****

***FPGA***

***SPECIFICATION***

**Project:** Iris3 (661)

**Document Title:** Iris3 FPGA specification

**Document Number and Revision:** 661-DEFP-004-0.20

**Author:** Jean-François Larin, ing #121322

Javier Mansilla

**Date:** 2015-10-15

**Matrox Confidential information.** Do not circulate outside Matrox without the prior consent of the Matrox Engineering department. Advance Information. Contains information on product in the design phase of development.

**Revision history**

**This section lists all revisions for the current document and explains the modifications made for each new revision (the table below is only given as an example):**

|  |  |
| --- | --- |
| 09/09/2014  (revision 0.01) | 1. Initial revision. |
| 22/09/2014  (revision 0.02) |  |
| 07/10/2014  (revision 0.03) | 1. Add sensor controller modes. |
| 21/10/2014  (revision 0.04) | 1. Added PCIe and DMA section. |
| 15/01/2015  (revision 0.05) | 1. Added Sensor powerup, subsampling and binning. |
| 09/03/2015  (revision 0.06) | 1. Added Interrupt definition. General review and rephrase since we know FPGA design is split in 2 FPGAs. |
| 10/03/2015  (revision 0.07) | 1. Reviewed Interrupt design to add local status/mask in register space. |
| 15/10/2015  (revision 0.10) | 1. Added design of interrupt queue, removed section on UART. |
| 20/04/2017  (revision 0.20) | 1. Changed queue design to leave the producer index as an inferred value from the content of the queue area. |

**Table of contents**

[1 Introduction 5](#_Toc505612568)

[2 Conventions 5](#_Toc505612569)

[3 Fpga Features 5](#_Toc505612570)

[4 Block Diagram 7](#_Toc505612571)

[5 Physical implementation of IRIS 3 fpga 8](#_Toc505612572)

[5.1 Artix 7 FPGA family 8](#_Toc505612573)

[5.2 FPGA Pin Count 10](#_Toc505612574)

[5.2.1 Python Sensor Interface 10](#_Toc505612575)

[5.2.2 IOs (One fpga version) 11](#_Toc505612576)

[5.2.3 IO’s (Two fpga version) 12](#_Toc505612577)

[5.2.4 I2C interface (Optional) 12](#_Toc505612578)

[5.2.5 FLASH SPI interface (Optional, G&O parameters) 12](#_Toc505612579)

[5.2.6 NC-SI Interface 13](#_Toc505612580)

[5.2.7 PCIe Interface 13](#_Toc505612581)

[5.2.8 Minimum Grab FPGA IO required (1 fpga implementation, no partitionning) 14](#_Toc505612582)

[5.2.9 Minimum Grab FPGA IO required (2 fpga implementation, SpiderIO in the Profiblaze FPGA) 14](#_Toc505612583)

[6 OnSemiconductor Python CMOS sensor family 15](#_Toc505612584)

[7 Spider Realtime IO 17](#_Toc505612585)

[8 Sensor Acquisition 18](#_Toc505612586)

[8.1 Sensor Controller 18](#_Toc505612587)

[8.1.1 Double buffering acquisition concept 20](#_Toc505612588)

[8.1.2 CMOS serial interface 22](#_Toc505612589)

[8.1.3 CMOS Python sensor powerup and LVDS training 23](#_Toc505612590)

[8.1.4 Subsampling and Binning with ROI’s 23](#_Toc505612591)

[8.2 Sensor Datapath 24](#_Toc505612592)

[9 DMA and PCIe interface 25](#_Toc505612593)

[9.1 DMA 25](#_Toc505612594)

[9.2 PCIe Target 27](#_Toc505612595)

[9.3 Message Signaled Interrupt 27](#_Toc505612596)

[9.3.1 Interrupt Queueing 28](#_Toc505612597)

[9.3.2 Iris3 Interrupt sources 29](#_Toc505612598)

[9.3.2.1 SOE (Start of Exposure) 29](#_Toc505612599)

[9.3.2.2 EOE (End of Exposure) 29](#_Toc505612600)

[9.3.2.3 SOS (Start of Strobe) 30](#_Toc505612601)

[9.3.2.4 EOS (End of Strobe) 30](#_Toc505612602)

[9.3.2.5 SOG (Start of Grab) 30](#_Toc505612603)

[9.3.2.6 EOG (End of Grab, aka End of DMA) 30](#_Toc505612604)

[9.3.2.7 ERR (ERRor) 30](#_Toc505612605)

[9.3.2.8 GBA (Grab Abort) 30](#_Toc505612606)

[9.3.3 Interrupt pending and per vector masks 30](#_Toc505612607)

# Introduction

The Iris3 FPGAs will be used in the Iris3 smart camera design. The functions of the FPGA are basically:

* Interface with the CMOS sensor to acquire the image and to control the sensor.
* Move that image into host memory into the format required by the host. This is done through a PCIe bus. This includes processing and reformatting the data.
* Accelerate Profinet functionality with the Real-Time Profiblaze processor design (see specification 661-arhw-002-x.xx).
* Control miscellaneous IO to interface with external world. This is trigger signal, real-time IOs like documented in Spider\_LPC documentation (see 627-arhw-003-xxxx).

The acquisition FPGA is named Athena and is connected between the image sensor and the PCIe bus. The control FPGA contains the Profinet acceleration and the real-time IOs and is named Ares.

# Conventions

**Integration** is defined as the interval during which the sensor is set to trap incident light and retain charge.

**Exposure** is the interval during which the sensor is exposed to incident light (by a shutter or by a strobe). The exposure is entirely independent of the integration.

**FOT** is the Field Overhead Time.

**EO-FOT** is the End of Field Overhead Time.

# Fpga Features

The proposed detailed feature set is shown below (subject to negotiation):

* Interface to OnSemi Python CMOS sensor, from VGA to 5MP size
  + Grab number of channel TBD, see discussion below
  + Programming interface through indirect programming of SPI interface
  + Color and Mono sensors
  + DeBayer compatible with MIL done in hardware for color sensor
  + 8-bit interface in the initial release; we should plan hardware design for 10-bit interface in second phase
  + Reverse in X direction available
* Host interface through PCIe interface
  + We consider using 1x or 2x PCIe interface, at Gen1 or Gen2 speed. We will need to balance power vs bandwidth and find the optimal solution with the FPGA selected
  + PCIe link configuration may have to change with effective data throughput (Frame rate, color space used), controlled by software or FPGA firmware
  + Will use MSI interrupts
* DMA engine writing host memory
  + No external memory used (we rely on host memory arbitration to guarantee bandwidth)
  + Color space conversion to the following formats:
    - Mono8 (for Mono sensors)
    - Mono8 (Y component, for color sensor)
    - BGR32 (wasting 25% BW)
    - YUV 4:2:2 (In full range)
    - Planar Mono8
* Real-time IO, SPIDER like:
  + Single Tick-Table
  + 3 general inputs
  + 1 trigger input (treated generally as another input)
  + 3 outputs
* One slow PWM output, converted to analog on the PCB.Profinet Acceleration using a Profiblaze
  + Connection to the outside world through the NCSI of the i210 Ethernet Controller
* EtherCAT may be considered, in slave or in master mode, possibly using another PCB assembly option
* Dedicated link to liquid lens control logic.

# Block Diagram

Figure 1 below shows the FPGA block diagram:



Figure 1 FPGA block diagram

Large arrows represent various internal busses or signal group, blue thin line is a register bus.

The reader can observe that the Profiblaze is only linked to the system through a register bus. That same design is currently being developed on the 4SightGPM platform using a LPC bus to connect to the host system. That sub-system is a good candidate for partitioning into a second FPGA. Also, the links between the acquisition interface and the Spider Realtime IO are few and slow speed (exposure, triggers?) This section is also a good candidate for partitioning.

# Physical implementation of IRIS 3 fpga

## Artix 7 FPGA family

The Xilinx’s Artix-7 may be used to implement the IRIS3 FPGA.

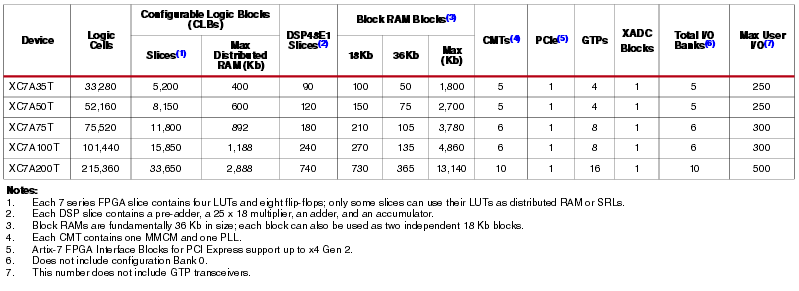


Figure 2 : Artix 7 Feature Summary

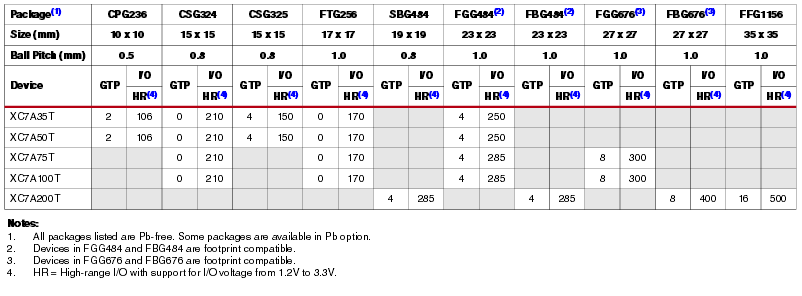


Figure 3: Artix 7 FPGA Device package and I/Os

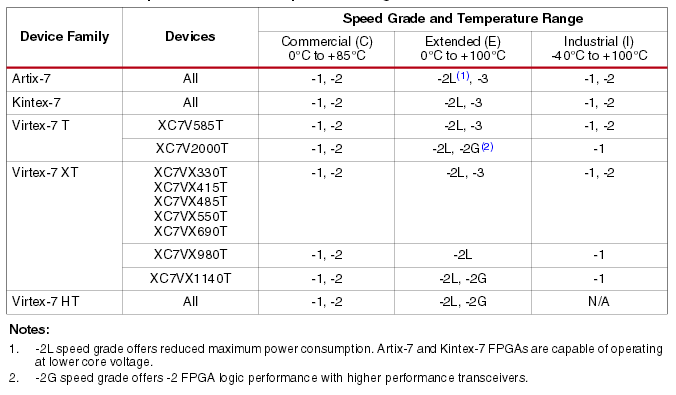


Figure 4 : 7 Series Speed Grade and Temperature Range

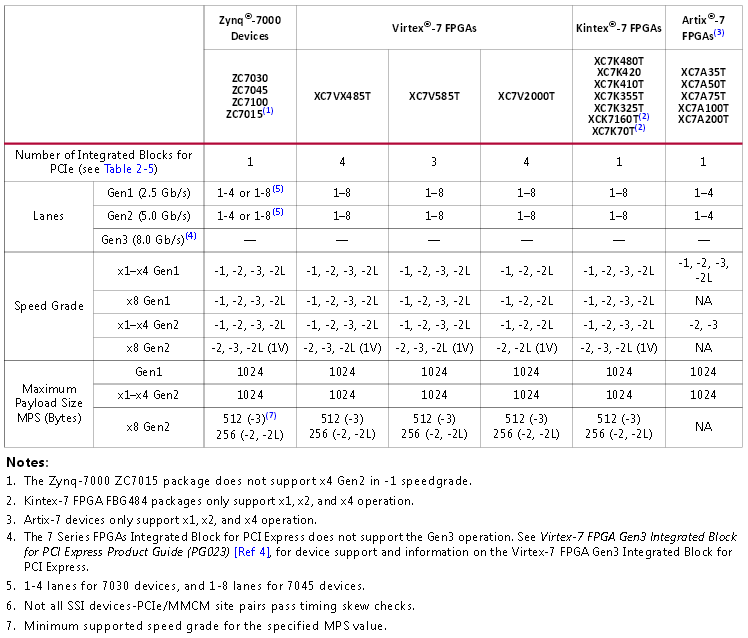


Figure 5 : PCIe Requirements for 7 Series

## FPGA Pin Count

### Python Sensor Interface

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | PIN Name | I/O Type | FPGA Pin Direction | Description |
| 1 | mosi | CMOS | Output | Sensor SPI Master OUT |
| 2 | miso | CMOS | Input | Sensor SPI Master IN |
| 3 | sclk | CMOS | Output | Sensor SPI Clock |
| 4 | ss\_n | CMOS | Output | Sensor SPI Slave Select (Active low) |
| 5 | clock\_outn | LVDS | Input | Sensor LVDS clock output Channel(N) |
| 6 | clock\_outp | LVDS | Input | Sensor LVDS clock output Channel(P) |
| 7 | Doutn0 | LVDS | Input | Sensor LVDS data output Channel #0 (N) |
| 8 | Doutp0 | LVDS | Input | Sensor LVDS data output Channel #0 (P) |
| 9 | Doutn1 | LVDS | Input | Sensor LVDS data output Channel #1 (N) |
| 10 | Doutp1 | LVDS | Input | Sensor LVDS data output Channel #1 (P) |
| 11 | Doutn2 | LVDS | Input | Sensor LVDS data output Channel #2 (N) |
| 12 | Doutp2 | LVDS | Input | Sensor LVDS data output Channel #2 (P) |
| 13 | Doutn3 | LVDS | Input | Sensor LVDS data output Channel #3 (N) |
| 14 | Doutp3 | LVDS | Input | Sensor LVDS data output Channel #3 (P) |
| 15 | Syncn | LVDS | Input | Sensor LVDS sync Channel output (N) |
| 16 | Syncp | LVDS | Input | Sensor LVDS sync Channel output (p) |
| 17 | Trigger0 | CMOS | Output | Sensor Trigger Input #0 |
| 18 | Trigger1 | CMOS | Output | Sensor Trigger Input #1 (HDR modes) |
| 19 | Trigger2 | CMOS | Output | Sensor Trigger Input #2 (HDR modes) |
| 20 | Monitor0 | CMOS | Input | Sensor Monitor Output #0 |
| 21 | Monitor1 | CMOS | Input | Sensor Monitor Output #1 |
| 21 | Reset\_n | CMOS | Output | Sensor Reset(Active low) |
| 22 | clk\_pll | CMOS | Output | Sensor Ref Clock for PLL |
| Optional connections | | | | |
| 23 | Doutn4 | LVDS | Input | Sensor LVDS data output Channel #0 (N) |
| 24 | Doutp4 | LVDS | Input | Sensor LVDS data output Channel #0 (P) |
| 25 | Doutn5 | LVDS | Input | Sensor LVDS data output Channel #1 (N) |
| 26 | Doutp5 | LVDS | Input | Sensor LVDS data output Channel #1 (P) |
| 27 | Doutn6 | LVDS | Input | Sensor LVDS data output Channel #2 (N) |
| 28 | Doutp6 | LVDS | Input | Sensor LVDS data output Channel #2 (P) |
| 29 | Doutn7 | LVDS | Input | Sensor LVDS data output Channel #3 (N) |
| 30 | Doutn7 | LVDS | Input | Sensor LVDS data output Channel #0 (N) |
| 31 | Lvds\_clock\_inn | LVDS | Output | Sensor LVDS clock Input (N) |
| 32 | Lvds\_clock\_inp | LVDS | Output | Sensor LVDS clock Input (P) |

### IOs (One fpga version)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | PIN Name | I/O Type | FPGA Pin Direction | Description |
| 1 | User\_in0 | CMOS | Input | User Input from opto-isolated device. |
| 2 | User\_in1 | CMOS | Input | User Input from opto--isolated device. |
| 3 | User\_in2 | CMOS | Input | User Input from opto-isolated device. |
| 4 | User\_in3 | CMOS | Input | User Input from partially opto-isolated device(Camera Gnd) |
| 5 | User\_Out0 | CMOS | Output | Configurable User Output |
| 6 | User\_Out1 | CMOS | Output | Configurable User Output |
| 7 | User\_Out2 | CMOS | Output | Configurable User Output |
| 8 | Led\_out0 | CMOS | Output | FPGA User Led Color0 |
| 9 | Led\_out1 | CMOS | Output | FPGA User Led Color 1 |
| 10 | Debug1 | CMOS | Output | FPGA Debug Pin 0 |
| 11 | Debug2 | CMOS | Output | FPGA Debug Pin 1 |
| 12 | Debug3 | CMOS | Output | FPGA Debug Pin 2 |
| 13 | Debug4 | CMOS | Output | FPGA Debug Pin 3 |
| 14 | Fpga\_var\_type0 | CMOS | Input | FPGA Variation Type 0 |
| 15 | Fpga\_var\_type1 | CMOS | Input | FPGA Variation Type 1 |

### IO’s (Two fpga version)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | PIN Name | I/O Type | FPGA Pin Direction | Description |
| 1 | Trigger\_in0 | CMOS | Input | Trigger Input from Spider fpga device. |
| 2 | Strobe\_out | CMOS | Output | Strobe Output to Spider fpga device. |
| 3 | Integration\_out | CMOS | Output | Integration Output to Spider fpga device. |
| 4 | Led\_out0 | CMOS | Output | FPGA User Led Color0 |
| 5 | Led\_out1 | CMOS | Output | FPGA User Led Color 1 |
| 6 | Debug1 | CMOS | Output | FPGA Debug Pin 0 |
| 7 | Debug2 | CMOS | Output | FPGA Debug Pin 1 |
| 8 | Debug3 | CMOS | Output | FPGA Debug Pin 2 |
| 9 | Debug4 | CMOS | Output | FPGA Debug Pin 3 |
| 10 | Fpga\_var\_type0 | CMOS | Input | FPGA Variation Type 0 |
| 11 | Fpga\_var\_type1 | CMOS | Input | FPGA Variation Type 1 |

### I2C interface (Optional)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | PIN Name | I/O Type | FPGA Pin Direction | Description |
| 1 | Smbclk | CMOS | Output | SMBus clock |
| 2 | Smbdata | CMOS | Bidir | SMBus data |

### FLASH SPI interface (Optional, G&O parameters)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | PIN Name | I/O Type | FPGA Pin Direction | Description |
| 1 | Spi\_sclk | CMOS | Output | SPI clock output. |
| 2 | Spi\_csN | CMOS | Output | SPI chip select output. |
| 3 | Spi\_sdout | CMOS | Output | SPI data output. |
| 4 | Spi\_sdin | CMOS | Input | SPI data input. |

### NC-SI Interface

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | PIN Name | I/O Type | FPGA Pin Direction | Description |
| 1 | Ncsi\_clk | CMOS | Output | Clock |
| 2 | Ncsi\_tx\_en | CMOS | Output | Transmit Enable |
| 3 | Ncsi\_txd[0:1] | CMOS | Output | Transmit Data |
| 4 | Ncsi\_rx\_crs\_dv | CMOS | Input | Receive Data Valid |
| 5 | Ncsi\_rxd[0:1] | CMOS | Input | Receive Data |

### PCIe Interface

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | PIN Name | I/O Type | FPGA Pin Direction | Description |
| 1 | pcie\_sys\_rst\_n | CMOS | Input | PCIe system reset |
| 2 | pcie\_sys\_clk\_p | MGTREFCLKP | Input | Dedicated MGT PCIe Ref Clock P |
| 3 | pcie\_sys\_clk\_n | MGTREFCLKN | Input | Dedicated MGT PCIe Ref Clock N |
| 4 | pcie\_tx\_p | MGTXTXP | Output | Dedicated MGT PCIe Tx P |
| 5 | pcie\_tx\_n | MGTXTXN | Output | Dedicated MGT PCIe Tx N |
| 6 | pcie\_rx\_p | MGTXRXP | Input | Dedicated MGT PCIe Rx P |
| 7 | pcie\_tx\_n | MGTXRXN | Input | Dedicated MGT PCIe Tx N |

### Minimum Grab FPGA IO required (1 fpga implementation, no partitionning)

|  |  |
| --- | --- |
| Interface | Number of pin |
| Sensor Interface (4x) | 22 |
| IO’s | 15 |
| SPI(Optional) | 4 |
| I2C(Optional) | 2 |
| Memory | 0 |
| PCIe(1x/2x) | 1 |
| NC-SI | 7 |
| Others | 0 |
| **TOTAL** | **51** |

### Minimum Grab FPGA IO required (2 fpga implementation, SpiderIO in the Profiblaze FPGA)

|  |  |
| --- | --- |
| Interface | Number of pin |
| Sensor Interface (4x) | 22 |
| IO’s | 11 |
| SPI(Optional) | 4 |
| I2C(Optional) | 2 |
| PCIe(1x/2x) | 1 |
| Others | 0 |
| **TOTAL** | **40** |

# OnSemiconductor Python CMOS sensor family

The Python CMOS sensors are 4.8umx4.8um square pixel. One single sensor PCB footprint supports 2 socket formats (48LCC and 84LCC); resolutions from 300K to 5MP can be supported with one single board design. All sensors have the same optical center on the PCB. Global shutter mode is supported with this sensor family. All sensors in the family will be available in monochrome and color versions. Improved NIR variants may be available in the future.

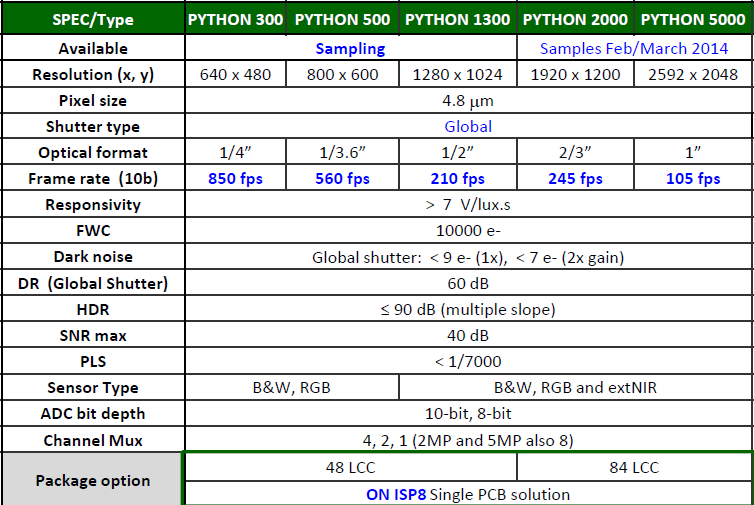


Figure 6 : Python CMOS family

The Python CMOS sensors family ca run in two different modes. The “Normal ROT mode” should be used to achive better optical performance, and lower power consumption. The Zero ROT mode gives us higher sensor frame rates, consumption will increase over the normal mode and the optical performance will degrade.

Table 1 Framerate of Python sensors operating in “Normal ROT mode”, best optical results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 8 LVDS Channels  BW Mono8  BW Mono10  BW RGB32 | 4 LVDS Channels  BW Mono8  BW Mono10  BW RGB32 | 2 LVDS Channels  BW Mono8  BW Mono10  BW RGB32 | 1 LVDS Channels  BW Mono8  BW Mono10  BW RGB32 |
| Python300 (640x480) | N/A | 620 fps  181 MB/s  363 MB/s  726 MB/s | 372 fps  108 MB/s  217 MB/s  435 MB/s | 206 fps  60 MB/s  120 MB/s  240 MB/s |
| Python500  (800x600) | N/A | 428 fps  195 MB/s  390 MB/s  780 MB/s | 249 fps  114 MB/s  228 MB/s  456 MB/s | 136 fps  62 MB/s  124 MB/s  248 MB/s |
| Python1300  (1280x1024) | N/A | 177 fps  221 MB/s  442 MB/s  884 MB/s | 98 fps  122 MB/s  244 MB/s  488 MB/s | 52 fps  65 MB/s  130 MB/s  260 MB/s |
| Python2000  (1920x1200) | 190 fps  417 MB/s  834 MB/s  1668 MB/s | 108 fps  237 MB/s  474 MB/s  948 MB/s | 58 fps  127 MB/s  254 MB/s  508 MB/s | 30 fps  65 MB/s  130 MB/s  260 MB/s |
| Python5000  (2592x2048) | 88 fps  445 MB/s  890 MB/s  1780 MB/s | 49 fps  248 MB/s  496 MB/s  992 MB/s | 26 fps  131 MB/s  262 MB/s  524 MB/s | 13 fps  65 MB/s  130 MB/s  260 MB/s |

Parallel Pixel clock is nominal 72 MHz. We will have to find what the product requirement is, and how to achieve best power efficiency (4 channels at 36 MHz or 2 channels at 72 MHz?) We also need to consider the required bandwidth for various color space compared to the available EFFECTIVE PCIe bandwidth.

The DMA is linked with the acquisition. We will have to verify, through experimentation on a valid use case system, what is the actual FIFO depth required to buffer the peak image rate and feed it on the PCIe bus. Since the Iris3 is a closed system, we can expect to have some software control over the arbitration of the memory cross-switch in the SOC.

# Spider Realtime IO

The Spider section is a reuse of the sub-modules of the Spider\_LPC, as documented in “LPC to uart and user IO fpga specification” with minor modifications:

* UART should not be included (see UART section for details)
* Number of input and output will be adapted to actual hardware, 2+1+1 inputs, 3 outputs
* Interface to the host should be in PCIe

It is still possible to consider connecting the Spider to the host using LPC, if the partitioning requires that functionality to be in a different FPGA than the acquisition/DMA unit and if we choose to use LPC to save power over a PCIe link.

We will have to watch how the Spider interrupt scheme can be ported onto MSI.

The RTIO timer’s functionality will be decoupled from the acquisition timers. The acquisition timers serving for exposure delay and strobe generation are much less complex than the RTio timers. The acquisition timer generating the strobe signal also needs a feedback from the sensor controller.



Figure 7 RTio vs Grab Timers

The Grab Timer 0 is used to delay the trigger received from the RTio logic. In conjunction with the Grab Timer 1, those two timers are used to generate a programmable strobe that may start before the exposure of the sensor. With the RTio logic the user will be able to add temporal and tic delays to the sensor trigger. The RTio will also be able to convert the strobe from the Grab Timer1 to a PWM pulse to drive external strobe circuits, if necessary.

# Sensor Acquisition

## Sensor Controller

The sensor controller acts like the master device for the CMOS sensor. It configures the CMOS sensor with a simple SPI interface, and it controls all the sequences send to the CMOS sensor. The primary function of the sensor controller is to organize the triggers received in a way the CMOS sensor will interpret them without any lost trigger or exposure change. The same acquisition modes supported in the IrisGT will be supported in the Iris3: continuous and triggered modes.

The triggered mode (software and hardware edge) supports simultaneous exposure and readout. The PET (Parallel Engine Trigger) receives the trigger and shift it in time until the end of the exposure ends fits with the end of the parallel readout. This logic was needed with the CCD sensors; the CCD sensor corrupts the image if the end of exposure arrives before the end of current readout. With the Python CMOS sensors, the sensor does not corrupt the image, but the exposure time is stretched until the end of the current readout. If this situation is not corrected, the consequence is a non-homogeneous images sequence (images with different levels).

In the Iris3 design, we will use this feature of the sensor. The trigger in level mode will be supported from an external pin, or RTio logic. The sensor controller will not be able to sequence this kind of trigger to the sensor, since the exposure time will be controlled by the length of the trigger pulse at its active level. The trigger received will not be shifted in time by the PET engine, and if the trigger input level de-asserts the exposure before the end of the current readout, the sensor will continue the exposure until the end of the current readout.

There are several features in the Python CMOS sensor family that will be used to reduce logic cost in the fpga, and increase performance and frame rate:

* Subsampling in monochrome sensors: Read-1-Skip-1 in X and Y direction
* Subsampling in color sensors: Read-2-Skip-2 in X and Y direction (Bayer code)
* Binning in monochrome sensors : 2x1 in X direction, Read-1-Skip-1 in Y direction
* Binning in color sensors: NOT available in SENSOR.
* Reverse Y
* 1 ROI (Multiple ROI per frame may be difficult to implement)
* HDR mode with up to 3 slopes to increase the dynamic range of the sensor (this feature may be optional)
* Test pattern implemented in the sensor
* Analog Gain
* Black Offset

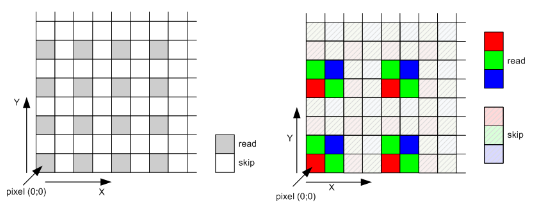


Figure 8 : Python sensor subsampling mode

Subsampling, binning, reverse Y and ROI will impact the datapath decoding logic, formatting and remapping sensor readout kernels will be needed to support those modes.

In the IrisGT camera, we support several subsampling modes in the X and Y directions without a framerate increase. In the case of using the Python subsampling mode (Read-1-Skip-1 in the monochrome cameras and Read-2-Skip-2 in the color camera) the framerate will increase, but we will lose the subsampling granularity.

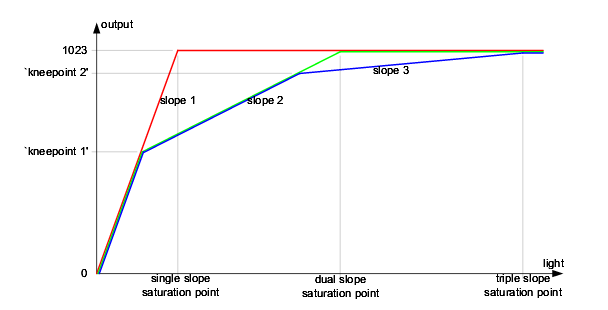


Figure 9 : Python sensor multiple slope integration (HDR), optional feature

### Double buffering acquisition concept

In the IrisGT, the concept of grab double buffering was only applied to the grab readout. The CCD controller was only capable of control one grab command at the time. In the Iris3, we can design a true double buffering acquisition without the use of a command buffer. Two grab commands will be pipelined in the logic with their own grab parameters. This method will allow applications like KNS back2back and Perceptron ROI’s to be easily implemented without the use of special features or custom modes.

The command queue list concept needs additional work in the fpga and in the software driver, compared with an implementation of a more simple double buffering acquisition. If in future, there is no plan to support more than 2 grab queued in the fpga, there is no need for such command queue. The software has already a built-in software grab queue. The use of a command queue list may be important when running with a very high framerate sensor (linescan). In the Iris3 family, there’s no plan to support line scan sensors. Keep in mind that our CMOS sensors maybe programmed with ROIs to increase frame rate. The Python 300, with a 640x8 lines ROI can achieve around 10883 fps; can this be a real application for the Iris3?

To start a grab in this mode, a WO register may be implemented. Let’s call it GRAB\_CMD. To map the fpga logic with MIL registers, a register called TRIGGER\_STATE may be implemented to differentiate a continuous grab command from a triggered grab command. When in TRIGGER\_STATE=1, a GRAB\_CMD acts like an ARM for a trigger. When the TRIGGER\_STATE register is set to ‘1’ a TRIGGER\_SRC register is needed to route the SW\_SNAPSHOT register or the HW\_TRIG to the internal logic. When in TRIGGER\_STATE=0, a GRAB\_CMD acts like a trigger.

In the fpga, the TRIGGER\_STATE register and the TRIGGER\_SRC register may be combined into one single register GRAB\_SRC[1:0]. GRAB\_SRC[0] maybe the TRIGGER\_STATE register and GRAB\_SRC[1] the TRIGGER\_SRC.

A GRAB\_CMD with GRAB\_SRC= 10 bin will indicate a continuous grab command.

A GRAB\_CMD with GRAB\_SRC= 11 bin will indicate a triggered grab command in software snapshot mode.

A GRAB\_CMD with GRAB\_SRC= 10 bin will indicate a triggered grab command in hardware mode.

The HW Level trigger is now supported with the IRIS3 family. Two modes are supported; the HW level Timed and the HW level exposure trigger width. When setting the register EXPOSURE\_LEV\_MODE to ‘0’, the exposure time used is programmed in the register EXPOSURE\_SS. The exposure-readout sequence will be repeated while the external trigger is at the level programmed in register TRIGGER\_ACT.

When setting the register EXPOSURE\_LEV\_MODE to ‘1’, the exposure time used is embedded in the trigger it-self. The exposure time is defined by external trigger at the level programmed in register TRIGGER\_ACT.

Two status registers will be implemented to inform on how many GRAB\_CMD are queued in the fpga: GRAB\_ACTIVE and GRAB\_PENDING. The GRAB command sequence starts with the GRAB\_CMD register write and ends at the end of the sensor FOT (EO-FOT, which happen after the end of exposure).

In the fpga, important grab registers like Exposures, Exposure Delay, Strobe Delay, Strobe Duration and also GRAB\_SRC will be double buffered. The sensor registers will also be double buffered in the fpga in the same manner. The current register set (registers in use) will be loaded from the registerfile register set when [GRAB\_CMD=1 and GRAB\_ACTIVE=0] or when [EO-FOT=1 and GRAB\_PENDING=1]. At the EO-FOT, if the GRAB\_PENDING status is set to 1, it will be reset and became the GRAB\_ACTIVE. Another GRAB\_CMD may be sent by the software at this time.

Sensor reprogrammation will always take place at the GRAB\_CMD or at the EO-FOT execution. The reprogrammation will take some time since the fpga transmit the new configuration with a serial interface to the sensor. This time will be absorbed by the minimum exposure time. The registers SENSOR\_GEN\_CNG, SENSOR\_GAIN\_ANALOG, SENSOR\_BLACK\_CAL, SENSOR\_ROI\_CONFx will be automatically transferred to the sensor before the start of readout. Programming a new gain, new subsampling, new RevY and new ROI, may take around 26 us (with a 10 MHz serial interface).

This is a typical sequence of grab commands in software triggered mode.



Figure 10 Triggered SW grabs

This is a sequence of one SW triggered grab command followed by one continuous grab command.



Figure 11 Triggered SW grab follow by continuous grab

To simplify the design of the CMOS controller, the entire controller logic will be clocked with the sys\_clock. All register will be synchronous to the clock in the fpga logic. To control the PET (parallel engine trigger) one important register is needed, and needs to be computed by the software for each grab command. The ReadOut Length is needed by the fpga logic to calculate where the integration may begin. The ReadOut Length is calculated by the following formula:

RdO\_length = [ RdO\_Incr+ (ROT(in pix clk)+Xend-Xstart) x (Yend-Ystart) x (Tpix)) ] / (Subsampling x Binning x Tsys)

All the grab timer registers will be programmed with the sys\_clock granularity domain (integration, integration delay, strobe delay, strobe duration). This opens the door to use the fpga with sensors with different pixel frequencies, without any modification in the driver. Running the entire controller module with the same clock domain makes easy the implementation of a remote sensor configuration (with SER/DES)

### CMOS serial interface

The CMOS serial interface is access by the FPGA with the registers ACQ\_SER\_CTRL, ACQ\_SERADDATA and ACQ\_SER\_STAT. The logic bloc is composed of three elements: the writer, the fifo and the reader. The instructions are sent through the writer to the fifo. The reader accesses the fifo to get the instructions, and executes them. In the IRIS 3, there are three possible instructions that can be programmed.

The first difference from the IRIS2 Family, is that the new reader does not start to read and execute the instructions from the fifo until the register SER\_RF\_SS is toggled.

The type of instruction is defined by register SER\_CMD. The first instruction is a CMOS sensor access; it can be a register read or write. The second instruction is a Timer. This instruction inserts a programmable delay in between two instructions; the SER\_DAT register will set the value of the delay. The third instruction is a STOP. This instruction makes the reader to stop reading from the fifo. The reader will wait for a new toggle of SER\_RF\_SS to continue the execution.

To make the dcf more powerful, a polling method for the driver should be created to be able to poll a status register before continue the execution of the dcf.

### CMOS Python sensor powerup and LVDS training

The fpga is responsible for the powerup sequence. When the register SENSOR\_POWERUP is toggled, the powerup sequence is started. It will first enable the 1.8V supply and wait for the 1.8V power good. When the 1.8V powergood is received, the logic will wait for 15us before enable the 3.3V supply. When the 3.3V power good is received, the logic will wait for another 15us before enable the VddPix supply. Again, the logic will wait for the VddPix power good and then wait for another 15 us. A timeout for the powergood has been implemented. If one of the powergood does not return before 2.1ms, the fpga will stop the power sequence.

After the VddPix is enabled and the 15us is done, the fpga will enable the sensor oscillator. The logic will wait for 15us before toggle the sensor ResetN. Another 15us will be added by the fpga after the de-assertion of the ResetN.

Register SENSOR\_STAT provides all information needed to follow or debug the power sequence. To start the power sequence, write ‘1’ to register SENSOR\_POWERUP. Then, poll the register SENSOR\_POWERUP\_DONE until the register is set to ‘1’. When SENSOR\_POWERUP\_DONE is set to ‘1’, the driver must read register SENSOR\_POWERUP\_STAT to see the results of the sequence.

If the powerup sequence fails, the camera must be restarted.

After a successful powerup sequence, the driver may load the dcf. The dcf will config the CMOS sensor and will enable the LVDS interface of the sensor.

When the LVDS interface of the sensor is enabled, the slave LVDS interface of the fpga must be trained. The first step is to de-assert the LVDS system reset by writing ‘0’ to register LVDS\_SYS\_RESET. Then poll register LVDS\_RDY to be sure that the IDELAYE2 module is calibrated. When the IDELAYE2 is calibrated, then the LVDS training may start. To start LVDS training, write ‘1’ to register LVDS\_START\_CALIB. This step will start a phase detection calibration followed by a bitslip calibration. Poll LVDS\_ACT register until this register is set to ‘0’. At this time the LVDS training is finished. Verify the result of the training with register LVDS\_CALIB\_OK.

### Subsampling and Binning with ROI’s

Iris3 will support the subsampling and binning inside the Python sensor. There are few constraints to satisfy to use Subsampling or Binning modes. To read out the image data through the output channels, the pixel array is organized in kernels. The kernel size is eight pixels in x−direction by one pixel in y−direction for the PYTHON 0.3/0.5/1.3 Megapixel sensors. The kernel size is sixteen pixels in x−direction by one pixel in y−direction for the PYTHON 5.0/2.0 MegaPixels.

* The Subsampling mode and the Binning mode cannot be used at the same time.
* XStart with Subsampling OR Binning: if a ROI is used with Subsampling or Binning, the start kernel and the start line MUST be EVEN.
* XEnd with Subsampling OR Binning: if a ROI is used with Subsampling or Binning, the end kernel MUST be ODD.
* YStart with Subsampling OR Binning: if a ROI is used with Subsampling or Binning, the start line MUST be EVEN.
* YEnd with subsampling OR binning: If a ROI is used with Binning, the end line must be ODD.

## Sensor Datapath

The following figure shows the Acquisition Interface data flow of a monochrome sensor in LVDS 4x configuration.



Figure 12 Monochrome Acquisition datapath interface block diagram

The following figure shows the Acquisition Interface data flow of a color sensor in LVDS 1x configuration.



Figure 13 Color Acquisition datapath interface block diagram

We have design choices to make related to the sensor connection. We can connect between 1 to 8 channels (in power of 2), yielding to various maximal frame rates, as shown below in Table 1 on page 16.

# DMA and PCIe interface

The PCIe interface enables the host processor to interact with the acquisition FPGA. It gives access to the host to a register interface through a single PCI BAR. This interface also pushes the image data to the host memory. Finally, this interface is used to transmit interrupts to the host.

## DMA

The DMA receives data in lines from the acquisition section and pushes it on the host system as fast as possible. The clock domain crossing is done through a shared memory, as shown above in Figure 12 and Figure 13. That shared memory must hold at least 2 lines of data. To support the worst case color space (BGRa32) with the largest sensor we need:

That can be fitted in 8 BRAM, if we round up to the next power of 2 to keep each line aligned on power of 2 and each component (in the planar case) aligned on a power of 2. We *could* consider the use of only 6 BRAM and strange address generation at design time if the design becomes limited by the number of BRAM.

The DMA engine is slave to the Acquisition Controller. It receives the DMA parameter from the controller. Those parameters are:

* Host write address (up to 64 bits, up to 3 addresses for planar mode).
* Host line pitch (16 bits)
* Host line size (sensor dependant maximum, up to 14 bits)

The handshake between the sensor clock domain and the PCIe clock domain is done on a line basis. This removes the complexity and imprecision of fifo count going from one clock domain to the other. This adds one-line latency to the transfer, which is a great improvement over the IrisGT implementation (about 1/3 of the frame latency, controlled by dividing DMA in multiple regions). This is shown below:



Figure 14 Pixel to PCI clock domain handskake

When a whole line is written in the shared ram, the logic on the pixel clock domain raises a Line Ready signal. That signal is resynchronized into the PCIe clock domain which starts the PCIe transfer of the line. At that time, the DMA parameters described above (host address, etc.) must be stable. The Line Ready must stay active.

When the transfer is completed on the PCIe, the DMA asserts the Line Transferred signal. The pixel clock domain logic then acknowledges that completion by lowering its Line Ready signal. The DMA, upon detecting a de-asserted Line Ready, then de-asserts its Line Transferred. The cycle can start again when the next line has been completely written in the shared memory.

All the formatting and color space conversion is done at the entrance of the Simple Dual Port shared memory, so the DMA engine does not have to alter the data in any ways: it does not suppress some bytes at the beginning of the line, does not subsample the data and does not re-order data.

## PCIe Target

A single BAR gives access to FPGA registers. It is prefetchable. It supports write up to the maximal TLP size. It supports read for up to the value allowed by the PCIe specification, 4 kbytes. In all cases, the PCIe target engine splits all accessed into multiple accesses (32 or 64 bits) on the internal register bus accessing the generated register file.

During design phase, we need to watch for non-posted receive limitation. As per PCI transaction ordering rules, we must not delay the acceptance of a posted (memory write) transaction because we cannot complete a posted (return read data) transaction.

## Message Signaled Interrupt

The interrupts are signaled through the MSI mechanism. That mechanism enables the host to receive multiple vectors without going through the time-wasting process of reading interrupt status register in the PCIe device. The FPGA can request up to 32 interrupt vectors. Each vector supports Per-Vector Masking.

Sadly experimentation has shown that the operating system cannot allocate 8 vectors on the Iris3 hardware. Furthermore, there is software/hardware architectural problem with multiple vectors, as described in Interrupt Queueing section below. Still, the in MSI mechanism is used, but with a single vector.

The diagram below shows the flow of interrupt:



Figure 15 Interrupt flow diagram

Various sources in the FPGA can generate event to interrupt the host processor. The events are punctual by their nature. The Rising Edge detection block ensures that any slow signal does not hold the status register in active state.

For every event source, there is a Mask Register that can be used by the driver to disable the generation of interrupt from that event source. When an event occurs and it is not disabled, it will set a status register bit.

In the PCI configuration space, there is Per Vector Mask for every MSI vector. This is a standard mask bit that can be controlled by the operating system, mapped on function call from the driver, or not. If an interrupt status bit becomes asserted while the vector is not masked (the normal situation), the PCIe core will send a MSI to the host. If an interrupt status bit becomes asserted while the vector is masked, a Pending bit, visible in the configuration space, will also become asserted and no MSI will be sent. From that situation, two outcomes are possible: the software could reset the Interrupt status bit, thereby resetting the Pending bit. Also, the software could re-enable the MSI by deasserting the Per Vector Mask, which will generate a MSI and reset the Pending bit.

### Interrupt Queueing

PCI rev 3.0 specification contains an implementation note about MSI (Servicing MSI and MSI-X Interrupts). It describes a scheme where interrupts are queued in host memory:

“A rather sophisticated but resource-intensive approach is to associate a dedicated event queue with each allocated vector, with producer and consumer pointers for managing each event queue. “

We implement that scheme to reduce the interrupt processing time by removing the read to the device. This also enables the driver to know in what order interrupt occurred, up to a certain precision and for a limited amount of time. This model fits with the Matrox patented driver framework where using multiple MSI or MSI-X for every interrupt source is incompatible with the exclusive access requirement of that same driver framework.

Interrupt coalescing is still possible under irregular conditions but much less likely to occur in normal conditions. Figure 16 below shows the data structure of the queue:

Interrupt Queue

PROD\_IDX

4kB

Register Space

CONS\_IDX

ADDRESS

ENABLE

NB\_DW

iQ

Source vector(N-1 :0)

~~Interrupt Queue~~

TLP

~~Interrupt Queue~~

interrupt

Figure 16 Interrupt Queue Data Structure and block diagram

The interrupt queue module receives a vector of all interrupt sources. Typically, the vector will be the concatenation of all interrupt status register, but other interrupt source that does not need to be reset can also be used. The goal is to include all sources, including the secondary interrupt status register, in such a way that the driver does not need to read from the PCI device to know how to handle the interrupt.

Whenever there is a rising edge on an interrupt source, the interrupt queue module will latch that event. When there is at least one latched event, the interrupt queue will try to write the latched vector into the interrupt queue located in the host memory. The queue of 4 kB long is configured by the driver through the address register. The interrupt queue module check if there is room by comparing the Producer pointer, an internal counter in the FPGA, and the consumer index located in the device register space. If there is enough space, it writes all the interrupt sources latched through the TLP interface (Transaction Layer Protocol), rounded-up to the next 32-bit boundary and rounded up to a power of 2. Then it updates its Producer Index and generates an interrupt to the host through normal interrupt mechanism (legacy INTa\_n, MSI or MSI-X).

The driver needs to clear the whole queue space to 0 before enabling the interrupt queue. The driver, upon receiving the interrupt has to infer the value of the producer index. The driver needs to read the next element (32 or 64 bits) after its shadowed consumer index and check that there is at least one bit set. If the element is non-zero, it means it has been written by the queue engine and it belongs to the driver. The driver must treat those events, clear back the element to 0 and then update the consumer index (CONS\_IDX). The driver should then check the next element in the queue to confirm that it is 0, which implies this element is pointed by the producer index and belong to the hardware, or that the element is non-zero and belongs to the driver. Since the number of element in the Source vector can be larger than 32, the driver has to divide the number of DW in the queue by the number of DW per element (NB\_DW, a static value) to find the number of interrupt elements in the queue.

The event source input in the queue can be an event (something that happens periodically) or an interrupt status bit that becomes asserted when an event occurs. In the case where an interrupt status bit is used, the driver should acknowledge that bit in the processing of the queue elements so rising edge of the bit can occur again and event can be signalled through the queue to the driver again. This is design dependant and should be negotiated between software and hardware designers.

The mapping between the interrupt source vector position and the interrupt source and status bit position in register space is design dependant. When some backward compatibility is desired, the FPGA designer should take care not to reuse the same bit for different purposes. The NB\_DW static register should be used to expand the interrupt source vector if new interrupt source are needed. Hence, the driver should read NB\_DW in the course of its initialization to be future proof.

Interrupt moderation scheme could also be implemented in the interrupt queue module, if required.

### Iris3 Interrupt sources

In a general PCIe adapter board, it is possible that the operating system does not allocate as many vectors as the PCIe device ask. Then, the FPGA and driver must alias many interrupt sources onto a reduced number of vectors (one in the case of recent Windows versions). Since we control both the FPGA design and the host system, we don’t want to implement aliasing. The vector 0 however can be used for interrupts related to power management event (PME) or Hot Plug events. Although we do not expect to use such events, we could we reserve vector 0 for those events and let the interrupt queue drive vector 1.

The interrupt source and vector assignment will also be documented in register file by the definition of the interrupt status and mask registers

#### SOE (Start of Exposure)

This source fires an interrupt at the beginning of the exposure. Refer to Figure 10 above.

#### EOE (End of Exposure)

This source fires an interrupt at the end of the exposure on the sensor. The triggering mode and trigger pattern may cause the actual exposure to be extended, because there is a readout taking place. This event will be delayed until the end of the actual exposure. This could be used in the end application to notify software that the object of camera can be moved, for example.

#### SOS (Start of Strobe)

This source fires an interrupt coincident with the turn-on of the strobe output.

#### EOS (End of Strobe)

This source fires an interrupt at the end of the active part of the strobe. This could be used in the end application to move the camera for example.

#### SOG (Start of Grab)

This source fires an interrupt at the beginning of the acquisition of valid data by the FPGA, during the readout. The SOG interrupt happens after the FOT and after the top blanking lines are transferred to the FPGA.

#### EOG (End of Grab, aka End of DMA)

This source fires an interrupt at the end of the acquisition of the image. This indicates that all data has reached the host memory. The driver does not have to read any register in the FPGA before it processes the input frame.

#### ERR (ERRor)

This source fires an interrupt when a catastrophic error occurs in the FPGA. This error source should only be fired in catastrophic failures where the driver cannot do anything to recover the situation. Occurrence of those errors should be notified to FPGA designers since it is expected that deep level hardware debugging will be needed to fix the problem.

This is the error that needs to be triggered on PCIe register space fifo underrun for example. If that condition occurs, read and write to the memory space will be corrupted from the overrun time. By design, the hardware must guarantee that this situation never occurs. If it occurs, the only thing the driver can do to recover is to reset the whole FPGA through the PCIe reset.

#### GBA (Grab Abort)

This source fires an interrupt to notify the driver that the abort process is completed. When the driver request an abort on an already sent grab command, it may take some time before the acquisition engine is ready to receive a new command. This delay can be related to the readout process that cannot be interrupted, for example. Since the hardware is aware of the uninterruptible ongoing process, it will notify the software when normal operation can resume.

### Interrupt pending and per vector masks

Please refer to PCI 3.0 specification, section 6.8.1 for the MSI capabilities standard structure. The mask bit definition can be found in section 6.8.1.7. The Pending Bits can be found in section 6.8.1.8. Since those bit definitions are standard, we expect that manipulation of those configuration bits is done through the operating system API. On the side, the interrupt status and mask register (which should be renamed ENABLE?) in the FPGA register space is accessed directly by the driver.

**APPENDIX A: RELATED DOCUMENTS**

Matrox Imaging, LPC to uart and user IO fpga specification. 627-arhw-003-0.18, October 10, 2013, <http://brainstorm/knowledgeTree/view.php?fDocumentId=2533>

Matrox Imaging, Profinet FPGA Specification. 661-arhw-002-0.02, April 4, 2014, <http://brainstorm/knowledgeTree/view.php?fDocumentId=2619>

PCI-SIG, PCI Local Bus Specification, Revision 3.0, August 12, 2002, [\\Urgent\mgihard\Public\Specifications\_&\_Standards\PCI\_Spec\PCI Rev 3.x\pci\_lb\_3\_0\_12AUG02.pdf](file:///\\Urgent\mgihard\Public\Specifications_&_Standards\PCI_Spec\PCI%20Rev%203.x\pci_lb_3_0_12AUG02.pdf)