Sensor HAL Developer Guide

Table of figures

Revision History

Revision	Date	Comments
V1.0	Oct 13, 2013	First version for DxO ISP2013

Table 1: Document Revision History

1 INTRODUCTION

1.1 Use of the present document

The present document is confidential and subject to the terms of the NDA signed between DxO Labs and the recipient of the present document.

The present document is the property of DxO Labs and must be considered as DxO Labs background intellectual property. This document discloses a number of technical characteristics protected by patents filed by DxO Labs, and therefore subject to royalties whatever the product they are used in. The content of this document cannot be used for any purpose without a contract from DxO Labs.

References to third party tools or hardware in the present document are not an authorization or recommendation by DxO Labs to use such third party tool or hardware. The use of any third party tool or hardware is the responsibility of the recipient of the present document.

The present document is preliminary and may contain errors in the description that do not reflect the actual behavior of the product. DxO Labs will update the present document if and when such errors are detected.

1.2 Glossary

AG: Analogue Gain
DG: Digital Gain
FOV: Field Of View
HAL: Hardware Abstraction layer

1.3 References

Ref[1] DxO_ISP2013-fwIntegrationGuide
Ref[2] DxOISP_SensorAPI.h

2 SOFTWARE ARCHITECTURE

This section aims to provide recommendations and advices on how to write a sensor hardware abstraction library (HAL). This document is dedicated for sensor drivers integrated into the DxO Labs ISP product.
The test code must be written in C programming language.
For better understanding of this chapter, please refer to **Error! Reference source not found.**for detailed description of the sensor HAL interface.

2.1 Internal Sensor HAL architecture

2.1.1 Introduction

Each sensor provides many registers to handle its behavior. It is truly recommended to create functions managing one specific feature. It means that there are a couple of functions to set and get analogue gain, a couple of functions to set and get exposure time, and so on. Each of these functions may in turn affect several registers in the sensor.

All registers that a sensor device provides may be not used by the ISP. It means that the sensor HAL either does not write these registers (i.e. keep default values) or set these registers to fixed values when initializing the device.

Frame after frame, the way the ISP communicates with the sensor HAL is roughly as follow:
Step 1: LibDxOISP reads the current status of the sensor device
i.e.: 1: DxOSensor_GetStartGroup()
2a…2x: DxOSensor_Get()
3: DxOSensor_GetEndGroup()
Step 2: LibDxOISP performs some computation to define the new settings to apply to the sensor.

Step 3: LibDxOISP applies the complete new settings to the sensor
i.e: 1: DxOSensor_SetStartGroup()
2a…2x: DxOSensor_Set()
3: DxOSensor_SetEndGroup()

Thus, the sensor HAL developer must keep in mind that the two most important functions are:

- ***DxOSensor_GetStartGroup()***
  - The sensor HAL must read the current settings from the sensor device while called and compute information that can be returned through the sensor HAL interface.
- ***DxOSensor_SetEndGroup()***
  - The sensor HAL must compute sensor register values from information provided through the HAL interface and apply them to the sensor device.

2.1.2 Detailed design

As explained above, to ease development and debug, splitting the code by functionalities is a good practice.
Thus, the preliminary functions to write should be:
setAnalogueGain() : convert an interface gain value into sensor register(s) value

getAnalogueGain() : convert sensor register(s) value into an interface gain value

The following functions should also be developed:
set/getExposureTime()
set/getDigitalGain()
set/getImageOrientation()
set/getImageCropping()
set/getFrameRate()

Then, develop two functions that aim, on the one hand to write in one shot all the computed sensor device registers, on the other hand to read in one shot all the required sensor device registers.
Let's call them *writeSensorRegisters()* and *readSensorRegisters().*

DxoSensor_GetStartGroup() should have a canvas as:

```
{
	readSensorRegisters();
	getAnalogueGain();
	getExposureTime();
	getDigitalGain();
	getImageOrientation();
	getImageCropping();
	getFrameRate();
}
```

DxoSensor_SetEndGroup() should have a canvas as:

```
{
	setAnalogueGain();
	setExposureTime();
	setDigitalGain();
	setImageOrientation();
	setImageCropping();
	setFrameRate();
	writeSensorRegisters();
}
```

2.1.3 Cache implementation

Communication to and from sensor device registers are usually performed through I2C bus. As its bandwidth is pretty slow, it is recommended to implement a software cache mechanism avoiding to write or to read unnecessary registers values.
Moreover, some registers must not be handled by a cache layer because values are either dynamically modified by the sensor device to report a state or because the applied value may not be exactly the same as the re-read value because the sensor has not been able to perform the setting (that could happen for gain values, integration time…)

DxO Labs sensor HAL provided as example source code two kinds of explicit functions:

```
	readDevice(offset)
	writeDevice(offset,data)
	readCacheDevice(offset)
	writeCacheDevice(offset,data)
```

The choice of access must be done according to the sensor datasheet.
A good practice is to allow at compilation time the enabling and the disabling of this software layer.
For more detailed example, instantiation of such a cache is provided into the sensor HAL example provided by DxO Labs. This is performed through the C macro READ_BY_CACHE() and WRITE_BY_CACHE() that can be defined to either the one or the other register access definition.

2.1.4 Multiple sensor interface for LibDxOISP

DxO Labs ISP offers to plug up to two sensors at its inputs. The camera application handles the desired sensor by providing the id of the currently used one through the software interface. Whatever the number of sensor devices plugged on the system, the software interface with LibDxOISP remains the same (See Ref[2]).

The interface offers for instance:

```
void DxOISP_SensorInitialize (uint8_t ucSensorId);
```

A good practice for the implementation of the sensor HAL is to design as follows:

```
#include "sensor_hal_one.h"
#include "sensor_hal_two.h"

static void (*S_TabFunc_Initialize [])(void) = {
      #ifdef SENSOR_DEVICE_ONE
      sensorDeviceOneInitialize,
      #endif
      #ifdef SENSOR_DEVICE_TWO
      sensorDeviceTwoInitialize
      #endif
};

void DxOISP_SensorInitialize(uint8_t ucSensorId) {
      S_TabFunc_Initialize[ucSensorId]();
}

#ifdef SENSOR_DEVICE_ONE
#include "sensor_hal_one.c"
#endif

#ifdef SENSOR_DEVICE_TWO
#include "sensor_hal_two.c"
#endif
```

Figure 1: multiple sensors implementation

2.1.5 Sensor settings latency

2.1.5.1 Required streaming off

This chapter provides a detailed explanation of the specific function of the software sensor HAL interface DxOISP_SensorFire().
Some registers of a sensor device are immediately taken into account while written to the device. This is usually the case for functional values like digital gain or analog gain.
Registers dealing with the timing aspect like frame rate, image size, blanking may need the streaming to be stopped to have these new settings applied. Refer to you sensor datasheet for more information.

Thus, if streaming has to be stopped to apply settings, things have to be done in this order:

1- The implementation of DxOISP_sensorCommandGroupClose() stops the sensor streaming
2- The implementation of DxOISP_sensorCommandGroupClose() writes registers to the sensor device
3- The implementation of DxOISP_sensorCommandGroupclose() raise to “1” ts_SENSOR_Status.isSensorFireNeeded

Later on, the libDxOISP will then call the function DxOISP_sensorFire().

DxOISP_sensorFire() body must:

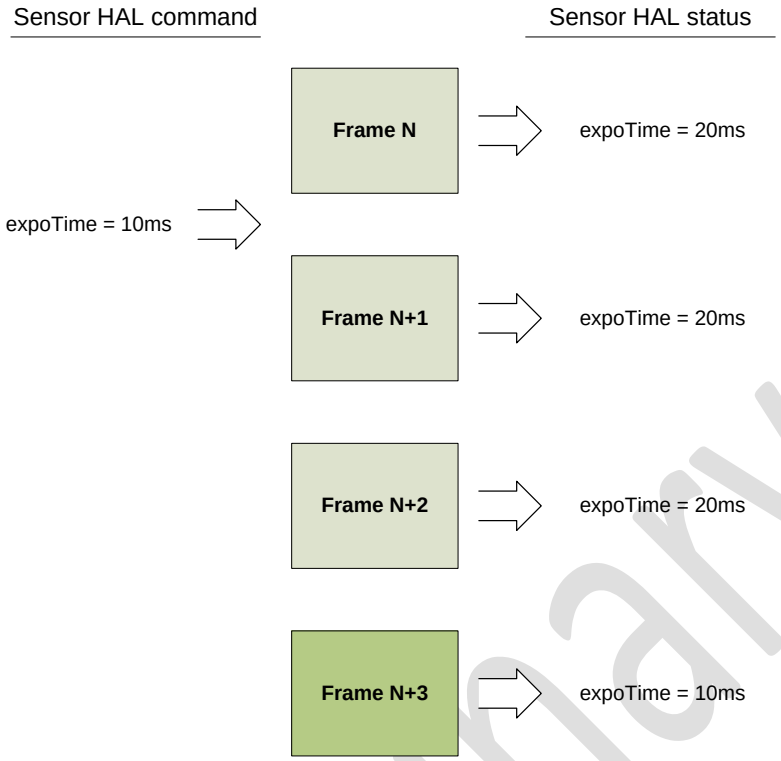
1- Fall down to “0” ts_SENSOR_Status.isSensorFireNeeded
2- starts the sensor streaming

2.1.5.2 Latency in returned settings

One of the most difficult things to implement into a sensor HAL is the taking into of latency for new settings applied. When a new analog gain (for instance) is set and written to the sensor, the sensor may apply the new gain couple of frame later. At the same while, the analog gain used by the sensor for the current output frame must be returned by the HAL. Thus, the sensor HAL must simulate the settings latency. Required information to instantiate this latency usually comes from the sensor datasheet. Some latency may depend on the number of settings changed and the same while and thus, sensor HAL has to be modified accordingly while validating it.

One appropriate way to handle these timing issues is to analyze the data in the MIPI data lines. This will guarantee an exact synchronization with the frame.

It is quite complex to describe a test validating the timings as it is extremely dependent on the complete bench architecture and its capabilities. The bench must be able to handle properly synchronization between grabbed frames and sequences of read and write sensors register to unitary perform that validation. The idea is to check the sequence described above.



Exposure time latency: two frames

Figure 2: setting latency description