface

This is the borderline between android framework and Camera HAL library. The classes expose the camera through [Camera module interface version 2 \(FelixCameraHAL\)](#) and Camera HAL device interface versions 3.0 ([FelixCamera](#)) and 3.2 ([FelixCamera3v2](#)).





FelixCameraHAL: FelixCameraHAL is the camera instance factory. This class is responsible of construction and initialisation of up to two instances of **FelixCamera[3v2]** class. The instances and supported API version is being chosen statically at build time. Please refer to *Library configuration* (page 318) for configuration method.

The design pattern of choice is a **Singleton**, which allows for deferred, in-place object construction.

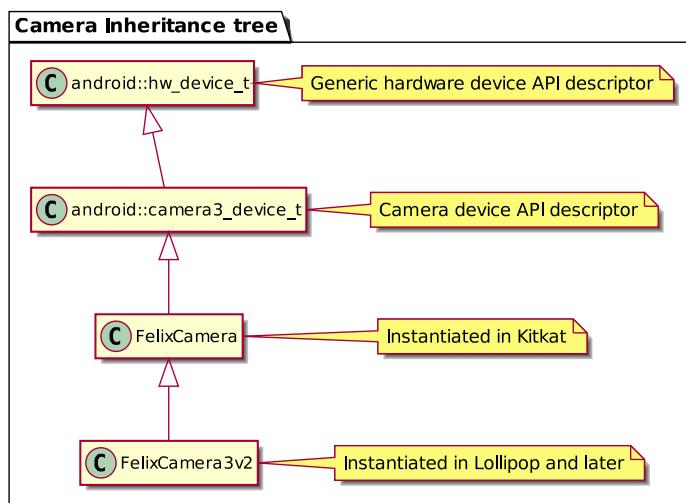
Implementation details

Each supported instance of **FelixCamera** class is being constructed and stored in internal container by **FelixCameraHAL** constructor. This instance is eventually passed indirectly to the framework as **hw_device_t*** pointer in the call:

```
FelixCameraHAL::device_open(const hw_module_t* module, const char* name,
    hw_device_t** device);
```

This effectively returns the pointer to **FelixCamera** object because the structure **camera3_device_t** is just an 'C' extension of **hw_device_t**.

These dependencies are shown on the Camera Inheritance tree diagram.



FelixCamera: This class implements Camera HAL API 3.0 interface, which is the maximum HAL API version supported in Android Kitkat (tested on Android 4.4 and up). This class also takes the role of the base class of **FelixCamera3v2** build in Lollipop distributions and later.

Note: Camera HAL v3.0 interface in Kitkat has been provided on the basis of support for early implementations of more advanced future camera drivers. As, at the same time, there was no official support for new high level **android.hardware.camera2** Java API, the media framework effectively translates legacy **android.hardware.camera** calls into native HAL 3.0 api calls. In effect, only a subset of HAL 3.0 features is being utilized.

Implementation details

The direct interface between the media framework and the HAL library is done using pointers to static member functions. An example of such interface call is:

```
static int initialize(const struct camera3_device *,
                     const camera3_callback_ops_t *callback_ops);
```

The interface call obtains the **FelixCamera** object instance from `struct camera3_device *` using simple cast operation (`FelixCamera::getInstance()`), and then pass the call parameters further to a actual implementation of API call in specific **FelixCamera** instance.

Each instance of the class is the owner of it's own instances of **ProcessingThread**, **JpegEncoder**, **CameraGpu** and **HwCaps**, and contains the reference to **HwManager** singleton instance. It also stores camera metadata (static info and request related) and manages currently configured streams by means of storing the dynamic list of **PrivateStreamInfo** objects.

FelixCamera3v2: This class implements Camera HAL API 3.2 interface, which is purposed for use in Android Lollipop and later (tested on Android 5.0 and up). Instead of aligning functionality of **FelixCamera** class to more advanced HAL 3v2 interface, the class inherits and extends **FelixCamera**. This approach has simplified maintenance of HAL functionality for Kitkat.

The separation made extension of static capabilities and default request metadata as well as addition of `flush()` call much easier.

Hardware handling

The group providing abstraction of ISP hardware and ISPC libraries.

HwCaps This class is a container class meant for storing stream related hardware capabilities.

The capabilities include supported pixel formats, resolutions and respective sensor modes and minimum frame durations. The caps are defined for all supported camera sensors and chosen at object construction phase.

Note: In order to add support for new sensors to HAL, this class must be extended with valid data.

HwManager HwManager class provides a separation layer between HAL and lower level ISPC and CI libraries, adds management of multiple contexts and their outputs and provides other helper methods.

Implementation details

The connection to CI kernel module is initialized at object construction time. Eventually, the **HwManager** object creates and configures the instance of **ISPC::Sensor**. This is done in order of reading the sensor parameters for early generation of static camera metadata. The camera pipelines (**ISPC::Camera** objects) are allocated when the application opens the camera HAL (`open_camera()`). Finally, the sensor object is connected to the main pipeline object.

A large group of **HwManager** methods operate on initialized contexts, that is, for example `HwManager::startCapture(void)` starts the pipeline capture on all **ISPC::Camera** objects owned by the specific instance of **HwManager**.

One of the most important methods provided by **HwManager** class is the

```
cameraHw_t* HwManager::attachStream(
    const
    camera3_stream_t* const newStream, CI_BUFTYPE& ciBuffType);
```

This method is being used by `configure_streams()` call to attach the output stream to one of the available HW outputs in one of the configured HW contexts. The association is done by storing the pairing between stream and the context in internal `buffersMap` container. Other action done after pairing is configuration of the context output in `ISPC::ModuleOUT` module.

Note: Output buffers purposed for capturing JPEG compressed images are allocated as BLOB type. As current HAL implementation uses software based JPEG compressor, this kind of buffers is not imported into ISP memory (that is, not mapped into ISP virtual memory space). Whether or not the BLOB buffers can be imported to HW MMU, is decided in `HwManager::getOutputParams()` method.

Hardware access lock

The `HwManager` class implements the global hardware lock mechanism, used internally by the class methods as well as exported to other entities by `android::Mutex HwManager::getHwLock()` method. This lock shall be used to control simultaneous access to ISPC library from more than one threads.

`HwTimeBase` Provides the time base synchronisation service for platforms, where the system clock read by `int64_t elapsedRealtimeNano()` call has different clock domain comparing to the 32 bit capture timestamps provided by v2500 ISP hardware.

Implementation details

As, each capture result shall be returned with the high resolution timestamp reflecting the moment of exposition start, and the framework (especially video recorder) would compare the frame timestamps with system clock, the hardware timestamps must be synchronized to the system clock domain for comparison purpose.

The principle of operation is very simple: the first capture after call to `HwManager::initHwNanoTimer()` stores the baseline system timestamp of the current capture session. This baseline comes from system clock and is converted to the HW clock domain using `uint64_t HwTimeBase::nsToHwClock(const nsecs_t nanoTime)`. The next timestamps are calculated by adding the calculated frame duration to the baseline. The resulting timestamp is converted back to system clock domain by `nsecs_t HwTimeBase::hwClockToNs(const uint64_t cycles)` and returned with the capture to the framework.

Note: Because system clock and ISP clock can have different hardware sources, some measureable clock drift will occur between hardware clock domain converted to system clock domain and current system clock. In time, the drift may grow to values having negative impact on other parts of A/V pipeline (such as video encoder or sync with audio). As a countermeasure, the baseline correction can be optionally applied so the clock. This can be enabled at build time by setting `MONITOR_CLOCK_DRIFT` in `HwManager.cpp`, and would rebase the timestamp clock if the difference between system based and HW based timestamp grows above 1ms.

Capture request processing

CaptureRequest The capture request abstraction, represents the single capture request across the whole Camera HAL. It also carries the request's metadata and “output buffer - hardware context” pairings.

Provides the implementation of different request processing phases:

- initial preprocessing
- programming to hardware
- shot acquisition from hardware
- sending capture results.

FelixMetadata The class derives from `android::CameraMetadata` class and extends it by the various processing methods for various metadata:

- AAA regions
- crop region
- statistics
- exposure parameters
- jpeg
- GPS etc.

The class also contains storage for chosen parameters carried by previous capture request. This is useful in monitoring changes in status of crop region and effect mode.

ProcessingThread Implementation of asynchronous capture request processing thread. It provides an interface for queuing `CaptureRequest` objects, signalling successful capture results and querying the state of internal queue and flush operation.

Implementation details

`ProcessingThread` derives from `android::Thread` and `JpegEncoder::JpegListener` in the purpose of asynchronous notification on finishing JPEG compression through `JpegEncoder::JpegListener::onJpegDone()` call. The class `android::Thread` adds the prototype of `bool threadLoop()` method, and its implementation `ProcesingThread::threadLoop()` is the actual request processing thread.

For more details on communication between FelixCamera and `threadLoop()` please refer to *Pipeline of capture request processing* (page 310).

AAA

ControlHalInterface An Abstract base class purposed to define a standard interface between HAL custom AAA control modules and the rest of HAL library.

Implementation details

The interface consists of the following pure virtual functions:

- `status_t initialize(void);` Post construction initialization of module

- **status_t processUrgentHALMetadata(FelixMetadata &settings);** Process AAA request metadata BEFORE enqueueing the shot in the ISP pipeline. Mainly used to program requested regions, sensor exposure settings etc. which apply for current request.
- **status_t processDeferredHALMetadata(FelixMetadata &settings);** Postprocess AAA request metadata BEFORE acquiring the specific shot from ISP. Used to set the proper state machine input. The state machine will be executed in FelixXXX::update() method AFTER acquiring the shot.
- **status_t updateHALMetadata(FelixMetadata &settings);** Update request metadata fields related to specific AAA functionality Used to fill the metadata with shot statistics and current AAA state before sending capture result.
- **status_t initRequestMetadata(FelixMetadata &settings, int type);** Initialization of request template related with specific AAA functionality
- **static void advertiseCapabilities(CameraMetadata &info);** Initialization of static camera info related with specific AAA functionality

FelixAF, **FelixAE**, **FelixAWB** Android Camera HAL specific AAA control modules. These modules are HAL specific ‘plug-ins’ for ISPC library control module functionality, and are created as direct replacements for their base classes **ISPC::ControlAF**, **ISPC::ControlAE** and **ISPC::ControlAWB**. The classes implement common **ControlHalInterface** methods and a **RegionHandler<>** template class, for convenient handling of AAA regions.

For more on handling AAA metadata flow please refer to *Metadata handling* (page 315).

JPEG compression

This group of classes is responsible of encoding image data to JPEG format and generation of Exif embedded headers.

JpegEncoder This class implements common encoder operations and supports JPEG compression run in separate thread.

Implementation details

JpegEncoder derives from **android::Thread** for threading support and an abstract base **JpegBasicIo**, for write operations inside jpeg data buffer. The proper implementation of **JpegBasicIo** has been left to it’s child classes.

At the same time the class implements a static **JpegEncoder::get()** factory method which, depending on the **USE_HARDWARE_JPEG_COMPRESSOR** environment variable, instantiates software or hardware based Jpeg compressor object.

Another supported interface is the external **JpegEncoder::JpegListener** object (usually caller of **JpegEncoder**) which is being asynchronously notified about finished compression by calling **onJpegDone()** on it.

The proposed sequence to initialize the compression using provided **StreamBuffer** buffers and **FelixMetadata** container is:

```
StreamBuffer encoderInputBuffer;
StreamBuffer encoderInputBuffer;
FelixMetadata FrameSettings;
...
```

```

// get or create instance of JpegEncoderSw
JpegEncoder& jpeg = JpegEncoder::get();
// set buffers
jpeg.setInputBuffer(encoderInputBuffer);
jpeg.setOutputBuffer(encoderOutputBuffer);
jpeg.setRequestMetadata(FrameSettings);
// the caller class derives from JpegEncoder::JpegListener
jpeg.setJpegListener(this);
// spawn the compression thread
if (jpeg.start() != OK) {
    return UNKNOWN_ERROR;
}
...
// wait for asynchronous onJpegDone() call

```

JpegEncoder::JpegListener This class defines a simple listener/observer pattern for every module using `JpegEncoder` class. The observer registration is made by `JpegEncoder::setJpegListener()` call and the observer notification is done by compression thread calling `JpegEncoder::JpegListener::onJpegDone()` method.

JpegEncoderSw The class implements purely software based JPEG image compression. The actual compression is done using external `libjpeg` library.

Implementation details

The implementation consist of custom `libjpeg` callbacks and image data preparation with support for YUV420/422 and RGB24 buffers. Because of specific requirements on YUV image data organisation imposed by `libjpeg`, the deinterleaving code has been implemented so the input image data for the library is reorganized in separate Y-U-V or R-G-B planes.

JpegBasicIo This is a simple byte write interface for `ExifWriter` class, designed to be implemented in `JpegEncoder` child classes. External `ExifWriter` object can use it to write the Exif data into current location of output buffer, known only by `Jpeg` encoder.

ExifWriter Abstraction for external `libexif` library implemented for Exif header generation with thumbnail support. The version of Exif header currently supported by Android Lollipop release is 2.1.

Implementation details

The object constructor is called with reference to `JpegBasicIo` and `FelixMetadata` objects. First provides byte byffer write operations, the latter is the container with source metadata to be written in Exif format.

The Exif header generation is done in `writeApp1()` method.

Please find below the list of exif tags directly generated by `ExifWriter` object. This list has been chosen as the bare minimum to pass the CTS tests suite in Lollipop:

- EXIF_TAG_COMPRESSION
- EXIF_TAG_FOCAL_LENGTH
- EXIF_TAG_EXPOSURE_TIME
- EXIF_TAG_APERTURE_VALUE
- EXIF_TAG_ISO_SPEED_RATINGS

- EXIF_TAG_FLASH
- EXIF_TAG_WHITE_BALANCE
- EXIF_TAG_ORIENTATION
- EXIF_TAG_PIXEL_X_DIMENSION
- EXIF_TAG_PIXEL_Y_DIMENSION
- EXIF_TAG_COLOR_SPACE
- EXIF_TAG_COMPONENTS_CONFIGURATION
- EXIF_TAG_MODEL
- EXIF_TAG_MAKE
- EXIF_TAG_DATE_TIME
- EXIF_TAG_DATE_TIME_DIGITIZED
- EXIF_TAG_DATE_TIME_ORIGINAL
- EXIF_TAG_SUB_SEC_TIME
- EXIF_TAG_SUB_SEC_TIME_DIGITIZED
- EXIF_TAG_SUB_SEC_TIME_ORIGINAL
- EXIF_TAG_GPS_LATITUDE
- EXIF_TAG_GPS_LATITUDE_REF
- EXIF_TAG_GPS_LONGITUDE
- EXIF_TAG_GPS_LONGITUDE_REF
- EXIF_TAG_GPS_ALTITUDE
- EXIF_TAG_GPS_ALTITUDE_REF
- EXIF_TAG_GPS_TIME_STAMP
- EXIF_TAG_GPS_DATE_STAMP
- EXIF_TAG_GPS_PROCESSING_METHOD
- EXIF_TAG_JPEG_INTERCHANGE_FORMAT
- EXIF_TAG_JPEG_INTERCHANGE_FORMAT_LENGTH

The complete execution flow of the Jpeg compression within Camera HAL has been pictured in *JPEG reprocessing request* (page 313).

Image processing

ImgScaler The abstract base class and at the same time a factory for **ScalerSw** and **ScalerHw** specializations.

Implementation details

The specialized child classes are instantiated by calling

```
static ImgScaler& ImgScaler::get();
```

The actual scaling is done by implementation of

```
status_t scale(StreamBuffer& src, StreamBuffer& dst);
```

ScalerSw This child class of `ImgScaler` implements software based downscaling. The implementation supports YUV and RGB24 images.

CameraGpu This class is a wrapper around `camera_gpu` class defined in external **Vision libraries** library package for Android. The `camera_gpu` implements PoverVR based, accelerated image transformations like:

- face detection
- image stabilization
- rolling shutter correction

Helper classes

PrivateStreamInfo This class represents an Camera HAL output stream. The instances of this class are managed by `FelixCamera` and passed through `void* priv` pointer in `camera3_stream_t` structure.

Implementation details

This class allows management of buffers accosiated with respective stream.

StreamBuffer This class is a utility wrapper class around framework provided `camera3_stream_buffer_t` buffer descriptor. It abstracts different `mmap` implementations for YUV and RGB buffer types via “lock()/unlock()“ methods as well as easy association of current buffer with proper HW output from a group of available contexts.

localStreamBuffer This class, derived from `StreamBuffer`, represents image buffers allocated in-place for HAL internal usage. It’s mainly used for allocation of local buffer purposed for non ZSL captures and for intermediate scaled/cropped images in JPEG compression execution flow.

sRegion Extension of “`android::Rect`“ class providing means for managing these objects as tiled weighted regions. This class is used throughout all AAA control modules in HAL.

RegionHandler<> This abstract base class implements the means of management of weighted regions group.

The support for different algorithms has been defined for internal use of AAA class deriving from `RegionHandler`.

The predefined operation modes are:

- **CENTER_WEIGHT** The output matrix calculation uses predefined matrix with centered weights (the weight is highest on center of the image and falls gradually towards the egde tiles)
- **BEST_TILE** The best tile from the weighted input set is determined and reported in `mBestRegion`
- **WEIGHTED_AVERAGE** This works like `CENTER_WEIGHT` but the weight matrix is provided in input region data.

- **SINGLE_ROI** The AAA measurement works on single region only, therefore the use of **weight** factor in calculations is pointless.

All algorithms should use **mCurrentRegions** as the source set for calculation of

```
output[tile] = sRegion[tile].weight * metering[tile];
```

product for each input tile.

The result from the algorithm (the tile with the highest output value) may be stored in **mBestRegion** member, and used as needed in AAA module.

For example, AF module would use the sharpest region to set and report the best focused region.

The exact operation mode which should be applied for the current request should be determined by calling

```
metering_mode_t RegionHandler<T>::getMeteringMode(  
    const regions_t& regions);
```

This method implements the rules for region handling, described in internal Android documentation file **Android Camera HAL3.0 Properties**².

Note: The exact choice which modes are supported and further implementation of chosen modes is left to specific AAA module.

9.2.3 Structure of source code

The following structure depicts the most important nodes of Android Camera HAL source hierarchy:

```
$ANDROID_BUILD_TOP/hardware/img/v2500  
|  
|   \- DDKSource           (ISP DDK)  
|   \--Android  
|       |  
|       \--CTS              (CTS test plans)  
|       \--FelixCamera       (Camera HAL sources)  
|           |  
|           \--AAA            (AAA handling)  
|           \--Helpers         (Helper classes)  
|           \--JpegEncoder     (Jpeg encoder)  
|           \--Scaler          (Scaler classes)  
|           |  
|           \--Android.mk  
|           |  
|           \--gralloc          (custom gralloc sources)  
|           |  
|           \--Android.mk  
|  
|   \--FelixDefines.mk      (Build time configuration)  
|   \--Android.mk           (Android build system Makefile)
```

² Dynamically generated in AOSP environment under \$ANDROID_BUILD_TOP/system/media/camera/docs/docs.html

9.2.4 External dependencies

the following lists all direct shared dependencies of **Felix** Camera HAL library.

libjpeg.so

Software JPEG compression/decompression library. Provided in Android AOSP and built automatically by Android build system from `$ANDROID_BUILD_TOP/external/libexif`.

libexif.so

Exif data processing library. Provided in Android AOSP and built automatically by Android build system from `$ANDROID_BUILD_TOP/external/libexif`.

libfelix_ispc, libfelix_sensor, libfelix_ci, libfelix_common

Userspace interface for handling **Felix** ISP processors. Built from **DDKSource**. For more information please refer to *Capture Interface* (page 140) and *ISP Control Library* (page 190) chapters.

libcameragpu.so

This shared module provides HAL interface for vision processing libraries which make use of Imagination Technologies GPU processors for the purpose of face detection, image stabilization, rolling shutter correction and other software based camera pipeline stages. The library and it's own dependencies are provided by Imagination Technologies in separate package and are built independently from Camera HAL. For more information please read *Vision Libraries Usage Instructions* (page 336).

Android native framework

The Android framework dependencies used throughout HAL:

- libcamera_metadata.so
- liblog.so
- libutils.so
- libcutils.so
- libcamera_client.so
- libui.so
- libstlport.so

9.2.5 Build system

The Camera HAL build system has been integrated with Android build system using standard `Android.mk` files. The main script, placed under ISP Camera HAL root `$ANDROID_BUILD_TOP/hardware/img/v2500` location, is responsible for build-time configuration and serialization of HAL dependency builds (`libfelix_ispc`, `libfelix_sensor`, `libfelix_ci`, `libfelix_common`, `gralloc`).

The `$ANDROID_BUILD_TOP/hardware/img/v2500/Android/FelixCamera/Android.mk` script, which is responsible of building main Camera HAL library, parses Android `$PLATFORM_VERSION` variable and decides on the version of HAL interface to be supported by Camera HAL library.

In effect, under Android 4.x Kitkat, the library will be built with support for **HAL v3.0** (implemented in `FelixCamera.cpp`). For Android 5.x Lollipop and later, the library is built with **HAL v3.2** camera interface (implemented in `FelixCamera3v2.cpp`).

Note: Due to backward compatibility reasons, Felix ISP Camera HAL implements support for both interface versions. This is because **Kitkat** media framework does not support Camera HAL interfaces later than v3.0.

For **Lollipop**, the Camera HAL library supports most recent interface version by design (the current is v3.2).

Note: The choice of supported version of HAL interface is done in two ways:

1. Value of major number of `$PLATFORM_VERSION` which switches optional build of "Felix-Camera3v2.cpp"
2. The value of `CAMERA_DEVICE_API_VERSION_CURRENT` defined in `camera_common.h`

The latter is used for

- Instantiation of specific camera device type (`FelixCamera` or `FelixCamera3v2`) in `FelixCameraHAL::instantiateCamera()`.
- Several `#ifdef` switches in code (used for smaller incompatibilities)

As the build target, the `LOCAL_MODULE` variable has been hardcoded to `felix.camera.default`. In effect, the target library filename is `felix.camera.default.so`.

Warning: The currently hardcoded `LOCAL_MODULE` library name is dependend on the patch `hardware-libhardware_felix_gralloc_camera_default.patch`. This patch changes the default system-wide Camera HAL library prefix from `camera` to `felix.camera` used in media framework `hardware.c` HAL loader module.

The customer is encouraged to switch the prefix back to `camera` for production platforms (by not applying the patch and updating the `Android.mk` accordingly).

For more info on configuring Android builds, please refer to [Building Android Camera HAL](#) (page 327).

The build time configuration of the library is done via `$ANDROID_BUILD_TOP/hardware/img/v2500/FelixDefines.mk` text file. More on library configuration has been described in [Library configuration](#) (page 318).

9.2.6 Hardware requirements

The requirements, which current version of Android Camera HAL imposes on **Felix** ISP processors, are listed below:

Number of pipeline contexts for each opened camera instance

This hardware requirement comes from the high level Android Camera framework requirement saying that at least 2 (for LIMITED) and 3 (for FULL mode devices) simultaneous processed output streams must be supported ³.

In ISP hardware each individual context supports a pair of RGB and YUV type outputs, each with different resolution.

The table below summarizes the requirement of number of pipelines required per each running camera instance.

	Output formats	Number of HW contexts
FPGA-PC	RGB preview YUV encode YUV 0 shutter lag	2
GPU enabled	YUV preview YUV encode YUV 0 shutter lag	3

Calculations for different types of platforms follow below.

1. On FPGA-PC development platform. This platform does not support GPU+GLES, the **software display composer** has to be used in AOSP Android builds.

Due to lack of support for YUV pixels for display surfaces on such builds, `RGBX_8888` format is used for best performance of Camera preview. Therefore, in order of enabling live camera preview on Android builds for FPGA based PC, the patch `frameworks-av-api1-rgb-preview-stream-format.patch` has to be applied on AOSP sources. This replaces hardware-agnostic `CAMERA2_HAL_PIXEL_FORMAT_OPAQUE` format requested for preview streams (commonly used as YUV) with `HAL_PIXEL_FORMAT_RGBX_8888`.

Because we are able to hardcode the preview output pixel type to RGB type on FPGA-PC this needs 1 HW context. Handling the additional 2 YUV outputs need another HW context. This in effect requires context 0 for RGB+YUV and context 1 for YUV only.

2. GPU enabled platform. This platform would most probably need YUV data as the source for hardware preview surfaces, there's no possibility to use RGB data for it, thus requiring 3 separate hardware contexts to output at least three simultaneous YUV streams for best possible performance.

9.3 Implementation of Camera HAL v3.x

The Camera HAL library for Felix ISP has been developed in C++ language with C++11 support enabled by compiler parameter in `FelixCamera/Android.mk` file.

³ `$ANDROID_BUILD_TOP/system/media/camera/docs/docs.html#static_android.request.maxNumOutputProc`

9.3.1 Interface calls

This chapter contains description of each camera interface call with additional explanation covering the implementation details.

The order of execution is shown in the lifetime diagram in *Lifetime of Camera HAL* (page 289).

`open()`

This call, declared in `hw_module_methods_t` structure and provided to the camera framework in `camera_module_t HAL_MODULE_INFO_SYM` symbol defined in `FelixCameraHAL.cpp` file, implements a delegate to target method

```
status_t FelixCamera::connectCamera(hw_device_t** device)
```

of chosen instance of camera, through

```
int FelixCameraHAL::cameraDeviceOpen(int camera_id, hw_device_t** device)
```

This call initializes the camera instance by:

- instantiation of `ProcessingThread` object
- initialization of all hardware contexts used in specific camera instance (currently hardcoded to 2 for each opened camera instance)
- instantiation of all control modules (including AAA modules)
- setting default pipeline configuration

and returns the pointer to the initialized camera instance object (`FelixCamera`).

`get_camera_info()`

This is a delegate to camera instance's

```
status_t FelixCamera::getCameraInfo(struct camera_info *info)
```

The purpose of this call is to return static camera info and capabilities to the media framework. The pointer to the static camera info must be valid throughout lifetime of specific camera instance, and is stored in `FelixCamera::mCameraStaticInfo` field.

`construct_default_request_settings()`

This call is implemented in `FelixCamera` class and is a delegate for

```
camera_metadata_t* FelixCamera::constructDefaultRequestSettings(
    int type)
```

The purpose of this call is to prepare the Camera HAL specific metadata generated for each possible request type defined in `camera3_request_template_t` (such as defined in *Main use cases* (page 288)).

This call instantiates the `FelixMetadata` object and fills it with default metadata configuration, used later as capture request template.

configure_streams()

The delegate to

```
status_t FelixCamera::configureStreams(  
    camera3_stream_configuration *streamList)
```

This call's purpose is to reset the camera processing pipeline and set up new input and output streams. This call replaces any existing stream configuration with the streams defined in the stream_list.

The core responsibility of the call is to associate each requested output stream with the available ISP context outputs and setup the required pipeline and sensor parameters.

The sequence diagram of the execution flow of configure_streams() is shown below.

register_stream_buffers()

A delegate of

```
status_t FelixCamera::registerStreamBuffers(  
    const camera3_stream_buffer_set *bufferSet)
```

This call implements importing a whole group of stream buffers to ISP hardware in a bulk.

Note: This call is required in HAL v3.0 for Kitkat, but has been set as deprecated in camera HAL v3.2 used since Lollipop, therefore **FelixCamera3v2** overloads this call with empty implementation.

process_capture_request()

A delegate of

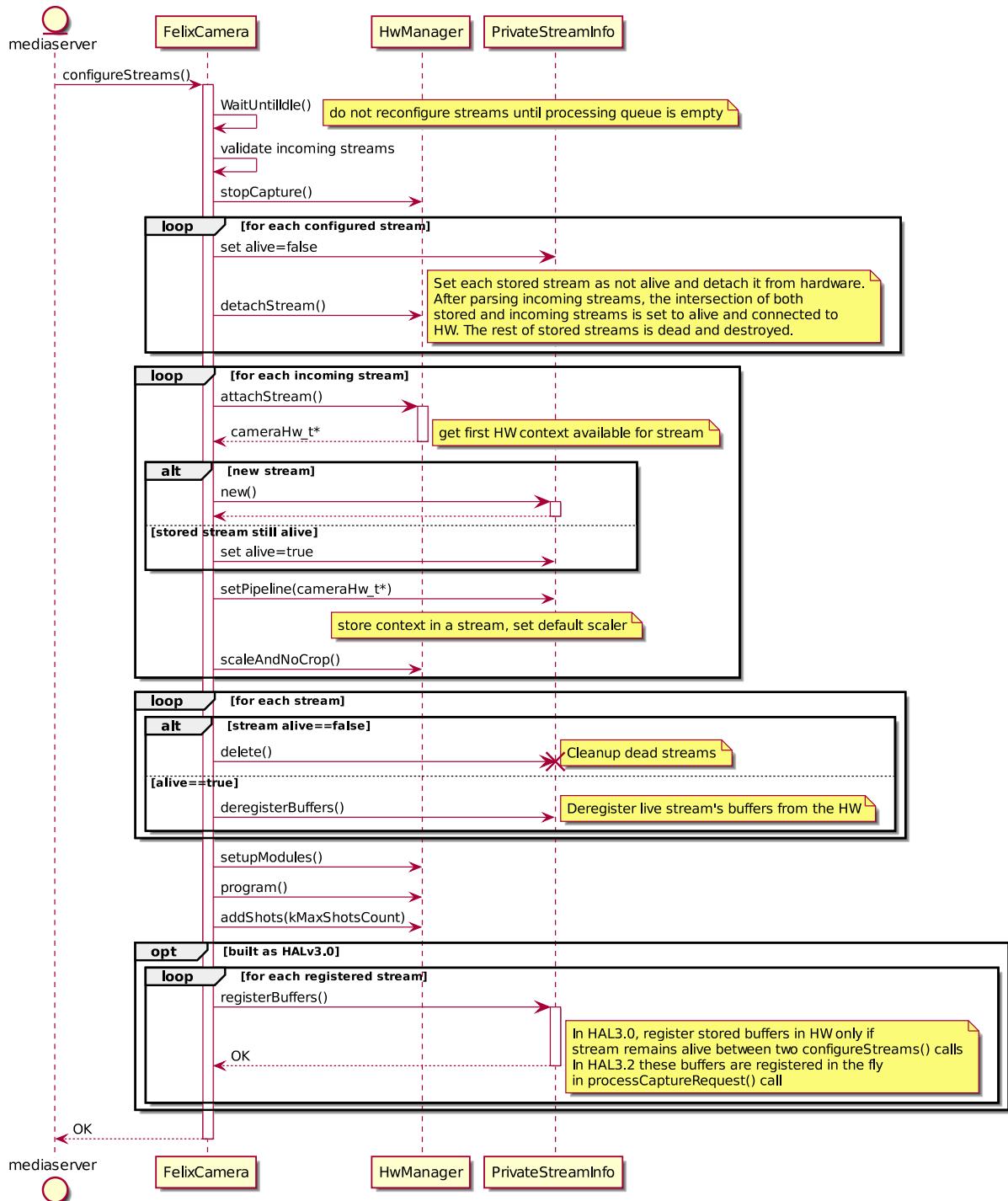
```
status_t FelixCamera::processCaptureRequest(  
    camera3_capture_request *request)
```

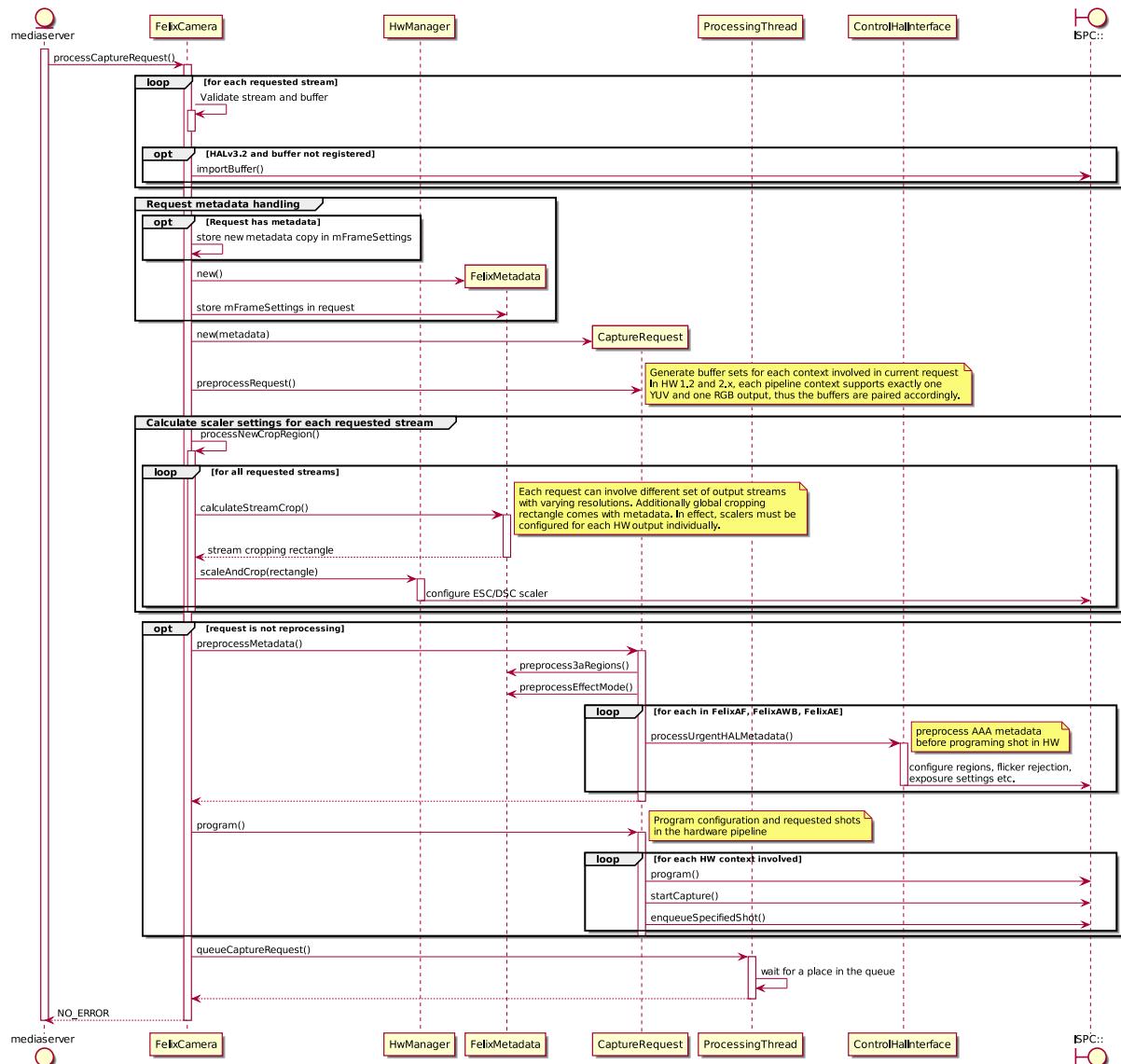
This call implements the preprocessing, programming and queueing the capture requests coming from camera framework, into ISP hardware.

During execution of **processCaptureRequest()**, the following steps are done:

- on-fly buffer import to ISP (in HAL v3.2 only)
- instantiation of **CaptureRequest** and **FelixMetadata** objects
- association of output buffers to correct context outputs
- crop region and scaling is calculated and set-up for each HW output involved
- request metadata is processed (3A, 3A regions, effects, scene modes etc)
- the pipeline is configured according to the requested metadata and started
- the shots are programmed in the ISP hardware
- the **CaptureRequest** object is enqueued into **ProcessingThread** request queue.

Below the execution flow is shown on the diagram below.





close()

A delegate of

```
status_t FelixCamera::closeCamera()
```

This call stops the capture on all ISP contexts and destroys all owned objects.

9.3.2 Pipeline of capture request processing

The architecture of request processing pipeline implemented in **Felix** Camera HAL library is of the asynchronous producer-consumer type.

The caller thread of `processCaptureRequest()` method takes the producer role. All programmed and queued requests are acquired from ISP and processed asynchronously in consumer thread which continuously executes `ProcessingThread::threadLoop()` method.

Note: The thread is implemented in such way that returning boolean `false` in `threadLoop()` causes the thread to exit. Due to this condition, the boolean `true` is always returned value in case of repeated request processing, except the occurrence of some fatal errors.

IPC

The IPC mechanisms used in Camera HAL are:

- conditional variables of type `android::Condition`
 - `FelixProcessing::mCaptureRequestQueuedSignal`
Signaling of new request enqueued to processing thread queue
 - `FelixProcessing::mCaptureResultSentSignal`
Signaling the event of sending request result to the framework, meaning the same as capture request slot has been freed in the processing queue.
- “`FelixProcessing::mLock`“ of `android::Mutex` type
Protects the conditional variables and the `CaptureRequest` objects queue

Note: These IPC classes are declared and documented in `$ANDROID_BUILD_TOP/system/core/include/utils` in `Condition.h` and `Mutex.h`.

Request processing flow

The execution flow of communication between producer and consumer threads in Camera HAL library has been pictured on the diagram below.

The picture shows the event of enqueueing capture request **N** when there's an immediate free slot available in the request queue. In this case the `mCaptureResultSentSignal::waitRelative()` is not called and the producer immediately signals the consumer thread with `ProcessingThread::mCaptureRequestQueuedSignal` and finally the processing of the enqueued request can start.

The optional execution branch (when `mCaptureResultSentSignal::waitRelative()` returns `TIMED_OUT` each `kWaitPerLoop` milliseconds) will be taken when the queue has no free slots for new requests. In this case if queue is still empty or `waitRelative()` returned `TIMED_OUT`, the `threadLoop()` exits with boolean `true` so the polling is run in the loop.

After successful push of the request into the queue slot, `processRegularRequest()` method is called. This method handles acquisition of all programmed shots from ISP hardware. After the shots got read successfully from hardware, the framework is notified with `notifyShutter()` callback with the request frame number and high resolution timestamp of start of frame capture.

In case of BLOB output buffer was requested, the consumer thread calls `processJpegEncodingRequest()`, which handles all steps needed for obtaining JPEG output image. More on this in *JPEG reprocessing request* (page 313) subchapter.

Capture request use cases

The `ProcessingThread` must support all possible combinations of capture requests which may occur during camera capture session (see *Main use cases* (page 288)). All these combinations are covered by calling `processRegularRequest()` and `processJpegEncodingRequest()` handlers.

The possible options of capture requests (please note that RAW outputs are not supported by HAL right now) are listed below.

1. Non-stalling captures

	Output streams				Input stream	
	YUV	RGB	JPEG	RAW	YUV	RAW
preview	0-3	0-1	N/A	N/A	N/A	N/A
video recording						
application callback						

In this case only `processRegularRequest()` is called to acquire output buffers from ISP hardware. Single capture result is sent with all non-stalling output buffers.

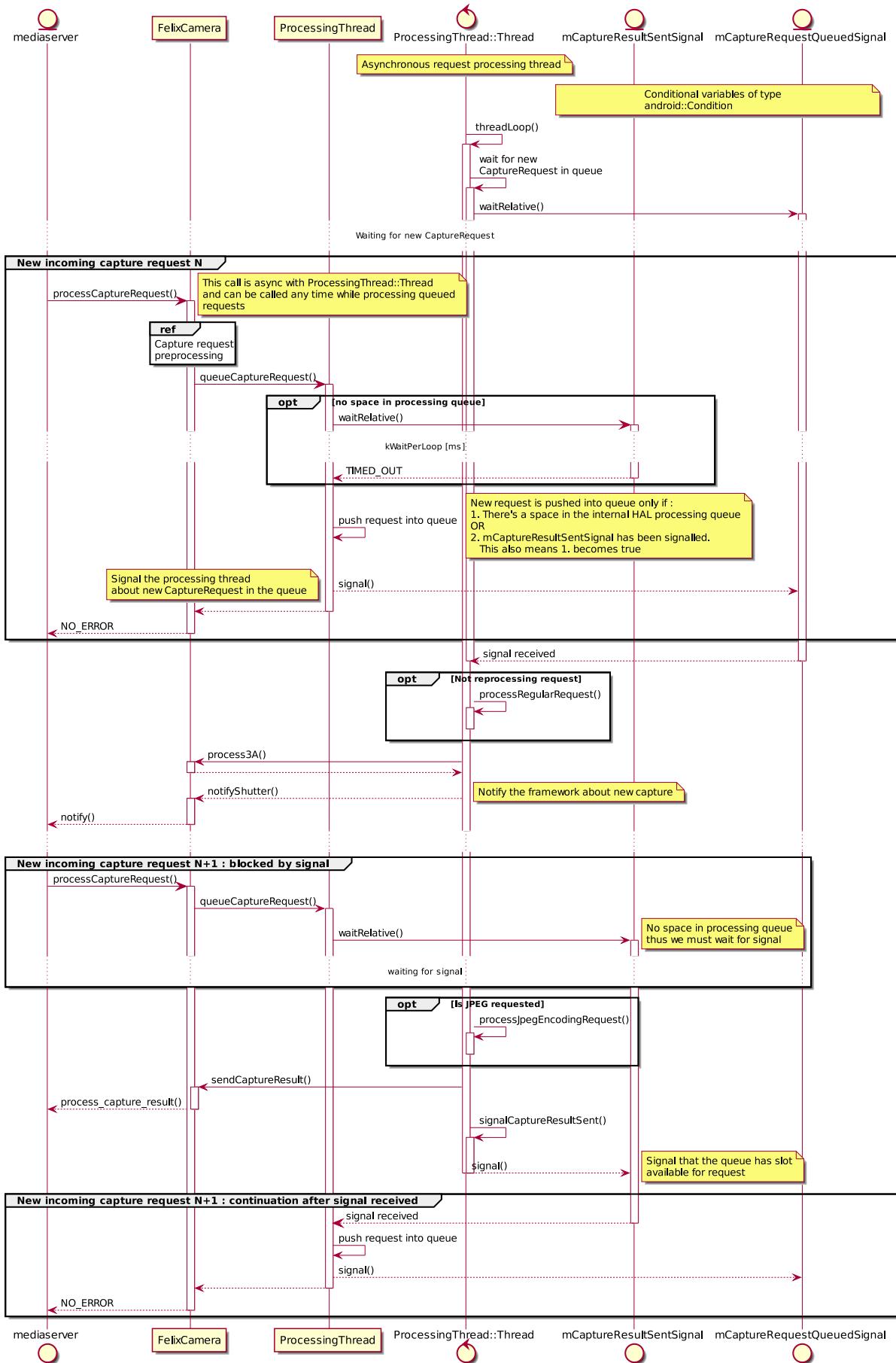
2. Non-stalling captures + stalling non ZSL JPEG capture

	Output streams				Input stream	
	YUV	RGB	JPEG	RAW	YUV	RAW
preview	0-3	0-1	1	N/A	N/A	N/A
video recording						
non ZSL still capture (JPEG)						
application callback						

`processRegularRequest()` is called to acquire data for non-stalling buffers from ISP hardware. Additionally, `FelixCamera::mNonZslOutputBuffer` is used as additional output for JPEG compressor data source, imported and programmed into ISP. After acquiring all programmed shots from ISP, all requested non-stalling buffers are returned in first partial capture result to the framework.

Further processing goes to `processJpegEncodingRequest()`, which is set up to use `FelixCamera::mNonZslOutputBuffer` as the input buffer for compressor. Finally, the second partial capture result is being sent to the framework with an outstanding JPEG image.

3. ZSL YUV to JPEG reprocessing



	Output streams				Input stream	
	YUV	RGB	JPEG	RAW	YUV	RAW
ZSL YUV to JPEG reprocessing	N/A	N/A	1	N/A	1	N/A

This request type does not involve `processRegularRequest()` call because an input buffer has been provided in the request. In this case, only `processJpegEncodingRequest()` is being called and single capture result with one output BLOB type buffer is sent to the framework.

9.3.3 JPEG reprocessing request

Generation of JPEG compressed images within **Felix** Android Camera HAL needs more explanation because it involves several steps of cropping and scaling of the input image, especially when thumbnail image has been requested in Exif.

Prescaling and cropping

The downscaling of source images is done in the following cases:

- The ZSL input buffer has different resolution than requested output image
- The Exif thumbnail is requested

Additionally, the aspect ratios can differ between source and target images, thus **zero-copy** cropping of the source image is being done.

Note: The zero-copy cropping is currently implemented in `StreamBuffer::cropAndLock()` instead of using `gralloc_lock()/lock_ycbcr()` with provided cropping rectangle (as it is not implemented in example `gralloc`). This `StreamBuffer::cropAndLock()` calls `gralloc_lock()` and calculates the top-left offset of the crop rectangle within the buffer and new cropped buffer size.

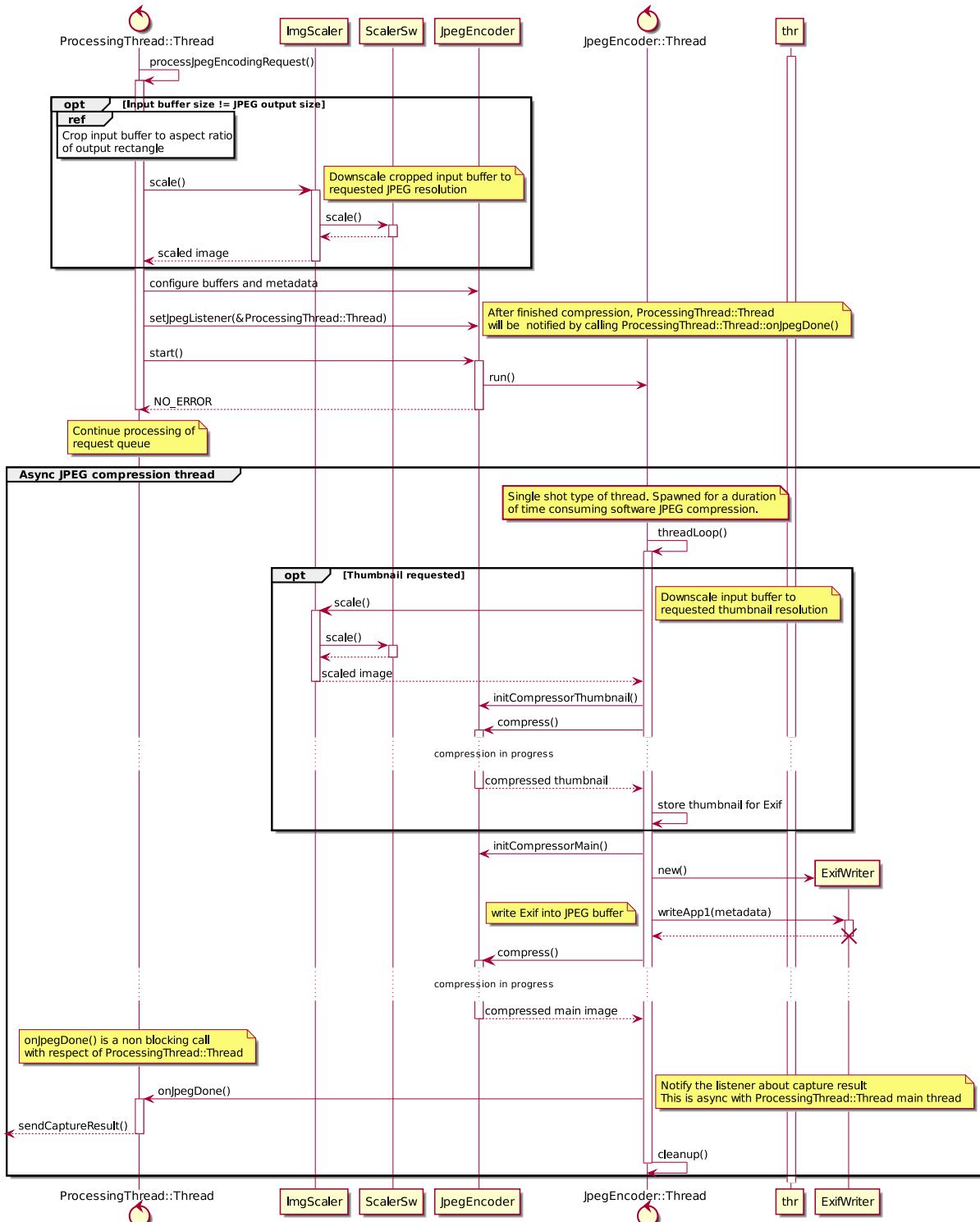
JPEG compression flow

Actual Jpeg compression is executed asynchronously in its own thread. Unlike `ProcessingThread` which is run in the loop, JPEG compression is run in single shot mode (`threadLoop()` returns boolean `false`).

The means of signaling about compression finish is implemented by **Observer/Listener** design pattern. Here, `ProcessingThread` registers itself as the observer to `JpegEncoder` using `JpegEncoder::setJpegListener()` call. This requires `ProcessingThread` to derive from `JpegEncoder::JpegListener` class and provide implementation of `JpegEncoder::JpegListener::onJpegdone()`.

Note: Current implementation does not support multiple observers.

The diagram below shows the timeline of JPEG processing as done in `processJpegEncodingRequest()` in **Felix** Camera HAL.



9.3.4 Metadata handling

Due to the asynchronous pipelined nature of shot processing done by Imagination Technologies ISP software stack, AAA data must be processed in specific order in order to maintain full integrity between requested input and processed output metadata. AAA data in **Felix Camera HAL** is being handled by three classes **FelixAE**, **FelixAWB** and **FelixAF**.

As programming and processing of Shots is performed in separate threads, the metadata requested for specific capture has to be paired with chosen output buffers and enqueued as **CaptureRequest** for further use in AAA state machines⁴ executed by consumer thread. A part of request metadata is used immediately to generate register configuration of hardware context and pipelined in CI library (implemented in **FelixXXX::processUrgentHALMetadata()**).

Note: Pipelining of the configuration register values to be programmed has been realized using internal list of Shots in CI library. A single Shot is a container of pipeline configuration and buffers used in a single capture (see *Shot and Buffer lifecycle* (page 179) in *Capture Interface* (page 140)).

The consumer thread uses the metadata to set the inputs of state machines implemented in AAA control modules (**FelixXXX::processDeferredHALMetadata()**).

Eventually the state machines are executed in **ISPC::ControlModule::update()** method, directly after the shot has been acquired from ISP (see *Module Setup and Update* (page 202) in *ISP Control Library* (page 190)).

After calculating output (that is next states) of AAA state machines, the result metadata has to be filled with valid values. This is done in **FelixXXX::updateHALMetadata()**.

The diagram below depicts high level view of AAA metadata handling sequence in IMG Android Camera HAL.

9.3.5 Vendor metadata tags

Current version of Felix Camera HAL does not support custom vendor metadata tags.

9.3.6 Current state of supported functionality

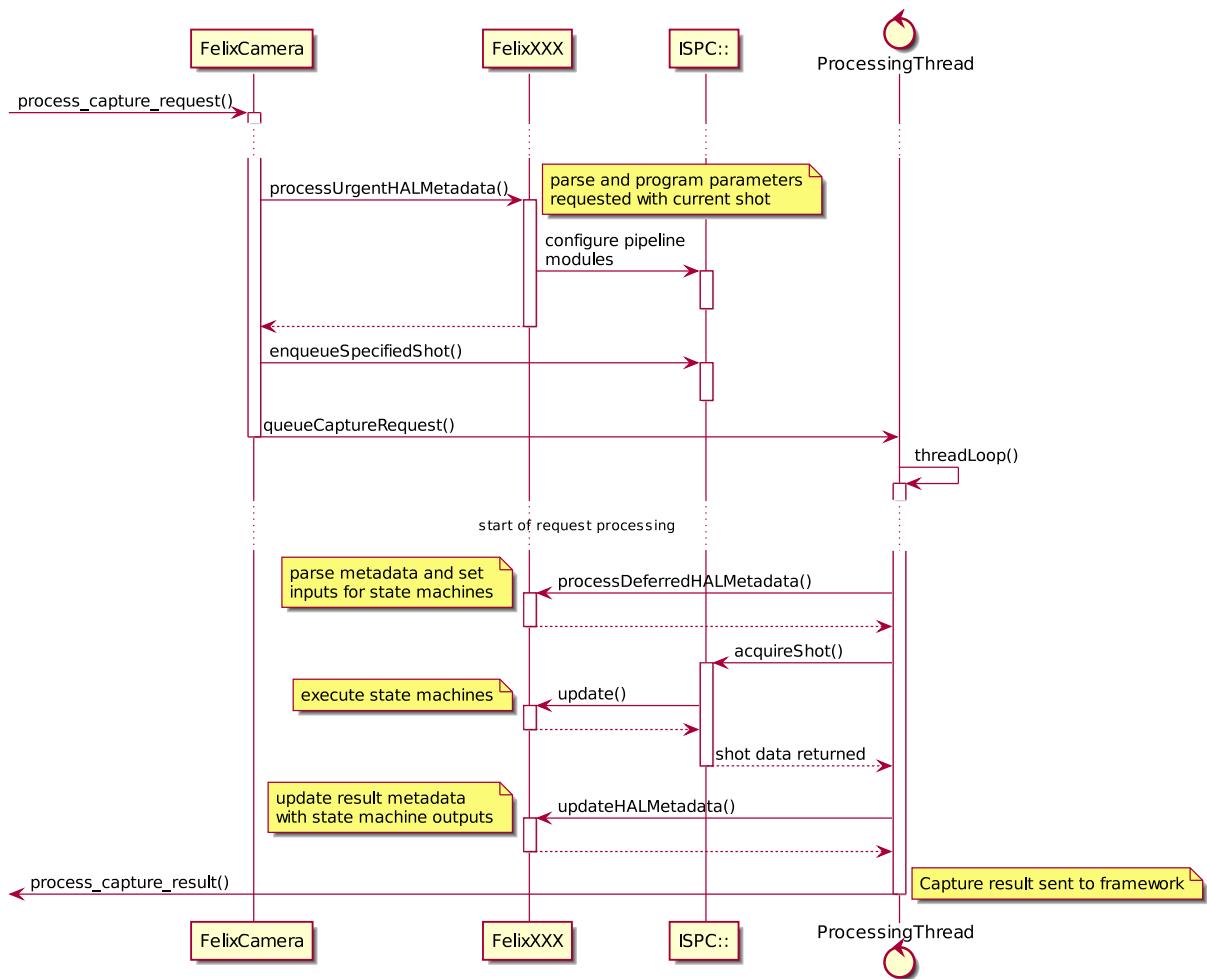
Please contact your FAE to obtain the current implementation status of supported for **LIMITED** and **FULL** camera hardware support levels.

9.4 Gralloc

The Android Camera HAL package provides an example implementation of **gralloc** module.

This **gralloc** has been customized to make full use of Android **ion** memory allocator and implements an extended **private_handle_t** structure in purpose of storing and exposing extended buffer metadata to other processes.

⁴ https://source.android.com/devices/camera/camera3_3Amodes.html



9.4.1 Dependencies in Camera HAL

Internally, `private_handle_t` stores the buffer metadata in `struct buffer_attrs`. In order of hiding the internal structure implementation from callers, the getter methods have been implemented and exposed to Camera HAL.

Warning: The Camera HAL depends on the getter API directly. The dependencies are defined in `Helpers.h`, and follow below:

```
#include "gralloc_priv.h"

ionFd_t getIonFd(const buffer_handle_t handle){
    return static_cast<const private_handle_t *>(handle)->getIonFd();
}

size_t getBufferSize(const buffer_handle_t handle){
    return static_cast<const private_handle_t *>(handle)->getSize();
}

size_t getTiled(const buffer_handle_t handle){
    return static_cast<const private_handle_t *>(handle)->getTiled();
}

size_t getBufferStride(const buffer_handle_t handle){
    return static_cast<const private_handle_t *>(handle)->getStride();
}
```

9.4.2 Supported buffer formats

The list of currently supported buffer formats, mostly used in Camera HAL (as defined in `HwCaps::kAvailableFormats[]` and `HwManager::bufferFormatConvArray[]`):

- `HAL_PIXEL_FORMAT_IMPLEMENTATION_DEFINED`

Opaque buffer type, translates to internal platform types depending on buffer usage flags.

Note: For FPGA-PC platforms, opaque buffer is translated using following code:

```
if ((usage & GRALLOC_USAGE_HW_VIDEO_ENCODER) ||
    ((usage & GRALLOC_USAGE_HW_CAMERA_ZSL) ==
     GRALLOC_USAGE_HW_CAMERA_ZSL)) {
    // YUV buffers used by Camera output - Encoder input (NV21)
    format = HAL_PIXEL_FORMAT_YCbCr_420_888;
} else if (usage & (GRALLOC_USAGE_HW_TEXTURE |
                     GRALLOC_USAGE_HW_COMPOSER |
                     GRALLOC_USAGE_HW_FB)) {
    // RGB buffers used by Camera / Display output.
    format = HAL_PIXEL_FORMAT_RGBA_8888;
} else {
    ALOGE("%s: Not expected buffer usage (%#x)!", __FUNCTION__, usage);
    return -EINVAL;
}
```

If new platforms start using YUV buffers for preview, this code would need customization.

- HAL_PIXEL_FORMAT_YCrCb_420_SP
ISP CI format used : YVU_420_PL12_8
- HAL_PIXEL_FORMAT_YCbCr_420_888
ISP CI format used : YUV_420_PL12_8

Note: Please note the chroma order which is different comparing to YCrCb_420_SP

- HAL_PIXEL_FORMAT_YCbCr_422_SP
ISP CI format used : YUV_422_PL12_8
- HAL_PIXEL_FORMAT_RGBA_8888, HAL_PIXEL_FORMAT_RGBX_8888
ISP CI format used : BGR_888_32
- HAL_PIXEL_FORMAT_RGB_888
24 bits RGB, not used in camera HAL.

Warning: Due to the issue in gralloc_alloc() api, which requires to return buffer stride in pixels, rather than in bytes, and the fact that ISP hardware requires the stride to be aligned to 64 Bytes, the use of 24 bit buffers in camera HAL has been disabled. The issue causes incorrect calculation of final aligned stride for some specific buffer resolutions, as in example for 720x576 below:

w	h	stride [B]	stride aligned [B]	stride [px]
1280	720	3840	3840	1280
720	576	2160	2176	725.3(3)

As shown, even legal resolutions can cause aligned stride to be non integer.

- HAL_PIXEL_FORMAT_RAW_SENSOR
ISP CI format used : BAYER_RGGB_10
- HAL_PIXEL_FORMAT_RGB_565
Used only for framebuffer allocations, not used in ISP camera HAL.
- HAL_PIXEL_FORMAT_BLOB
JPEG image buffer of width=size of buffer, height=1

9.5 Library configuration

Many aspects of the Camera HAL library can be customized to suit the capabilities of the target hardware platform.

For example, every hardware platform may be equipped with:

1. Sensor hardware * Various supported sensor resolutions * Different framerates * Variable or fixed focus * Phase detection autofocus * Optional flash light * RGB + infrared (IR)
2. Hardware JPEG encoder
3. Hardware blitter module with image scaling capability or GPU.

The following list describes quickly how to configure and extend the Camera HAL for specific target platform.

9.5.1 Sensor

Sensor driver

Sensors expose control interfaces which most probably need custom driver to be used within Imagination Technologies ISP camera stack. This driver must be written as described in chapter *Sensor driver* (page 34) from *Platform Integration Guide* (page 27).

Calibration

As next step, to obtain best output quality from camera, the calibration data should be prepared. The standalone calibration procedure has been described in *User Tuning Guide* (page 130). The method of calibration while running Android can be referred in *Android Appendix: Live Tuning* (page 383).

Camera instance

After completing calibration the camera instance, which will use the new sensor, must be chosen.

Note: Most commonly, mobile Android devices support no more than 2 separate camera instances, **front camera** and **back camera**.

The association between sensor and specific camera instance is currently done at build-time, and requires modification of `FelixDefines.mk` script.

The configuration consists of following sensor related fields:

- `CAM0_SENSOR` The name for sensor used by **back** camera
- `CAM1_SENSOR` The name for sensor used by **front** camera
- `CAMx_PARAMS` The absolute path to the calibration file
- `CAMx_FLIP_SENSOR` The sensor flip. The possible values are:
 - `SENSOR_FLIP_NONE`
 - `SENSOR_FLIP_HORIZONTAL`
 - `SENSOR_FLIP_VERTICAL`
 - `SENSOR_FLIP_BOTH`
- `CAMx_DG_FILENAME` The source file for internal data generator, used only if `CAMx_SENSOR := "DATA_GENERATOR"`

Note: Change `CAMx` to `CAM0` for **back** and `CAM1` for **front** camera instance.

Warning: DATA_GENERATOR and CAMx_DG_FILENAME support is legacy therefore provided in Camera HAL on ‘as is’ basis. Initially, these parameters were used in simulator environments for the purpose of HAL internal development and supported only external data generator. Enabling data generator requires setting both FELIX_DATA_GENERATOR and possibly ANDROID_EMULATOR in *FelixDefines.mk* to true.

Please find the examples of build-time sensor configuration below.

1. Two sensors, OV4688 sensor is used by back camera and flipped horizontally and vertically, and AR330 sensor is connected to front camera instance with no flipping.

```
CAM0_SENSOR := "OV4688"
CAM0_PARAMS := "/data/felix/FelixDemoParams11.txt"
CAM0_SENSOR_FLIP := SENSOR_FLIP_BOTH
CAM1_SENSOR := "AR330"
CAM1_PARAMS := "/data/felix/ISPC2_AR330_telescope.txt"
```

2. Front camera only, horizontal flip

```
CAM1_SENSOR := "AR330"
CAM1_PARAMS := "/data/felix/ISPC2_AR330_telescope.txt"
CAM1_SENSOR_FLIP := SENSOR_FLIP_HORIZONTAL
```

Sensor capabilities

Finally, static camera info, reported by HAL to the Android camera framework, has to be aligned with specific sensor hardware capabilities.

In order of customizing the hardware capabilities depending on the sensor, the following files and methods have to be reviewed and/or edited accordingly:

- HwCaps.cpp
- FelixCamera::initLensCapabilities()
- FelixCamera::initSensorCapabilities()

Note: The implementation of HwCaps.cpp provides example configuration of two sensor types, OV4688 and AR330.

HwCaps.cpp contains hardcoded arrays of available output resolutions, framerates and sensor modes, used to set sensor in different, more specialized operating mode for some specific resolutions and framerates.

The primary purpose of switching of sensor mode is the high speed capture capability which needs special sensor configuration and possibly clocking, incompatible with default capture mode.

Warning: In current state, Camera HAL does not support dynamic switching sensor modes between camera sessions, therefore the SensorResCaps_t::sensorMode field is not used anywhere in HAL. The default sensor mode has been hardcoded to 0 in *FelixCameraHAL::FelixCameraHAL()*.

FelixCamera::initLensCapabilities() and *FelixCamera::initSensorCapabilities()* contain hardcoded camera capabilities metadata.

9.5.2 Image scaler

The current Camera HAL provides example, software based downscaler. This implementation is enabled by default at build-time because `USE_HARDWARE_SCALER` preprocessor define has not been defined in `ScalerSw.cpp`. The code below shows the current scaler factory method to be customized:

```
ImgScaler& ImgScaler::get() {
    #ifdef USE_HARDWARE_SCALER
    #error Hardware scaler not configured!
        return ScalerHw::getInstance();
    #else
        return ScalerSw::getInstance();
    #endif
}
```

The custom implementation of scaler must at least be able to handle non-stalling (that is not causing increase of min. frame duration) downscale operation of the following pixel formats:

- `HAL_PIXEL_FORMAT_YCrCb_420_SP`
- `HAL_PIXEL_FORMAT_YCbCr_420_888`
- `HAL_PIXEL_FORMAT_YCbCr_422_SP`
- `HAL_PIXEL_FORMAT_RGBA_8888`
- `HAL_PIXEL_FORMAT_RGBX_8888`
- `HAL_PIXEL_FORMAT_RGB_888`

Note: If the hardware scaler had support for up-scaling, it could be possibly used to enable digital zoom feature in the Camera HAL. Though, using separate up-scaling step in the capture processing would require changes in the internal management of stream buffers in HAL and add additional processing latency.

9.5.3 Jpeg encoder

The current Camera HAL provides example, software based Jpeg encoder class. Similar to image scaler, Jpeg encoder implementation is chosen at build-time in the following factory method:

```
JpegEncoder& JpegEncoder::get() {
    #ifdef USE_HARDWARE_JPEG_COMPRESSOR
    #error Hardware JPEG not configured!
        mInstance = &JpegEncoderHw::getInstance();
    #else
        mInstance = &JpegEncoderSw::getInstance();
    #endif
        return *mInstance;
}
```

Stream capabilities

`HwCaps.cpp` contains the available resolutions and minimum frame durations of the JPEG output streams. In case of Jpeg output streams, Android camera framework treats them as possibly

stalling, that is the capture result is allowed to be returned not in real-time (jpeg processing duration can be longer than preview frame duration).

Note: A stall duration is how much extra time would get added to the normal minimum frame duration for a repeating request that has streams with non-zero stall.

The supported duration of Jpeg stalling stream to be advertised with static camera info, is defined in `HwCaps.cpp`. In case of software encoder, this had been hardcoded to 500ms.

Below is the example definition of stalling streams supported in OV4688 sensor configured to maximum native resolution of 1280x720.

```
static const SensorResCaps_t kAvailableJpegImageParams_OV4688[] = {
    { 1280, 720, 0, 500*MSEC }, // 16:9, 0.5sec jpeg processing time in SW mode
    { 640, 480, 0, 500*MSEC }, // 4:3
    { 320, 240, 0, 500*MSEC } // 4:3
};
```

This value may need to be changed in case of hardware compressor implementation.

9.5.4 Default log level

The very verbose logging from Felix Camera HAL can be completely suppressed by editing `FelixDefines.mk` and changing the variable `FELIX_LOCAL_CFLAGS` to the form shown below:

```
FELIX_LOCAL_CFLAGS := -DLOG_NDEBUG=1
```

Note: `FELIX_LOCAL_CFLAGS` is propagated down to all `Android.mk` files executed by command

```
$ mmm hardware/img/v2500
```

It can be used to add other build-time customizations as well.

9.6 Testing Camera HAL

9.6.1 Native Camera HAL test suite

Android AOSP provides a native test application which purpose is to execute a set of low-level API tests on Camera HAL library. Although, not considered as acceptance tests, these tests can be used as the entry gate for HAL to run more detailed exhaustive CTS test suite.

The following steps are required to build and upload the suite to running target system.

```
$ mmm hardware/libhardware/tests/camera2
$ adb push $ANDROID_PRODUCT_OUT/data/nativetest/camera2_test/camera2_test /data/
$ adb chmod 770 /data/camera2_test
```

Running the tests:

```
$ adb shell /data/camera2_test
```

9.6.2 Android CTS

Acceptance suite for Camera HAL consists of many test cases, defined in *android.hardware* test package. Though, by running this package directly, many other, not camera related tests will be run as well (for example hundreds of sensor tests). To be able to run camera tests and filter out the tests which are out of our interest, the custom test plans have been produced and provided within the release package. The plans can be found in `./Android/CTS/` folder and are:

- `android.hardware.camera-4.4.xml` - to be used in Kitkat CTS
- `android.hardware.camera-5.1.xml` - Lollipop CTS

To make the plans available, one must copy the respective xml file to `android-cts/repository/plans/` folder.

Listing and executing the plan within Lollipop `cts-tradefed` shell is done by the following commands:

```
$ pwd
~/workspace/cts/5.1.1_r2/android-cts
$ ./tools/cts-tradefed
Android CTS 5.1_r1
cts-tf > list plans
CTS-DEQP
CTS-staging
android.hardware.camera-5.1
CTS-kitkat-small
AppSecurity
CTS-l-tests
CTS-stable
CTS-kitkat-medium
CTS-No-Media-Stream
CTS-webview
CTS-TF
Harmony
SDK
Java
CTS-media
CTS-mediatress
CTS
CTS-hardware
VM-TF
PDK
CTS-flaky
Android
cts-tf > run cts --plan android.hardware.camera-5.1
...
```

9.6.3 Android Camera Imaging Test Suite / CTS Verifier

The below is the excerpt from ITS test suite documentation:

The ITS is a framework for running tests on the images produced by an Android camera. The general goal of each test is to configure the camera in a desired manner and capture one or more shots, and then examine the shots to see if they contain the expected image data. Many of the tests will require that the camera is pointed at a specific target chart or be illuminated at a specific intensity.

ITS tests are written in python using `scipy`, `numpy`, and `opencv` frameworks, and require special physical setup using a tripod, light box, LED lamp and a grey card with 18% reflectance.

The test harness and documentation can be found in AOSP source tree under `$ANDROID_BUILD_TOP/cts/apps/CameraITS`.

Chapter 10

Android Build Instructions

These are some basic instructions to get started building the V2500 ISP drivers for Android.

The instructions cover building the low level Linux drivers using an Android toolchain and kernel, it then also includes instructions on building the Android tree with the V2500 Camera HAL component. These instructions may require adjustments based on the platform in use, where possible this is highlighted.

It is recommended to have read the *Getting Started Guide* (page 6) before attempting to build for Android. Note that building for any platform will need some code modification as explained in the *Platform Integration Guide* (page 27).

If using Imagination's GPU the “vision libraries” should also be provided (as an external package). The android package contains options to enable integration of those libraries. More information is available in the *Vision Libraries Usage Instructions* (page 336).

10.1 Build instructions

These instructions are targeted to build the emulator version of android using our CSIM in a virtual machine to replace the actual HW. Building instruction for a real HW should be very similar if not identical.

10.1.1 Create a toolchain

Building the drivers for a specific Android system requires some tools specific to the platform being used. Building the tools/test apps which come with the drivers requires a compiler which supports STL which the prebuilt compiler that comes with the Android tree does not support. Because of this we would recommend using a standalone toolchain. This can either be downloaded from an outside source (e.g. CodeSourcery) or can be built from the Android NDK using

```
$ Android_root_path/ndk/build/tools/make-standalone-toolchain.sh \
--platform=android-18 --install-dir=/tmp/my-android-toolchain
```

Make sure to use full path both for `make-standalone-toolchain.sh` file and for `--install-dir` argument.

10.1.2 Setup Toolchain file

The V2500 build system needs to be made aware of this toolchain in order to build using the correct compiler and headers. To do this you must provide a customised CMake file that sets up some variables according to the desired toolchain. A sample one for an arm system is provided with this document. Edit or duplicate the sample and setup the variables `CMAKE_C_COMPILER` and `CMAKE_CXX_COMPILER` in the toolchain cmake file.

10.1.3 Configure V2500 Module using CMake

These instructions are relative to the delivered package sources as described in the *Getting Started Guide* (page 6). Note that the package required for the kernel module is separated from the Android package.

```
$ cd V2500_package_root/cmake
$ cmake -DLINUX_KERNEL_BUILD_DIR=<Android kernel root path> \
-DCMAKE_TOOLCHAIN_FILE=<path to Toolchain-arm.cmake> -DANDROID_BUILD=ON \
-DFORCE_32BIT_BUILD=ON -DCI_EXT_DATA_GENERATOR=ON ..
```

This corresponds to the CMake step in the *Getting Started Guide* (page 6), but with additional parameters defining the Android Kernel headers and a toolchain to use.

Note: The `CI_EXT_DATA_GENERATOR` switch is optional as not supported by real HW. But when using the emulator with the simulator it is used to fake the sensor.

When trying to cross compile only the kernel module from another computer, it is possible that compiling the test applications is not needed (or even fails if ION is missing on the system's user-space includes). The building of the test applications can be disabled by adding `-DBUILD_TEST_APPS=OFF` to the CMake flags.

Additional important cmake flags used for development FPGA card are:

```
-DCI_DEVICE=PCI -DCI_ALLOC=PAGEALLOC -DFPGA_REF_CLK=40
```

For build systems which set a different location for kernel objects instead of default kernel sources directory, additional parameter has to be used:

```
-DLINUX_KERNEL_SOURCE_DIR=<Android kernel sources path>
```

This is needed because Felix.ko build for Android needs to find specific ION headers, and it can't be found in `LINUX_KERNEL_BUILD_DIR` (at least for kernels up to 3.10).

For example, to build Felix.ko for Android-x86 distribution for FPGA machine, the following commands shall be issued:

```
$ mkdir output_x86 && cd output_x86
$ cmake -DCI_DEVICE=PCI -DCI_ALLOC=PAGEALLOC -DANDROID_BUILD=ON \
-DFORCE_32BIT_BUILD=ON \
-DLINUX_KERNEL_BUILD_DIR=$ANDROID_PRODUCT_OUT/obj/kernel/ \
-DLINUX_KERNEL_SOURCE_DIR=$ANDROID_BUILD_TOP/kernel \
-DCI_EXT_DATA_GENERATOR=ON \
-DBUILD_TEST_APPS=OFF -DFPGA_INSMOD_RECLOCK=ON -DCI_HW_REF_CLK=40 \
.../DDKSource
$ make CI_Kernel_KO
```

10.1.4 Build V2500 kernel module

It is important to know that the kernel module expects the Linux kernel being used to have modules enabled. To do this modify the kernel.config file and set

```
CONFIG_MODULES=y
```

then rebuild the kernel.

```
$ make ARCH=arm CROSS_COMPILE=<Cross compiler path>
```

This corresponds to the *Kernel-module build (GNU/Linux)* (page 15) in the Getting Started Guide, with additional parameters defining the target architecture and compiler to use. Note that you should specify the architecture you are using, arm is used here as an example.

Notes

Toolchain not used

If cmake was used to generate a build folder before the inclusion of the toolchain file described above then you will need to delete the CMakeCache.txt file. The easiest way to do this is by simply deleting the cmake folder and starting again.

x86 build

If you are building for an x86 platform then you will not need a cross compilation toolchain, you simply need to specify the kernel build directory for your Android kernel.

10.2 Building Android Camera HAL

10.2.1 Patch the Android tree

The Android Camera HAL component is built as part of the Android tree. It is dependent on having below patches applied to the tree:

- `Android/patches/android/kitkat` – patches common for all targets
- `Android/patches/android/kitkat/goldfish` – patches only for Android emulator (goldfish)
- `Android/patches/android/kitkat/x86` – patches only for android-x86

Apply these using:

```
$ cd Android_root_path
$ patch -p1 < V2500_package_root/Android/patches/android/kitkat/<patch_name>.patch
```

Note: Patch names convention is related to package root path, e.g. `device-generic-x86_imgion.patch` should be applied in dir `device/generic/x86`. There is also one manual correction needed. File `device-generic-x86_Felix_module.patch` contains path to kernel driver, which is specific for build PC used.

10.2.2 Link Felix Camera HAL to Android tree

To keep the modifications to the Android tree to a minimum we recommend keeping the camera HAL component in a separate location and linking it into the android tree using a symbolic link e.g.

```
$ cd Android_root_path/hardware
$ mkdir -p img && cd img
$ ln -s V2500_package_root v2500
```

10.2.3 Build Android Tree

It should now be possible to build the Android tree using

```
$ cd Android_root_path
$ source build/envsetup.sh
$ lunch aosp_arm-eng
$ make -j 5
```

Instead of `lunch aosp_arm-eng` choose the number which describes your platform i.e. `lunch <number>`

Run `lunch` without options to see available platforms.

10.2.4 Using GPU for Image Stabilization and Face Detection

If your hardware contains Imagination's PowerVR G6230 GPU you can enable additional image stabilisation and face detection hardware acceleration. To be able to do so, you need Imagination's Vision Library present in your Android source tree (`Android_root_path/hardware/img/gpu`). Library activation can be done by C pre-processor flag `USE_GPU_LIB`, located in `FelixDefines.mk` file.

Please note that for proper Vision Library compilation you need standalone NDK toolchain (look at <https://developer.android.com/tools/sdk/ndk/index.html> for more information). You should use the same toolchain which was generated in the chapter *Create a toolchain* (page 325). The path to generated toolchain should be provided in `HALDefines.mk` file, located in Vision Library dir.

10.3 Example: building V2500 software for android-x86

This section is an example on how we build the HAL for the android-x86 project. We use the android-x86 to test the HAL with our FPGA system in a desktop computer. The steps are similar with android AOSP.

10.3.1 Getting android-x86 sources

Get the sources for android-x86. We used version `android-x86-4.4-r1 (kitkat-x86)` for this example.

10.3.2 Prepare android-x86 kernel

Android kernel is part of the android-x86 source tree. It's being built automatically together with Android build. Kernel source tree can be found under following location:

- <android-x86_root_dir>/kernel

This kernel needs to be patched using below file:

- Android/patches/kernel/x86/android_kernel_3_10.patch

Default configuration of android-x86 kernel should be also changed using below command:

```
$ cp V2500_root_path/Android/patches/kernel/x86/felix_defconfig_3_10 \
<android-x86_root_path>/kernel/arch/x86/configs/android-x86_defconfig
```

The kernel also needs an additional file if the i2c device will be used (our implementation uses `i2c-dev.h`). Add it if required as explained in the *Compiling errors* (page 331).

10.3.3 Apply android-x86 patches

Apply patches from below locations:

- Android/patches/android/kitkat
- Android/patches/android/kitkat/x86

e.g.:

```
$ cd frameworks/av
$ patch -p1 < \
V2500_root_path/Android/patches/android/kitkat/frameworks-av_stagefright_recording.patch
```

10.3.4 Link the V2500 DDK sources

```
$ cd Android_root_path/hardware
$ mkdir -p img && cd img
$ ln -s V2500_root_path v2500
```

10.3.5 Configure and build V2500 kernel module

Configure V2500 module using CMake:

```
$ mkdir -p <DDKSource_out_path>
$ cd V2500_root_path/cmake
$ cmake -DCI_DEVICE=PCI -DCI_ALLOC=PAGEALLOC -DANDROID_BUILD=ON \
-DFORCE_32BIT_BUILD=ON \
-DLINUX_KERNEL_BUILD_DIR=<android-x86_out_path>/target/product/x86/obj/kernel/ \
..
```

Build V2500 kernel module:

```
$ make
```

Successful build should generate below kernel module:

- V2500_package_root/cmake/CI/felix/felix_lib/km/Felix.ko

Alternatively it could be installed and retrieved from the install location as described in the *Main Libraries* (page 10) of the Getting Started Guide.

10.3.6 Configure V2500 Android Camera HAL

Edit and set needed features of Camera HAL in below file:

- V2500_package_root/FelixDefines.mk

Most important setting is to select proper camera sensor. Set proper path to V2500 kernel module in file:

- Android_root_path/device/generic/x86/device.mk

```
FELIX_MODULE_PATH := V2500_package_root/cmake/CI/felix/felix_lib/km/Felix.ko
```

10.3.7 Build android-x86

```
$ cd Android_root_path
$ export OUT_DIR_COMMON_BASE=path_to_output_dir
$ source build/envsetup.sh
$ lunch 5 //android_x86-eng
$ make
```

10.3.8 Build Legacy Camera application

To be able to use camera without GPU hardware acceleration, LegacyCamera application is used.

```
$ mmm Android_root_path/packages/apps/LegacyCamera
```

10.3.9 Prepare Android x86 images for GRUB

Successful build generates Android x86 image files under following location:

- Android_root_path/target/product/x86

Following files are needed to boot Android using GRUB:

- ramdisk.img
- kernel
- system.img
- initrd.img

initrd.img file is not automatically generated and following command needs to be executed:

```
$ cd Android_root_path
$ make out/target/product/x86/initrd.img
```

In case initrd.img needs modifications or some analysis, this file is generated from following directory:

- `Android_root_path/bootable/newinstaller/initrd/`

10.3.10 Configure GRUB to run Android x86 image

Add below GRUB menu item in `/etc/grub.d/40_custom` file

```
menuentry 'Android x86' --class gnu-linux --class gnu --class os {
    echo "Loading Android x86 kernel 3.4.67+"
    gfxmode $linux_gfx_mode
    linux android_out_path/kernel androidboot.hardware=android_x86 vmalloc=768M nomodeset
    initrd android_out_path/initrd.img
}
```

It's important to pass `nomodeset` kernel parameter which disables hardware GPU acceleration. In such case Android uses regular frame buffer (instead of DRM) which can be shared with camera buffers. To select needed resolution, `vga=kernel` parameter should be set with one of below values:

Colour depth	640x480	800x600	1024x768	1280x1024	1400x1050	1600x1200
8 (256)	769	771	773	775		
15 (32K)	784	787	790	793		
16 (65K)	785	788	791	794	834	884
24 (16M)	786	789	792	795		

Update grub configuration with following command (as root)

```
$ update-grub
```

10.3.11 Compiling errors

Some Android versions do not compile due to missing Linux kernel headers (i.e. `i2c-dev.h`). In such a case there is a need to update kernel headers for Bionic. To update bionic headers, below steps should be executed:

1. Copy missing header to `external/kernel-headers/original` directory
2. Enter `bionic/libc/kernel` directory
3. Execute `tools/update_all.py` script

10.3.12 Android-x86 tips and tricks

Persistent /data partition

To make Android data persistent, the best is to create `/data` directory on the hard disk. Creating data directory in the same location where the kernel and `initrd.img` file exist, cause automatic binding of data directory into Android system. It's done by init process from `initrd.img` file.

Mount Ubuntu partition under android-x86

It's possible to mount hard disk partitions under `android-x86`. Hard disk partitions are detected under `/dev/block/sd*` location. Example of mounting Ubuntu partition:

```
$ mount -t ext4 /dev/block/sda2 /data/ubuntu
```

ADB remote connection

ADB is able to connect using TCP/IP networking. Just execute below command:

```
$ adb connect IP_address
```

10.4 Example: Building Android Camera HAL for AOSP Lollipop

Warning: This is an internal only section.

This section is an example of how we build the HAL for the Android Lollipop. We use the AOSP sources and test the HAL with PC containing FPGA card.

10.4.1 Set up environment variables

```
$ export ANDROID_ROOT=/path/where/src/will/be/kept
$ export FELIX_PATH=/path/to/Android_DDK_HW_2.1
$ export ANDROID_PRODUCT_OUT=$ANDROID_ROOT/out/target/product/pc
```

10.4.2 Getting Lollipop sources

- Main repo

```
$ cd $ANDROID_ROOT
$ repo init -u https://android.googlesource.com/platform/manifest \
    -b android-5.1.1_r4
$ cd .repo/manifests
$ git checkout d3c00c789b3ab2f87c88c6d7b516d8cc8a41efcf default.xml
```

- Kernel sources

```
$ cd $ANDROID_ROOT
$ patch -N -p0 < \
    $FELIX_PATH/Android/patches/android/lollipop/aosp_Lollipop_kernel_3.10.0_local_manifest.patch
$ repo sync -c
```

10.4.3 Copy Imagination specific files

```
$ cd $ANDROID_ROOT
$ mkdir -p device/img
$ cp -fa i$FELIX_PATH/Android/device/5.0/pc device/img/.
```

10.4.4 Patch kernel

```
$ cd $ANDROID_ROOT

$ cat $FELIX_PATH/Android/patches/kernel/x86/ion-staging-x86.patch \
| patch -N -p1
$ cat $FELIX_PATH/Android/patches/kernel/x86/ion-staging-device-create.patch \
| patch -N -p0 -d kernel/common

$ cp -fv $FELIX_PATH/Android/patches/kernel/x86/felix_defconfig_3.10.0_x86_L \
$ANDROID_ROOT/kernel/common/arch/x86/configs/felix_defconfig
```

10.4.5 Configure and build kernel

```
$ cd $ANDROID_ROOT
$ make -C kernel/common ARCH=x86 CROSS_COMPILE= felix_defconfig

$ make -C kernel/common ARCH=x86 CROSS_COMPILE= -j8 bzImage
$ make -C kernel/common ARCH=x86 CROSS_COMPILE= -j8 modules
$ make -C kernel/common ARCH=x86 CROSS_COMPILE= M=$FELIX_PATH/Android/imgion modules
```

10.4.6 Update bionic c-lib

```
$ cp -fv $ANDROID_ROOT/kernel/common/include/uapi/linux/i2c-dev.h \
$ANDROID_ROOT/external/kernel-headers/original/uapi/linux/.
$ cd $ANDROID_ROOT/bionic/libc/kernel/
$ python tools/update_all.py
```

10.4.7 Link Felix HAL code into the Android tree

```
$ cd $ANDROID_ROOT/hardware/
$ mkdir -p img
$ cd img/
$ ln -s $FELIX_PATH V2500
```

10.4.8 Build the driver

```
$ export DDK_SRC=$ANDROID_ROOT/hardware/img/V2500/DDKSource
$ export CMAKE_DIR=$DDK_SRC/obj
$ mkdir -p $CMAKE_DIR
$ cd $CMAKE_DIR
$ rm -rf *
$ cmake -DCI_DEVICE=PCI -DCI_ALLOC=PAGEALLOC -DANDROID_BUILD=ON \
-DFORCE_32BIT_BUILD=ON -DLINUX_KERNEL_BUILD_DIR="$ANDROID_ROOT/kernel/common/" \
-DCI_EXT_DATA_GENERATOR=OFF -DBUILD_TEST_APPS=OFF -DFPGA_REF_CLK=40 \
-DFPGA_INSMOD_RECLOCK=ON -DLINUX_KERNEL_SOURCE_DIR="$ANDROID_ROOT/kernel/common" ..

$ make

$ ln -fvs $CMAKE_DIR/CI/felix/regdefs/regdefs $DDK_SRC/CI/felix/regdefs/include
$ ln -fvs $CMAKE_DIR/CI/felix/regdefs/vhc_out $DDK_SRC/CI/felix/regdefs/vhc_out
```

10.4.9 Apply patches

```
$ cd $ANDROID_ROOT/frameworks/av/
$ patch -N -p1 < \
    $FELIX_PATH/Android/patches/android/lollipop/frameworks-av-api1-rgb-preview-stream-format.patch
$ patch -N -p1 < \
    $FELIX_PATH/Android/patches/android/lollipop/frameworks-av_stagefright_recording.patch
$ cd $ANDROID_ROOT/hardware/libhardware/
$ patch -N -p1 < \
    $FELIX_PATH/Android/patches/android/lollipop/hardware-libhardware_felix_gralloc_camera_default.patch
```

10.4.10 Clear old generated files

```
$ rm -rf $ANDROID_PRODUCT_OUT/obj/SHARED_LIBRARIES/*felix*
```

10.4.11 Choose camera sensor

Edit \$FELIX_PATH/FelixDefines.mk and choose sensor OV4688 or AR330.

10.4.12 Build Android

```
$ cd $ANDROID_ROOT
$ rm -fv $ANDROID_ROOT/out/target/product/pc/system/build.prop
$ source build/envsetup.sh
$ lunch $ANDROID_TARGET
$ make -j4
$ echo "ro.kernel.qemu=1" >> out/target/product/pc/system/build.prop
$ echo "ro.kernel.gles=0" >> out/target/product/pc/system/build.prop
$ mmm packages/apps/LegacyCamera
$ mmm packages/apps/Gallery
$ mmm frameworks/av/cmds/stagefright
```

10.4.13 Install kernel modules

```
$ cd $ANDROID_ROOT
$ cp kernel/common/arch/x86/boot/bzImage $ANDROID_PRODUCT_OUT/vmlinuz
$ make -C kernel/common ARCH=x86 CROSS_COMPILE= \
    INSTALL_MOD_PATH=$ANDROID_ROOT/out/target/product/pc/system
$ modules_install
$ make -C kernel/common ARCH=x86 CROSS_COMPILE= M=$FELIX_PATH/Android/imgion \
    INSTALL_MOD_PATH=$ANDROID_ROOT/out/target/product/pc/system
$ modules_install
$ cp -fv $CMAKE_DIR/CI/felix/felix_lib/km/Felix.ko \
    $ANDROID_PRODUCT_OUT/system/lib/modules/.
```

10.4.14 Quick android image regeneration

```
$ cd $ANDROID_ROOT
$ source build/envsetup.sh
$ lunch $ANDROID_TARGET
$ make snod
```

10.4.15 Finalizing

- Copy android image files to common directory

```
$ mkdir /boot/android
$ cp $ANDROID_PRODUCT_OUT/system.img /boot/android/.
$ cp $ANDROID_PRODUCT_OUT/ramdisk.img /boot/android/.
$ cp $ANDROID_PRODUCT_OUT/vmlinuz /boot/android/.
$ cp $FELIX_PATH/Android/BuildScripts/lollipop/AOSP/initrd.img /boot/android/.
```

- In grub.cfg add following entry

```
set gfxpayload=1280x1024x16

menuentry "Android-Lollipop" {
    linux /boot/android/vmlinuz root=/dev/ram0 androidboot.hardware=pc nomodeset
    initrd /boot/android/initrd.img
}
```

- Reboot PC and start Android
- Please remember to update camera sensor config file in Android system. Connect Android using adb and send a proper file to device

```
$ adb push $FELIX_PATH/sensorConfig/ov4688/FelixSetupOV4688.txt \
    /data/felix/FelixDemoParams11.txt
```

Note, that target path/file name can be changed in FelixDefines.mk.

Chapter 11

Vision Libraries Usage Instructions

The CameraGPU library is a wrapper for a set of GPU based image processing functions, intended for real time processing of video data to allow advanced functionality for the Android Camera HAL. These functions are optimised for the PowerVR architecture and provide

- Face Detection and
- Image Stabilisation
- Rolling Shutter Correction

The CameraGPU library is for Android and wrappers the libPVRVision Linux/Android libraries. This document comes with the CameraGPU library and describes how to build this library. The *V2500 DDK Android Build Instructions document* (page 325) describes how to integrate these functions with the V2500 Camera HAL.

Vision stabilisation and rolling shutter are combined in one library `libPVRVisionStabilisation.so`. Face detection is performed by `libPVRFaceDetect.so`. Both libraries use `libPVRVisionCore`.

11.1 Code structure and usage

The vision libraries code needs to be placed in:

```
$ Android_root_path/hardware/img/gpu
```

Symbolic link `gpu` to `vision_libraries` folder will be sufficient.

```
$ cd Android_root_path/hardware/img  
$ ln -s VisionLibrariesPath/vision_libraries gpu
```

11.2 Build instructions

11.2.1 Setup environment

In android root folder:

```
$ . build/envsetup.sh
$ lunch android_x86-eng
```

11.2.2 Apply patches

The patches which need to be applied for cameragpu library to work properly are in vision_libraries/patches folder and can be accessed through symbolic link gpu.

build_add_cascade.bin

The patch `build_add_cascade.bin` allows to copy `cascade.bin` file into `/system/bin` of `system.img` during android compilation. The `cascade.bin` is used by face detection library.

```
$ croot
$ cd build
$ patch -p1 < ../hardware/img/gpu/patches/build_add_cascade.bin.patch
$ croot
$ make
```

The `make` command will build android and copy `cascade.bin` into appropriate location in order to be included in `system.img`.

11.2.3 Build libraries

In android root folder

```
$ make libcameragpu
```

11.2.4 Use CameraGPU

In order to use `cameragpu` functionality:

- modify code
- modify `Android.mk`

Code

Include `cameragpu_api.h` and calls to CameraGPU functions in your code.

```
#include "cameragpu_api.h"
```

Define pointer to `camera_gpu` object;

```
camera_gpu *mGPU;
```

Create `cameragpu` class object. Provide the size of the input image.

```
mGPU = new camera_gpu(800, 600);
```

Call API functions. There are two functional areas grouped by CameraGPU API

- image stabilisation/rolling shutter
- face detection

Image stabilisation/rolling shutter

```
status_t gpuGetStabilizedFrame( int64_t           inTimeStamp,
                                uint8_t const * const inYUV,
                                uint8_t      * const outYUV);
```

The inputYuvFrame and outputYuvFrame can be the same pointer.

Face detection

Obtain max possible detected faces. This is needed to allocate space for array of detected faces for early allocation.

```
status_t gpuGetMaxFaces(uint8_t * const outNumFaces);
```

Obtain actual number of detected faces.

```
status_t gpuGetNumFaces( uint8_t * const outNumFaces);
```

Detect faces and obtain pointer to location where result is stored.

```
status_t gpuDetectFaces( uint8_t const * const inYUV);
status_t gpuGetMetaData( camera_frame_metadata_t **mdata);
```

Detect faces providing the storage for location where results will be stored.

```
status_t gpuDetectFaces( uint8_t const * const inYUV,
                        camera_frame_metadata_t *mdata);
```

Android.mk

Provide path to cameragpu_api.h.

```
GPU_TOP:=$(ANDROID_BUILD_TOP)/hardware/img/gpu
LOCAL_C_INCLUDES += $(GPU_TOP)/include
```

Link to CameraGPU library.

```
LOCAL_SHARED_LIBRARIES += libcameragpu
```

Chapter 12

Frequently Asked Questions

This section contains the frequently asked questions and has links to other parts of this documentation that solves them.

How to read/write hardware registers for debug purposes: This is covered by the low level driver (CI) documentation: [Register access for debugging purposes](#) (page 170).

When using the high level library (ISPC) one can use or modify the `PrintGasketInfo()` and `PrintRTMInfo()` in `info_helper.cpp`.

Debugging an MMU page fault: See [CI Appendix: Debugging a Page Fault](#) (page 366).

Debugging a fail to acquire buffer (missed frame): See [CI Appendix: Debugging a Failed to acquire frame](#) (page 370).

Allocating buffers with specific size requirements (stride, memory layout): See CI documentation: [CI: Output sizes](#) (page 161) and [CI: Output layout](#) (page 162).

See ISPC documentation [ISPC: Output sizes](#) (page 197) and [ISPC: Output layout](#) (page 197).

Porting driver to a new platform: This is covered by the platform integration guide and is usually done in two steps:

- Integration of the device access and discovery ([Device Access](#) (page 28))
- Integration of the memory access ([Memory management](#) (page 30))

Implementing a new sensor driver: This is covered by the platform integration guide ([Sensor driver](#) (page 34)).

Chapter 13

Appendix: Errors troubleshooting

13.1 Building GUI tools: troubleshooting

13.1.1 Failure to find Qt or OpenCV

It is possible that Qt or OpenCV are not detected by CMake on your system. This can also be used if several versions of Qt or OpenCV are available on the build computer and a specific one has to be selected. To solve this the user is expected to specify the location to Qt and OpenCV using the standard CMake variables.

E.g. Qt not found:

```
CMake Error at CMakeLists.txt:24 (message):  
  qmake is not in your path, try defining QT_BINARY_DIR to your QT  
  installation /bin path
```

E.g. OpenCV not found:

```
CMake Error at GUI/tuning/FindVisionTuning.cmake:9 (find_package):  
  Could not find module FindOpenCV.cmake or a configuration file for  
  package OpenCV.  
  
  Adjust CMAKE_MODULE_PATH to find FindOpenCV.cmake or set OpenCV_DIR to  
  the directory containing a CMake configuration file for OpenCV. The  
  file will have one of the following names:  
    OpenCVConfig.cmake  
    opencv-config.cmake
```

The variables to define are:

QT_BINARY_DIR: The location of qmake, e.g. on windows:

```
C:\QtSDK\Desktop\Qt\4.8.1\msvc2010\bin
```

E.g. on manual installation on linux:

```
/opt/Qt/5.4/gcc_64/bin/
```

QT_LIBRARY_CMAKE_DIR: The location of the cmake folder - if not specified it is guessed in cmake:

```
set(QT_LIBRARY_CMAKE_DIR ${QT_BINARY_DIR}/../lib/cmake)
```

This is the top level folder where Qt5Widgets, Qt5Concurrent and any other needed modules can be found.

OpenCV_DIR: The location of the `OpenCVConfig.cmake` of your OpenCV installation, e.g. on windows:

```
C:\opencv\opencv2.4.9\build
```

```
$ cmake ../DDKSource -DBUILD_GUI=ON -DQT_BINARY_DIR=<...> \
-DOpenCV_DIR=<...>
```

It is possible that some Qt installation under windows do not install the needed `.dll` and `.lib` in the same folder. In case of failure of the `FindQtD11` CMake function please ensure the `.dll` are in the same folder than the `.lib` (copy should be enough).

13.2 Running the GUI: troubleshooting

These are some potential run time errors that can occur if the system is not configured properly when running or compiling the tools.

13.2.1 OpenCV and Qt mismatch

If OpenCV was installed on the computer with GUI support and was compiled with a different version of Qt than the one the DDK uses failures may occur when running such as the cryptic message:

```
QMetaType::registerType: Binary compatibility break -- size mismatch for
type 'QPaintBufferCacheEntry' [1024]. Previously registered size 0 now
registering size 16.
```

To solve this problem download the recommended OpenCV library sources and compile them without Qt support. Then use the appropriate location at CMake (see [Standard CMake options](#) (page 22)) to use it instead of the system wide library.

This solution may require that you export the correct `LD_LIBRARY_PATH` on linux to be able to run the application.

13.2.2 Qt dll missing on windows

If some dlls are missing from the path of the application (including the environment variable `PATH`) the following message can occur when starting a GUI application:

```
The application was unable to start correctly (0xc000007b). Click OK to
close the application.
```

This is most likely due to fact that the following dlls not starting with the Qt5 prefix in the `QT_BINARY_DIR` (specified for compilation) are not available (copy them in your local folder or include their location in the system's `PATH`), for example:

- `icudt53.dll`

- icuin53.dll
- icuuc53.dll

13.2.3 No icons in VisionLive

On some systems the icons do show in the VisionLive or VisionTuning and Jpeg libraries errors can be seen on the stderr output. This is most likely caused by a different libJPEG used for Qt and OpenCV.

To fix it compile OpenCV without PNG support (-DBUILD_PNG=OFF).

13.3 Kernel module insertion errors

It is possible that insmod fails to initialise the driver. Here is a list of possible failures and how to check which one has occurred.

13.3.1 Invalid module

Insmod:

```
Error: could not insert module Felix.ko: Invalid module format
```

Dmesg: none

Reason: Most likely the kernel used to compile the module and the one currently used on the system are different.

13.3.2 Device not found

Insmod:

```
Success
```

But no device are available in /dev

Dmesg:

```
[757543.881595] FELIX driver initialisation
[757543.881634] imgfelix PCI board not found
```

Reason: The device was not found when registering the driver, therefore the initialisation did not occur. Ensure that the device is present on the system and verify that the mean of discovery uses the correct identifiers (e.g. by using lspci)

13.3.3 Device is incompatible

This is a variation of the previous problem, where a device was found but not considered compatible.

Insmod:

Success

But no device are available in /dev

Dmesg:

```
[ 6540.629206] FELIX driver initialisation...
[ 6540.629246] FELIX:KRN_CI_DriverCreate:1122 unsupported HW found!
Driver supports design 0 2.0 - found design 0 2.1
[ 6540.630972] FELIX shutting down HW
[ 6540.630975] FELIX:probeSuccess:415 felix driver creation failed
- returned 22
[ 6540.632706] Driver found but refused device!
[ 6540.652624] failed to request HW
[ 6540.652631] imgfelix 0000:01:00.0: PCI INT A disabled
[ 6540.652638] imgfelix: probe of 0000:01:00.0 failed with error -1
[ 6540.652660] imgfelix: PCI board not found
```

Reason: The device was found when registering the driver, but the discovered HW is not compatible with the driver (e.g. different register map or the discovered HW is not Felix).

13.3.4 Initialisation fails

The driver gets initialised by insmod correctly, the /dev is populated but the user side fails to open the device. In that case the first obvious thing to verify is that the user-side is trying to open the correct device (e.g. /dev/imgfelix0). It is also possible that the entry in /dev is protected (`ls -l` will show that). If it is the case udev rules should be added to automatically change the accessibility of the /dev entries (or a chmod done by root after insmod). Example of udev rule from `/etc/udev/rules.d/img.rules`:

```
KERNEL==”img*”,”MODE=0777”
```

The device name that the user-space is trying to open is available in `ci_api.c` (`FELIX_DEV` macro). The device name that the kernel-space creates is available in `ci_init_km.c` (`DEV_NAME` macro, /dev/ and device number will be added to it by the Linux kernel device registration).

Chapter 14

CI Appendix: Adding a new DebugFS counter

The DebugFS interface allows a direct read of some variables from a file in the `/sys/kernel/FELIX` folder (usually 1 file per variable).

To enable it the driver must be compiled with the cmake option `CI_DEBUG_FUNCTIONS` turned on (see [Getting Started Guide](#) (page 24)).

Note: The user needs to have root privileges to access `/sys/kernel/debug` unless the folder permissions were modified.

Most of the handling of the DebugFS variables is done in `kernel_src/ci_init_km.c`:

- Add the `struct dentry` pointer must be defined next to the others
- Add a macro containing the name of the entry file with `DEBUGFS_` prefix
- Add the variable that will be used a global variable
- `KRN_CI_ResetDebugFS()` should be updated to reset the variable to a default state
- `DebugFs_Create()` in `ci_init_km.c` should be modified so that the DebugFS entry is created into the FELIX folder

Other files may need modification too. If the variable should be used in CI (for example in the interrupt handler or to keep track of the number of triggered shot) `ci_debug.h` should be modified to include the previously defined variable as an extern. If the variable should be used in the DG then `dg_debug.h` should be modified to include the extern variable. Of course the files using this variable should check for the `CI_DEBUGFS` pre-processor macro before accessing the variable.

Chapter 15

CI Appendix: Frame Size Computation

This section will describe the arithmetic behind the frame sizes computation done by the FelixCommon functions:

- CI_ALLOC_RGBSizeInfo()
- CI_ALLOC_Raw2DSizeInfo()
- CI_ALLOC_YUVSizeInfo()

This section assumes that the reader is aware of the different pixel formats that the ISP can generate (see *Output Formats and sizes* (page 155) or *Output formats (OUT)* (page 210)) and for the tiling section aware of the HW limitations (see *MMU Tiling information* (page 179)).

The FelixCommon library and the CI layer use a common structure to represent the information about a pixel format. These fields are the same regardless of the format (see PIXELTYPE struct)

For example here is an extract of the structure relevant for the size computation:

```
typedef struct PIXELTYPE {
    /** @brief horizontal subsampling */
    IMG_UINT8 ui8HSubsampling;
    /** @brief vertical subsampling */
    IMG_UINT8 ui8VSubsampling;

    /** @brief number of elements represented */
    IMG_UINT8 ui8PackedElements;
    /** @brief in Bytes */
    IMG_UINT8 ui8PackedStride;
} PIXELTYPE;
```

These fields will be used in the following sections to explain the size computation.

The unit of allocation is a block-of-pixel (bop) that covers PackedElements pixels of the buffer and needs PackedStride Bytes of allocation. For example our YUV 10b formats have a bop of 3 elements in 6 Bytes which means we need to round up the width to multiple of 3 and each bop of 3 pixels need 6 Bytes of memory.

The ISP hardware needs all allocation strides to be multiples of SYSMEM_ALIGNMENT which is 64 Bytes. All of our strides **have to** comply with that rule with the exception of the TIFF formats which are Byte addressable by definition. In the following sections the pseudo-code function `round_up_sysmem()` is used to perform the rounding up to the nearest 64B multiple.

15.1 Bayer output (RGGB)

All the Bayer outputs do not have subsampling horizontally or vertically (i.e. the value is 1). A block of pixel (bop) is one line of CFA pattern (one line of RG or GB pattern). The size a the output of the imager interface is used to compute the allocation for data-extraction Bayer formats (CI_MODULE_IIF::ui16ImagerSize).

The details can be found for each format in PixelTransformBayer() (format list *Data Extraction formats (Bayer)* (page 158)):

Format	Elements	Stride (B)
BAYER_8	4	4
BAYER_10	3	4
BAYER_12	16	24

The allocation follows the following algorithm, which is the same for RGB outputs (see CI_ALLOC_RGBSizeInfo()):

```
bop = ceil(width/packedElements);
stride = bop*packedStride;
stride = round_up_sysmem(stride); // 64B

// apply tiling here if RGB

alloc = off + stride*height; // height in lines not in CFA pattern
```

Warning: Unlike RGB, Bayer formats cannot be tiled.

The HW could generated tiled data extraction but the SW chose to limit tiled output to processed images.

15.2 Bayer TIFF output

The TIFF format is extracted in the RAW2D part of the pipeline. It is a Bayer format organised in MSB order rather the LSB of other formats. It is Byte packed and Byte aligned unlike other formats which are burst aligned (i.e. other formats need alignment to the SYS-MEM_ALIGNMENT but not TIFF formats). As a Bayer format they do not support subsampling. The size a the output of the imager interface is used to compute the allocation for data-extraction Bayer formats (CI_MODULE_IIF::ui16ImagerSize).

The information about the formats are available in the same function that other bayer formats (PixelTransformBayer() - for the list of formats is available in see *Raw 2D Extraction (TIFF)* (page 160)):

Format	Elements	Stride (B)
TIFF_10	4 (in MSB)	5
TIFF_12	2 (in MSB)	3

The allocation follows the following algorithm (see CI_ALLOC_Raw2DSizeInfo()):

```
bop = ceil(width/packedElements);
stride = bop*packedStride;
// no rounding to system alignment!
```

```
// no tiling!
alloc = stride*height;
```

Warning: TIFF formats are access by Bytes and cannot be tiled.

15.3 RGB input/output

The RGB outputs can be at HDR extraction or Display output as shown in *HDR Extraction* (page 159) and *Display pipeline formats (RGB)* (page 157). The input can be used to insert frames at HDR insertion point.

As for the Bayer formats the HDR formats use the size at the output of the imager interface as allocation size (CI_MODULE_IIF::ui16ImagerSize). The display output will use the maximum size of the scaler output defined in CI_PIPELINE::ui16MaxDispOutWidth and CI_PIPELINE::ui16MaxDispOutHeight.

The information about the formats is available in PixelTransformRGB():

Formats	Elements	Stride (B)
RGB_888_24 or BGR_888_24	1	3
RGB_888_32 or BGR_888_32	1	4
RGB_101010_32 or BGR_101010_32	1	4
BGR_161616_64	1	8

Note: Display pipeline output in HW is by default RGB (i.e. B in LSB) but use of the R2Y and Y2R block allow the change to BGR when using the ISPC library.

BGR 10b in 32 Bytes cannot be obtained as RGB when using HDR extraction as it extracted from the HW before the conversion blocks.

BGR 16b in 64 Bytes cannot be used as output - it is the HDR insertion format.

The algorithm followed to allocate is the same than Bayer formats.

15.4 YUV output

YUV output is a scaled output. It therefore uses the CI_PIPELINE::ui16MaxEncOutWidth as `width` and CI_PIPELINE::ui16MaxEncOutHeight as `height`.

Available formats (see *Encoder pipeline formats (YUV)* (page 155) and PixelTransformYUV()):

	H Sub	V Sub	Elements	Stride (B)
NV21 8b or NV12 8b	2	2	1	1
NV21 10b or NV12 10b	2	2	3	6
NV61 8b or NV16 8b	2	1	1	1
NV61 10b or NV16 10b	2	1	3	6

Note: The encoder pipeline is output YVU by default. But the usage of the Y2R allow the high level to revert the matrix to allow an output of YUV.

The allocation follows the following algorithm, which is slightly different from the Bayer/RGB allocation (see CI_ALLOC_YUVSizeInfo()):

```
bop = ceil(width/packedElements);
YStride = bop*packedStride;
YStride = round_up_sysmem(YStride); // 64B

// 2*width because UV interleaved in chroma
CWidth = 2*width/HSubsampling;
bop = ceil(Cwidth/PackedElements);
CStride = bop*PackedStride;
CStride = round_up_sysmem(CStride); // 64B

// apply tiling stride here

// coff being the offset between luma and chroma
// not the offset at which chroma starts
// offsets HAVE to be multiple of 64 Bytes too
alloc = yoff + YStride*height
    + coff + CStride*height/VSubsampling;
```

Note: The CI test application has several parameters (see *Allocation size options* (page 53)) to change the allocation size, strides and offsets. The chroma offset asked is the **offset at which chroma starts** (`cbcroff`) not the distance between luma and chroma.

In that case the needed allocation is: $alloc = cbcroff + CStride * \frac{height}{VSubsampling}$

This also involves the verification that $cbcroff > yoff + YStride * height$.

15.5 Tiled output

As per HW limitations the tiling stride has to be a power of 2 and the same **for all tiled buffers** for a given context. This limitation is applied to both YStride and CStride for YUV buffers.

The tiling stride has to be a multiple of the chosen tiling scheme which is an insmod parameter. The tiling stride can be forced using an insmod parameter as well to ensure that all tiled buffer use at least that stride (see *V2500 Insertion options* (page 17)).

Note: The allocation size is usually the same for physical and virtual addresses. However in

the tiling case we need to allocate a bit more virtual address as the first address of the buffer has to be aligned as explained in [MMU Tiling information](#) (page 179).

15.6 Output sizes examples

15.6.1 1920x1088 NV61-10bit output

Simple example to show how to round up the height to a multiple of 16 (often needed for encoders).

Following the computation section, the stride should be (10b formats are packed 3 elements in 6 Bytes as shown in previous table):

$$\begin{aligned} bopy &= \frac{1,920}{3} = 640 \\ stride_Y &= bopy \times 6 \\ &= 3,840 \text{ Bytes} \end{aligned}$$

$stride_Y$ is therefore 3840 Bytes (stride is a multiple of 64).

$stride_C$ should be the same because:

$$width_C = 2 \times \frac{1,920}{2} = 1,920$$

NV61 is a 4:2:2 format therefore the total allocation size is:

$$\begin{aligned} allocation_{YUV} &= stride_Y \times height_{YUV} + 2 \times stride_C \times \frac{height_{YUV}}{1} \\ &= 3,840 \times 1,088 + 2 \times 3,840 \times \frac{1,088}{1} \\ &= 8,355,840 \text{ Bytes} \end{aligned}$$

15.6.2 1080x720 NV21 output containing 2 frames one next to each other

This scenario assumes that we want to capture 2 interleave frames A and B (for example from 2 different sensors) following the layout bellow:

+-----+	+-----+	+-----+
Ay	By	
+-----+	+-----+	+-----+
ACbCr	BCbCr	
+-----+	+-----+	+-----+

Note: The DDK does not support this scenario by default with a single context. But the buffers could be imported and triggered with different offsets on each contexts.

Following the previous section, allocation for the A frame should be:

- $stride_Y$ of 1088 Bytes (1080 is not a multiple of 64).
- $stride_C$ of 1088 Bytes (420 makes the $width_C = 2 \times \frac{width_{YUV}}{2}$ so the chroma bop is the same than the luma).

Therefore we can use those strides as offsets for the B frame. Which will result in the following:

$$\begin{aligned} off_Y &= 1,088 \text{ Bytes} \\ off_C &= 1,088 \text{ Bytes} \\ off_{CbCr} &= stride_Y \times height_{YUV} + off_C \\ &= 783,360 + 1,088 \\ &= 784,448 \text{ Bytes} \end{aligned}$$

The allocation to contain the 2 frame should therefore be have a double stride for luma and chroma (to skip the other frame's location) and the needed offset depending on which image we want to capture.

For both frames:

$$stride_Y = stride_C = 2,176 \text{ Bytes}$$

For frame A:

$$\begin{aligned} A_off_Y &= 0 \\ A_off_C &= 0 \\ A_off_{CbCr} &= 783,360 \text{ Bytes} \end{aligned}$$

For frame B:

$$\begin{aligned} B_off_Y &= 1,088 \text{ Bytes} \\ B_off_C &= 1,088 \text{ Bytes} \\ B_off_{CbCr} &= 784,448 \text{ Bytes} \end{aligned}$$

Therefore the frame B needs the biggest allocation. To cope with both the buffers should allocate:

$$\begin{aligned} allocation_{YUV} &= B_off_{CbCr} + stride_C \times \frac{height_{YUV}}{Subsv} \\ &= 784,448 + 2,176 \times \frac{720}{2} \\ &= 784,448 + 783,360 \\ &= 1,567,808 \text{ Bytes} \end{aligned}$$

15.6.3 4096x2160 NV21 and 1920x1080 RGB888_32

This is a typical example of getting a full resolution YUV output and a scaled down RGB output to display on a screen.

This example is not using tiling therefore each output can be considered individually.

NV21 is 4:2:0 and we can follow similar steps than for previous examples to compute the allocation sizes:

$$\begin{aligned} stride_Y &= 4,096 \text{ Bytes} \\ stride_C &= 4,096 \text{ Bytes} \\ allocation_{YUV} &= 4,096 \times 2,160 + 4,096 \times \frac{2,160}{2} \\ &= 3,271,040 \text{ Bytes} \end{aligned}$$

The RGB 8b in 32 Bytes allocation will produce:

$$\begin{aligned} bop_{RGB} &= \frac{1,920}{1} \\ stride_{RGB} &= bop_{RGB} \times 3 \\ &= 5,760 \text{ Bytes} \end{aligned}$$

The stride is a multiple of the system alignment therefore the total allocation is:

$$\begin{aligned} allocation_{RGB} &= stride_{RGB} \times height_{RGB} \\ &= 5,760 \times 1,080 \\ &= 6,220,800 \text{ Bytes} \end{aligned}$$

The final allocation should therefore be 3,271,040 Bytes for YUV frames and 6,220,800 Bytes for RGB frames.

15.6.4 4096x2160 NV21 tiled and 1920x1080 RGB888_32 tiled

This example is the same than the previous one but using tiled output.

Tiling involves that all tiled output share the same tiling stride. This means we have to compute the which one of all the tiled output has the biggest stride (as a power of 2) and use that as the stride for all tiled outputs.

The width and height have to be rounded up to cover the tiling scheme. Let's assume that we are using the 256x16 scheme.

The YUV output is 4096x2160 which is 16 horizontal tile, 135 vertical tiles. Therefore the resolution does not need to change. The computed stride of 4096 is a power of 2 therefore can be used as the tiling stride for YUV outputs.

Warning: A $stride_Y$ and $stride_C$ of 4096 Bytes is not final because RGB may have a bigger stride!

The RGB output is 1920x1080 which 7.5 horizontal tiles (need to round up to 8) and 67.5 vertical tiles (need to round up to 68). The stride computation therefore becomes based on:

$$\begin{aligned} width_{RGB} &= 8 \times 256 = 2,048 \\ height_{RGB} &= 68 \times 16 = 1,088 \end{aligned}$$

$$\begin{aligned} bop_{RGB} &= \frac{2,048}{1} \\ stride_{RGB} &= 2,048 \times 3 \\ &= 6,144 \text{ Bytes} \end{aligned}$$

However 6144 is not a power of two, the closest power of 2 is 8192.

The found tiling stride for YUV is 4096 and the tiling stride found for RGB is 8192. Because of

the HW limitation both should use the biggest: 8192. The allocation should therefore be:

$$\begin{aligned} allocation_{YUV} &= 8,192 \times 2,160 + 8,192 \times \frac{2,160}{2} \\ &= 26,542,080 \text{ Bytes} \end{aligned}$$

$$\begin{aligned} allocation_{RGB} &= 8,192 \times 1,088 \\ &= 8,912,896 \text{ Bytes} \end{aligned}$$

15.6.5 1920x1080 12b Bayer and RAW 2D 12b extraction

This example will cover the allocation for a Bayer output at 1080p using both data-extraction scheme: the DE point and the RAW 2D points.

The 12b Bayer allocation will be:

$$\begin{aligned} bop_{Bayer} &= \frac{width_{IIF}}{16} = \frac{1,920}{16} \\ &= 120 \\ stride_{Bayer} &= bop_{Bayer} \times 24 \\ &= 2,880 \text{ Bytes} \end{aligned}$$

The block of pixel of 120 is a multiple of the number of elements (16) therefore no rounding is needed for that example. Same goes for the $stride_{Bayer}$ which is a multiple of the system alignment of 64 Bytes. The allocation size is therefore:

$$\begin{aligned} allocation_{Bayer} &= stride_{Bayer} \times height_{IIF} \\ &= 2,880 \times 1,080 \\ &= 3,110,400 \text{ Bytes} \end{aligned}$$

The 12b Tiff allocation will be:

$$\begin{aligned} bop_{Tiff} &= \frac{width_{IIF}}{2} = 960 \\ stride_{Tiff} &= bop_{Tiff} \times 5 \\ &= 4,800 \text{ Bytes} \end{aligned}$$

The size of the input is a multiple of 2 (always, as Bayer formats supported by the ISP have a 2x2 mosaic) so no rounding is needed. In this particular example the $stride_{Tiff}$ is a multiple of the system alignment but for other resolutions it does not have to be. The total allocation is:

$$\begin{aligned} allocation_{Tiff} &= stride_{Tiff} \times height_{IIF} \\ &= 4,800 \times 1,080 \\ &= 5,184,000 \text{ Bytes} \end{aligned}$$

Chapter 16

CI Appendix: System calls mapping

CI_DriverInit() `CI_DriverInit()` creates a connection object and adds it to the driver's list.

System call: `open()`

Kernel function: `DEV_CI_Open()`

CI_DriverFinalise() `CI_DriverFinalise()` delegates to `INT_CI_DriverDisconnect()`.

Destroys the connection object and removes it from the driver's list.

System call: `close()`

Kernel function: `DEV_CI_Close()`

Several functions Delegates to different other functions depending on the IOCTL command received as detailed in *CI Appendix: IOCTL tables* (page 355).

System call: `ioctl()`

Kernel function: `DEV_CI_Ioctl()`

CI_PipelineAllocateBuffer() `CI_PipelineAllocateBuffer()` is allocating device memory and mapping it to user-space.

Memory mapping is done only to share frames memory in user space and kernel space without having to copy any data.

The kernel function looks for the given memory offset in connection's list of non-mapped buffers.

Calls `memunmap()` on failure on previously mapped buffers for that call, see `CI_PipelineDeregisterBuffer()` for details on unmapping.

System call: `memmap()`

Kernel function: `DEV_CI_Mmap()`

CI_PipelineImportBuffer() `CI_PipelineImportBuffer()` is importing memory allocated by an external allocator (supported by the selected `SYS_MEM` implementation).

See `CI_PipelineAllocateBuffer()` about memory mapping as it is the same process.

System call: `memmap()`

Kernel function: `DEV_CI_Mmap()`

`CI_PipelineAllocateLSHMatrix` `CI_PipelineAllocateLSHMatrix()` is allocating a LSH de-shading grid of a given size using the same IOCTL mechanism than `CI_PipelineAllocateBuffer()` and `CI_PipelineImportBuffer()`. It will therefore call follow a similar memory mapping process.

The memory mapping process is the same than for `CI_PipelineAllocateBuffer()`.

Un-mapping is done with `CI_PipelineDeregisterLSHMatrix()`.

System call: `memmap()`

Kernel function: `DEV_CI_Mmap()`

`CI_DatagenAllocateFrame()` `CI_DatagenAllocateFrame()` is allocating device memory for the internal data-generator.

See `CI_PipelineAllocateBuffer()` about memory mapping as it is the same process.

System call: `memmap()`

Kernel function: `DEV_CI_Mmap()`

`CI_PipelineDeregisterBuffer()` `memumap()` is called when handling errors at creation of buffers (unmapped already mapped buffers) resulting in same path on the kernel side.

`CI_PipelineDeregisterBuffer()` destroys of a Buffer or Frame object.

System call: `memunmap()`

Kernel function: Not needed for real driver (done by OS)

`CI_PipelineDeregisterLSHMatrix` `memumap()` is called when handling errors at creation of buffers (unmapped already mapped buffers) resulting in same path on the kernel side.

`CI_PipelineDeregisterLSHMatrix()` destroys a LSH grid object.

System call: `memunmap()`

Kernel function: Not needed for real driver (done by OS)

`CI_DatagenDestroyFrame()` `CI_DatagenDestroyFrame()` destroys a data-generator frame.

System call: `memunmap()`

Kernel function: Not needed for real driver (done by OS)

Chapter 17

CI Appendix: IOCTL tables

This paragraph maps the different calls from the user-side CI library to the kernel side functions that are done through `ioctl()`. This information can be found in `ci_ioctl.h` in the kernel sources.

Any return value from `ioctl()` that is negative is always considered erroneous. The kernel side converts IMG errors into `errno` values that are then reconverted in IMG errors on the user side (see *IMG Errors and Errno values* (page 166)).

Warning: This table of IOCTL may be incomplete or incorrect but gives a good starting point to get information about the different IOCTL available for the CI layer. Check the `ci_ioctl.h` and `ci_ioctl_km.c` on the kernel side for more information.

17.1 Driver commands

Driver commands are used to get shared information about the driver. It also allows changing some of the shared data (e.g. line-store and gamma look up table). These commands should come from `ci_api.c`.

CI_DriverInit() `CI_DriverInit()` is called just after `open()` to get the information about the HW from the registers.

IOCTL command: `CI_IOCTL_INFO`

Parameters to kernel: none

Parameters to user: `CI_CONNECTION *`

Kernel function: `INT_CI_DriverConnect()`

CI_DriverGetLinestore() `CI_DriverGetLinestore()` requests an updated version of the line-store structure from kernel-side.

IOCTL command: `CI_IOCTL_LINE_GET`

Parameters to kernel: none

Parameters to user: `CI_LINESTORE *`

Kernel function: `INT_CI_DriverGetLineStore()`

CI_DriverSetLinestore() `CI_DriverSetLinestore()` can be used to propose a new line-store configuration using the knowledge about the sensor.

IOCTL command: `CI_IOCTL_LINE_SET`

Parameters to kernel: `CI_LINESTORE *`

Parameters to user: none

Kernel function: `INT_CI_DriverSetLineStore()`

CI_DriverGetGammaLUT() `CI_DriverGetGammaLUT()` requests an updated version of the Gamma LUT from kernel-side.

IOCTL command: `CI_IOCTL_GMAL_GET`

Parameters to kernel: none

Parameters to user: `CI_MODULE_GMA_LUT *`

Kernel function: `INT_CI_DriverGetGammaLUT()`

CI_DriverSetGammaLUT() `CI_DriverSetGammaLUT()` can be used to propose a new Gamma LUT to the kernel side.

IOCTL command: `CI_IOCTL_GMAL_SET`

Parameters to kernel: `CI_MODULE_GMA_LUT *`

Parameters to user: none

Kernel function: `INT_CI_DriverSetGammaLUT()`

CI_DriverGetTimestamp() `CI_DriverGetTimestamp()` allows to access to the HW register that contains the timestamp information.

IOCTL command: `CI_IOCTL_TIME_GET`

Parameters to kernel: none

Parameters to user: `uint32 *`

Kernel function: `INT_CI_DriverGetTimestamp()`

CI_DriverGetAvailableInternalDPFBuffer() `CI_DriverGetAvailableInternalDPFBuffer()` allows to access the current status of the DPF internal buffer (shared between all contexts).

IOCTL command: `CI_IOCTL_DPFI_GET`

Parameters to kernel: none

Parameters to user: `uint32 *`

Kernel function: `INT_CI_DriverGetDPFInternal()`

CI_DriverGetRTMInfo() `CI_DriverGetRTMInfo()` allows to access the Real Time Monitoring to debug the HW state (called RTM_A). This will give the status of the pipeline at this point of time. This will also read all of the context status registers.

IOCTL command: `CI_IOCTL_RTM_GET`

Parameters to kernel: none

Parameters to user: CI_RTM_INFO *

Kernel function: INT_CI_DriverGetRTM()

CI_DriverDebugRegRead and CI_DriverDebugRegWrite

Warning: These two functions only work if the kernel module was compiled with debug functions (macro CI_DEBUG_FCT, see *CI options* (page 24) for CMake switch to enable it).

CI_DriverDebugRegRead() allows the user-space to read any register and CI_DriverDebugRegWrite() to write any register.

The choice of reading or writing is the parameters (CI_DEBUG_REG_PARAM). The register bank is chosen with an enumerator.

Note: It is also possible to read memory using that function using the CI_BANK_MEM bank. **The offset has to be the CPU-mapped virtual address.**

Reading memory is not possible when running the fake driver.

If the kernel-side driver was compiled without the debug functions this IOCTL will be ignored and treated as an invalid command (return -ENOTTY).

IOCTL command: CI_IOCTL_DBG_REG

Parameters to kernel: CI_DEBUG_REG_PARAM *

Parameters to user: CI_DEBUG_REG_PARAM *

Kernel function: INT_CI_DebugReg()

17.2 Configuration commands

Configuration commands are used to update the Pipeline object from user to kernel and add/remove allocated buffers. These commands should come from `ci_pipeline.c`.

CI_PipelineRegister() CI_PipelineRegister() registers the Pipeline configuration to the kernel-side. The kernel-side creates a new KRN_CI_PIPELINE object.

IOCTL command: CI_IOCTL_PIPE_REG

Parameters to kernel: CI_PIPELINE *

Parameters to user: int Pipeline's ID as IOCTL return code; must be >0

Kernel function: INT_CI_PipelineRegister()

CI_PipelineDestroy() CI_PipelineDestroy() deregisters the Pipeline configuration from the kernel-side. The KRN_CI_PIPELINE object is also destroyed.

IOCTL command: CI_IOCTL_PIPE_DEL

Parameters to kernel: int Pipeline's ID

Parameters to user: none

Kernel function: INT_CI_PipelineDeregister()

CI_PipelineHasWaiting() CI_PipelineHasWaiting() allows to know the number of captured frames that can be acquired by the user.

IOCTL command: CI_IOCTL_PIPE_WAIT

Parameters to kernel: int Pipeline's ID

Parameters to user: int (as IOCTL return code) number of waiting frames – changed to Boolean in user-space

Kernel function: INT_CI_PipelineHasWaiting()

CI_PipelineHasAvailable() CI_PipelineHasAvailable() allows to know the number of frames that can potentially be captured (this depends on the HW waiting list as well).

IOCTL command: CI_IOCTL_PIPE_AVL

Parameters to kernel: CI_HAS_AVAIL *, including Pipeline's ID

Parameters to user: CI_HAS_AVAIL * updated number of available shots and buffers

Kernel function: INT_CI_PipelineHasAvailable()

CI_PipelineHasPending() CI_PipelineHasPending() allows to know the number of frames that were submitted to the HW but are not captured yet. This is not guaranteed to be correct but can be used as information of fullness of the pipeline.

IOCTL command: CI_IOCTL_PIPE_PEN

Parameters to kernel: int Pipeline's ID

Parameters to user: int (as IOCTL return code) 0 if no frames are pending, 1 if some frames are pending

Kernel function: INT_CI_PipelineHasPending()

CI_PipelineAddPool() CI_PipelineAddPool() allocate enough device memory for a configuration for 1 Shot (not image data). Once the IOCTL succeeded the user side calls `mmap()` to have access to the new information available to user-space.

When user-space needs several Shots this function has to be called once per Shot.

IOCTL command: CI_IOCTL_PIPE_ADD

Parameters to kernel: CI_POOL_PARAM *, including Pipeline's ID

Parameters to user: CI_POOL_PARAM * update with information to perform the memory mapping and computed values

Kernel function: INT_CI_PipelineAddShot()

CI_PipelineDeleteShots() CI_PipelineDeleteShots() deletes all the allocated Shots for a Pipeline. Once the IOCTL succeeded the `munmap()` is called and user-side list cleared.

The capture should not be running and all acquired shots should have been released by the user.

IOCTL command: CI_IOCTL_PIPE_REM

Parameters to kernel: int Pipeline's ID

Parameters to user: none

Kernel function: INT_CI_PipelineDeleteShots()

CI_PipelineUpdate() **CI_PipelineUpdate()** updates the kernel-side KRN_CI_PIPELINE::userPipeline from the one currently available in user-side. A module flag can be used to update the structure module by module. The ASAP flag can be set to 1 to force the update on already submitted frames.

IOCTL command: CI_IOCTL_PIPE_UPD

Parameters to kernel: CI_PIPE_PARAM * containing user pipeline structure, ID, update mask to only update some modules and the ASAP flag

Parameters to user: none

Kernel function: INT_CI_PipelineUpdate()

CI_PipelineImportBuffer() and **CI_PipelineAllocateBuffer()** Information to allocate or import an image buffer. Created buffer is mapped to user-space. When user-space needs several Buffers this ioctl is called once per Buffer.

This can be done when the capture is started but will require the buffer to be mapped to the MMU (may require MMU cache flushing which may add memory latency).

The size of the buffer can be 0 to let the kernel-space compute it using ci_alloc_info.h or the size can be provided to the kernel side but must respect the system's limitations. The buffer can also be imported if a file descriptor is provided and the SYS_MEM interface was compiled with an import capability.

Note: The user-space functions are calling IMG_CI_PipelineCreateBuffer() to delegate the work.

The called kernel function will use KRN_CI_PipelineCreateBuffer() function.

Warning: CI_PipelineAllocateLSHMatrix() also uses that ioctl but is treated in its own section as the behaviour in user-space and kernel-space is slightly different.

IOCTL command: CI_IOCTL_CREATE_BUFF

Parameters to kernel: CI_ALLOC_PARAM * containing Pipeline's ID, file descriptor if importing, type of output buffer

Parameters to user: CI_ALLOC_PARAM * updated information to perform the memory mapping and computed size

Kernel function: INT_CI_PipelineCreateBuffer()

CI_PipelineAllocateLSHMatrix() Information to allocate a LSH de-shading grid. Unlike for allocation of image buffers the size HAS to be given. Created buffer is mapped to user-space.

When user-space needs several Buffers this ioctl is called once per Buffer.

This can be done when the capture is started but will require the buffer to be mapped to the MMU (may require MMU cache flushing which may add memory latency).

Warning: CI_PipelineImportBuffer() and CI_PipelineAllocateBuffer() also uses that ioctl but is treated in its own section as the behaviour in user-space and kernel-space is slightly different.

Note: The called kernel function will use the KRN_CI_PipelineCreateLSHBuffer() function.

IOCTL command: CI_IOCTL_CREATE_BUFF

Parameters to kernel: CI_ALLOC_PARAM * containing Pipeline's ID, size

Parameters to user: CI_ALLOC_PARAM * updated information to perform the memory mapping

Kernel function: INT_CI_PipelineCreateBuffer()

CI_PipelineDeregisterBuffer() CI_PipelineDeregisterBuffer() to liberate an allocated/imported image buffer.

This can be done on available buffers only. If the capture is started this requires unmapping from the MMU (may require MMU cache flushing which may add memory latency).

Warning: CI_PipelineDeregisterLSHMatrix() also uses that ioctl but is treated in its own section as the behaviour in user-space and kernel-space is slightly different.

Note: The called kernel function will use the KRN_CI_PipelineDeregisterBuffer() function.

IOCTL command: CI_IOCTL_DEREG_BUFF

Parameters to kernel: CI_ALLOC_PARAM * containing Pipeline's ID and buffer ID

Parameters to user: none

Kernel function: INT_CI_PipelineDeregBuffer()

CI_PipelineDeregisterLSHMatrix() CI_PipelineDeregisterLSHMatrix() to liberate an allocated LSH grid.

This can be done on matrices not currently in use by the HW. If the capture is started this requires unmapping from the MMU (may require MMU cache flushing which may add memory latency).

Warning: CI_PipelineDeregisterBuffer() also uses that ioctl but is treated in its own section as the behaviour in user-space and kernel-space is slightly different.

Note: The called kernel function will use the KRN_CI_PipelineDeregisterLSHBuffer() function.

IOCTL command: CI_IOCTL_DEREG_BUFF

Parameters to kernel: CI_ALLOC_PARAM * containing Pipeline's ID and buffer ID

Parameters to user: none

Kernel function: INT_CI_PipelineDeregBuffer()

17.3 Capture commands

Commands used to control the capture process. These commands should be coming from `ci_pipeline.c`.

CI_PipelineStartCapture() CI_PipelineStartCapture() asks the kernel-side to reserve the HW needed for this Pipeline configuration (can fail if HW is not available).

IOCTL command: CI_IOCTL_CAPT_STA

Parameters to kernel: int Pipeline's ID

Parameters to user: none

Kernel function: INT_CI_PipelineStartCapture()

CI_PipelineStopCapture() CI_PipelineStopCapture() releases the reserved HW needed for this Pipeline configuration.

IOCTL command: CI_IOCTL_CAPT_STP

Parameters to kernel: int Pipeline's ID

Parameters to user: none

Kernel function: INT_CI_PipelineStopCapture()

CI_PipelineTriggerShoot() and CI_PipelineTriggerShootNB()

CI_PipelineTriggerShoot() adds a frame to the pending list of the HW. Can fail if there is no available frame. If the HW pending list is full the blocking version will sleep until an element is available in the HW list, the non-blocking version will return an error code.

The specified version can be used to specify which buffer to use for the capture if the Pipeline was configured accordingly.

IOCTL command: CI_IOCTL_CAPT_TRG

Parameters to kernel: CI_BUFFER_TRIGG * containing Pipeline's ID, blocking flag and optionally encoder and display buffer IDs to use

Parameters to user: none

Kernel function: INT_CI_PipelineTriggerShoot()

CI_PipelineAcquireShot() and CI_PipelineAcquireShotNB()

CI_PipelineAcquireShot() and CI_PipelineAcquireShotNB() request a frame from the Processed list. The given frame is moved to the Sent list and cannot be used until released. The function may sleep until a frame becomes available, unless the non-blocking version (NB) is used and an error code will be returned.

IOCTL command: CI_IOCTL_CAPT_BAQ

Parameters to kernel: CI_BUFFER_PARAM * containing Pipeline's ID, blocking flag

Parameters to user: CI_BUFFER_PARAM * updated with Shot's ID and size information

Kernel function: INT_CI_PipelineAcquireShot()

CI_PipelineReleaseShot() CI_PipelineReleaseShot() moves back an acquired frame to the Available list and can be re-used for captures.

IOCTL command: CI_IOCTL_CAPT_BRE

Parameters to kernel: CI_BUFFER_TRIGG * Pipeline's ID and Shot

Parameters to user: none

Kernel function: INT_CI_PipelineReleaseShot()

CI_PipelineIsStarted() CI_PipelineIsStarted() synchronises the user-side from kernel-side started status (when status is unexpected because resume/suspend may have been called).

IOCTL command: CI_IOCTL_CAPT_ISS

Parameters to kernel: int Pipeline's ID

Parameters to user: int (status as IOCTL return) 0 is stopped, 1 is started and <0 is error

Kernel function: INT_CI_PipelineIsStarted()

17.4 Gasket commands

These commands are used to control the HW gasket reservation. These commands should come from `ci_gasket.c`.

CI_GasketAcquire() Use CI_GasketAcquire() for the Connection object to acquire the specified gasket. If successful write the given configuration to the gasket registers.

IOCTL command: CI_IOCTL_GASK_ACQ

Parameters to kernel: CI_GASKET * containing gasket number and configuration to apply

Parameters to user: none

Kernel function: INT_CI_GasketAcquire()

CI_GasketRelease() CI_GasketRelease() releases the specified gasket. Successful only if the Connection is the same than the one that acquired the gasket.

IOCTL command: CI_IOCTL_GASK_REL

Parameters to kernel: IMG_UINT8 gasket ID

Parameters to user: none

Kernel function: INT_CI_GasketRelease()

CI_GasketGetInfo() Use CI_GasketGetInfo() to get information about a given gasket. Possible even if the gasket is not available.

IOCTL command: CI_IOCTL_GASK_NFO

Parameters to kernel: CI_GASKET_PARAM * containing gasket number and CI_GASKET_INFO to use for output.

Parameters to user: CI_GASKET_INFO *

Kernel function: INT_CI_GasketGetInfo()

17.5 Internal Data Generator commands

Commands used to control the IIF Datagen. These commands should come from `ci_intdg.c`.

CI_DatagenCreate() and **CI_DatagenDestroy()** CI_DatagenCreate() creates an internal data-generator object that can then be destroyed using CI_DatagenDestroy().

When registering an object given identifier is <0. When destroying an object the identifier is >0 (0 is an invalid identifier).

Note: CI_DatagenDestroy() deletes all remaining frames calling CI_DatagenDestroyFrame().

IOCTL command: CI_IOCTL_INDG_REG

Parameters to kernel: int as the Datagen object identifier, if <0 means the data-generator object should be created.

Parameters to user: datagen ID as IOCTL return code (when registering only); must be >0.

Kernel function: INT_CI_DatagenRegister() (including destruction)

CI_DatagenStart() CI_DatagenStart() acquires HW to start capturing frames.

IOCTL command: CI_IOCTL_INDG_STA

Parameters to kernel: CI_DG_PARAM * containing datagen ID and its configuration

Parameters to user: none

Kernel function: INT_CI_DatagenStart()

CI_DatagenIsStarted() Use `CI_DatagenIsStarted()` to know if the datagen is started. May have been stopped because of errors or suspend/resume.

IOCTL command: `CI_IOCTL_INDG_ISS`

Parameters to kernel: `int` as datagen ID

Parameters to user: 1 if started, 0 otherwise.

Kernel function: `INT_CI_DatagenIsStarted()`

CI_DatagenStop() `CI_DatagenStop()` stops a started Data Generator. Releases the acquired HW.

IOCTL command: `CI_IOCTL_INDG_STP`

Parameters to kernel: `int` as datagen ID

Parameters to user: none

Kernel function: `INT_CI_DatagenStop()`

CI_DatagenAllocateFrame() Use `CI_DatagenAllocateFrame()` to allocate a frame for the data-generator.

IOCTL command: `CI_IOCTL_INDG_ALL`

Parameters to kernel: `CI_DG_FRAMEINFO *` containing the frame size and datagen ID

Parameters to user: `CI_DG_FRAMEINFO *` updated with frame ID and mmap ID

Kernel function: `INT_CI_DatagenAllocate()`

CI_DatagenGetAvailableFrame() and **CI_DatagenGetFrame()** Acquire a frame that is available for conversion. `CI_DatagenGetAvailableFrame()` get 1st available frame while `CI_DatagenGetFrame()` allows to specify a frame ID.

IOCTL command: `CI_IOCTL_INDG_ACQ`

Parameters to kernel: `CI_DG_FRAMEINFO *` containing datagen ID and frameID (0 to get 1st available)

Parameters to user: `CI_DG_FRAMEINFO *` updated with frame ID

Kernel function: `INT_CI_DatagenAcquireFrame()`

CI_DatagenInsertFrame() and **CI_DatagenReleaseFrame()** Release an acquired frame either as a frame trigger with `CI_DatagenInsertFrame()` or just as a cancelled release with `CI_DatagenReleaseFrame()`.

IOCTL command: `CI_IOCTL_INDG_TRG`

Parameters to kernel: `CI_DG_FRAMETRIG *` with frame and datagen information as well as other options needed for the frame to be triggered (or no more information for release).

Parameters to user: none

Kernel function: `INT_CI_DatagenTrigger()`

CI_DatagenWaitProcessedFrame() Wait for a triggered frame to be fully processed by the data-generator. When fully processed the frame is pushed back to the available list.

This can be blocking or wait for the semaphore timeout.

Parameters to kernel: CI_DG_FRAMEWAIT * with datagen information and blocking option.

Parameters to user: none

Kernel function: INT_CI_DatagenWaitProcessed()

CI_DatagenDestroyFrame() Use CI_DatagenDestroyFrame() to free a specific frame device memory.

IOCTL command: CI_IOCTL_INDG_TRG

Parameters to kernel: CI_DG_FRAMETRIG * with the frame and datagen information.

Parameters to user: none

Kernel function: INT_CI_DatagenFreeFrame()

Chapter 18

CI Appendix: Debugging a Page Fault

This section will try to describe how to debug a page fault occurring while using the driver. Page faults should **not** occur if the driver was correctly integrated.

The section includes a scenario that assumes that the device was wrongly integrated and that the memory offset is wrong (i.e. all mapping is done but the HW is not looking at the correct location). This will try to prove it by ensuring the mapping is correct. Information about memory offset in regard to CPU view and Device view are available in the *Platform Integration Guide* (page 32).

This section uses the *Video Bus 4 MMU Function Specification* document delivered with the HW as a reference on how the MMU works. Please refer to the *Directory Table lookup and Page Table lookup* figures to understand the way virtual addresses are converted into physical address.

18.1 Initial Information

From the run of the driver we assume that this information has been gathered (through logging present in the delivered driver or known information):

- Which contexts are used for the test
- MMU option used (mmucontrol at insmod)
- Physical addresses used for MMU directories (KRN_CI_MMUConfigure() should write MMU_DIR_BASE_ADDR register with or without shifting depending on mmucontrol).
- Relation of physical addresses and virtual address (including size) for all allocation (SYS_MemAlloc() and IMG_CI_ShotMallocBuffers() are good locations to look at).
- Page fault information from the MMU: MMU_STATUS0 and MMU_STATUS1 registers (HW_CI_DriverThreadHandleInterrupt() should be printing them on pagefault).

It is also assumed that the system is in a valid state after the failure so that memory can be accessed to verify the pages look-up (otherwise code may have to be added to interrupt handler to do the crawling).

Section *MMU Requestors* (page 175) explains the relation between context, data generator and MMU directory used.

Note: MMU virtual address size is always 32b and is used to compute the extended address range shift in the MMU documentation as: EXT_ADDR_RANGE=phys_size-virt_size.

-
- When using `mmucontrol=2, phys_size=40 virt_size=32 EXT_ADDR_RANGE=8`
 - When using `mmucontrol=1` (NOT TESTED as invalid in HW), `phys_size=32 virt_size=32 EXT_ADDR_RANGE=0`
 - When using `mmucontrol=0` MMU is not used so no page faults should be possible.
-

18.1.1 Information for our example

Our example is running context 1 only with internal data generator for a very small frame (40x44) with YUV output only (1 buffer allocated only). Our system is 4KB CPU pages (4KB MMU device pages) with `mmucontrol=2` (40b MMU).

We assume that the device memory is located from `0x6000 0000` to `0xF000 0000` of the physical memory but that the device interprets its first address as being `0x0` (i.e. where CPU sees `0x6000 0000`, ISP sees `0x0000 0000`).

The logging of allocations reported:

- Allocation of directories (2 contexts means 2 directories): `p0x6000 0000, p0x6000 1000`
- IIF DG frame `v0xF000 0000 for 0x2000B, ‘‘p0x6000 2000`
- Save structure (ctx 1): `v0x1000 for 0x2000B, p0x6000 4000`
- Load structure (ctx 1): `v0x3000 for 0x1000B, p0x6000 6000`
- Linked list (ctx 1): `v0x4000 for 0x1000B, p0x6000 7000`
- Output YUV (ctx 1): `v0x1000 0000 for 0x2000B, p0x6000 8000`

Page fault reports (`MMU_STATUS0` and `MMU_STATUS1` registers are also partially analysed as we are doing in next section).

```
MMU fault status0=0x4001 status1=0x11000001 addr_ctrl=0x10 (page fault 1, read fault 1, write fault 0)
MMU fault @ v0x4000 on directory 1 - direct entry = 0xd1 - page entry = 0x85
```

Note: Without even going further we can deduce that the error is due to misalignment of the device memory:

- `v0x4000` is the linked list which is the 1st element of memory accessed by the HW (see HW TRM on how linked list operates).

This printing also confirms that we are using context 1 (directory 1) and that MMU is in extended address mode (ext addressing 1) i.e. in 40b mode.

- Fault is reported as “page fault” (meaning fault occurred in directory look up as MMU HW documentation explains). This occurs because the physical memory given as initial page will contains the correct data but the HW is not looking at the correct location
-

18.2 MMU fault analysis

Using the MMU HW documentation we can deduce several things from the `MMU_STATUS0` and `MMU_STATUS1`:

- If the fault was in directory look up and in page look up
- If the fault was during reading or writing access
- Directory associated with the memory access
- The virtual address causing the fault
- From the virtual address we can find the directory entry and page entry.

Using this information and the physical address we know the directories are in we should be able to validate the MMU page tree:

- Ensure the physical address registered as directory entry looks like a directory entry look up (should be 4KB long and mostly be composed of 0s unless we mapped most of the virtual memory)
- Ensure that the directory entry reported is valid (bottom bits are 0x1).

If they are not valid the driver did not map this virtual address correctly and the problem is most likely located in mapping section (done at context start). This is unlikely to occur but check *We proved that the mapping structure was wrong...* (page 369) to have ideas where to look.

- Use the directory entry to find the physical address of the page table if bottom bits are valid.
- Ensure the page table looks like a page table look up (similar than directory table)
- Ensure the page table value at the page entry is valid (bottom bits are 0x1) and have the correct RO or WO flag (R0=0x4 W0=0x2 WR=0x6)
- Find the physical address associated with the virtual address

18.2.1 Analyse our example

Using the information we know:

- MMU_DIR_BASE[1] should be 0x6000 1000>>8=0x0060 0010 because we use 40bits MMU: (LOG2_PAGESIZE-EXT_ADDR_RANGE=12-(40-32)=8)
- At physical address p0x6000 1000 most of the entries are 0 (only few buffers mapped)
- We want to look at entry 0xD1 of the directory. Each entry being 4 Bytes we know the location in physical memory of the directory entry is p0x6000 1344.

The value in p0x6000 1344 should contain the physical address to the page table shift and masked as explained in MMU documentation. In our case the value is 0x0600 0A01 Bottom bit 0x1 therefore it is a valid mapping.

- We are using extended address range (40b MMU) therefore physical address is p0x6000 A000
- We want to look at entry 0x85 of the page table. Each entry being 4 Bytes we know the location in physical memory of the page entry is p0x6000 A214.

The value in p0x6000 A214 should contain the physical address of the linked list and its mask. In our case 0x0600 0705. Bottom bit are 0x5 meaning R0 and valid (expected for linked list – HW only loads it).

- Physical address is extracted the same way than for the page table giving p0x6000 7000 which matches our logged linked list physical address.

We therefore proved that our MMU mapping structure is correct (we manage to discover the correct physical address starting from the MMU directory address for the linked list).

18.3 We proved that the mapping structure is correct...

If we proved that the mapping structure is correct it means that the device does not follow that structure the way it is expected to by the drivers.

If the accessed virtual address is at the end of an allocation it may mean the driver does not allocate enough memory for a buffer (unlikely). In that case verification of allocation sizes in `ci_alloc_info.h` is required. If using imported buffers they have to comply with equivalent sizes than `ci_alloc_info.c` implementation would return. Internal buffers (such as linked list, save structure, load structure, ENS output, DPF output, LSH matrix) should not suffer from that problem.

If the accessed virtual address is the first part of an allocation it is very likely that the HW sees the memory with a different offset than what the driver expects. In that case the `SYS_DEVICE` implementation is not giving the correct address in the `SYS_DEVICE::uiMemoryPhysical` or `SYS_DEVICE::uiDevMemoryPhysical` entries. The *Platform Integration Guide* (page 32) should have more information about CPU/DEVICE offset issues.

18.4 We proved that the mapping structure was wrong...

If we proved that the mapping structure was wrong several actions can be taken:

- Ensure the particular buffer was mapped (additional log in `SYS_MemMap()`)
- Ensure that particular buffer is big enough when imported – including its virtual address (physical allocation could be big enough but size reported when imported could be too small – check `SYS_MemImport()`).

If using tiling ensure that the tiling stride is correctly computed and that the base address has the correct offset. More information about tiling is available in the HW MMU documentation and section.

Chapter 19

CI Appendix: Debugging a Failed to acquire frame

The first step is to validate that the current issue is a frame acquisition problem. The driver works by providing buffers to fill to the HW. Therefore we expect that frames that enqueued on a correctly configured HW will produce a processed frame if a sensor is running on the context's gasket (more details in [Shot and Buffer lifecycle](#) (page 179)).

The “acquisition” of a processed frame is done by waiting on a semaphore that counts the number of “frame end” interrupts generated by the HW. If no interrupts were yet generated then the kernel-side will wait. The waiting is *never* infinite and the amount of time is configurable in the CI kernel module (see [Blocking and non-blocking calls](#) (page 189)). The amount of time is an insmod parameter (see [V2500 Insertion options](#) (page 17)) and the value should be compatible with the sensor’s frame-rate.

The system can fail to receive processed frames for several reasons:

1. Frames are not being received correctly by the hardware (gasket connection issues).
2. The frame is partially processed by hardware but frame end is not generated (configuration invalid, or insufficient bandwidth).
3. The software interrupt could be delayed.

Note: The register information is crucial to debug a missing frame. The context status, the RTM information and the gasket status registers values should be checked using the HW TRM.

For all the cases above the use of Debug FS (see [DebugFS \(real driver\)](#) (page 168)) and the other sources of interrupt of the system can be of significant help. We recommend to use a script similar to the one bellow to gather system information:

```
#  
# need to be root  
#  
# mount debug file system  
mount -t debugfs none /sys/kernel/debug  
  
# make copy of debug information before run  
cp /proc/interrupts pre_interrupts.txt  
cp -r /sys/module/Felix/parameters pre_parameters  
cp -r /sys/kernel/debug/imgfelix pre_imgfelix
```

Then run the test, for example:

```
# runtest
./ISPC_loop -setupFile config.txt -sensor AR330 -sensorMode 0 -nBuffers 3
```

Then gather the information again:

```
# 
# need to be root
#
# copy debug filesystem after test
cp -r /sys/module/Felix/parameters post_parameters
cp -r /sys/kernel/debug/imgfelix post_imgfelix
cp /proc/interrupts post_interrupts.txt
```

19.1 Frames not received by hardware

To see if this is the case we should check the status of the hardware gasket registers. The gasket contains a frame counter which can be compared with the expected frame count. The debug FS counter can help compare the number of pushed frames and the number the gasket received.

It is also possible that the gasket could contain errors (only the MIPI gasket have error counters). If errors are not 0 then it is possible that the frame was considered as erroneous and not transmitted to the ISP.

It is possible to access the status of the gasket using CI_GasketGetInfo().

19.2 Partial frame

In the case that the gasket information seems to produce the correct number of frames but the ISP does not generate a frame end interrupt we have to verify the status of the hardware. The CI_DriverGetRTMInfo() function - also printed by most applications - which gives access to several useful information:

1. Is there any shot in the HW queue? The Linked list emptiness should be the maximum - usually 0x10 - if all shots have been loaded.

If shots are present it means the HW is ready to receive a frame but for some reason is not accepting new frames. Verify that the gasket frame counter is correct.

2. The status reported by the HW context (context status).

The register description will help understand the values printed here. A status of 0 means the HW is idle and ready to process frames.

Any other state may show the HW is blocked during an operation and may be limited by bandwidth.

3. How far through processing the hardware is through a frame (the Context position 0-7 will contain the line that various stages of the pipeline are at).

A value smaller than the expected resolution at a position will show where the frame is blocked.

Example:

Output of 720p YUV image.

```
RTM Context 0 status 0x0    <-- idle
RTM Context 0 linked list emptiness 0x10    <---- empty
RTM Context 0 position 0 0x2d0  <---- group 0 processed 720 lines
RTM Context 0 position 1 0x2d0  <---- group 1 processed 720 lines
RTM Context 0 position 2 0x2d0  <---- group 2 processed 720 lines
RTM Context 0 position 3 0x2d0  <---- group 3 processed 720 lines
RTM Context 0 position 4 0x2d0  <---- group 4 processed 720 lines
RTM Context 0 position 5 0x2d0  <---- group 5 processed 720 lines
RTM Context 0 position 6 0x168  <---- group 6 processed 360 lines
RTM Context 0 position 7 0x0    <---- group 7 processed 0 lines
```

This shows that the hardware has completed writing a 720p frame, is not processing currently and has no outstanding frames to process. The position 6 is displaying half the number of lines, which is expected as it is the encoder output (YUV format is half the number of lines). The position 7 shows that no lines were processed, which is expected as the display output is disabled for that test.

19.3 Software Interrupt delay

If all the previous steps show that the hardware has finished processing we can assume that the software may not have propagated a frame end.

In this case we should examine the time taken to receive an interrupt. Clearly if we have failed to receive a frame we cannot establish the timing for the current frame however we can potentially examine other frames to see if some problem might exist.

When a frame is acquired several timestamps data are stored in the statistics. It is possible to compute the latency between the generation of the frame end interrupt by the HW and the time the SW handled the event by doing a simple subtraction:

```
// assuming CI_SHOT *pShot has been acquired
// and that CI_CONNECTION *pConnection is valid
MC_STATS_TIMESTAMP timestamp;
const float clock = pConnection->sHWInfo.ui32RefClockMhz / 1000000.0; // Mhz to Hz
int latency = 0;

MC_TimestampExtract(pShot->pStatistics, &timestamp);

latency = timestamp.ui32InterruptServiced - timestamp.ui32EndFrameEncOut;
printf("latency of enc output %f sec\n", latency * clock);

latency = timestamp.ui32InterruptServiced - timestamp.ui32EndFrameDispOut;
printf("latency of disp output %f sec\n", latency * clock);
```

The timestamps are reported in clock cycles of the HW. We expect the customers to know the clocking of the ISP (and to have CI_HWINFO::ui32RefClockMhz at the correct value). Multiplying by the clock period should provide the time in seconds. A number of pre-issued buffers can in some cases smooth out occasional frames with long latency but we cannot guarantee this.

A unusually big latency may be the cause of the timeout when waiting on the acquisition of the semaphore when waiting for a frame to be processed. Regular latency peaks may be the result of other drivers delaying the interrupt handler threaded interrupt execution.

Chapter 20

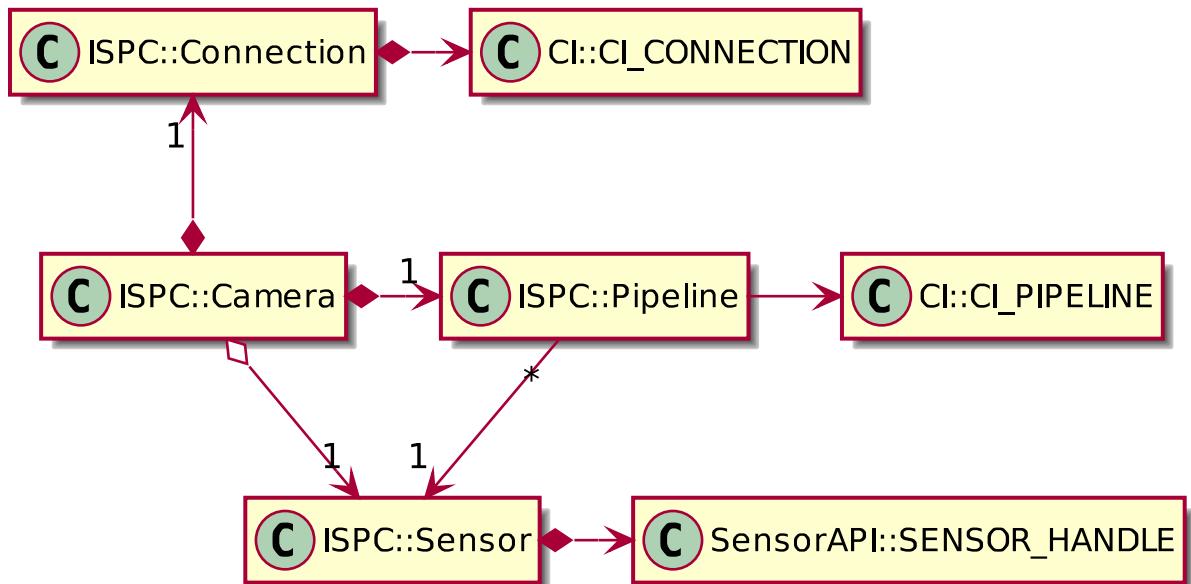
ISPC Appendix: From ISPC to kernel module

This section contains several UML sequence diagrams that picture the interaction from the ISPC layer function calls to the `ioctl()` called by the CI user-side library. The goal is to illustrate the interaction of the ISPC layer with the CI and SensorAPI.

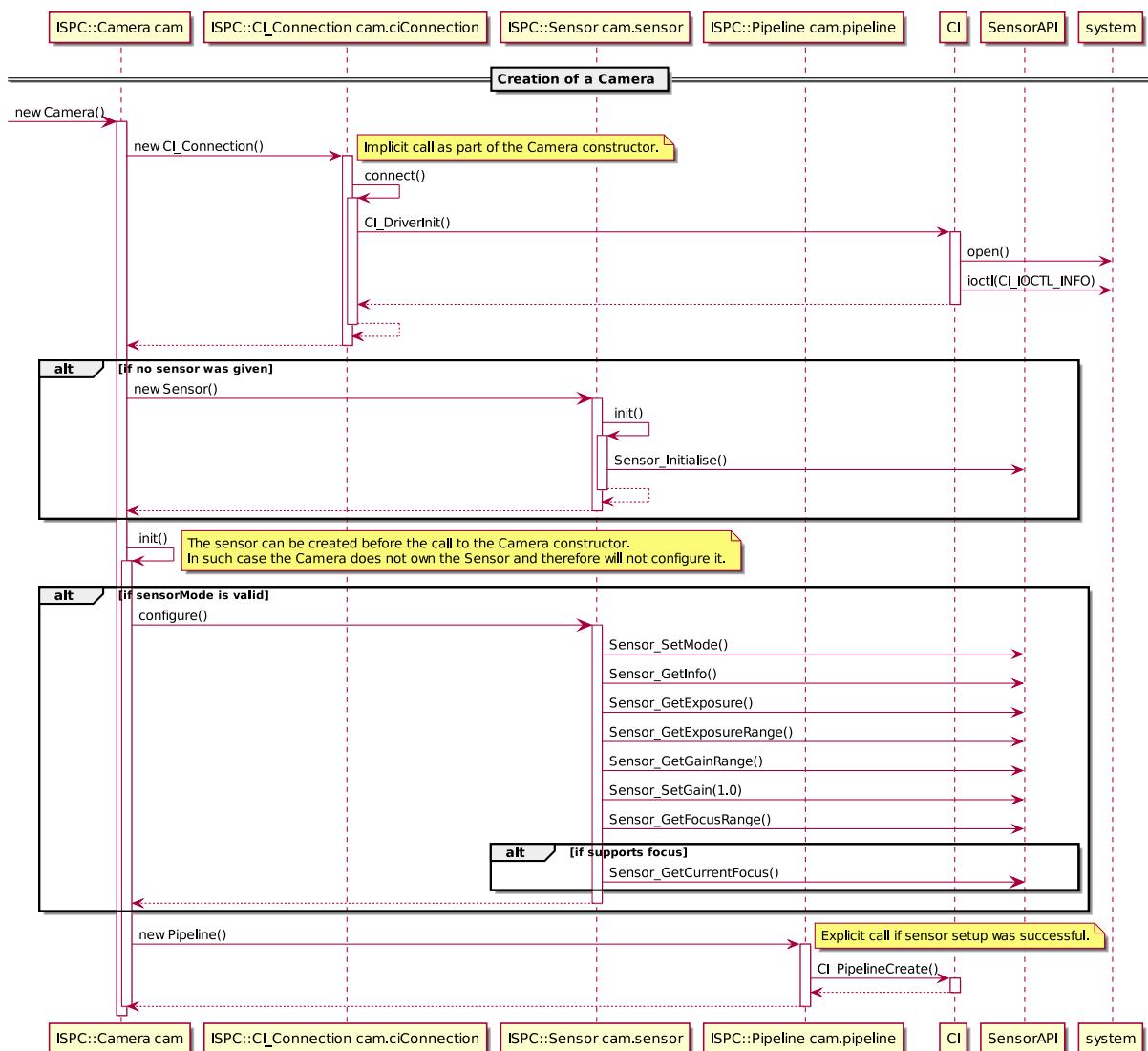
Refere to the doxygen documentation for more details on each function call.

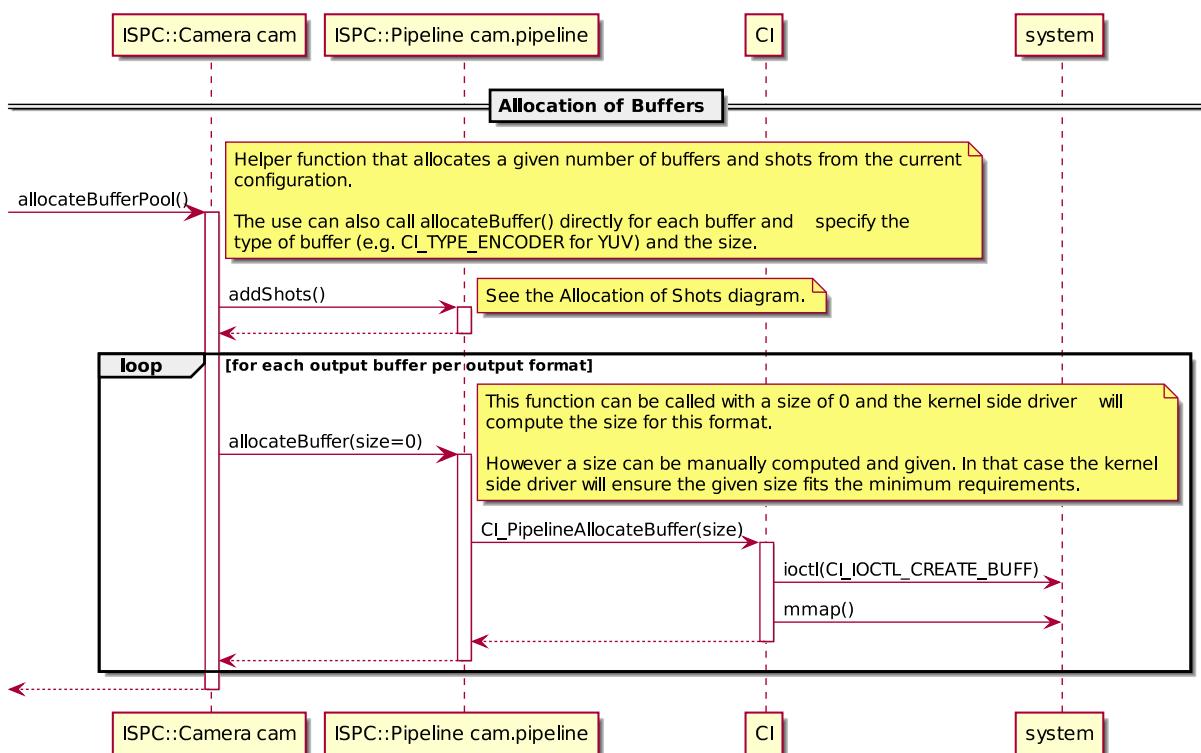
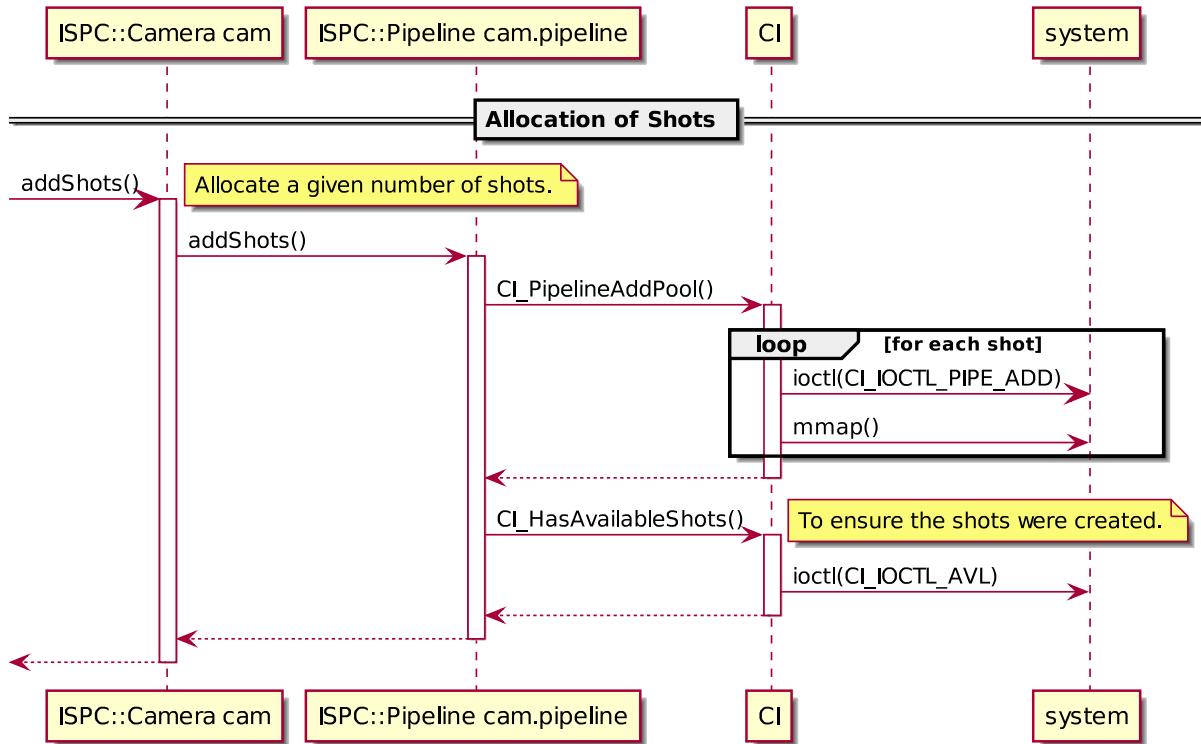
The following diagrams should illustrate several operations from the ISPC lifetime:

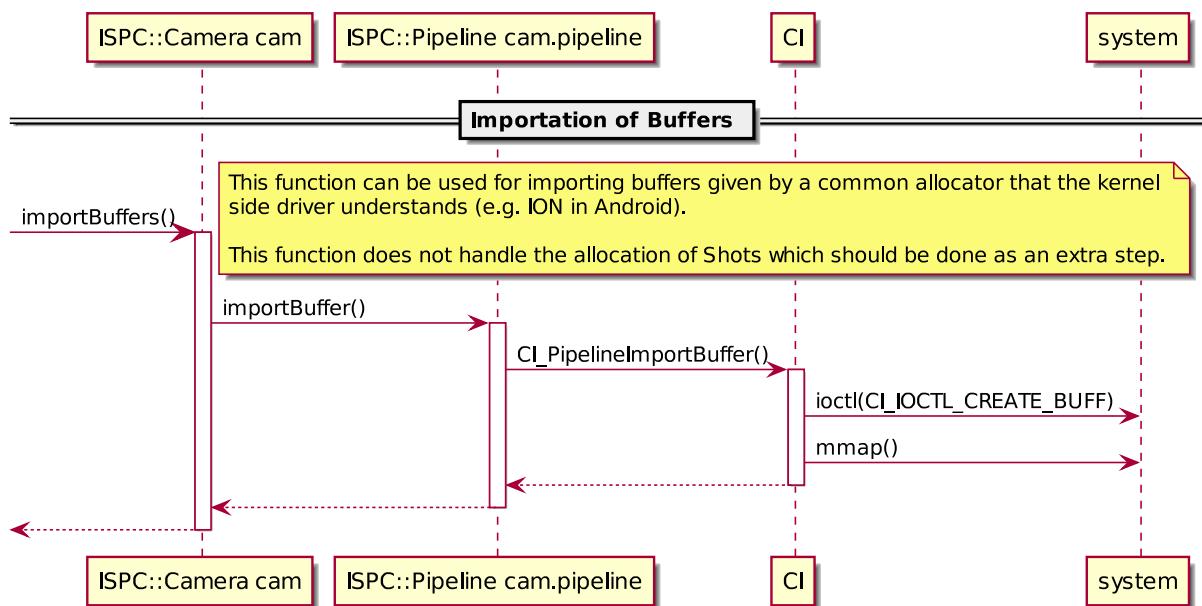
- Creation of a Camera
- Allocation of Shots
- Allocation of Buffers
- Importation of Buffers
- Starting the Camera
- Trigger the capture of a frame
- Acquire the result of a capture
- Release the result of a capture
- Stopping the Camera
- Destruction of Buffers
- Destruction of Shots
- Destruction of the Camera

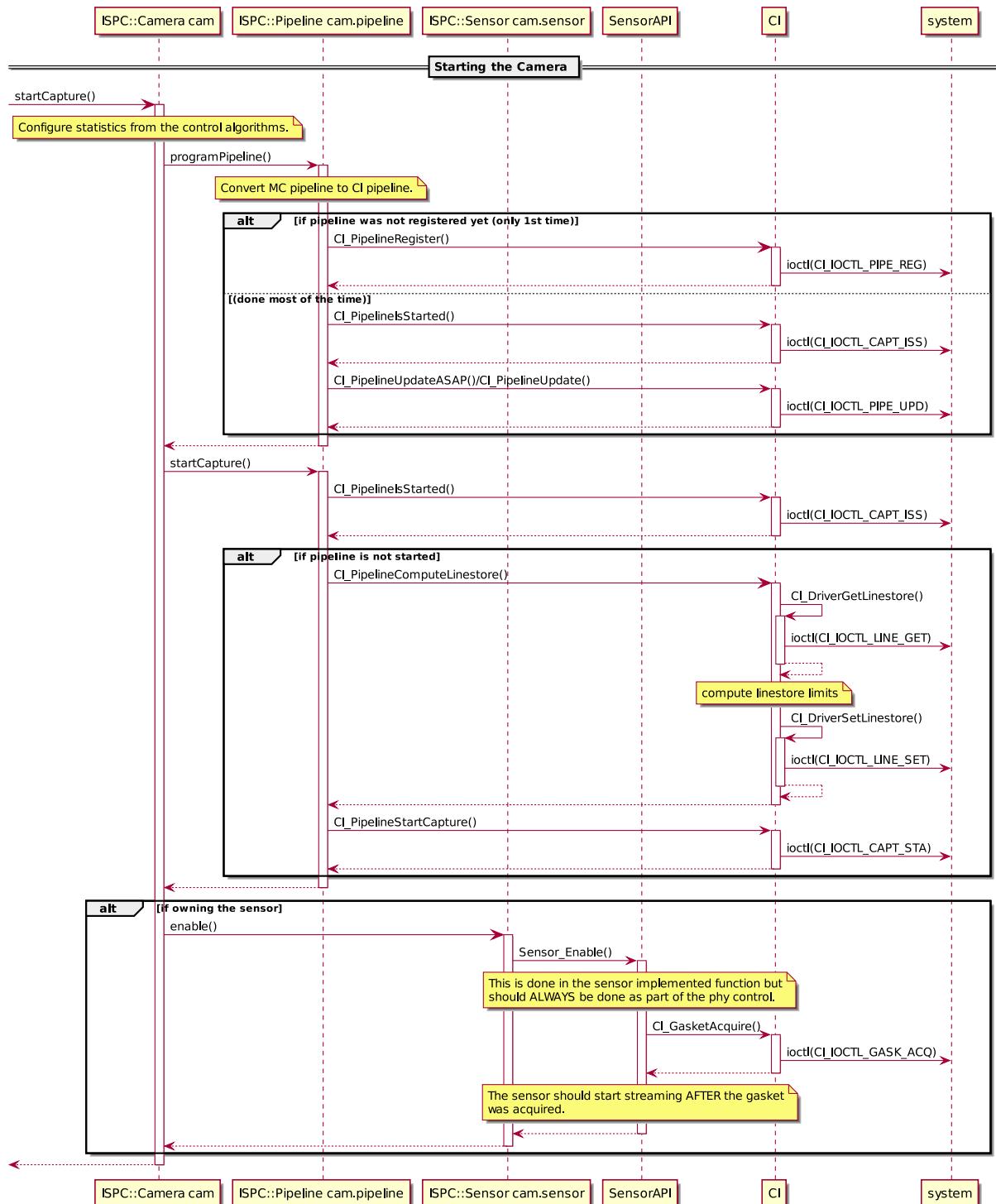


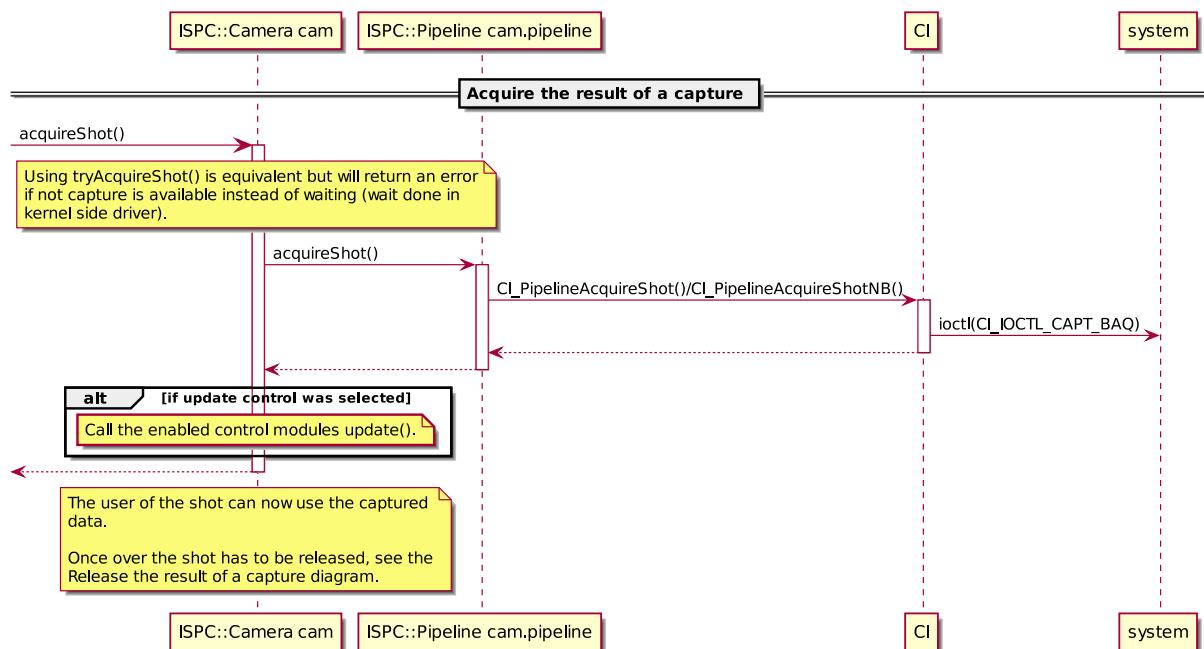
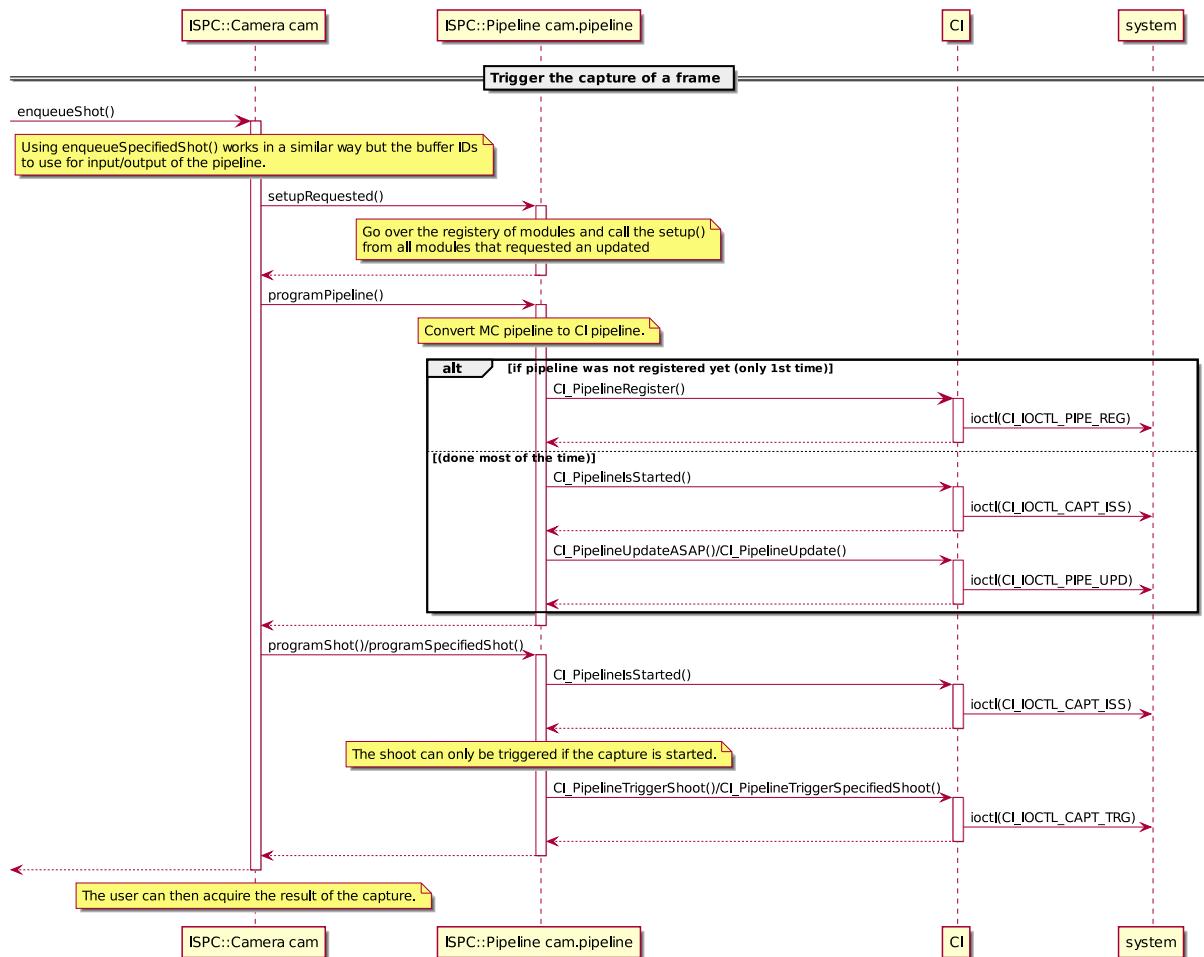
Extract from the ISPC::Camera relations

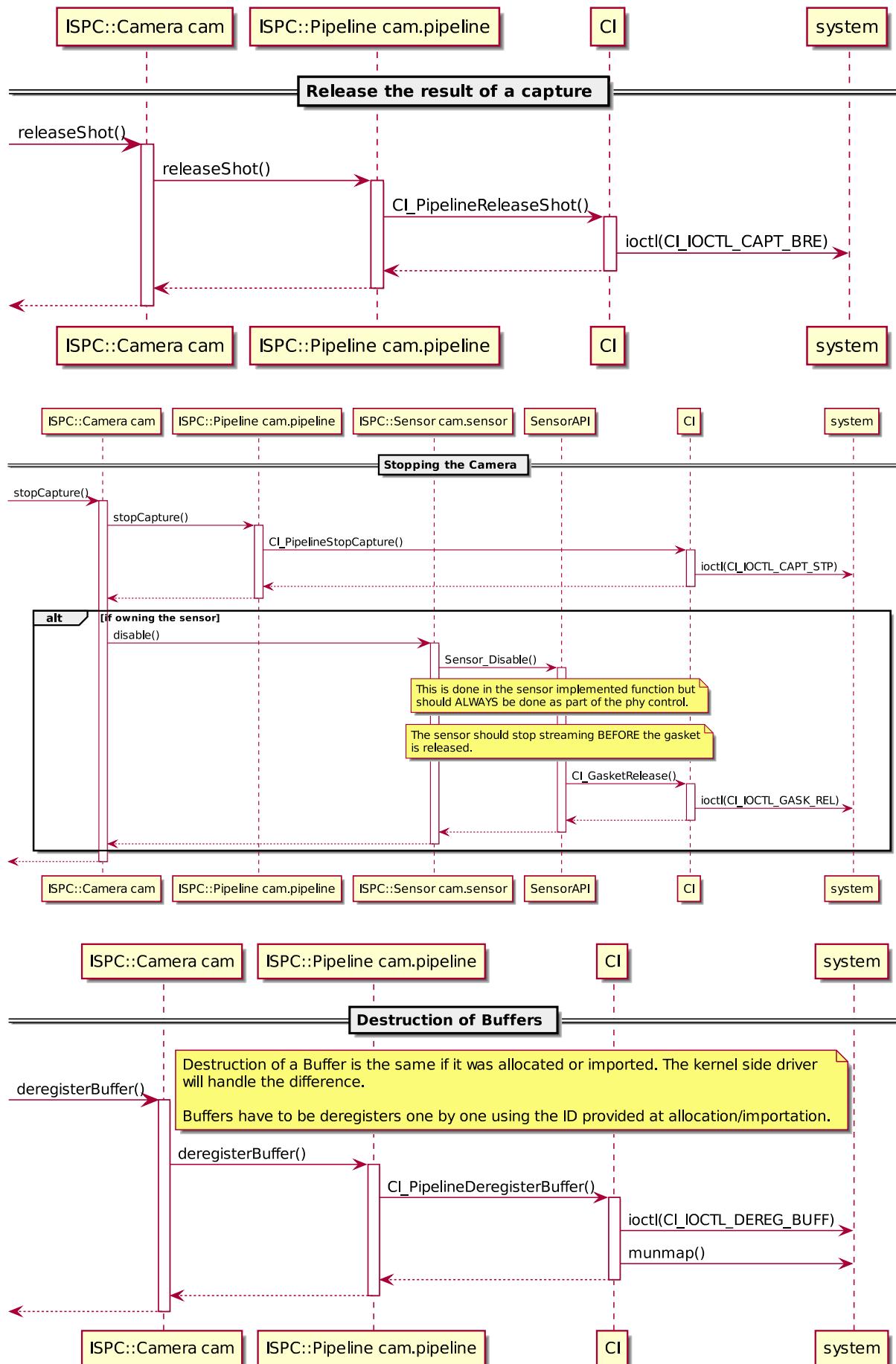


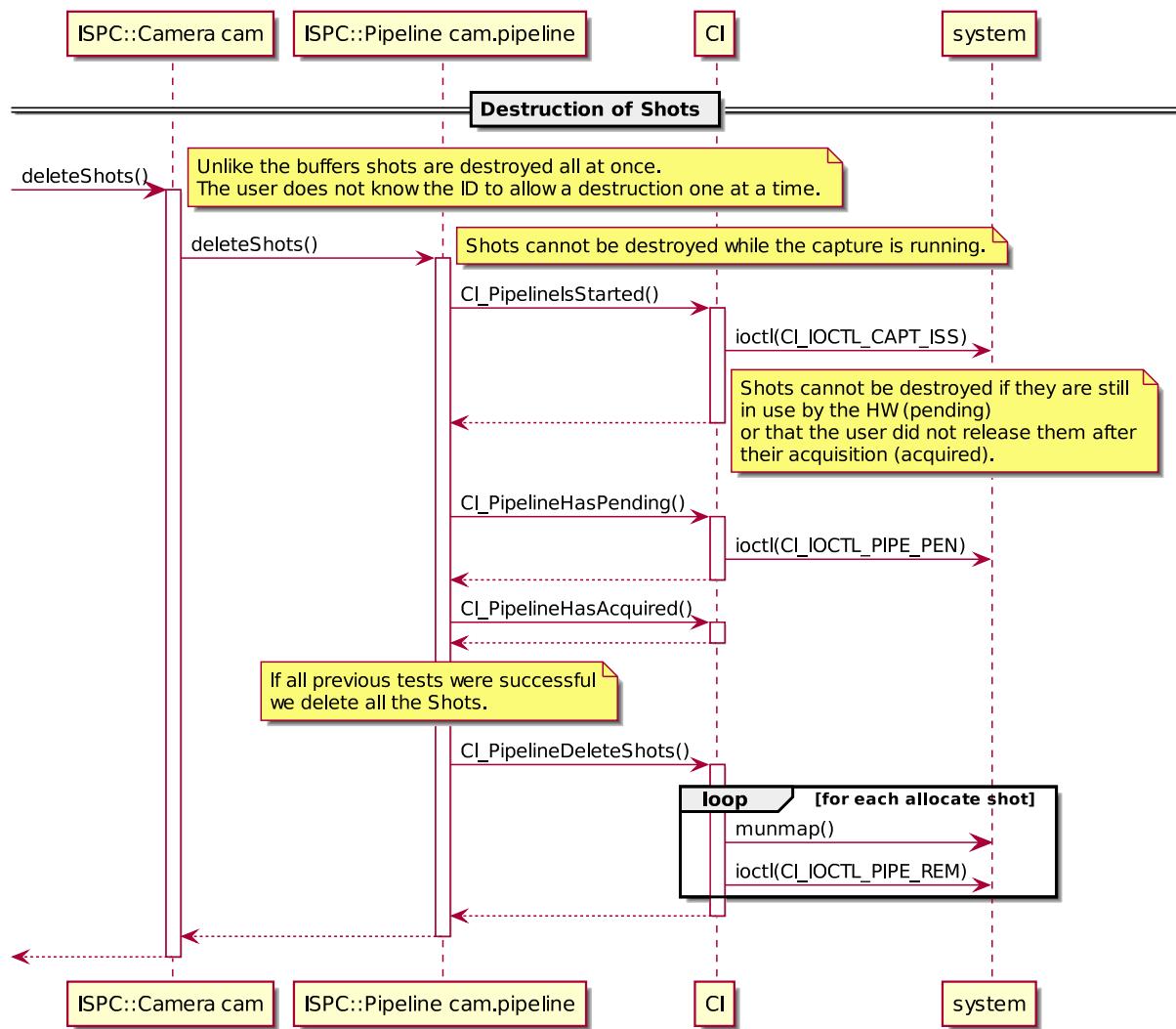


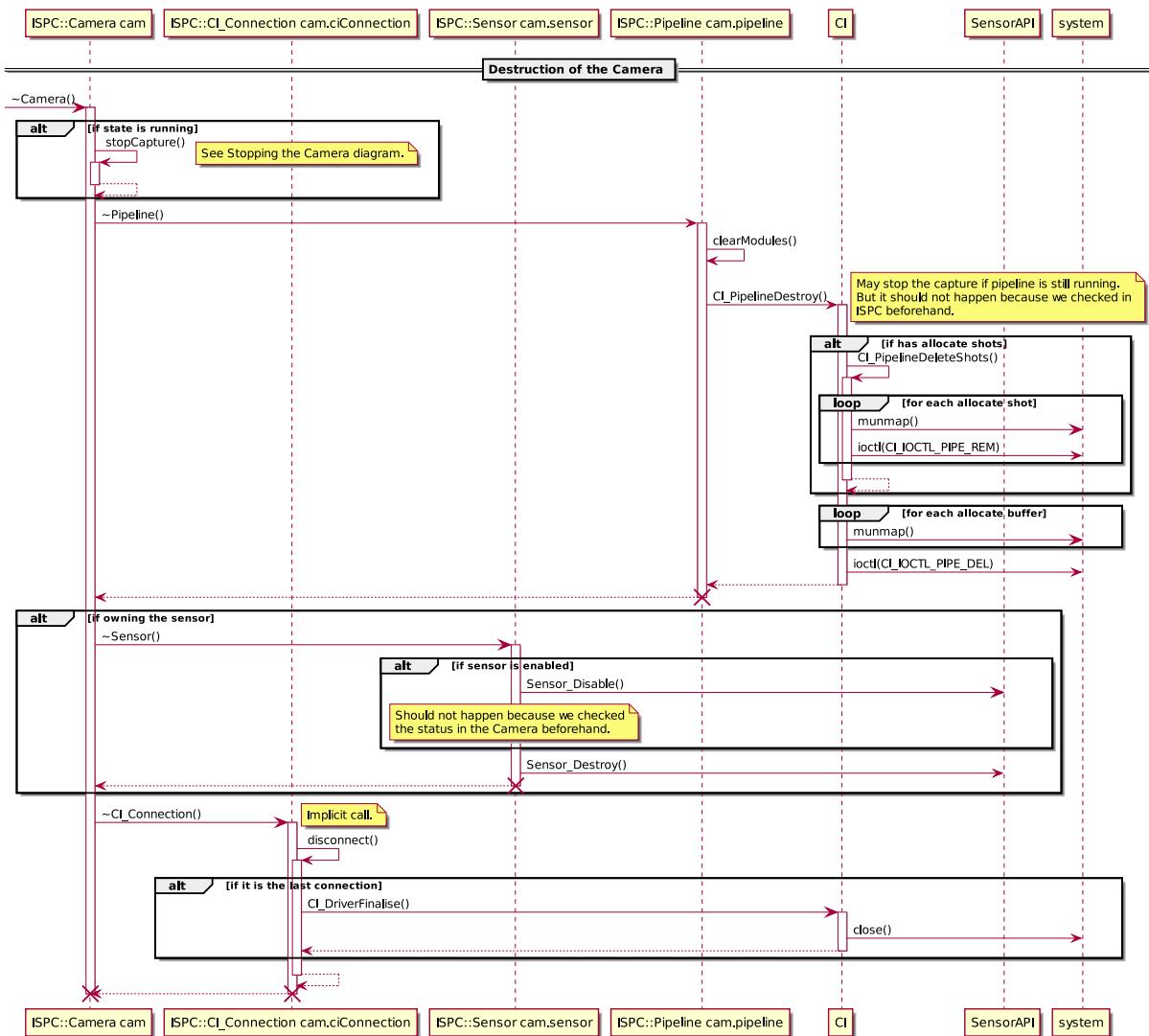












Chapter 21

Android Appendix: Debugging

21.1 Dumping Jpeg captures

JpegEncoder class has the capability of saving every compressed image as files. This debug option is very helpful when running CTS tests which do not store the actual result images.

To configure the camera HAL library to save the JPEG images to specific location, the system property **img.felix.jpeg.savepath** has to be set with the target location, writable by ‘media’ group. This has be done prior to loading the library.

For example, the sequence shown below will enable Jpeg dumps to `/data/felix/jpeg/` folder. Please note that these commands have to be executed from within target adb shell.

```
$ mkdir -p /data/felix/jpeg  
$ chown media:media /data/felix/jpeg  
$ setprop img.felix.jpeg.savepath /data/felix/jpeg/  
$ stop media  
$ start media
```

Chapter 22

Android Appendix: Live Tuning

This section shows how to use the *VisionLive* and *ISPC_tcp* (page 87) applications to tune the V2500 ISP under Android.

It is assumed that the reader is familiar with the VisionLive tool.

22.1 Enabling ISPC_tcp in Android

The current Android build doesn't compile ISPC_tcp by default. To enable the application you need to edit the Android.mk file located in DDKSource directory of the project. Following line has to be uncommented:

```
include $(FELIX_TOP)/DDKSource/ISP_Control/ISPC_tcp2/Android.mk
```

The next step is rebuilding Android and the re-creation of a new `system.img` file, which will contain ISPC_tcp. Assuming an Android build environment is already configured (see *Android Build Instructions* (page 325)) the operation is simple (for example building x86):

```
$ cd /path/to/android/root
$ source build/envsetup.sh
$ lunch android_x86-eng
$ make
```

After these steps the `system.img` should be loaded on the device. If using a PC with and FPGA refer to the *Prepare Android x86 images for GRUB* (page 330) section.

22.2 Running ISPC_tcp

This assumes that both ISPC_tcp and VisionLive are already built.

This also assumes that the device with the ISP HW is connected to the network and has an available IP address (`IP_ADDRESS`).

1. Start a connection in VisionLive following the normal steps
2. On the command line go to Android sources, activate environment and select combo:

```
$ source build/envsetup.sh && lunch aosp_x86-eng
```

3. Connect to the device with ISP (assuming it has an IP) to it using adb:

```
$ adb connect IP_ADDRESS
```

4. On Android console go to system/bin and run ISPC_tcp giving it the IP address of where VisionLive is running, and providing chosen TCP port. Choose camera sensor the platform is using, for example OV4688:

```
$ cd system/bin  
$ ISPC_tcp -sensor OV4688 -setupIP 10.80.2.91 -setupPort 2346
```

5. VisionLive should connect and live stream image should start.

Warning: Do not run the Android camera application if ISPC_tcp is running. Both will try to reserve the ISP HW and one will fail.