SMARTLOGIC

Continuous data streaming with the user mode library / SLDMA device driver

Thomas Zerrer

May 2022

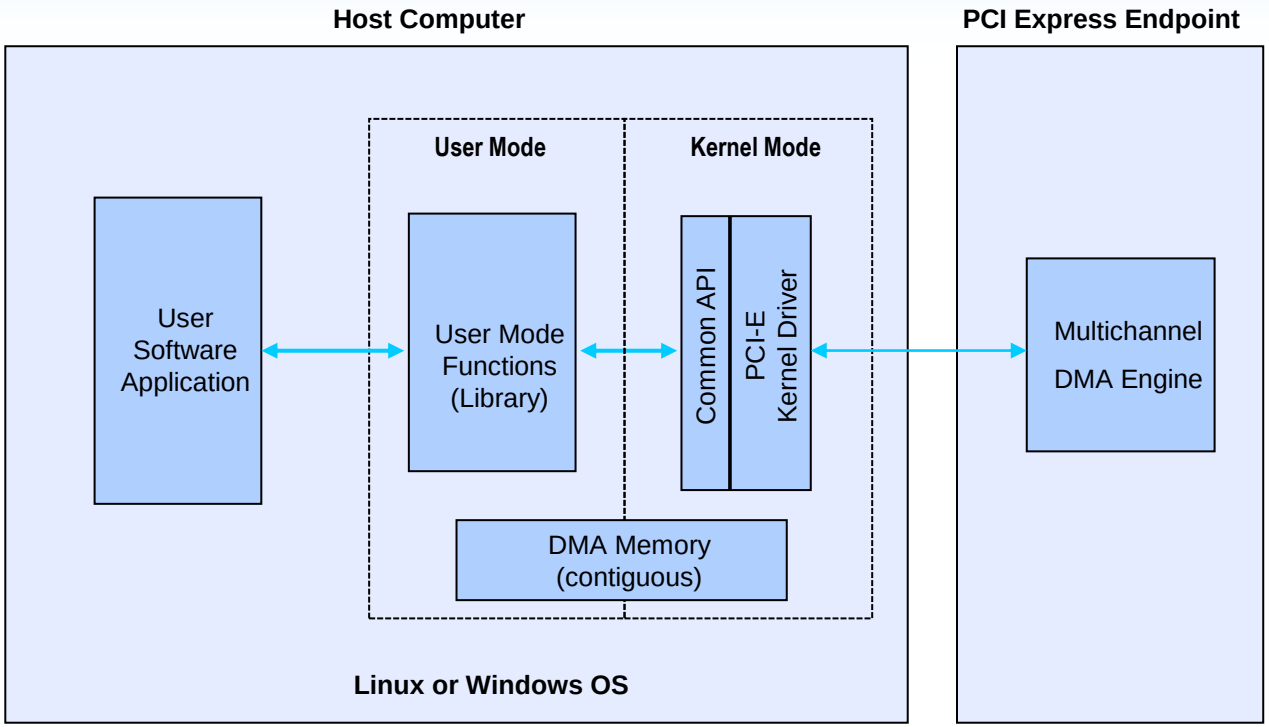
Typical applications

SMARTLOGIC

Application	DMA mode	Interrupt Source (upstream case)
Video streaming without metadata	fixed size mode (i.e. 1 video frame per DB)	Data channel
Video streaming with metadata, where metadata is transferred before the frame as header	variable size mode (i.e. data buffer contains a variable amount of data)	Data channel
Video streaming with metadata, where metadata is transferred after the frame as footer	variable size mode (i.e. DB contains a variable amount of data)	Meta channel
High speed data acquisition	Typically fixed size mode (i.e. 1 fixed amount of samples per data buffer)	Data channel
Streaming of Ethernet packets (style A)	variable size mode for up- and downstream (length of datapacket is sent as metadata with separate TDEST after data)	Meta channel
Streaming of Ethernet packets (style B)	Upstream case : fixed size mode (but the actual fill level of the data buffer is variable), length of packet is contained in a user header before the payload. There is no separate meta data channel Downstream case : variable size	Data channel

Most applications are contained in the table above. If your targeted application is not listed there, choose the use case which comes closest to your application.

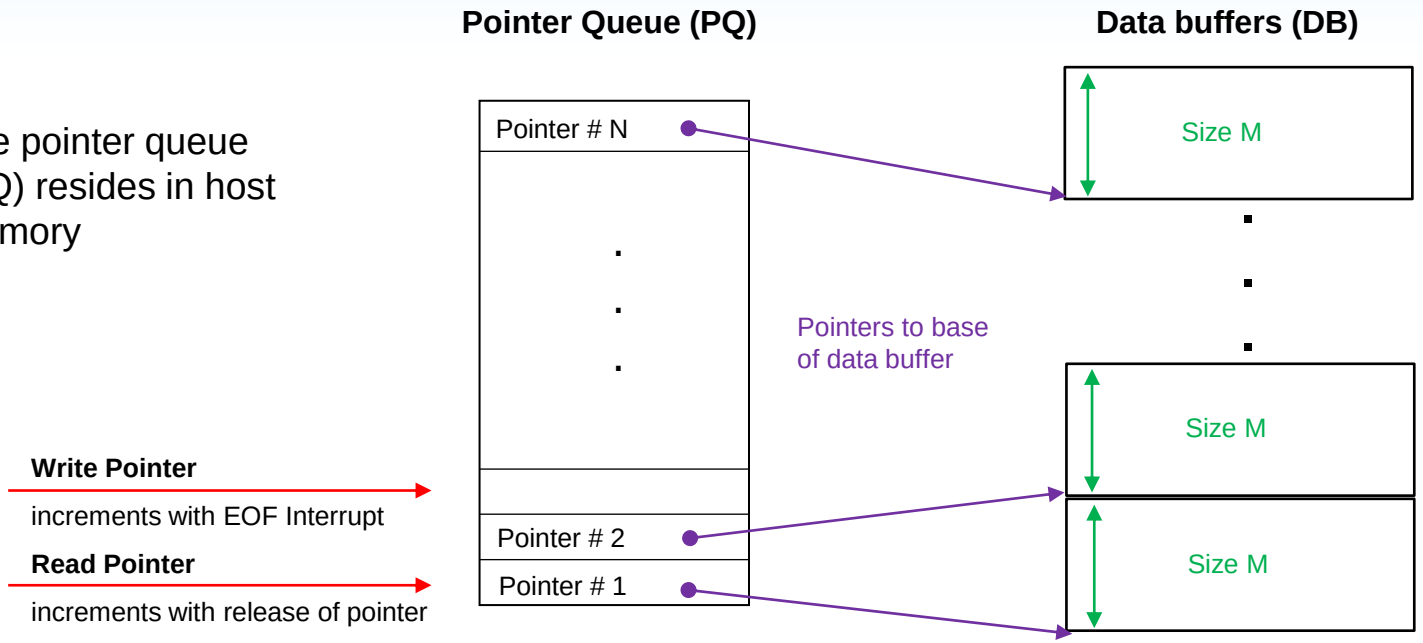
Device driver - Overview



Properties:

- Smartlogic provides a comprehensive and easy to use user mode library (C++ class) that communicates with the kernel mode driver
- DMA Memory is allocated by the kernel mode driver and mapped to user space. For Linux the user has the choice to either work with kernel mode allocated DMA Buffers or hugepages.
- For Linux the amount of DMA Memory of the driver is adjustable at compile time. For Windows it is adjustable in the registry.

Card to host transfers (Upstream / DMA Write)



Properties:

- User informs user mode driver to map N data buffers of fixed size M to a specified DMA channel
- User mode library builds the initial PQ, transmits a copy of it to the address FIFO within the FPGA and activates the EOF interrupt. The order of data buffers in the PQ can change over time, if the user returns the pointers out of order.
- User mode library maintains 2 pointers :
  write pointer : points to the data buffer that is currently written and increments when an interrupt for this channel arrives
  read_pointer : increments when the user has released a data buffer

Function overview for DMA Write (FPGA to Host)

SMARTLOGIC

User mode library Function	Purpose	Parameters	Return Parameters
upstream_channel_setup	Build initial PQ, initialize DMA Engine including interrupt	DMA Channel, Number of DMA Buffers in Queue	Success / errorcode
upstream_channel_enable / upstream_channel_disable	Enable / Disable the DMA channel	DMA channel	Success / errorcode
upstream_engine_enable / upstream_engine_disable	Enable / disable the DMA Write Channel engine	none	Success / errorcode
upstream_channel_get_next_rx_buffer	Poll PQ for next filled buffer (used for pull interface)	DMA channel, Pointer	Success / errorcode, Pointer to DB and optional to Metabuffer
upstream_channel_release_rx_buffer	Flag data buffer as processed and release pointer to the PQ	DMA channel, Pointer to DB	Success / errorcode
upstream_channel_set_callback	Register a user callback function, that is called, when a data buffer is filled (used for push interface)	DMA channel, callback function, user_data_structure	Success / errorcode
release_all_channels_and_buffers	Release and unmap all PQs and DBs for upstream and downstream	none	None
upstream_channel_suspend	Stop DMA upstream transmission for a specific channel and clear address FIFO	DMA channel	Success / errorcode
upstream_channel_resume	Resume DMA upstream transmission for a specific channel and fill address FIFO with RX buffers	DMA channel	Success / errorcode
upstream_channel_get_statistics	get statistic information (upstream)	channel	Success / errorcode

All these functions are contained in the C++ class SLDMA. The errorcodes and argument datatypes are defined in sldma.h. A demoproject is available (sldma_test.cpp)

Important for upstream applications (DMA Write) :

- The number of data buffers should be well calculated by the user, to have enough buffering capacity and to insure no data loss

1. The recording depth should be high enough, so that data can be buffered during the time the SW might have short processing problems*
2. The interrupt rate should not be too high*

- The maximum size of the data buffers can be selected individually for each DMA channel
- The user polls the dma write queue, processes the data buffer and returns the pointer. If he polls too slow, he will loose data
- It is allowed to return the data_buffers out of order (i.e. not in the order as they were fetched)
  The returned pointer is stored in the PQ again and transmitted to the FPGA
- Two interface styles are available
  a) pull style interface : The user polls, if a new DMA Buffer is available
  b) push style interface : The user is able to register a callback function, that is called when a buffer is available

* For the DMA_Demo3 Design the interrupt rate is approximately 1 kHz and the recording depth is 32 ms when 8 DMA buffers are used per queue.

Working upstream : PQ and DB setup

Application specific preparations:

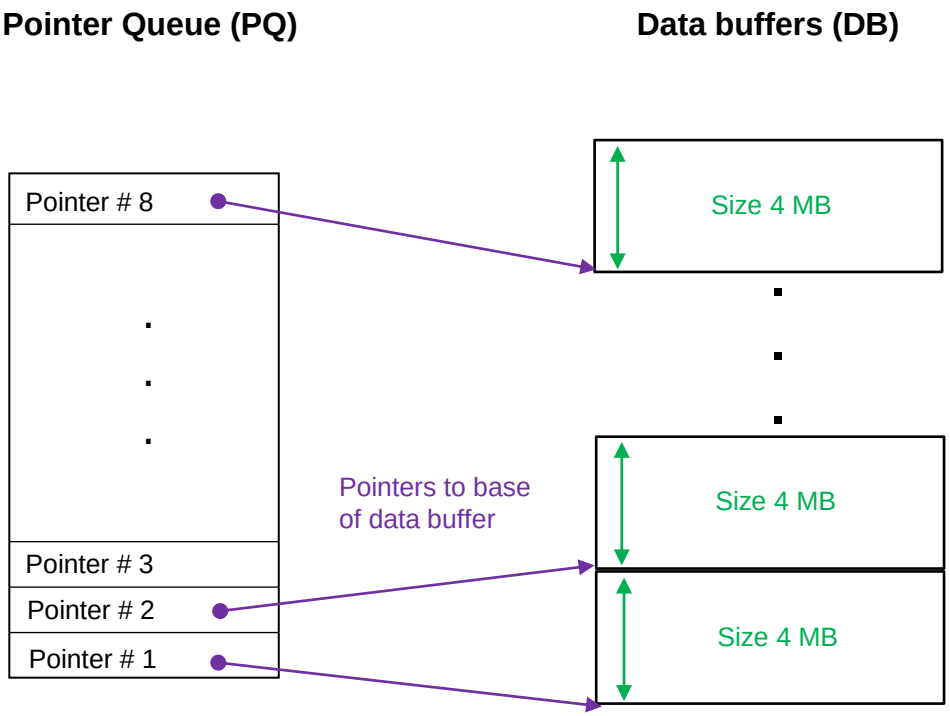
- Define the number of data buffers for each data channel

  In this example this constant is set to 8

- Initialize the internal datastructure with a default setting by simply calling upstream_channel_setup_default_mapping()

- Define the parameters (channel number, buffer size etc) by calling upstream_channel_setup_mapping_fixed_mode for fixed mode streaming or upstream_channel_setup_mapping_variable_mode for variable streaming

  In this example the buffer size is set to 4 MB



Card open (single endpoint)

In order to communicate with the FPGA, create an instance of the C++ class SLDMA:

```
SLDMA sldma1 (card_index1, number_of_channels);
```

…

Note : When working with only 1 card, the card index is 0

Open the card to get the driver handle:

```
driver_handle1 = sldma1.open ();
```

Important : The number of channels for upstream and downstream has to be identical, when instantiating the SLDMA class with two parameters. If they are not the same, you can use the 5 parameter version:

```
SLDMA sldma1 (card_index1, number_of_upstream_channels,
number_of_downstream_channels, First_DMA_Memory, Number_of_DMA_Memories);
```

DMA Memory definition:
DMA memories have nothing to to with the data buffers. For Linux a DMA memory is one of the 4 MB memories and for Windows it is one of the contiguous DMA memories defined in the registry (see here)

For guidelines on how to open the same card from 2 processes, refer to here

For guidelines on how to open the card in a multifunction or in a multi-card case, refer to here

Working upstream : PQ and DB setup with Plug & Play

The FPGA designer can store all relevant parameters for DMA setup in the Plug and Play ROM Table within the FPGA. If this ROM is configured correctly, it is sufficient to issue the following two function calls:

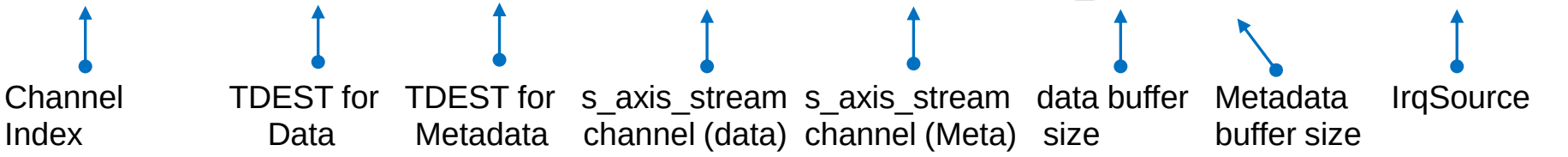
```
// initialize internal structures with Plug and Play Table from FPGA
Bool enable_pnp true;
sldma.upstream_channel_setup_default_mapping (enable_pnp);

// configure channel with PNP values

result = sldma.upstream_channel_setup (channel_index, buffer_count)
```

s_axis_stream channel Number of DBs for this channel

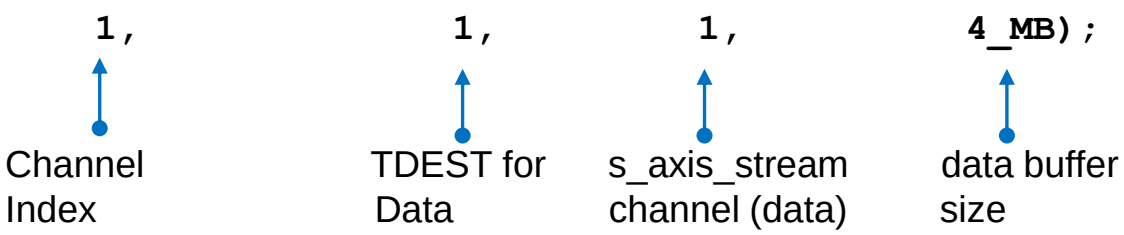
Working upstream : PQ and DB setup without Plug & Play

SMARTLOGIC

```
// initialize internal structures without PNP

bool enable_pnp false;
sldma.upstream_channel_setup_default_mapping (enable_pnp);

// Supply channel informations manually with the fixed and variable mode fn's
// in order to work with a datastream that contains data and metadata, use the
// variable mode. Here is an example :

result = sldma.upstream_channel_setup_mapping_variable_mode (
    0,          0,          8,          0,          8,          4_MB,    4, SLDMA::EofMeta);
```




```
// in order to work with a channel of that transmits fixed sized data, use the
// fixed mode. Here is an example :

result = sldma.upstream_channel_setup_mapping_fixed_mode (
    1,          1,          1,          4_MB);
```



Working upstream : PQ and DB setup

```
// In order to create the pointer Queue and to map the data buffers call the
// function upstream_channel_setup for each DMA channel (required for pnp or non
// pnp mode) :
```

```
result = sldma.upstream_channel_setup (channel_index, buffer_count)
```

Working upstream : Enabling DMA engine and channels

SMARTLOGIC

Once the PQ is setup for all channels, you enable the upstream engine by calling

```
result = sldma.upstream_engine_enable ();
```

The global enablement of the upstream is not sufficient for initiating DMA transfers. Each data channel has to be enabled with:

```
result = sldma.upstream_channel_enable (channel_index);
```

It is possible to temporarily disable a upstream channel with

```
result = sldma.upstream_channel_disable (channel_index);
```

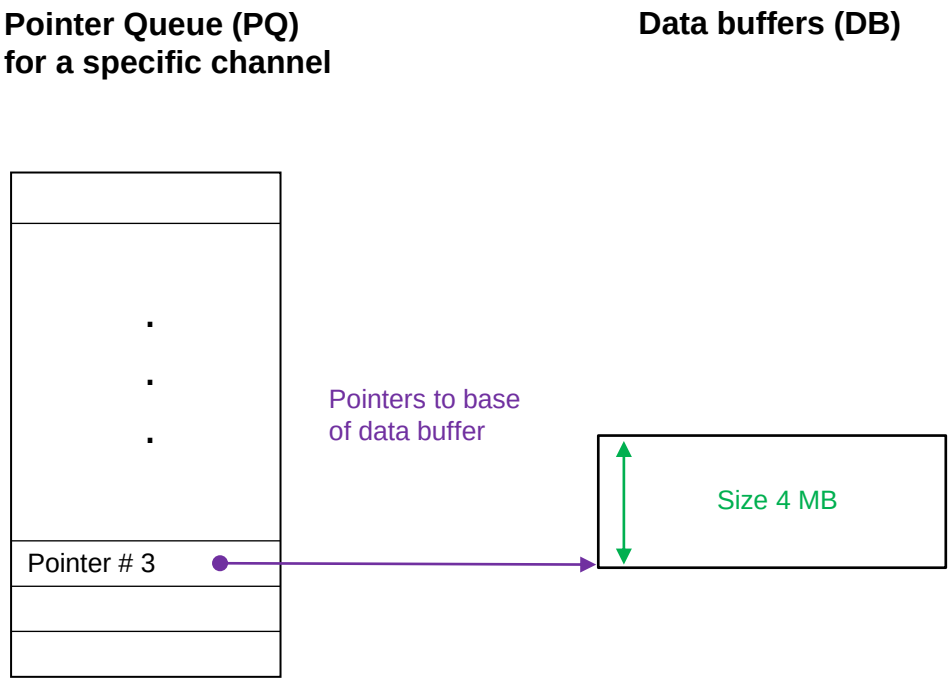
In order to avoid sideeffects, the user should only disable a channel when all RX data buffers are transmitted to Host memory.

Working upstream : Accessing RX data buffers

In order to get access to the next data buffer in the RX (upstream) queue with DMA data from the FPGA, call upstream_channel_get_next_rx_buffer.

If a data buffer is available, the pointer to the next data buffer in the RX queue for the specific channel is returned.

In case there is no DB available a nullptr is returned and the return code is „QueueEmpty“.




```
result = sldma.upstream_channel_get_next_rx_buffer (
     channel_index,     &buffer_ptr,       &meta_buffer_ptr  )
```

DMA channel pointer to DB pointer to Meta (nullptr in fixed mode)

Working upstream : Returning RX data buffers back

In order to maintain continuous data streaming, the RX data buffers have to be returned back to the UML, when their content has been accessed. If the user does not return them, DMA transmission will stop when no DBs are avaioable anymore. In order to prevent this condition, the RX DBs have to be returned as fast as possible to the UML so that they can be re-used again.

This is done by calling the function upstream_channel_release_rx_buffer:

```
result = sldma.upstream_channel_release_rx_buffer (channel_index, &buffer_ptr)
```

Working upstream : Suspend / Resume datatransmission

SMARTLOGIC

In case that the upstream DMA datatransfer should be stopped for a while and restarted at a later time, use the two functions upstream_channel_suspend and upstream_channel_resume. Please note, that only a disabled channel can be suspended and that after a resume the channel must be enabled again.

```
result = sldma.upstream_channel_disable (channel_index);
result = sldma.upstream_channel_suspend (channel_index);

...

result = sldma.upstream_channel_resume (channel_index);
result = sldma.upstream_channel_enable (channel_index);
```

Upstream_channel_suspend clears the queue entries in the upstream address fifos.
Upstream_channel_resume refills the address fifos with new queue entries in order to ensure that no old queue entries (prior to upstream_channel_suspend) are used.

Working upstream : Releasing PQ and DBs

In case the application must be terminated, it is necessary to unmap and release all pointer queues. In this case make sure to disable all channels and the DMA engine before releasing the DBs:

```
result = sldma.upstream_channel_disable (channel_index);
result = sldma.upstream_engine_disable(); /* do not issue this for MF /*
result = sldma.release_all_channels_and_buffers();
```

Please note that release_all_channels_and_buffers should only be used when the application is terminated. In case that upstream data transmission should only be temporarily suspended, use the suspend/release functions of the last slide.

*Note : If you are working in multifunction mode, please be aware that the DMA engine might also be used for other functions. Since upstream_engine_disable has a global effect across all functions, it has to be carefully considered, if the engine has to be disabled. For single function applications, it is recommended to disable the upstream DMA engine as shown above.

Working upstream : Registering callbacks for RX buffers

The user has two options in requesting the next RX data buffer (DB)

Option 1 :

User applications simply polls the PQ, if a new RX DB is available with upstream_channel_get_next_rx_buffer

Option 2 :

User application can register a callback funtion that is called in case a buffer has been successfully received. The registration of a callback function is done with:

```
Result = sldma.upstream_channel_set_callback (channel_index,
        function_to_be_called, user_data_structure);
```

Important:

The callback function itself should be as short as possible. Copy or data processing tasks should be carried out by threads which are activated by the callback function !

The callback function may not be called for each received data packet separately. Therefore the user is required to call the upstream_channel_get_next_rx_buffer function within the registered callback function until the data buffer is empty. See the sldma_test.cpp design for an example.

Registering user interrupt callbacks

User application can register a callback function that is called in case a specific user interrupt was triggered. The registration of a callback function is done with:

```
Result = sldma.user_interrupt_set_callback (irq_number,
      function_to_be_called, user_data_structure);
```

Important:

The callback function itself should be as short as possible. Copy or data processing tasks should be carried out by threads which are activated by the callback function !

By default the User Interrupt inputs 9:0 of the FPGA IP core are mapped to irq_numbers 13 downto 4.

Registering logging callback

The user mode library provides additional useful debug outputs. In order to prevent printf statements in the sldma class, the user can register a function that should be called in case a new message is available.

```
Result = sldma.logging_callback_set (function_to_be_called, &sldma);
```

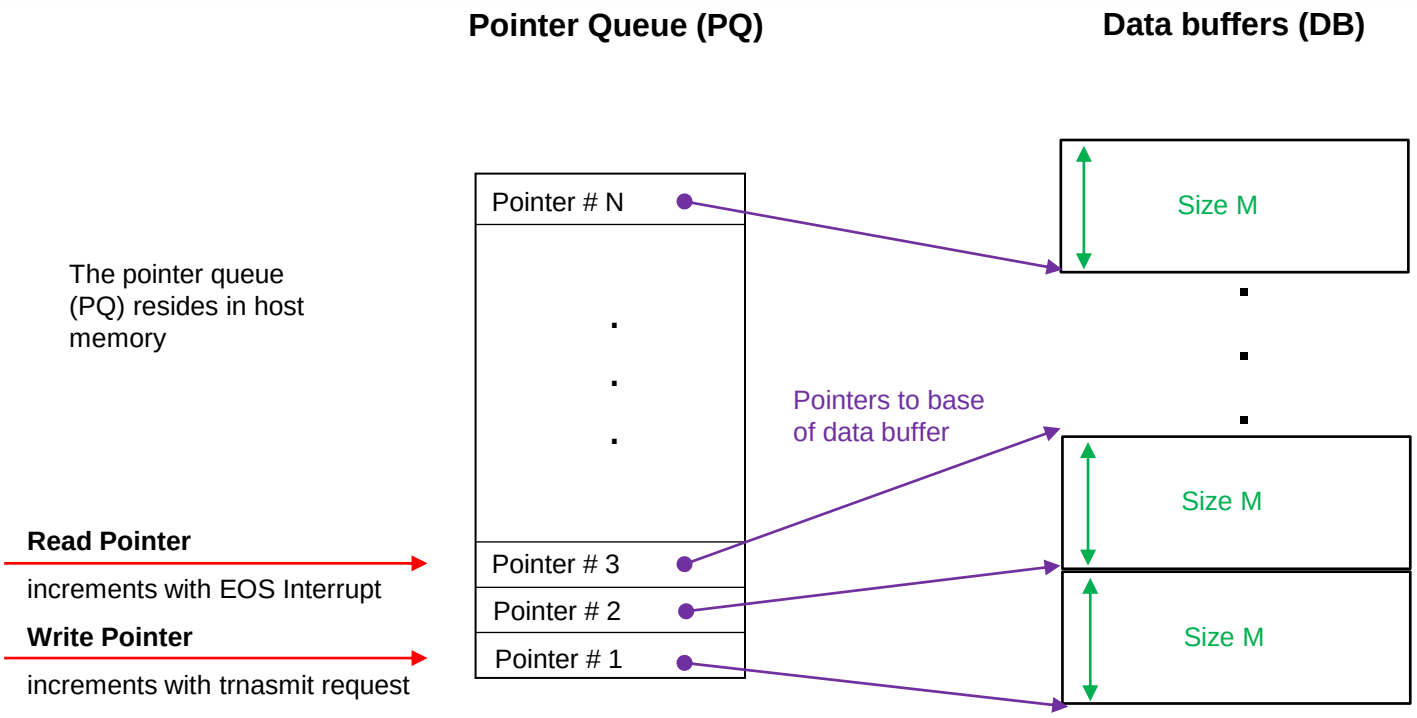
In order to clear the callback, issue

```
Result = sldma.logging_callback_clear ();
```

Note:

An example, how this callback is registered and used, can be found in the example design sldma_test.cpp

Host to card transfers (Downstream / DMA Read)



Properties:

- User informs user mode driver to map N data buffers of fixed size M to a specified DMA channel
- User mode library maintains 2 internal pointers for each PQ :
  read pointer : point to the channel currently read and increments when an interrupt for this channel arrives
  write pointer increments when the user has released a data buffer

Card close and deletion of the SLDMA class

In order to close the add-in card and to delete the sldma1 class, the user has to include the following code sequence:

```
{
SLDMA sldma1 (<card_index1>, <number_of_channels>);
driver_handle1 = sldma1.open ();
...
sldma1.close ();
}
```

Important :

The curly braces are important to ensure, that sldma1 is deleted automatically when the close is executed. The curly braces ensure that SLDMA::~SLDMA() is automatically called.

Function overview for DMA Read (Host to FPGA)

SMARTLOGIC

User mode library Function	Purpose	Parameters	Return
downstream_channel_setup	Build initial PQ, initialize pagesize and interrupt	DMA Channel, number of Queue entries	Success / errorcode
downstream_channel_enable / downstream_channel_disable	Enable / Disable the DMA channel	DMA channel	Success / errorcode
downstream_engine_enable / downstream_engine_disable	Enable the DMA Write Channel engine	none	Success / errorcode
downstream_channel_get_next_tx_buffer	request for an empty TX buffer out of the PQ	DMA Channel, pointer	Success / errorcode, Pointer
downstream_channel_release_tx_buffer	Trigger to transmit the data buffer and relase data buffer to the PQ	DMA channel, Pointer, data length	Success / errorcode
downstream_channel_set_callback	Register a user callback function, that is called, when a data buffer is transmitted (used for push interface)	DMA channel, callback function, user_data_structure)	Success / errorcode
release_all_channels_and_buffers	Release and unmap all PQs and DBs for upstream and downstream	none	None
downstream_channel_set_flex_core_block_length_ multiplier (only needed in flex core, not HCC core)	Set block length multiplier for individual channels	channel, multiplier	Success / errorcode
downstream_channel_get_statistics	Get statistic information (downstream)	channel	Success / errorcode

All these functions are included in the C++ class SLDMA

The errorcodes and argument datatypes are defined in sldma.h

A demoproject is available (sldma_test.cpp)

Important for downstream applications (DMA Read) :

- The size of the data buffers should introduce a maximum interrupt rate of 1 ms per channel
- The maximum size of the data buffers can be selected individually for each DMA channel. It is possible to transmit either the full size of the data buffer or only a user defined portion of the data buffer per transfer.
- The user can request data buffers out of the PQ as long as free pointers are available
- The user triggers a downstream transfer with a filled data buffer that has been previously requested. This can be in a different order than the buffers were requested.

Working downstream : PQ and DB setup

Application specific preparations:

- Define the number of data buffers for each data channel

  In this example this constant is set to 8

- Initialize the internal datastructure with a default setting by simply calling *downstream_channel_setup_default_mapping()*

- Define the parameters (channel number, buffer size etc) by calling

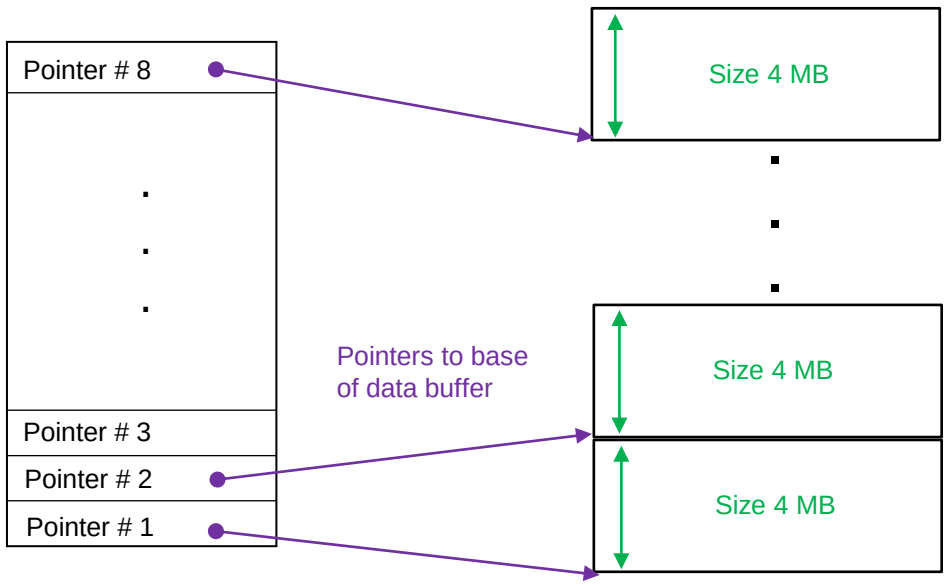
downstream_channel_setup_mapping_fixed_mode

for fixed mode streaming or

downstream_channel_setup_mapping_variable_mode

for variable streaming

In this example the buffer size is set to 4 MB



Working downstream : PQ and DB setup with Plug & Play

The FPGA designer can store all relevant parameters for DMA setup in the Plug and Play ROM Table within the FPGA. If this ROM is configured correctly, it is sufficient to issue the following two function calls:

```
// initialize internal structures from the Plug and Play ROM table
bool enable_pnp true;
sldma.downstream_channel_setup_default_mapping (enable_pnp);
```

```
// In order to create the pointer Queue and to map the data buffers call the
// function downstream_channel_setup for each DMA channel :
```

```
result = sldma.downstream_channel_setup (channel_index, buffer_number)
```

Working downstream : PQ and DB setup without Plug & Play

SMARTLOGIC

```
// initialize internal structures with generic default values

bool enable_pnp false;
sldma.downstream_channel_setup_default_mapping (false);


// in order to work with a datastream that transmits a data buffer only partially,
// use the variable mode. Here is an example :

result = sldma.downstream_channel_setup_mapping_variable_mode (
    1,              1,              4_MB);
```

Channel index — m_axis_stream channel (data) — data buffer size

```
// in order to work with a channel of that transmits fixed sized data, use the
// fixed mode for the specific channel. Here is an example :

result = sldma.downstream_channel_setup_mapping_fixed_mode (
    1,              1,              4_MB);
```

Channel index — m_axis_stream channel (data) — data buffer size

Working upstream : PQ and DB setup

```
// In order to create the pointer Queue and to map the data buffers call the
// function downstream_channel_setup for each DMA channel(required for pnp
// or non-pnp mode) :
```

```
result = sldma.downstream_channel_setup (channel_index, buffer_number)
```

m_axis_stream channel Number of DBs for this channel

Working downstream : Enabling DMA engine and channels

SMARTLOGIC

Once the PQ is setup for all channels, you enable the upstream engine by calling

```
result = sldma.downstream_engine_enable ();
```

The global enablement of the downstream is not sufficient for initiating DMA transfers. Each data channel has to be enabled with:

```
result = sldma.downstream_channel_enable (channel_index);
```

It is possible to temporarily disable a downstream channel with

```
result = sldma.downstream_channel_disable (channel_index);
```

In order to avoid sideeffects, the user should only disable a channel when all TX data buffers are transmitted to the FPGA.

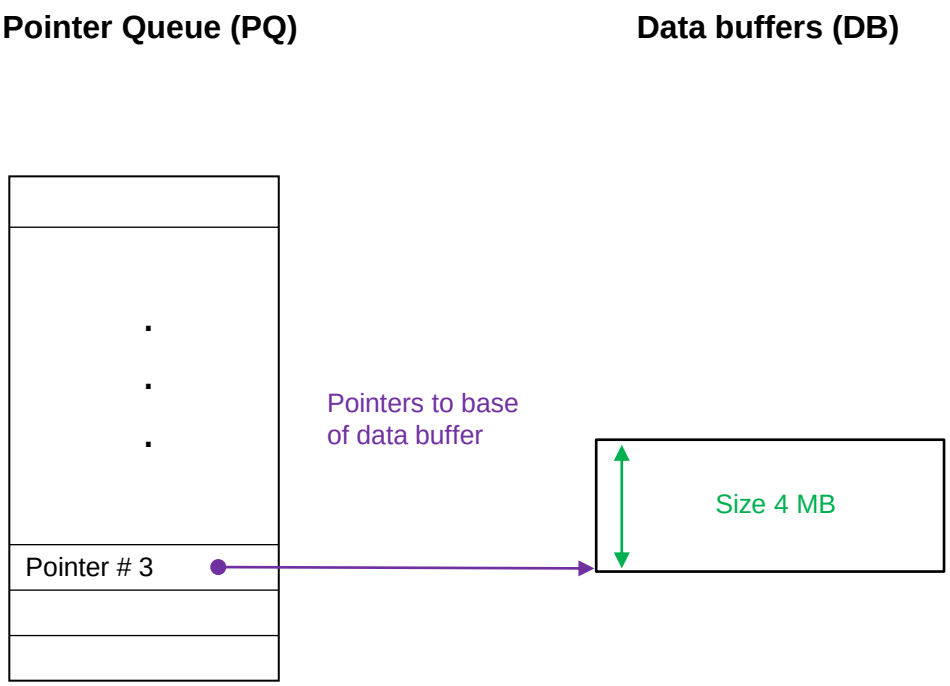
Working downstream : Requesting TX data buffers

In order to get access to the next data buffer in the downstream queue, call downstream_channel_get_next_tx_buffer.

If a data buffer is available, the pointer to the next data buffer in the TX queue for the specific channel is returned.

Now the user can fill this DB with application specific data to be transferred to the FPGA

In case there is no free DB anymore a nullptr is returned and the return code is „QueueEmpty“




```
result = sldma.downstream_channel_get_next_tx_buffer (channel_index, &buffer_ptr)
```

m_axis_stream channel pointer to requested DB

Working downstream : Initiate TX data buffer transfers

Once the requested DB is ready for transmission, issue

```
result = sldma.downstream_channel_release_tx_buffer (channel_index,
                                buffer_ptr, size);
```

With this function call the DB will be transferred to the FPGA. After transmission the User Mode Library will automatically take care that the DB is marked as free in the PQ and ready to be requested again.

The size argument is only needed for variable size data transmission. It can be omitted for fixed size transmission.

Technical background information : The IP core will issue an interrupt when the data buffer is successfully transferred to the FPGA. This interrupt is automatically handled by the User Mode Library.

Working downstream : Releasing PQ and DBs

In case the application must be terminated, it is necessary to unmap and release all pointer queues. In this case make sure to disable all channels and the DMA engine before releasing the DBs:

```
result = sldma.downstream_channel_disable (channel_index);
result = sldma.downstream_engine_disable (); /* do not issue this for MF */
result = sldma.release_all_channels_and_buffer ();
```

*Note : If you are working in multifunction mode, please be aware that the DMA engine might also be used for other functions. Since downstream_engine_disable has a global effect across all functions, it has to be carefully considered, if the engine has to be disabled. For single function applications, it is recommended to disable the downstream DMA engine as shown above.

Working downstream : Registering callbacks

The user has two options in requesting the next TX DB

Option 1 :

User applications simply polls the PQ, if a new TX DB is available with downstream_channel_get_next_tx_buffer

Option 2 :

User application can register a callback funtion that is called in case a buffer has been successfully transmitted. The registration of a callback function is done with:

```
Result = sldma.downstream_channel_set_callback
(channel_index,function_to_be_called, user_data_structure);
```

Important:

The callback function itself should be as short as possible. Copy or data processing tasks should be carried out by threads which are activated by the callback function !

The callback function may not be called for each transmitted data packet separately. Therefore the user can try to get more than one TX buffer when the callback function is called. See the sldma_test.cpp design for an example.

FlexCore : Optimizing downstream performance

The total read request is composed of several subrequests with a defined blocklength

The user can optimize the throughput of a m_axis interface by defining the blocklength of a subrequest.

```
Result = sldma.downstream_channel_set_flex_core_block_length_multiplier
                (channel, multiplier);
```

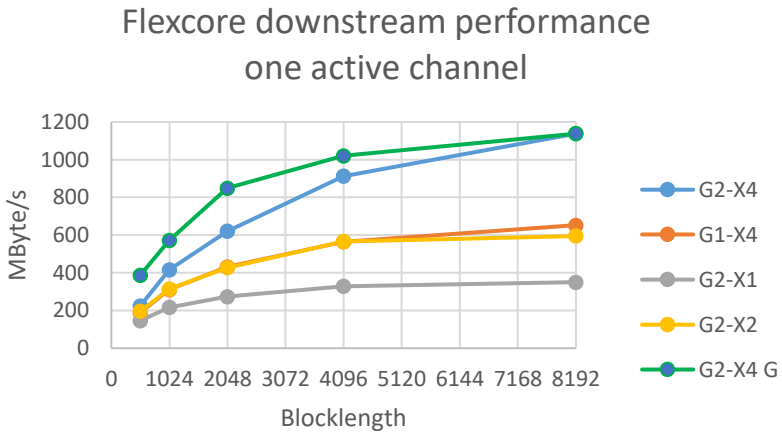
Multiplier defines the blocklength of a subrequest
blocklength = MRRS * multiplier

The multiplier must be in the following range
non ext tag mode : 1 … 2
ext tag mode : 1 … 16

This function is useful when only a few m_axis interfaces are in use that require a high throughput

Extended Tag usage should be turned on for best throughput

Due to its optimized architecture the HCC IP core does not require this function.



Collecting statistic information

Two functions provide statistic information:

- upstream_channel_get_statistics
- downstream_channel_get_statistics

Provided information:

- packet_counter : Number of packets transferred / received
- queue_empty_counter : Counter how often the address FIFO reached empty
- merged_irqs_counter : Counts merged interrupts
- used_buffers : provides infromation how many data buffers were actually used

Working with multiple cards (endpoints)

It is possible to work with the User Mode Library and more than 1 add-in card (PCIe endpoint) or when using multifunction devices.

Create an instance of the C++ class SLDMA with the 5 parameter version

```
SLDMA sldma1 (card_index1, number_of_upstream_channels1,
number_of_downstream_channels1, first_DMA_Buffer1, number_of_DMA_buffers1);

SLDMA sldma2 (card_index2, number_of_upstream_channels2 ,
number_of_downstream_channels2, first_DMA_Buffer2, number_of_DMA_buffers2);

…
```

Note : Card_index = 0 opens the first visible card, 1 the next, etc. In order to ensure that each card operates on its own dedicated/exclusive DMA Memory, it is important to add two more parameters to inform the UML which DMA buffers are mapped to each card. For example set

first_DMA_Buffer2 = first_DMA_Buffer1 + number_of_DMA_buffers1

Open each card to get the driver handle:

```
driver_handle1 = sldma1.open ();

driver_handle2 = sldma2.open ();
```

Accessing the driver from two processes

SMARTLOGIC

It is possible to open the driver two times, so that 2 separate applications (processes) can access the driver for the same card or pcie function.

Create an instance of the C++ class SLDMA in each application with the 5 parameter version

Application 1 source code :

```
SLDMA sldma1 (card_index, number_of_upstream_channels1,
number_of_downstream_channels1, first_DMA_Buffer1, number_of_DMA_buffers1);
```

Application 2 source code:

```
SLDMA sldma2 (card_index, number_of_upstream_channels2 ,
number_of_downstream_channels2, first_DMA_Buffer2, number_of_DMA_buffers2);
```

…

Note : Card_index must be set to the same value for both applications. In order to ensure that each application operates on its own dedicated/exclusive DMA Memory, it is important to add two more parameters to inform the UML which DMA buffers are mapped to each application. For example set

first_DMA_Buffer2 = first_DMA_Buffer1 + number_of_DMA_buffers1

There is a two process example design available which can be run with the DMA_Demo3 example bitstream. The two examples are located in the folders sldma/test_rx and sldma/test_tx. Simply start both applications and see how 100 packets are transmitted. The example is very short and easy to understand. Further details are given in the source code.

Linux : Hugepage Support (1)

SMARTLOGIC

Hugepage support for Linux:

- DMA Buffers allocated by the Linux driver in kernel mode are limited to a size of 4 MB. Since only a maximum of 256 4-MB buffers can be allocated, the total amount of DMA memory is limited to 1024 Mbytes. In order to work with more contiguous DMA memory, the user has the option to work with hugepages.
- Hugepages are contiguous memory blocks that can be reserved at boot time. Once such a huge page pool is reserved, the user can freely request or release hugepages at any time.
- The size of a Hugepage depends on the CPU system, but can be up to 1 GB
- In order to detect the supported hugepagesizes, type

```
hugeadm --page-sizes-all
```

- In order to see the status of the hugepage pool type : hugeadm --pool-list
-
- When working with GB Hugepages, it is possible to work with data buffers greater than 4 MB
- Currently only a maximum of 4 Hugepages are supported

Linux : Hugepage Support (2)

SMARTLOGIC

How to setup and to work with hugepages instead of 4 MB DMA memoryblocks:

- Prepare your Linuxsystem with the following boot kernel parameters for 1 1GByte Page:

default_hugepagesz=1G hugepagesz=1G hugepages=1

- In case you are working with two applications that open the driver in parallel, you have to reserve two hugepages.
- Add the following line to your /etc/sysctl.conf

```
vm.hugetlb_shm_group = 1000
```

- reboot your Linux system
- Set the define **`OPT_HUGEPAGE_SUPPORT`** in your makefile or pro file
- In your application call sldma.huge_page_pool_alloc (SLDMA::HugePageType_1G) for each hugepage you want to use in order to instruct the UML that you want to work with hugepages. See sldma_test.cpp for an example
- compile your application with the user mode library (sldma class)
- If you are working only with hugepages and multiple processes, each process will automatically operate with ist own dedicated hugepage.

Now the user mode library will automatically use hugepages for all DMA channels. The user mode library tries to map as many as possible channels and their data buffers to one hugepage.

Linux : Optimizing DMA performance

For latency sensitive applications, follow these guidelines on linux:

- open /etc/security/limits.conf (as root) and add the following entries:

```
<account>    -  rtprio    99
<account>    -  priority  99
<account>    -  nice     -20
<account>    -  memlock   8336191488
```

- The account name is reported when you type „whoami“
- reboot your Linux system

Technical background:

Whenever an interrupt arrives on the CPU, a user mode thread has to be started that calls the user defined callbacks. When this thread has only the default priority, it can be delayed by any OS operation with higher priority. It has been observed that this delay can be in the range of 10-40 ms (machine configuration dependent), which introduces additional latency and demands significant DMA buffering capacity to avoid data loss.

In order to make use of higher priorities and to prevent the machine code of this thread to be swapped to hard disc, several OS calls were added to the UML. In order to prevent that sldma has to be run under root the entries above have to be present in limits.conf

As a result of these optimizations latency could be reduced by a factor of 8 on a Smartlogic lab PC

Important:
If the callback function starts other threads, the user is responsible to give these threads the right priority.

Windows 10 : DMA Memory

SMARTLOGIC

For Windows the number and size of DMA memories can be selected with dedicated Registry entries.

They are located in :

HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\sldemo\Settings

Registry Key Name	Function	Default Setting	Comment
DmaBuffer00SizeMb	Memory size of DMA Buffer 0 in Megabytes	4	Max value is 256 MB for Windows
DmaBuffer01SizeMb	Memory size of DMA Buffer 1 in Megabytes	4	
DmaBuffer02SizeMb	Memory size of DMA Buffer 2 in Megabytes	4	
…			
DmaBuffer14SizeMb	Memory size of DMA Buffer 14 in Megabytes	4	
DmaBuffer15SizeMb	Memory size of DMA Buffer 15 in Megabytes	4	
DmaBufferCount	Number of needed DMA buffers : Range 1 to 16	4	
DMABufferLowerAddressMask	Each bit specifies, if the associated DMA buffer has to be allocated below (‚1‘) or above the 4 GB boundary (‚0‘)	3	Set to 0

Example Design:

- An example program (sldma_test.cpp) is available that shows how the user mode functions are used
- With this example design it is very easy to integrate the user mode library into the user‘s specific software application
- sldma_test –h shows the available command line switches and examples
- There is also an FPGA Reference Design available (DMA_Demo3) that works together with sldma_test.cpp
- Two further light weight example designs are sldma_test_rx and sldma_test_tx located in sldma/test_rx and sldma/test_tx. These two examples show how two separate applications can access the driver (currently only available on Linux)

For the FPGA Designer : Important IP Core Settings (I)

SMARTLOGIC

Upstream fixed size transfers :

If you are working with fixed sized data blocks upstream and you don't want to take care of the s<nn>_axis_tuser signals, you can set

DMA_Write_s_axis_tuser_not_driven_c : boolean := true;

This is especially useful when you are working with Vivado Blockdesigner, where you might have no direct access to these ports.

For data transfers of variable size however, you have to set this constant to false. This parameter can be found in DMA_pkg.vhd.

Downstream variable size transfers :

If you want to work with variable data sized blocks downstream, you have to activate the length FIFO in order to inform the IP core on the actual transfer size of each dma buffer that is sent downstream.

use_Image_Format_fifo_c : boolean := true;

Image_Format_fifo_depth_c is the log2 of the actual length FIFO depth. Set this parameter to a value that is greater or equal to twice the number of data buffers of a channel. If the calculated value is below 4, set it to 4.

e.g. : If the downstream PQ consists of 8 entries, the length fifo needs 16 entries. Therefore the use_Image_Format_fifo_c constant is log2(16) = 4

Image_Format_fifo_BRAM_c selects, if you want to build the length FIFO with blockram (true) or distributed RAM (false). Important is, that the length FIFO is actually 1 FIFO for all m_axis interfaces. As a rule of thumb you should set this parameter to true, if you have more than 4 m_axis interfaces in use.

Configuring MSI-X Interrupts:

It is highly recommended to work with MSI-X Interrupts wherever possible. In cases where it is not possible (e.g. FLEX core users) to use MSI-X interrupt signaling, use the following guidelines to work with MSI interrupts:

Configuring MSI Interrupts:

It is important to configure the correct size of MSI messages.

Use case	Minimum MSI message number required	Alternate settings
Upstream only	**8**	**32 or 1***
Downstream only	**16**	**32 or 1***
Upstream and downstream	**16**	**32 or 1***

The number of MSI messages can be selected with the parameter PCIE_MSI_CAP_MULTIMSGCAP_C in pcie_ep_config_pkg.vhd (which is actually the log2 of the requested number of MSI messages) and the parameter PCIe_MSI_Vector_Number in generate_pcie_xci.tcl when you are working with FPGA devices where a core has to be generated.

Always make sure that both parameters reflect the same setting.

Since the number of MSI messages is negotiated with the root complex of the CPU at boot time, it might be possible that you get less MSI messages than requested. Therefore try to set the value in the recommended column.

* Although the User mode library can operate with only 1 MSI vector, it is recommended to set the number of MSI messages according to the «minimum MSI message number required» column.

For the FPGA Designer : Important IP Core Settings (III)

SMARTLOGIC

Optimizing downstream throughput performance:

In order to maximize downstream performance, make sure to configure the following settings:

Activate Extended Tag usage:

If you are working with extended Tags, make sure, that the data fifo capacity is at least 8 kB (DMA_Read_Fifo_params_c in dma_pkg.vhd).

For simulation:

Set PCIe_DEV_CAP_EXT_TAG_SUPPORTED_C (pcie_ep_config_pkg.vhd) to „TRUE“

For Xilinx FPGAs:

configure this in generate_pcie_xci.tcl (PCIe_Extended_Tag_Ena „true“). This ensures, that the IP core can work with a maximum of outstanding read requests.

For Intel FPGAs:

Configure this in the GUI of the HIP „Number of tags supported per function“ by choosing the maximum offered value

Completion Sorting (only needed, if you are working downstream)

For some systems (especially AMD CPUs and some Intel CPUs) it might be necessary to turn on the completion sorter.

Details on this feature can be found in the Application Note „AN_Completion_Sorting“ from Smartlogic. If you don‘t follow the guidelines there, you might receive the data in the FPGA out of expected order.

About gstreamer

Gstreamer allows to display live video streams and is an open source video framework available for Linux and Windows. The gstreamer framework can be easily integrated in new user applications without the need to develop video functions.

Support for gstreamer

The user mode library is compatible for use with gstreamer 1.0
It has been successfully tested with Gstreamer Version 1.0.1.20.3 (Linux and Windows/mingw)

Example design:

- An example design (VHDL Source code and C++ Source Code) is available that shows how to interface the Smartlogic IP cores with GStreamer
- With this example design it is very easy to understand the basic design principles and to build new custom applications.

Installing Gstreamer on Linux

SMARTLOGIC

Install the following packages

sudo apt-get install libgstreamer1.0-dev

sudo apt-get install gstreamer1.0-plugins-bad

sudo apt-get install gstreamer1.0-plugins-ugly

sudo apt-get install gstreamer1.0-libav

sudo apt-get install gstreamer1.0-qt5

sudo apt-get install libgstreamer-plugins-base1.0-dev

sudo apt-get install libgstreamer-plugins-bad1.0-dev

sudo apt-get install gstreamer1.0-plugins-base

sudo apt-get install gstreamer1.0-plugins-good

sudo apt-get install gstreamer1.0-tools

sudo apt-get install gstreamer1.0-x

sudo apt-get install gstreamer1.0-alsa

sudo apt-get install gstreamer1.0-gl

sudo apt-get install gstreamer1.0-gtk3

sudo apt-get install gstreamer1.0-pulseaudio

Installing Gstreamer on Windows

SMARTLOGIC

Gstreamer can be downloaded for Windows from : https://gstreamer.freedesktop.org/

The user has the choice to install MSVC or MINGW based versions of the Gstreamer framework

In any case make sure to do the following:

Install the runtime first and select the “custom” install

Select all modules manually and set to “install entire feature on local disc”

After the runtime install the development part of the framework in the same way (custom install)

After successful installation check the following:

The environment variable GSTREAMER_1_0_ROOT_X86_64 must point to the selected folder from the custom install

Add %GSTREAMER_1_0_ROOT_X86_64%\bin to your path variable

You can check your gstreamer installation by typing in a windows console

```
gst-launch-1.0 -v videotestsrc ! autovideosink
or
gst-inspect-1.0 videotestsrc
```

If this succeeds, your Gstreamer installation was successful

Known limitations

Currently the following limitations (as of June 2022) are known:

- Linux : Trying to allocate data buffers greater than 4 MB without using hugepages will result in a segmentation fault. Solution : Use hugepages instead
- Windows SLDMA : Plug and Play does not work correctly
- Windows : The driver can only be opened for one process