DxO ISP2013
Hardware test vector managed through register access description

Revision History

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

Table 1: Document Revision History

Table of contents

Table of figures

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.

1.2 Purpose

This document details the content of a single hardware level test scenario delivered by DxO Labs and intended to be run by the customer.

1.3 References

Ref[1] DxO_ISP2013-hardwareIntegrationGuide

2 HARDWARE TEST SCENARIO

2.1 Overview

A hardware level test scenario is composed of:

1. A set of DxO ISP register writes and reads that performs DxO ISP configuration or status verification (for read access)
2. One or several reference images to provide at the input of DxO ISP
3. One or several reference images to collect at the outputs of DxO ISP
4. Several synchronization points that are intended to sequence actions on the bench.

The benefit of this approach is that the verification of the DxO ISP integration into the customer's hardware at system level can be then fully achieved without managing it via the DxO ISP firmware running on a dedicated micro controller that is usually done very late during software system level.

Furthermore, each hardware level scenario is automatically generated from a high level DxO ISP2013 application (internally developed at DxO Labs' site) in order to create some realistic use cases for hardware simulation.

In order to run one DxO hardware test vector in his test environment, the customer will have mainly to:

1. implement the functions used into DxO test vectors that fits to its verification environment,
2. feed and collect input and output images through the specified interfaces at the appropriate time,
3. run sequential checks of value returned by hardware register read,
4. check collected images and register status to declare the test passes or not.

2.2 Scenario description

Each test scenario is described into a single text file. The scenario is expressed in C programming language syntax for automation convenience. Each test may thus be simply executed. The contents of the C functions have to be implemented by the customer for its specific bench. A short example of a test scenario canvas is given below.

```
setDxOIp(&context,(uint32_t*)(0x00104000+DXO_IP_BASE_ADDR),0x00000001) ;
setDxOIp(&context,(uint32_t*)(0x00108804+DXO_IP_BASE_ADDR),0x333FFF00) ;

…
val = getDxOIp(&context,(uint32_t*)(0x00104000+DXO_IP_BASE_ADDR)) ;
assert(val!=0xdeadbeef)
…
// configuring output channel
{
  int IOid        = E_DxOIP_OUTPUT_DISPLAY ;
  int width       = 640 ;
  int height      = 480 ;
  int sampleWidth = 8 ;
  setDxOIo(&context,IOid,width,height,sampleWidth) ;
}

val1 = getDxOIp(&context,(uint32_t*)(0x00104008+DXO_IP_BASE_ADDR)) ;
val2 = getDxOIp(&context,(uint32_t*)(0x0010400A+DXO_IP_BASE_ADDR)) ;

…

// configuring input channel (and send pixel samples)
{
  int IOid        = E_DxOIP_INPUT_SENSOR ;
  int width       = 656 ;
  int height      = 512 ;
  int sampleWidth = 10 ;
  setDxOIo(&context,IOid,width,height,sampleWidth) ;
}

…

// wait for completion on the output display
waitDxOSync(&context,E_DxOIP_OUTPUT_DISPLAY) ;
```

Figure 1: test scenario example

2.3 Context explanation

Every single function of hardware level scenario owns as first argument a context.
Its declaration within a function prototype is `void* ctx;`
This pointer has to be understood as a pointer to a private buffer that the customer may use to keep useful information from one call to another or any other convenience.

2.4 DxO IP hardware access

DxO IP may be accessed through two functions:

- `setDxOIp()`
- `getDxOIp()`

It is up to the customer to define the body of these functions to access the DxO IP depending on the customer's system.
From a software point of view, these functions are considered as system calls. They must not be preempted by any other thread or process that may access DxO IP.

All addresses within a scenario are C typed `uint32_t` defined by inclusion of the `<stdint.h>` system header file.
All data are C typed `uint32_t` as DxO IP is accessed either in read or write by 32 bits wide word.

2.4.1 setDxOIp()

This function indicates that a write access has to be performed to DxO IP.

Prototype:

```
void setDxOIp (
      void*      ctx
,     uint32_t   addr
,     uint32_t   data
) ;
```

`ctx` : pointer to private customer's working buffer
`addr` : physical address in customer's chip bench, expressed in bytes
`data` : 32bit word to write

Requirement:

- The chip vendor must define a function that writes the DxO IP in a 32bit access.
- `ctx` has to be declared (even to NULL) to get the scenario compiling.
- DXO_IP_BASE_ADDR has to be defined by chip vendor to provide the mapped base address of the DxO IP. Addr are typically expressed as `DXO_IP_BASE_ADDR+0x000ABD0`
- This function expects the write access succeeds. In case of error, customer must exit and resume the scenario and assume it failed.

2.4.2 getDxOIp()

This function indicates that a read access has to be performed from DxO IP and must return this value.

Prototype:

```
uint32_t getDxOIp (
      void*      ctx
,     uint32_t   addr
) ;
```

`ctx` : pointer to private customer's working buffer
`addr` : physical address in customer's chip bench, expressed in bytes

This function returns the 32bit word that has been read at `addr`.

Requirement:

- The chip vendor must define a function that read the DxO IP in a 32bit access.
- `ctx` has to be declared (even to NULL) to get the scenario compiling.
- DXO_IP_BASE_ADDR has to be preliminary defined by chip vendor to provide the mapped base address of the DxO IP.
- This function expects the read access succeeds. In case of error, customer must exit and resume the scenario and assume it failed.

2.5 Input and Output pixel configuration

Usually input of DxO IP is fed with a sensor streaming on Pixstream bus. Some outputs (typically output to a display or a JPEG encoder) are also sent out through a Pixstream bus.
Some inputs or outputs may also be handled by a memory mapped in read and write access through internal DMA controllers monitored by DxO ISP2013 firmware. Such case is described in 2.6
In the context of a hardware level scenario, these inputs or outputs must be respectively read from a file or checked against a file. These files are provided along with a scenario by DxO Labs.

2.5.1 Pixstream setting – setDxOIo()

One single function is declared to handle the input and output settings through Pixstream. Depending on the I/O, the test environment will have to either send pixels from a file to the Pixstream input or collect pixels from a Pixstream output for a subsequent verification with a file.

Prototype:

```
void setDxOIo (
      void*            ctx
,     char const*      file
,     uint32_t         IOid
,     uint32_t         width
,     uint32_t         height
,     uint32_t         sampleWidth
) ;
```

`ctx` : pointer to private customer's working buffer
`file` : name of the file containing pixels to feed or pixels to check against future collected pixels (see chapter 3.2.1 for file content description).
`IOid` : Input or output identifier to handle
`width` : width in pixel sample of the buffer to provide or collect
`height` : height in pixel sample of the buffer to provide or to collect
`sampleWidth` : indicates the bit width of pixel sample handled at this stage of the test

Requirement:

- this function must:
  - configure the buffer for pixel sample reception if IOid refers to an output channel
  - feed with pixel samples the channel if IOid refers to an input channel

***Caveat*:** for integration ease, these functions are always called before any pixel sending. In case of output setting, this function does not mean “wait for output pixel” but merely prepare the environment for subsequent streaming out.

2.6 Memory configuration

In case of input or output handled by memory access through DMA management, the hardware level scenario requires to initialize with specific value a particular segment of memory or to save into a file the content of a particular segment of memory.

2.6.1 Memory description information

Caveat: All memory addresses handle dealing with memory under DMA management are always assumed to be in a logical space.
Caveat: Each segment of memory defined by its first and last byte address are assumed to logically contiguous.

A segment of memory is described through the following structure:

```
typedef struct {
      uint32_t   start ;
      uint32_t     end ;
} DxOArea ;
```

`start` : address of the first byte of the logical memory segment
`end` : address of the last byte of the logical memory segment

DxO describe another part of memory within a memory segment that is called payload area. It is the area containing the payload, the useful information that will have to effectively be accessed either in read or write by the two function `setDxOMem()` and `getDxOMem()` described below.

DxO ISP2013 embeds one or more DMA controllers that support the memory stride management. In other words, it can access a restricted part of memory within a bigger segment with efficient stride jumps.

An inner part of memory within a memory segment may be represented as follow.

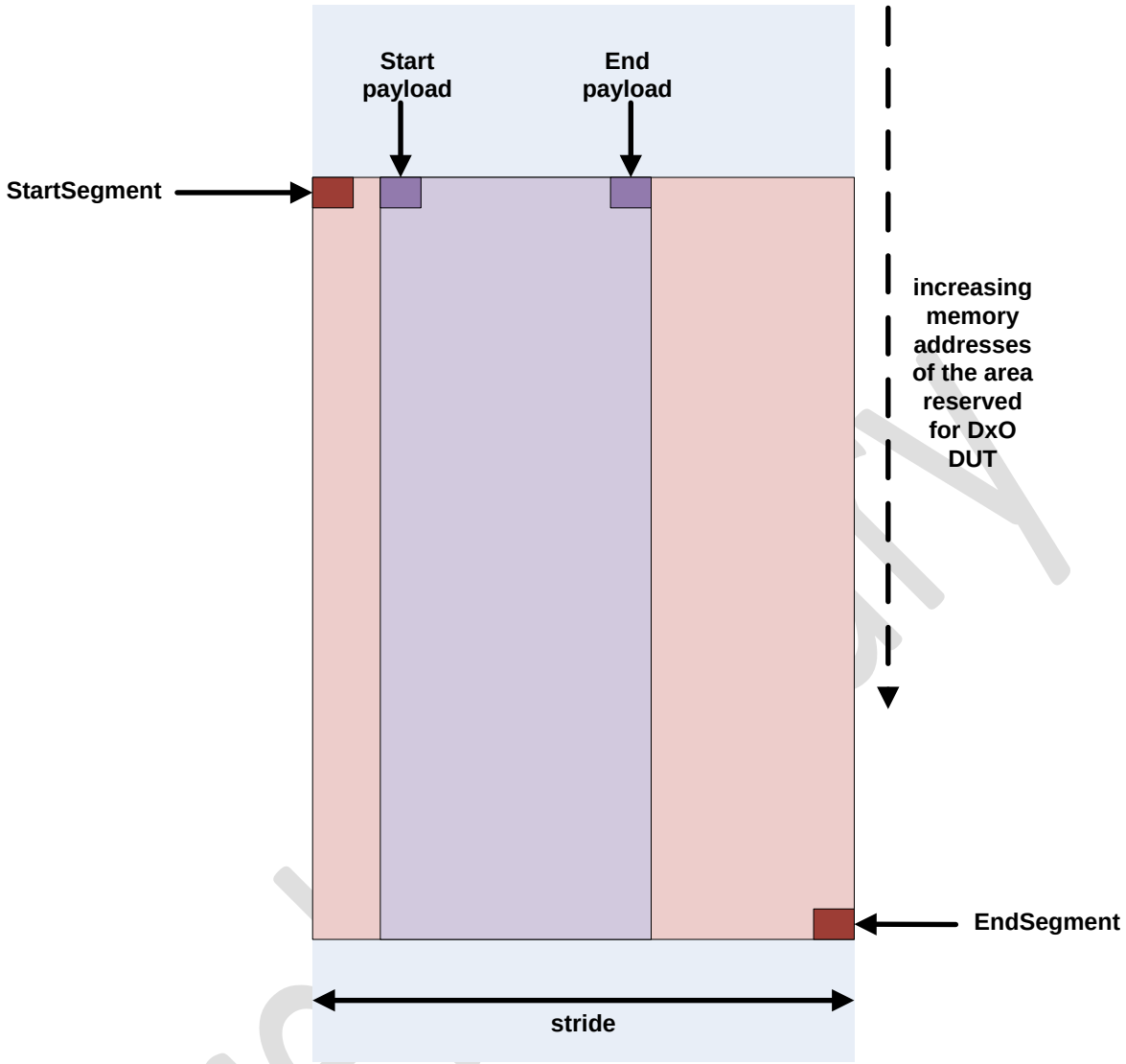

Figure 2: memory view description

StartSegment : start address 32bits of the logical memory segment
EndSegment : end address 32bits of the logical memory segment
StartPayload : offset within a stride of the first byte of data
EndPayload : offset within a stride of the last byte of data

Caveat: one requirement has to be respected to allow a proper stride management.

Requirement:

- EndSegment+1 – StartSegment must be stride modulo

This view of payload within a memory segment is defined below:

```
typedef struct {
      DxOArea     segment ;
      DxOArea     payload ;
      uint32_t    stride ;
} DxOMem ;
```

`segment` : contiguous area memory delimited by **StartSegment** and **EndSegment**
`payload` : area within a stride delimited in offset by **StartPayload** and **EndPayload**

2.6.2 allocDxOMem()

As a replay needs a specific allocated memory buffer when using the frame buffer for TNR for instance, the customer must be able to allocate a specific sized. As the address of this memory segment is not known in advance, allocDxOMem() must return a logical address that will be subsequently used through the scenario.

```
uint32_t allocDxOMem (
      void*           ctx
,     uint32_t        IOid
,     uint32_t        size
) ;
```

`ctx` : pointer to private customer's working buffer
`IOid` : input or output specifier connected to the frame buffer that memory has been allocated
`size` : number of bytes to allocate

This function returns a 32 bits logical address .

caveat: `allocDxOMem()` must return a 64 bytes aligned memory.

2.6.3 freeDxOMem()

This function is the pending of the `allocDxOMem()`.

```
void freeDxOMem (
      void*           ctx
,     uint32_t        IOid
,     uint32_t        ptr
) ;
```

`ctx` : pointer to private customer's working buffer
`IOid` : input or output specifier connected to the frame buffer that memory is freed

`ptr` : address pointing the segment to free

2.6.4 setDxOMem()

Hardware level scenario requires to have some memory initialized. The function to be defined is prototyped as followed:

```
void setDxOMem (
      void*            ctx
,     char const*      file
,     DxOMem const*    dst
) ;
```

`ctx` : pointer to private customer's working buffer
`file` : name of the file containing the buffer of bytes to write within the memory area defined by `dst`
`dst` : destination memory area description where the reference bytes must be written

2.6.5 getDxOMem()

Hardware level scenario requires to collect some memory content at a special step of a test to check this content subsequently against a provided reference. The function to be defined is prototyped as followed:

```
void getDxOMem (
      void*            ctx
,     char const*      file
,     DxOMem const*    src
) ;
```

`ctx` : pointer to private customer's working buffer
`file` : name of the file containing the buffer of bytes that have to be asserted against the memory content described by `src`
`src` : source memory area description where the reference bytes must be read for a subsequent checking against reference file `file`

2.7 Timing and synchronization events

Hardware level scenario declares two types of function to handle the timing sequence of a scenario.

2.7.1 waitDxOSync()

This function aims to trigger some synchronization waiting. This function is usually used to wait on pixel processing completion or on the reception of an interrupt. The wait action depends on the specified Id.
The scenario may require to wait on different triggers, it may be:

- the reception of the last pixel for a specified output pixel streaming
- the sending of the last pixel for a specified input pixel feeding
- the issuing of the specified interruption

Prototype:

```
void waitDxOSync(
      void*       ctx
,     uint32_t    IOid
,     int32_t     timeOut
) ;
```

`ctx` : pointer to private customer's working buffer
`IOid` : input channel, output channel or interrupt identifier
`timeOut` : maximum number of cycles for the specified trigger occurrence. If the trigger has not occurred before `timeOut` cycles after the call of the function, the customer must exit and set the test as failing. If `timeOut` is a negative value, then the waiting is infinite. If `timeOut` is set to `0`, the function returns immediately without any error.

Requirement:

- this function must:
  - block until the required synchronization is reached
  - exit in error if time out has occurred

2.7.2 waitDxOCycles()

```
void waitDxOCycles(
      void*       ctx
,     uint32_t    nbCycles
) ;
```

`ctx` : pointer to private customer's working buffer
`nbCycles` : number of cycles to wait. If `nbCycles` is set to `0`, the function returns immediately without any error.

2.8 Comments

Comments occur into the test scenario as regular C programming language comments. These are intended to add some human information not intended to be triggered or handled.

```
// this is just a comment, believe me!!!
```

3 TEST VECTOR

3.1 Contents

A hardware test level delivery is a collection of one or more test vectors that are all self-contained and stand-alone. It is typically delivered within a tbz (tared and zipped file).

Each test vector is a collection of the following items:

```
<TEST_NAME>/<file>.h
<TEST_NAME>/scenario.c
<TEST_NAME>/input/
<TEST_NAME>/input/in1.raw
<TEST_NAME>/input/ in2.raw
<TEST_NAME>/output/
<TEST_NAME>/output/out1.raw
<TEST_NAME>/output/out2.raw
```

Figure 3: test vector example content description

`<file>.h` are various C header files included by scenario.c.

3.2 Reference file description

Reference files provided are all binary files. Except for `scenario.c` and `<file>.h`, that are C programming language files, no single file has ascii content.

3.2.1 Pixstream reference input and output

In case of files specified for input and output Pixstream, these files are little endianess ordered with a packing of sampleWidth bits.
To sum up, a file containing a VGA input image of 10bit pixel has a size of 384000 bytes.
Any single reference file defined within the scenario is relative to the working directory of the test.

3.2.2 Memory content

In case of file used for memory content, these files simple binary containing byte values. No dealing with endianess must be taken into.

3.3 Test passing or failing?

The test passes if all collected buffers on the different outputs are all bit exact with the provided references in the test vector detailed above.