QDMA Subsystem for PCI Express v2.0

Product Guide

Vivado Design Suite

PG302 (v2.0) September 4, 2018

Table of Contents

IP Facts

The Xilinx® QDMA Subsystem for PCI Express® (PCIe®) implements a high performance DMA for use with the PCI Express® 3.x Integrated Block with the concept of multiple queues that is different from the DMA/Bridge Subsystem for PCI Express which uses multiple C2H and H2C channels.

Features

- The PCIe Integrated Block is supported in UltraScale+™ devices, including Virtex® UltraScale+™ devices with high bandwidth memory (HBM).
- Supports 64, 128, 256, and 512-bit data path.
- Supports x1, x2, x4, x8, or x16 link widths.
- Supports Gen1, Gen2, and Gen3 link speeds. Gen4 for PCI4C block.
- Support for both the AXI4 Memory Mapped and AXI4-Stream interfaces per queue.
- 2048 queue sets
    - 2048 H2C Descriptor rings.
    - 2048 C2H Descriptor rings.
    - 2048 C2H.
- Supports Polling Mode (Status Descriptor Write Back) and Interrupt Mode.
- Interrupts
    - 2048 MSI-X vectors.
    - Up to 8 MSI-X per function.
    - Interrupt aggregation.
- C2H Stream interrupt moderation.
- C2H Stream Completion queue entry coalescence.
- Descriptor and DMA customization through user logic

- Allows custom Descriptor format.
- Traffic Management.

- Supports SR-IOV with up to 4 Physical Functions (PF) and 252 Virtual Functions (VF)
    - Thin hypervisor model.
    - QID virtualization.
    - Allows only privileged/Physical functions to program contexts and registers.
    - Function Level Reset (FLR) support.
    - Mailbox.
- Rich programmability on per queue basis, e.g., AXI-MM versus AXI-ST.

The current version of this IP is marked as pre-production.

IP Facts

LogiCORE IP Facts Table	
Subsystem Specifics	
Supported Device Family[1]	UltraScale+™
Supported User Interfaces	AXI4 Memory Map, AXI4-Stream, AXI4-Lite
Resources	Not Provided
Subsystem	
Design Files	Encrypted System Verilog
Example Design	Verilog
Test Bench	Verilog
Constraints File	Xilinx® Constraints File (XDC)
Simulation Model	Verilog
Supported S/W Driver[2]	Linux, DPDK Drivers **Note:** Windows Drivers supported in a future release.
Tested Design Flows[3]	
Design Entry	Vivado Design Suite
Simulation	For supported simulators, see the Xilinx Design Tools: Release Notes Guide.
Synthesis	Vivado Synthesis

LogiCORE IP Facts Table
Support
Provided by Xilinx® at the Xilinx Support web page

Notes:

1. For a complete list of supported devices, see the Vivado IP catalog.
2. Linux driver details can be found in AR 70928. For DPDK driver details, contact Xilinx Support.
3. For the supported versions of the tools, see the Xilinx Design Tools: Release Notes Guide.

Overview

The Queue Direct Memory Access (QDMA) subsystem is a PCI Express (PCIe) based DMA engine that is optimized for for both high bandwidth and high packet count data transfers. The QDMA is composed of the UltraScale+ Integrated Block for PCI Express IP, and an extensive DMA and bridge infrastructure that enables the ultimate in performance and flexibility.

The QDMA Subsystem for PCIe offers a wide range of setup and use options, many selectable on a per-queue basis, such as memory mapped DMA or stream DMA, interrupt mode or polling, etc. The subsystem provides many options for customizing the Descriptor and DMA through user logic to provide complex traffic management capabilities.

The primary mechanism to transfer data using the QDMA is for the QDMA engine to operate on instructions (descriptors) provided by the host operating system. Using the descriptors, the QDMA can move data in both the Host to Card (H2C) direction, or the Card to Host (C2H) direction. You can select on a per queue basis whether DMA traffic goes to an AXI4 memory map interface or to an AXI4-Stream interface. In addition, the QDMA has the option to implement both an AXI4 Master port and an AXI4 Slave port, allowing PCIe traffic to bypass the DMA engine completely. A complete list of all available interfaces can be found in Port Descriptions.

The main difference between QDMA and other DMA offerings is the concept of Queues. The idea of Queues is derived from the “queue set” concepts of Remote Direct Memory Access (RDMA) from high performance computing (HPC) interconnects. These Queues can be individually configured by interface type, and they function in many different modes. Based on how the DMA descriptors are loaded for a single Queue, each Queue provides a very low overhead option for setup and continuous update functionality. By assigning Queues as resources to multiple PCIe Physical and Virtual Functions, a single QDMA core and PCI Express interface can be used across a wide variety of multi-function and virtualized application spaces.

The QDMA Subsystem for PCIe can be used and exercised with a Xilinx® provided QDMA reference driver, and then built out to meet a variety of application spaces.

Glossary

The following table contains frequently used acronyms in this document.

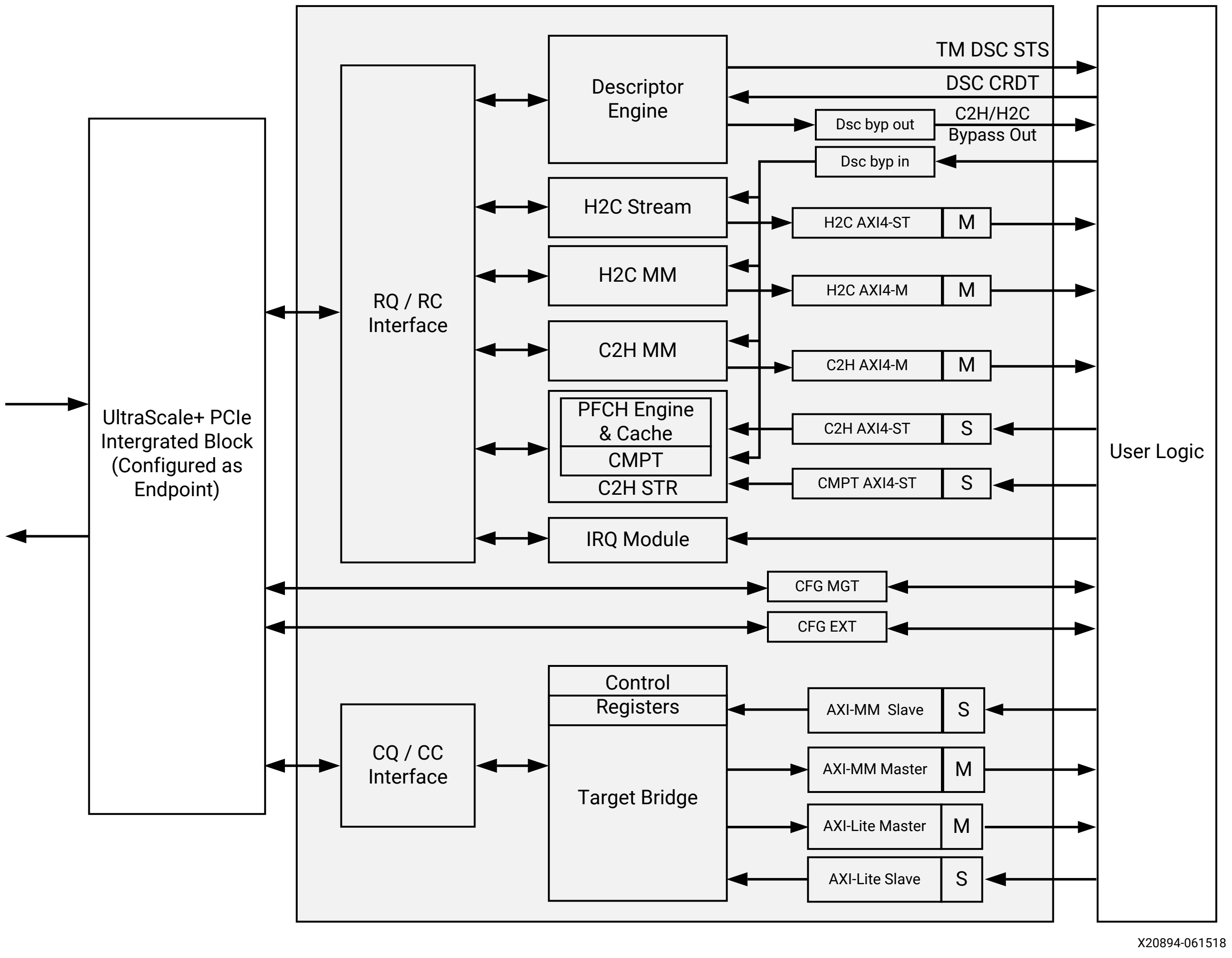
Table 1: **Glossary of Terms**

Acronym	Full Name
APP	Application
AXI-ST	AXI4-Stream
AXI-MM	AXI4-Memory Mapped
BADDR	Base Address
C2H	Card to Host
CC	Completer Completion
CIDX	Consumer index pointer
CMPT	C2H AXI4-Stream Completion
CQ	Completer Request
CRDT	Credit
CSR	Control/Status register
CTXT	Context
DSC	Descriptor
FLR	Function Level Reset
H2C	Host to Card
HW	Hardware
OS	Operating System
PF	Physical function
PFCH	Prefetch Block
PIDX	Producer index pointer
QDMA	Queue Direct Memory Access
Qx	Queue, Q0 is 0th Queue
QID	Queue Identification
RC	Requester Completion
RDMA	Remote Direct Memory Access
RQ	Requester Request
RXQ	Receive Queue
SRIOV	Single root input/output virtualization
ST	Streaming
SW	Software
TM	Traffic Manager
TXQ	Transfer Queue
VM	Virtual Machine
VF	Virtual function
VFG	Virtual Function Group

QDMA Architecture

The following figure shows the block diagram of the QDMA Subsystem for PCIe.

Figure 1: **QDMA Architecture**



DMA Engines

Descriptor Engine

The H2C and C2H descriptors are fetched by the Descriptor Engine in one of two modes: Internal mode, and Descriptor bypass mode. The descriptor engine maintains per queue contexts where it tracks SW PIDX, CIDX, base address of the queue (BADDR), and queue configurations for each queue. The descriptor engine uses a round robin algorithm for fetching the descriptors. The descriptor engine has separate buffers for H2C and C2H, and ensures it never fetches more descriptors than available space. The Descriptor Engine will have only one DMA read outstanding per queue at a time and can many descriptors that can fit in MRRS. The Engine is responsible for reordering the out of order completions and ensures that descriptors for queue are always in order.

The descriptor bypass can be enabled on a per-queue basis and the fetched descriptors, after buffering are sent, to the respective bypass output interface instead of directly to the the H2C or C2H engine. In internal mode, based on the context settings the descriptors are sent to per H2C MM, C2H MM, H2C Stream, or C2H Stream engines.

The descriptor engine is also responsible for generating the status descriptor for the completion of the DMA operations.With the exception of C2H Stream mode, all modes use this mechanism to convey completion of each DMA operation so that software can reclaim descriptors and free up any associated buffers. This is indicated by CIDX field of status descriptor.

RECOMMENDED: *If a queue is associated with interrupt aggregation, Xilinx recommends that the status descriptor be turned off, and instead the DMA status be received from the interrupt aggregation ring. For details about the interrupt aggregation ring, see* Interrupt Aggregation Ring.

To put a limit on the number of fetched descriptors (for example, to limit the amount of buffering required to store the descriptor), it is possible to turn-on and throttle credit on a per-queue basis. In this mode, the descriptor engine fetches the descriptors up to available credit, and the total number of descriptors fetched per queue is limited to the credit provided. The user logic can return the credit through the `dsc_crdt` interface. The credit is in the granularity of the size of the descriptor.

To help the traffic manager prioritize the job, the available descriptor to be fetched (incremental PIDX value) of the PIDX update is sent to the user logic on the `tm_dsc_sts` interface. Using this interface it is possible to both prioritize and optimize the descriptor storage, and implement a DMA descriptor pull mode.

H2C MM Engine

The H2C MM Engine moves data from the host memory to card memory through the H2C AXI-MM interface. The engine generates reads on PCIe, splitting descriptors into multiple read requests based on the MRRS and the requirement that PCIe reads not to cross 4 KB boundaries. Once completion data for a read request is received, an AXI write is generated on the H2C AXI-MM interface. For source and destination addresses that are not aligned, the hardware will shift the data and split writes on AXI-MM to prevent 4K boundary crossing. Each completed descriptor is checked to determine whether a writeback and/or interrupt is required.

For Internal mode, the descriptor engine delivers memory mapped descriptors straight to H2C MM engine. The user logic can also inject the descriptor into the H2C bypass interface to move data from host to card memory. This gives the ability to do interesting things such as mixing control and DMA commands in the same queue. Control information can be sent to a control processor indicating the completion of DMA operation.

C2H MM Engine

The C2H MM Engine moves data from card memory to host memory through the C2H AXI-M interface. The engine generates AXI reads on the C2H AXI-M bus, splitting descriptors into multiple requests based on 4 KB boundaries. Once completion data for the read request is received on the AXI4 interface, a PCIe write is generated using the data from the AXI read as the contents of the write. For source and destination addresses that are not aligned, the hardware will shift the data and split writes on PCIe to obey Maximum Payload Size (MPS) and prevent 4 KB boundary crossings. Each completed descriptor is checked to determine whether a writeback and/or interrupt is required.

For Internal mode, the descriptor engine delivers memory mapped descriptors straight to C2H MM engine. As with H2C MM Engine, the user logic can also inject the descriptor into C2H bypass interface to move data from card to host memory.

For multi-function configuration support, the PCIe function number information will be provided in the `aruser` bits of the AXI-MM interface bus to help virtualization of card memory by the user logic. A parity bus, separate from the data and user bus, is also provided for end-to-end parity support.

H2C Stream Engine

The H2C stream engine moves data from the host to the H2C Stream interface. For internal mode, descriptors are delivered straight to the H2C stream engine; for a queue in bypass mode, the descriptors can be reformatted and fed to the bypass input interface. The engine is responsible for breaking up DMA reads to MRRS size, guaranteeing the space for completions, and also makes sure completions are reordered to ensure H2C stream data is delivered to user logic in-order.

The engine has sufficient buffering for up to 256 DMA reads and up to 32 KB of data. DMA fetches the data and aligns to the first byte to transfer on the AXI4 interface side This allows every descriptor to have random offset and random length. The total length of all descriptors must be less than 64 KB.

For internal mode queues, each descriptor defines a single AXI4-Stream packet to be transferred to the H2C AXI-ST interface. A packet with multiple descriptors straddling is not allowed due to the lack of per queue storage. However, packets with multiple descriptors straddling can be implemented using the descriptor bypass mode. In this mode, the H2C DMA engine can be initiated when the user logic has enough descriptors to form a packet. The DMA engine is initiated by delivering the multiple descriptors straddled packet along with other H2C ST packet descriptors through bypass interface, making sure they are not interleaved. Also, in bypass interface, the user logic can control the generation of the status descriptor.

C2H Stream Engine

The C2H streaming engine is responsible for reading data from the user logic and writing to the Host memory address provided by the C2H descriptor for a given Queue. It allows the user logic to send up to 252 bits of metadata along with packet, which will be placed into the C2H AXI-Stream Completion (CMPT) queue entry.

The C2H engine has two major blocks to accomplish C2H streaming DMA, Descriptor Prefetch Cache (PFCH), and CMPT. The PFCH and CMPT have per queue contexts to enhance the performance of their function and the software that is expected to program them.

PFCH cache has three main modes, on a per queue basis, called Simple Bypass Mode, Internal Cache Mode, and Cached Bypass Mode.

- In Simple Bypass Mode, the engine does not track anything for the queue, and the user logic can define its own method to receive descriptors. The user logic is then responsible for delivering the packet and associated descriptor in simple bypass interface. The ordering of the descriptors fetched by a queue in the bypass interface and the C2H stream interface must be maintained across all queues in bypass mode.
- In Internal Cache Mode and Cached Bypass Mode, the PFCH module offers storage for up to 512 descriptors and these descriptors can be used by up to 64 different queues. In this mode, the engine controls the descriptors to be fetched by managing the C2H descriptor queue credit on demand based on received packets in the pipeline. Pre-fetch mode can be turned on a per queue basis, and when enabled, causes the descriptors to be opportunistically pre-fetched so that descriptors are available before the packet data is available. The status can be found in prefetch context. This significantly reduces the latency by allowing packet data to be transferred to the PCIe integrated block almost immediately, instead of having to wait for the relevant descriptor to be fetched. The size of the buffer is fixed for a queue (PFCH context) and the engine can scatter the packet across as many as seven descriptors. In cached bypass mode descriptor is bypassed to user logic for further processing, such as address translation, and sent back on the bypass in interface. This mode does not assume any ordering descriptor and C2H stream packet interface, and the pre-fetch can match the packet and descriptors.

After the DMA completes the data transfer to Host, the DMA also updates Completion information to the CMPT ring in the Host for the corresponding Queue. Queue state and configuration are stored in CMPT context. CMPT context stores base address, CIDX, PIDX, and configurations. The software identifies the new CMPT entry being written based on the color bit of the status descriptor. The engine also can be configured on a per queue basis to generate an interrupt or a status descriptor update, or both, based on the needs of the software. If the interrupts for multiple queues are aggregated into the interrupt aggregation ring, the status descriptor information is available in the interrupt aggregation ring as well.

The CMPT engine has a cache of 32 entries to coalesce the multiple smaller CMPT writes into 64B writes to improve the PCIe efficiency. At any time, completions can be simultaneously coalesced for up to 32 queues. Beyond this, any additional queue that needs to write a CMPT entry will cause the eviction of the least recently used queue from the cache.

Bridge Interfaces

Bridge Master AXI Memory Mapped Interface

The Bridge AXI-MM Master interface is used for high bandwidth access to AXI Memory Mapped space from the host. The interface supports up to 32 outstanding AXI reads and writes. One or more PCIe BAR of any physical function (PF) or virtual function (VF) can be mapped to the Bridge master AXI-MM interface. This selection must be done at the point of configuring the IP. The function ID, BAR ID, VF group and VF group offset will be made available as part of `aruser` and `awuser` of the AXI-MM interface allowing the user logic to identify the source of each memory access. The `m_axib_awuser/m_axib_aruser` user bits mapping is as follows:

- `m_axib_awuser/m_axib_aruser[29:0]` is of 30 bits
- Where,
    - `m_axib_awuser/m_axib_aruser[7:0]`= Function number
    - `m_axib_awuser/m_axib_aruser[15:8]`= Reserved
    - `m_axib_awuser/m_axib_aruser[18:16]`= Bar id
    - `m_axib_awuser/m_axib_aruser[26:19]`= vfg offset
    - `m_axib_awuser/m_axib_aruser[28:27]`= vfg id

Virtual function group (VFG) refers to the VF group number. It is equivalent to the PF number associated with the corresponding VF. VFG_OFFSET refer to the VF number with respect to a particular PF. Note that this is not the FIRST_VF_OFFSET of each PF.

For example, if both PF0 and PF1 has 8 VFs, and FIRST_VF_OFFSET for PF0 and PF1 is 4 and 11 and below is the mapping for VFG and VFG_OFFSET.

Table 2: **AXI-MM Interface VFG**

Function Number	PF Number	VFG	VFG_OFFSET
0	0	0	0
1	1	0	0
4	0	0	0 (Because FIRST_VF_OFFSET for PF0 is 4, the first VF of PF0 starts at FN_NUM=4 and VFG_OSSET=0 indicates this is the first VF for PF0)
5	0	0	1 (VFG_OSSET=1 indicates this is the second VF for PF0)
...	...	...	...
12	1	1	0 (VFG=1 indicates this VF is associated with PF1)
13	1	1	1

Each host initiated access can be uniquely mapped to the 64 bit AXI address space through the PCIe to AXI BAR translation.

Since all functions shares the same AXI Master address space, a mechanism is needed to map request from different functions to a distinct address space on the AXI master side. This below shows how PCIe to AXI translation vector is used. Note that all VFs belonging to the same PF shares the same PCIe to AXI translation vector. Therefore, the AXI address space of each VF is concatenated together. Use VFG_OFFSET to calculate the actual starting address of AXI for a particular VF.

To summarize, `m_axib_awaddr` is determined as:

- For PF, `m_axib_awaddr` = `pcie2axi_vec` + `axib_offset`.
- For VF, `m_axib_awaddr` = `pcie2axi_vec` + (VFG_OFFSET + 1)*`vf_bar_size` + `axib_offset`.

Where `pcie2axi_vec` is PCIe to AXI BAR translation (that can be set during IP configuration.).

And `axib_offset` is the address offset in the requested target space.

Bridge Master AXI4-Lite Interface

One or more PCIe BAR of any physical function (PF) or virtual function (VF) can be mapped to the master AXI4-Lite interface. This selection must be done at the point of configuring the IP. The function ID, BAR ID (BAR hit), VF group, and VF group offset will be made available as part of `aruser` and `awuser` of the AXI4-Lite interface to help the user logic identify the source of memory access.

The `m_axil_awuser/m_axil_aruser` user bits mapping is as follows:

- `m_axil_awuser/m_axil_aruser[29:0]` is of 30 bits
- Where,
  - `m_axil_awuser/m_axil_aruser[7:0]` = Function number
  - `m_axil_awuser/m_axil_aruser[15:8]` = Reserved
  - `m_axil_awuser/m_axil_aruser[18:16]` = Bar id
  - `m_axil_awuser/m_axil_aruser[26:19]` = vfg offset
  - `m_axil_awuser/m_axil_aruser[28:27]` = vfg id

Virtual function group (VFG) refers to the VF group number. It is equivalent to the PF number associated with the corresponding VF. VFG_OFFSET refer to the VF number with respect to a particular PF. Note that this is not the FIRST_VF_OFFSET of each PF.

For example, if both PF0 and PF1 has 8 VFs, and FIRST_VF_OFFSET for PF0 and PF1 is 4 and 11 and below is the mapping for VFG and VFG_OFFSET.

Table 3: **AXI4-Lite Interface VFG**

Function Number	PF Number	VGF	VFG_OFFSET
0	0	0	0
1	1	0	0
4	0	0	0 (Because FIRST_VF_OFFSET for PF0 is 4, the first VF of PF0 starts at FN_NUM=4 and VFG_OSSET=0 indicates this is the first VF for PF0)
5	0	0	1 (VFG_OSSET=1 indicates this is the second VF for PF0)
...	...	...	...
12	1	1	0 (VFG=1 indicates this VF is associated with PF1)
13	1	1	1

Each host initiated access can be uniquely mapped to the 64 bit AXI address space through the PCIe to AXI BAR translation.

Because all functions shares the same AXI Master address space, a mechanism is needed to map requests from different functions to a distinct address space on the AXI master side. This below shows how PCIe to AXI translation vector is used. Note that all VFs belonging to the same PF shares the same PCIe to AXI translation vector. Therefore, the AXI address space of each VF is concatenated together. Use VFG_OFFSET to calculate the actual starting address of AXI for a particular VF.

To summarize, m_axil_awaddr is determined as:

- For PF, `m_axil_awaddr` = `pcie2axi_vec` + `axil_offset`.
- For VF, `m_axil_awaddr` = `pcie2axi_vec` + (VFG_OFFSET + 1)*`vf_bar_size` + `axil_offset`

Where `pcie2axi_vec` is PCIe to AXI BAR translation (that can be set during IP configuration.).

And `axib_offset` is the address offset in the requested target space.

Each of the host initiated access can be uniquely mapped to the 64 bit AXI address space. One outstanding read and one outstanding write are supported on this interface.

Expansion BAR can also be mapped to AXI4-Lite interface at the IP configuration time.

PCIe to AXI BARs

For each physical function, the PCIe configuration space consists of a set of six 32-bit memory BARs and one 32-bit EXPROM BAR. When SR-IOV is enabled, an additional six 32-bit BARs are enabled for each Virtual Functions. These BARs provide address translation to the AXI4 memory mapped spaced capability, interface routing, and AXI4 request attribute configuration. Any pairs of BARs can be configured as a single 64-bit BAR. Each BAR can be configured to route its requests to the QDMA register space, the Bridge AXI4-Lite Master interface, or the Bridge AXI4 MM Master interface. A programming example can be found in the Address Translation section (Example 3) of *AXI Bridge for PCI Express Gen3 Subsystem Product Guide* (PG194).

Request Memory Type

The memory type can be set for each PCIe BAR through attributes `attr_dma_pciebar2axibar_*_cache_pf*`.

- AxCache[0] is set to 1 for Modifiable, and 0 for Non-modifiable.
- AxCache[1] is set to 1 for Cacheable, and 0 for Non-cacheable.

Bridge Slave AXI Memory Mapped Interface

The Bridge AXI-MM Slave Interface is used for high bandwidth memory transfers between the user logic and the Host. AXI to PCIe translation is supported through the AXI to PCIe BARs. The interface will split requests as necessary to obey PCIe MPS and 4KB boundary crossing requirements. Up to 32 outstanding read and write requests are supported.

Bridge Slave AXI4-Lite Interface

The AXI4-Lite slave interface is used to access the AXI Bridge and QDMA internal registers. Upper 4 address bits indicate the access is for QDMA registers or Bridge registers.

- When `s_axil_awaddr[28]` = 1'b1, the write access is for QDMA registers.
- When `s_axil_awaddr[28] = 1'b0`, the write access is for Bridge registers (When accessing Bridge Registers, access from address 0x000 to 0xDFF will be redirected to PCIe core configuration space access and from address 0xE00 will be directed towards Bridge registers).
- When `s_axil_araddr[28]` = 1'b1, the read access is for QDMA registers.
- When `s_axil_araddr[28]` = 1'b0, the read access is for Bridge registers (When accessing Bridge Registers, access from address 0x000 to 0xDFF will be redirected to PCIe core configuration space access and from address 0xE00 will be directed towards Bridge registers).

The QDMA registers are virtualized for VFs and PFs. For example, VFs and PFs can access different parts of the address space, and each has access to its own queues. To accommodate the function specific accesses, the user logic can provide function ID on `s_axil_awuser[7:0]` for write access and `s_axil_aruser[7:0]` read access, which gives the QDMA proper internal register access. One outstanding read request and one outstanding write request are supported on Slave AXI4-Lite Interface.

The AXI4-Lite slave interface is also used to generate Vendor defined messages using the Bridge registers. For Vendor defined messages, see VDM.

AXI to PCIe BARs

In the Bridge Slave interface, there are six BARs which can be configured as 32 bits or 64 bits. These BARs provide address translation from AXI address space to PCIe address space. The address translation is configured for each AXI BAR through the following Vivado IP customization settings: **Aperture Base Address**, **Aperture High Address**, and **AXI to PCIe Translation**.

A programming example can be found in the Address Translation section (Example 4) of *AXI Bridge for PCI Express Gen3 Subsystem Product Guide* (PG194).

Interrupt Module

The IRQ module aggregates interrupts from various sources into the PCIe® integrated block core interface. The interrupt sources are queue-based interrupts, user interrupts and error interrupts.

Queue-based interrupts and user interrupts are allowed on PFs and VFs, but error interrupts are allowed only on PFs. If the SRIOV is not enabled, each PF has the choice of MSI-X, MSI interrupts, or both. With SRIOV enabled, only MSI-X interrupts are supported across all functions.

Support for MSI-X or MSI interrupts can be specified by attributes. Host system (Root Complex) will enable one or all of the interrupt types supported in hardware. If MSI-X is enabled, it takes precedence over MSI.

The PCIe integrated block core in UltraScale+™ devices offers up to eight interrupts per function. To allow many queues on a given PCIe function and each to have interrupts, the QDMA Subsystem for PCIe offers a novel way of aggregating interrupts from multiple queues to single interrupt vector. In this way, all 2048 queues could in principle be mapped to a single interrupt vector. QDMA offers 256 interrupt aggregation rings that can be flexibly allocated among the 256 available functions.

PCIe Block Interface

PCIe CQ/CC

The PCIe CQ/CC modules receive and process TLP requests from the remote PCIe agent. This interface to the UltraScale+™ Integrated Block for PCIe IP operates in address aligned mode. The module uses the BAR information from the Integrated Block for PCIe IP to determine where the request should be forwarded. The three possible destinations for these requests are:

- the internal configuration module
- the AXI4 MM Bridge Master interface
- the AXI4-Lite Bridge Master interface

Non-posted requests are expected to receive completions from the destination, which are forwarded to the remote PCIe agent. For further details, see the *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213).

PCIe RQ/RC

The role of the PCIe RQ/RC interface is to generate PCIeTLPs on the RQ bus and process PCIe Completion TLPs from the RC bus. This interfaces to the UltraScale+™ Integrated Block for PCIe® core operates in DWord aligned mode. With a 512-bit interface, straddling must also be enabled. While straddling is supported, all combinations of RQ straddled transactions as defined in the *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213) may not be implemented.

PCIe Configuration

Several factors can throttle outgoing non-posted transactions. Outgoing non-posted transactions are throttled based on flow control information from the PCIe® integrated block to prevent head of line blocking of posted requests. PCIe® Finite Completion Credits can be enabled when customizing the IP in the Vivado® Integrated Design Environment. This option is not enabled by default. If not enabled, the DMA will meter non-posted transactions based on the PCIe Receive FIFO space.

General Design of Queues

The multi-queue DMA engine of the QDMA Subsystem for PCIe uses RDMA model queue pairs to allow RNIC implementation in the user logic. Each queue set consists of Host to Card (H2C), Card to Host (C2H), and a C2H Stream Completion (CMPT). The elements of each queue are descriptors.

H2C and C2H are always written by the driver/software; hardware always reads from these queues. H2C carries the descriptors for the DMA read operations from Host. C2H carries the descriptors for the DMA write operations to the Host.

In internal mode, H2C descriptors carry address and length information and are called gather descriptors. They support 32 bits of meta data that can be passed from software to hardware along with every descriptor. The descriptor can be memory mapped (where it carries host address, card address, and length of DMA transfer) or streaming (only host address, and length of DMA transfer) based on context settings. Through descriptor bypass, the arbitrary descriptor format can be defined, where software can pass immediate data and/or additional metadata along with packet.

C2H queue memory mapped descriptors include the card address, the host address and the length. In streaming internal cached mode, descriptors carry only the host address. The buffer size of the descriptor, which is programmed by the driver, is expected to be of fixed size for the whole queue. Actual data transferred associated with each the descriptor does not need to be the full length of the buffer size.

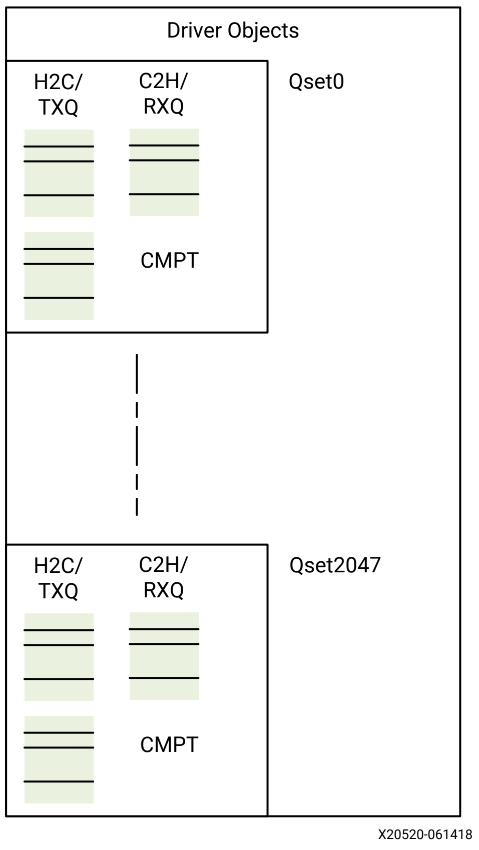
The software advertises valid descriptors for H2C and C2H queues by writing its producer index (PIDX) to the hardware. The status descriptor is the last entry of the descriptor ring, except for a C2H stream ring. The status descriptor carries the consumer index (CIDX) of the hardware so that the driver knows when to reclaim the descriptor and deallocate the buffers in the host.

For the C2H stream mode, C2H descriptors will be reclaimed based on the CMPT queue entry. Typically, this carries one entry per C2H packet, indicating one or more C2H descriptors is consumed. The CMPT queue entry carries enough information for software to claim all the descriptors consumed. Through external logic, this can be extended to carry other kinds of completions or information to host.

CMPT entry written by hardware to the ring can be detected by the driver using either the color bit in the descriptor or the status descriptor at the end of the CMPT ring. This CMPT descriptor can carry meta data for C2H stream packet and can also serve as a custom completion or immediate notification for user application. The CMPT supports two formats:

- **Internal format:** The descriptor conveys meta data, color bit, descriptor format, error, and packet length.
- **User format:** The length is not provided by the QDMA Subsystem for PCIe, which then becomes the responsibility of the user logic to provide.

Figure 2: **Queue Ring Architecture**



The software can program 16 different ring sizes. The ring size for each queue can be selected from context programing. The last queue entry is the descriptor status, and allowable entries are queue size -1.

For example, if queue size is 8, which contains the entry index 0 to 7, the last entry (index 7) is reserved for status. This index should never be used for PIDX update, and PIDX update should never be equal to CIDX. For this case, if CIDX is 0, the maximum PIDX update would be 6.

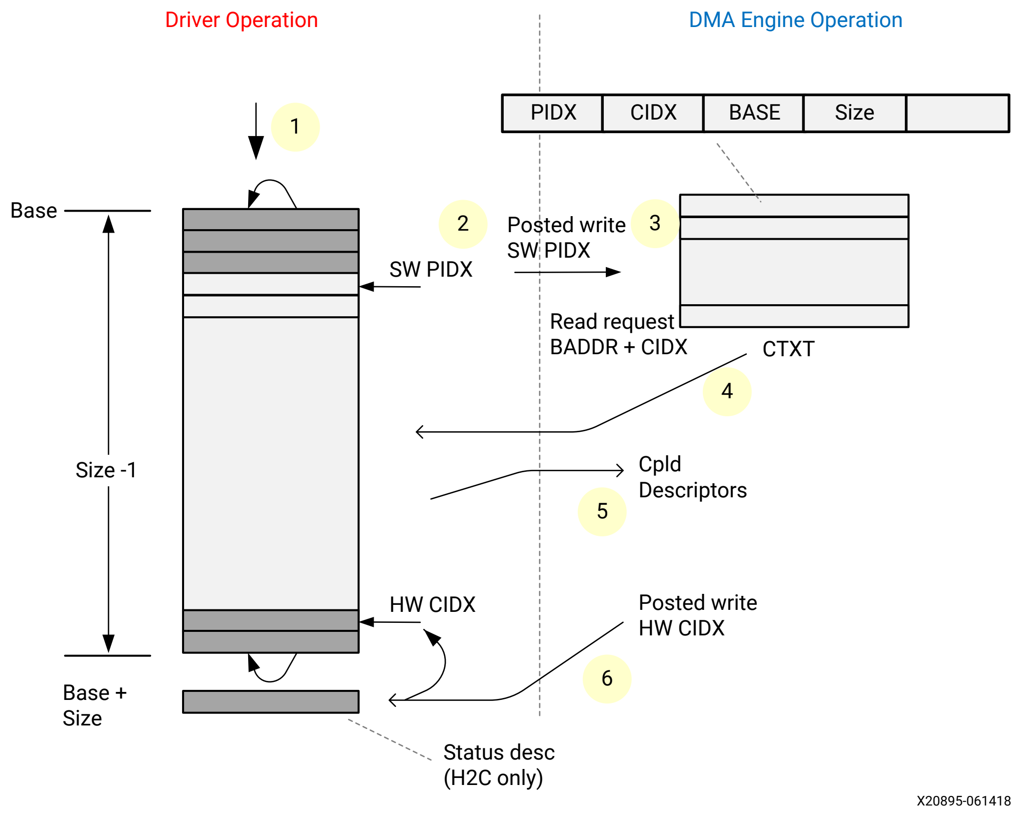
In the example above, if traffic has already started and CIDX is at 4, the maximum PIDX update would be 3.

H2C and C2H Queues

H2C/C2H queues are circular rings, located in host memory. For both type of queues, the producer is software and consumer is the descriptor engine. The software maintains producer index (PIDX) and a copy of hardware consumer index (HW CIDX) to avoid overwriting unread descriptor. The descriptor engine also maintains consumer index (CIDX) and a copy of SW PIDX to make sure, the engine does not read unwritten descriptor. Last entry in the queue is dedicated for status descriptor where the engine writes the HW CIDX and other status.

The engine maintains total of 2048 H2C and 2048 C2H contexts in local memory. The context stores properties of the circular queue, such as base address (BADDR), SW PIDX, CIDX, and depth of the queue.

Figure 3: **Simple H2C and C2H Queue**



The figure above shows the H2C and C2H fetch operation.

1. For H2C, the Driver writes payload into host buffer, forms the H2C descriptor with the payload buffer information and puts it into H2C queue at the PIDX location. For C2H, the driver forms the descriptor with free buffer for hardware to DMA write the packet.
2. The software sends the posted write to PIDX register in the descriptor engine for the associated Queue ID (QID) with its current PIDX value.

3. Upon reception of the PIDX update, the engine calculates the absolute QID of the pointer update based on address offset and function ID. Using the QID, the engine will fetch the context for the absolute QID from the memory associated with the QDMA Subsystem for PCIe.
4. The engine determines the number of descriptors that are allowed to be fetched based on the context. The engine calculates the descriptor address using the base address (BADDR), CIDX, and descriptor size, and the engine issues the DMA read request.
5. After the descriptor engine receives the read completion from the host memory, the engine forms descriptors and delivers them to the H2C Engine or C2H Engine in internal mode. In case of bypass, the descriptors are sent out to the associated descriptor bypass output interface.
6. For memory mapped or H2C stream queues programmed as internal mode, after the fetched descriptor is completely processed, the engine writes the CIDX value to the status descriptor. For queues programmed as bypass mode, user logic controls the write back through bypass in interface. The status descriptor could be moderated based on context settings. C2H stream queues always use the CMPT ring for the completions.

For C2H, the fetch operation is implicit through the write back ring.

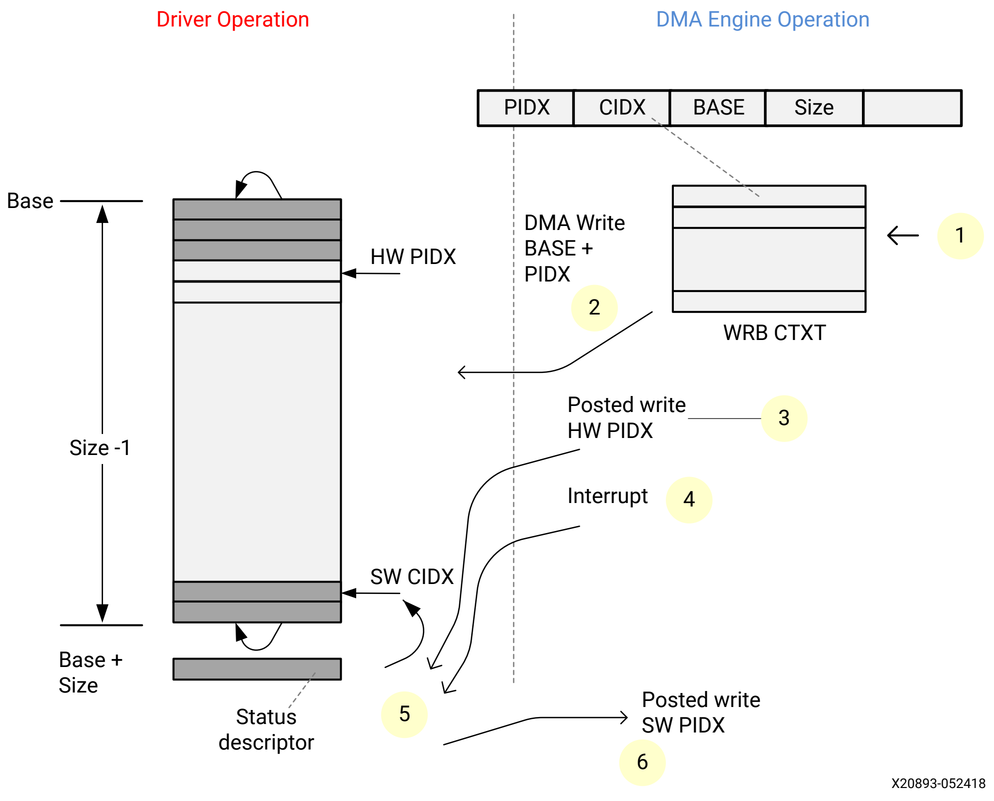
***Note*:** C2H operates in pull mode of the descriptor, and H2C can be either pull or push mode.

Completion Queue

The CMPT queue is a circular ring, located in host memory. The consumer is software, and the produce is the CMPT engine. The software maintains producer index (PIDX) and a copy of hardware consumer index (HW CIDX) to avoid reading unwritten completion. The CMPT engine also maintains PIDX and a copy of software consumer index (SW CIDX) to make sure, the engine does not overwrite unread completion. Last entry in the queue is dedicated for status descriptor where the engine writes the HW PIDX and other status.

The engine maintains total of 2048 CMPT contexts in local memory. The context stores properties of the circular queue such as base address, SW CIDX, PIDX, depth of the queue, etc.

Figure 4: **Simple CMPT Queue Flow**



C2H stream is expected to use CMPT queue for completions to host, but it can also be used for other types of completion or for sending the messages to host driver. The message through CMPT is guaranteed to not bypass the corresponding C2H stream packet DMA.

Simple flow of DMA CMPT queue operation with respect to the numbering above.

1. The CMPT engine receives the completion message through the CMPT interface, but the QID for the completion message comes from the C2H stream interface. The engine reads the QID index of CMPT context RAM.

2. The DMA writes the CMPT entry to address BASE+PIDX.
3. If all the conditions are met, optionally write PIDX to status descriptor of CMPT queue with color bit.
4. If interrupt mode is enabled, generate the interrupt event message to interrupt module.
5. The software could be in polling or interrupt mode. Either way, the software identifies the new CMPT entry either by matching color bit or by comparing the PIDX value in status descriptor against its current software CIDX value.
6. Sends posted write back to queue DMA with SW CIDX. This allows the hardware to reuse the descriptors again. After the software finishes processing the CMPT, that is, before it stops polling or leaving the interrupt handler, the software issues a write to CIDX update register for the associated queue.

SR-IOV Support

The QDMA Subsystem for PCIe provides an optional feature to support the Single Root I/O Virtualization (SR-IOV). The PCI-SIG® Single Root I/O Virtualization and Sharing (SR-IOV) specification (available from *PCI-SIG Specifications*(www.pcisig.com/specifications) standardizes the method for bypassing the VMM involvement in datapath transactions and allows a single PCI Express® Endpoint to appear as multiple separate PCI Express Endpoints. SR-IOV classifies the functions as:

- **Physical Functions (PF)**: Full featured PCIe® functions which include SR-IOV capabilities among others.
- **Virtual Functions (VF)**: PCIe functions featuring configuration space with Base Address Registers (BARs) but lacking the full configuration resources and controlled by the PF configuration. The main role of the VF is data transfer.

Apart from PCIe defined configuration space, QDMA Subsystem for PCI Express virtualizes data path operations, such as pointer updates for queues, and interrupts. The rest of the management and configuration functionality (or a slow path) is deferred to the physical function driver. The Drivers that do not have sufficient privilege must communicate with the privileged Driver through the mailbox interface which is provided in part of the QDMA Subsystem for PCI Express.

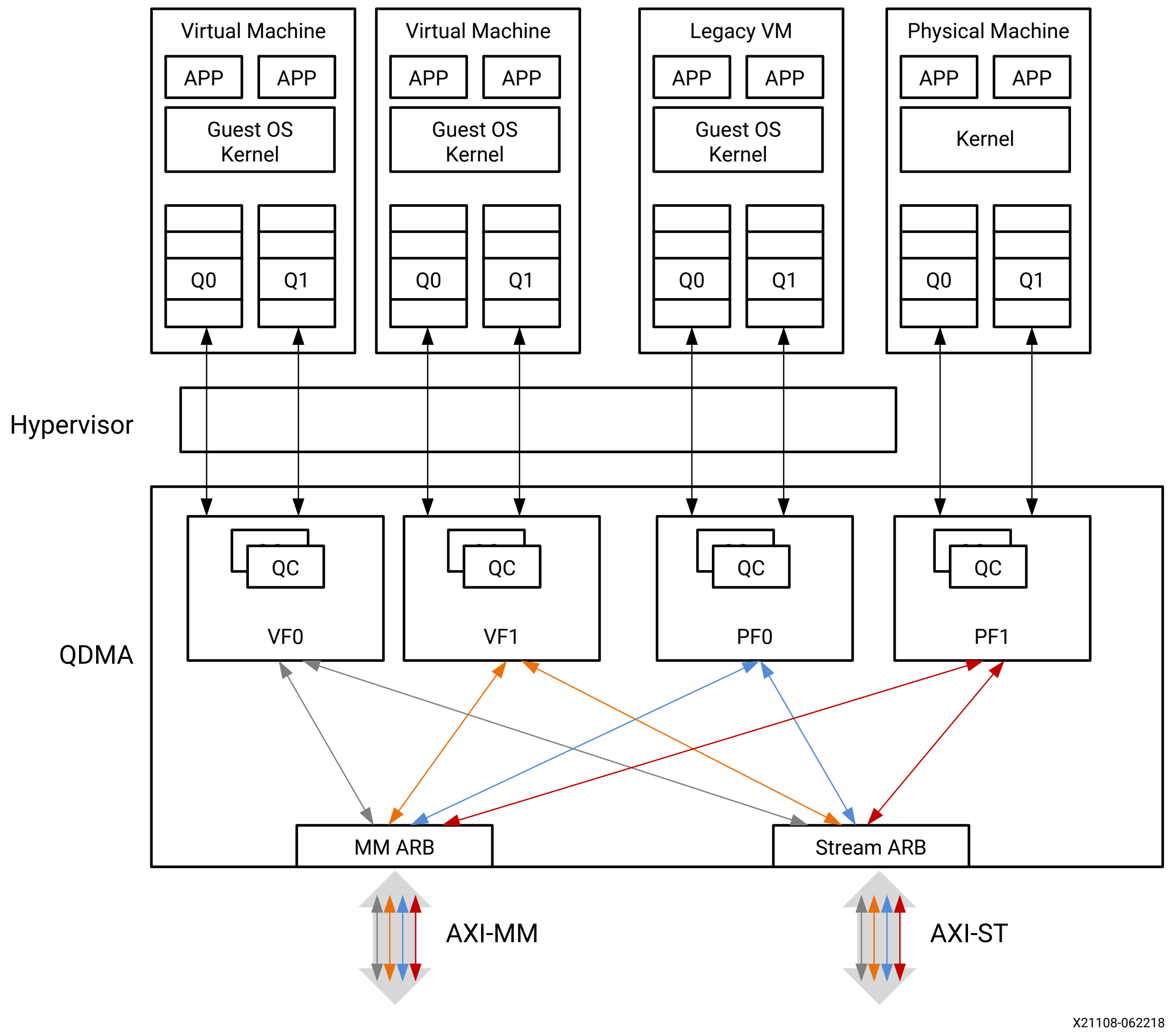
The security is an important aspect of virtualization. The QDMA Subsystem for PCI Express offers the following security functionality:

- QDMA allows only privileged PF to configure the per queue context and registers.
- Drivers are allowed to do pointer updates only for the queue allocated to them.
- The system IOMMU can be turned on to check that the DMAs being requested by PFs and VFs. The ARID comes from queue context programmed by a privileged function.

Any PF or VF can communicate to a PF (not itself) through mailbox. Each function implements one 64B inbox and 64B outbox. These mailboxes are visible to the driver in the DMA BAR (typically BAR0) of its own function. At any given time, any function can have one outgoing mailbox and one incoming mailbox message outstanding per function.

The diagram below shows how a typical system can use QDMA with different functions and Operating system. Different Queues can be allocated to different functions and how each function can transfer DMA packets independent of each other.

Figure 5: **QDMA in a System**



Applications

The QDMA Subsystem for PCIe is used in a broad range of networking, computing, and data storage applications.

A common usage example for the QDMA Subsystem for PCIe is to implement Data Center and Telco applications, such as Compute accelerations, Smart NIC, NVMe, RDMA-enabled NIC (RNIC), server virtualization, and NFV in the user logic. Multiple applications can be implemented to share the QDMA by assigning different queue sets and PCIe functions to each application. These Queues can then be scaled in the user logic to implement rate limiting, traffic priority, and custom work queue entry (WQE).

Licensing and Ordering

This Xilinx® LogiCORE™ IP module is provided at no additional cost with the Xilinx Vivado® Design Suite under the terms of the Xilinx End User License.

For more information about this subsystem, visit the QDMA Subsystem for PCIe product page web page.

Chapter 3

Product Specification

Standards

This subsystem adheres to the following standard(s):

- *AMBA AXI4-Stream Protocol Specification* (ARM IHI 0051A)
- PCI Express Base Specification v3.1
- PCI Local Bus Specification
- PCI-SIG® Single Root I/O Virtualization and Sharing (SR-IOV) Specification

The PCI specifications are available at *PCI-SIG Specifications* (http://www.pcisig.com/specifications).

Minimum Device Requirements

Gen3x16 capability requires a minimum of a -2 speed grade.

Table 4: **Minimum Device Requirements**

Capability Link Speed	Capability Link Width	Supported Speed Grades
UltraScale+ ™ Family		
Gen1/Gen2	x1, x2, x4, x8, x16	-1, -1L, -1LV, -2, -2L, -2LV, -3
Gen3	x1, x2, x4	-1, -1L, -1LV, -2, -2L, -2LV, -3
	x8	-1, -2, -2L, -2LV, -3
	x16	-2, -2L, -3
Virtex® UltraScale+ with HBM		
Gen1/Gen2	x1, x2, x4, x8, x16	-1, -2, -2L, -2LV, -3
Gen3	x1, x2, x4	-1, -2, -2L, -2LV, -3
	x8	-1, -2, -2L, -2LV, -3
	x16	-2, -2L, -3

***Note*:** This IP currently supports XCVU3P/5P/7P/9P/11P/13P, XQVU7P/11P, XCVU37P,XCZU11EG/17EG/19EG, XQZU19EG, and XCKU5P/11P/15P devices only.

QDMA Operations

Descriptor Engine

The descriptor engine is responsible for managing the consumer side of the H2C and C2H descriptor ring buffers for each queue. The context for each queue determines how the descriptor engine will process each queue individually. When descriptors are available and other conditions are met, the descriptor engine will issue read requests to PCIe to fetch the descriptors. Received descriptors are offloaded to either the descriptor bypass out interface (bypass mode) or delivered directly to a DMA engine (internal mode). When a H2C Stream or Memory Mapped DMA engine completes a descriptor, status can be written back to the status descriptor, an interrupt, and/or a marker response can be generated to inform software and user logic of the current DMA progress. The descriptor engine also provides a Traffic Manager Interface which notifies user logic of certain status for each queue. This allows the user logic to make informed decisions if customization and optimization of DMA behavior is desired.

Descriptor Context

The Descriptor Engine stores per queue configuration, status and control information in descriptor context that can be stored in block RAM or URAM, and the context is indexed by H2C or C2H QID. Prior to enabling the queue, the hardware and credit context must first be cleared. After this is done, the software context can be programmed and the `qen` bit can be set to enable the queue. After the queue is enabled, the software context should only be updated through the direct mapped address space to update the Producer Index and Interrupt Arm bit, unless the queue is being disabled. For details, see QDMA_DMAP_SEL_H2C_DSC_PIDX[2048] (0x6404) and QDMA_DMAP_SEL_C2H_DSC_PIDX[2048] (0x6408). The hardware context and credit context contain only status. It is only necessary to interact with the hardware and credit contexts as part of queue initialization in order to clear them to all zeros. Once the queue is enabled, context is dynamically updated by hardware. Any modification of the context through the indirect bus when the queue is enabled can result in unexpected behavior. Reading the context when the queue is enabled is not recommended as it can result in reduced performance.

Software Descriptor Context Structure (0x0 C2H and 0x1 H2C)

The descriptor context is used by the descriptor engine.

Table 5: **Software Descriptor Context Structure Definition**

Bit	Bit Width	Field Name	Description
[127:64]	64	dsc_base	Base address of Descriptor Ring.
[63]	1	is_mm	This field is applicable only for internal mode. If this field is set then the descriptors will be delivered to associated H2C or C2H MM engine.
[62]	1	mrkr_dis	If set, disables the marker response in internal mode. Not applicable for C2H ST.
[61]	1	irq_req	Interrupt due to error waiting to be sent (waiting for irq_arm). This bit should be cleared when the queue context is initialized. Not applicable for C2H ST.
[60]	1	err_wb_sent	A writeback/interrupt was sent for an error. Once this bit is set no more writebacks or interrupts will be sent for the queue. This bit should be cleared when the queue context is initialized. Not applicable for C2H ST.
[59:58]	2	err	Error status. Bit[1] dma – An error occurred during DMA operation. Check engine status registers. Bit[0] dsc – An error occured during descriptor fetch or update. Check descriptor engine status registers. This field should be set to 0 when the queue context is initialized.
[57]	1	irq_no_last	No interrupt was sent and pidx/cidx was idle in internal mode. When the irq_arm bit is set, the interrupt will be sent. This bit will clear automatically when the interrupt is sent or if the PIDX of the queue is updated. This bit should be initialized to 0 when the queue context is initialized. Not applicable for C2H ST.
[56:54]	3	port_id	Port_id The port id that will be sent on user interfaces for events associated with this queue.
[53]	1	irq_en	Interrupt enable. An interrupt to the host will be sent on host status updates. Set to 0 for C2H ST.
[52]	1	wbk_en	Writeback enable. A memory write to the status descriptor will be sent on host status updates.
[51]	1	mm_chn	Set to 0.
[50]	1	bypass	If set, the queue will operate under Bypass mode, otherwise it will be in Internal mode.
[49:48]	2	dsc_sz	Descriptor size. 0: 8B, 1:16B; 2:32B; 3:rsv 32B is required for Memory Mapped DMA. 16B is required for H2C Stream DMA. 8B is required for C2H Stream DMA.
[47:44]	4	rng_sz	Descriptor ring size index to ring size registers.
[43:36]	8	fnc_id	Function ID. The function to which this queue belongs.

Table 5: **Software Descriptor Context Structure Definition** *(cont'd)*

Bit	Bit Width	Field Name	Description
[35]	1	wbi_intvl_en	Write back/Interrupt interval. Enables periodic status updates based on the number of descriptors processed. Applicable to Internal mode. Not Applicable to C2H ST. The writeback interval is determined by QDMA_GLBL_DSC_CFG.wb_acc_int.
[34]	1	wbi_chk	Writeback/Interrupt after pending check. Enable status updates when the queue has completed all available descriptors. Applicable to Internal mode.
[33]	1	fcrd_en	Enable fetch credit. The number of descriptors fetched will be qualified by the number of credits given to this queue. Set to 1 for C2H ST.
[32]	1	qen	Indicates that the queue is enabled.
[31:17]	15	rsv	Reserved.
[16]	1	irq_arm	Interrupt Arm. When this bit is set, the queue is allowed to generate an interrupt.
[15:0]	16	pidx	Producer Index.

Hardware Descriptor Context Structure (0x2 C2H and 0x3 H2C)

Table 6: **Hardware Descriptor Structure Definition**

Bit	Bit Width	Field Name	Description
[47:43]	5	rsvd	Reserved
[42]	1	fetch_pnd	Descriptor fetch pending.
[41]	1	idl_stp_b	Queue invalid and no descriptors pending. This bit is set when the queue is enabled. The bit is cleared when the queue has been disabled (software context qen bit) and no more descriptor are pending.
[40]	1	dsc_pnd	Descriptors pending. Descriptors are defined to be pending if the last CIDX completed does not match the current PIDX.
[39:32]	8	reserved	
[31:16]	16	crd_use	Credits consumed. Applicable if fetch credits are enabled in the software context.
[15:0]	16	cidx	Consumer Index of last fetched descriptor.

Credit Descriptor Context Structure

Table 7: **Credit Descriptor Context Structure Definition**

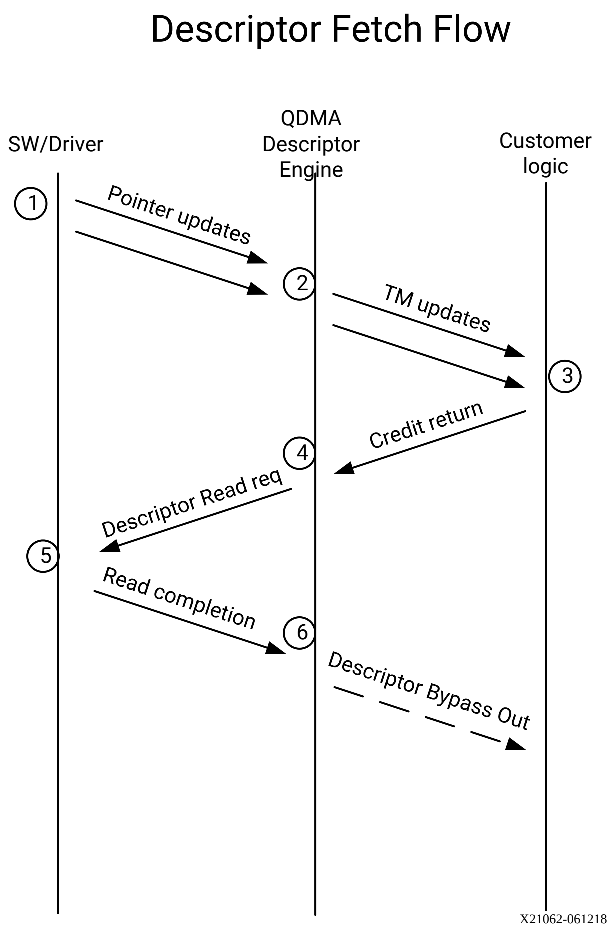
Bit	Bit Width	Field Name	Description
[31:16]	16	rsvd	Reserved

Table 7: **Credit Descriptor Context Structure Definition** *(cont'd)*

Bit	Bit Width	Field Name	Description
[15:0]	16	credt	Fetch credits received. Applicable if fetch credits are enabled in the software context.

Descriptor Fetch

Figure 6: **Descriptor Fetch Flow**



1. The descriptor engine is informed of the availability of descriptors through an update to a queue's descriptor PIDX context. This portion of the context is direct mapped to the QDMA_DMAP_SEL_H2C_DSC_PIDX and QDMA_DMAP_SEL_C2H_DSC_PIDX address space.
2. On a PIDX update, the descriptor engine evaluates the number of descriptors available based on the last fetched consumer index (CIDX). The availability of new descriptors is communicated to the user logic through the Traffic Manager Status Interface.
3. If fetch crediting is enabled, the user logic is required to provide a credit for each descriptor that should be fetched.

4. If descriptors are available and either fetch credits are disabled or are non-zero, the descriptor engine will generate a descriptor fetch to PCIe. The number of descriptors fetch is further qualified by the PCIe Max Read Request Size (MRRS) and descriptor fetch credits, if enabled. A descriptor fetch can also be stalled due to insufficient completion space. In each direction, C2H and H2C are allocated 256 entries for descriptor fetch completions. Each entry is the width of the datapath. If sufficient space is available, the fetch is allowed to proceed. A given queue can only have one descriptor fetch pending on PCIe at any time.
5. The host receives the read request and provides the descriptor read completion to the descriptor engine.
6. Descriptors are stored in a buffer until they can be offloaded. If the queue is configured in bypass mode, the descriptors are sent to the Descriptor Bypass Output port. Otherwise they are delivered directly to a DMA engine. Once delivered, the descriptor fetch completion buffer space is deallocated.

At any time, the software should not update PIDX more than a ring_size of - 2 (available descriptors are always as ring size -2).

Internal Mode

A queue can be configured to operate in Descriptor bypass mode or Internal mode by setting the software context bypass field. In internal mode, the queue requires no external user logic to handle descriptors. Descriptors that are fetched by the descriptor engine are delivered directly to the appropriate DMA engine and processed. Internal mode allows fetch crediting and status updates to user logic for run time customization of the descriptor fetch behavior.

Internal Mode Writeback and Interrupts (AXI MM and H2C ST)

Status writebacks and/or interrupts are generated automatically by hardware based on the queue context. When “wbi_intvl_en” is set, writebacks/interrupts will be sent based on the interval selected in the register QDMA_GLBL_DSC_CFG.wb_intvl. Due to the slow nature of interrupts, in interval mode, interrupts may be late or skip intervals. If the wbi_chk context bit is set, a writeback/interrupt will be sent when the descriptor engine has detected that the last descriptor at the current PIDX has completed. It is recommended the `wbi_chk` bit be set for all internal mode operation, including when interval mode is enabled. An interrupt will not be generated until the `irq_arm` bit has been set by software. Once an interrupt has been sent the `irq_arm` bit is cleared by hardware. Should an interrupt be needed when the `irq_arm` bit is not set, the interrupt will be held in a pending state until the `irq_arm` bit is set.

Descriptor completion is defined to be when the descriptor data transfer has completed and its write data has been acknowledged on AXI (H2C bresp for AXI MM, Valid/Ready of ST), or been accepted by the PCIe Controller’s transaction layer for transmission (C2H MM).

Bypass Mode

Bypass mode also supports crediting and status updates to user logic. In addition, bypass mode allows user logic to customize processing of descriptors and status updates. Descriptors fetched by the descriptor engine are delivered to user logic through the descriptor bypass out interface. This allows user logic to pre-process or store the descriptors, if desired. On the bypass out interface, the descriptors can be a custom format (adhering to the descriptor size). To perform DMA operations, the user logic drives descriptors (must be QDMA format) into the descriptor bypass input interface.

If the user logic already has descriptors, which must be in QDMA format, it can be provided directly to the DMA through the descriptor bypass ports. The user logic does not need to fetch descriptors from the host if the descriptors are already in the user logic.

Bypass Mode Writeback/Interrupts

In bypass mode, the user logic has explicit control over status updates to the host, and marker responses back to user logic. Along with each descriptor submitted to the Descriptor Bypass Input Port for a Memory Mapped Engine or H2C Stream DMA engine, there is a CIDX, and `wbi` field. The CIDX is used to identify which descriptor has complete in any status update (host writeback, marker response, or coalesced interrupt) generated at the completion of the descriptor. If the `wbi` field of the descriptor was input, then a writeback to the host will be generated if the context `wbk_en` bit is set. An interrupt can also be sent if the `wbi` bit is set if the context `irq_en` and `irq_arm` bits are set.

If interrupts are enabled, the user logic must monitor the traffic manager output for the `irq_arm`. Only once the `irq_arm` bit has been observed for the queue, should a descriptor with the `wbi` bit set be sent to the DMA. Once a descriptor with the `wbi` bit has been sent, another `irq_arm` assertion must be observed before another descriptor with the `wbi` bit can be sent. If the user sets the `wbi` bit when the arm bit has not be properly observed, an interrupt may or may not be sent, and software waiting indefinitely for an interrupt. When interrupts are not enabled, setting the `wbi` bit has no restriction. However excessive writebacks events can severly reduce the descriptor engine performance and consume write bandwidth to the host.

Descriptor completion is defined to be when the descriptor data transfer has completed and its write data has been acknowledged on AXI (H2C bresp for AXI MM, Valid/Ready of ST), or been accepted by the PCIe Controller’s transaction layer for transmission (C2H MM).

Bypass Mode Marker Response

Marker responses can be generated for any descriptor by setting the `mrkr_req` bit. Marker responses are generated after the descriptor is completed. Similar to host writebacks, excessive marker response requests can reduce descriptor engine performance. Marker responses to the user logic can also be sent with the `wbi` bit if configured in the context. The marker response sent can be identified by the CIDX associated with the descriptor, as well as the queue id, and direction of the DMA.

Descriptor completion is defined to be when the descriptor data transfer has completed and its write data has been acknowledged on AXI (H2C bresp for AXI MM, Valid/Ready of ST), or been accepted by the PCIe Controller's transaction layer for transmission (C2H MM).

Traffic Manager Output Interface

The traffic manager interface provides details of a queue's status to user logic, allowing user logic to manage descriptor fetching and execution. In normal operation, for an enabled queue, each time the `irq_arm` bit is asserted or PIDX of a queue is updated, the descriptor engine asserts `tm_dsc_sts_valid`. The `tm_dsc_sts_avl` signal indicates the number of new descriptors available since the last update. Through this mechanism, user logic can track the amount of work available for each queue. This can be used for prioritizing fetches through the descriptor engine's fetch crediting mechanism or other user optimizations. On the valid cycle, the `tm_dsc_sts_irq_arm` indicates that the `irq_arm` bit was zero and was set. In bypass mode, this is essentially a credit for an interrupt for this queue. See Bypass Mode Interrupts above. When a queue is invalidated by software or due to error, the `tm_dsc_sts_qinv` bit will be set. If this bit is observed, the descriptor engine will have halted new descriptor fetches for that queue. In this case, the contents on `tm_dsc_sts_avl` indicate the number of available fetch credits held by the descriptor engine. This information can be used to help user logic reconcile the number of credits given to the descriptor engine, and the number of descriptors it should expect to receive. Even after `tm_dsc_sts_qin` is asserted, valid descriptors already in the fetch pipeline will continue to be delivered to the DMA engine (internal mode) or delivered to the descriptor bypass output port (bypass mode).

Other fields of the `tm_dsc_sts` interface identify the queue id, DMA direction (H2C or C2H), internal or bypass mode, stream or memory mapped mode, queue enable status, queue error status, and port ID.

While the `tm_dsc_sts` interface is a valid/ready interface, it should not be back-pressured for optimal performance. Since multiple events trigger a `tm_dsc_sts` cycle, if internal buffering is filled, descriptor fetching will be halted to prevent generation of new events.

For detailed port information, see the QDMA Traffic Manager Credit Output Ports.

Descriptor Credit Input Interface

The credit interface is relevant when a queue's `fcrd_en` context bit is set. It allows the user logic to prioritize and meter descriptors fetched for each queue. You can specify the DMA direction, qid, credit value, and operand. In normal use, the operand should be 2'h0, which adds credits to the descriptor engine, allowing it to fetch descriptors. Internally, credits received and consumed are tracked for each queue. If credits are added when the queue is not enabled, the credits will be returned through the Traffic Manager Output Interface with `tm_dsc_sts_qinv` asserted, and the returned credits in `tm_dsc_sts_avl`. For cases involving C2H ST queue invalidation and cleanup, the operand 2'h2 can be used to rollback the queue's CIDX. This should only be done when the queue is not enabled.

For more detailed port information, see QDMA Descriptor Credit Input Ports.

Errors

Errors can potentially occur during both descriptor fetch and descriptor execution. In both cases, once an error is detected for a queue it will invalidate the queue, log an error bit in the context, stop fetching new descriptors for the queue which encountered the error, and can also log errors in status registers. If enabled for writeback, interrupts, or marker response, the DMA will generate a status update to these interfaces. Once this is done, no additional writeback, interrupts, or marker responses (internal mode) will be sent for the queue until the queue context is cleared. As a result of the queue invalidation due to an error, a Traffic Manager Output cycle will also be generated to indicate the error and queue invalidation.

Although additional descriptor fetches will be halted, fetches already in the pipeline will continue to be processed and descriptors will be delivered to a DMA engine or Descriptor Bypass Out interface as usual. If the descriptor fetch itself encounters an error, the descriptor will be marked with an error bit. If the error bit is set, the contents of the descriptor should be considered invalid. It is possible that subsequent descriptor fetches for the same queue do not encounter an error and will not have the error bit set.

Memory Mapped DMA

In memory mapped DMA operations, both the source and destination of the DMA are memory mapped space. In an H2C transfer, the source address belongs to PCIe address space while the destination address belongs to AXI MM address space. In a C2H transfer, the source address belongs to AXI MM address space while the destination address belongs to PCIe address space. PCIe-to-PCIe, and AXI MM-to-AXI MM DMAs are not supported. Aside from the direction of the DMA, transfer H2C and C2H DMA behave similarly and share the same descriptor format.

Operation

The memory mapped DMA engines (H2C and C2H) are enabled by setting the `run` bit in the Memory Mapped Engine Control Register. When the `run` bit is deasserted, descriptors can be dropped. Any descriptors that have already started the source buffer fetch will continue to be processed. Reassertion of the `run` bit will result in resetting internal engine state and should only be done when the engine is quiesced. Descriptors are received from either the descriptor engine directly or the Descriptor Bypass Input interface. Any queue that is in internal mode should not be given descriptors through the Descriptor Bypass Input interface. Any descriptor sent to an MM engine that is not running will be dropped. For configurations where a mix of Internal Mode queues and Bypass Mode queues are enabled, round robin arbitration is performed to establish order.

The DMA Memory Mapped engine first generates the read request to the source interface, splitting the descriptor at alignment boundaries specific to the interface. Both PCIe and AXI read interfaces can be configured to split at different alignments. Completion space for read data is preallocated when the read is issued. Likewise for the write requests, the DMA engine will split at appropriate alignments. On the AXI interface each engine will use a single AXI ID. The DMA engine will reorder the read completion/write data to the order in which the reads were issued. Once sufficient read completion data is received the write request will be issued to the destination interface in the same order that the read data was requested. Before the request is retired, the destination interfaces must accept all the write data and provide a completion response. For PCIe the write completion is issued when the write request has been accepted by the transaction layer and will be sent on the link next. For the AXI Memory Mapped interface, the `bresponse` is the completion criteria. Once the completion criteria has been met, the host writeback, interrupt and/or marker response is generated for the descriptor as appropriate. See Descriptor Engine Internal Mode Writeback and Interrupts, and Bypass Mode Writeback and Interrupts.

The DMA Memory Mapped engines also support the `no_dma` field of the Descriptor Bypass Input, and zero-length DMA. Both cases are treated identically in the engine. The descriptors propagate through the DMA engine as all other descriptors, so descriptor ordering within a queue is still observed. However no DMA read or write requests are generated. The status update (writeback, interrupt, and/or marker response) for zero-length/`no_dma` descriptors is processed when all previous descriptors have completed their status update checks.

Errors

There are two primary error categories for the DMA Memory Mapped Engine. The first is an error bit that is set with an incoming descriptor. In this case, the DMA operation of the descriptor is not processed but the descriptor will proceed through the engine to status update phase with an error indication. This should result in a writeback, interrupt, and/or marker response depending on context and configuration. It will also result in the queue being invalidated. See Descriptor Engine Errors. The second category of errors for the DMA Memory Mapped Engine are errors encountered during the execution of the DMA itself. This can include PCIe read completions errors, and AXI Bresponse errors (H2C), or AXI Rresponse errors and PCIe write errors due to bus master enable or function level reset (FLR), as well as RAM ECC errors. The first enabled error is logged in the DMA engine. Please refer to the Memory Mapped Engine error logs. If an error occurs on the read, the DMA write will be aborted if possible. If the error was detected when pulling write data from RAM, it is not possible to abort the request. Instead invalid data parity will be generated to ensure the destination is aware of the problem. Once the descriptor which encountered the error has gone through the DMA engine, it will proceed to generate status updates with an error indication. As with descriptor errors, it will result in the queue being invalidated. See Descriptor Engine Errors.

AXI4 Memory Mapped Descriptor for H2C and C2H (32B)

Table 8: **AXI4 Memory Mapped Descriptor Structure for H2C and C2H**

Bit	Bit Width	Field Name	Description
[255:192]	64	rsvd1	Reserved
[191:128]	64	dst_addr	Destination Address
[127:92]	36	rsvd0	Reserved
[91:64]	28	lengthInByte	Read length in byte
[63:0]	64	src_addr	Source Address

Internal mode memory mapped DMA must configure the descriptor queue to be 32B and follow the above descritor format. In bypass mode, the descriptor format is defined by the user logic, which must drive the H2C or C2H MM bypass input port.

AXI4 Memory Mapped Writeback Status Structure for H2C and C2H

The MM writeback status register is located after the last entry of the (H2C or C2H) descriptor.

Table 9: **AXI4 Memory Mapped Writeback Status Structure for H2C and C2H**

Bit	Bit Width	Field Name	Description
[63:48]	16	rsvd1	Reserved
[47:32]	16	pidx	Producer Index at time of writeback
[31:16]	16	cidx	Consumer Index
[15:2]	14	rsvd0	Reserved
[1:0]	2	err	Error bit 1: Descriptor fetch error bit 0: DMA error

Stream Mode DMA

H2C Stream Engine

The H2C Stream Engine is responsible for transferring streaming data from the host and deliver it to the user logic. The H2C Stream Engine operates on H2C stream descriptors. Each descriptor specifies the start address and the length of the data to be transferred to the user logic. The H2C Stream Engine parses the descriptor and issues read requests to the host over PCIe, splitting the read requests at MRRS boundary. There can be up to 256 requests outstanding in the H2C Stream Engine to hide the host read latency. The H2C Stream Engine implements a re-ordering buffer of 32 KB to re-order the TLPs as they come back. Data is issued to the user logic in order of the requests sent to PCIe.

If the status descriptor is enabled in the associated H2C context, the engine could additionally send a status write back to host once it is done issuing data to the user logic.

Internal and Bypass Modes

Each queue in QDMA Subsystem for PCIe can be programmed in either of the two H2C Stream modes: internal and bypass. This is done by specifying the mode in the queue context. The H2C Stream Engine knows whether the descriptor being processed is for a queue in internal or bypass mode.

The following figures show the internal mode and bypass mode flows.

Figure 7: **H2C Internal Mode Flow**

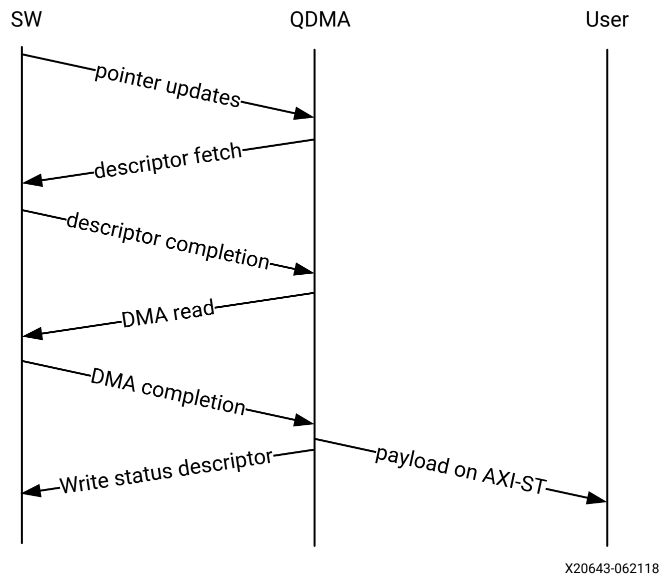
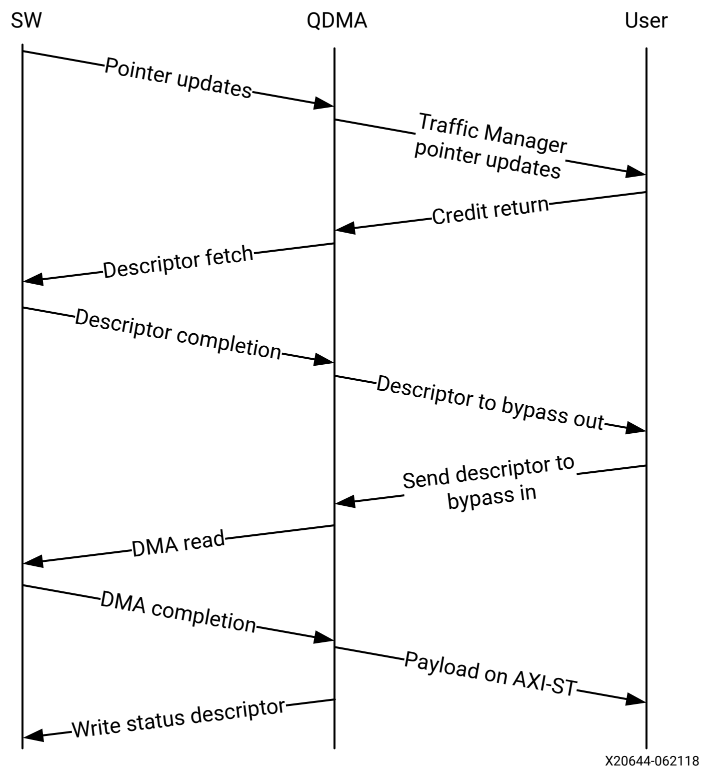


Figure 8: **H2C Bypass Mode Flow**



For a queue in internal mode, after the descriptor is fetched from the host, it is fed straight to the H2C Stream Engine for processing. In this case, a packet of data cannot span over multiple descriptors. Thus for a queue in internal mode, each descriptor generates exactly one AXI4-Stream packet on the QDMA H2C AXI Stream output. If the packet is present in host memory in non-contiguous space, then it has to be defined by more than one descriptor and this requires that the queue be programmed in bypass mode.

In the bypass mode, after the descriptors are fetched from the host, they are sent straight to the user logic via the QDMA bypass output port. The QDMA does not parse these descriptors at all. The user logic can store these descriptors and then send the required information from these descriptors back to QDMA using the QDMA H2C Stream descriptor bypass-in interface. Using this information, the QDMA constructs descriptors which are then fed to the H2C Stream Engine for processing. The following are the advantages of using the bypass mode:

- The user logic can have a custom descriptor format. This is possible because QDMA Subsystem for PCIe does not parse descriptors for queues in bypass mode. The user logic parses these descriptors and provides the information required by the QDMA on the H2C Stream bypass-in interface.
- Immediate data can be passed from the software to the user logic without DMA operation.
- The user logic can do traffic management by sending the descriptors to the QDMA when it is ready to sink all the data. Descriptors can be cached in local RAM.

- Perform address translation.

There are some requirements imposed on the user logic when using the bypass mode. Because the bypass mode allows a packet to span multiple descriptors, the user logic needs to indicate to QDMA which descriptor marks at the Start-Of-Packet (SOP) and which marks the End-Of-Packet (EOP). At the QDMA H2C Stream bypass-in interface, among other pieces of information, the user logic needs to provide: Address, Length, SOP, and EOP. It is required that once the user logic feeds an SOP descriptor information into QDMA, it must eventually feed an EOP descriptor information also. Descriptors for these multi-descriptor packets must be fed in sequentially. Other descriptors not belonging to the packet must not be interleaved within the multi-descriptor packet. The user logic must accumulate the descriptors up to the EOP descriptor, before feeding them back to QDMA. Not doing so can result in a hang. The QDMA will generate a TLAST at the QDMA H2C AXI Stream data output once it issues the the last beat for the EOP descriptor. This is guaranteed because the user is required to submit the descriptors for a given packet sequentially.

The H2C stream interface is shared by all the queues, it has the potential for head of the line blocking issue if the user logic does not reserve the space to sink the packet. Quality of service can be severely affected if the packet sizes are large. The Stream engine is designed to saturate PCIe for packet sizes as low as 128B, so Xilinx recommends that you restrict the packet size to be host page size or maximum transfer unit as required by the user application.

A performance control provided in the H2C Stream Engine is the ability to stall requests from being issued to the PCIe RQ/RC if a certain amount of data is outstanding on the PCIe side as seen by the H2C Stream Engine. To use this feature, you must program a threshold value in the H2C_DATA_THRESH (0xE24) register. After the H2C Stream Engine has more data outstanding to be delivered to the user logic than this threshold, it stops sending further read requests to the PCIe RQ/RC. This feature is disabled by default and can be enabled with the H2C_DATA_THRESH (0xE24) register. This feature helps improve the C2H Stream performance, because the H2C Stream Engine can make requests at a much faster rate than the C2H Stream Engine. This can potentially use up the PCIe side resources for H2C traffic which results in C2H traffic suffering.

H2C Stream Descriptor (16B)

Table 10: **H2C Descriptor Structure**

Bit	Bit Width	Field Name	Description
[127:96]	32	addr_h	Address High. Higher 32 bits of the source address in Host
[95:64]	32	addr_l	Address Low. Lower 32 bits of the source address in Host
[63:48]	16	rsv	Reserved

Table 10: **H2C Descriptor Structure** *(cont'd)*

Bit	Bit Width	Field Name	Description
[47:32]	16	len	Packet Length. Length of the data to be fetched for this descriptor. This is also the packet length since in internal mode, a packet cannot span multiple descriptors. The maximum length of the packet can be 64K-1 bytes.
[31:0]	32	metadata	Metadata. QDMA passes this field on the H2C-ST TUSER along with the data on every beat. For a queue in internal mode, it can be used to pass messages from SW to user logic along with the data.

This H2C descriptor format is only applicable for internal mode. For bypass mode, the user logic can define its own format as needed by the user application.

Descriptor Metadata

Similar to bypass mode, the internal mode also provides a mechanism to pass information directly from the software to the user logic. In addition to address and length, the H2C Stream descriptor also has a 32b metadata field. This field is not used by the QDMA Subsystem for PCIe for the DMA operation. Instead, it is passed on to the user logic on the H2C AXI4-Stream `tuser` on every beat of the packet. Passing metadata on the `tuser` is not supported for a queue in bypass mode and consequently there is no input to provide the metadata on the QDMA H2C Stream bypass-in interface.

Zero Length Descriptor

The length field in a descriptor can be zero. In this case, the H2C Stream Engine will issue a zero byte read request on PCIe. After the QDMA receives the completion for the request, the H2C Stream Engine will send out one beat of data with `tlast` on the QDMA H2C AXI Stream interface. The zero byte packet will be indicated on the interface by setting the `zero_b_dma` bit in the `tuser`. The user logic must set both the SOP and EOP for a zero byte descriptor. If not done, an error will be flagged by the H2C Stream Engine.

H2C Stream Status Descriptor Writeback

When feeding the descriptor information on the bypass input interface, the user logic can request the QDMA Subsystem for PCIe to send a status write back to the host when it is done fetching the data from the host. The user logic can also request that a status be issued to it when the DMA is done. These behaviors can be controlled using the `sdi` and `mrkr_req` inputs in the bypass input interface. See QDMA Descriptor Bypass Input Ports for details.

AXI4-Stream H2C Writeback Status Structure

The H2C writeback status register is located after the last entry of the H2C descriptor list.

Table 11: **AXI4-Stream H2C Writeback Status Descriptor Structure**

Bit	Bit Width	Field Name	Description
[63:32]	32	Reserved1	Reserved
[31:16]	16	cidx	Consumer Index
[15:0]	16	Reserved0	Producer Index (Reserved)

H2C Stream Data Aligner

The H2C engine has a data aligner that aligns the data to zero Bytes (0B) boundary before issuing it to the user logic. This allows the start address of a descriptor to be arbitrarily aligned and still receive the data on the H2C AXI Stream data bus without any holes at the beginning of the data. The user logic can send a batch of descriptors from SOP to EOP with arbitrary address and length alignments for each descriptor. The aligner will align and pack the data from the different descriptors and will issue a continuous stream of data on the H2C AXI Stream data bus. The `tlast` on that interface will be asserted when the last beat for the EOP descriptor is being issued.

Handling Descriptors With Errors

If an error is encountered while fetching a descriptor, the QDMA Descriptor Engine flags the descriptor with error. For a queue in internal mode, the H2C Stream Engine handles the error descriptor by not performing any PCIe or DMA activity. Instead, it waits for the error descriptor to pass through the pipeline and forces a writeback after it is done. For a queue in bypass mode, it is the responsibility of the user logic to not issue a batch of descriptors with an error descriptor. Instead, it must send just one descriptor with error input asserted on the H2C Stream bypass-in interface and set the SOP, EOP, `no_dma` signal, and `sdi` or `mrkr-req` signal to make the H2C Stream Engine send a writeback to Host.

Handling Errors in Data From PCIe

If the H2C Stream Engine encounters an error coming from PCIe on the data, it keeps the error sticky across the full packet. The error is indicated to the user on the `err` bit on the H2C Stream Data Output. Once the H2C Stream sends out the last beat of a packet that saw a PCIe data error, it also sends a Writeback to the Software to inform it about the error.

C2H Stream Engine

The C2H Stream Engine DMA writes the stream packets to the host memory into the descriptor provided by the host driver through the C2H descriptor queue.

The Prefetch Engine is responsible for calculating the number of descriptors needed for the DMA that is writing the packet. The buffer size is fixed per queue basis. For internal and cached bypass mode, the prefetch module can fetch up to 512 descriptors for a maximum of 64 different queues at any given time.

The Prefetch Engine also offers low latency feature `pfch_en=1`, where the engine can prefetch up to `num_pfch` descriptors upon receiving the packet, so that subsequent packets can avoid the PCIe latency.

C2H Stream Descriptor (8B)

Table 12: **AXI4-Stream C2H Descriptor Structure**

Bit	Bit Width	Field Name	Description
[63:0]	64	addr	Destination Address

C2H Prefetch Engine

The prefetch engine interacts between the descriptor fetch engine and C2H DMA write engine to pair up the descriptor and its payload.

Table 13: **C2H Prefetch Context Structure**

Bit	Bit Width	Field Name	Description
[45]	1	valid	Context is valid
[44:29]	16	sw_crdt	Software credit This field is written by the HW for internal use. The SW must initialize it to 0 and then treat it as read-only.
[28]	1	pfch	Queue is in prefetch This field is written by the HW for internal use. The SW must initialize it to 0 and then treat it as read-only.
[27]	1	pfch_en	Enable prefetch
[26]	1	err	Error detected on this queue
[25:8]	18	rsv	Reserved
[7:5]	3	port_id	Port ID
[4:1]	4	buf_size_idx	Buffer size index
[0]	1	bypass	C2H is in bypass mode

C2H Stream Modes

The C2H descriptors can be from the descriptor fetch engine or C2H bypass input interfaces. The descriptors from the descriptor fetch engine are always in cache mode. The prefetch engine keeps the order of the descriptors to pair with the C2H data packets from the user. The descriptors from the C2H bypass input interfaces have one interface for the simple mode, and another interface for the cache mode. For the simple mode, the user application keeps the order of the descriptors to pair with the C2H data packets. For the cache mode, the prefetch engine keeps the order of the descriptors to pair with the C2H data packet from the user.

The prefetch context has a bypass bit. When it is 1’b1, the user application sends the credits for the descriptors. When it is 1’b0, the prefetch engine handles the credits for the descriptors.

The descriptor context has a bypass bit. When it is 1’b1, the descriptor fetch engine sends out the descriptors on the C2H bypass output interface. The user application can convert it and later loop it back to the QDMA on the C2H bypass input interface. When the bypass context bit is 1’b0, the descriptor fetch engine sends the descriptors to the prefetch engine directly.

On a per queue basis, three cases are supported.

Table 14: **C2H Stream Modes**

	c2h_byp_in	desc_ctxt.desc_byp	pfch_ctxt.bypass
Simple bypass mode	simple byp in	1	1
Cache bypass mode	cache byp in	1	0
Cache internal mode	N/A	0	0

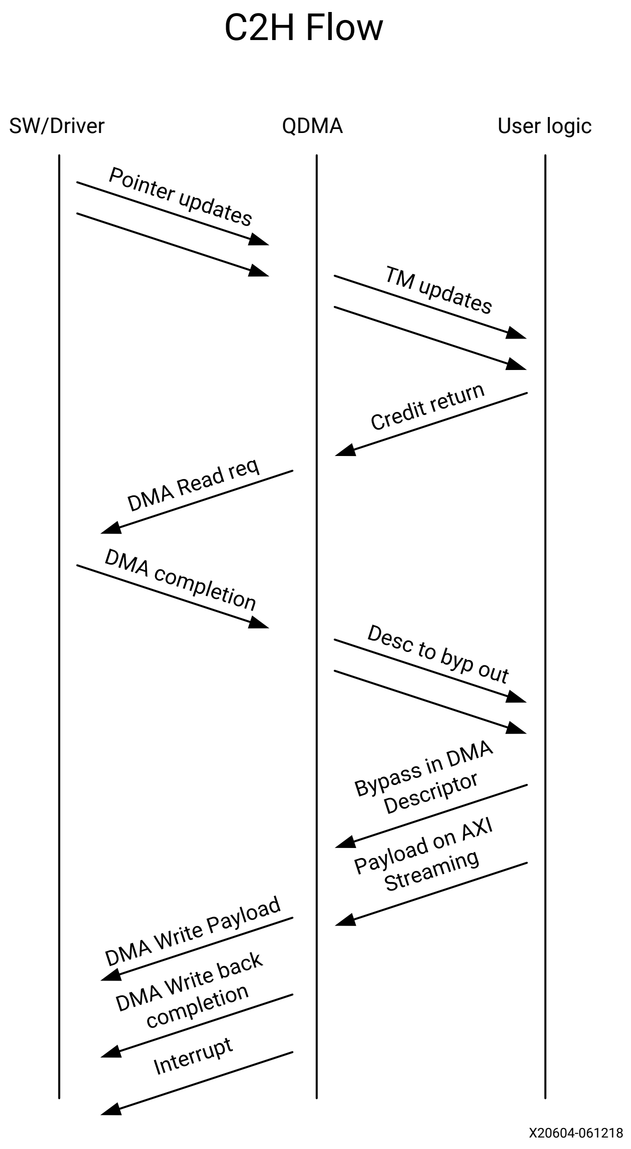
For simple bypass mode, the descriptor fetch engine sends the descriptors out on the C2H bypass out interface. The user application converts the descriptor and loops it back to the QDMA on the simple mode C2H bypass input interface. The user application sends the credits for the descriptors, and it also keeps the order of the descriptors.

For cache bypass mode, the descriptor fetch engine sends the descriptors out on the C2H bypass output interface. The user application converts the descriptor and loops it back to the QDMA on the cache mode C2H bypass input interface. The prefetch engine sends the credits for the descriptors, and it keeps the order of the descriptors.

For cache internal mode, the descriptor fetch engine sends the descriptors to the prefetch engine. The prefetch engine sends out the credits for the descriptors and keeps the order of the descriptors. In this case, the descriptors do not go out on the C2H bypass output and do not come back on the C2H bypass input interfaces.

The C2H descriptor bypass flow is as shown below.

Figure 9: **C2H Descriptor Bypass Flow**



For port descriptions, see QDMA Descriptor Bypass Input Ports and QDMA Descriptor Bypass Output Ports.

C2H Stream Packet Type

The following are some of the different C2H stream packets.

Regular data packet

The regular C2H data packet can be multiple bits.

- s_axis_c2h_ctrl_qid = qid of the packet

- s_axis_c2h_ctrl_len = length of the packet
- s_axis_c2h_mty = empty byte in the beat

Immediate data packet

The user logic can mark a data packet as 'immediate' to write to just the Completion ring without having a corresponding data packet transfer to the host. For the immediate data packet, the QDMA will not send the data payload, but it will write to the CMPT Queue. The immediate packet does not consume a descriptor.

The following is the setting of the immediate data packet:

- 1 beat of data
- s_axis_c2h_ctrl_imm_data = 1'b1
- s_axis_c2h_ctrl_len = datapath width in bytes (i.e., 64 if datawidth is 512 bits)
- s_axis_c2h_mty = 0

Marker packet

The C2H Stream Engine of the QDMA provides a way for the user logic to insert a marker into the QDMA along with a C2H packet. This marker then propagates through the C2H Engine pipeline and comes out on the C2H Stream Descriptor Bypass Out interface. The marker is inserted by setting the `marker` bit in the C2H Stream Control input. The marker response is indicated by QDMA to the user logic by setting the `mrkr_rsp` bit on the C2H Stream Descriptor Bypass Out interface. For a maker, QDMA does not send out a payload packet but still writes to the Completion Ring. Not all marker responses are generated because of a corresponding marker request. The QDMA sometimes generates marker responses when it encounters exceptional events. See the following section for details about when QDMA internally generates marker responses.

The primary purpose of giving the user logic the ability of sending in a marker into QDMA is to determine when all the traffic prior to the marker has been flushed. This can be used in the shutdown sequence in the user logic. Although not a requirement, the marker must be sent by the user logic with the `user_trig` bit set when sending in the marker into QDMA. This allows the QDMA to generate an interrupt and truly ensures that all traffic prior to the marker is flushed out. The QDMA Completion Engine takes the following actions when it receives a marker from the user logic:

- Sends the Completion that came along with the marker to the C2H Stream Completion Ring
- Generates Status Descriptor if enabled (if user_trig was set when maker was inserted)
- Generates an Interrupt if enabled and not outstanding

- Sends the marker response. If an Interrupt was not sent due to it being enabled but outstanding, the 'retry_mrkr' bit in the marker response is set to inform the user that an Interrupt could not be sent for this marker request. See the C2H Stream Descriptor Bypass Output interface description for details of these fields.

The following is the setting of the marker data packet:

- 1 beat of data
- s_axis_c2h_ctrl_marker = 1'b1
- s_axis_c2h_ctrl_len = data width (ex. 64 if data width is 512 bits)
- s_axis_c2h_mty = 0

The immediate data packet and the marker packet don't consume the descriptor, but they write to the C2H Completion Ring. The software needs to size the C2H Completion Ring large enough to accommodate the outstanding immediate packets and the marker packets.

Zero length packet

The length of the data packet can be 0. On the input, the user needs to send 1 beat of data. The zero length packet consumes the descriptor. The QDMA will send out 1DW payload data.

The following is the setting of the zero length packet:

- 1 beat of data
- s_axis_c2h_ctrl_len = 0
- s_axis_c2h_mty = 0

Disable completion packet

The user can disable the completion for a specific packet. The QDMA will DMA the payload, but it will not write to the C2H Completion Ring.

The following is the setting of the disable completion packet:

- s_axis_c2h_ctrl_disable_cmp = 1

C2H Completion

When the DMA write of the data payload is done, the QDMA writes the CMPT packet into the CMPT queue. Besides the user defined data, the CMPT packet also includes some other information, such as error, color, and the length. It also has a `desc_used` bit to indicate if the packet consumes a descriptor. A C2H data packet of immediate-data or marker type does not consume any descriptor.

C2H Completion Context Structure

The completion context is used by the completion engine.

Table 15: **C2H Completion Context Structure Defintion**

Bit	Bit Width	Field Name	Description
[127:126]	2	rsvd	Reserved
[125]	1	full_upd	Full Update If reset, then the Completion-CIDX-update is allowed to update only the CIDX in this context. If set, then the Completion CIDX update can update the following fields in this context: timer_ix counter_ix trig_mode en_int en_stat_desc
[124]	1	timer_running	If set, it indicates that a timer is running on this queue. This timer is for the purpose of C2H interrupt moderation. Ideally, the software must ensure that there is no running timer on this QID before shutting the queue down. This is a field used internally by HW. The SW must initialize it to 0 and then treat it as read-only.
[123]	1	user_trig_pend	If set, it indicates that a user logic initiated interrupt is pending to be generated. The user logic can request an interrupt through the s_axis_c2h_ctrl_user_trig signal. This bit is set when the user logic requests an interrupt while another one is already pending on this QID. When the next Completion CIDX update is received by QDMA, this pending bit may or may not generate an interrupt depending on whether or not there are entries in the Completion ring waiting to be read. This is a field used internally by HW. The SW must initialize it to 0 and then treat it as read-only.
[122:121]	2	err	Indicates that the C2H Completion Context is in error. This is a field written by HW. The SW must initialize it to 0 and then treat it as read-only. The following errors are indicated here: 0: No error. 1: A bad CIDX update from software was detected. 2: A descriptor error was detected. 3: A Completion packet was sent by the user logic when the Completion Ring was already full.
[120]	1	valid	Context is valid.
[119:104]	16	cidx	Current value of the hardware copy of the Completion Ring Consumer Index.
[103:88]	16	pidx	Completion Ring Producer Index. This is a field written by HW. The SW must initialize it to 0 and then treat it as read-only.

Table 15: **C2H Completion Context Structure Defintion** *(cont'd)*

Bit	Bit Width	Field Name	Description
[87:86]	2	desc_size	Completion Entry Size: 0: 8B 1: 16B 2: 32B 3: Unknown
[85:28]	58	baddr_64	Base address of Completion ring – bit [63:6].
[27:24]	4	qsize_idx	Completion ring size index to ring size registers.
[23]	1	color	Color bit to be used on Completion.
[22:21]	2	int_st	Interrupt State: 0: ISR 1: TRIG This is a field used internally by HW. The SW must initialize it to 0 and then treat it as read-only. Because it is out of reset, the HW initializes into ISR state, it is not sensitive to trigger events. If SW desires interrupts or status writes, it must send an initial Completion CIDX update. This makes the HW move into TRIG state and as a result it becomes sensitive to any trigger conditions.
[20:17]	4	timer_idx	Index to timer register for TIMER based trigger modes.
[16:13]	4	counter_idx	Index to counter register for COUNT based trigger modes.
[12:5]	8	fnc_id	Function ID
[4:2]	3	trig_mode	Interrupt and Completion Status Write Trigger Mode: 0x0: Disabled 0x1: Every 0x2: User_Count 0x3: User 0x4: User_Timer 0x5: User_Timer_Count
[1]	1	en_int	Enable Completion interrupts.
[0]	1	en_stat_desc	Enable Completion Status writes.

C2H Completion Status Structure

The C2H completion status is located at the last location of completion ring, that is, Completion Ring Base Address + (Size of the completion length (8,16,32) * (Completion Ring Size – 1)).

When C2H Streaming Completion is enabled, after the packet is transferred, CMPT entry and CMPT status are written to C2H Completion ring. PIDX in the Completion status can be used to indicate the currently available completion to be processed.

Table 16: **AXI4-Stream C2H Completion Status Structure**

Bit	Bit Width	Field Name	Description
[63:35]	29	Reserve	Reserved

Table 16: **AXI4-Stream C2H Completion Status Structure** *(cont'd)*

Bit	Bit Width	Field Name	Description
[34:33]	2	int_state	Interrupt State. 0: ISR 1: TRIG
[32]	1	color	Color status bit
[31:16]	16	cidx	Consumer Index (RO)
[15:0]	16	pidx	Producer Index

C2H Completion Entry Structure

The following is the C2H Completion ring entry structure for User format when the data format bit is set to 1’b1.

Table 17: **C2H Completion Entry User Format Structure**

Name	Size	Index
User defined bits for 32 Bytes settings	252 bits	[255:4]
User defined bits for 16 Bytes settings	124 bits	[127:4]
User defined bits for 8 Bytes settings	60 bits	[63:4]
desc_used	1	[3:3]
err	1	[2:2]
color	1	[1:1]
Data format	1	[0:0]

The following is the C2H Completion ring entry structure for Standard format when the data format bit is set to 1’b0.

Table 18: **C2H Completion Entry Standard Format Structure**

Name	Size	Index
User defined bits for 32 Bytes settings	236 bits	[255:20]
User defined bits for 16 Bytes settings	108 bits	[127:20]
User defined bits for 8 Bytes settings	44 bits	[63:20]
Len	16	[19:4]
desc_used	1	[3:3]
err	1	[2:2]
color	1	[1:1]
Data format	1	[0:0]

C2H Completion Input Packet

The Completion Ring entry structure is shown in C2H Stream Engine.

When the user application sends the C2H data packet, it also sends the CMPT (completion) packet to the QDMA. The CMPT packet has two formats: Standard Format and User Format.

The following is the CMPT packet from the user application in the standard format, which is when the data format bit is 1'b0.

Table 19: **CMPT Packet in Standard Format**

Name	Size	Index
User defined	44 bits-236 bits	[255:20]
rsvd	8	[19:12]
Qid	11	[11:1]
Data format	1	[0:0]

The following is the CMPT packet from the user application in the user format, which is when the data format bit is 1'b1.

Table 20: **CMPT Packet in User Format**

Name	Size	Index
User defined	61 bits-253 bits	[255:3]
rsvd	2	[2:1]
Data format	1	[0:0]

The CMPT packet has three types: 8B, 16B, or 32B. When it is 8B or 16B, it only needs one pump of the data. When it is 32B, it needs two pumps of data. Each data pump is 128 bits.

C2H Interrupt/Completion Status Moderation

The QDMA Subsystem for PCIe provides a means to moderate the C2H completion interrupts and Completion status writes on a per queue basis. The software can select one out of five modes for each queue. The selected mode for a queue is stored in the QDMA Subsystem for PCIe in the C2H completion ring context for that queue. After a mode has been selected for a queue, the driver can always select another mode when it sends the completion ring CIDX update to QDMA.

The C2H completion interrupt moderation is handled by the completion engine inside the C2H engine. The completion engine stores the C2H completion ring contexts of all the queues. It is possible to individually enable or disable the sending of interrupts and C2H completion status descriptors for every queue and this information is present in the completion ring context. It is worth mentioning that the modes being described here moderate not only interrupts but also completion status writes. Also, since interrupts and completion status writes can be individually enabled/disabled for each queue, these modes will work only if the interrupt/completion status is enabled in the Completion context for that queue.

The QDMA Subsystem for PCIe keeps only one interrupt outstanding per queue. This policy is enforced by QDMA even if all other conditions to send an interrupt have been met for the mode. The way the QDMA Subsystem for PCIe considers an interrupt serviced is by receiving a CIDX update for that queue from the driver.

The basic policy followed in all the interrupt moderation modes is that when there is no interrupt outstanding for a queue, the QDMA Subsystem for PCIe keeps monitoring the trigger conditions to be met for that mode. Once the conditions are met, an interrupt is sent out. While the QDMA subsystem is waiting for the interrupt to be served, it remains sensitive to interrupt conditions being met and remembers them. When the CIDX update is received, the QDMA subsystem evaluates whether the conditions are still being met. If they are still being met, another interrupt is sent out. If they are not met, no interrupt is sent out and QDMA resumes monitoring for the conditions to be met again.

Note that the interrupt moderation modes that the QDMA subsystem provides are not necessarily precise. Thus, if the user application sends two C2H packets with an indication to send an interrupt, it is not necessary that two interrupts will be generated. The main reason for this behavior is that when the driver is interrupted to read the completion ring, and it is under no obligation to read exactly up to the completion for which the interrupt was generated. Thus, the driver may not read up to the interrupting completion descriptor, or it may even read beyond the interrupting completion descriptor if there are valid descriptors to be read there. This behavior requires the QDMA Subsystem for PCIe to re-evaluate the trigger conditions every time it receives the CIDX update from the driver.

The detailed description of each mode is given below:

- **TRIGGER_EVERY:** This mode is the most aggressive in terms of interruption frequency. The idea behind this mode is to send an interrupt whenever the completion engine determines that an unread completion descriptor is present in the completion ring.

- **TRIGGER_USER:** The QDMA Subsystem for PCIe provides a way to send a C2H packet to the subsystem with an indication to send out an interrupt when the subsystem is done sending the packet to the host. This allows the user application to perform interrupt moderation when the TRIGGER_USER mode is set.
- **TRIGGER_USER_COUNT:** This mode allows the QDMA Subsystem for PCIe is sensitive to either of two triggers. One of these triggers is sent by the user along with the C2H packet. The other trigger is the presence of more than a programmed threshold of unread Completion entries in the Completion Ring, as seen by the HW. This threshold is driver programmable on a per-queue basis. The QDMA evaluates whether or not to send an interrupt when either of these triggers is detected. As explained in the preceding sections, other conditions must be satisfied in addition to the triggers for an interrupt to be sent.
- **TRIGGER_USER_TIMER:** In this mode, the QDMA Subsystem for PCIe is sensitive to either of two triggers. One of these triggers is sent by the user along with the C2H packet. The other trigger is the expiration of the timer that is associated with the C2H queue. The period of the timer is driver programmable on a per-queue basis. The QDMA evaluates whether or not to send an interrupt when either of these triggers is detected. As explained in the preceding sections, other conditions must be satisfied in addition to the triggers for an interrupt to be sent. For more information, see C2H Timer.
- **TRIGGER_USER_TIMER_COUNT:** This mode allows the QDMA Subsystem for PCIe is sensitive to any of three triggers. One of these triggers is sent by the user along with the C2H packet. The second trigger is the expiration of the timer that is associated with the C2H queue. The period of the timer is driver programmable on a per-queue basis. The third trigger is the presence of more than a programmed threshold of unread Completion entries in the Completion Ring, as seen by the HW. This threshold is driver programmable on a per-queue basis. The QDMA evaluates whether or not to send an interrupt when any of these triggers is detected. As explained in the preceding sections, other conditions must be satisfied in addition to the triggers for an interrupt to be sent.
- **TRIGGER_DIS:** In this mode, the QDMA Subsystem for PCIe does not send C2H completion interrupts in spite of them being enabled for a given queue. The only way that the driver can read the completion ring in this case is when it regularly polls the ring. The driver will have to make use of the color bit feature provided in the completion ring when this mode is set as this mode also disables the sending of any completion status descriptors to the completion ring.

The following are the flowcharts of different modes. These flowcharts are from the point of view of the C2H Completion Engine. The Completion packets come in from the user logic and are written to the Completion Ring. The software (SW) update refers to the Completion Ring CIDX update sent from software to hardware.

Figure 10: **Flowchart for EVERY Mode**

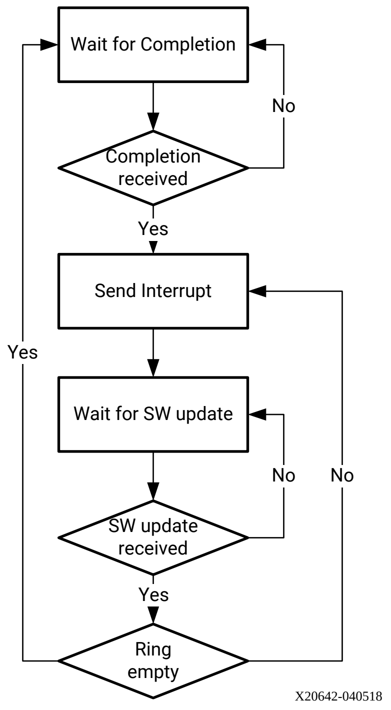


Figure 11: **Flowchart for USER Mode**

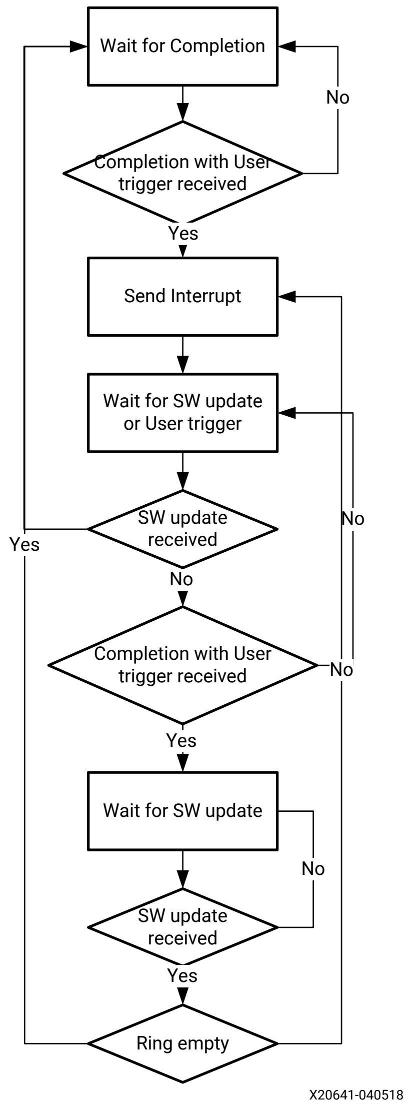


Figure 12: **Flowchart for USER_COUNT Mode**

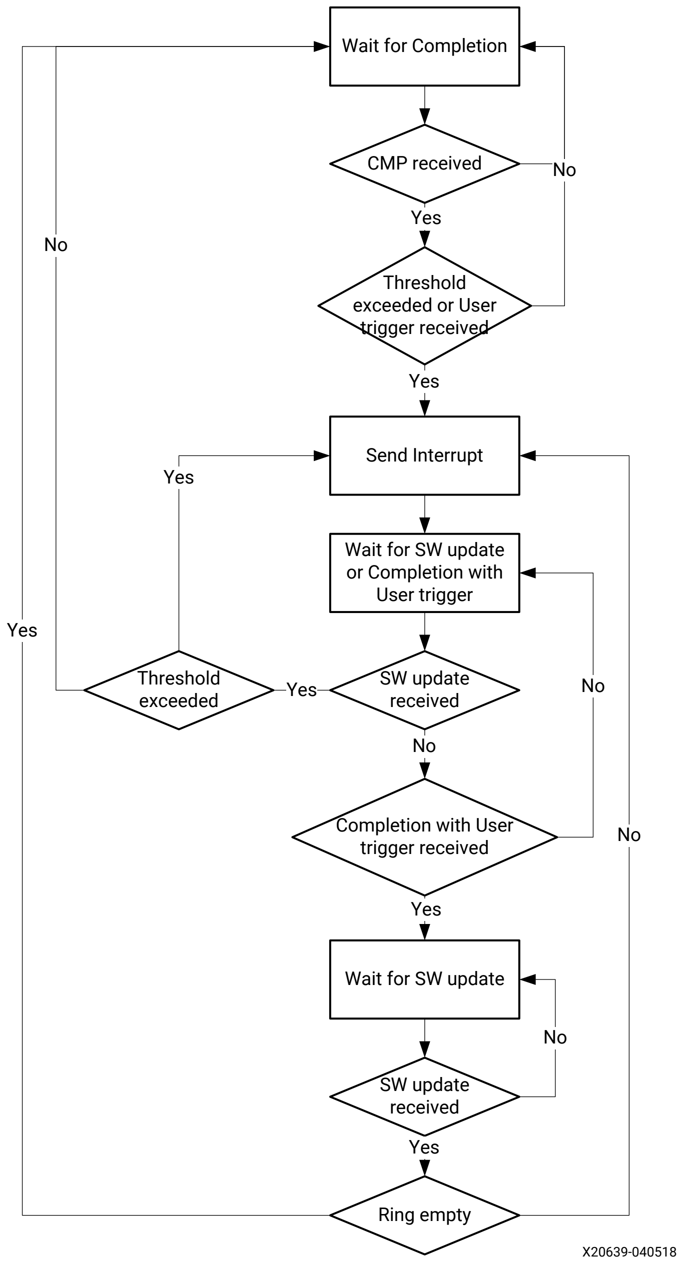
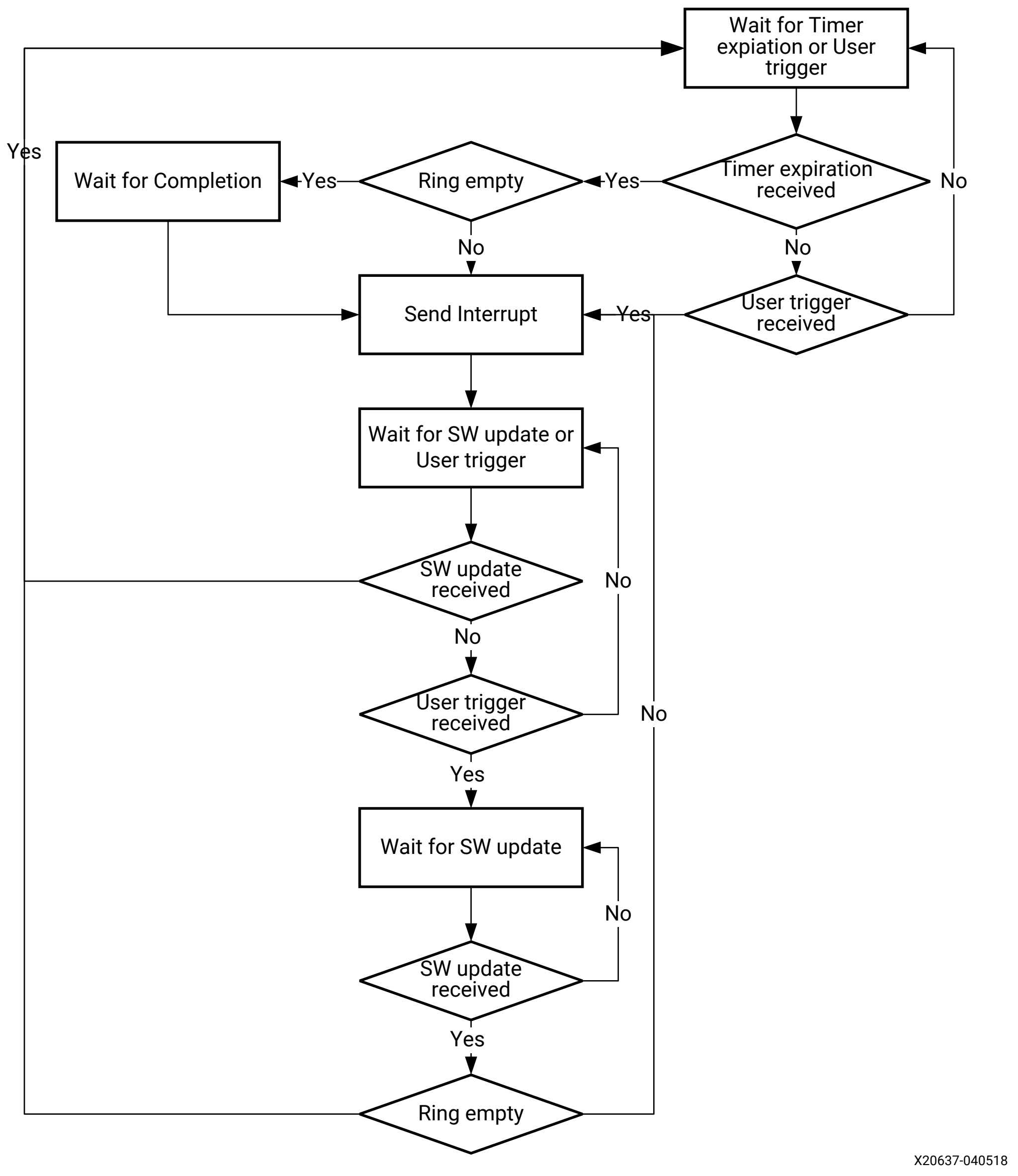


Figure 13: **Flowchart for USER_TIMER Mode**



C2H Timer

The C2H timer is a trigger mode in the Completion context. It supports 2048 queues, and each queue has its own timer. When the timer expires, a timer expire signal is sent to the Completion module. If multiple timers expire at the same time, then they are sent out in a round robin manner.

Reference Timer

The reference timer is based on the timer tick. The register QDMA_C2H_INT (0xB0C) defines the value for a timer tick. The 16registers QDMA_C2H_TIMER_CNT (0xA00-0xA3c) has the timer counts based on the timer tick. The timer_idx in the Completion context is the index to the 16 QDMA_C2H_TIMER_CNT registers. Each queue can choose its own timer_idx.

Handling Exception Events

C2H Completion On Invalid Queue

When QDMA receives a Completion on a queue which has an invalid context as indicated by the Valid bit in the C2H CMPT Context, the Completion is silently dropped.

C2H Completion On A Full Ring

The maximum number of Completion entries in the Completion Ring is 2 less than the total number of entries in the Completion Ring. The C2H Completion Context has PIDX and CIDX in it. This allows the QDMA to calculate the number of Completions in the Completion Ring. When the QDMA receives a Completion on a queue that is full, QDMA takes the following actions:

- Invalidates the C2H Completion Context for that queue.
- Marks the C2H Completion Context with error.
- Drops the Completion.
- If enabled, sends a Status Descriptor marked with error.
- If enabled and not outstanding, sends an Interrupt.
- Sends a Marker Response with error.
- Logs the error in the C2H Error Status Register.

C2H Completion With Descriptor Error

When the QDMA C2H Engine encounters a Descriptor Error, the following actions are taken in the context of the C2H Completion Engine:

- Invalidates the C2H Completion Context for that queue.
- Marks the C2H Completion Context with error.
- Sends the Completion out to the Completion Ring. It is marked with an error.
- If enabled, sends a Status Descriptor marked with error.
- If enabled and not outstanding, sends an Interrupt.
- Sends a Marker Response with error.

C2H Completion With Invalid CIDX

The C2H Completion Engine has logic to detect that the CIDX value in the CIDX update points to an empty location in the Completion Ring. When it detects such error, the C2H Completion Engine:

- Invalidates the Completion Context.
- Marks the Completion Context with error.
- Logs an error in the C2H error status register.

Bridge

The Bridge core is an interface between the AXI4 and PCI Express. It contains the memory mapped AXI4 to AXI4-Stream Bridge and the AXI4-Stream Enhanced Interface Block for PCIe. The memory-mapped AXI4 to AXI4-Stream Bridge contains a register block and two functional half bridges, referred to as the Slave Bridge and Master Bridge. The slave bridge connects to the AXI4 Interconnect as a slave device to handle any issued AXI4 master read or write requests. The master bridge connects to the AXI4 Interconnect as a master to process the PCIe generated read or write TLPs. The core uses a set of interrupts to detect and flag error conditions.

The Register block contains registers used in the Bridge core for dynamically mapping the AXI4 memory mapped (MM) address range provided using the AXIBAR parameters to an address for PCIe® range.

The slave bridge provides termination of memory-mapped AXI4 transactions from an AXI master device (such as a processor). The slave bridge provides a way to translate addresses that are mapped within the AXI4 memory mapped address domain to the domain addresses for PCIe.Write transactions to the Slave Bridge are converted into one or more MemWr TLPs, depending on the configured Max Payload Size setting, which are passed to the integrated block for PCI Express. When a remote AXI master initiates a read transaction to the slave bridge, the read address and qualifiers are captured and a MemRd request TLP is passed to the core and a completion timeout timer is started. Completions received through the core are correlated with pending read requests and read data is returned to the AXI master. The Slave Bridge can support up to 32 AXI write requests, and 32 AXI read requests.

The master bridge processes both PCIe MemWr and MemRd request TLPs received from the Integrated Block for PCI Express and provides a means to translate addresses that are mapped within the address for PCIe domain to the memory mapped AXI4 address domain. Each PCIe MemWr request TLP header is used to create an address and qualifiers for the memory mapped AXI4 bus and the associated write data is passed to the addressed memory mapped AXI4 Slave. The Master Bridge can support up to 32 active PCIe MemWr request TLPs. PCIe MemWr request TLPs support is as follows:

- 4 for 64-bit AXI data width

- 8 for 128-bit AXI data width
- 16 for 256-bit AXI data width
- 32 for 512-bit AXI data width

Each PCIe MemRd request TLP header is used to create an address and qualifiers for the memory-mapped AXI4 bus. Read data is collected from the addressed memory mapped AXI4 slave and used to generate completion TLPs which are then passed to the integrated block for PCI Express. The Master Bridge in AXI Bridge mode can support up to 32 active PCIe MemRd request TLPs with pending completions for improved AXI4 pipelining performance.

Interrupts

The QDMA Subsystem for PCIe supports up to 2K total MSI-X vectors, with up to 8 MSI-X Vectors per function. Legacyinterrupts are not supported in QDMA. A single MSI-X vector can be used to support multiple queues.

The QDMA supports Interrupt Aggregation. Each vector has an associated Interrupt Aggregation Ring. The QID and status of queues requiring service are written into the Interrupt Aggregation Ring. When a PCIe MSI-X interrupt is received by the Host, the software reads the Interrupt Aggregation Ring to determine which queue needs service. Mapping of queues to vectors is programmable. It has independent table programming per physical function (PF). It supports MSI/MSI-X interrupt modes for non-SRIOV and MSI-X for SRIOV.

Asynchronous and Queue Based Interrupts

The QDMA supports both asynchronous interrupts and queue-based interrupts.

The asynchronous interrupts are used for capturing events that are not synchronous to any DMA operations, namely, errors, status, and debug conditions. There is one asynchronous interrupt per PF. Every asynchronous interrupt is configurable to any one of the PF.

In a queue based scheme, interrupts are broadcast to all PFs and maintain status for each PF while in the asynchronous internal scheme all asynchronous interrupts are configuredtoany one of the PF.

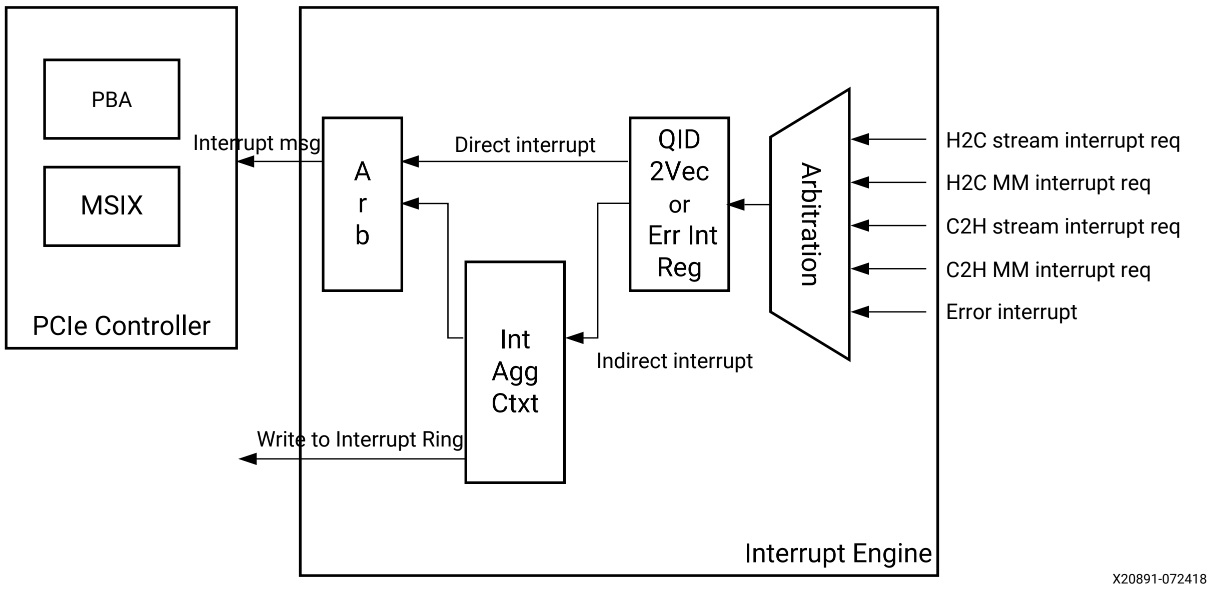
The queue based interrupts include the interrupts from the H2C MM, H2C stream, C2H MM, and C2H stream.

Interrupt Engine

The QDMA Subsystem for PCIe Interrupt Engine handles the queue based interrupts and the error interrupt.

This block diagram is of the Interrupt Engine.

Figure 14: **Interrupt Engine Block Diagram**



When the H2C or C2H interrupt occur, it first reads the QID to vector table. The table has 2K entries to support up to 2K queues. Each entry of the table includes two portions: one for H2C interrupts, and one for C2H interrupts. The table maps the QID to the vector, and indicates if the interrupt is direct interrupt mode or indirect interrupt mode. If it is direct interrupt mode, the vector is used to generate the PCIe MSI-X message. If it is indirect interrupt mode, the vector is the ring index, which is the index of the Interrupt Context for the Interrupt Aggregation Ring.

The following is the data in the QID to vector table.

Table 21: **QID to Vector Table**

Signal	Bit	Owner	Description
H2c_en_coal	[17:17]	Driver	1’b1: indirect interrupt mode. 1’b0: direct interrupt mode for H2C interrupt.
H2c_vector	[16:9]	Driver	For direct interrupt, it is the interrupt vector index of msix table; for indirect interrupt, it is the ring index.
C2h_en_coal	[8:8]	Driver	1’b1: indirect interrupt mode. 1’b0: direct interrupt mode for C2H interrupt.
C2h_vector	[7:0]	Driver	For direct interrupt, it is the interrupt vector index of msix table; for indirect interrupt, it is the ring index.

The QID to Vector table is programmed by the context access.

- Context access through QDMA_TRQ_SEL_IND:
    - QDMA_IND_CTXT_CMD.Qid = Qid
    - QDMA_IND_CTXT_CMD.Sel = MDMA_CTXT_SEL_INT_QID2VEC (0xC)
    - QDMA_IND_CTXT_CMD.Op = MDMA_CTXT_CMD_WR or MDMA_CTXT_CMD_RD (MDMA_CTXT_CMD_CLR and MDMA_CTXT_CMD_INV are not supported for Qid to Vector table)

Direct Interrupt

For direct interrupt, the QDMA processes the interrupt with the following steps.

- Look up the QID to Vector Table.
- Send out the PCIe MSI-X message.

Interrupt Aggregation Ring

For indirect interrupt, it does interrupt aggregation. The following are some restrictions for the interrupt aggregation.

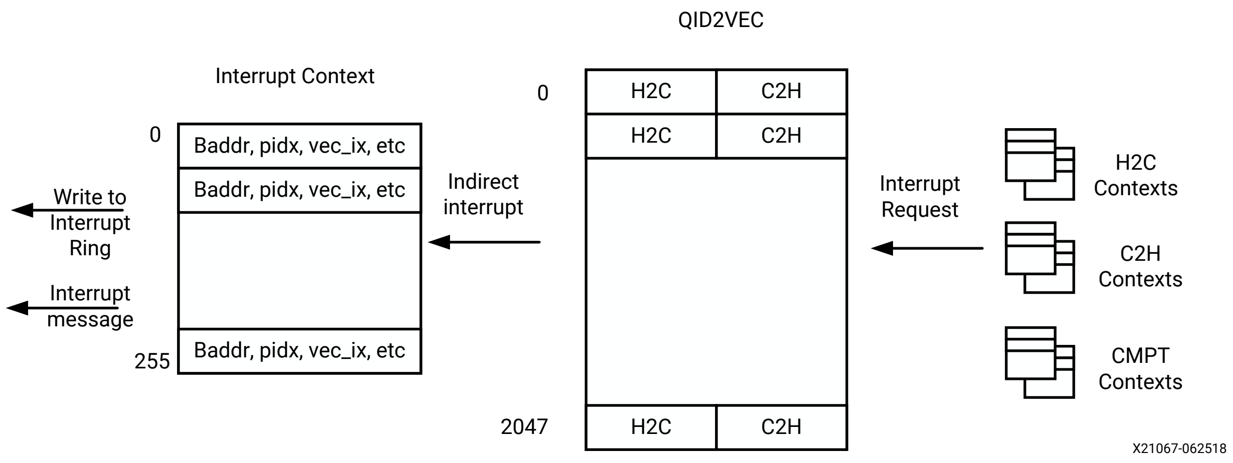
- Each Interrupt Aggregation Ring can only be associated with one function. But multiple rings can be associated with the same function.
- There are up to 3 messages in the entry per interrupt source.

In the indirect interrupt, the QDMA processes the interrupt with the following steps.

- Look up the QID to Vector Table.
- Look up the Interrupt Context.
- Write to the Interrupt Aggregation Ring.
- Send out the PCIe MSI-X message.

This is block diagram for the indirect interrupt.

Figure 15: **Indirect Interrupt**



The Interrupt Context includes the information of the Interrupt Aggregation Ring. It has 256 entries to support up to 256 Interrupt Aggregation Rings.

The following is the Interrupt Context Structure (0x8).

Table 22: **Interrupt Context Structure (0x8)**

Signal	Bit	Owner	Description
pidx	[75:64]	DMA	Producer Index
page_size	[63:61]	Driver	Interrupt Aggregation Ring size: 0: 4KB 1: 8KB 2: 12KB 3: 16KB 4: 20KB 5: 24KB 6: 28KB 7: 32KB
baddr_4k	[60:9]	Drive	Base address of Interrupt Aggregation Ring – bit[63:12]
color	[8]	DMA	Color bit
int_st	[7]	DMA	Interrupt State: 0: WAIT_TRIGGER 1: ISR_RUNNING
Rsvd	[6]	NA	Reserved
vec	[5:1]	Driver	Interrupt vector index in msix table
valid	[0]	Driver	Valid

The software needs to size the Interrupt Aggregation Ring appropriately. Each source can send up to three messages to the ring. Therefore, the size of the ring needs satisfy the following formula.

Number of entry >= 3 X (number of queues + error interrupts that are mapped to this ring)

The Interrupt Context is programmed by the context access. The QDMA_IND_CTXT_CMD.Qid has the ring index, which is from the Qid to Vector Table. The operation of MDMA_CTXT_CMD_CLR can clear all of the bits in the Interrupt Context. The MDMA_CTXT_CMD_INV can clear the valid bit.

- Context access through QDMA_TRQ_SEL_IND:
  - QDMA_IND_CTXT_CMD.Qid = Ring index
  - QDMA_IND_CTXT_CMD.Sel = MDMA_CTXT_SEL_INT_COAL (0x8)
  - QDMA_IND_CTXT_CMD.cmd.Op =

    MDMA_CTXT_CMD_WR,
    MDMA_CTXT_CMD_RD,
    MDMA_CTXT_CMD_CLR , or
    MDMA_CTXT_CMD_INV.

After it looks up the Interrupt Context, it then writes to the Interrupt Aggregation Ring. It also updates the Interrupt Context with the new pidx, color, and the interrupt state.

This is the Interrupt Aggregation Ring entry structure. It has 8B data.

Table 23: **Interrupt Aggregation Ring Entry Structure**

Signal	Bit	Owner	Description
Coal_color	[63:63]	DMA	The color bit of the Interrupt Aggregation Ring. This bit inverts every time pidx wraps around on the Interrupt Aggregation Ring.
Qid	[62:52]	DMA	This is from Interrupt source. Queue ID.
Int_type	[51:51]	DMA	0: H2C 1: C2H
Err_int	[50:50]	DMA	0: non-error interrupt 1: error interrupt
Rsvd	[49:39]	DMA	Reserved
Stat_desc	[38:0]	DMA	This is the status descriptor of the Interrupt source.

The following is the information in the stat_desc.

Table 24: **stat_desc Information**

Signal	Bit	Owner	Description
Error	[38:35]	DMA	This is from interrupt source: {c2h_err[1:0], h2c_err[1:0]}
Int_st	[34:33]	DMA	This is from Interrupt source. Interrupt state. 0: WRB_INT_ISR 1: WRB_INT_TRIG 2: WRB_INT_ARMED
Color	[32:32]	DMA	This is from Interrupt source. This bit inverts every time pidx wraps around and this field gets copied to color field of descriptor.
Cidx	[31:16]	DMA	This is from Interrupt source. Cumulative consumed pointer
Pidx	[15:0]	DMA	This is from Interrupt source. Cumulative pointer of total interrupt Aggregation Ring entry written

When the software allocates the memory space for the Interrupt Aggregation Ring, the coal_color starts with 1’b0. The software needs to initialize the color bit of the Interrupt Context to be 1’b1. When the hardware writes to the Interrupt Aggregation Ring, it reads color bit from the Interrupt Context, and writes it to the entry. When the ring wraps around, the hardware will flip the color bit in the Interrupt Context. In this way, when the software reads from the Interrupt Aggregation Ring, it will know which entries got written by the hardware by looking at the color bit.

The software reads the Interrupt Aggregation Ring to get the Qid, the int_type (H2C or C2H), and the err_int. From the Qid, the software can identify it the queue is stream or MM.

When the err_int is set, it is an error interrupt. The software can then check the error status register of the Central Error Aggregator QDMA_GLBL_ERR_STAT (0x248). The register shows the error source. The software can then read the error status register of the Leaf Error Aggregator of the corresponding error.

The stat_desc in the Interrupt Aggregation Ring is the status descriptor from the Interrupt source. When the status descriptor is disabled, the software can get the status descriptor information from the Interrupt Aggregation Ring.

The two cases are as follows:

- The interrupt source is C2H stream, then it is the status descriptor of the C2H Completion Ring. The software can read the pidx of the C2H Completion Ring.
- The interrupt source is others (H2C stream, H2C MM, C2H MM), then it is the status descriptor of that source. The software can read the cidx.

Finally, the QDMA Subsystem for PCIe sends out the PCIe MSI-X message using the interrupt vector from the Interrupt Context.

When the PCIe MSI-X interrupt is received by the Host, the software reads the Interrupt Aggregation Ring to determine which queue needs service. After the software reads the Interrupt Aggregation Ring, it will do a dynamic pointer update for the software CIDX to indicate the cumulative pointer that the software reads to. The software does the dynamic pointer update using the register QDMA_DMAP_SEL_INT_CIDX[2048] (0x6400). If the software CIDX is equal to the pidx, this will trigger a write to the Interrupt Context on the interrupt state of that queue. This is to indicate the QDMA that the software already reads all of the entries in the Interrupt Aggregation Ring. If the software CIDX is not equal to the PIDX, it will send outanother PCIe MSI-X message. Therefore, the software can read the Interrupt Aggregation Ring again. After that, the software can do a pointer update of the interrupt source ring. For example, if it is C2H stream interrupt, the software will update pointer of the interrupt source ring, which is the C2H Completion Ring.

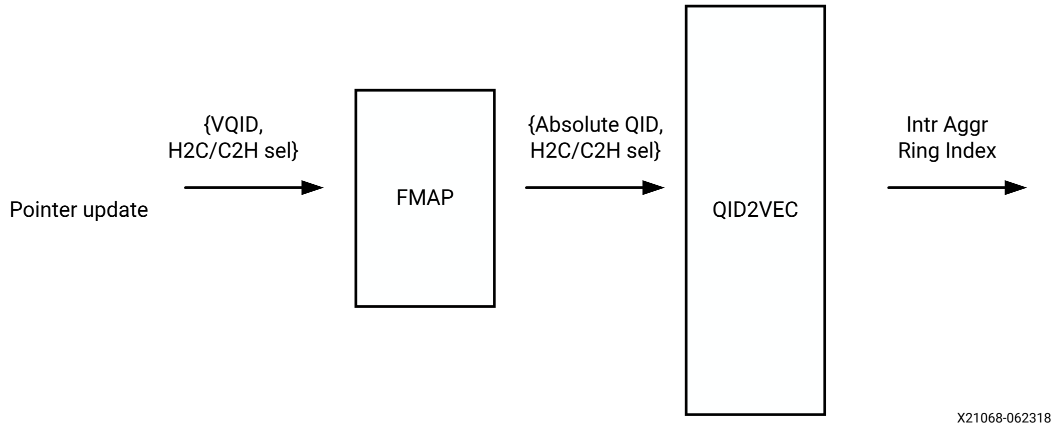
These are the steps for the software:

1. After the software gets the PCIE MSI-X message, it reads the Interrupt Aggregation Ring entries.
2. The software uses the `coal_color` bit to identify the written entries. Each entry has `Qid` and `Int_type` (H2C or C2H). From the `Qid` and `Int_type`, the software can check if it is stream or MM. This points to a corresponding source ring. For example, if it is C2H stream, the source ring is the C2H Completion Ring. The software can then read the source ring to get information, and do a dynamic pointer update of the source ring after that.
3. After the software finishes reading of all written entries, it does one dynamic point update of the software cidx using the register QDMA_DMAP_SEL_INT_CIDX[2048] (0x6400). The `Qid` in the register is the `Qid` in the last written entry. This tells hardware the pointer of the Interrupt Aggregation Ring that the software reads to.

   If the software cidx is not equal to the pidx, the hardware will send out another PCIE MSI-X message, so that the software can read the Interrupt Aggregation Ring again

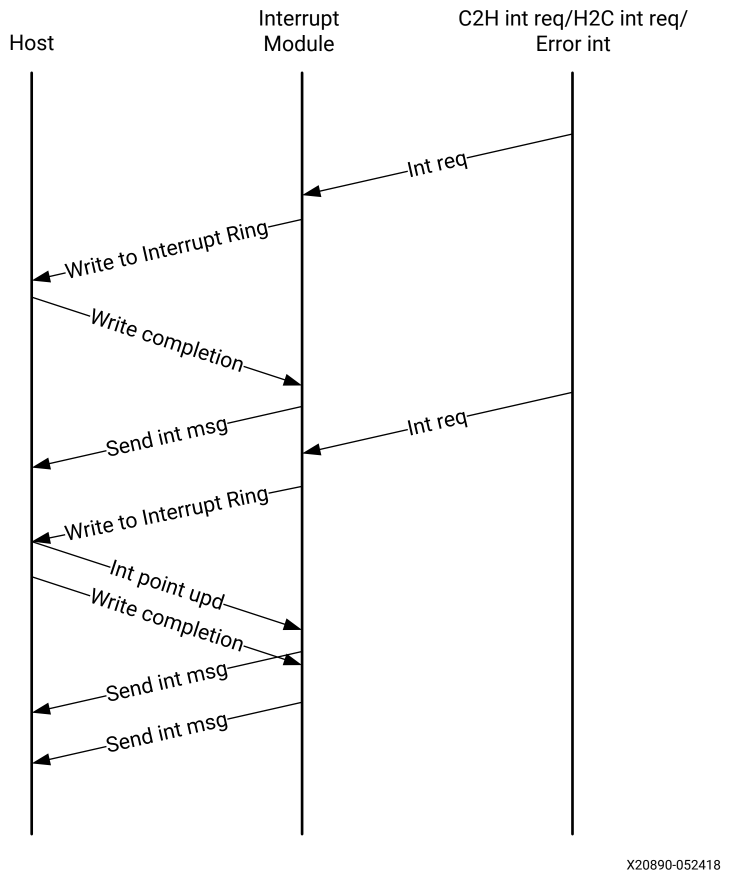
When the software does the dynamic point update for the Interrupt Aggregation Ring using the register QDMA_DMAP_SEL_INT_CIDX[2048] (0x6400), it needs to use the virtual Qid. The FMAP block in the hardware translates the virtual Qid to absolute Qid. The interrupt Engine uses the absolute Qid when it looks up the Qid to Vector Table.

Figure 16: **Interrupt Engine**



The following diagram shows the indirect interrupt flow. The Interrupt module gets the interrupt requests. It first writes to the Interrupt Aggregation Ring. Then it waits for the write completions. After that, it sends out the PCIe MSI-X message. The interrupt requests can keep on coming, and the Interrupt module keeps on processing them. In the meantime, the software reads the Interrupt Aggregation Ring and it does the dynamic pointer update. If the software CIDX is not equal to the PIDX, it will send out another PCIe MSI-X message.

Figure 17: **Interrupt Flow**



Error Interrupt

There are Leaf Error Aggregators in different places. They log the errors and propagate the errors to the Central Error Aggregator. Each Leaf Error Aggregator has an error status register and an error mask register. The error mask is enable mask. Only when the error mask is enabled, the Leaf Error Aggregator will propagate the error to the Central Error Aggregator.

TheCentral Error Aggregator aggregates all of the errors together. When any error occurs, it can generate an Error Interrupt if the err_int_arm bit is set in the error interrupt register QDMA_GLBL_ERR_INT (0B04). The err_int_arm bit is set by the software and cleared by the hardware when the Error Interrupt is taken by the Interrupt Engine. The Error Interrupt is for all of the errors including the H2C errors and C2H errors. The Software must set this `err_int_arm` bit[17] to generate interrupt again.

The Error Interrupt supports the direct interrupt only. Keep the `en_coal` bit unset in the error interrupt register QDMA_GLBL_ERR_INT. The interrupt aggregation entry write will be blocked in the case of fatal error such as parity and double bit ECC error causing the QDMA to hang without the software noticing it.

The Error Interrupt doesn't get the vector from the Qid to Vector table. It gets the vector from the error interrupt register QDMA_GLBL_ERR_INT. For the direct interrupt, the vector is the interrupt vector index of the msix table.

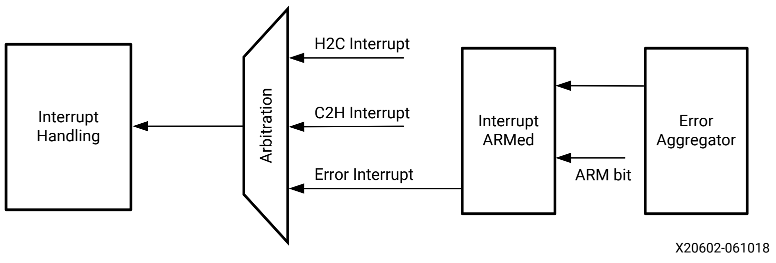
Here are the processes of the Error Interrupt.

1. Reads the Error Interrupt register QDMA_C2H_GLBL_INT (0B04).

   It supports the direct interrupt only.
2. Sends out the PCIe MSI-X message.

The following shows the block diagram of the error interrupt register.

Figure 18: **Error Interrupt Handling**



Queue Management

Function Map Table

The Function Map Table is used to allocate queues to each function. The index into the RAM is the function number. Each entry contains the base number of the physical QID and the number of queues allocated to the function. It provides a function based, queue access protection mechanism by translating and checking accesses to logical queues (through QDMA_TRQ_SEL_QUEUE_PF and QDMA_TRQ_SEL_QUEUE_VF address space) to their physical queues. Direct register accesses to queue space beyond what is allocated to the function in the table will be cancelled and an error will be logged.

The table can be programmed through the QDMA_TRQ_SEL_FMAP address space. Because this space only exists in the PF address map, only a physical function can modify this table.

Context Programming

- Program all four mask registers: QDMA_IND_CTXT_MASK_3 (0x814), QDMA_IND_CTXT_MASK_2 (0x818), QDMA_IND_CTXT_MASK_1 (0x81C), and QDMA_IND_CTXT_MASK_0 (0x820) to 1.
- Program context values to registers QDMA_IND_CTXT_DATA_3 (0x804), QDMA_IND_CTXT_DATA_2 (0x808), QDMA_IND_CTXT_DATA_1 (0x80C), and QDMA_IND_CTXT_DATA_0 (0x810).
- Refer to 'Software Descriptor Context Structure', 'C2H Prefetch Context Structure' and 'C2H Prefetch Context Structure' to program context data register
- Program Context command register QDMA_IND_CTXT_CMD (0x824) to program any context to corresponing Queue
- Qid is given in bits[17:7]
- Opcode bits[6:5] selects what operations need to be done
- Which context is accessed is given in bits [4:1]
- Context programing write/read will not happen when bit[0] byst bit is set

Queue Setup

- Clear Descriptor Software Context.
- Clear Descriptor Hardware Context.
- Clear Descriptor Credit Context.
- Set-up Descriptor Software Context.
- Set-up PasID Context (need to use same ID between H2C/C2H Queues).
- Clear Prefetch Context.

- Clear Completion Context.
- Set-up Completion Context.
    - If interrupts/status writes are desired (enabled in the Completion Context), an initial Completion CIDX update is required to send the HW into a state where it is sensitive to trigger conditions. This initial CIDX update is required, because when out of reset, the HW initializes into an unarmed state.
- Set-up Prefetch Context.

Queue Teardown

Queue Tear-down (C2H Stream):

- Send Marker packet to drain the pipeline.
- Wait for “Marker” completion.
- Invalidate/Clear Descriptor Software Context.
- Invalidate/Clear Prefetch Context.
- Invalidate/Clear Completion Context.
- Invalidate Timer Context (clear cmd is not supported).

Queue Tear-down (H2C Stream & MM):

- Invalidate/Clear Descriptor Software Context.

Virtualization

QDMA implements SR-IOV passthrough virtualization where the adapter exposes a separate virtual function (VF) for use by a virtual machine (VM). A physical function (PF) can be optionally made privileged with full access to QDMA registers and resources, but only VFs implement per queue pointer update registers and interrupts. VF drivers must communicate with the driver attached to the PF through the mailbox for configuration, resource allocation, and exception handling. The QDMA implements function level reset (FLR) to enable operating system on VM to reset the device without interfering with the rest of the platform.

Table 25: **Privileged Access**

Type	Notes
Queue context/other control registers	Registers for Context access only controlled by PFs (All 4 PFs).
Status and statistics registers	Mainly PF only registers. VFs need to coordinate with a PF driver for error handling. VFs need to communicate through the mailbox with driver attached to PF.

Table 25: **Privileged Access** *(cont'd)*

Type	Notes
Data path registers	Both PFs and VFs must be able to write the registers involved in data path without needing to go through a hypervisor. Pointer update for H2C/C2H Descriptor Fetch can be done directly by VF or PF for the queues associated with the function using its own BAR space. Any pointer updates to queue that do not belong to the function will be dropped with error logged.
Other protection recommendations	Turn on IOMMU to protect bad memory accesses from VMs.
PF driver and VF driver communication	The VF driver needs to communicate with the PF driver to request operations that have global effect. This communication channel needs this ability to pass messages and generate interrupts. This communication channel utilizes a set of hardware mailboxes for each VF.

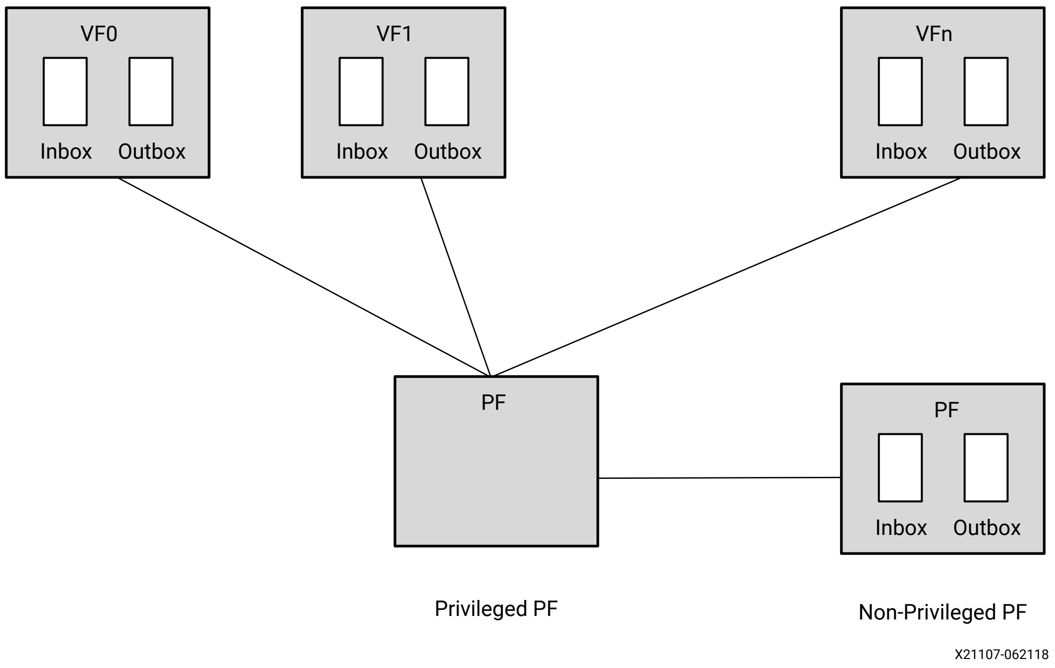
Mailbox

In a virtualized environment, the driver attached to PF has enough priviledge to program and access QDMA registers. For all the lesser priviledged functions, certain PFs and all VFs must communicate with priviledged drivers using the mailbox mechanism. The communication API must defined by the driver. The QDMA IP does not define it.

Each function (both PF and VF) has an inbox and an outbox that can fit the message size of 128B. VF accesses its own mailbox, and PF accesses its own mailbox and all the functions (PF or VF) associated with that PF. The QDMA mailbox allows the following access:

- From a VF to the associated PF.
- From a PF to any VF belonging to its own virtual function group (VFG).
- From a PF (typically a driver that does not have access to QDMA registers) to another PF.

Figure 19: **Mailbox**

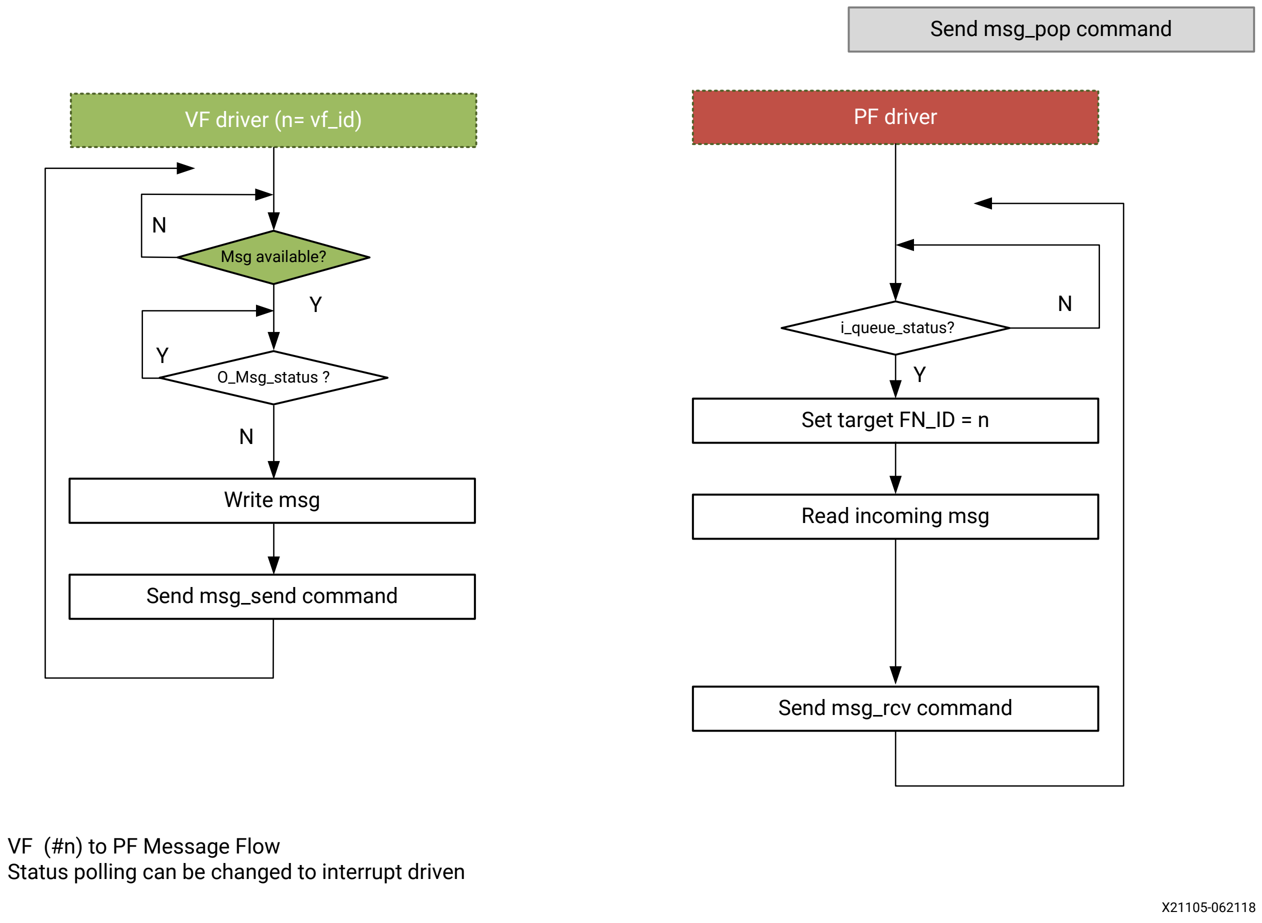


VF To PF Messaging

VF is allowed to post one message to target PF mailbox until the target function (PF) accepts it. Before posting the message the source function should make sure its `o_msg_status` is cleared, then the VF can write the message to its Outgoing Message Registers. After finishing message writing, the VF driver sends `msg_send` command through write 0x1 at the CSR address 0x04. The mailbox hardware then informs the PF driver by asserting `i_msg_status` field.

The function driver should enable the periodic polling of the `i_msg_status` to check the availability of incoming messages. At a PF side, `i_msg_status = 0x1` indicates one or more message is pending for the FP driver to pick up. The `cur_src_fn` in the Mailbox Status Register gives the function ID of the first pending message. The PF driver should then set Mailbox Target Function Register to the source function ID of the first pending message. Then access to a PF's Incoming Message Registers is indirectly, which means the mailbox hardware will always return the corresponding message bytes sent by the Target function. Upon finishing the message reading, the PF driver should also send `msg_rcv` command through write 0x2 at the CSR address 0x04. The hardware will deassert the `o_msg_status` at the source function side. The following figure illustrates the messaging flow from a VF to PF at both the source and destination sides.

Figure 20: **VF to PF Messaging Flow**



PF To VF Messaging

The messaging flow from a PF to the VFs that belong to its VFG is slightly different than the VF to PF flow since:

A PF can send messages to multiple destination functions, therefore, it may receives multiple acknowledgements at the moment when checking the status. As illustrated in the following figure, a PF driver must set Mailbox Target Function Register to the destination function ID before doing any message operation; for example, checking the incoming message status, write message, or send the command. At the VF side (receiving side), whenever a VF driver get the `i_msg_status = 0x1`, the VF driver should read its Incoming Message Registers to pick up the message. Depends on the application, the VF driver can send the `msg_rcv` immediately after reading the message or after the corresponding message being processed.

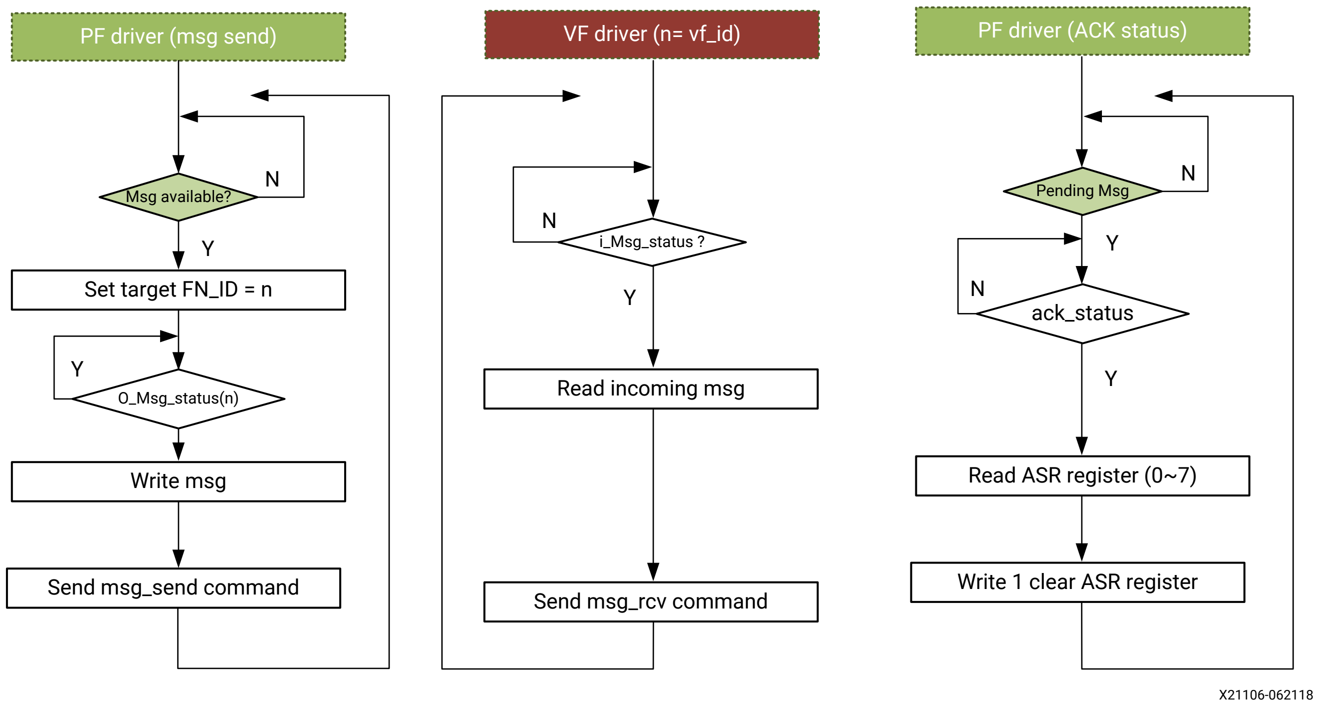
To avoid one-by-one polling of the status of outgoing messages, the mailbox hardware provides a set of Acknowledge Status Registers for each PF. Upon the mailbox receiving the `msg_rcv` command from a VF, it deasserts the `o_msg_status` field of the source PF and it also sets the corresponding bit in the Acknowledge Status Registers. For a given VF with function ID <N>, acknowledge status is at:

- Acknowledge Status Register address: <N> / 32 + <0x020 Register Address>
- Acknowledge Status bit location: <N> % 32

***Note*:** For more information about the 0x020 Register Address, see C2H_PACKET_COUNT (0x020).

The mailbox hardware asserts the `ack_status` filed in the Status Register (0x0) when there is any bit was asserted in the Acknowledge Status Register. The PF driver can poll the `ack_status` before actually read out the Acknowledge status registers. The PF driver may detect multiple completions through one register access. After being processed, the PF driver should also write the value back to the same register address to clear the status.

Figure 21: **PF to VF Messaging Flow**



Mailbox Interrupts

The mailbox module supports interrupt as the alternative event notification mechanism. Each mailbox has an Interrupt Control Register (at the offset 0x2410 for a PF, or at the offset 0x1010 for a VF). Set 1 to this register to enable the interrupt. Once the interrupt is enabled, the mailbox will send the interrupt to the QDMA through the internal user interrupt interrupt interface given there is any pending event for the mailbox to process, namely, any incoming message pending or any acknowdgement for the outgoing messages. The user interrupt interface requires an interrupt to be associated with an interrupt vector. Configure the interrupt vector through the Function Interrupt Vector Register (0x2408 for a FP, or 0x1008 for a VF) according to the driver configuration.

Enabling the interrupt does not change the event logging mechanism, which means the user must check the pending events through reading the Function Status Registers. The first step to respond to an interrupt request is disabling the interrupt. It is possible that the actual number of the pending events is more than the number of the events at the moment when the mailbox send the interrupt. It is recommended that the user interrupt handler process all the pending events that present in the status register. Upon finishing the interrupt response, the user re-enables the interrupt.

The mailbox will check its event status at the time the interrupt control change from disabled to enabled. If there is any new events that arrived the mailbox between reading the interrupt status and the re-enabling the interrupt, the mailbox will generate a new interrupt request immediately.

***Note*:** For any pre-production version of the QDMA IP, please consult Xilinx for the SR-IOV based interrupt support for the specific revision.

Function Level Reset

The FLR mechanism enables software to quiesce and reset Endpoint hardware with Function-level granularity. When a VF is reset, only the resources associated with this VF is reseted. When a PF is reset, all resources of the PF, including that of its associated VFs, will be reseted. Since FLR is a previledged operation, it must be performed by the PF driver running in the management system.

Use Mode

1. Hypervisor requests for FLR when a function is attached and detached (i.e. power on and off).
2. User can request FLR by doing

   -
     ```
     echo 1 > /sys/bus/pci/devices/$BDF/reset
     ```
   - $BDF is the bus device function number of the targeted function.

FLR Process

A complete FLR process involves of three major steps.

1. **Pre-FLR**: Pre-FLR reset all QDMA context structure, mailbox, and user logic of the target function.
   - Each function has a register called MDMA_PRE_FLR_STATUS, which keeps track of the Pre-FLR status of the function. The offset is calculated as
   - MDMA_PRE_FLR_STATUS_OFFSET = MB_base + 0x100

     which is located at offset 0x100 from the mailbox memory space of the function. Note that PF and VF have different MB_base. The definition of MDMA_PRE_FLR_STATUS is shown in the table below.
   - Software writes 1 to MDMA_PRE_FLR_STATUS[0] (bit 0) of the target function to initiate Pre-FLR. Hardware will clear MDMA_PRE_FLR_STATUS[0] when Pre-FLR completes. Software keeps polling on MDMA_PRE_FLR_STATUS[0], and only proceed to the next step when the it returns 0.

Table 26: **MDMA_PRE_FLR_STATUS Register**

Register	Offset	Field	R/W Type	Width	Default	Description
MDMA_PRE_FLR_STATUS	0x100		RW	32	0	
		reserved	RW	32:1	0	

Table 26: **MDMA_PRE_FLR_STATUS Register** *(cont'd)*

Register	Offset	Field	R/W Type	Width	Default	Description
		pre_flr_st	RW	0	0	1: Initiate Pre-FLR 0: Pre-FLR done It is set by the driver and cleared by the hardware.

2. **Quiesce**: The software must ensure all pending transaction is completed. This can be done by polling the Transaction Pending bit in the Device Status register (in PCIe Config Space) until it is clear or time out after certain period of time.
3. **PCIe-FLR**: PCIe-FLR resets all resources of the target function in PCIe controller.
    - Initiate Function Level Reset bit (bit 15 of PCIe Device Control Register) of the target function should be set to 1 to trigger FLR process in PCIe.

OS Support

If the PF driver is loaded and alive (i.e., use mode 1), all three steps aforementioned are performed by the driver. However, for UltraScale+, if an user wants to perform FLR before loading the PF driver (i.e., use mode 2), an OS kernel patch is provided to allow OS to perform the correct FLR sequence through functions defined in `//…/source/drivers/pci/quick.c`.

System Management

Resets

The QDMA Subsystem for PCIe supports all the PCIe defined resets, such as link down, reset, hot reset, and function level reset (FLR) (supports only Quiesce mode).

Soft Reset

Reset the QDMA logic through the `soft_reset_n` port. This port needs to be held in reset for a minimum of 100 clock cycles (`axi_aclk` cycles).

This does not reset PCIe hard block. It resets only the DMA portion of logic.

VDM

Vendor Defined Messages (VDMs) are an expansion of the existing messaging capabilities with PCI Express. PCI Express Specification defines additional requirements for Vendor Defined Messages, header formats and routing information. For details, see *PCI-SIG Specifications* (http://www.pcisig.com/specifications).

QDMA allows the transmission as well as reception of VDMs. TO enable this feature, select **Enable Bridge Slave Mode** in the Vivado Customize IP dialog box.

RX Vendor Defined Messages:

1. When QDMA receives a VDM, the incoming messages will be received on the `st_rx_msg` port.
2. The incoming data stream will be captured on the `st_rx_msg_data` port (per-DW).
3. The user application needs to drive the `st_rx_msg_rdy` to signal if it can accept the incoming VDMs.
4. Once `st_rx_msg_rdy` is High, the incoming VDM is forwarded to the user application.
5. The user application needs to store this incoming VDMs and track of how many packets were received.

For port details, see VDM Ports.

TX Vendor Defined Messages:

1. To enable transmission of VDM from QDMA, program the TX Message registers in the Bridge through the AXI-Lite Slave interface.
2. Bridge has TX Message Control, Header L (bytes 8-11), Header H (bytes 12-15) and TX Message Data registers as shown in the PCIe TX Message Data FIFO Register (TX_MSG_DFIFO) table below.
3. Issue a Write to offset 0xE64 through AXI-Lite Slave interface for the TX Message Header L register.
4. Program offset 0xE68 for the required VDM TX Header H register.
5. Program up to 16DW of Payload for the VDM message starting from DW0 – DW15 by sending Writes to offset 0xE6C one by one.
6. Program the `msg_routing`, `msg_code`, data length, requester function field and `msg_execute` field in the TX_MSG_CTRL register in offset 0xE60 to send the VDM TX packet.
7. The TX Message Control register also indicates the completion status of the message in bit 23. User needs to read this bit to confirm the successful transmission of the VDM packet.
8. All the fields in the registers are RW except bit 23 (`msg_fail`) in TX Control register which is cleared by writing a 1.

9. VDM TX packet will be sent on the AXI-ST RQ transmit interface.

For details about the registers, see:

- PCIe TX Message Control Register (0xE60) (TX_MSG_CTRL)
- PCIe TX Message Header L Register (0xE64) (TX_MSG_HDR_L)
- PCIe TX Message Header H Register (0xE68) (TX_MSG_HDR_H)
- PCIe TX Message Data FIFO Register (0xE6C) (TX_MSG_DFIFO)

Config Extend

PCIe extended interface can be selected for more configuration space. When the Configuration Extend Interface is selected, you are responsible for adding logic to extend the interface to make it work properly.

Expansion ROM

If selected, the Expansion ROM is activated and can be a value from 2 KB to 4 GB. According to the PCI 3.0 Local Bus Specification (*PCI-SIG Specifications* (http://www.pcisig.com/specifications)), the maximum size for the Expansion ROM BAR should be no larger than 16 MB. Selecting an address space larger than 16 MB can result in a non-compliant core.

Errors

Linkdown Errors

If the PCIe link goes down during DMA operations, transactions may be lost and the DMA may not be able to complete. In such cases, the AXI4 interfaces will continue to operate. Outstanding read requests on the C2H Bridge AXI4 MM interface receive correct completions or completions with a slave error response. The DMA will log a link down error in the status register. It is the responsibility of the driver to have a timeout and handle recovery of a link down situation.

Parity Errors

Pass through parity is supported on the primary data paths. Parity error can occur on C2H streaming, H2C streaming, Memory Mapped, Bridge Master and Bridge Slave interfaces. Parity error on Write payload can occur on C2H streaming, Memory Mapped and Bridge Slave. Double bit error on write payload and read completions for Bridge Slave interface causes parity error. Parity errors on requests to the PCIe are dropped by the Integrated Block for PCIe core, and a fatal error is logged by the PCIe. Parity errors are not recoverable and can result in unexpected behavior. Any DMA during and after the parity error should be considered invalid.

DMA Errors

Error Aggregator

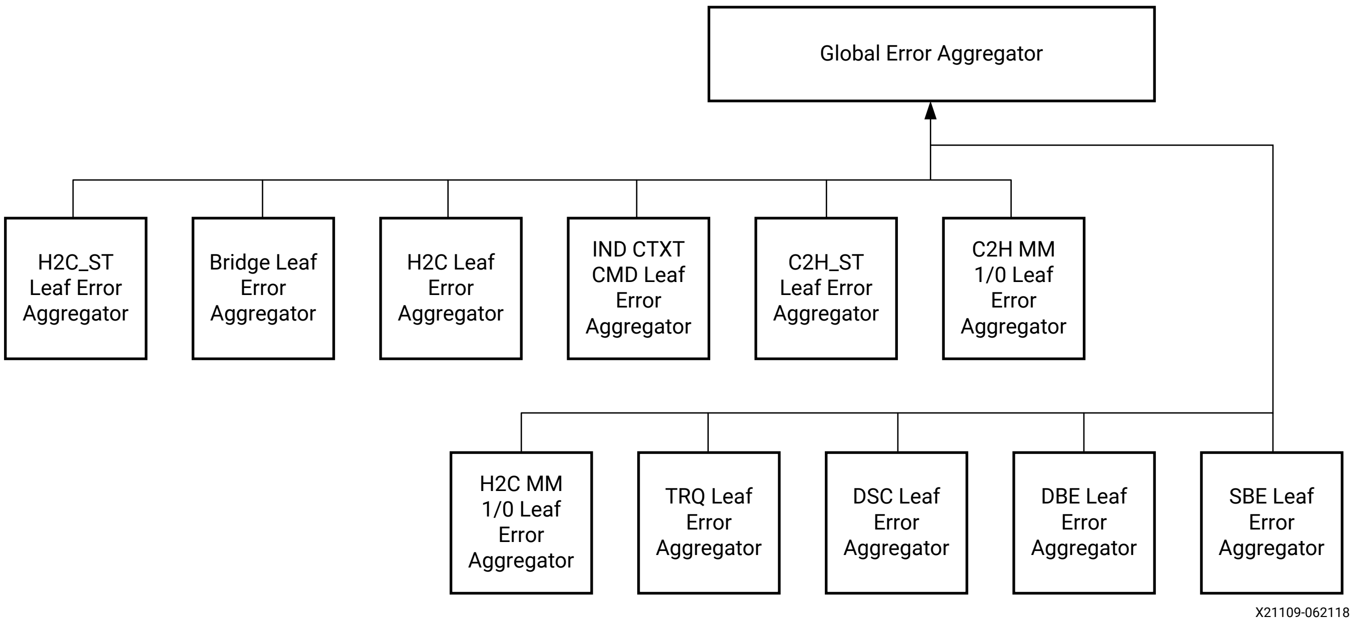
There are Leaf Error Aggregators in different places. They log the errors and propagate them to the central place. The Central Error Aggregator aggregates the errors from all of the Leaf Error Aggregators.

The QDMA_GLBL_ERR_STAT register is the error status register of the Central Error Aggregator. The bit fields indicate the locations of Leaf Error Aggregators. Then, look for the error status register of the individual Leaf Error Aggregator to find the exact error. For details, see QDMA_GLBL_ERR_STAT (0X248).

The register QDMA_GLBL_ERR_MASK is the error mask register of the Central Error Aggregator. It has the mask bits for the corresponding errors. When the mask bit is set, it will enable the corresponding error to be propagated to the next level to generate an Interrupt. The detail information of the error generated interrupt is described in the interrupt section. For details, see QDMA_GLBL_ERR_MASK (0X24C). Error interrupt is controlled by QDMA_GLBL_ERR_INT (0xB04).

Each Leaf Error Aggregator has an error status register and an error mask register. The error status register logs the error. The hardware sets the bit when the error happens, and the software can write 1’b1 to clear the bit if needed. The error mask register has the mask bits for the corresponding errors. When the mask bit is set, it will enable the propagation of the corresponding error to the Central Error Aggregator. The error mask register doesn’t affect the error logging to the error status register.

Figure 22: **Error Aggregator**


Global Error Aggregator
H2C_ST Leaf Error Aggregator
Bridge Leaf Error Aggregator
H2C Leaf Error Aggregator
IND CTXT CMD Leaf Error Aggregator
C2H_ST Leaf Error Aggregator
C2H MM 1/0 Leaf Error Aggregator
H2C MM 1/0 Leaf Error Aggregator
TRQ Leaf Error Aggregator
DSC Leaf Error Aggregator
DBE Leaf Error Aggregator
SBE Leaf Error Aggregator
X21109-062118

Links to the error status registers and the error mask registers of the Leaf Error Aggregators are as follows.

C2H Streaming Error

QDMA_C2H_ERR_STAT (0xAF0): This is the error status register of the C2H streaming errors.

QDMA_C2H_ERR_MASK (0xAF4): This the error mask register. The software can set the bit to enable the corresponding C2H streaming error to be propagated to the Central Error Aggregator.

QDMA_C2H_FIRST_ERR_QID (0xB30): This is the Qid of the first C2H streaming error.

C2H MM Error

QDMA_C2H MM Status (0x1040)

C2H MM Error Code Enable Mask (0x1054)

C2H MM Error Code (0x1058)

C2H MM Error Info (0x105C)

QDMA H2C0 MM Error

H2C0 MM Status (0x1240)

H2C MM Error Code Enable Mask (0x1254)

H2C MM Error Code (0x1258)

H2C MM Error Info (0x125C)

TRQ Error

QDMA_GLBL_TRQ_ERR_STS (0x260): This is the error status register of the Trq errors.

QDMA_GLBL_TRQ_ERR_MSK (0x264): This is the error mask register.

QDMA_GLBL_TRQ_ERR_LOG_A (0x268): This is the error logging register. It shows the select, function and the address of the access when the error happens.

Descriptor Error

QDMA_GLBL_DSC_ERR_STS (0x254)

QDMA_GLBL_DSC_ERR_MSK (0x258)

This is the error logging register. It has the QID, DMA direction, and the consumer index of the error.

QDMA_GLBL_DSC_ERR_LOG0 (0x25C)

QDMA_GLBL_TRQ_ERR_STS (0x260): This is the error status register of the Trq errors.

RAM Double Bit Error

QDMA_RAM_DBE_STS_A (0xfc)

QDMA_RAM_DBE_MSK_A (0xf8)

RAM Single Error

QDMA_RAM_SBE_STS_A (0xf4)

QDMA_RAM_SBE_MSK_A (0xf0)

C2H Streaming Fatal Error Handling

QDMA_C2H_FATAL_ERR_STAT (0xAF8): The error status register of the C2H streaming fatal errors.

QDMA_C2H_FATAL_ERR_MASK (0xAFC): The error mask register. The SW can set the bit to enable the corresponding C2H fatal error to be sent to the C2H fatal error handling logic.

QDMA_C2H_FATAL_ERR_ENABLE (0xB00): This register enables two C2H streaming fata error handling processes:

- Stop the data transfer by disabling the WRQ from the C2H DMA Write Engine.
- Invert the WPL parity on the data transfer.

Port Descriptions

The QDMA Subsystem for PCIe connects directly to the PCIe Integrated Block. The data path interfaces to the PCIe Integrated Block IP are 64, 128, 256 or 512-bits wide, and runs at up to 250 MHz depending on the configuration of the IP. The data path width applies to all data interfaces. Ports associated with this core are described below.

The subsystem interfaces are shown in QDMA Architecture.

Table 27: **Parameters**

Parameter Name	Description
PL_LINK_CAP_MAX_LINK_WIDTH	Phy lane width
C_M_AXI_ADDR_WIDTH	AXI Master interface Address width
C_M_AXI_ID_WIDTH	AXI Master interface id width
C_M_AXI_DATA_WIDTH	AXI Master interface data width 64 or 128 or 256 or 512 bits
C_S_AXI_ID_WIDTH	AXI Slave interface id width
C_S_AXI_ADDR_WIDTH	AXI Slave interface Address width
C_S_AXI_DATA_WIDTH	AXI Slave interface data width 64 or 128 or 256 or 512 bits
C_S_AXI_ID_WIDTH	AXI Slave interface id width
AXI_DATA_WIDTH	AXI DMA transfer data width. Example 64 or 128 or 256 or 512 bits

QDMA Global Ports

Table 28: **QDMA Global Port Descriptions**

Port Name	I/O	Description
sys_clk	I	Should be driven by the ODIV2 port of reference clock IBUFDS_GTE4. See the UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide (PG213).
sys_clk_gt	I	PCIe reference clock. Should be driven from the port of reference clock IBUFDS_GTE4. See the UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide (PG213).
sys_rst_n	I	Reset from the PCIe edge connector reset signal.
pci_exp_txp [PL_LINK_CAP_MAX_LINK_WIDTH-1:0]	O	PCIe TX serial interface.
pci_exp_txn [PL_LINK_CAP_MAX_LINK_WIDTH-1:0]	O	PCIe TX serial interface.
pci_exp_rxp [PL_LINK_CAP_MAX_LINK_WIDTH-1:0]	I	PCIe RX serial interface.
pci_exp_rxn [PL_LINK_CAP_MAX_LINK_WIDTH-1:0]	I	PCIe RX serial interface.
user_lnk_up	O	Output Active-High Identifies that the PCI Express core is linked up with a host device.
axi_aclk	O	User clock out. PCIe derived clock output for for all interface signals output from and input to QDMA. Use this clock to drive inputs and gate outputs from QDMA.
axi_aresetn	O	User reset out. AXI reset signal synchronous with the clock provided on the axi_aclk output. This reset should drive all corresponding AXI Interconnect aresetn signals.
user_reset_n	I	User reset (active-Low). If you need to reset the DMA logic for any reason use this port to assert reset to the DMA logic.
phy_ready	O	Phy ready out status.

All AXI interfaces are clocked out and in by the `axi_aclk` signal. You are responsible for using `axi_aclk` to driver all signals into the DMA.

AXI Bridge Master Ports

Table 29: **AXI4 Memory Mapped Master Bridge Read Address Interface Port Descriptions**

Signal Name	I/O	Description
m_axib_araddr [C_M_AXI_ADDR_WIDTH-1:0]	O	This signal is the address for a memory mapped read to the user logic from the host.
m_axib_arid [C_M_AXI_ID_WIDTH-1:0]	O	Master read address ID.
m_axib_arlen[7:0]	O	Master read address length.
m_axib_arsize[2:0]	O	Master read address size.
m_axib_arprot[2:0]	O	Master read protection type.
m_axib_arvalid	O	The assertion of this signal means there is a valid read request to the address on m_axib_araddr.
m_axib_arready	I	Master read address ready.
m_axib_arlock	O	Master read lock type.
m_axib_arcache[3:0]	O	Master read memory type.
m_axib_arburst[1:0]	O	Master read address burst type.
m_axib_aruser[28:0]	O	Master read user bits. m_axib_aruser[7:0] = function number m_axib_aruser[15:8] = reserved m_axib_aruser[18:16] = bar id m_axib_aruser[26:19] = vf offset m_axib_aruser[28:27] = vf id

Table 30: **AXI4 Memory Mapped Master Bridge Read Interface Port Descriptions**

Signal Name	I/O	Description
m_axib_rdata [C_M_AXI_DATA_WIDTH-1:0]	I	Master read data.
m_axib_ruser [C_M_AXI_DATA_WIDTH/8-1:0]	I	m_axib_ruser[C_M_DATA_WIDTH/8-1:0] = read data odd parity, per byte.
m_axib_rid [C_M_AXI_ID_WIDTH-1:0]	I	Master read ID.
m_axib_rresp[1:0]	I	Master read response.
m_axib_rlast	I	Master read last.
m_axib_rvalid	I	Master read valid.
m_axib_rready	O	Master read ready.

Table 31: **AXI4 Memory Mapped Master Bridge Write Address Interface Port Descriptions**

Signal Name	I/O	Description
m_axib_awaddr [C_M_AXI_ADDR_WIDTH-1:0]	O	This signal is the address for a memory mapped write to the user logic from the host.
m_axib_awid [C_M_AXI_ID_WIDTH-1:0]	O	Master write address ID.
m_axib_awlen[7:0]	O	Master write address length.
m_axib_awsize[2:0]	O	Master write address size.
m_axib_awburst[1:0]	O	Master write address burst type.
m_axib_awprot[2:0]	O	Master write protection type.
m_axib_awvalid	O	The assertion of this signal means there is a valid write request to the address on m_axib_araddr.
m_axib_awready	I	Master write address ready.
m_axib_awlock	O	Master write lock type.
m_axib_awcache[3:0]	O	Master write memory type.
m_axib_awuser[28:0]	O	Master write user bits. m_axib_awuser[7:0] = function number m_axib_awuser[15:8] = reserved m_axib_awuser[18:16] = bar id m_axib_awuser[26:19] = vf offset m_axib_awuser[28:27] = vf id

Table 32: **AXI4 Memory Mapped Master Bridge Write Interface Port Descriptions**

Signal Name	I/O	Description
m_axib_wdata [C_M_AXI_DATA_WIDTH-1:0]	O	Master write data.
m_axib_wuser [C_M_AXI_DATA_WIDTH/8-1:0]	O	m_axib_wuser [C_M_AXI_DATA_WIDTH/8-1:0] = write data odd parity, per byte.
m_axib_wlast	O	Master write last.
m_axib_wstrb [C_M_AXI_DATA_WIDTH/8-1:0]	O	Master write strobe.
m_axib_wvalid	O	Master write valid.
m_axib_wready	I	Master write ready.

Table 33: **AXI4 Memory Mapped Master Bridge Write Response Interface Port Descriptions**

Signal Name	I/O	Description
m_axib_bvalid	I	Master write response valid.
m_axib_bresp[1:0]	I	Master write response.
m_axib_bid [C_M_AXI_ID_WIDTH-1:0]	I	Master write response ID.

Table 33: **AXI4 Memory Mapped Master Bridge Write Response Interface Port Descriptions** *(cont'd)*

Signal Name	I/O	Description
m_axib_bready	O	Master response ready.

AXI Bridge Slave Ports

Table 34: **AXI4 Memory Mapped Slave Bridge Write Address Interface Port Descriptions**

Port Name	I/O	Description
s_axib_awid [C_S_AXI_ID_WIDTH-1:0]	I	Slave write address ID.
s_axib_awaddr [C_S_AXI_ADDR_WIDTH-1:0]	I	Slave write address.
s_axib_awuser[7:0]	I	s_axib_awuser[7:0] indicates function_number.
s_axib_awregion[3:0]	I	Slave write region decode.
s_axib_awlen[7:0]	I	Slave write burst length.
s_axib_awsize[2:0]	I	Slave write burst size.
s_axib_awburst[1:0]	I	Slave write burst type.
s_axib_awvalid	I	Slave address write valid.
s_axib_awready	O	Slave address write ready.

Table 35: **AXI4 Memory Mapped Slave Bridge Write Interface Port Descriptions**

Port Name	I/O	Description
s_axib_wdata [C_S_AXI_DATA_WIDTH-1:0]	I	Slave write data.
s_axib_wstrb [C_S_AXI_DATA_WIDTH/8-1:0]	I	Slave write strobe.
s_axib_wlast	I	Slave write last.
s_axib_wvalid	I	Slave write valid.
s_axib_wready	O	Slave write ready.
s_axib_wuser [C_S_AXI_DATA_WIDTH/8-1:0]	I	s_axib_wuser [C_S_AXI_DATA_WIDTH/8-1:0] = write data odd parity, per byte.

Table 36: **AXI4 Memory Mapped Slave Bridge Write Response Interface Port Descriptions**

Port Name	I/O	Description
s_axib_bid [C_S_AXI_ID_WIDTH-1:0]	O	Slave response ID.
s_axib_bresp[1:0]	O	Slave write response.

Table 36: **AXI4 Memory Mapped Slave Bridge Write Response Interface Port Descriptions** *(cont'd)*

Port Name	I/O	Description
s_axib_bvalid	O	Slave write response valid.
s_axib_bready	I	Slave response ready.

Table 37: **AXI4 Memory Mapped Slave Bridge Read Address Interface Port Descriptions**

Port Name	I/O	Description
s_axib_arid [C_S_AXI_ID_WIDTH-1:0]	I	Slave read address ID.
s_axib_araddr [C_S_AXI_ADDR_WIDTH-1:0]	I	Slave read address.
s_axib_arregion[3:0]	I	Slave read region decode.
s_axib_arlen[7:0]	I	Slave read burst length.
s_axib_arsize[2:0]	I	Slave read burst size.
s_axib_arburst[1:0]	I	Slave read burst type.
s_axib_arvalid	I	Slave read address valid.
s_axib_arready	O	Slave read address ready.

Table 38: **AXI4 Memory Mapped Slave Bridge Read Interface Port Descriptions**

Port Name	I/O	Description
s_axib_rid [C_S_AXI_ID_WIDTH-1:0]	O	Slave read ID tag.
s_axib_rdata [C_S_AXI_ID_WIDTH-1:0]	O	Slave read data.
s_axib_ruser [C_S_AXI_DATA_WIDTH/8-1:0]	O	s_axib_aruser[C_S_AXI_ID_WIDTH/8-1:0] = read data odd parity, per byte.
s_axib_rresp[1:0]	O	Slave read response.
s_axib_rlast	O	Slave read last.
s_axib_rvalid	O	Slave read valid.
s_axib_rready	I	Slave read ready.

AXI4-Lite Master Ports

Table 39: **Config AXI4-Lite Memory Mapped Write Master Interface Port Descriptions**

Signal Name	I/O	Description
m_axil_awaddr[31:0]	O	This signal is the address for a memory mapped write to the user logic from the host.

Table 39: **Config AXI4-Lite Memory Mapped Write Master Interface Port Descriptions** *(cont'd)*

Signal Name	I/O	Description
m_axil_awprot[2:0]	O	Protection type.
m_axil_awvalid	O	The assertion of this signal means there is a valid write request to the address on m_axil_awaddr.
m_axil_awready	I	Master write address ready.
m_axil_awuser [29:0]		m_axil_awuser[7:0] = function number m_axil_awuser[15:8]= Reserved m_axil_awuser[18:16] = bar id m_axil_awuser[26:19] = vfg offset m_axil_awuser[28:27]= vfg id
m_axil_wdata[31:0]	O	Master write data.
m_axil_wstrb[3:0]	O	Master write strobe.
m_axil_wvalid	O	Master write valid.
m_axil_wready	I	Master write ready.
m_axil_bvalid	I	Master response valid.
m_axil_bresp[1:0]	I	
m_axil_bready	O	Master response valid.

Table 40: **Config AXI4-Lite Memory Mapped Read Master Interface Port Descriptions**

Signal Name	I/O	Description
m_axil_araddr[31:0]	O	This signal is the address for a memory mapped read to the user logic from the host.
m_axil_aruser[28:0]		m_axil_aruser[7:0] = function number m_axil_aruser[15:8] = reserved m_axil_aruser[18:16] = bar id m_axil_aruser[26:19] = vfg offset m_axil_aruser[28:27] = vfg id
m_axil_arprot[2:0]	O	Protection type.
m_axil_arvalid	O	The assertion of this signal means there is a valid read request to the address on m_axil_araddr.
m_axil_arready	I	Master read address ready.
m_axil_rdata[31:0]	I	Master read data.
m_axil_rresp[1:0]	I	Master read response.
m_axil_rvalid	I	Master read valid.
m_axil_rready	O	Master read ready.

AXI4-Lite Slave Ports

Table 41: **Config AXI4-Lite Memory Mapped Write Slave Interface Signals**

Signal Name	I/O	Description
s_axil_awaddr[31:0]	I	This signal is the address for a memory mapped write to the DMA from the user logic. s_axil_awaddr[31:28]: 4’b0011 – QDMA register 4’b0000 – Bridge register
s_axil_awvalid	I	The assertion of this signal means there is a valid write request to the address on s_axil_awaddr.
s_axil_awuser	I	[7:0]: Function number
s_axil_awprot[2:0]	I	Protection type.(unused)
s_axil_awready	O	Slave write address ready.
s_axil_wdata[31:0]	I	Slave write data.
s_axil_wstrb[3:0]	I	Slave write strobe.
s_axil_wvalid	I	Slave write valid.
s_axil_wready	O	Slave write ready.
s_axil_bvalid	O	Slave write response valid.
s_axil_bresp[1:0]	O	Slave write response.
s_axil_bready	I	Save response ready.

Table 42: **Config AXI4-Lite Memory Mapped Read Slave Interface Signals**

Signal Name	I/O	Description
s_axil_araddr[31:0]	I	This signal is the address for a memory mapped read to the DMA from the user logic. s_axil_awaddr[31:28]: 4’b0011 – QDMA register 4’b0000 – Bridge register
s_axil_arprot[2:0]	I	Protection type.(unused)
s_axil_arvalid	I	The assertion of this signal means there is a valid read request to the address on s_axil_araddr.
s_axil_aruser	I	[7:0]: Function number
s_axil_arready	O	Slave read address ready.
s_axil_rdata[31:0]	O	Slave read data.
s_axil_rresp[1:0]	O	Slave read response.
s_axil_rvalid	O	Slave read valid.
s_axil_rready	I	Slave read ready.

AXI4 Memory Mapped DMA Ports

Table 43: **AXI4 Memory Mapped DMA Read Address Interface Signals**

Signal Name	Direction	Description
m_axi_araddr [C_M_AXI_ADDR_WIDTH-1:0]	O	This signal is the address for a memory mapped read to the user logic from the DMA.
m_axi_arid [3:0]	O	Standard AXI4 description, which is found in the AXI4 Protocol Specification *AMBA AXI4-Stream Protocol Specification* (ARM IHI 0051A).
m_axi_aruser[7:0]	O	[7:0]: function number
m_axi_arlen[7:0]	O	Master read burst length.
m_axi_arsize[2:0]	O	Master read burst size.
m_axi_arprot[2:0]	O	Protection type.
m_axi_arvalid	O	The assertion of this signal means there is a valid read request to the address on m_axi_araddr.
m_axi_arready	I	Master read address ready.
m_axi_arlock	O	Lock type.
m_axi_arcache[3:0]	O	Memory type.
m_axi_arburst[1:0]	O	Master read burst type.

Table 44: **AXI4 Memory Mapped DMA Read Interface Signals**

Signal Name	Direction	Description
m_axi_rdata [C_M_AXI_DATA_WIDTH-1:0]	I	Master read data.
m_axi_rid [3:0]	I	Master read ID.
m_axi_rresp[1:0]	I	Master read response.
m_axi_rlast	I	Master read last.
m_axi_rvalid	I	Master read valid.
m_axi_rready	O	Master read ready.
m_axi_ruser [C_M_AXI_DATA_WIDTH/8-1:0]	I	Master read odd data parity, per byte. This port is enabled only in Propagate Parity mode.

Table 45: **AXI4 Memory Mapped DMA Write Address Interface Signals**

Signal Name	Direction	Description
m_axi_awaddr [C_M_AXI_ADDR_WIDTH-1:0]	O	This signal is the address for a memory mapped write to the user logic from the DMA.
m_axi_awid[3:0]	O	Master write address ID.
m_axi_aruser[7:0]	O	[7:0]: function number
m_axi_awlen[7:0]	O	Master write address length.
m_axi_awsize[2:0]	O	Master write address size.

Table 45: **AXI4 Memory Mapped DMA Write Address Interface Signals** *(cont'd)*

Signal Name	Direction	Description
m_axi_awburst[1:0]	O	Master write address burst type.
m_axi_awprot[2:0]	O	Protection type.
m_axi_awvalid	O	The assertion of this signal means there is a valid write request to the address on m_axi_araddr.
m_axi_awready	I	Master write address ready.
m_axi_awlock	O	Lock type.
m_axi_awcache[3:0]	O	Memory type.

Table 46: **AXI4 Memory Mapped DMA Write Interface Signals**

Signal Name	Direction	Description
m_axi_wdata [C_M_AXI_DATA_WIDTH-1:0]	O	Master write data.
m_axi_wlast	O	Master write last.
m_axi_wstrb[31:0]	O	Master write strobe.
m_axi_wvalid	O	Master write valid.
m_axi_wready	I	Master write ready.
m_axi_wuser [C_M_AXI_DATA_WIDTH/8-1:0]	O	Master write user. m_axi_wuser[C_M_AXI_DATA_WIDTH/8-1:0] = write data odd parity, per byte. This port is enabled only in Propagate Parity mode.

Table 47: **AXI4 Memory Mapped DMA Write Response Interface Signals**

Signal Name	Direction	Description
m_axi_bvalid	I	Master write response valid.
m_axi_bresp[1:0]	I	Master write response.
m_axi_bid[3:0]	I	Master response ID.
m_axi_bready	O	Master response ready.

AXI4-Stream H2C Ports

Table 48: **AXI4-Stream H2C Port Descriptions**

Port Name	I/O	Description
m_axis_h2c_tdata [AXI_DATA_WIDTH-1:0]	O	Data output for H2C AXI4-Stream.
m_axis_h2c_dpar [AXI_DATA_WIDTH/8-1:0]	O	Odd parity calculated bit-per-byte over m_axis_h2c_tdata. m_axis_h2c_dpar[0] is parity calculated over m_axis_h2c_tdata[7:0]. m_axis_h2c_dpar[1] is parity calculated over m_axis_h2c_tdata[15:8] and so on.
m_axis_h2c_tuser [54:0]	O	H2C AXI4-Stream TUSER. For details, refer to AXI4-Stream H2C TUSER Description Table 49: AXI4-Stream H2C TUSER Description

Table 48: **AXI4-Stream H2C Port Descriptions** *(cont'd)*

Port Name	I/O	Description
m_axis_h2c_tvalid	O	Valid
m_axis_h2c_tlast	O	Indicates that this is the last cycle of the packet transfer
m_axis_h2c_tready	I	Ready

Table 49: **AXI4-Stream H2C TUSER Description**

Port Name	Location	Description
qid [10:0]	[10:0]	Queue ID
rsv	[11]	Reserved
port_id [2:0]	[14:12]	Port ID
err	[15]	If set, indicates the packet has an error. The error could be coming from PCIe, or QDMA might have encountered a double bit error.
mdata [31:0]	[47:16]	QDMA passes the lower 32 bits of the H2C AXI4-Stream descriptor on this field.
mty [5:0]	[53:48]	The number of bytes that are invalid on the last beat of the transaction. This field is 0 for a 64B transfer.
zero_byte	[54]	When set, it indicates that the current beat is an empty beat (zero bytes are being transferred).

AXI4-Stream C2H Ports

Table 50: **AXI4-Stream C2H Port Descriptions**

Port Name	I/O	Description
s_axis_c2h_tdata [AXI_DATA_WIDTH-1:0]	I	It supports 4 data widths: 64 bits, 128 bits, 256 bits, and 512 bits. Every C2H data packet has a corresponding C2H completion packet.
s_axis_c2h_dpar [AXI_DATA_WIDTH/8-1:0]	I	Odd parity computed as bit per byte.
s_axis_c2h_ctrl_len [15:0]	I	Length of the packet. For ZERO byte write, the length is 0.
s_axis_c2h_ctrl_qid [10:0]	I	Queue ID.
s_axis_c2h_ctrl_user_trig	I	User trigger; this can trigger the interrupt and the status descriptor write if they are enabled.
s_axis_c2h_ctrl_dis_cmp	I	Disable completion.
s_axis_c2h_ctrl_imm_data	I	Immediate data; This will allow only the completion and no DMA on the data payload.
s_axis_c2h_ctrl_marker	I	Marker message used for making sure pipeline is completely flushed. After that, you can safely do queue invalidation. When this bit is set, the imm_data bit has to be set also.
s_axis_c2h_ctrl_port_id [2:0]	I	Port ID.
s_axis_c2h_mty [5:0]	I	Empty byte in the last data packet.
s_axis_c2h_tvalid	I	Valid.
s_axis_c2h_tlast	I	Indicate last packet.
s_axis_c2h_tready	O	Ready.

AXI4-Stream C2H Completion Ports

Table 51: **AXI4-Stream C2H Completion Port Descriptions**

Port Name	I/O	Description
s_axis_c2h_cmpt_tdata[127:0]	I	Completion data from the user application. This contains information that is written to the completion ring in the host. This information includes the length of the packet transferred in bytes, error, color bit and user data. Based on completion size, this could be 1 or 2 beats. Every C2H completion packet has a corresponding C2H data packet.
s_axis_c2h_cmpt_size [1:0]	I	00: 8B completion. 01: 16B completion. 10: 32B completion. 11: unknown.
s_axis_c2h_cmpt_dpar [3:0]	I	Odd parity computed as bit per word. s_axis_c2h_cmpt_dpar[0] is parity over s_axis_c2h_cmpt_tdata[31:0]. s_axis_c2h_cmpt_dpar[1] is parity over s_axis_c2h_cmpt_tdata[63:31] and so on.
s_axis_c2h_cmpt_tvalid	I	Valid.
s_axis_c2h_cmpt_tlast	I	Indicates the end of the completion data transfer.
s_axis_c2h_cmpt_tready	O	Ready.

AXI4-Stream Status Ports

Table 52: **AXI-ST C2H Status Port Descriptions**

Port Name	I/O	Description
axis_c2h_status_valid	O	Valid per descriptor or per packet for the C2H stream packet of imm_data or marker
axis_c2h_status_qid [10:0]	O	QID of the packet
axis_c2h_status_drop	O	If QDMA does not have either sufficient data buffer to store a C2H packet or does not have enough descriptors to transfer the full packet to host, it drops the packet. This bit indicates if the packet was dropped or not. A packet that is not dropped is considered as having been accepted. 0: packet was not dropped 1: packet was dropped
axis_c2h_status_last	O	Last descriptor
axis_c2h_status_imm_or_marker	O	C2H packet of imm_data or marker
axis_c2h_status_cmp	O	0: Dropped packet or C2H packet with disable_wrb 1: C2H packet that has completions

AXI4-Stream C2H Write Cmp Ports

Table 53: **AXI-ST C2H Write Cmp Port Descriptions**

Port Name	I/O	Description
axis_c2h_write_cmp	O	This signal is asserted when the last data payload Wrq of the packet gets the completion of Wcp. It is just one pulse per completion.

VDM Ports

Table 54: **VDM Port Descriptions**

Port Name	I/O	Description
st_rx_msg_valid	O	Valid
st_rx_msg_data	O	Beat 1: {REQ_ID[15:0], VDM_MSG_CODE[7:0], VDM_MSG_ROUTING[2:0], VDM_DW_LENGTH[4:0]} Beat 2: VDM Lower Header [31:0] or {(Payload_length=0), VDM Higher Header [31:0]} Beat 3 to Beat <n>: VDM Payload
st_rx_msg_last	O	Indicate the last beat
st_rx_msg_rdy	I	Ready. When this interface is not used, Ready must be tied-off to 1. **Note:** When this interface is not used, Ready must be tied-off to 1.

Configuration Extend Interface Ports

The Configuration Extend interface allows the core to transfer configuration information with the user application when externally implemented configuration registers are implemented.

Table 55: **Configuration Extend Interface Port Descriptions**

Port Name	I/O	Width	Description
cfg_ext_read_received	O	1	Configuration Extend Read Received The core asserts this output when it has received a configuration read request from the link. Set when PCI Express Extended Configuration Space Enable is selected in the user defined configuration Capabilities tab in in the Vivado IDE. • All received configuration reads with cfg_ext_register_number in the range of 0xb0-0xbf is considered to be PCIe Legacy Extended Configuration Space. • All received configuration reads with cfg_ext_register_number in the range of 0x120-13F is considered to be PCIe Extended Configuration Space. • All received configuration reads regardless of their address will be indicated by 1 cycle assertion of cfg_ext_read_received. Valid data is driven on cfg_ext_register_number and cfg_ext_function_number. • Only received configuration reads within the two aforementioned ranges need to be responded by the user application outside of the IP.
cfg_ext_write_received	O	1	Configuration Extend Write Received The core asserts this output when it has received a configuration write request from the link. Set when PCI Express Extended Configuration Space Enable is selected in Capabilities tab in the Vivado IDE. • Data corresponding to all received configuration writes with cfg_ext_register_number in the range 0xb0-0xbf is presented on cfg_ext_register_number, cfg_ext_function_number, cfg_ext_write_data and cfg_ext_write_byte_enable. • All received configuration writes with cfg_ext_register_number in the range 0x120-13F is presented on cfg_ext_register_number, cfg_ext_function_number, cfg_ext_write_data and cfg_ext_write_byte_enable.
cfg_ext_register_number	O	10	Configuration Extend Register Number The 10-bit address of the configuration register being read or written. The data is valid when cfg_ext_read_received or cfg_ext_write_received is High.
cfg_ext_function_number	O	8	Configuration Extend Function Number. The 8-bit function number corresponding to the configuration read or write request. The data is valid when cfg_ext_read_received or cfg_ext_write_received is High.
cfg_ext_write_data	O	32	Configuration Extend Write Data Data being written into a configuration register. This output is valid when cfg_ext_write_received is High.
cfg_ext_write_byte_enable	O	4	Configuration Extend Write Byte Enable Byte enables for a configuration write transaction.

Table 55: **Configuration Extend Interface Port Descriptions** *(cont'd)*

Port Name	I/O	Width	Description
cfg_ext_read_data	I	32	Configuration Extend Read Data You can provide data from an externally implemented configuration register to the core through this bus. The core samples this data on the next positive edge of the clock after it sets cfg_ext_read_received High, if you have set cfg_ext_read_data_valid.
cfg_ext_read_data_valid	I	1	Configuration Extend Read Data Valid The user application asserts this input to the core to supply data from an externally implemented configuration register. The core samples this input data on the next positive edge of the clock after it sets cfg_ext_read_received High. The core expects the assertions of this signal within 262144 ('h4_0000) clock cycles of user clock after receiving the read request on cfg_ext_read_received signal. If no response is received by this time, the core will send auto-response with 'h0 payload, and the user application must discard the response and terminate that particular request immediately

FLR Ports

Table 56: **FLR Port Descriptions**

Port Names	I/O	Description
usr_flr_fnc [7:0]	O	Function The function number of the FLR status change.
usr_flr_set	O	Set Asserted for 1 cycle indicating that the FLR status of the function indicated on usr_flr_fnc[7:0] is active.
usr_flr_clr	O	Clear Asserted for 1 cycle indicating that the FLR status of the function indicated on usr_flr_fnc[7:0] is completed.
usr_flr_done_fnc [7:0]	I	Done Function The function for which FLR has been completed by user logic.
usr_flr_done_vld	I	Done Valid Assert for one cycle to signal that FLR for the function on usr_flr_done_fnc[7:0] has been completed.

QDMA Descriptor Bypass Input Ports

Table 57: **QDMA H2C-Streaming Bypass Input Port Descriptions**

Port Name	I/O	Description
h2c_byp_in_st_addr [63:0]	I	64-bit starting address of the DMA transfer.
h2c_byp_in_st_len [15:0]	I	The number of bytes to transfer.
h2c_byp_in_st_sop	I	Indicates start of packet. Set for the first descriptor. Reset for the rest of the descriptors.

Table 57: **QDMA H2C-Streaming Bypass Input Port Descriptions** *(cont'd)*

Port Name	I/O	Description
h2c_byp_in_st_eop	I	Indicates end of packet. Set for the last descriptor. Reset for the rest of the descriptors
h2c_byp_in_st_sdi	I	H2C Bypass In Status Descriptor/Interrupt If set, it is treated as an indication from the user application to the QDMA to send the status descriptor to host, and to generate an interrupt to host when the QDMA has fetched the last byte of the data associated with this descriptor. The QDMA honors the request to generate an interrupt only if interrupts have been enabled in the H2C SW context for this QID and armed by the driver. This can only be set for an EOP descriptor.
h2c_byp_in_st_mrkr_req	I	H2C Bypass In Marker Request When set, the descriptor passes through the H2C Engine pipeline and once completed, produces a marker response on the H2C Streaming Bypass-Out interface. This can only be set for an EOP descriptor.
h2c_byp_in_st_no_dma	I	H2C Bypass In No DMA When sending in a descriptor through this interface with this signal asserted, it informs the QDMA to not send any PCIe requests for this descriptor. Because no PCIe request is sent out, no corresponding DMA data is issued on the H2C Streaming output interface. This is typically used in conjunction with h2c_byp_in_st_sdi to cause Status Descriptor/Interrupt when the user logic is out of the actual descriptors and still wants to drive the h2c_byp_in_st_sdi signal. If h2c_byp_in_st_mrkr_req and h2c_byp_in_st_sdi are reset when sending in a no-DMA descriptor, the descriptor is treated as a NOP and is completely consumed inside the QDMA without any interface activity. If h2c_byp_in_st_no_dma is set, then both h2c_byp_in_st_sop and h2c_byp_in_st_eop must be set. If h2c_byp_in_st_no_dma is set, the QDMA ignores the address and length fields of this interface.
h2c_byp_in_st_qid [10:0]	I	The QID associated with the H2C descriptor ring.
h2c_byp_in_st_error	I	This bit can be set to indicate an error for the queue. The descriptor will not be processed. Context will be updated to reflect and error in the queue
h2c_byp_in_st_func [7:0]	I	PCIe function ID
h2c_byp_in_st_cidx [15:0]	I	The CIDX that will be used for the status descriptor update and/or interrupt (aggregation mode). Generally the CIDX should be left unchanged from when it was received from the descriptor bypass output interface.
h2c_byp_in_st_port_id [2:0]	I	QDMA port ID
h2c_byp_in_st_vld	I	Valid. High indicates descriptor is valid, one pulse for one descriptor.
h2c_byp_in_st_rdy	O	Ready to take in descriptor

Table 58: **QDMA H2C-MM Descriptor Bypass Input Port Descriptions**

Port Name	I/O	Description
h2c_byp_in_mm_radr[63:0]	I	The read address for the DMA data.
h2c_byp_in_mm_wadr[63:0]	I	The write address for the dma data.

Table 58: **QDMA H2C-MM Descriptor Bypass Input Port Descriptions** *(cont'd)*

Port Name	I/O	Description
h2c_byp_in_mm_len[27:0]	I	The dma data length. The upper 12 bits must be tied to 0. Thus only the lower 16 bits of this field can be used for specifying the length.
h2c_byp_in_mm_sdi	I	H2C-MM Bypass In Status Descriptor/Interrupt If set, it is treated as an indication from the User to QDMA to send the status descriptor to host and generate an interrupt to host when the QDMA has fetched the last byte of the data associated with this descriptor. The QDMA will honor the request to generate an interrupt only if interrupts have been enabled in the H2C ring context for this QID and armed by the driver
h2c_byp_in_mm_mrkr_req	I	H2C-MM Bypass In Completion Request Indication from the User that the QDMA must send a completion status to the User once the QDMA has completed the data transfer of this descriptor
h2c_byp_in_mm_qid [10:0]	I	The QID associated with the H2C descriptor ring
h2c_byp_in_mm_error	I	This bit can be set to indicate an error for the queue. The descriptor will not be processed. Context will be updated to reflect and error in the queue.
h2c_byp_in_mm_func [7:0]	I	PCIe function ID
h2c_byp_in_mm_cidx [15:0]	I	The CIDX that will be used for the status descriptor update and/or interrupt (aggregation mode). Generally the CIDX should be left unchanged from when it was received from the descriptor bypass output interface.
h2c_byp_in_mm_port_id [2:0]	I	QDMA port ID
h2c_byp_in_mm_vld	I	Valid. High indicates descriptor is valid, one pulse for one descriptor.
h2c_byp_in_mm_rdy	O	Ready to take in descriptor

Table 59: **QDMA C2H-Streaming Simple Bypass Input Port Descriptions**

Port Name	I/O	Description
c2h_byp_in_st_sim_addr [63:0]	I	64bit address where QDMA will DMA the data to
c2h_byp_in_st_sim_qid [10:0]	I	The QID associated with the C2H descriptor ring
c2h_byp_in_st_sim_error	I	This bit can be set to indicate an error for the queue. The descriptor will not be processed. Context will be updated to reflect and error in the queue.
c2h_byp_in_st_sim_func [7:0]	I	PCIe function ID
c2h_byp_in_st_sim_port_id[2:0]	I	QDMA port ID
c2h_byp_in_st_sim_vld	I	Valid. High indicates descriptor is valid, one pulse for one descriptor.
c2h_byp_in_st_sim_rdy	O	Ready to take in descriptor

Table 60: **QDMA C2H-Streaming Cache Bypass Input Port Descriptions**

Port Name	I/O	Description
c2h_byp_in_st_csh_addr [63:0]	I	64bit address where QDMA will DMA the data to
c2h_byp_in_st_csh_qid [10:0]	I	The QID associated with the C2H descriptor ring

Table 60: **QDMA C2H-Streaming Cache Bypass Input Port Descriptions** *(cont'd)*

Port Name	I/O	Description
c2h_byp_in_st_csh_error	I	This bit can be set to indicate an error for the queue. The descriptor will not be processed. Context will be updated to reflect and error in the queue.
c2h_byp_in_st_csh_func [7:0]	I	PCIe function ID
c2h_byp_in_st_csh_port_id[2:0]	I	QDMA port ID
c2h_byp_in_st_csh_vld	I	Valid. High indicates descriptor is valid, one pulse for one descriptor.
c2h_byp_in_st_csh_rdy	O	Ready to take in descriptor

Table 61: **QDMA C2H-MM Descriptor Bypass Input Port Descriptions**

Port Name	I/O	Description
c2h_byp_in_mm_raddr [63:0]	I	The read address for the dma data.
c2h_byp_in_mm_wadr[63:0]	I	The write address for the dma data.
c2h_byp_in_mm_len[27:0]	I	The dma data length.
c2h_byp_in_mm_sdi	I	C2H Bypass In Status Descriptor/Interrupt If set, it is treated as an indication from the User to QDMA to send the status descriptor to host, and generate an interrupt to host when the QDMA has fetched the last byte of the data associated with this descriptor. The QDMA will honor the request to generate an interrupt only if interrupts have been enabled in the C2H ring context for this QID and armed by the driver
c2h_byp_in_mm_mrkr_req	I	C2H Bypass In Marker Request Indication from the User that the QDMA must send a completion status to the User once the QDMA has completed the data transfer of this descriptor
c2h_byp_in_mm_qid [10:0]	I	The QID associated with the C2H descriptor ring
c2h_byp_in_mm_error	I	This bit can be set to indicate an error for the queue. The descriptor will not be processed. Context will be updated to reflect and error in the queue.
c2h_byp_in_mm_func [7:0]	I	PCIe function ID
c2h_byp_in_mm_cidx [15:0]	I	The User must echo the CIDX from the descriptor that it received on the bypass-out interface
c2h_byp_in_mm_port_id[2:0]	I	QDMA port ID
c2h_byp_in_mm_vld	I	Valid. High indicates descriptor is valid, one pulse for one descriptor.
c2h_byp_in_mm_rdy	O	Ready to take in descriptor

QDMA Descriptor Bypass Output Ports

Table 62: **QDMA H2C Descriptor Bypass Output Port Descriptions**

Port Name	I/O	Description
h2c_byp_out_dsc [255:0]	O	The H2C descriptor fetched from the host. For Streaming descriptor, use the lower 64b of this field as the address. The remaining bits can be ignored.

Table 62: **QDMA H2C Descriptor Bypass Output Port Descriptions** *(cont'd)*

Port Name	I/O	Description
h2c_byp_out_mrkr_rsp	O	Indicates completion status in response to h2c_byp_in_st_mrkr_req (Stream) or h2c_byp_in_mm_mrkr_req (MM).
h2c_byp_out_st_mm	O	Indicates whether this is a streaming data descriptor or memory-mapped descriptor. 0: streaming 1: memory-mapped
h2c_byp_out_dsc_sz [1:0]	O	Descriptor size. 0: 8B 1: 16B 2: 32B 3: Reserved
h2c_byp_out_qid [10:0]	O	The QID associated with the H2C descriptor ring.
h2c_byp_out_error	O	Indicates that an error was encountered in descriptor fetch or execution of a previous descriptor.
h2c_byp_out_func [7:0]	O	PCIe function ID
h2c_byp_out_cidx [15:0]	O	H2C Bypass Out Consumer Index The ring index of the descriptor fetched. The User must echo this field back to QDMA when submitting the descriptor on the bypass-in interface.
h2c_byp_out_port_id [2:0]	O	QDMA port ID
h2c_byp_out_vld	O	Valid. High indicates descriptor is valid, one pulse for one descriptor.
h2c_byp_out_rdy	I	Ready. When this interface is not used, Ready must be tied-off to 1.

Table 63: **QDMA C2H Descriptor Bypass Output Port Descriptions**

Port Name	I/O	Description
c2h_byp_out_dsc [255:0]	O	The C2H descriptor fetched from the host. For Streaming descriptor, use the lower 64b of this field as the address. The remaining bits can be ignored. If 'c2h_byp_out_mrkr_rsp' is asserted see table below for response.
c2h_byp_out_mrkr_rsp	O	Indicates completion status in response to s_axis_c2h_ctrl_marker (Stream) or c2h_byp_in_mm_mrkr_req (MM). For the completions status for s_axis_c2h_ctrl_marker (Stream), the details are given in the table below.
c2h_byp_out_st_mm	O	Indicates whether this is a streaming data descriptor or memory-mapped descriptor. 0: streaming 1: memory-mapped
c2h_byp_in_dsc_sz [1:0]	O	Descriptor size. 0: 8B 1: 16B 2: 32B 3: reserved
c2h_byp_out_qid [10:0]	O	The QID associated with the H2C descriptor ring.
c2h_byp_out_error	O	Indicates that an error was encountered in descriptor fetch or execution of a previous descriptor.

Table 63: **QDMA C2H Descriptor Bypass Output Port Descriptions** *(cont'd)*

Port Name	I/O	Description
c2h_byp_out_func [7:0]	O	PCIe function ID.
c2h_byp_out_cidx [15:0]	O	C2H Bypass Out Consumer Index The ring index of the descriptor fetched. The User must echo this field back to QDMA when submitting the descriptor on the bypass-in interface.
c2h_byp_out_port_id [2:0]	O	QDMA port ID
c2h_byp_out_vld	O	Valid. High indicates descriptor is valid, one pulse for one descriptor.
c2h_byp_out_rdy	I	Ready. When this interface is not used, Ready must be tied-off to 1.

Table 64: **QDMA C2H Descriptor Bypass Output Marker Response Descriptions**

Field Name	Location	Description
err[1:0]	[1:0]	Error code reported by the C2H Engine. 0: No error 1: SW gave bad Completion CIDX update 2: Descriptor error received while processing the C2H packet 3: Completion dropped by the C2H Engine because Completion Ring was full
retry_marker_req	[2]	The marker request could not be completed because an Interrupt could not be generated in spite of being enabled. This happens when an Interrupt is already outstanding on the queue when the marker request was received. The User logic must wait and retry the marker request again.
rsv	[255:3]	Reserved

QDMA Descriptor Credit Input Ports

Table 65: **QDMA Descriptor Credit Input Port Descriptions**

Port Name	I/O	Description
dsc_crdt_in_vld	I	Valid. When asserted the user must be presenting valid data on the bus and maintain the bus values until both valid and ready are asserted on the same cycle.
dsc_crdt_in_rdy	O	Ready. Assertion of this signal indicates the DMA is ready to accept data from this bus.
dsc_crdt_in_dir	I	Indicates whether credits are for H2C or C2H descriptor ring. 0: H2C 1: C2H
dsc_crdt_in_qid [10:0]	I	The QID associated with the descriptor ring for the credits are being added.
dsc_crdt_in_crdt [15:0]	I	The number of descriptor credits that the User is giving to QDMA to fetch descriptors from the host

QDMA Traffic Manager Credit Output Ports

Table 66: **QDMA TM Credit Output Port Descriptions**

Port Name	I/O	Description
tm_dsc_sts_vld	O	Valid. Indicates valid data on the output bus. Valid data on the bus is held until tm_dsc_sts_rdy is asserted by the user.
tm_dsc_sts_rdy	I	Ready. Assertion indicates that the user logic is ready to accept the data on this bus. When this interface is not used, Ready must be tied-off to 1. **Note:** When this interface is not used, Ready must be tied-off to 1.
tm_dsc_sts_byp	O	Shows the bypass bit in the SW descriptor context
tm_dsc_sts_dir	O	Indicates whether the status update is for a H2C or C2H descriptor ring. 0: H2C 1: C2H
tm_dsc_sts_mm	O	Indicates whether the status update is for a streaming or memory-mapped queue. 0: streaming 1: memory-mapped
tm_dsc_sts_qid [10:0]	O	The QID of the ring
tm_dsc_sts_avl [15:0]	O	If tm_dsc_sts_qinv is set, this is the number of credits available in the descriptor engine. If tm_dsc_sts_qinv is not set this is the number of new descriptors that have been posted to the ring since the last time this update was sent.
tm_dsc_sts_qinv	O	If set, it indicates that the queue has been invalidated. This is used by the user application to reconcile the credit accounting between the user application and QDMA.
tm_dsc_sts_qen	O	The current queue enable status.
tm_dsc_sts_irq_arm	O	If set, it indicates to the User that the driver is ready to accept interrupts
tm_dsc_sts_error	O	Set to 1 if the PIDX update is beyond the current CIDX of associated queue.
tm_dsc_sts_port_id [2:0]	O	The port id associated with the queue from the queue context.

User Interrupts

Table 67: **User Interrupts Port Descriptions**

Port Name	I/O	Description
usr_irq_in_vld	I	Valid An assertion indicates that an interrupt associated with the vector, function, and pending fields on the bus should be generated to PCIe. Once asserted, Usr_irq_in_vld must remain high until usr_irq_out_ack is asserted by the DMA.
usr_irq_in_vec [4:0]	I	Vector The MSIX vector to be sent.
usr_irq_in_fnc [7:0]	I	Function The function of the vector to be sent.

Table 67: **User Interrupts Port Descriptions** *(cont'd)*

Port Name	I/O	Description
usr_irq_out_ack[4:0]	O	Interrupt Acknowledge An assertion of the acknowledge bit indicates that the interrupt was transmitted on the link the user logic must wait for this pulse before signaling another interrupt condition.
usr_irq_out_fail	O	Interrupt Fail An assertion of fail indicates that the interrupt request was aborted before transmission on the link.

Register Space

QDMA PF Address Register Space

Table 68: **QDMA PF Address Register Space**

Target Name	Base (Hex)	Byte size (dec)	Notes
QDMA_TRQ_SEL_GLBL1 (0x00000)	00000000	256	QDMA Configuration CSR space
QDMA_TRQ_SEL_GLBL2 (0x00100)	00000100	256	Driver visible attribute space
QDMA_TRQ_SEL_GLBL (0x00200)	00000200	512	QDMA CSR space
QDMA_TRQ_SEL_FMAP (0x00400)	00000400	1024	Function to Queue mapping register space
QDMA_TRQ_SEL_IND (0x00800)	00000800	512	Indirect context register space
QDMA_TRQ_SEL_C2H (0x00A00)	00000A00	512	Card to Host Streaming register space
QDMA_TRQ_SEL_H2C (0x00E00)	00000E00	512	Host to Card Streaming register space
QDMA_TRQ_SEL_C2H_MM (0x1000)	00001000	256	Card to Host AXI-MM register space
QDMA_TRQ_SEL_H2C_MM (0x1200)	00001200	256	Host to Card AXI-MM register space
QDMA_TRQ_MSIX (0x1400)	00001400	4096	Space for 32 MSIX vectors and PBA
QDMA_PF_MAILBOX (0x2400)	00002400	16384	Mailbox/FLR register space
QDMA_TRQ_SEL_QUEUE_PF (0x6400)	00006400	32768	PF Direct QCSR (16B per Q, up to max of 2048 Qs per function)

QDMA_TRQ_SEL_GLBL1 (0x00000)

Table 69: **QDMA_TRQ_SEL_GLBL1 (0x00000) Register Space**

Register Name	Address (hex)	Description
Config Block Identifier (0x00)	0x00	Configuration block Identifier register
Config Block BusDev (0x04)	0x04	Bus device function register
Config Block PCIE Max Payload Size (0x08)	0x08	Max Payload size
Config Block PCIE Max Read Request Size (0x0C)	0x0C	Max read request size

Table 69: **QDMA_TRQ_SEL_GLBL1 (0x00000) Register Space** *(cont'd)*

Register Name	Address (hex)	Description
Config Block System ID (0x10)	0x10	System ID register
Config Block MSI Enable (0x14)	0x14	Interrupt config register
Config Block PCIE Data Width (0x18)	0x18	PCIe data width register
Config PCIE Control (0x1C)	0x1C	PCIe control register
Config AXI User Max Payload Size (0x40)	0x40	AXI Max Payload size register
Config AXI User Max Read Request Size (0x44)	0x44	AXI Max Read Request register
Config Block Misc Control (0x4C)	0x4C	Miscellaneous controls
Config Block Scratch7-0 (0x80-0x9C)	0x80-0x9C	General purpose scratch registers
QDMA_RAM_SBE_MSK_A (0xf0)	0xF0	ECC Mask register for Single bit error
QDMA_RAM_SBE_STS_A (0xf4)	0xF4	ECC Single bit error status
QDMA_RAM_DBE_MSK_A (0xf8)	0xF8	ECC Mask register for double bit error
QDMA_RAM_DBE_STS_A (0xfc)	0xFC	ECC double bit error status

Config Block Identifier (0x00)

Table 70: **Config Block Identifier (0x00)**

Bit	Default	Access Type	Field	Description
31:20	12’h1fd	RO	Identifier	DMA Subsystem for PCIe identifier
19:16	4’h3	RO	Config_block_identifier	Config Identifier
15:8	8’h0	RO	Reserved	Reserved
7:0	8'h00	RO	Version	Version

Config Block BusDev (0x04)

Table 71: **Config Block BusDev (0x04)**

Bit	Default	Access Type	Field	Description
[15:0]	PCIe IP	RO	BDF	bus_dev Bus, device, and function

Config Block PCIE Max Payload Size (0x08)

Table 72: **Config Block PCIE Max Payload Size (0x08)**

Bit	Default	Access Type	Field	Description
[2:0]	PCIe IP	RO	pcie_max_payload	pcie_max_payload Maximum write payload size. This is the lesser of the PCIe IP MPS and DMA Subsystem for PCIe parameters. 3'b000: 128 bytes 3'b001: 256 bytes 3'b010: 512 bytes 3'b011: 1024 bytes 3'b100: 2048 bytes 3'b101: 4096 bytes

Config Block PCIE Max Read Request Size (0x0C)

Table 73: **Config Block PCIE Max Read Request Size (0x0C)**

Bit	Default	Access Type	Field	Description
[2:0]	PCIe IP	RO	pcie_max_read	pcie_max_read Maximum read request size. This is the lesser of the PCIe IP MRRS and DMA Subsystem for PCIe parameters. 3'b000: 128 bytes 3'b001: 256 bytes 3'b010: 512 bytes 3'b011: 1024 bytes 3'b100: 2048 bytes 3'b101: 4096 bytes

Config Block System ID (0x10)

Table 74: **Config Block System ID (0x10)**

Bit	Default	Access Type	Field	Description
[15:0]	16’hff01	RO	system_id	system_id DMA Subsystem for PCIe system ID

Config Block MSI Enable (0x14)

Table 75: **Config Block MSI Enable (0x14)**

Bit	Default	Access Type	Field	Description
[17]	PCIe IP	RO	MSI_enable3	MSI Enable status for PF3
[16]	PCIe IP	RO	MSIX_enable3	MSIX Enable status for PF3
[13]	PCIe IP	RO	MSI_enable2	MSI Enable status for PF2
[12]	PCIe IP	RO	MSIX_enable2	MSIX Enable status for PF2
[9]	PCIe IP	RO	MSI_enable1	MSI Enable status for PF1
[8]	PCIe IP	RO	MSIX_enable1	MSIX Enable status for PF1
[1]	PCIe IP	RO	MSI_enable0	MSI Enable status for PF0
[0]	PCIe IP	RO	MSIX_enable0	MSIX Enable status for PF0

Config Block PCIE Data Width (0x18)

Table 76: **Config Block PCIE Data Width (0x18)**

Bit	Default	Access Type	Field	Description
[2:0]	C_DAT_WIDTH	RO	Datapath Width	Datapath Width 0: 64 bits 1: 128 bits 2: 256 bits 3: 512 bits

Config PCIE Control (0x1C)

Table 77: **Config PCIE Control (0x1C)**

Bit	Default	Access Type	Field	Description
[1]	1'b1	RW	rrq_disable	Disable read requests to PCIe. Bridge slave, and DMA reads toward PCIe will be cancelled and return a completion error.
[0]	1'b1	RW	Relaxed_ordering	Relaxed Ordering. PCIe read request TLPs are generated with the relaxed ordering bit set.

Config AXI User Max Payload Size (0x40)

Table 78: **Config AXI User Max Payload Size (0x40)**

Bit	Default	Access Type	Field	Description
6:4	3'h5	RO	user_max_payload_issued	user_eff_payload The actual maximum payload size issued to the user application. This value might be lower than user_prg_payload due to IP configuration or datapath width. 3'b000: 128 bytes 3'b001: 256 bytes 3'b010: 512 bytes 3'b011: 1024 bytes 3'b100: 2048 bytes 3'b101: 4096 bytes
2:0	3'h5	RW	user_max_payload_prog	user_prg_payload The programmed maximum payload size issued to the user application application for DMA. This register should only be changed when the DMA is idle. 3'b000: 128 bytes 3'b001: 256 bytes 3'b010: 512 bytes 3'b011: 1024 bytes 3'b100: 2048 bytes 3'b101: 4096 bytes

Config AXI User Max Read Request Size (0x44)

Table 79: **Config AXI User Max Read Request Size (0x44)**

Bit	Default	Access Type	Field	Description
6:4	3'h5	RO	usr_max_read_request_issued	user_eff_read Maximum read request size issued to the user application. This value may be lower than user_max_read due to PCIe configuration or datapath width. 3'b000: 128 bytes 3'b001: 256 bytes 3'b010: 512 bytes 3'b011: 1024 bytes 3'b100: 2048 bytes 3'b101: 4096 bytes

Table 79: **Config AXI User Max Read Request Size (0x44)** *(cont'd)*

Bit	Default	Access Type	Field	Description
2:0	3'h5	RW	usr_max_read_request_prog	user_prg_read Maximum read request size issued to the user application for DMA. This register should only be changed when the DMA is idle. 3'b000: 128 bytes 3'b001: 256 bytes 3'b010: 512 bytes 3'b011: 1024 bytes 3'b100: 2048 bytes 3'b101: 4096 bytes

Config Block Misc Control (0x4C)

Table 80: **Config Block Misc Control (0x4C)**

Bit	Default	Access Type	Field	Description
[19:8]	NUM_TAGS	RW	num_tag	Limits the number of tags used. Hardware enforces that the programmed value is less than or equal to the number of tags configured in the IP. This register should only be updated when the Bridge Slave and DMA are idle.
[4:0]	based on datapath width: 64: 5'h2 128: 5'h3 256: 5'h6 512: 6'h9	RW	rq_metering_multiplier	Limits the max outstanding read data to prevent overflow of the PCIe controller completion buffer. This must be programmed appropriately for the configured PCIe Controller completion buffer sizing. This register should only be updated wen the Bridge Slave and DMA are idle. Metering limit = (value +1) * 32 *64 Bytes

Config Block Scratch7-0 (0x80-0x9C)

Table 81: **Config Block Scratch (0x80-9C)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW	scratch	General purpose scratch registers. These fields do not affect any QDMA hardware functions.

QDMA_RAM_SBE_MSK_A (0xf0)

Table 82: **QDMA_RAM_SBE_MSK_A (0xf0)**

Bit	Default	Access Type	Field	Description
[31:0]			mask	Error logging enable masks. See QMD_RAM_SBE_STS for definitions

QDMA_RAM_SBE_STS_A (0xf4)

Table 83: **QDMA_RAM_SBE_STS_A (0xf4)**

Bit	Default	Access Type	Field	Description
[31]			reserved	
[30]			pfch_ll_ram	C2H ST prefetch list RAM single bit ECC error.
[29]			wrb_ctxt_ram	C2H ST completion context RAM single bit ECC error.
[28]			pfch_ctxt_ram	C2H ST prefetch RAM single bit ECC error.
[27]			desc_req_fifo_ram	C2H ST descriptor request RAM single bit ECC error.
[26]			int_ctxt_ram	Interrupt context RAM single bit ECC error.
[25]			int_qid2vec_ram	Interrupt QID2VEC RAM single bit ECC error.
[24]			wrb_coal_data_ram	Completion Coalescing RAM single bit ECC error.
[23]			tuser_fifo_ram	C2H ST TUSER RAM single bit ECC error.
[22]			qid_fifo_ram	C2H ST QID FIFO RAM single bit ECC error.
[21]			payload_fifo_ram	C2H ST payload RAM single bit ECC error.
[20]			timer_fifo_ram	Timer fifo RAM single bit ECC error.
[19]			pasid_ctxt_ram	PASID configuration RAM single bit ECC error.
[18]			dsc_cpld	Descriptor engine fetch completion data RAM single bit ECC error.
[17]			dsc_cpli	Descriptor engine fetch completion information RAM single bit ECC error.
[16]			dsc_sw_ctxt	Descriptor engine software context RAM single bit ECC error.
[15]			dsc_crd_rcv	Descriptor engine receive credit context RAM single bit ECC error.
[14]			dsc_hw_ctxt	Descriptor engine hardware context RAM single bit ECC error.
[13]			func_map	Function map RAM single bit ECC error.
[12]			c2h_wr_brg_dat	Bridge slave write data buffer single bit ECC error.
[11]			c2h_rd_brg_dat	Bridge slave read data buffer single bit ECC error.
[10]			h2c_wr_brg_dat	Bridge master write single bit ECC error.
[9]			h2c_rd_brg_dat	Bridge master read single bit ECC error.
[8:5]			reserved	
[4]			mi_c2h0_dat	C2H MM data buffer single bit ECC error.

Table 83: **QDMA_RAM_SBE_STS_A (0xf4)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[3:1]			reserved	
[0]			mi_h2c0_dat	H2C MM data buffer single bit ECC error.

QDMA_RAM_DBE_MSK_A (0xf8)

Table 84: **QDMA_RAM_DBE_MSK_A (0xf8)**

Bit	Default	Access Type	Field	Description
[31:0]			mask	Error logging enable masks. See QMD_RAM_DBE_STS for definitions

QDMA_RAM_DBE_STS_A (0xfc)

Table 85: **QDMA_RAM_DBE_STS_A (0xfc)**

Bit	Default	Access Type	Field	Description
[31]			reserved	
[30]			pfch_ll_ram	C2H ST prefetch list RAM double bit ECC error.
[29]			wrb_ctxt_ram	C2H ST completion context RAM double bit ECC error.
[28]			pfch_ctxt_ram	C2H ST prefetch RAM double bit ECC error.
[27]			desc_req_fifo_ram	C2H ST descriptor request RAM double bit ECC error.
[26]			int_ctxt_ram	Interrupt context RAM double bit ECC error.
[25]			int_qid2vec_ram	Interrupt QID2VEC RAM double bit ECC error.
[24]			wrb_coal_data_ram	Completion Coalescing RAM double bit ECC error.
[23]			tuser_fifo_ram	C2H ST TUSER RAM double bit ECC error.
[22]			qid_fifo_ram	C2H ST QID FIFO RAM double bit ECC error.
[21]			payload_fifo_ram	C2H ST payload RAM double bit ECC error.
[20]			timer_fifo_ram	Timer fifo RAM double bit ECC error.
[19]			pasid_ctxt_ram	PASID configuration RAM double bit ECC error.
[18]			dsc_cpld	Descriptor engine fetch completion data RAM double bit ECC error.
[17]			dsc_cpli	Descriptor engine fetch completion information RAM double bit ECC error.
[16]			dsc_sw_ctxt	Descriptor engine software context RAM double bit ECC error.
[15]			dsc_crd_rcv	Descriptor engine receive credit context RAM double bit ECC error.
[14]			dsc_hw_ctxt	Descriptor engine hardware context RAM double bit ECC error.
[13]			func_map	Function map RAM double bit ECC error.

Table 85: **QDMA_RAM_DBE_STS_A (0xfc)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[12]			c2h_wr_brg_dat	Bridge slave write data buffer double bit ECC error.
[11]			c2h_rd_brg_dat	Bridge slave read data buffer double bit ECC error.
[10]			h2c_wr_brg_dat	Bridge master write double bit ECC error.
[9]			h2c_rd_brg_dat	Bridge master read double bit ECC error.
[8:5]			reserved	
[4]			mi_c2h0_dat	C2H MM data buffer double bit ECC error.
[3:1]			reserved	
[0]			mi_h2c0_dat	H2C MM data buffer double bit ECC error.

QDMA_TRQ_SEL_GLBL2 (0x00100)

Table 86: **QDMA_TRQ_SEL_GLBL2 (0x00100) Register Space**

Register	Address	Description
QDMA_GLBL2_IDENTIFER (0x100)	0x100	Identifier 0x1FD3xxxx.
QDMA_GLBL2_PF_BARLITE_INT (0x104)	0x104	PF BAR information for internal DMA registers.
QDMA_GLBL2_PF_VF_BARLITE_INT (0x108)	0x108	VF BAR information for internal DMA registers.
QDMA_GLBL2_PF_BARLITE_EXT (0x10C)	0x10C	PF BAR information for External AXI-Lite Master.
QDMA_GLBL2_PF_VF_BARLITE_EXT (0x110)	0x110	VF BAR information for External AXI-Lite Master.
QDMA_GLBL2_CHANNEL_INST (0x114)	0x114	DMA channel instantiations.
QDMA_GLBL2_CHANNEL_MDMA (0x118)	0x118	DMA channel QDMA mode.
QDMA_GLBL2_CHANNEL_STRM (0x11C)	0x11C	DMA channel stream mode.
QDMA_GLBL2_CHANNEL_QDMA_CAP (0x120)	0x120	QDMA config settings.
QDMA_GLBL2_CHANNEL_PASID_CAP (0x128)	0x128	Pasid Capability.
QDMA_GLBL2_CHANNEL_FUNC_RET (0x12C)	0x12C	Function Return.
QDMA_GLBL2_SYSTEM_ID (0x130)	0x130	System ID.
QDMA_GLBL2_MISC_CAP (0x134)	0x134	Misc Capabilities.
QDMA_GLBL2_DBG_PCIE_RQ0 (0x1B8)	0x1B8	RQ interface debug information.
QDMA_GLBL2_DBG_PCIE_RQ1 (0x1BC)	0x1BC	RQ interface debug information.
QDMA_GLBL2_DBG_AXIMM_WR0 (0x1C0)	0x1C0	DMA AXIMM interface debug information.
QDMA_GLBL2_DBG_AXIMM_WR1 (0x1C4)	0x1C4	DMA AXIMM interface debug information.
QDMA_GLBL2_DBG_AXIMM_RD0 (0x1C8)	0x1C8	DMA AXIMM interface debug information.
QDMA_GLBL2_DBG_AXIMM_RD1 (0x1CC)	0x1CC	DMA AXIMM interface debug information.

QDMA_GLBL2_IDENTIFER (0x100)

Table 87: **QDMA_GLBL2_IDENTIFIER (0x100)**

Bit	Default	Access Type	Field	Description
[31:8]	24’h1fd700	RO	identifier	identifier
[7:0]	8’h0	RO	version	version

QDMA_GLBL2_PF_BARLITE_INT (0x104)

Table 88: **QDMA_GLBL2_PF_BARLITE_INT (0x104)**

Bit	Default	Access Type	Field	Description
[23:18]		RO	pf3_bar_map[5:0]	Pf3_bar_map consists of 6 bits – one bit for each bar. A one in the bar’s bit position indicates that requests which hit this bar will be routed to dma registers. The corresponding bit should not be set in both this register and QDMA_GLBL2_PF_BARLITE_EXT register. If neither register redirects the request, the request will be send to the Bridge AXI-MM Master interface
[17:12]		RO	pf2_bar_map[5:0]	See description for pf3_bar_map.
[11:6]		RO	pf1_bar_map[5:0]	See description for pf3_bar_map.
[5:0]		RO	pf0_bar_map[5:0]	See description for pf3_bar_map.

QDMA_GLBL2_PF_VF_BARLITE_INT (0x108)

Table 89: **QDMA_GLBL2_PF_VF_BARLITE_INT (0x108)**

Bit	Default	Access Type	Field	Description
[23:18]		RO	pf3_vf_bar_map[5:0]	Pf3_vf_bar_map consists of 6 bits – one bit for each VF bar of PF3. A one in the bar’s bit position indicates that requests which hit this bar will be routed to dma registers. The corresponding bit should not be set in both this register and QDMA_GLBL2_PF_BARLITE_EXT register. If neither register redirects the request, the request will be send to the Bridge AXI-MM Master interface.
[17:12]		RO	pf2_vf_bar_map[5:0]	See description for pf3_bar_map.
[11:6]		RO	pf1_vf_bar_map[5:0]	See description for pf3_bar_map.
[5:0]		RO	pf0_vf_bar_map[5:0]	See description for pf3_bar_map.

QDMA_GLBL2_PF_BARLITE_EXT (0x10C)

Table 90: **QDMA_GLBL2_PF_BARLITE_EXT (0x10C)**

Bit	Default	Access Type	Field	Description
[23:18]		RO	pf3_bar_map[5:0]	Pf3_bar_map consists of 6 bits – one bit for each bar. A one in the bar's bit position indicates that requests which hit this bar will be routed to the Bridge AXI-Lite Master interface. The corresponding bit should not be set in both this register and QDMA_GLBL2_PF_BARLITE_INT register. If neither register redirects the request, the request will be send to the Bridge AXI-MM Master interface
[17:12]		RO	pf2_bar_map[5:0]	See description for pf3_bar_map.
[11:6]		RO	pf1_bar_map[5:0]	See description for pf3_bar_map.
[5:0]		RO	pf0_bar_map[5:0]	See description for pf3_bar_map.

QDMA_GLBL2_PF_VF_BARLITE_EXT (0x110)

Table 91: **QDMA_GLBL2_PF_VF_BARLITE_EXT (0x110)**

Bit	Default	Access Type	Field	Description
[23:18]		RO	pf3_vf_bar_map[5:0]	Pf3_vf_bar_map consists of 6 bits – one bit for each VF bar of PF3. A one in the bar's bit position indicates that requests which hit this bar will be routed to the Bridge AXI-Lite Master interface. The corresponding bit should not be set in both this register and QDMA_GLBL2_PF_BARLITE_INT register. If neither register redirects the request, the request will be send to the Bridge AXI-MM Master interface.
[17:12]		RO	pf2_vf_bar_map[5:0]	See description for pf3_vf_bar_map.
[11:6]		RO	pf1_vf_bar_map[5:0]	See description for pf3_vf_bar_map.
[5:0]		RO	pf0_vf_bar_map[5:0]	See description for pf3_vf_bar_map.

QDMA_GLBL2_CHANNEL_INST (0x114)

Table 92: **QDMA_GLBL2_CHANNEL_INST (0x114)**

Bit	Default	Access Type	Field	Description
[31:18]			Reserved	Reserved
[17]		RO	c2h_st	A one indicates the C2H ST engine is instantiated
[16]		RO	h2c_st	A one indicates the H2C ST engine is instantiated
[15:9]			Reserved	Reserved
[8]		RO	c2h_eng[0]	A one indicates the C2H MM engine is instantiated
[7:1]			Reserved	Reserved
[0]		RO	h2c_eng[0]	A one indicates the H2C MM engine is instantiated

QDMA_GLBL2_CHANNEL_MDMA (0x118)

Table 93: **QDMA_GLBL2_CHANNEL_MDMA (0x118)**

Bit	Default	Access Type	Field	Description
[31:18]			Reserved	Reserved
[17]		RO	c2h_st	A one indicates the C2H ST engine is QDMA
[16]		RO	h2c_st	A one indicates the H2C ST engine is QDMA
[15:9]			Reserved	Reserved
[8]		RO	c2h_eng[0]	A one indicates the C2H MM engine is QDMA
[7:1]			Reserved	Reserved
[0]		RO	h2c_eng[0]	A one indicates the H2C MM engine is QDMA

QDMA_GLBL2_CHANNEL_STRM (0x11C)

Table 94: **QDMA_GLBL2_CHANNEL_STRM (0x11C)**

Bit	Default	Access Type	Field	Description
[31:18]			Reserved	Reserved
[17]		RO	c2h_st	A one indicates it is a Stream mode dma engine.
[16]		RO	h2c_st	A one indicates it is a Stream mode dma engine.
[15:9]			Reserved	Reserved
[8]		RO	c2h_eng[0]	A one indicates it is a Memory-Mapped mode dma engine.
[7:1]			Reserved	Reserved
[0]		RO	h2c_eng[0]	A one indicates it is a Memory-Mapped mode dma engine.

QDMA_GLBL2_CHANNEL_QDMA_CAP (0x120)

Table 95: **QDMA_GLBL2_CHANNEL_QDMA_CAP (0x120)**

Bit	Default	Access Type	Field	Description
[31:12]			Reserved	Reserved
[11:0]		RO	multq_max	The number of queues supported – 1.

QDMA_GLBL2_CHANNEL_PASID_CAP (0x128)

Table 96: **QDMA_GLBL2_CHANNEL_PASID_CAP (0x128)**

Bit	Default	Access Type	Field	Description
[31:16]			Reserved	Reserved
[15:4]		RO	bridge_pasid_offset[11:0]	Pasid table offset for bridge slave requests. The PASID table entry used is determined by adding the function number of the requests to the PASID table offset.
[3:2]			Reserved	Reserved
[1]		RO	bridge_pasid_en	A one indicates that the Bridge slave requests are PASID capable.
[0]		RO	dma_pasid_en	A one indicates that the DMA requests are PASID capable.

QDMA_GLBL2_CHANNEL_FUNC_RET (0x12C)

Table 97: **QDMA_GLBL2_CHANNEL_FUNC_RET (0x12C)**

Bit	Default	Access Type	Field	Description
[31:8]			Reserved	Reserved
[7:0]		RO	function[7:0]	Returns the completer function number of the register read.

QDMA_GLBL2_SYSTEM_ID (0x130)

Table 98: **QDMA_GLBL2_SYSTEM_ID (0x130)**

Bit	Default	Access Type	Field	Description
[31:8]			Reserved	Reserved
[15:0]		RO	system_id[15:0]	Returns the system_id attribute/parameter

QDMA_GLBL2_MISC_CAP (0x134)

Table 99: **QDMA_GLBL2_MISC_CAP (0x134)**

Bit	Default	Access Type	Field	Description
[31:0]		RO	Reserved	Miscellaneous capabilities advertised in attributes

QDMA_GLBL2_DBG_PCIE_RQ0 (0x1B8)

Table 100: **QDMA_GLBL2_DBG_PCIE_RQ0 (0x1B8)**

Bit	Default	Access Type	Field	Description
[31:20]		RO	nph_avl[11:0]	NPH credits available
[19:10]		RO	rcb_avl[9:0]	RCB credits available (32B granularity).
[9:4]		RO	slv_rd_credits[5:0]	Bridge slave read ordering credits
[3:2]		RO	tag_ep[1:0]	Tag pool empty status
[1:0]		RO	tag_fl[1:0]	Tag pool full status

QDMA_GLBL2_DBG_PCIE_RQ1 (0x1BC)

Table 101: **QDMA_GLBL2_DBG_PCIE_RQ1 (0x1BC)**

Bit	Default	Access Type	Field	Description
[31:17]				Reserved
[16]		RO	wtlp_req	Wtlp req
[15]]		RO	wtlp_header_fifo_fl	Wtlp header fifo_fl
[14]		RO	wtlp_header_fifo_ep	Wtlp header fifo ep
[13]		RO	rq_fifo_ep	rq fifo empty
[12]		RO	rq_fifo_fl	rq fifo full
[11:9]		RO	tlpsm[2:0]	tlp state
[8:6]		RO	tlpsm512[2:0]	tlp512 state
[5]		RO	rreq0_rcb_ok	Read request slot0 has sufficient RCB
[4]		RO	rreq0_slv	Read request slot0 is slave request
[3]		RO	rreq0_vld	Read request slot0 pending
[2]		RO	rreq1_rcb_ok	Read request slot1 has sufficient RCB
[1]		RO	rreq1_slv	Read request slot1 is slave request
[0]		RO	rreq1_vld	Read request slot1 pending

QDMA_GLBL2_DBG_AXIMM_WR0 (0x1C0)

Table 102: **QDMA_GLBL2_DBG_AXIMM_WR0 (0x1C0)**

Bit	Default	Access Type	Field	Description
[31:27]				Reserved
[26]		RO	wr_req	wr_req
[25:23]		RO	wr_chn[2:0]	wr_chn
[22]		RO	wtlp_dat_fifo_ep	wtlp_dat_fifo_ep
[21]		RO	wpl_fifo_ep	wpl_fifo_ep

Table 102: **QDMA_GLBL2_DBG_AXIMM_WR0 (0x1C0)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[20:18]		RO	brsp_claim_chnl[2:0]	brsp_claim_chnl
[17:12]		RO	wrreq_cnt[5:0]	wrreq_cnt
[11:9]		RO	bid[2:0]	bid
[8]		RO	bvalid	bvalid
[7]		RO	bready	bready
[6]		RO	wvalid	wvalid
[5]		RO	wready	wready
[4:2]		RO	awid[2:0]	awid
[1]		RO	awvalid	awvalid
[0]		RO	awready	awready

QDMA_GLBL2_DBG_AXIMM_WR1 (0x1C4)

Table 103: **QDMA_GLBL2_DBG_AXIMM_WR1 (0x1C4)**

Bit	Default	Access Type	Field	Description
[31:30]				Reserved
[29:24]		RO	brsp_cnt4[5:0]	brspcnt4
[23:18]		RO	brsp_cnt3[5:0]	brspcnt3
[17:12]		RO	brsp_cnt2[5:0]	brspcnt2
[11:6]		RO	brsp_cnt1[5:0]	brspcnt1
[5:0]		RO	brsp_cnt0[5:0]	brspcnt0

QDMA_GLBL2_DBG_AXIMM_RD0 (0x1C8)

Table 104: **QDMA_GLBL2_DBG_AXIMM_RD0 (0x1C8)**

Bit	Default	Access Type	Field	Description
[31:23]				Reserved
[22:17]		RO	pnd_cnt[5:0]	pnd_cnt
[16:14]		RO	rd_chnl[2:0]	rd_chnl
[13]		RO	rd_req	rd_req
[12:10]		RO	rrsp_claim_chnl[2:0]	rrsp_claim_chnl
[9:7]		RO	rid[2:0]	rid
[6]		RO	rvalid	rvalid
[5]		RO	rready	rready
[4:2]		RO	arid[2:0]	arid
[1]		RO	arvalid	arvalid

Table 104: **QDMA_GLBL2_DBG_AXIMM_RD0 (0x1C8)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[0]		RO	arready	arready

QDMA_GLBL2_DBG_AXIMM_RD1 (0x1CC)

Table 105: **QDMA_GLBL2_DBG_AXIMM_RD1 (0x1CC)**

Bit	Default	Access Type	Field	Description
[31:30]				Reserved
[29:24]		RO	rrsp_cnt4[5:0]	rrspcnt4
[23:18]		RO	rrsp_cnt3[5:0]	rrspcnt3
[17:12]		RO	rrsp_cnt2[5:0]	rrspcnt2
[11:6]		RO	rrsp_cnt1[5:0]	rrspcnt1
[5:0]		RO	rrsp_cnt0[5:0]	rrspcnt0

QDMA_TRQ_SEL_GLBL (0x00200)

Table 106: **QDMA_TRQ_SEL_GLBL (0x00200) Register Space**

Registers (Address)	Address	Description
QDMA_GLBL_RNG_SZ (0x204-0x240)	0x204-0x240	Global ring size registers. 16 different ring size can be set
QDMA_GLBL_ERR_STAT (0X248)	0x248	Global Error status
QDMA_GLBL_ERR_MASK (0X24C)	0x24C	Global Error mask enable
QDMA_GLBL_DSC_CFG (0x250)	0x250	Descriptor configuration and C2H completion accumulation
QDMA_GLBL_DSC_ERR_STS (0x254)	0x254	Descriptor Error status bits
QDMA_GLBL_DSC_ERR_MSK (0x258)	0x258	Descriptor Error mask enable
QDMA_GLBL_DSC_ERR_LOG0 (0x25C)	0x25C	Descriptor Error information
QDMA_GLBL_DSC_ERR_LOG1 (0x260)	0x260	Descriptor Type of error
QDMA_GLBL_TRQ_ERR_STS (0x264)	0x264	Address Target Error status
QDMA_GLBL_TRQ_ERR_MSK (0x268)	0x268	Address Target Error mask enable
QDMA_GLBL_TRQ_ERR_LOG (0x26C)	0x26C	Address Target Error information
QDMA_GLBL_DSC_DBG_DAT0 (0x270)	0x270	Descriptor engine debug info
QDMA_GLBL_DSC_DBG_DAT1 (0x274)	0x274	Descriptor engine debug info

QDMA_GLBL_RNG_SZ (0x204-0x240)

Table 107: **QDMA_GLBL_RNG_SZ (0x204-0x240)**

Bit	Default	Access Type	Field	Description
31:16	16'h0	NA		Reserved
15:0	NA	RW	Ring_size	Ring Size (including Write back status location)

Global ring size is a group of 16 registers that is used by the descriptor and completion context to select its ring size via the ring size index field.

Address = 0x200 + ((index + 1) *4)
For index=0, Ring Size Register 0 is located at address 0x204
For index=1, Ring Size Register 1 is located at address 0x208

These registers must be written before using them. There are no reset values for these registers.

QDMA_GLBL_ERR_STAT (0X248)

Table 108: **QDMA_GLBL_ERR_STAT (0X248)**

Bit	Default	Access Type	Field	Description
[31:12]	0	NA	Reserved	Reserved
11	0	RW1C	err_h2c_st	Indicates an error was encountered by H2C-ST.
10	0	RW1C	err_bdg	Indicates an error was encountered by the Bridge.
9	0	RW1C	ind_ctxt_cmd_err	
8	0	RW1C	err_c2h_st	Indicates an error was encountered by C2H-ST.
7	0	RW1C	err_c2h_mm_1	Indicates an error was encountered by C2H-MM Channel1.
6	0	RW1C	err_c2h_mm_0	Indicates an error was encountered by C2H-MM Channel0.
5	0	RW1C	err_h2c_mm_1	Indicates an error was encountered by H2C-MM Channel1.
4	0	RW1C	err_h2c_mm_0	Indicates an error was encountered by H2C-MM Channel0.
3	0	RW1C	err_trq	
2	0	RW1C	err_dsc	
1	0	RW1C	err_ram_dbe	
0	0	RW1C	err_ram_sbe	

QDMA_GLBL_ERR_MASK (0X24C)

Table 109: **QDMA_GLBL_ERR_MASK (0X24C)**

Bit	Default	Access Type	Field	Description
[31:9]	0		Reserved	Reserved
[8:0]	0	RW	mask	Output error enable mask. See QDMA_GLBL_ERR_STAT_A definition

QDMA_GLBL_DSC_CFG (0x250)

Table 110: **QDMA_GLBL_DSC_CFG (0x250)**

Bit	Default	Access Type	Field	Description
[31:10]	0	NA	Reserved	Reserved
[9]	0	RW	Unc_ovr_cor	Uncorrectable log overwrite correctable
[8]	0	RW	ctxt_fer_dis	Log both dsc and dma error bit in context, not just first
[7:6]	0	NA	Reserved	Reserved
[5:3]	6	RW	Max_dsc_fetch	Max number of descriptors to fetch in one request. 8 * 2^val // Max value is 6
[2:0]	0	RW	Wb_acc_int	Completion accumulation interval if completions are enabled for a queue configured for internal mode. 3'h0: 4 3'h1: 8 3'h2: 16 3'h3: 32 3'h4: 64 3'h5: 128 3'h6: 256 3'h7: 512

Completion accumulation can be disabled using queue context settings.

QDMA_GLBL_DSC_ERR_STS (0x254)

Table 111: **QDMA_GLBL_DSC_ERR_STS (0x254)**

Bit	Default	Access Type	Field	Description
[31:25]	0		Reserved	Reserved
[24]	0	RW1C	sbe	COR_ERR_ RAM_SBE
[23]	0	RW1C	dbe	UNC_ERR_ _RAM_DBE
[22]	0	RW1C	rq_cancel	Descriptor fetch was cancelled in DMA due to disable register status.
[20]	0	RW1C	dma	UNC_ERR_DMA. Dma engine reported an error.

Table 111: **QDMA_GLBL_DSC_ERR_STS (0x254)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[21]	0	RW2C	dsc	Invalid PIDX update.
[19]	0	RW1C	flr_cancel	Descriptor fetch was cancelled in DMA due to FLR
[18:17]	0		Reserved	Reserved
[16]	0	RW1C	dat_poison	Descriptor fetch completion contained poison data
[9]	0	RW1C	timeout	Descriptor fetch completion timed out
[5]	0	RW1C	flr	Descriptor fetch completion had flr error.
[4]	0	RW1C	tag	Descriptor fetch completion had unexpected tag.
[3]	0	RW1C	addr	Descriptor fetch completion had address mismatch
[2]	0	RW1C	param	Descriptor fetch completion had parameter mismatch.
[1]	0	RW1C	ur_ca	Descriptor fetch completion had unsupported request or completer abort status.
[0]	0	RW1C	poison	Descriptor fetch completion had header poison status.

QDMA_GLBL_DSC_ERR_MSK (0x258)

Table 112: **QDMA_GLBL_DSC_ERR_MSK (0x258)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW	mask	Error logging enable masks. See QDMA_GLBL_DSC_ERR_STS_A .

QDMA_GLBL_DSC_ERR_LOG0 (0x25C)

Table 113: **QDMA_GLBL_DSC_ERR_LOG0 (0x25C)**

Bit	Default	Access Type	Field	Description
[31]	0	RW	valid	Error logs are valid
[30:29]			Reserved	Reserved
[28:17]	0	RW	qid	Queue id of error
[16]	0	RW	sel	DMA direction of error 0: H2C 1: C2H
[15:0]	0	RW	cidx	Consumer index of error

QDMA_GLBL_DSC_ERR_LOG1 (0x260)

Table 114: **QDMA_GLBL_DSC_ERR_LOG1 (0x260)**

Bit	Default	Access Type	Field	Description
[31:9]	0	RW	Reserved	Reserved
[3:0]	0	RW	sub_type	Error sub-type. For update_err only. 0: non update_err 2: PIDX update overflow. Too many descriptors posted compared to ring size.
[4:0]	0	RW	err_type	Error type. If QMDA_GLBL_DSC_ERR_LOG0 valid is set, this indicates which unmasked error happened first and the error type in the status register that is recorded in the logs.

QDMA_GLBL_TRQ_ERR_STS (0x264)

Table 115: **QDMA_GLBL_TRQ_ERR_STS (0x264)**

Bit	Default	Access Type	Field	Description
[31:3]	0	NA	Reserved	Reserved
[3]	0	RW1C	tcp_timeout	Timeout on request to dma internal register.
[2]	0	RW1C	vf_access_err	A VF attempted to access Global register space or Function map.
[1]	0	RW1C	qid_range	A function attempted to access a qid beyond the queues allocated to it in the function map RAM.
[0]	0	RW1C	unmapped	Access targeted unmapped register space.

QDMA_GLBL_TRQ_ERR_MSK (0x268)

Table 116: **QDMA_GLBL_TRQ_ERR_MSK (0x268)**

Bit	Default	Access Type	Field	Description
[31:0]	0	NA	mask	Enable logging mask. See QDMA_GLBL_TRQ_ERR_STS definition.

QDMA_GLBL_TRQ_ERR_LOG (0x26C)

Table 117: **QDMA_GLBL_TRQ_ERR_LOG (0x26C)**

Bit	Default	Access Type	Field	Description
[31:28]	0	NA	Reserved	Reserved

Table 117: **QDMA_GLBL_TRQ_ERR_LOG (0x26C)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[27:24]	0	RW	target	The target of the register access. 0 : Reserved 1 : QDMA_TRQ_SEL_GLBL1 2: QDMA_TRQ_SEL_GLBL2 3: QDMA_TRQ_SEL_GLBL 4: QDMA_TRQ_SEL_FMAP 5: QDMA_TRQ_SEL_IND 6: QDMA_TRQ_SEL_C2H 7: Reserved 8: Reserved 9: QDMA_TRQ_SEL_C2H_MM0 10: Reserved 11: QDMA_TRQ_SEL_H2C_MM0 12: Reserved 13: QDMA_TRQ_SEL_QUEUE_PF
[23:16]	0	RW	function	Register access space function
[15:0]	0	RW	address	Register access space address

QDMA_GLBL_DSC_DBG_DAT0 (0x270)

Table 118: **QDMA_GLBL_DSC_DBG_DAT0 (0x270)**

Bit	Default	Access Type	Field	Description
[31:30]	0	NA	Reserved	Reserved
[29]	0	RO	ctxt_arb_dir	DMA direction of arbitration request for descriptor queue context. Use QDMA_DSC_DBG_CTL to select which arbiter source is read.
[28:17]	0	RO	ctxt_arb_qid[11:0]	Qid of arbitration request for descriptor queue context. Use QDMA_DSC_DBG_CTL to select which arbiter source is read.
[16:12]	0	RO	ctxt_arb_req[4:0]	Vector of ctxt arbitration requesters. Bit position map: EVT_SRC =0, TRQ_SRC =1, WBC_SRC=2, CRD_SRC=3, IND_SRC=4
[11]	0	RO	irq_fifo_fl	Immediate Irq fifo is full
[10]	0	RO	tm_dsc_stall	Tm_dsc_sts output is backpressured.
[9:8]	0	RO	rrq_stall[1:0]	Bit1: C2H read request stall Bit0: H2C read request stall
[7:6]	0	RO	rcp_fifo_spc_stall[1:0]	Bit1: C2H read completion space stall Bit0: H2C read completion space stall
[5:4]	0	RO	rrq_fifo_spc_stall[1:0]	Bit1: C2H read request fifo space stall Bit0: H2C read request fifo space stall
[3:2]	0	RO	fab_mrkr_rsp_stall[1:0]	Bit1: C2H mrkr_rsp stall Bit0: H2C mrkr_rsp_stall

Table 118: **QDMA_GLBL_DSC_DBG_DAT0 (0x270)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[1:0]	0	RO	dsc_out_stall[1:0]	Bit 1: C2H descriptor bypass out stall (vld && ~rdy) Bit 0: H2C descriptor bypass out stall (vld && ~rdy)

QDMA_GLBL_DSC_DBG_DAT1 (0x274)

Table 119: **QDMA_GLBL_DSC_DBG_DAT1 (0x274)**

Bit	Default	Access Type	Field	Description
[31:28]	0	NA	Reserved	Reserved
[27:22]	0	RO	evt_spc_c2h[5:0]	Event space for C2H.
[21:16]	0	RO	evt_spc_h2c[5:0]	Event space for H2C.
[15:8]	0x80	RO	dsc_spc_c2h[7:0]	Descriptor fetch completion RAM space for C2H.
[7:0]	x80	RO	dsc_spc_h2c[7:0]	Descriptor fetch completion RAM space for H2C.

QDMA_TRQ_SEL_FMAP (0x00400)

Table 120: **QDMA_TRQ_SEL_FMAP (0x00400) Register Space**

Registers (Address)	Address	description
QDMA_TRQ_SEL_FMAP (0x400-0x7FC)	0x400 -0x7FC	Function map

QDMA_TRQ_SEL_FMAP (0x400-0x7FC)

Function map is used to map a consecutive block of queue(s) to a function. This can be done from any PF.

Table 121: **QDMA_TRQ_SEL_FMAP (0x400-0x7FC)**

Bit	Default	Access Type	Field	Description
[31:23]	0	NA	Reserved	Reserved
[22:11]	0	RW	Qid_max	Queue count
[10:0]	0	RW	Qid_base	Number of Queues for a function

Register address for each function is calculated as 0x400+(Function number *4):

- function number 0, will be written to address 0x400
- function number 1, will be written to address 0x404, etc.
- last function number will be written to address 0x7FC

QDMA_TRQ_SEL_IND (0x00800)

Table 122: **QDMA_TRQ_SEL_IND (0x00800) Register Space**

Registers (Address)	Address	Description
QDMA_IND_CTXT_DATA_3 (0x804)	0x804	Context data (refer to individual context structure)
QDMA_IND_CTXT_DATA_2 (0x808)	0x808	Context data (refer to individual context structure)
QDMA_IND_CTXT_DATA_1 (0x80C)	0x80C	Context data (refer to individual context structure)
QDMA_IND_CTXT_DATA_0 (0x810)	0x810	Context data (refer to individual context structure)
QDMA_IND_CTXT_MASK_3 (0x814)	0x814	Write enable mask
QDMA_IND_CTXT_MASK_2 (0x818)	0x818	Write enable mask
QDMA_IND_CTXT_MASK_1 (0x81C)	0x81C	Write enable mask
QDMA_IND_CTXT_MASK_0 (0x820)	0x820	Write enable mask
QDMA_IND_CTXT_CMD (0x824)	0x824	Context Command

QDMA_IND_CTXT_DATA_3 (0x804)

Table 123: **QDMA_IND_CTXT_DATA_3 (0x804)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW	data	Context data [31:0]

QDMA_IND_CTXT_DATA_2 (0x808)

Table 124: **QDMA_IND_CTXT_DATA_2 (0x808)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW	data	Context data [63:32]

QDMA_IND_CTXT_DATA_1 (0x80C)

Table 125: **QDMA_IND_CTXT_DATA_1 (0x80C)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW	data	Context data [95:64]

QDMA_IND_CTXT_DATA_0 (0x810)

Table 126: **QDMA_IND_CTXT_DATA_0 (0x810)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW	data	Context data [127:96]

All 4 registers (0x804, 0x808, 0x80C, 0x810) constitute context data for a given queue.

QDMA_IND_CTXT_MASK_3 (0x814)

Set the mask to write corresponding data bits. Data masking is only supported on the software descriptor context.

Table 127: **QDMA_IND_CTXT_MASK_3 (0x814)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW	mask	Context Mask [127:96]

QDMA_IND_CTXT_MASK_2 (0x818)

Set the mask to write corresponding data bits. Data masking is only supported on the software descriptor context.

Table 128: **QDMA_IND_CTXT_MASK_2 (0x818)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW	mask	Context Mask [95:64]

QDMA_IND_CTXT_MASK_1 (0x81C)

Set the mask to write corresponding data bits. Data masking is only supported on the software descriptor context.

Table 129: **QDMA_IND_CTXT_MASK_1 (0x81C)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW	mask	Context Mask [63:32]

QDMA_IND_CTXT_MASK_0 (0x820)

Set the mask to write corresponding data bits. Data masking is only supported on the software descriptor context.

Table 130: **QDMA_IND_CTXT_MASK_0 (0x820)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW	mask	Context Mask [31:0]

All 4 registers (0x814, 0x818, 0x81C, 0x820) constitute context mask for a given queue.

QDMA_IND_CTXT_CMD (0x824)

Table 131: **QDMA_IND_CTXT_CMD (0x824)**

Bit	Default	Access Type	Field	Description
[31:18]	0	NA	Reserved	Reserved
[17:7]	0	RW	qid	Queue ID for context
[6:5]	0	RW	op	Opcode 2'h0 = QDMA_CTXT_CMD_CLR 2'h1 = QDMA_CTXT_CMD_WR 2'h2 = QDMA_CTXT_CMD_RD 2'h3 = QDMA_CTXT_CMD_INV
[4:1]	0	RW	sel	4'h0 = QDMA_CTXT_SELC_DEC_SW_C2H 4'h1 = QDMA_CTXT_SELC_DEC_SW_H2C 4'h2 = QDMA_CTXT_SELC_DEC_HW_C2H 4'h3 = QDMA_CTXT_SELC_DEC_HW_H2C 4'h4 = QDMA_CTXT_SELC_DEC_CR_C2H 4'h5 = QDMA_CTXT_SELC_DEC_CR_H2C 4'h6 = QDMA_CTXT_SELC_WRB 4'h7 = QDMA_CTXT_SELC_PFTCH 4'h8 = QDMA_CTXT_SELC_INT_COAL 4'h9 = Reserved 4'hA = Reserved 4'hB = QDMA_CTXT_SELC_TIMER 4'hC = QDMA_CTXT_SELC_INT_QID2VEC
[0]	0	RO	busy	Write will be dropped when busy = 1 Read data is invalid when busy = 1

QDMA_TRQ_SEL_C2H (0x00A00)

Table 132: **QDMA_TRQ_SEL_C2H (0x00A00) Register Space**

Registers (Address)	Address	Description
QDMA_C2H_TIMER_CNT[16] (0xA00-0xA3C)	0xA00-0xA3C	CMPT timer threshold indirection table.
QDMA_C2H_CNT_TH[16] (0xA40-0xA7C)	0xA40-0xA7C	CMPT counter threshold indirection table.
QDMA_C2H_QID2VEC_MAP_QID (0xA80)	0xA80	The Queue ID index of the Qid to Vec RAM
QDMA_C2H_QID2VEC_MAP (0xA84)	0xA84	Map Queue ID to Vector
QDMA_C2H_STAT_S_AXIS_C2H_ACCEPTED (0XA88)	0xA88	Debug status register. Number of C2H packet accepted
QDMA_C2H_STAT_S_AXIS_WRB_ACCEPTED (0xA8C)	0xA8C	Debug status register. Number of C2H WRB packet accepted
QDMA_C2H_STAT_DESC_RSP_PKT_ACCEPTED (0xA90)	0xA90	Debug status register. Number of desc_rsp packet accepted from the Prefetch
QDMA_C2H_STAT_AXIS_PKG_CMP (0xA94)	0xA94	Debug status register. Number of axis packet completed from the C2H DMA Write Engine
QDMA_C2H_STAT_DESC_RSP_ACCEPTED (0xA98)	0xA98	Debug status register. Number of desc_rsp accepted including drop and error from the Prefetch
QDMA_C2H_STAT_DESC_RSP_CMP (0xA9C)	0xA9C	Debug status register. Number of desc_rsp completed including drop and error in the C2H DMA Write Engine
QDMA_C2H_STAT_WRQ_OUT (0xAA0)	0xAA0	Debug status register. Number of WRQ driven from the C2H DMA Write Engine
QDMA_C2H_STAT_WPL_REN_ACCEPTED (0xAA4)	0xAA4	Debug status register. Number of WPL REN accepted in the C2H DMA Write Engine
QDMA_C2H_STAT_TOTAL_WRQ_LEN (0xAA8)	0xAA8	Debug status register. Number of total WRQ length (including the empty packets) from the C2H DMA Write Engine
QDMA_C2H_STAT_TOTAL_WPL_LEN (0xAAC)	0xAAC	Debug status register. Number of total WPL length (including the empty packets) from the C2H DMA Write Engine
QDMA_C2H_BUF_SZ[16] (0xAB0-0xAEC)	0xAB0-0xAEC	Buffer size choices
QDMA_C2H_ERR_STAT (0xAF0)	0xAF0	C2H error status
QDMA_C2H_ERR_MASK (0xAF4)	0xAF4	C2H error enable mask
QDMA_C2H_FATAL_ERR_STAT (0xAF8)	0xAF8	C2H fatal error status
QDMA_C2H_FATAL_ERR_MASK (0xAFC)	0xAFC	C2H fatal error enable mask
QDMA_C2H_FATAL_ERR_ENABLE (0xB00)	0xB00	Enable the C2H fatal error action process
QDMA_GLBL_ERR_INT (0xB04)	0xB04	C2H error generated interrupt
QDMA_C2H_PFCH_CFG (0xB08)	0xB08	Prefetch configuration
QDMA_C2H_INT_TIMER_TICK (0xB0C)	0xB0C	C2H interrupt timer tick
QDMA_C2H_STAT_DESC_RSP_DROP_ACCEPTED (0xB10)	0xB10	Debug status register. Number of dsc rsp with drop accepted
QDMA_C2H_STAT_DESC_RSP_ERR_ACCEPTED (0xB14)	0xB14	Debug status register. Number of dsc rsp with error accepted
QDMA_C2H_STAT_DESC_REQ (0xB18)	0xB18	Debug status register. Number of dsc request sent out from the C2H DMA Write Engine
QDMA_C2H_STAT_DEBUG_DMA_ENG_0 (0xB1C)	0xB1C	Debug registers 0

Table 132: **QDMA_TRQ_SEL_C2H (0x00A00) Register Space** *(cont'd)*

Registers (Address)	Address	Description
QDMA_C2H_STAT_DEBUG_DMA_ENG_1 (0xB20)	0xB20	Debug registers 1
QDMA_C2H_STAT_DEBUG_DMA_ENG_2 (0xB24)	0xB24	Debug registers 2
QDMA_C2H_STAT_DEBUG_DMA_ENG_3 (0xB28)	0xB28	Debug registers 3
QDMA_C2H_DBG_PFCH_ERR_CTXT (0xB2C)	0xB2C	Debug status register.
QDMA_C2H_FIRST_ERR_QID (0xB30)	0xB30	The Qid of the first C2H error
QDMA_STAT_NUM_WRB_IN (0xB34)	0xB34	Debug status register. Number of WRB passed from DmaWrEnginre to Wrb block
QDMA_STAT_NUM_WRB_OUT (0xB38)	0xB38	Debug status register. Number of WRB(excluding STAT_DESC) passed from Wrb to WrbCoal block
QDMA_STAT_NUM_WRB_DRP (0xB3C)	0xB3C	Debug status register. Number of WRB dropped inside Wrb block
QDMA_STAT_NUM_STAT_DESC_OUT (0xB40)	0xB40	Debug status register. Number of STAT_DESC issued from Wrb to WrbCoal block
QDMA_STAT_NUM_DSC_CRDT_SENT (0xB44)	0xB44	Debug status register. An accounting of the number of descriptor credits sent out v/s received(as a result of q invalidations)
QDMA_STAT_NUM_FCH_DSC_RCVD (0xB48)	0xB48	Debug status register. Number of descriptors received from the fetch engine
QDMA_STAT_NUM_BYP_DSC_RCVD (0XB4C)	0xB4C	Debug status register. Number of descriptors received from the bypass path
QDMA_C2H_WRB_COAL_CFG (0xB50)	0xB50	C2H completion coalesce configuration
QDMA_C2H_INTR_H2C_REQ (0xB54)	0xB54	Debug status register. Number of H2C interrupt requests
QDMA_C2H_INTR_C2H_MM_REQ (0xB58)	0xB58	Debug status register. Number of C2H MM interrupt requests
QDMA_C2H_INTR_ERR_INT_REQ (0xB5C)	0xB5C	Debug status register. Number of error generated interrupt requests
QDMA_C2H_INTR_C2H_ST_REQ (0xB60)	0xB60	Debug status register. Number of C2H stream interrupt requests
QDMA_C2H_INTR_H2C_ERR_C2H_MM_MSIX_ACK (0xB64)	0xB64	Debug status register. Number of msix Ack for the H2C, C2H MM, and error generated interrupts
QDMA_C2H_INTR_H2C_ERR_C2H_MM_MSIX_FAIL (0xB68)	0xB68	Debug status register. Number of msix Fail for the H2C, C2H MM, and error generated interrupts
QDMA_C2H_INTR_H2C_ERR_C2H_MM_MSIX_NO_MSIX (0xB6C)	0xB6C	Debug status register. Number of no msix for the H2C, C2H MM, and error generated interrupts
QDMA_C2H_INTR_H2C_ERR_C2H_MM_CTXT_INVAL (0xB70)	0xB70	Debug status registers. Number of invalid Interrupt Ring cases for the H2C, C2H MM, and error generated interrupts
QDMA_C2H_INTR_C2H_ST_MSIX_ACK (0xB74)	0xB74	Debug status register. Number of msix Ack for the C2H stream interrupt
QDMA_C2H_INTR_C2H_ST_MSIX_FAIL (0xB78)	0xB78	Debug status register. Number of msix Fail for the C2H stream interrupt
QDMA_C2H_INTR_C2H_ST_NO_MSIX (0xB7C)	0xB7C	Debug status register. Number of no msix for the C2H stream interrupt

Table 132: **QDMA_TRQ_SEL_C2H (0x00A00) Register Space** *(cont'd)*

Registers (Address)	Address	Description
QDMA_C2H_INTR_C2H_ST_CTXT_INVAL (0xB80)	0xB80	Debug status register. Number of invalid Interrupt Ring cases for the C2H stream interrupt
QDMA_C2H_STAT_WR_CMP (0xB84)	0xB84	Debug status register. Number of payload write completion from the DMA Write Engine
QDMA_C2H_STAT_DEBUG_DMA_ENG_4 (0xB88)	0xB88	Debug register in DMA Write Engine
QDMA_C2H_STAT_DEBUG_DMA_ENG_5 (0xB8C)	0xB8C	Debug register in DMA Write Engine
QDMA_C2H_DBG_PFCH_QID (0xB90)	0xB90	Debug register in Prefetch module
QDMA_C2H_DBG_PFCH (0xB94)	0xB94	Debug register in Prefetch module
QDMA_C2H_INT_DEBUG (0xB98)	0xB98	Debug register in Interrupt module
QDMA_C2H_STAT_IMM_ACCEPTED (0xB9C)	0xB9C	Debug status register. Number of immediate data packets accepted
QDMA_C2H_STAT_MARKER_ACCEPTED (0xBA0)	0xBA0	Debug status register. Number of marker packets accepted
QDMA_C2H_STAT_DISABLE_CMP_ACCEPTED (0xBA4)	0xBA4	Debug status register. Number of disable completion packets accepted
QDMA_C2H_PAYLOAD_FIFO_CRDT_CNT (0xBA8)	0xBA8	Debug status register. Number of payload FIFO credit count in the DMA Write Engine.

QDMA_C2H_TIMER_CNT[16] (0xA00-0xA3C)

Table 133: **QDMA_C2H_TIMER_CNT[16] (0xA00-0xA3C)**

Bit	Default	Access Type	Field	Description
[31:8]	0	NA	Reserved	Reserved
[7:0]	0	RW	Timer_count	Timer threshold

Timer Threshold is a group of 16 registers that is used by the C2H completion context to select its timer value using the timer count index field.

QDMA_C2H_CNT_TH[16] (0xA40-0xA7C)

Table 134: **QDMA_C2H_CNT_TH[16] (0xA40-0xA7C)**

Bit	Default	Access Type	Field	Description
[31:8]	0	NA	Reserved	Reserved
[7:0]	0	RW	Threshold_count	Count threshold

Count Threshold is a group of 16 registers that is used by the C2H completion context to select its count threshold using the count threshold index field.

QDMA_C2H_QID2VEC_MAP_QID (0xA80)

Table 135: **QDMA_C2H_QID2VEC_MAP_QID (0xA80)**

Bit	Default	Access Type	Field	Description
[31:11]	0	NA	Reserved	Reserved
[10:0]	0	RW	Qid	Qid

QDMA_C2H_QID2VEC_MAP (0xA84)

Table 136: **QDMA_C2H_QID2VEC_MAP (0xA84)**

Bit	Default	Access Type	Field	Description
[31:19]	0	NA	Reserved	Reserved
[18]	0	RW	H2c_en_coal	Enable the H2C interrupt aggregation
[17:9]	0	RW	H2c_vector	H2C Vector
[8]	0	RW	C2h_en_coal	Enable the C2H interrupt aggregation
[7:0]	0	RW	C2h_vector	C2H Vector

The interrupt Qid to Vector mapping is used to map each queue to the respective interrupt vector (max 32) or an interrupt aggregation ring index (max 256) if the interrupt aggregation is enabled.

The SW first writes to the QDMA_C2H_QID2VEC_MAP_QID register to indicate the Qid. Then it can read or write to the QDMA_C2H_QID2VEC_MAP register to write or read the information of the vector mapping and the interrupt aggregation enable.

QDMA_C2H_STAT_S_AXIS_C2H_ACCEPTED (0XA88)

Table 137: **QDMA_C2H_STAT_S_AXIS_C2H_ACCEPTED (0XA88)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RO	C2h_accepted	Number of C2H packet accepted from the user application.

QDMA_C2H_STAT_S_AXIS_WRB_ACCEPTED (0xA8C)

Table 138: **QDMA_C2H_STAT_S_AXIS_WRB_ACCEPTED (0xA8C)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RO	Wrb_accepted	Number of C2H write back packet accepted from the user application.

QDMA_C2H_STAT_DESC_RSP_PKT_ACCEPTED (0xA90)

Table 139: **QDMA_C2H_STAT_DESC_RSP_PKT_ACCEPTED (0xA90)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RO	Dsc_rsp_pkt_accepted	Number of desc_rsp packet accepted.

QDMA_C2H_STAT_AXIS_PKG_CMP (0xA94)

Table 140: **QDMA_C2H_STAT_AXIS_PKG_CMP (0xA94)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RO	Pkg_cmp	The number of C2H packets completed from the C2H DMA write engine.

QDMA_C2H_STAT_DESC_RSP_ACCEPTED (0xA98)

Table 141: **QDMA_C2H_STAT_DESC_RSP_ACCEPTED (0xA98)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RO	Dsc_rsp_accepted	The number of desc_rsp accepted including drop and error.

QDMA_C2H_STAT_DESC_RSP_CMP (0xA9C)

Table 142: **QDMA_C2H_STAT_DESC_RSP_CMP (0xA9C)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RO	Dsc_rsp_cmp	The number of desc_rsp completed, including drop and error in the C2H DMA write engine.

QDMA_C2H_STAT_WRQ_OUT (0xAA0)

Table 143: **QDMA_C2H_STAT_WRQ_OUT (0xAA0)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RO	Wrq_out	The number of WRQ driven from the C2H DMA write engine.

QDMA_C2H_STAT_WPL_REN_ACCEPTED (0xAA4)

Table 144: **QDMA_C2H_STAT_WPL_REN_ACCEPTED (0xAA4)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RO	Wpl_ren_accepted	The number of WPL REN accepted in the C2H DMA write engine.

QDMA_C2H_STAT_TOTAL_WRQ_LEN (0xAA8)

Table 145: **QDMA_C2H_STAT_TOTAL_WRQ_LEN (0xAA8)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RO	Total_wrq_len	The number of total WRQ length (including the empty packets) from the C2H DMA write engine.

QDMA_C2H_STAT_TOTAL_WPL_LEN (0xAAC)

Table 146: **QDMA_C2H_STAT_TOTAL_WPL_LEN (0xAAC)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RO	Total_wpl_len	The number of total WPL length (including the empty packets) from the C2H DMA write engine.

QDMA_C2H_BUF_SZ[16] (0xAB0-0xAEC)

Table 147: **QDMA_C2H_BUF_SZ[16] (0xAB0-0xAEC)**

Bit	Default	Access Type	Field	Description
[31:16	0	NA	Rsvd	Reserved
[15:0]	0	RW	Size	C2H Buffer size for each descriptor in a given queue. (max 64K-1)

There are 16 registers which can have different C2H buffer sizes. Buffer selection can be done in context programming.

QDMA_C2H_ERR_STAT (0xAF0)

Table 148: **QDMA_C2H_ERR_STAT (0xAF0)**

Bit	Default	Access Type	Field	Description
[31:13]	0	NA	Rsvd	Reserved
[15]	0	RW	wrb_prty_err	Parity error detected on the C2H Completion.
[14]	0	RW	wrb_cidx_err	A bad CIDX update was sent by the SW to the C2H-ST Completion engine.
[13]	0	RW	wrb_qfull_err	The completion queue gets full.
[12]	0	RW	wrb_inv_q_err	This error is flagged when the SW sends a CMPT CIDX update to an invalid queue.
[11]	0	RW	port_id_byp_in_mismatch	Port_id from the C2H packet and the Port_id from the bypass_in do not match.
[10]	0	RW	port_id_ctxt_mismatch	Port_id from the C2H packet and the Port_id in the Prefetch context do not match.
[9]	0	RW	err_desc_cnt	Flag the error if the number of the descriptors in a packet is larger than 7.
[8]	0	RW	Rsvd	Reserved
[7]	0	RW	msi_int_fail	The msix interrupt message got a FAIL response.
[6]	0	RW	eng_wpl_data_par_err	Data parity error
[5]	0	RW	Rsvd	Reserved
[4]	0	RW	desc_rsp_err	The descriptor has error bit set.
[3]	0	RW	qid_mismatch	Flag the error if the Qid from the s_axis_c2h_ctrl.qid do not match the Qid on the s_axis_wrb_data.
[2]	0	RW	Rsvd3	Reserved
[1]	0	RW	len_mismatch	Flag the error if the total packet length do not match the signal from the s_axis_c2h_ctrl.len
[0]	0	RW	mty_mismatch	The Mty should be 0 if it is not the last packet. Flag the error if it is not the case.

This is the error logging register for the C2H errors. The HW writes to the register when the error happens. The SW can write `1'b1` to clear the error if it wants to. The QDMA_C2H_ERR_MASK register doesn't affect the error logging.

QDMA_C2H_ERR_MASK (0xAF4)

Table 149: **QDMA_C2H_ERR_MASK (0xAF4)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW	c2h_err_en_mask	C2H error enable mask

The software can set the bit to enable the corresponding C2H error to be propagated to the error aggregator.

QDMA_C2H_FATAL_ERR_STAT (0xAF8)

Table 150: **QDMA_C2H_FATAL_ERR_STAT (0xAF8)**

Bit	Default	Access Type	Field	Description
[31:21]	0	RO	Rsvd	Reserved
[18]	0	RO	wpl_data_par_err	Ram double bit error
[17]	0	RO	payload_fifo_ram_rdbe	Ram double bit error
[16]	0	RO	qid_fifo_ram_rdbe	Ram double bit error
[15]	0	RO	tuser_fifo_ram_rdbe	Ram double bit error
[14]	0	RO	wrb_coal_data_ram_rdbe	Ram double bit error
[13]	0	RO	int_qid2vec_ram_rdbe	Ram double bit error
[12]	0	RO	int_ctxt_ram_rdbe	Ram double bit error
[11]	0	RO	desc_req_fifo_ram_rdbe	Ram double bit error
[10]	0	RO	pfch_ctxt_ram_rdbe	Ram double bit error
[9]	0	RO	wrb_ctxt_ram_rdbe	Ram double bit error
[8]	0	RO	pfch_ll_ram_rdbe	Ram double bit error
[7:4]	0	RO	timer_fifo_ram_rdbe	Ram double bit error
[3]	0	RO	Qid_mismatch	Flag the error if the Qid from the s_axis_c2h_ctrl.qid doesn't match the Qid on the s_axis_wrb_data
[2]	0	RO	Rsvd	Reserved
[1]	0	RO	Len_mismatch	Flag the error if the total packet length doesn't match the signal from the s_axis_c2h_ctrl.len
[0]	0	RO	Mty_mismatch	The Mty should be 0 if it is not the last packet. Flag the error if it is not the case

This is the error logging register for the C2H fatal errors. The HW writes to the register when the error happens. The software can write 1'b1 to clear the error if it wants to. The QDMA_C2H_FATAL_ERR_MASK register does not affect the error logging.

QDMA_C2H_FATAL_ERR_MASK (0xAFC)

Table 151: **QDMA_C2H_FATAL_ERR_MASK (0xAFC)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW	c2h_fatal_err_en_mask	C2H fatal error enable mask

The software can set the bit to enable the corresponding C2H fatal error to be sent to the C2H fatal error handling logic.

QDMA_C2H_FATAL_ERR_ENABLE (0xB00)

Table 152: **QDMA_C2H_FATAL_ERR_ENABLE (0xB00)**

Bit	Default	Access Type	Field	Description
[31:2]	0	RW	Reserved	
[1]	0	RW	Enable_wpl_par_inv	Enable the C2H Wpl parity inversion when fatal error happens
[0]	0	RW	Enable_wrq_dis	Enable the C2H Wrq disable when fatal error happens

This register can enable the C2H fatal error handling process.

- Stop the data transfer by disabling the Wrq
- Invert the WPL parity on the data transfer

QDMA_GLBL_ERR_INT (0xB04)

Table 153: **QDMA_GLBL_ERR_INT (0xB04)**

Bit	Default	Access Type	Field	Description
[31:18]	0	NA	Reserved	Reserved
[17]	0	RW	Err_int_arm	The SW sets the bit to arm the error interrupt. The HW clears the bit when the interrupt is taken by the interrupt module. The SW need to re-arm this bit to generate the next error interrupt.
[16]	0	RW	En_coal	1’b1: indirect error interrupt; 1’b0: direct error interrupt
[15:8]	0	RW	Vec	For the direct error interrupt, this is the interrupt vector; For the indirect error interrupt, this is the interrupt aggregation context RAM index
[7:0]	0	RW	Func	Function

This register is for the error generated interrupt.

QDMA_C2H_PFCH_CFG (0xB08)

Table 154: **QDMA_C2H_PFCH_CFG (0B08)**

Bit	Default	Access Type	Field	Description
[31:25]	56	RW	Evt_qcnt_th	Hardware starts eviction when number of prefetch queue count >= evt_qcnt_th; The evc_qcnt_th should always be less than pfch_qcnt. Recommended value is 14.
[24:18]	60	RW	Pfch_qcnt	Max number of prefetch queue count allowed. Maximum value is 60. Recommended value is 16.

Table 154: **QDMA_C2H_PFCH_CFG (0B08)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[17:9]	16	RW	Num_pfch	Controls number of entries prefetched in cache per queue. Recommended value is 8.
[8:0]	16	RW	Pfch_fl_th	Stop prefetch when available free descriptor space <= pfch_fl_th, minimum value is 16. Recommended value is 256.

QDMA_C2H_INT_TIMER_TICK (0xB0C)

Table 155: **QDMA_C2H_INT_TIMER_TICK (0xB0C)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW	Timer_tick	Value of a C2H timer tick in terms of user clock.

QDMA_C2H_STAT_DESC_RSP_DROP_ACCEPTED (0xB10)

Table 156: **QDMA_C2H_STAT_DESC_RSP_DROP_ACCEPTED (0xB10)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RO	dsc_rsp_drop_accepted	Number of desc rsp with drop accepted

QDMA_C2H_STAT_DESC_RSP_ERR_ACCEPTED (0xB14)

Table 157: **QDMA_C2H_STAT_DESC_RSP_ERR_ACCEPTED (0xB14)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RO	dsc_rsp_err_accepted	Number of desc rsp with error accepted

QDMA_C2H_STAT_DESC_REQ (0xB18)

Table 158: **QDMA_C2H_STAT_DESC_REQ (0xB18)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RO	Desc_req	Number of desc request sent out from the C2H DMA Write Engine

QDMA_C2H_STAT_DEBUG_DMA_ENG_0 (0xB1C)

Table 159: **QDMA_C2H_STAT_DEBUG_DMA_ENG_0 (0xB1C)**

Bit	Default	Access Type	Field	Description
[31]	0	NA	Reserved	Reserved
[30:28]	0	RO	wrb_fifo_out_cnt	count of wrb fifo
[27:18]	0	RO	qid_fifo_out_cnt	count of qid fifo
[17:8]	0	RO	payload_fifo_out_cnt	count of payload fifo
[7:5]	0	RO	wrq_fifo_out_cnt	count of wrq fifo
[4]	0	RO	wrb_sm_cs	write back state machine
[3:0]	0	RO	main_sm_cs	main state machine

This is the debug register for the C2H DMA Write Engine.

QDMA_C2H_STAT_DEBUG_DMA_ENG_1 (0xB20)

Table 160: **QDMA_C2H_STAT_DEBUG_DMA_ENG_1 (0xB20)**

Bit	Default	Access Type	Field	Description
[31]	1	NA	tuser_comb_in_rdy	signal 'tuser_comb_in_rdy
[30]	0	RO	desc_rsp_last	desc_rsp_last signal
[29:20]	0	RO	payload_fifo_in_cnt	number of incoming entries to payload fifo
[19:10]	0	RO	payload_fifo_output_cnt	number of popup entries from payload fifo
[9:0]	0	RO	qid_fifo_in_cnt	number of incoming entries to qid fifo

This is the debug register for the C2H DMA Write Engine.

QDMA_C2H_STAT_DEBUG_DMA_ENG_2 (0xB24)

Table 161: **QDMA_C2H_STAT_DEBUG_DMA_ENG_2 (0xB24)**

Bit	Default	Access Type	Field	Description
[31:30]	0	NA	Reserved	Reserved
[29:20]	0	RO	wrb_fifo_in_cnt	number of incoming entries to wrb fifo
[19:10]	0	RO	wrb_fifo_output_cnt	number of popup entries from wrb fifo
[9:0]	0	RO	qid_fifo_output_cnt	number of popup entries from qid fifo

This is the debug register for the C2H DMA Write Engine.

QDMA_C2H_STAT_DEBUG_DMA_ENG_3 (0xB28)

Table 162: **QDMA_C2H_STAT_DEBUG_DMA_ENG_3 (0xB28)**

Bit	Default	Access Type	Field	Description
[31:30]	0	NA	Reserved	Reserved
[29:20]	0	RO	addr_4k_split_cnt	number of cases when it crosses the 4k address boundary
[19:10]	0	RO	wrq_fifo_in_cnt	number of incoming entries to wrq fifo
[9:0]	0	RO	wrq_fifo_output_cnt	number of popup entries from wrq fifo

QDMA_C2H_DBG_PFCH_ERR_CTXT (0xB2C)

Table 163: **QDMA_C2H_DBG_PFCH_ERR_CTXT (0xB2C)**

Bit	Default	Access Type	Field	Description
[31:14]	0	RW	Reserved	Reserved
[13]	0	RW	Err_stat	Error status For read command if Queue is valid, err_stat = 0 If Queue is invalid err_stat = 1
[12]	0	RW	Cmd_wr	Command to write or read. 1: write 0: read
[11:1]	0	RW	Qid	Queue ID.
[0]	0	RW	done	Done. Operation finished

QDMA_C2H_FIRST_ERR_QID (0xB30)

Table 164: **QDMA_C2H_FIRST_ERR_QID (0xB30)**

Bit	Default	Access Type	Field	Description
[31:20]		NA	Reserved	Reserved

Table 164: **QDMA_C2H_FIRST_ERR_QID (0xB30)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[19:16]	0	RO	Err_type	4’b1111: NA 4'b1110: wrb_cidx_err 4’b1101: wrb_qfull_err 4’b1100: wrb_inv_q_err 4’b1011: port_id_ctxt_mismatch 4’b1010: port_id_byp_in_mismatch 4’b1001: err_desc_cnt 4’b0111: msi_int_fail 4’b0110: eng_wpl_data_par_err 4’b0100: desc_rsp_error 4’b0011: qid_mismatch 4’b0001: len_mismatch 4’b0000: mty_mismatch
[15:11]	0	NA	Reserved	
[10:0]	0	RO	Qid	The Qid of the first C2H error

This register records the first C2H error type and Qid. The software can write to this register to clear the `err_type` to be `4’b1111`.

QDMA_STAT_NUM_WRB_IN (0xB34)

Table 165: **QDMA_STAT_NUM_WRB_IN (0xB34)**

Bit	Default	Access Type	Field	Description
[31:16]	0	NA	Reserved	Reserved
[15:0]	0	RO	wrb_cnt	The number of WRB passed from DmaWrEnginre to Wrb block.

QDMA_STAT_NUM_WRB_OUT (0xB38)

Table 166: **QDMA_STAT_NUM_WRB_OUT (0xB38)**

Bit	Default	Access Type	Field	Description
[31:16]	0	NA	Reserved	Reserved
[15:0]	0	RO	wrb_cnt	Number of WRB(excluding STAT_DESC) passed from Wrb to WrbCoal block

QDMA_STAT_NUM_WRB_DRP (0xB3C)

Table 167: **QDMA_STAT_NUM_WRB_DRP (0xB3C)**

Bit	Default	Access Type	Field	Description
[31:16]	0	NA	Reserved	Reserved
[15:0]	0	RO	wrb_cnt	Number of WRB dropped inside Wrb block

QDMA_STAT_NUM_STAT_DESC_OUT (0xB40)

Table 168: **QDMA_STAT_NUM_STAT_DESC_OUT (0xB40)**

Bit	Default	Access Type	Field	Description
[31:16]	0	NA	Reserved	Reserved
[15:0]	0	RO	stat_desc_cnt	Number of STAT_DESC issued from Wrb to WrbCoal block

QDMA_STAT_NUM_DSC_CRDT_SENT (0xB44)

Table 169: **QDMA_STAT_NUM_DSC_CRDT_SENT (0xB44)**

Bit	Default	Access Type	Field	Description
[31:16]	0	NA	Reserved	Reserved
[15:0]	0	RO	crdt_cnt	An accounting of the number of descriptor credits sent out versus received (as a result of q invalidations).

QDMA_STAT_NUM_FCH_DSC_RCVD (0xB48)

Table 170: **QDMA_STAT_NUM_FCH_DSC_RCVD (0xB48)**

Bit	Default	Access Type	Field	Description
[31:16]	0	NA	Reserved	Reserved
[15:0]	0	RO	dsc_cnt	Number of descriptors received from the fetch engine

QDMA_STAT_NUM_BYP_DSC_RCVD (0XB4C)

Table 171: **QDMA_STAT_NUM_BYP_DSC_RCVD (0XB4C)**

Bit	Default	Access Type	Field	Description
[31:11]	0	NA	Reserved	Reserved

Table 171: **QDMA_STAT_NUM_BYP_DSC_RCVD (0XB4C)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[10:0]	0	RO	dsc_cnt	Number of descriptors received from the bypass path

QDMA_C2H_WRB_COAL_CFG (0xB50)

Table 172: **QDMA_C2H_WRB_COAL_CFG (0xB50)**

Bit	Default	Access Type	Field	Description
[31:26]	32	RW	max_buf_sz	To make the design see a certain size of the coalesce buffer.
[25:14]	20	RW	tick_val	Coalesce buffer timer tick value
[13:2]	4	RW	tick_cnt	Coalesce buffer timer count value
[1]	0	RW	set_glb_flush	Makes coalesce buffer flush an entry as soon as it as a Completion in it
[0]	0	RO	done_glb_flush	Coalesce buffer sets this bit when it flushes an entry.

QDMA_C2H_INTR_H2C_REQ (0xB54)

Table 173: **QDMA_C2H_INTR_H2C_REQ (0xB54)**

Bit	Default	Access Type	Field	Description
[31:18]	0	NA	Rsvd	Reserved
[17:0]	0	RO	Cnt	Debug status register. Number of H2C interrupt requests.

QDMA_C2H_INTR_C2H_MM_REQ (0xB58)

Table 174: **QDMA_C2H_INTR_C2H_MM_REQ (0xB58)**

Bit	Default	Access Type	Field	Description
[31:18]	0	NA	Rsvd	Reserved
[17:0]	0	RO	Cnt	Debug status register. Number of C2H MM interrupt requests.

QDMA_C2H_INTR_ERR_INT_REQ (0xB5C)

Table 175: **QDMA_C2H_INTR_ERR_INT_REQ (0xB5C)**

Bit	Default	Access Type	Field	Description
[31:18]	0	NA	Rsvd	Reserved
[17:0]	0	RO	Cnt	Debug status register. Number of error generated interrupt requests

QDMA_C2H_INTR_C2H_ST_REQ (0xB60)

Table 176: **QDMA_C2H_INTR_C2H_ST_REQ (0xB60)**

Bit	Default	Access Type	Field	Description
[31:18]	0	NA	Rsvd	Reserved
[17:0]	0	RO	Cnt	Debug status register. Number of C2H stream interrupt requests.

QDMA_C2H_INTR_H2C_ERR_C2H_MM_MSIX_ACK (0xB64)

Table 177: **QDMA_C2H_INTR_H2C_ERR_C2H_MM_MSIX_ACK (0xB64)**

Bit	Default	Access Type	Field	Description
[31:18]	0	NA	Rsvd	Reserved
[17:0]	0	RO	cnt	Debug status register. Number of msix Ack for the H2C, C2H MM, and error generated interrupts.

QDMA_C2H_INTR_H2C_ERR_C2H_MM_MSIX_FAIL (0xB68)

Table 178: **QDMA_C2H_INTR_H2C_ERR_C2H_MM_MSIX_FAIL (0xB68)**

Bit	Default	Access Type	Field	Description
[31:18]	0	NA	Rsvd	Reserved
[17:0]	0	RO	Cnt	Debug status register. Number of msix Fail for the H2C, C2H MM, and error generated interrupts.

QDMA_C2H_INTR_H2C_ERR_C2H_MM_MSIX_NO_MSIX (0xB6C)

Table 179: **QDMA_C2H_INTR_H2C_ERR_C2H_MM_MSIX_NO_MSIX (0xB6C)**

Bit	Default	Access Type	Field	Description
[31:18]	0	NA	Rsvd	Reserved

Table 179: **QDMA_C2H_INTR_H2C_ERR_C2H_MM_MSIX_NO_MSIX (0xB6C)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[17:0]	0	RO	Cnt	Debug status register. Number of no msix for the H2C, C2H MM, and error generated interrupts.

QDMA_C2H_INTR_H2C_ERR_C2H_MM_CTXT_INVAL (0xB70)

Table 180: **QDMA_C2H_INTR_H2C_ERR_C2H_MM_CTXT_INVAL (0xB70)**

Bit	Default	Access Type	Field	Description
[31:18]	0	NA	Rsvd	Reserved
[17:0]	0	RO	Cnt	Debug status register. Number of Interrupt Context invalid cases for the H2C, C2H MM, and error generated interrupts.

QDMA_C2H_INTR_C2H_ST_MSIX_ACK (0xB74)

Table 181: **QDMA_C2H_INTR_C2H_ST_MSIX_ACK (0xB74)**

Bit	Default	Access Type	Field	Description
[31:18]	0	NA	Rsvd	Reserved
[17:0]	0	RO	Cnt	Debug status register. Number of MSIX Ack for the C2H stream interrupts.

QDMA_C2H_INTR_C2H_ST_MSIX_FAIL (0xB78)

Table 182: **QDMA_C2H_INTR_C2H_ST_MSIX_FAIL (0xB78)**

Bit	Default	Access Type	Field	Description
[31:18]	0	NA	Rsvd	Reserved
[17:0]	0	RO	Cnt	Debug status register. Number of msix Fail for the C2H stream interrupts.

QDMA_C2H_INTR_C2H_ST_NO_MSIX (0xB7C)

Table 183: **QDMA_C2H_INTR_C2H_ST_NO_MSIX (0xB7C)**

Bit	Default	Access Type	Field	Description
[31:18]	0	NA	Rsvd	Reserved
[17:0]	0	RO	Cnt	Debug status register. Number of no msix for the C2H stream interrupts.

QDMA_C2H_INTR_C2H_ST_CTXT_INVAL (0xB80)

Table 184: **QDMA_C2H_INTR_C2H_ST_CTXT_INVAL (0xB80)**

Bit	Default	Access Type	Field	Description
[31:18]	0	NA	Rsvd	Reserved
[17:0]	0	RO	Cnt	Debug status register. Number of Interrupt Context invalid cases for the C2H interrupts.

QDMA_C2H_STAT_WR_CMP (0xB84)

Table 185: **QDMA_C2H_STAT_WR_CMP (0xB84)**

Bit	Default	Access Type	Field	Description
[31:18]	0	NA	Rsvd	Reserved
[17:0]	0	RO	Cnt	Debug status register. Number of payload write completion from the DMA Write Engine.

QDMA_C2H_STAT_DEBUG_DMA_ENG_4 (0xB88)

Table 186: **QDMA_C2H_STAT_DEBUG_DMA_ENG_4 (0xB88)**

Bit	Default	Access Type	Field	Description
[31]	0	NA	tuser_fifo_out_vld	tuser fifo out valid.
[30]	1	RO	wrb_fifo_in_rdy	wrb fifo in rdy signal.
[29:20]	0	RO	tuser_fifo_in_cnt	Number of incoming entries to tuser fifo.
[19:10]	0	RO	tuser_fifo_output_cnt	Number of popup entries from tuser fifo.
[9:0]	0	RO	tuser_fifo_out_cnt	Number of incoming entries to tuser fifo.

QDMA_C2H_STAT_DEBUG_DMA_ENG_5 (0xB8C)

Table 187: **QDMA_C2H_STAT_DEBUG_DMA_ENG_5 (0xB8C)**

Bit	Default	Access Type	Field	Description
[31:25]	0	NA	rsvd	Reserved
[24]	0	RO	tuser_comb_out_vld	tuser comb out valid
[23]	1	RO	tuser_fifo_in_rdy	tuser fifo in rdy signal
[22:13]	0	RO	tuser_comb_in_cnt	Number of incoming entries to tuser comb.
[12:3]	0	RO	tuse_comb_output_cnt	Number of popup entries from tuser comb.
[2:0]	0	RO	tuser_comb_cnt	Number of entries in the tuser comb.

QDMA_C2H_DBG_PFCH_QID (0xB90)

Table 188: **QDMA_C2H_DBG_PFCH_QID (0xB90)**

Bit	Default	Access Type	Field	Description
[31:15]	0	RW	Rsvd	Reserved
[14:14]	0	RW	Err_ctxt	Data written to the Error Context RAM
[13:11]	0	RW	Target	3'h0: Key_cam, 3'h1: Tag_st 3'h2: Tag_used_cnt 3'h3: Tag_desc_cnt 3'h4: Error Context RAM
[10:0]	0	RW	Qid_or_tag	Qid or Tag

QDMA_C2H_DBG_PFCH (0xB94)

Table 189: **QDMA_C2H_DBG_PFCH (0xB94)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW	Data	Data

The above are two debug registers for the Prefetch module. First, write to the QDMA_C2H_DBG_PFCH_QID register to set up the target and qid (or tag). Then, read or write to the QDMA_C2H_DBG_PFCH register to do the following:

- Read Key cam of each tag
- Read tag_st of each tag
- Read tag_used_cnt of each tag
- Read tag_desc_cnt of each tag
- Read or write Error Context RAM of each queue

QDMA_C2H_INT_DEBUG (0xB98)

Table 190: **QDMA_C2H_INT_DEBUG (0xB98)**

Bit	Default	Access Type	Field	Description
[31:8]	0	NA	Rsvd	Reserved
[7:4]	0	RO	Int_coal_sm	State machine in the Interrupt Aggregation module.
[3:0]	0	RO	Int_sm	State machine in the Interrupt module.

QDMA_C2H_STAT_IMM_ACCEPTED (0xB9C)

Table 191: **QDMA_C2H_STAT_IMM_ACCEPTED (0xB9C)**

Bit	Default	Access Type	Field	Description
[31:18]	0	NA	Rsvd	Reserved
[17:0]	0	RO	Cnt	Number of immediate data packets accepted.

QDMA_C2H_STAT_MARKER_ACCEPTED (0xBA0)

Table 192: **QDMA_C2H_STAT_MARKER_ACCEPTED (0xBA0)**

Bit	Default	Access Type	Field	Description
[31:18]	0	NA	Rsvd	Reserved
[17:0]	0	RO	Cnt	Number of marker packets accepted.

QDMA_C2H_STAT_DISABLE_CMP_ACCEPTED (0xBA4)

Table 193: **QDMA_C2H_STAT_DISABLE_CMP_ACCEPTED (0xBA4)**

Bit	Default	Access Type	Field	Description
[31:18]	0	NA	Rsvd	Reserved
[17:0]	0	RO	Cnt	Number of disable completion packets accepted.

QDMA_C2H_PAYLOAD_FIFO_CRDT_CNT (0xBA8)

Table 194: **QDMA_C2H_PAYLOAD_FIFO_CRDT_CNT (0xBA8)**

Bit	Default	Access Type	Field	Description
[31:18]	0	NA	Rsvd	Reserved
[18:0]	0	RO	Cnt	Payload FIFO credit count in the DMA Write Engine.

QDMA_TRQ_SEL_H2C (0x00E00)

Table 195: **QDMA_TRQ_SEL_H2C (0x00E00) Register Space**

Register Name	Address (hex)	Description
QDMA_H2C_ERR_STAT (0xE00)	0xE00	H2C error status
QDMA_H2C_ERR_MASK (0xE04)	0xE04	H2C error mask
QDMA_H2C_FIRST_ERR_QID (0xE08)	0xE08	The QID and type of the first error encountered on H2C-ST

Table 195: **QDMA_TRQ_SEL_H2C (0x00E00) Register Space** *(cont'd)*

Register Name	Address (hex)	Description
QDMA_H2C_DBG_REG0 (0xE0C)	0xE0C	H2C-ST debug register 0
QDMA_H2C_DBG_REG1 (0xE10)	0xE10	H2C-ST debug register 1
QDMA_H2C_DBG_REG2 (0xE14)	0xE14	H2C-ST debug register 2
QDMA_H2C_DBG_REG3 (0xE18)	0xE18	H2C-ST debug register 3
QDMA_H2C_DBG_REG4 (0xE1C)	0xE1C	H2C-ST debug register 4
QDMA_H2C_FATAL_ERR_EN (0xE20)	0xE20	H2C-ST fatal error enable

QDMA_H2C_ERR_STAT (0xE00)

Table 196: **QDMA_H2C_ERR_STAT (0xE00)**

Bit	Default	Access Type	Field	Description
[31:3]	0	NA	rsv	Reserved
[0]	0	RW	zero_len_dsc_err	A zero length descriptor was received with either SOP or EOP reset.
[1]	0	RW	sdi_mrkr_req_mop_err	A non-EOP descriptor was received with either sdi or mrkr_req set.
[2]	0	RW	no_dma_dsc_err	A no_dma descriptor was received with either SOP or EOP reset.
[3]	0	RW	dbe	Doube bit error detected on H2C-ST data.
[4]	0	RW	sbe	Single bit error corrected on H2C-ST data.

This is the error logging register for the H2C errors. The HW writes to the register when the error happens. The SW can write 1'b1 to clear the error if desired. The QDMA_H2C_ERR_MASK register does not affect error logging.

QDMA_H2C_ERR_MASK (0xE04)

Table 197: **QDMA_H2C_ERR_MASK (0xE04)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW	h2c_err_en_mask	H2C error enable mask

The software can set a bit to enable the corresponding H2C error from being propagated to the error aggregator.

QDMA_H2C_FIRST_ERR_QID (0xE08)

Table 198: **QDMA_H2C_FIRST_ERR_QID (0xE08)**

Bit	Default	Access Type	Field	Description
[31:20]		NA	Reserved	Reserved
[19:16]	0	RO	Err_type	4'b1111: NA 4'b0000: zero_len_dsc_err 4'b0001: wbi_mop_err 4'b0010: no_dma_dsc_err 4'b0011: sbe 4'b0100: dbe
[15:11]	0	NA	rsv	Reserved
[10:0]	0	RO	Qid	The Qid of the first H2C error

QDMA_H2C_DBG_REG0 (0xE0C)

Table 199: **QDMA_H2C_DBG_REG0 (0xE0C)**

Bit	Default	Access Type	Field	Description
[31:16]	0	RO	num_dsc_rcvd	Number of descriptors received by the H2C-ST engine.
[15:0]	0	RO	num_wrb_sent	Number of status write packets sent from the H2C-ST engine to the H2C status-write engine to request the descriptor engine to send the status write out to Host.

QDMA_H2C_DBG_REG1 (0xE10)

Table 200: **QDMA_H2C_DBG_REG1 (0xE10)**

Bit	Default	Access Type	Field	Description
[31:16]	0	RO	num_req_sent	Number of PCIe requests sent by the H2C-ST engine.
[15:0]	0	RO	num_cmp_rcvd	Number of PCIe responses received by the H2C-ST engine.

QDMA_H2C_DBG_REG2 (0xE14)

Table 201: **QDMA_H2C_DBG_REG2 (0xE14)**

Bit	Default	Access Type	Field	Description
[31:16]	0	RO	rsv	Reserved
[15:0]	0	RO	num_err_dsc_rcvd	Number of descriptors received with error by the H2C-ST engine.

QDMA_H2C_DBG_REG3 (0xE18)

Table 202: **QDMA_H2C_DBG_REG3 (0xE18)**

Bit	Default	Access Type	Field	Description
[31]	0	RO	Debug	Reserved
[30]	1	RO	dsco_fifo_empty	H2C-ST status write fifo empty.
[29]	0	RO	dsco_fifo_full	H2C-ST status write fifo full.
[28:26]	1	RO	cur_rc_state	H2C-ST data FSM state.
[25:16]	0	RO	rdreq_lines	The number of lines the descriptor being processed in the request FSM of the H2C-ST will fetch.
[15:6]	512	RO	rdata_lines_avail	number of lines available in the H2C-ST data buffer.
[5]	1	RO	pend_fifo_empty	H2C-ST pending request fifo empty.
[4]	0	RO	pend_fifo_full	H2C-ST pending request fifo full.
[3:2]	01	RO	cur_rq_state	H2C-ST request FSM state.
[1]	0	RO	dsci_fifo_full	H2C-ST descriptor in put fifo full.
[0]	1	RO	dsci_fifo_empty	H2C-ST descriptor in put fifo empty.

QDMA_H2C_DBG_REG4 (0xE1C)

Table 203: **QDMA_H2C_DBG_REG4 (0xE1C)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RO	rdreq_addr	The address of the descriptor being processed in the request FSM of the H2C-ST.

QDMA_H2C_FATAL_ERR_EN (0xE20)

Table 204: **QDMA_H2C_FATAL_ERR_EN (0xE20)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RO	rsv	Reserved
[0]	0	RW	h2c_fatal_err_en	If set, the H2C-ST data double bit errors are passed to the user. If reset, they are ignored.

QDMA_H2C_DATA_TRESH (0xE24)

Table 205: **QDMA_H2C_DATA_TRESH (0xE24)**

Bit	Default Value	Access Type	Field	Description
[31:17]	0	RO		Reserved
[16]	0	RW		Throttle enable. Set this bit to enable throttle

Table 205: **QDMA_H2C_DATA_TRESH (0xE24)** *(cont'd)*

Bit	Default Value	Access Type	Field	Description
[15:0]	0x800	RW		Data threshold (in bytes), to start H2C-ST request throttle

QDMA_TRQ_SEL_C2H_MM (0x1000)

Table 206: **QDMA_TRQ_SEL_C2H_MM (0x1000) Register Space**

Registers	Address	Description
C2H MM Control	0x1004	Channel control bits.
	0x1008	Channel control bits W1S.
	0x100C	Channel control bits C1S.
C2H MM Status	0x1040	Status bits.
	0x1044	Status clear.
C2H Completed Descriptor Count	0x1048	Completed Descriptor. count
C2H MM Error Code Enable Mask (0x1054)	0x1054	Error masking.
C2H MM Error Code (0x1058)	0x1058	Error code.
C2H MM Error Info (0x105C)	0x105C	Error information.
C2H MM Performance Monitor Control (0x10C0)	0x10C0	Performance monitor control.
C2H MM Performance Monitor Cycle Count0 (0x10C4)	0x10C4	Performance monitor cycle count[31:0].
C2H MM Performance Monitor Cycle Count1 (0x10C8)	0x10C8	Performance monitor cycle count [41:32].
C2H MM Performance Monitor Data Count0 (0x10CC)	0x10CC	Performance monitor data count [31:0].
C2H MM Performance Monitor Data Count1 (0x10D0)	0x10D0	Performance monitor data count [41:32].
C2H MM Debug (0x10E8)	0x10E8	Debug info.

C2H MM Control

Table 207: **C2H Channel Control (0x1004)**

Bit	Default	Access Type	Field	Description
31:1			Reserved	Reserved
0	1’b0	RW	run	run Set to 1 to start the SGDMA engine. Reset to 0 to stop the transfer, if the engine is busy it completes the current descriptor.

Table 208: **C2H Channel Control (0x1008)**

Bit	Default	Access Type	Field	Description
		W1S		Control Bit descriptions are the same as in C2H Channel Control (0x04).

Table 209: **C2H Channel Control (0x100C)**

Bit	Default	Access Type	Field	Description
		W1C		Control Bit descriptions are the same as in C2H Channel Control (0x04).

C2H MM Status

Table 210: **QDMA_C2H MM Status (0x1040)**

Bit	Default	Access Type	Field	Description
[31:1]				Reserved
[0]			sts_bsy	Busy If set, the engine is running.

C2H Completed Descriptor Count

Table 211: **C2H Channel Completed Descriptor Count (0x1048)**

Bit	Default	Access Type	Field	Description
31:0	32'h0	RO	c2h_compl_desc_count	c2h_compl_desc_count The number of completed descriptors update by the engine after completing each descriptor in the list. Reset to 0 on rising edge of Control register, run bit (See C2H Channel Control (0x1004)).

C2H MM Error Code Enable Mask (0x1054)

Table 212: **C2H MM Error Code Enable Mask (0x1054)**

Bit	Default	Access Type	Field	Description
[31]	0	RW		Reserved
[30]	0	RW	wr_uc_ram	If set, enables Write error, RAM uncorrectable, and error code logging.

Table 212: **C2H MM Error Code Enable Mask (0x1054)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[29]	0	RW	wr_ur	If set, enables Write error, unsupported request error, and code logging.
[28]	0	RW	wr_flr	If set, enables Write error, FLR reset, and error code logging.
[27:2]	0	RW		Reserved
[1]	0	RW	rd_slv_err	If set, enables Read slave error code logging.
[0]	0	RW	wr_slv_err	If set, enables Read decode error logging.

C2H MM Error Code (0x1058)

Table 213: **C2H MM Error Code (0x1058)**

Bit	Default	Access Type	Field	Description
[31:18]	0	RW		Reserved
[17]	0	RW	valid	Error code is valid
[16]	0	RW	rdwr	Read or Write Error 0: Read error 1: Write error
[4:0]	0	RW	error_code	If Write Error: 2: RAM uncorrectable error 1: Unsupported request 0: Function level reset Other bits reserved If Read Error: 1: Slave error 0: Decode error

C2H MM Error Info (0x105C)

Table 214: **C2H MM Error Info (0x105C)**

Bit	Default	Access Type	Field	Description
[31:29]	0	RW		Reserved
[28:17]	0	RW	qid	Queue ID of the descriptor.
[16]	0	RW	dir	Direction of descriptor.
[15:0]	0	RW	cidx	Consumer index of the descriptor.

C2H MM Performance Monitor Control (0x10C0)

Table 215: **C2H MM Performance Monitor Control (0x10C0)**

Bit	Default	Access Type	Field	Description
[31:4]				Reserved
[3]	0	RW	imm_start	Start counters immediately.
[2]	0	RW	run_start	Set to 1 to arm counters. Counters will start when the run bit is asserted.
[1]	0	WO	imm_clear	Clear counter immediately.
[0]	0	RW	run_clear	Clear counters on run bit assertion.

C2H MM Performance Monitor Cycle Count0 (0x10C4)

Table 216: **C2H MM Performance Monitor Cycle Count0 (0x10C4)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RO	cyc_cnt[31:0]	Cycle count. Increments by one for each clock while running.

C2H MM Performance Monitor Cycle Count1 (0x10C8)

Table 217: **C2H MM Performance Monitor Cycle Count1 (0x10C8)**

Bit	Default	Access Type	Field	Description
[31:10]				Reserved.
[9:0]	0	RO	cyc_cnt[41:32]	Cycle count.

C2H MM Performance Monitor Data Count0 (0x10CC)

Table 218: **C2H MM Performance Monitor Data Count0 (0x10CC)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RO	dcnt[31:0]	Data count. Increments by one for each data beat received.

C2H MM Performance Monitor Data Count1 (0x10D0)

Table 219: **C2H MM Performance Monitor Data Count 1(0x10D0)**

Bit	Default	Access Type	Field	Description
[31:10]				Reserved.
[9:0]	0	RO	dcnt[41:32]	Data count.

C2H MM Debug (0x10E8)

Table 220: **C2H MM Debug (0x10E8)**

Bit	Default	Access Type	Field	Description
[31:24]				Reserved.
[23:17]	0	RO	rrq_entries[6:0]	Outstanding requests.
[16:7]	512	RO	dat_fifo_spc[9:0]	Data fifo space.
[6]	0	RO	rd_stall	Read stall.
[5]	0	RO	rrq_fifo_fl	Read fifo full.
[4]	0	RO	wr_stall	Write stall.
[3]	0	RO	wrq_fifo_fl	Write fifo full.
[2]	0	RO	wbk_stall	Writeback stall.
[1]	1	RO	dsc_fifo_ep	Descriptor fifo empty.
[0]	0	RO	dsc_fifo_fl	Descriptor fifo full.

QDMA_TRQ_SEL_H2C_MM (0x1200)

Table 221: **QDMA_TRQ_SEL_H2C_MM (0x1200) Register Space**

Register	Address	Description
H2C MM Control	0x1204	Channel control bits.
	0x1208	Channel control bits W1S.
	0x120C	Channel control bits C1S.
H2C MM Status	0x1240	Status bits.
H2C Completed Descriptor Count	0x1248	Completed descriptor count.
H2C MM Error Code Enable Mask (0x1254)	0x1254	Error masking.
H2C MM Error Code (0x1258)	0x1258	Error code.
H2C MM Error Info (0x125C)	0x125C	Error information.
H2C MM Performance Monitor Control (0x12C0)	0x12C0	Performance monitor control.
H2C MM Performance Monitor Cycle Count0 (0x12C4)	0x12C4	Performance monitor cycle count[31:0].
H2C MM Performance Monitor Cycle Count1 (0x12C8)	0x12C8	Performance monitor cycle count [41:32].

Table 221: **QDMA_TRQ_SEL_H2C_MM (0x1200) Register Space** *(cont'd)*

Register	Address	Description
H2C MM Performance Monitor Data Count0 (0x12CC)	0x12C8	Performance monitor data count[31:0].
H2C MM Performance Monitor Data Count 1(0x12D0)	0x12D0	Performance monitor data count [41:32].
H2C MM Debug (0x12E8)	0x12E8	Debug info.

H2C MM Control

Table 222: **H2C Channel Control (0x04)**

Bit	Default	Access Type	Field	Description
31:1			Reserved	Reserved
0	1'b0	RW	run	run Set to 1 to start the SGDMA engine. Reset to 0 to stop transfer; if the engine is busy it completes the current descriptor.

ie_* register bits are interrupt enabled. When this condition is met and proper interrupt masks are set interrupt will be generated.

Table 223: **H2C Channel Control (0x1208)**

Bit	Default	Access Type	Field	Description
0		W1S		Control Bit descriptions are the same as in H2C Channel Control (0x04).

Table 224: **H2C Channel Control (0x120C)**

Bit	Default	Access Type	Field	Description
0		W1C		Control Bit descriptions are the same as in H2C Channel Control (0x04).

H2C MM Status

Table 225: **H2C Channel Status (0x240)**

Bit	Default	Access Type	Field	Description
31:1				Reserved

Table 225: **H2C Channel Status (0x240)** *(cont'd)*

Bit	Default	Access Type	Field	Description
0	1'b0	RO	busy	busy Set if the SGDMA engine is busy. Zero when it is idle.

H2C Completed Descriptor Count

Table 226: **H2C Channel Completed Descriptor Count (0x1248)**

Bit	Default	Access Type	Field	Description
31:0	32'h0	RO	h2c_compl_desc_count	The number of competed descriptors update by the engine after completing each descriptor in the list. Reset to 0 on rising edge of Control register Run bit. See H2C Channel Control (0x1204).

H2C MM Error Code Enable Mask (0x1254)

Table 227: **H2C MM Error Code Enable Mask (0x1254)**

Bit	Default	Access Type	Field	Description
[31:30]		RW		Reserved.
[29]	0	RW	wr_slv_error	If set, enables write slave error code logging.
[28]	0	RW	wr_dec_err	If set, enables write decode error code logging.
[27:23]	0	RW		Reserved.
[22]	0	RW	rd_rq_dis_err	If set, enables read rq disable error code logging.
[21:17]	0	RW		Reserved.
[16]	0	RW	rd_dat_poison_err	If set, enables read data poison error code logging.
[15:9]	0	RW		Reserved.
[8]	0	RW	rd_flr_err	If set, enables read flr error code logging.
[7:6]	0	RW		
[5]	0	RW	rd_hdr_adr_err	If set, enables read completion header address mismatch error code logging.
[4]	0	RW	rd_hdr_param_err	If set, enables read completion header param mismatch error code logging.
[3]	0	RW	rd_hdr_byte _err	If set, enables read completion header byte count mismatch error code logging.
[2]	0	RW	rd_ur_ca	If set, enables read completion unsupported request or completer abort error code logging.
[1]	0	RW	rd_hrd_poison_err	If set, enables read completion header poison error code logging.
[0]	0	RW		Reserved.

H2C MM Error Code (0x1258)

Table 228: **H2C MM Error Code (0x1258)**

Bit	Default	Access Type	Field	Description
[31:17]	0	RW		Reserved.
[17]	0	RW	valid	Error log is valid.
[16]	0	RW	rdwr	Read or Write Error. 0: Read error 1: Write error
[4:0]	0	RW	error_code	If Read Error: 1: Header poisoned 2: Unsupported request or Completer Abort 3: Header byte count mismatch 4: Header param mismatch 5: Header address mismatch 8: Function level reset 16 : Data poisoned 22: PCIe reads disabled Other bits reserved If Write Error: 1: Slave error 0: Decode error

H2C MM Error Info (0x125C)

Table 229: **H2C MM Error Info (0x125C)**

Bit	Default	Access Type	Field	Description
[31:29]	0	RW		Reserved.
[28:17]	0	RW	qid	Queue ID of the descriptor.
[16]	0	RW	dir	Direction of descriptor.
[15:0]	0	RW	cidx	Consumer index of the descriptor.

H2C MM Performance Monitor Control (0x12C0)

Table 230: **H2C MM Performance Monitor Control (0x12C0)**

Bit	Default	Access Type	Field	Description
[31:4]				Reserved.
[3]	0	RW	imm_start	Start counters immediately.
[2]	0	RW	run_start	Set to 1 to arm counters. Counters start when the run bit is asserted.
[1]	0	WO	imm_clear	Clear counter immediately.

Table 230: **H2C MM Performance Monitor Control (0x12C0)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[0]	0	RW	run_clear	Clear counters on run bit assertion.

H2C MM Performance Monitor Cycle Count0 (0x12C4)

Table 231: **H2C MM Performance Monitor Cycle Count0 (0x12C4)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RO	cyc_cnt[31:0]	Cycle count. Increments by one for each clock while running.

H2C MM Performance Monitor Cycle Count1 (0x12C8)

Table 232: **H2C MM Performance Monitor Cycle Count1 (0x12C8)**

Bit	Default	Access Type	Field	Description
[31:10]				Reserved.
[9:0]	0	RO	cyc_cnt[41:32]	Cycle count.

H2C MM Performance Monitor Data Count0 (0x12CC)

Table 233: **H2C MM Performance Monitor Data Count0 (0x12CC)**

Bit	Default	Access Type	Fields	Description
[31:0]	0	RO	dcnt[31:0]	Data count. Increments by one for each data beat received.

H2C MM Performance Monitor Data Count 1(0x12D0)

Table 234: **H2C MM Performance Monitor Data Count 1(0x12D0)**

Bit	Default	Access Type	Field	Description
[31:10]				Reserved.
[9:0]	0	RO	dcnt[41:32]	Data count.

H2C MM Debug (0x12E8)

Table 235: **H2C MM Debug (0x12E8)**

Bit	Default	Access Type	Field	Description
[31:24]				Reserved.
[23:17]	0	RO	rrq_entries[6:0]	Outstanding requests.
[16:7]	512	RO	dat_fifo_spc[9:0]	Data fifo space.
[6]	0	RO	rd_stall	Read stall.
[5]	0	RO	rrq_fifo_fl	Read fifo full.
[4]	0	RO	wr_stall	Write stall.
[3]	0	RO	wrq_fifo_fl	Write fifo full.
[2]	0	RO	wbk_stall	Writeback stall.
[1]	1	RO	dsc_fifo_ep	Descriptor fifo empty.
[0]	0	RO	dsc_fifo_fl	Descriptor fifo full.

QDMA_TRQ_MSIX (0x1400)

QDMA_PF_MAILBOX (0x2400)

Table 236: **QDMA_PF_MAILBOX (0x2400) Register Space**

Register	Address	Description
Function Status Register (0x2400)	0x2400	Status bits
Function Command Register (0x2404)	0x2404	Command register bits
Function Interrupt Vector Register (0x2408)	0x2408	Interrupt vector register
Target Function Register (0x240C)	0x240C	Target Function register
Function Interrupt Vector Register (0x2410)	0x2410	Interrupt Control Register
PF Acknowledge Registers (0x2420-0x243C)	0x2420-0x243C	PF acknowledge
FLR Control/Status Register (0x2500)	0x2500	FLR control and status
Incoming Message Memory (0x2800-0x287C)	0x2800-0x287C	Incoming message (128 bytes)
Outgoing Message Memory (0x2C00-0x2C7C)	0x2C00-0x2C7C	Outgoing message (128 bytes)

Mailbox Addressing

PF addressing:

```
Addr = PF_Bar_offset + CSR_addr
```

VF addressing:

```
Addr = VF_Bar_offset + VF_Start_offset + VF_offset + CSR_addr
```

Function Status Register (0x2400)

Table 237: **Function Status Register (0x2400)**

Bit	Default	Access Type	Field	Description
[31:12]	0	NA	Reserved	Reserved
11-4	0	RO	cur_src_fn	This field is for PF use only. The source function number of the message on the top of the incoming request queue.
2	0	RO	ack_status	This field is for PF use only. The status bit will be set when any bit in the acknowledgement status register is asserted.
1	0	RO	o_msg_status	For VF: The status bit will be set when VF driver write msg_send to its command register. When The associated PF driver send acknowledgement to this VF, the hardware clear this field. The VF driver is not allow to update any content in its outgoing mailbox memory (OMM) while o_msg_status is asserted. Any illegal write to the *OMM* will be discarded (optionally, case an error in the AXI4L response channel) For PF: The field indicated the message status of the target FN which is specified in the *Target FN Register*. The status bit will be set when PF driver sends msg_send command. When the corresponding function driver send acknowledgement by sending msg_rcv, the hardware clear this field. The PF driver is not allow to update any content in its outgoing mailbox memory (OMM) while o_msg_status(target_fn_id) is asserted. Any illegal write to the *OMM* will be discarded (optionally, case an error in the AXI4L response channel)
0	0	RO	i_msg_status	For VF: When asserted, a message in the VF's incoming Mailbox memory is pending for process. The field will be cleared once the VF driver write msg_rcv to its command register. For PF: When asserted, the messages in the incoming Mailbox memory are pending for process. The field will be cleared only when the event queue is empty.

Function Command Register (0x2404)

Table 238: **Function Command Register (0x2404)**

Bit	Default	Access Type	Field	Description
[31:3]	0	NA	Reserved	Reserved
2	0	RO	Reserved	Reserved

Table 238: **Function Command Register (0x2404)** *(cont'd)*

Bit	Default	Access Type	Field	Description
1	0	RW	msg_rcv	For VF: VF marks the message in its Incoming Mailbox Memory as received. Hardware asserts the acknowledgement bit of the associated PF. For PF: PF marks the message send by target_fn as received. The hardware will Refresh the i_msg_status of the PF Clear the o_msg_status of the target_fn
0	0	RW	msg_send	For VF: VF marks the current message in its own Outgoing Mailbox as valid. For PF: Current target_fn_id belongs to a VF: PF finished writing a message into the Incoming Mailbox memory of the VF with target_fn_id. The hardware sets the i_msg_status field of the target FN's status register. Current target_fn_id belongs to a PF: PF finished writing a message into its own outgoing Mailbox memory. Hardware will push the message to the event queue of the PF with target_fn_id.

Function Interrupt Vector Register (0x2408)

Table 239: **Function Interrupt Vector Register (0x2408)**

Bit	Default	Access Type	Field	Description
[4:0]	0	RW	int_vect	5-bit interrupt vector assigned by the driver. software

Target Function Register (0x240C)

Table 240: **Target Function Register (0x0C)**

Bit	Default	Access Type	Field	Description
[31:8]	0	NA	Reserved	Reserved
[7:0]	0	RW	target_fn_id	This field is for PF use only. The FN number which the current operation is targeting at.

Function Interrupt Vector Register (0x2410)

Table 241: **Function Interrupt Vector Register (0x2410)**

Bit	Default	Access Type	Field	Description
0	0	RW	int_en	Interrupt enable.

PF Acknowledge Registers (0x2420-0x243C)

Table 242: **PF Acknowledge Registers (0x2420-0x243C)**

Register	Addr	Default	Access Type	Field	Width	Description
Ack0	0x2420	0	RW		32	Acknowledgement from FN 31~0
Ack1	0x2424	0	RW		32	Acknowledgement from FN 63~32
Ack2	0x2428	0	RW		32	Acknowledgement from FN 95~64
Ack3	0x242C	0	RW		32	Acknowledgement from FN 127~96
Ack4	0x2430	0	RW		32	Acknowledgement from FN 159~128
Ack5	0x2434	0	RW		32	Acknowledgement from FN 191~160
Ack6	0x2438	0	RW		32	Acknowledgement from FN 223~192
Ack7	0x243C	0	RW		32	Acknowledgement from FN 255~224

FLR Control/Status Register (0x2500)

Table 243: **FLR Control/Status Register (0x2500)**

Bit	Default	Access Type	Field	Description
[31:1]	0	NA	Reserved	Reserved
0	0	RW	Flr_status	Software write 1 to initiate the Function Level Reset (FLR) for the associated function. The field is kept asserted during the FLR process. Once the FLR is done, the hardware de-asserts this field.

Incoming Message Memory (0x2800-0x287C)

Table 244: **Incoming Message Memory (0x2800-0x287C)**

Register	Addr	Default	Access Type	Field	Width	Description
i_msg_i	0x2800 + i*4	0	RW		32	The *i*th word of the incoming message ($0 \leq I < 128$).

Outgoing Message Memory (0x2C00-0x2C7C)

Table 245: **Outgoing Message Memory (0x2C00-0x2C7C)**

Register	Addr	Default	Access Type	Field	Width	Description
o_msg_i	0x2C00 + i *4	0	RW		32	The *i*th word of the outgoing message ($0 \leq I < 128$).

QDMA_TRQ_SEL_QUEUE_PF (0x6400)

Table 246: **QDMA_TRQ_SEL_QUEUE_PF (0x6400) Register Space**

Register	Address	Description
QDMA_DMAP_SEL_INT_CIDX[2048] (0x6400)	0x6400-0xB3F0	Interrupt Ring Consumer Index (CIDX)
QDMA_DMAP_SEL_H2C_DSC_PIDX[2048] (0x6404)	0x6404-0xB3F4	H2C Descriptor producer index (PIDX)
QDMA_DMAP_SEL_C2H_DSC_PIDX[2048] (0x6408)	0x6408-0xB3F8	C2H Descriptor Producer Index (PIDX)
QDMA_DMAP_SEL_WRB_CIDX[2048] (0x640C)	0x640C-0xB3FC	C2H Write back Consumer Index (CIDX)

There are 2048 Queues, each Queue will have each of above 4 registers. All these registers can be dynamically updated at any point of time. This set of register can be accessed based on the Queue number.

Queue number is absolute *Qnumber* [0 to 2047].
Interrupt CIDX address = 0x6400 + Qnumber*16
H2C PIDX address = 0x6404 + Qnumber*16
C2H PIDX address = 0x6408 + Qnumber*16
Write Back CIDX address = 0x640C + Qnumber*16

For Queue 0:

0x6400 correspond to QDMA_DMAP_SEL_INT_CIDX
0c6404 correspond to QDMA_DMAP_SEL_H2C_DSC_PIDX
0x6408 correspond to QDMA_DMAP_SEL_C2H_DSC_PIDX
0x640C correspond to QDMA_DMAP_SEL_WRB_CIDX

For Queue 1:

0x6410 correspond to QDMA_DMAP_SEL_INT_CIDX
0c6414 correspond to QDMA_DMAP_SEL_H2C_DSC_PIDX
0x6418 correspond to QDMA_DMAP_SEL_C2H_DSC_PIDX
0x641C correspond to QDMA_DMAP_SEL_WRB_CIDX

For Queue 2:

0x6420 correspond to QDMA_DMAP_SEL_INT_CIDX
0c6424 correspond to QDMA_DMAP_SEL_H2C_DSC_PIDX
0x6428 correspond to QDMA_DMAP_SEL_C2H_DSC_PIDX
0x642C correspond to QDMA_DMAP_SEL_WRB_CIDX

QDMA_DMAP_SEL_INT_CIDX[2048] (0x6400)

Table 247: **QDMA_DMAP_SEL_INT_CIDX[2048] (0x6400)**

Bit	Default	Access Type	Field	Description
[31:17]	0	NA	Reserved	Reserved
[16]	0	RW	Sel	1’b0: H2C 1’b1: C2H
[15:0]	0	RW	Sw_cdix	Software Consumer index (CIDX)

QDMA_DMAP_SEL_H2C_DSC_PIDX[2048] (0x6404)

Table 248: **QDMA_DMAP_SEL_H2C_DSC_PIDX[2048] (0x6404)**

Bit	Default	Access Type	Field	Description
[31:17]	0	NA	Reserved	Reserved
[16]	0	RW	irq_en	Interrupt arm, interrupt enable
[15:0]	0	RW	h2c_pidx	H2C Producer Index

QDMA_DMAP_SEL_C2H_DSC_PIDX[2048] (0x6408)

Table 249: **QDMA_DMAP_SEL_C2H_DSC_PIDX[2048] (0x6408)**

Bit	Default	Access Type	Field	Description
[31:17]	0	NA	Reserved	Reserved
[16]	0	RW	irq_en	Interrupt arm, interrupt enable
[15:0]	0	RW	c2h_pidx	C2H Producer Index

QDMA_DMAP_SEL_WRB_CIDX[2048] (0x640C)

Table 250: **QDMA_DMAP_SEL_WRB_CIDX[2048] (0x640C)**

Bit	Default	Access Type	Field	Description
[31:29]	0	NA	Reserved	Reserved
[28]	0	RW	irq_en_wrb	Enable Interrupt for WRB
[27]	0	RW	en_sts_desc_wrb	Enable Status Descriptor for WRB

Table 250: **QDMA_DMAP_SEL_WRB_CIDX[2048] (0x640C)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[26:24]	0	RW	trigger_mode	Interrupt and Status Descriptor Trigger Mode: 0x0: Disabled 0x1: Every 0x2: User_Count 0x3: User 0x4: User_Timer 0x5: User_Timer_Count
[23:20]	0	RW	c2h_timer_cnt_index	Index to QDMA_C2H_TIMER_CNT
[19:16]	0	RW	c2h_count_threshhold	Index to QDMA_C2H_CNT_TH
[15:0]	0	RW	wrb_cidx	Write back Consumer Index (CIDX)

QDMA VF Address Register Space

Table 251: **QDMA VF Address Register Space**

Target Name	Base (Hex)	Byte size (dec)	Notes
QDMA_TRQ_MSIX_VF (0x0000)	00000000	4096	Space for 32 MSIX vectors and PBA
QDMA_VF_MAILBOX (0x1000)	00001000	8192	Mailbox address space
QDMA_TRQ_SEL_QUEUE_VF (0x3000)	00003000	32768	VF Direct QCSR (16B per Q, up to max of 2048 Qs per function)

QDMA_TRQ_MSIX_VF (0x0000)

VF functions can access the MSIX table with offset (0x0000) from that function. The description for this register space is the same as QDMA_TRQ_MSIX (0x1400).

QDMA_VF_MAILBOX (0x1000)

Function Status Register (0x1000)

Table 252: **Function Status Register (0x1000)**

Bit Index	Default	Access Type	Field	Description
[31:12]	0	NA	Reserved	Reserved
11-4	0	RO	cur_src_fn	This field is for PF use only. The source function number of the message on the top of the incoming request queue.
2	0	RO	ack_status	This field is for PF use only. The status bit will be set when any bit in the acknowledgement status register is asserted.

Table 252: **Function Status Register (0x1000)** *(cont'd)*

Bit Index	Default	Access Type	Field	Description
1	0	RO	o_msg_status	For VF: The status bit will be set when VF driver write msg_send to its command register. When the associated PF driver sends acknowledgement to this VF, the hardware clears this field. The VF driver is not allow to update any content in its outgoing mailbox memory (OMM) while o_msg_status is asserted. Any illegal writes to the OMM are discarded (optionally, case an error in the AXI4-Lite response channel). For PF: The field indicated the message status of the target FN which is specified in the Target FN Register. The status bit is set when PF driver sends the msg_send command. When the corresponding function driver sends acknowledgement through msg_rcv, the hardware clears this field. The PF driver is not allow to update any content in its outgoing mailbox memory (OMM) while o_msg_status(target_fn_id) is asserted. Any illegal writes to the OMM are discarded (optionally, case an error in the AXI4L response channel).
0	0	RO	i_msg_status	For VF: When asserted, a message in the VF's incoming Mailbox memory is pending for process. The field is cleared after the VF driver writes msg_rcv to its command register. For PF: When asserted, the messages in the incoming Mailbox memory are pending for process. The field is cleared only when the event queue is empty.

Function Command Register (0x1004)

Table 253: **Function Command Register (0x1004)**

Bit Index	Default	Access Type	Field	Description
[31:3]	0	NA	Reserved	Reserverd
2	0	RO	Reserved	Reserved
1	0	RW	msg_rcv	For VF: VF marks the message in its Incoming Mailbox Memory as received. The hardware asserts the acknowledgement bit of the associated PF. For PF: PF marks the message send by target_fn as received. The hardware refreshes the i_msg_status of the PF, and clears the o_msg_status of the target_fn.
0	0	RW	msg_send	For VF: VF marks the current message in its own Outgoing Mailbox as valid. For PF: Current target_fn_id belongs to a VF: PF finished writing a message into the Incoming Mailbox memory of the VF with target_fn_id. The hardware sets the i_msg_status field of the target FN's status register. Current target_fn_id belongs to a PF: PF finished writing a message into its own outgoing Mailbox memory. The hardware pushes the message to the event queue of the PF with target_fn_id.

Function Interrupt Vector Register (0x1008)

Table 254: **Function Interrupt Vector Register (0x1008)**

Bit Index	Default	Access Type	Field	Description
[4:0]	0	RW	int_vect	5-bit interrupt vector assigned by the driver software.

Target Function Register (0x100C)

Table 255: **Target Function Register (0x100C)**

Bit Index	Default	Access Type	Field	Description
[31:8]	0	NA	Reserved	Reserved
[7:0]	0	RW	target_fn_id	This field is for PF use only. The FN number that the current operation is targeting.

Function Interrupt Vector Register (0x1010)

Table 256: **Function Interrupt Vector Register (0x1010)**

Bit Index	Default	Access Type	Field	Description
0	0	RW	int_en	Interrupt enable.

PF Acknowledge Registers (0x2420-0x243C)

Table 257: **PF Acknowledge Registers (0x2420-0x243C)**

Register	Addr	Default	Access Type	Field	Width	Description
Ack0	0x1020	0	RW		32	Acknowledgement from FN 31~0
Ack1	0x1024	0	RW		32	Acknowledgement from FN 63~32
Ack2	0x1028	0	RW		32	Acknowledgement from FN 95~64
Ack3	0x102c	0	RW		32	Acknowledgement from FN 127~96
Ack4	0x1030	0	RW		32	Acknowledgement from FN 159~128
Ack5	0x1034	0	RW		32	Acknowledgement from FN 191~160
Ack6	0x1038	0	RW		32	Acknowledgement from FN 223~192

Table 257: **PF Acknowledge Registers (0x2420-0x243C)** *(cont'd)*

Register	Addr	Default	Access Type	Field	Width	Description
Ack7	0x103c	0	RW		32	Acknowledgement from FN 255~224

Incoming Message Memory (0x1400-0x147C)

Table 258: **Incoming Message Memory (0x1400-0x147C)**

Register	Addr	Default	Access Type	Field	Width	Description
i_msg_i	0x1400 + i*4	0	RW		32	The *i*th word of the incoming message (i < 128).

Outgoing Message Memory (0x1800-0x187C)

Table 259: **Outgoing Message Memory (0x1800-0x187C)**

Register	Addr	Default	Access Type	Field	Width	Description
o_msg_i	0x1800 + i *4	0	RW		32	The *i*th word of the outgoing message (i < 128).

QDMA_TRQ_SEL_QUEUE_VF (0x3000)

VF functions can access direct update registers per queue with offset (0x3000). The description for this register space is the same as QDMA_TRQ_SEL_QUEUE_PF (0x6400).

These sets of registers can be accessed based on Queue number. And Queue number is absolute Qnumber. [0 to 2047].

Interrupt CIDX address = 0x3000 + Qnumber*16
H2C PIDX address = 0x3004 + Qnumber*16
C3H PIDX address = 0x3008 + Qnumber*16
Write Back CIDX address = 0x300C + Qnumber*16

For Queue 0:

0x3000 correspond to QDMA_DMAP_SEL_INT_CIDX
0x3004 correspond to QDMA_DMAP_SEL_H2C_DSC_PIDX
0x3008 correspond to QDMA_DMAP_SEL_C2H_DSC_PIDX
0x300C correspond to QDMA_DMAP_SEL_WRB_CIDX

For Queue 1:

0x3010 correspond to QDMA_DMAP_SEL_INT_CIDX
0x3014 correspond to QDMA_DMAP_SEL_H2C_DSC_PIDX
0x3018 correspond to QDMA_DMAP_SEL_C2H_DSC_PIDX
0x301C correspond to QDMA_DMAP_SEL_WRB_CIDX

AXI4-Lite Slave Register Space

Through the AXI4-Lite Slave interface, the user can access the Bridge registers and DMA register space.

- **Bridge registers**: When the AXI4-Lite Slave Address bit [28] is set to 0, access to the Bridge and PCIe configuration registers is available.
- **DMA registers**: When AXI4-Lite Slave Address bit [28] is set to 1, access to the DMA registers is available. For more information about the DMA register, see:
  - QDMA PF Address Register Space
  - QDMA VF Address Register Space

Bridge Register Space

Bridge register addresses start at 0xE00. Addresses from 0x00 to 0xE00 are directed to the PCIe Core configuration register space.

Bridge Register Memory Map

Table 260: **Bridge Register Memory Map**

Accessibility	Offset	Contents	Location
RO	0xE00	VSEC Capability	AXI Bridge Defined Memory Mapped register-space
RO	0xE04	VSEC Header	
RO	0xE08	Bridge Info	
R/W	0xE0C	Bridge Status and Control	
R/W	0xE10	Interrupt Decode	
R/W	0xE14	Interrupt Mask	
RO	0xE18	Bus Location	
RO	0xE1C	Physical-Side Interface (PHY) Status/Control	
RO	0xE20 - 0xE34	Reserved Space	
R/W	0xE38	Interrupt Decode 2	
R/W	0xE3C	Interrupt Mask 2	
RO	0xE40	Configuration Control	
RO	0xE44	Slave Error AID	
RO	0xE48 - 0xE54	Reserved Space	
R/W	0xE58	User IRQ Request	
R/W	0xE5C	User IRQ Acknowledge	
R/W	0xE60	PCIe TX Message Control	
R/W	0xE64	PCIe TX Message Header Lower Bit	
R/W	0xE68	PCIe TX Message Header Higher Bit	
R/W	0xE6C	PCIe TX Message Data FIFO	
R/W	0xE70	PCIe RX Message Control and Status	
R/W	0xE74	PCIe RX Message FIFO	
RO	0xE78	Master Pending Counter	
R/W	0xE7C	PCIe TX MSI / MSI-X Control and Status	
RO	0xE80 - 0xED0	Reserved Space	
RO	0xED8	VSEC Capability 2	
RO	0xEDC	VSEC Header 2	
R/W	0xEE0-0xF0C	AXI Base Address Translation Configuration Registers	

VSEC Capability Register (0xE00)

Register to allow the memory space for the core to appear as though it is a part of the underlying integrated block PCIe configuration space. The VSEC is inserted immediately following the last enhanced capability structure.

Table 261: **VSEC Capability Register (0xE00)**

Bit	Default	Access Type	Field	Description
[15:0]	0xB	RO	cap_id	Indicates the PCIe defined ID identifying this Enhanced Capability as a Vendor-Specific capability.
[19:16]	0x1	RO	cap_ver	Indicates the version of this capability structure.
[31:20]	0xED8	RO	nxt_cap_offset	Indicates the offset of the next capability.

VSEC Header Register (0xE04)

Register to provide a unique (within a given vendor) identifier for the layout and contents of the VSEC structure, as well as its revision and length. VSEC Header Register is part of the AXI Bridge that contains main Bridge Registers which start immediately after VSEC Header Register (Offset 0xE08).

Table 262: **VSEC Header Register (0xE04)**

Bit	Default	Access Type	Field	Description
[15:0]	0x0	RO	vsec_id	Indicates the ID value uniquely identifying the nature and format of this VSEC structure.
[19:16]	0x3	RO	vsec_rev	Indicates the version of this capability structure.
[31:20]	0x80	RO	vsec_length	Indicates the length of the entire VSEC Capability structure, in bytes, including the VSEC Capability register.

Bridge Info Register (0xE08)

Register to provide general configuration information about the PCIe AXI Bridge. Information in this register is static and does not change during operation.

Table 263: **Bridge Info Register 0xE08)**

Bit	Default	Access Type	Field	Description
[0]	0	RO	gen2_cap	If set, underlying integrated block supports PCIe Gen2 speed.
[1]	0	RO	rootport_present	Indicates the underlying integrated block is a Root Port when this bit is set. If set, Root Port registers are present in this interface.
[2]	0	RO	upconfig_cap	Indicates the underlying integrated block is upconfig capable when this bit is set.
[3]	0	RO	gen3_cap	If set, underlying integrated block supports PCIe Gen3 speed.
[4]	0	RO	gen4_cap	If set, underlying integrated block supports PCIe Gen4 speed.
[31:5]	0	RO	reserved	reserved

Bridge Control and Status Register (0xE0C)

Register to provide information about the current state of the PCIe AXI Bridge.

Table 264: **Bridge Control and Status Register (0xE0C)**

Bit	Default	Access Type	Field	Description
[7:0]	0	RO	reserved	reserved
[8]	0	RW	glb_int_dis	When set, disables interrupt line from being asserted. Does not prevent Bit in Interrupt Decode register from being set.
[9]	0	RW	cfg_space_en	When set, enables PCIe to generate regular completions (non-CRS) in response to Configuration requests. Otherwise, PCIe returns CRS. This control bit is only valid when the attribute "cfg_space_delay_en" is set to 1. (Only applicable to End Point)
[31:10]	0	RO	reserved	reserved

Interrupt Decode Register (0xE10)

Register to determine what is causing the interrupt line[0] to be asserted and how to clear the interrupt. Writing a 1'b1 to any bit clears that bit except for the Correctable, Non-Fatal, and Fatal bits which require Error FIFO being empty first.

Table 265: **Interrupt Decode Register (0xE10)**

Bit	Default	Access Type	Field	Description
[0]	0	RW1C	link_down	Indicates that Link-Up on the PCI Express link was lost. Not asserted unless link-up had previously been seen.
[1]	0	RW1C	sw_ctrl_int	Indicates a Software Interrupt (from Host and etc.) was set in the DMA registers
[2]	0	RW1C	flr_is_hit	Indicates a Slave Transaction hitting FLR
[3]	0	RW1C	hot_reset	Indicates a Hot Reset was detected.
[17:4]	0	RO	reserved	Reserved
[18]	0	RW1C	vdm_rcvd	Indicates a VDM message was received. The message should be read from the RX_MFIFO_READ register.
[19]	0	RW1C	pme_turn_off_rcvd	Indicates a pme_turn_off message was received. (Only applicable to End Point.)
[20]	0	RW1C	slv_ur	Indicates that a completion TLP was received with a status of 3'b001 - Unsupported Request.
[21]	0	RW1C	slv_tz_violation	Indicates a TrustZone violation was detected on the Bridge Slave port. Violated AXI Request ID is logged in the Slave Error AID register.
[22]	0	RW1C	slv_cpl_timeout	Indicates that the expected completion TLP(s) for a read request for PCIe was not returned within the time period selected by the C_COMP_TIMEOUT parameter.

Table 265: **Interrupt Decode Register (0xE10)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[23]	0	RW1C	slv_err_poison	Indicates the error poison (EP) bit was set in a completion TLP.
[24]	0	RW1C	slv_ca	Indicates that a completion TLP was received with a status of 3'b100 - Completer Abort.
[25]	0	RW1C	slv_illegal_burst	Indicates that a burst type other than INCR was requested by the AXI master.
[26]	0	RW1C	mst_decerr	Indicates a Decoder Error (DECERR) response was received.
[27]	0	RW1C	mst_slverr	Indicates a Slave Error (SLVERR) response was received.
[28]	0	RW1C	slv_pcie_timeout	Indicates that a pcie timeout completion was received.
[29]	0	RW1C	ecc_parity_err_rcvd	Indicates that a RAM ECC Error or Parity Error was received. The source of error should be read from bit[9:0] of the Interrupt Decode 2 register.
[30]	0	RW1C	pcie_local_err_rcvd	Indicates that a PCIe Local Error was received. The error code should be read from bit [24:20] of the Interrupt Decode 2 register.
[31]	0	RW1C	dma_int_rcvd	Indicates that a DMA interrupt was received. The user application should check for the 2nd level DMA registers. (Only applicable when DMA is enabled.) For XDMA: IRQ Block User Interrupt Request Register (0x2040). IRQ Block Engine Interrupt Request Register (0x2044).

Interrupt Mask Register (0xE14)

Register to control whether each individual interrupt source can cause the interrupt line[0] to be asserted. A one in any location allows the interrupt source to assert the interrupt line. This register initializes to all zeros. Therefore, by default no interrupt is generated for any event.

Table 266: **Interrupt Mask Register (0xE14)**

Bit	Default	Access Type	Field	Description
[0]	0	RW	link_down	Enables interrupts for Link Down events when bit is set.
[1]	0	RW	sw_ctrl_int	Enable interrupts for Software Interrupts (from Host and etc.) when bit is set.
[2]	0	RW	flr_is_hit	Enables interrupts for Slave Transactions hitting FLR events when bit is set.
[3]	0	RW	hot_reset	Enables interrupts for Hot Reset events when bit is set. (Only writable for EP configurations, otherwise = 0)
[17:4]	0	RO	reserved	Reserved

Table 266: **Interrupt Mask Register (0xE14)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[18]	0	RW	vdm_rcvd	Enables interrupts for VDM events when bit is set.
[19]	0	RW	pme_turn_off_rcvd	Enables interrupts for PME_Turn_Off events when bit is set. (Only writable for EP configurations, otherwise = 0)
[20]	0	RW	slv_ur	Enables the Slave Unsupported Request interrupt when bit is set.
[21]	0	RW	slv_tz_violation	Enables the Slave TrustZone Violation interrupt when bit is set.
[22]	0	RW	slv_cpl_timeout	Enables the Slave Completion Timeout interrupt when bit is set.
[23]	0	RW	slv_err_poison	Enables the Slave Error Poison interrupt when bit is set.
[24]	0	RW	slv_ca	Enables the Slave Completer Abort interrupt when bit is set.
[25]	0	RW	slv_illegal_burst	Enables the Slave Illegal Burst interrupt when bit is set.
[26]	0	RW	mst_decerr	Enables the Master DECERR interrupt when bit is set.
[27]	0	RW	mst_slverr	Enables the Master SLVERR interrupt when bit is set.
[28]	0	RW	slv_pcie_timeout	Enables the Slave PCIe Timeout interrupt when bit is set.
[29]	0	RW	ecc_parity_err	Enables the RAM ECC/Parity Error interrupt when bit is set.
[30]	0	RW	pcie_local_err	Enables the PCIe Local Error interrupt when bit is set.
[31]	0	RW	dma_int	Enables the DMA interrupt when bit is set.

Bus Location Register (0xE18)

Register to report the Bus, Device, and Function number, and the Port number for the PCIe port.

Table 267: **Bus Location Register (0xE18)**

Bit	Default	Access Type	Field	Description
[2:0]	0	RO	func_num	Function number of the port for PCIe. Hard-wired to 0.
[7:3]	0	RO	dev_num	Device number of port for PCIe. For Endpoint, this register is RO and is set by the Root Port.
[15:8]	0	RO	bus_num	Bus number of port for PCIe. For Endpoint, this register is RO and is set by the external Root Port.
[23:16]	0	RW	port_num	Sets the Port number field of the Link Capabilities register. EP: Always Read 0 and is not writeable. RP: Is writeable.
[31:24]	0	RO	reserved	Reserved

PHY Control and Status Register (0xE1C)

Register to provide the status of the current PHY state, as well as control of speed and rate switching for PCIe core.

Table 268: **PHY Control and Status Register (0xE1C)**

Bit	Default	Access Type	Field	Description
[0]	0	RO	link_is_gen2	Reports whether the current link rate is 5.0 GT/s.
[2:1]	0	RO	link_width	Reports the current link width. 00b = x1, 01b = x2, 10b = x4, 11b= x8.
[8:3]	0	RO	ltssm_state	Reports the current Link Training and Status State Machine (LTSSM) state. Encoding is specific to the underlying integrated block.
[10:9]	0	RO	reserved	Reserved
[11]	0	RO	link_up	Reports the current PHY Link-up state. 1b: Link up 0b: Link down Link up indicates the core has achieved link up status, meaning the LTSSM is in the L0 state and the core can send/receive data packets.
[12]	0	RO	link_is_gen3	Reports whether the current link rate is 8.0 GT/s.
[13]	0	RO	link_width_is_x16	Reports the current link width. 0b = See bit[2:1]. 1b = x16
[14]	0	RO	link_is_gen4	Reports whether the current link rate is 16.0 GT/s.
[31:15]	0	RO	reserved	Reserved

Interrupt Decode 2 Register (0xE38)

Register to determine what is causing the interrupt line[0] to be asserted and how to clear the interrupt. Writing a 1'b1 to any bit clears that bit.

Table 269: **Interrupt Decode 2 Register (0xE38)**

Bit	Default	Access Type	Field	Description
[0]	0	RW1C	slv_axis_par_err	Indicates a parity error was detected on the AXIST RC interface
[1]	0	RW1C	slv_r_ecc_err	Indicates an ECC uncorrectable error was detected by the Slave Read RAM
[2]	0	RW1C	slv_w_ecc_err	Indicates an ECC uncorrectable error was detected by the Slave Write RAM
[3]	0	RW1C	mst_axis_par_err	Indicates a parity error was detected on the AXIST CQ interface
[4]	0	RW1C	mst_r_ecc_err	Indicates an ECC uncorrectable error was detected by the Master Read RAM
[5]	0	RW1C	mst_w_ecc_err	Indicates an ECC uncorrectable error was detected by the Master Write RAM

Table 269: **Interrupt Decode 2 Register (0xE38)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[6]	0	RW1C	slv_r_ecc_cerr	Indicates an ECC correctable error was detected by the Slave Read RAM
[7]	0	RW1C	slv_w_ecc_cerr	Indicates an ECC correctable error was detected by the Slave Write RAM
[8]	0	RW1C	mst_r_ecc_cerr	Indicates an ECC correctable error was detected by the Master Read RAM
[9]	0	RW1C	mst_w_ecc_cerr	Indicates an ECC correctable error was detected by the Master Write RAM
[10]	0	RW1C	slv_lite_par_err	Indicates a parity error was detected on the Slave LITE Write interface
[19:11]	0	RO	reserved	Reserved

Table 269: **Interrupt Decode 2 Register (0xE38)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[24:20]	0	RW1C	pcie_local_err_code	Indicates PCIe Local Error was received. The first error code is held till clear. 00000b - Reserved 00001b - Physical Layer Error Detected 00010b - Link Replay Timeout 00011b - Link Replay Rollover 00100b - Link Bad TLP Received 00101b - Link Bad DLLP Received 00110b - Link Protocol Error 00111b - Replay Buffer RAM Correctable ECC Error 01000b - Replay Buffer RAM Uncorrectable ECC Error 01001b - Receive Posted Request RAM Correctable ECC Error 01010b - Receive Posted Request RAM Uncorrectable ECC Error 01011b - Receive Completion RAM Correctable ECC Error 01100b - Receive Completion RAM Uncorrectable ECC Error 01101b - Receive Posted Buffer Overflow Error 01110b - Receive Non Posted Buffer Overflow Error 01111b - Receive Completion Buffer Overflow Error 10000b - Flow Control Protocol Error 10001b - Transmit Parity Error Detected 10010b - Unexpected Completion Received 10011b - Completion Timeout Detected 10100b - AXI4ST RQ INTFC Packet Drop 10101b - AXI4ST CC INTFC Packet Drop 10110b - AXI4ST CQ Poisoned Drop 10111b - User Signaled Internal Correctable Error 11000b - User Signaled Internal Uncorrectable Error 11001b - TPH RAM Internal Correctable Error 11010b - TPH RAM Internal Uncorrectable Error 11011b - MSIX RAM Internal Correctable Error 11100b - MSIX RAM Internal Uncorrectable Error 11101b - DVSEC RAM Internal Correctable Error 11110b - DVSEC RAM Internal Uncorrectable Error 11111b - Reserved
[31:25]	0	RO	reserved	Reserved

Interrupt Mask 2 Register (0xE3C)

Register to control whether each individual interrupt source can cause the interrupt line [0] to be asserted. A one in any location allows the interrupt source to assert the interrupt line. This register initializes to all zeros. Therefore, by default no interrupt is generated for any event.

Table 270: **Interrupt Mask 2 Register (0xE3C)**

Bit	Default	Access Type	Field	Description
[0]	0	RW	slv_axis_par_err	Enables interrupts for AXIST RC parity error events when bit is set.
[1]	0	RW	slv_r_ecc_err	Enables interrupts for Slave Read RAM ECC uncorrectable error events when bit is set.
[2]	0	RW	slv_w_ecc_err	Enables interrupts for Slave Write RAM ECC uncorrectable error events when bit is set.
[3]	0	RW	mst_axis_par_err	Enables interrupts for AXIST CQ parity error events when bit is set.
[4]	0	RW	mst_r_ecc_err	Enables interrupts for Master Read RAM ECC uncorrectable error events when bit is set.
[5]	0	RW	mst_w_ecc_err	Enables interrupts for Master Write RAM ECC uncorrectable error events when bit is set.
[6]	0	RW	slv_r_ecc_cerr	Enables interrupts for Slave Read RAM ECC correctable error events when bit is set.
[7]	0	RW	slv_w_ecc_cerr	Enables interrupts for Slave Write RAM ECC correctable error events when bit is set.
[8]	0	RW	mst_r_ecc_cerr	Enables interrupts for Master Read RAM ECC uncorrectable error events when bit is set.
[9]	0	RW	mst_w_ecc_cerr	Enables interrupts for Master Write RAM ECC uncorrectable error events when bit is set.
[10]	0	RW	slv_lite_par_err	Enables interrupts for Slave LITE Write parity error events when bit is set.
[19:11]	0	RO	reserved	Reserved
[31:20]	0	RO	reserved	Reserved

Configuration Control Register (0xE40)

Register to allow the user application to indicate if a correctable or uncorrectable error has occurred and report it in the respective AER Error Status Register.

Table 271: **Configuration Control Register (0xE40)**

Bit	Default	Access Type	Field	Description
[0]	0	RW	uc_err	Uncorrectable Error Detected. The user application writes a 1 to this bit to indicate an Uncorrectable error was detected within the user logic that needs to be reported as an internal error through the PCI Express Advanced Error Reporting mechanism. In response, the core sets the Uncorrected Internal Error Status bit in the AER Uncorrectable Error Status Register of all enabled functions, and also sends an error message if enabled to do so. This error is not considered function-specific. This bit only asserts for 1 clock cycle and automatically resets to 0 in the next clock cycle.

Table 271: **Configuration Control Register (0xE40)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[1]	0	RW	c_err	Correctable Error Detected. The user application writes a 1 to this bit to indicate a Correctable error was detected within the user logic that needs to be reported as an internal error through the PCI Express Advanced Error Reporting mechanism. In response, the core sets the Corrected Internal Error Status bit in the AER Correctable Error Status Register of all enabled functions, and also sends an error message if enabled to do so. This error is not considered function-specific. This bit only asserts for 1 clock cycle and automatically resets to 0 in the next clock cycle.
[31:2]	0	RO	reserved	Reserved

Slave Error AID Register (0xE44)

Register to determine which ID was violated the Slave checkers. Only the first violated ID was logged after clearing the corresponding interrupt bit in the Interrupt Decode register.

Table 272: **Slave Error AID Register (0xE44)**

Bit	Default	Access Type	Field	Description
[7:0]	0	RO	tz_violation_aid	Reports the first ID of Slave TrustZone Violoation after clearing the interrupt bit of TrustZone Violation.
[15:8]	0	RO	tz_violation_func	Reports the first Function Number (SMID) of Slave TrustZone Violoation after clearing the interrupt bit of TrustZone Violation.
[23:16]	0	RO	cpl_err_aid	Reports the first ID of Slave Completion error after clearing any interrupt bit of Slave UR, Slave CA, or Slave Error Poison.
[31:24]	0	RO	cpl_err_func	Reports the first Function Number (SMID) of Slave Completion error after clearing any interrupt bit of Slave UR, Slave CA, or Slave Error Poison.

User IRQ Request Register (0xE58)

Register to allow the user application to access usr_irq_req interface.

Table 273: **User IRQ Request Register (0xE58)**

Bit	Default	Access Type	Field	Description
[15:0]	0	RW	usr_irq_req_set	Sets usr_irq_req[n] by writing 1 to bit[n]. Read returns the current value of usr_irq_req. See DMA IRQ Block for the definition of usr_irq_req. Bit[n] is automatically cleared when usr_irq_ack[n] is received for MSI or MSI-X.

Table 273: **User IRQ Request Register (0xE58)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[19:16]	0	wo	usr_irq_req_clr	Clears usr_irq_req[n] by writing 1 to bit[n] for INTx. See DMA IRQ Block for the definition of usr_irq_req.
[31:20]	0	RO	reserved	Reserved

User IRQ Acknowledge Register (0xE5C)

Register to allow the user application to access usr_irq_ack interface.

Table 274: **User IRQ Acknowledge Register (0xE5C)**

Bit	Default	Access Type	Field	Description
[15:0]	0	RO	usr_irq_ack	Indicates the value of usr_irq_ack. See DMA IRQ Block for the definition of usr_irq_ack. Bit[n] is automatically cleared when usr_irq_req[i] is set for INTx/MSI/MSI-X or clear when usr_irq_req[i] is cleared for INTx.
[31:16]	0	RO	usr_irq_fail	Indicates the value of usr_irq_fail. See DMA IRQ Block for the definition of usr_irq_fail. Bit[n] is only valid when usr_irq_ack[n] is set.

PCIe TX Message Control Register (0xE60)

Register to generate PCIe TX messages and send to the remote component. Use this register with TX_MSG_HDR_L, TX_MSG_HDR_H, and TX_MSG_DFIFO to synthesize the message.

Table 275: **PCIe TX Message Control Register (0xE60)**

Bit	Default	Access Type	Field	Description
[0]	0	RW	msg_execute	Writes 1 to send the PCIe TX message defined by TX_MSG_CSR, TX_MSG_HDR_L, TX_MSG_HDR_H, and TX_MSG_DFIFO which should be programmed ahead. Read returns the sending status of the message: 0b: Delivered to PCIe. Checks msg_fail for the completion status. 1b: In progress
[3:1]	0	RW	msg_routing	Programs Message Rounting field of the PCIe TX message. 000b: Routed to Root Complex 010b: Routed by ID 011b: Broadcast from Root Complex 100b: Local - Terminate at Receiver 101b: Gathered and routed to Root Complex Others: Reserved See PCIe spec for valid settings for each message.

Table 275: **PCIe TX Message Control Register (0xE60)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[7:4]	0	RW	msg_data_ptr_sel	Overwrites TX_MSG_DFIFO pointers. This is for debug purposes and should be set to 0 under normal conditions.
[15:8]	0	RW	msg_code	Programs Message Code field of the PCIe TX message. 0001_0100b: PM_Active_State_Nak 0001_1000b: PM_PME 0001_1001b: PME_Turn_Off 0001_1011b: PME_TO_Ack 0111_1110b: Vendor_Defined Typo0 0111_1111b: Vendor_Defined Typo1 Others: Reserved
[20:16]	0	RW	msg_data_length	Programs Dword Length of the PCIe TX message. 0: No payload (Msg) 1: 1 Dword (MsgD) 16: 16 Dwords (MsgD) Others: Reserved See the *PCI-SIG Specifications* (http://www.pcisig.com/specifications) for valid settings for each message. Vendor_Defined messages can support up to 16 Dwords (64 Bytes).
[22:21]	0	RO	reserved	Reserved
[23]	0	RW1C	msg_fail	Indicates the completion status of the message. Valid when the message is delivered. Writing a 1 clears this bit. Writing a 1 to msg_execute also clears this bit. 0b: Completed 1b: Failed
[31:24]	0	RW	msg_function	Programs Requester Function Number field of the PCIe TX message.

PCIe TX Message Header L Register (0xE64)

Register to program header byte 8-11 of PCIe TX messages.

Table 276: **PCIe TX Message Header L Register (0xE64)**

Bit	Default	Access Type	Field	Description
[7:0]	0	RW	msg_tlp_hdr8	Programs Message Header Byte 8 field of the PCIe TX message. See PCIe spec for valid settings for each message.
[15:8]	0	RW	msg_tlp_hdr9	Programs Message Header Byte 9 field of the PCIe TX message. See PCIe spec for valid settings for each message.
[23:16]	0	RW	msg_tlp_hdr10	Programs Message Header Byte 10 field of the PCIe TX message. See PCIe spec for valid settings for each message.

Table 276: **PCIe TX Message Header L Register (0xE64)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[31:24]	0	RW	msg_tlp_hdr11	Programs Message Header Byte 11 field of the PCIe TX message. See PCIe spec for valid settings for each message.

PCIe TX Message Header H Register (0xE68)

Register to program header byte 12-15 of PCIe TX messages.

Table 277: **PCIe TX Message Header H Register (0xE68)**

Bit	Default	Access Type	Field	Description
[7:0]	0	RW	msg_tlp_hdr12	Programs Message Header Byte 12 field of the PCIe TX message. See PCIe spec for valid settings for each message.
[15:8]	0	RW	msg_tlp_hdr13	Programs Message Header Byte 13 field of the PCIe TX message. See PCIe spec for valid settings for each message.
[23:16]	0	RW	msg_tlp_hdr14	Programs Message Header Byte 14 field of the PCIe TX message. See PCIe spec for valid settings for each message.
[31:24]	0	RW	msg_tlp_hdr15	Programs Message Header Byte 15 field of the PCIe TX message. See PCIe spec for valid settings for each message.

PCIe TX Message Data FIFO Register (0xE6C)

Register to program payload to be sent with the PCIe TX Message (MsgD).

Table 278: **PCIe TX Message Data FIFO Register (0xE6C)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW	msg_tlp_data	Writes PCIe TX message payload one by one from the 1st Dword to the message length. Each write increases the write pointer by 1 upto 15. The write pointer returns to 0 upon reset or commitment of the previous TX message, For debug purpose, the write pointer and the read pointer can be overwritten by programming msg_data_ptr_sel, Write programs the value of the selected Dword (0-15). Read returns the value from the selected Dword (0-15).

PCIe RX Message Control and Status Register (0xE70)

Register to provide access to the PCIe RX Message specific status and control.

Table 279: **PCIe RX Message Control and Status Register (0xE70)**

Bit	Default	Access Type	Field	Description
[0]	0	RO	mfifo_not_empty	Indicates that the Message FIFO has VDM messages to read.
[1]	0	RW1C	mfifo_overflow	Indicates that the Message FIFO overflowed and a VDM message was dropped. Writing a 1 clears the overflow status.
[3:2]	0	RO	reserved	Reserved
[7:4]	0	RO	msg_count	Indicates the count of VDM messages stored in the Message FIFO. The user application can know how many message to read.
[12:8]	0	RO	mfifo_read_ptr	Indicates the current read pointer of RX_MFIFO READ. This is for debug purposes.
[15:13]	0	RO	reserved	Reserved
[31:16]	0	RO	overflow_rid	Indicates the Requester ID of the 1st dropped VDM message after reset or clearing of mfifo_overflow.

PCIe RX Message FIFO Register (0xE74)

Reads from this location return a VDM message. Reads are non-destructive. Removing the message from the FIFO requires a write. The write value is ignored.

Table 280: **PCIe RX Message FIFO Register (0xE74)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW1C	msg_tlp_dw	Indicates a Dword from VDM messages sequentially from Header Dwords to Payload Dwords. After each read, the user application should write to this register to remove a Dword. The fields for 1st Header Dword: [31:16] Requester ID [15:8] Message Code [7:5] Message Routing [4:0] Payload DW Length. 0 means no payload and the user application has to read two more Dwords to get the remaining Header. The fields for 2nd Header Dword: [31:0] Header Byte 11 - 8 The fields for 3rd Header Dword: [31:0] Header Byte 15 - 12 The fields for Payload Dword: [31:0] Payload Byte (N+3) - N

Master Pending Counter Register (0xE78)

Register to provide the counts of pending requests on the Bridge Master port for debug and performance monitor.

Table 281: **Master Pending Counter Register (0xE78)**

Bit	Default	Access Type	Field	Description
[7:0]	0	RO	wr_pend_cnt	Pending Write Count on the Bridge Master port.
[15:8]	0	RO	rd_pend_cnt	Pending Read Count on the Bridge Master port.
[31:16]	0	RO	reserved	Reserved

PCIe TX MSI / MSI-X Control and Status Register (0xE7C)

Register to generate PCIe TX MSI (not valid for SR-IOV) or MSI-X interrupts.

Table 282: **PCIe TX MSI / MSI-X Control and Status Register (0xE7C)**

Bit	Default	Access Type	Field	Description
[4:0]	0	RW	int_vector	Vector Number of MSI or MSI-X. This field should be programmed along with setting int_set.
[7:5]	0	RO	reserved	Reserved
[15:8]	0	RW	int_function	Function Number of MSI or MSI-X. This field should be programmed along with setting int_set. For MSI, only Physical Functions are valid.
[16]	0	RW	int_set	Writes 1 to send the PCIe TX MSI / MSI-X interrupts. Read returns the sending status of the interrupt: 0b: Delivered to PCIe. Checks int_fail for the completion status. 1b: In progress
[17]	0	RW	int_is_msix	MSI-X or MSI. This field should be programmed along with setting int_set. 0b: MSI 1b: MSI-X
[19:18]	0	RO	reserved	Reserved
[20]	0	RW1C	int_fail	Indicates the completion status of the interrupt. Valid when the interrupt is delivered. Writing a 1 clears this bit. Writing a 1 to int_set also clears this bit. 0b: Completed 1b: Failed
[31:21]	0	RO	reserved	Reserved

VSEC Capability 2 Register (0xED8)

Register to allow the memory space for the core to appear as though it is a part of the underlying integrated block PCIe configuration space. The VSEC is inserted immediately following the last enhanced capability structure.

Table 283: **VSEC Capability 2 Register (0xED8)**

Bit	Default	Access Type	Field	Description
[15:0]	0xB	RO	cap_id	Indicates the PCIe defined ID identifying this Enhanced Capability as a Vendor-Specific capability.
[19:16]	0x1	RO	cap_ver	Indicates the version of this capability structure.
[31:20]	0	RO	nxt_cap_offset	Indicates the offset of the next capability.

VSEC Header 2 Register (0xEDC)

Register to provide a unique (within a given vendor) identifier for the layout and contents of the VSEC structure, as well as its revision and length. VSEC Header 2 Register is part of the AXI Bridge that contains AXI Base Address Translation Configuration Registers which start immediately after VSEC Header 2 Register (Offset 0xEE0).

Table 284: **VSEC Header 2 Register (0xEDC)**

Bit	Default	Access Type	Field	Description
[15:0]	0x2	RO	vsec_id	Indicates the ID value uniquely identifying the nature and format of this VSEC structure.
[19:16]	0	RO	vsec_rev	Indicates the version of this capability structure.
[31:20]	0x38	RO	vsec_length	Indicates the length of the entire VSEC Capability structure, in bytes, including the VSEC Capability register.

AXI Base Address Translation Configuration Registers (Offset - 0xEE0 - 0xF0C)

The AXI Base address translation configuration registers and their offsets a basedre shown in the first table below and the register bit are described below. This set of registers can be used in two configurations based on the address width of PCIe BARs. When the PCIe BAR is set to 32-bit address space, then the translation vector should be placed into the AXIBAR2PCIEBAR_nL register where n is the PCIe BAR number. When the BAR is set to a 64-bit address space, then the most significant 32 bits are written into the AXIBAR2PCIEBAR_nU and the least significant 32 bits are written into AXIBAR2PCIEBAR_nL. Care should be taken so that invalid values are not written to the address translation registers.

Table 285: **AXI Basr Address Translation Configuration Registers (Offset 0xEE0 - 0xF0C)**

Offset	Bits	Register Mnemonic
0xEE0	[31:0]	AXIBAR2PCIEBAR_0U
0xEE4	[31:0]	AXIBAR2PCIEBAR_0L
0xEE8	[31:0]	AXIBAR2PCIEBAR_1U
0xEEC	[31:0]	AXIBAR2PCIEBAR_1L

Table 285: **AXI Basr Address Translation Configuration Registers (Offset 0xEE0 - 0xF0C)** *(cont'd)*

Offset	Bits	Register Mnemonic
0xEF0	[31:0]	AXIBAR2PCIEBAR_2U
0xEF4	[31:0]	AXIBAR2PCIEBAR_2L
0xEF8	[31:0]	AXIBAR2PCIEBAR_3U
0xEFC	[31:0]	AXIBAR2PCIEBAR_3L
0xF00	[31:0]	AXIBAR2PCIEBAR_4U
0xF04	[31:0]	AXIBAR2PCIEBAR_4L
0xF08	[31:0]	AXIBAR2PCIEBAR_5U
0xF0C	[31:0]	AXIBAR2PCIEBAR_5L

Chapter 4

Designing with the Subsystem

General Design Guidelines

Use the Example Design

Each instance of the QDMA Subsystem for PCIe created by the Vivado® design tool is delivered with an example design that can be implemented in a device and then simulated. This design can be used as a starting point for your own design or can be used to sanity-check your application in the event of difficulty. See the Example Design content for information about using and customizing the example designs for the subsystem.

Registering Signals

To simplify timing and increase system performance in a programmable device design, keep all inputs and outputs registered between the user application and the subsystem. This means that all inputs and outputs from the user application should come from, or connect to, a flip-flop. While registering signals might not be possible for all paths, it simplifies timing analysis and makes it easier for the Xilinx® tools to place and route the design.

Recognize Timing Critical Signals

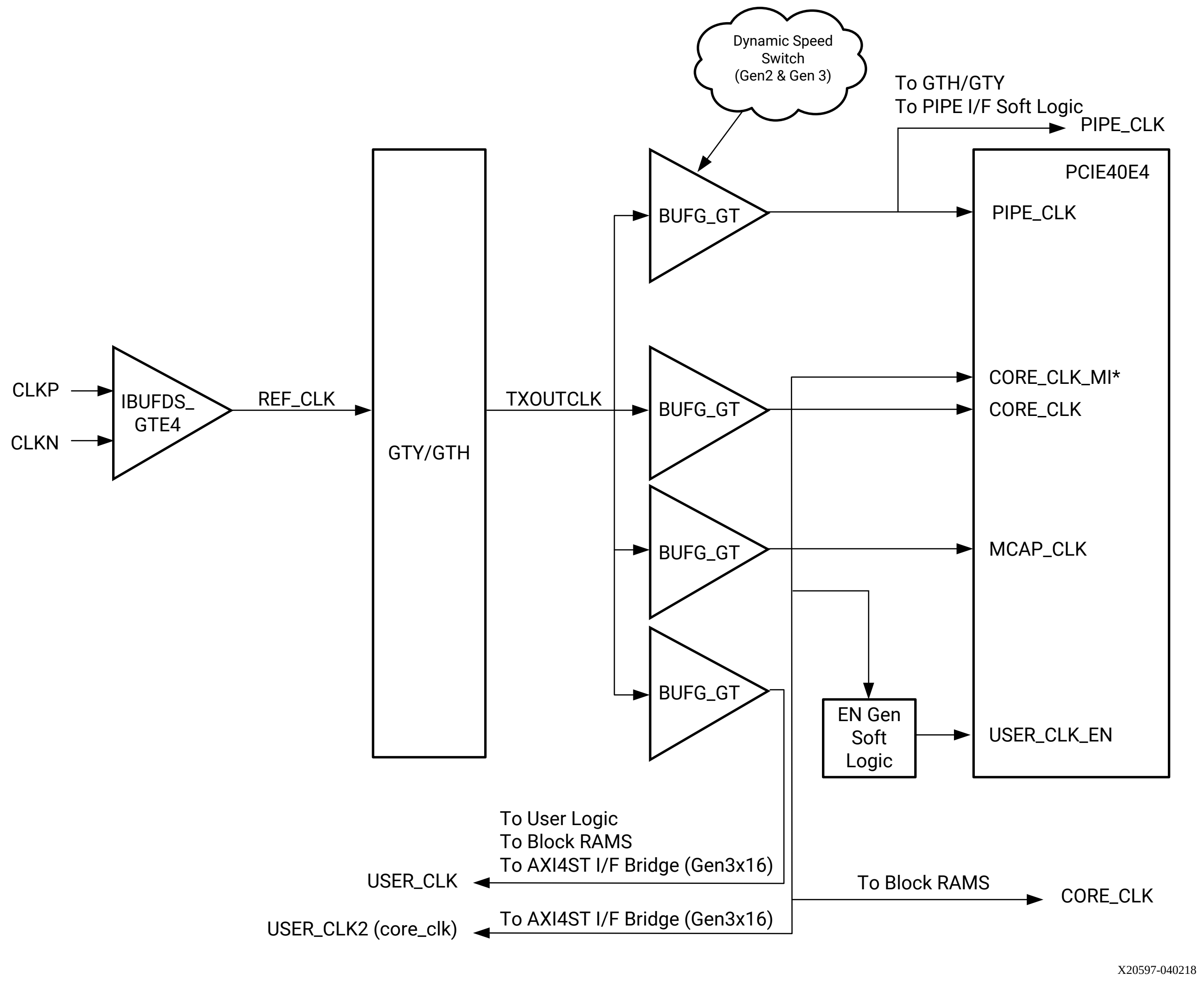
The constraints provided with the example design identify the critical signals and timing constraints that should be applied.

Make Only Allowed Modifications

You should not modify the subsystem. Any modifications can have adverse effects on system timing and protocol compliance. Supported user configurations of the subsystem can only be made by selecting the options in the customization IP dialog box when the subsystem is generated.

Clocking

Figure 23: **Clocking**



PCIe clocks (`pipe_clk`, `core_clk`, `user_clk`, and `mcap_clk`) are all driven by `bufg_gt` sourced from `txoutclk` pin. These clocks are derived clock from `gtrefclk0` through a CPLL. In an application where QPLL is used, QPLL is only provided to the GT PCS/ PMA block while `txoutclk` continues to be derived from a CPLL. All user interface signals of the IP are timed with respect to the same clock (`user_clk`) which can have a frequency of 62.5, 125, or 250 MHz depending on the link speed and width configured. The QDMA Subsystem for PCIe and the user logic primarily work on `user_clk`.

Chapter 5

Design Flow Steps

This section describes customizing and generating the subsystem, constraining the subsystem, and the simulation, synthesis, and implementation steps that are specific to this IP subsystem. More detailed information about the standard Vivado® design flows and the IP integrator can be found in the following Vivado Design Suite user guides:

- *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
- *Vivado Design Suite User Guide: Designing with IP* (UG896)
- *Vivado Design Suite User Guide: Getting Started* (UG910)
- *Vivado Design Suite User Guide: Logic Simulation* (UG900)

Customizing and Generating the Subsystem

This section includes information about using Xilinx® tools to customize and generate the subsystem in the Vivado® Design Suite.

If you are customizing and generating the subsystem in the Vivado IP integrator, see the *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994) for detailed information. IP integrator might auto-compute certain configuration values when validating or generating the design. To check whether the values do change, see the description of the parameter in this chapter. To view the parameter value, run the `validate_bd_design` command in the Tcl console.

You can customize the IP for use in your design by specifying values for the various parameters associated with the IP subsystem using the following steps:

1. Select the IP from the IP catalog.
2. Double-click the selected IP or select the Customize IP command from the toolbar or right-click menu.

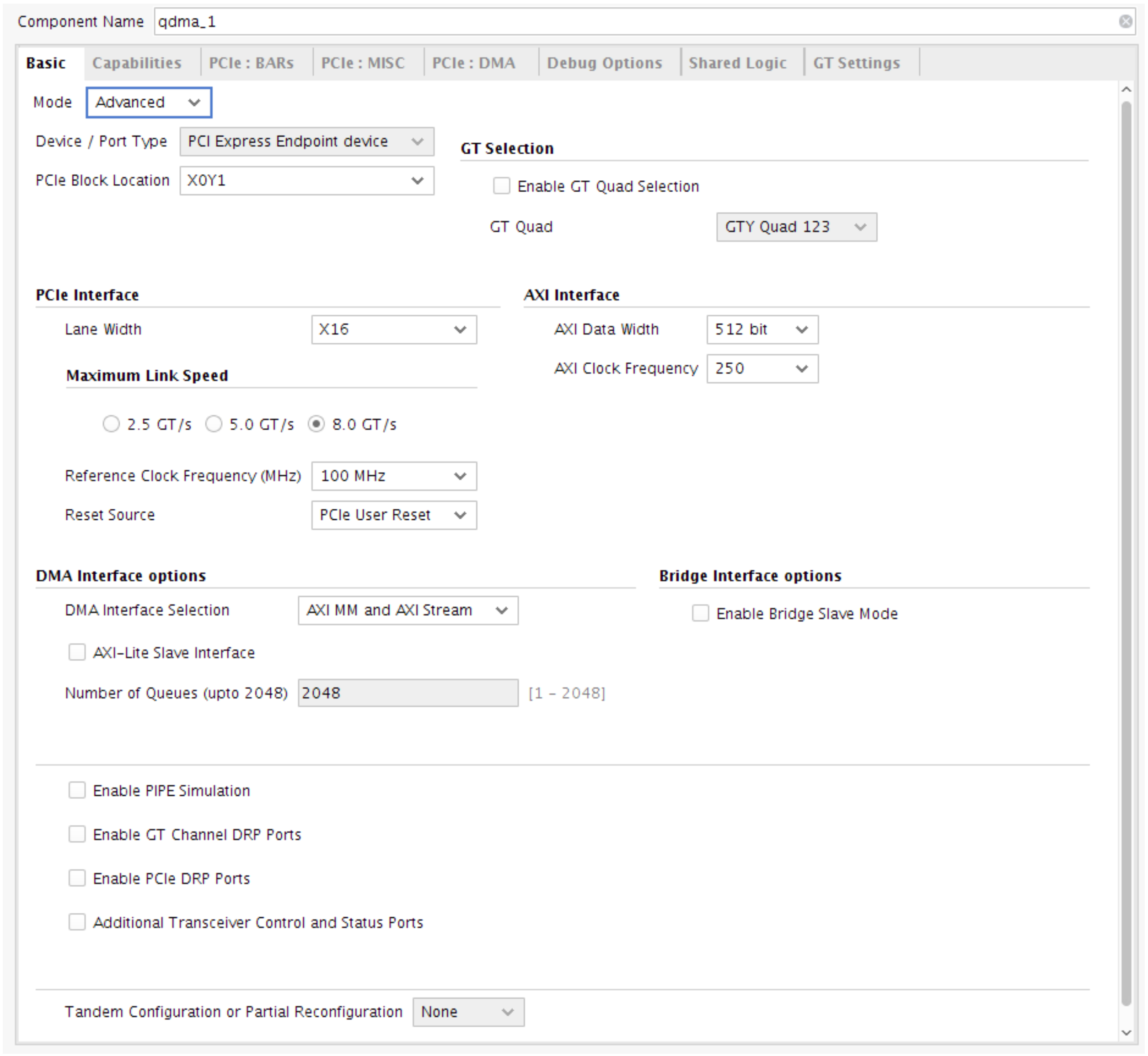
For details, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) and the *Vivado Design Suite User Guide: Getting Started* (UG910).

Figures in this chapter are illustrations of the Vivado IDE. The layout depicted here might vary from the current version.

Basic Tab

The Basic Tab is shown in the following figure.

Figure 24: **Basic Tab**



Component Name: qdma_1

Mode: Advanced

Device / Port Type: PCI Express Endpoint device

PCIe Block Location: X0Y1

GT Selection

☐ Enable GT Quad Selection

GT Quad: GTY Quad 123

PCIe Interface

Lane Width: X16

Maximum Link Speed

2.5 GT/s 5.0 GT/s [✓ 8.0 GT/s]

Reference Clock Frequency (MHz): 100 MHz

Reset Source: PCIe User Reset

AXI Interface

AXI Data Width: 512 bit

AXI Clock Frequency: 250

DMA Interface options

DMA Interface Selection: AXI MM and AXI Stream

☐ AXI-Lite Slave Interface

Number of Queues (upto 2048): 2048 [1 - 2048]

Bridge Interface options

☐ Enable Bridge Slave Mode

☐ Enable PIPE Simulation

☐ Enable GT Channel DRP Ports

☐ Enable PCIe DRP Ports

☐ Additional Transceiver Control and Status Ports

Tandem Configuration or Partial Reconfiguration: None

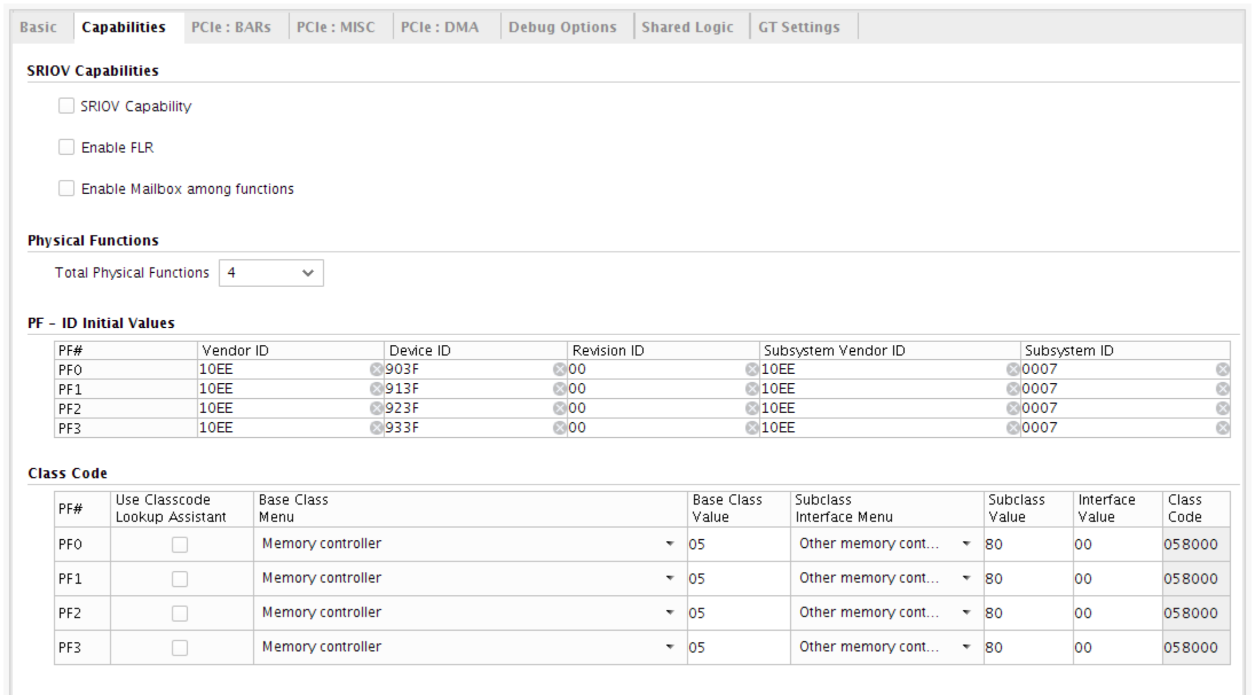
- **Mode:** Allows you to select the Basic or Advanced mode of the configuration of core.
- **Device /Port Type:** Only PCI Express® Endpoint device mode is supported.
- **GT Selection/Enable GT Quad Selection:** Select the Quad in which lane 0 is located.

- **PCIe Block Location:** Selects from the available integrated blocks to enable generation of location-specific constraint files and pinouts. This selection is used in the default example design scripts. This option is not available if a Xilinx Development Board is selected.
- **Lane Width:** The core requires the selection of the initial lane width. The *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213) define the available widths and associated generated core. Wider lane width cores can train down to smaller lane widths if attached to a smaller lane-width device. Options are 4, 8, or 16 lanes.
- **Maximum Link Speed:** The core allows you to select the Maximum Link Speed supported by the device. The*UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213) define the lane widths and link speeds supported by the device. Higher link speed cores are capable of training to a lower link speed if connected to a lower link speed capable device. The default option is Gen3.
- **Reference Clock Frequency:** The default is 100 MHz.
- **Reset Source:** You can choose one of:
  - **PCIe User Reset:** The user reset comes from PCIe core after the link is established. When the PCIe link goes down, the user reset is asserted and the core goes to reset mode. And when the link comes back up, the user reset is deasserted.
  - **Phy Ready:** When selected, the core is not affected by PCIe link status.
- **AXI Data Width:** Select 128, 256 bit, or 512 bit (only for UltraScale+). The core allows you to select the Interface Width, as defined in the *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213). The default interface width set in the Customize IP dialog box is the lowest possible interface width.
- **AXI Clock Frequency:** 250 MHz depending on the lane width/speed.
- **DMA Interface Option:** You can select one of three options:
  - AXI4 Memory Mapped and AXI4-Stream.
  - AXI4 Memory Mapped only.
  - AXI4-Stream only.
- **AXI Lite Slave Interface:** Select to enable the AXI4-Lite slave interface.
- **Enable Bridge Slave Mode:** Select to enable the AXI-MM Slave interface.
- **Enable PIPE Simulation:** Enable pipe simulation for faster simulation. This is used only for simulation.
- **Enable GT DRP Ports:** Enable GT-specific DRP ports.
- **Enable PCIe DRP Ports:** Enable PCIe-specific DRP ports.
- **Additional Transceiver Control and Status Ports:** Select to enable any additional ports.
- **Tandem Configuration or Partial Reconfiguration:** Select the Tandem Configuration or Partial Reconfiguration feature, if applicable to your design.

Capabilities Tab

The Capabilities Tab is shown in the following figure.

Figure 25: **Capabilities Tab**



SRIOV Capabilities

- [ ] SRIOV Capability
- [ ] Enable FLR
- [ ] Enable Mailbox among functions

Physical Functions

Total Physical Functions: 4

PF - ID Initial Values

PF#	Vendor ID	Device ID	Revision ID	Subsystem Vendor ID	Subsystem ID
PF0	10EE	903F	00	10EE	0007
PF1	10EE	913F	00	10EE	0007
PF2	10EE	923F	00	10EE	0007
PF3	10EE	933F	00	10EE	0007

Class Code

PF#	Use Classcode Lookup Assistant	Base Class Menu	Base Class Value	Subclass Interface Menu	Subclass Value	Interface Value	Class Code
PF0	☐	Memory controller	05	Other memory cont...	80	00	058000
PF1	☐	Memory controller	05	Other memory cont...	80	00	058000
PF2	☐	Memory controller	05	Other memory cont...	80	00	058000
PF3	☐	Memory controller	05	Other memory cont...	80	00	058000

- **SRIOV Capability:** Enables Single Root Port I/O Virtualization (SR-IOV) capabilities. The integrated block implements extended SR-IOV PCIe. When this is enabled, SR-IOV is implemented on all selected physical functions. When SR-IOV capabilities are enabled only MSI-X interrupt is supported.
- **Enable Mailbox among functions:** This is a Mailbox system to communicate between different functions. When **SR-IOV Capability** (above) is enabled, this option is enabled by default.
- **Enable FLR:** Enables Functionl level Reset port. When SR-IOV capability (above) is enabled, this option is enabled by default.
- **Total Physical Functions:** A maximum of four Physical Functions can be enabled.
- **PF - ID Initial Values:**
- **Vendor ID:** Identifies the manufacturer of the device or application. Valid identifiers are assigned by the PCI Special Interest Group to guarantee that each identifier is unique. The default value, `10EEh`, is the Vendor ID for Xilinx. Enter a vendor identification number here. `FFFFh` is reserved.
- **Device ID:** A unique identifier for the application; the default value, which depends on the configuration selected, is 70h. This field can be any value; change this value for the application.

The Device ID parameter is evaluated based on:

- The device family. 9 for UltraScale+™, 8 for UltraScale™, and 7 for 7 series devices.
- EP or RP mode
- Link width
- Link speed

If any of the above values are changed, the Device ID value will be re-evaluated, replacing the previous set value.

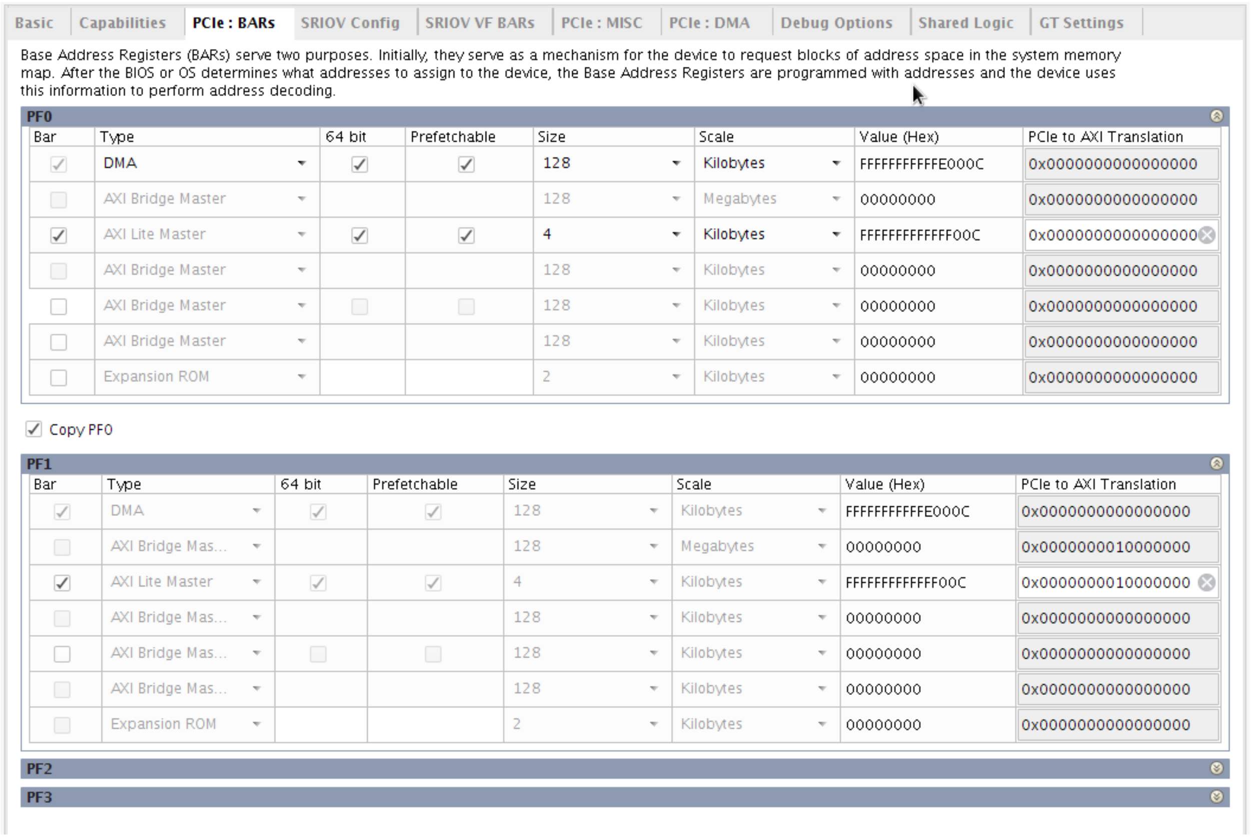
RECOMMENDED: *It is always recommended that the link width, speed and Device Port type be changed first and then the Device ID value. Make sure the Device ID value is set correctly before generating the IP.*

- **Revision ID:** Indicates the revision of the device or application; an extension of the Device ID. The default value is `00h`; enter values appropriate for the application.
- **Subsystem Vendor ID:** Further qualifies the manufacturer of the device or application. Enter a Subsystem Vendor ID here; the default value is 10EEh. Typically, this value is the same as Vendor ID. Setting the value to 0000h can cause compliance testing issues.
- **Subsystem ID:** Further qualifies the manufacturer of the device or application. This value is typically the same as the Device ID; the default value depends on the lane width and link speed selected. Setting the value to 0000h can cause compliance testing issues.
- **Class Code:** The Class Code identifies the general function of a device.
- **Use Classcode Lookup Assistant:** If selected, the Class Code Look-up Assistant provides the Base Class, Sub-Class and Interface values for a selected general function of a device. This Look-up Assistant tool only displays the three values for a selected function. You must enter the values in **Class Code** for these values to be translated into device settings..
- **Base Class:** Broadly identifies the type of function performed by the device..
- **Subclass:** More specifically identifies the device function..
- **Interface:** Defines a specific register-level programming interface, if any, allowing device-independent software to interface with the device.

PCIe BARs Tab

The PCIe BARs Tab is shown in the following figure.

Figure 26: **PCIe BARs Tab**



Basic | Capabilities | PCIe : BARs | SRIOV Config | SRIOV VF BARs | PCIe : MISC | PCIe : DMA | Debug Options | Shared Logic | GT Settings

Base Address Registers (BARs) serve two purposes. Initially, they serve as a mechanism for the device to request blocks of address space in the system memory map. After the BIOS or OS determines what addresses to assign to the device, the Base Address Registers are programmed with addresses and the device uses this information to perform address decoding.

PF0

Bar	Type	64 bit	Prefetchable	Size	Scale	Value (Hex)	PCIe to AXI Translation
✓	DMA	✓	✓	128	Kilobytes	FFFFFFFFFFFE000C	0x0000000000000000
☐	AXI Bridge Master			128	Megabytes	00000000	0x0000000000000000
✓	AXI Lite Master	✓	✓	4	Kilobytes	FFFFFFFFFFFFF00C	0x0000000000000000
☐	AXI Bridge Master			128	Kilobytes	00000000	0x0000000000000000
☐	AXI Bridge Master	☐	☐	128	Kilobytes	00000000	0x0000000000000000
☐	AXI Bridge Master			128	Kilobytes	00000000	0x0000000000000000
☐	Expansion ROM			2	Kilobytes	00000000	0x0000000000000000

✓ Copy PF0

PF1

Bar	Type	64 bit	Prefetchable	Size	Scale	Value (Hex)	PCIe to AXI Translation
✓	DMA	✓	✓	128	Kilobytes	FFFFFFFFFFFE000C	0x0000000000000000
☐	AXI Bridge Mas...			128	Megabytes	00000000	0x0000000010000000
✓	AXI Lite Master	✓	✓	4	Kilobytes	FFFFFFFFFFFFF00C	0x0000000010000000
☐	AXI Bridge Mas...			128	Kilobytes	00000000	0x0000000000000000
☐	AXI Bridge Mas...	☐	☐	128	Kilobytes	00000000	0x0000000000000000
☐	AXI Bridge Mas...			128	Kilobytes	00000000	0x0000000000000000
☐	Expansion ROM			2	Kilobytes	00000000	0x0000000000000000

PF2

PF3

- **Base Address Register Overview:** In Endpoint configuration, the core supports up to six 32-bit BARs or three 64-bit BARs, and the Expansion read-only memory (ROM) BAR. In Root Port configuration, the core supports up to two 32-bit BARs or one 64-bit BAR, and the Expansion ROM BAR. BARs can be one of two sizes:
    - **32-bit BARs:** The address space can be as small as 128 bytes or as large as 2 gigabytes. Used for DMA, AXI Lite Master or AXI Bridge Master.
    - **64-bit BARs:** The address space can be as small as 128 bytes or as large as 8 Exabytes. Used for DMA, AXI Lite Master or AXI Bridge Master.

All BAR register share these options:

- **BAR:** Click the checkbox to enable the BAR. Deselect the checkbox to disable the BAR.
- **Type:** Select from **DMA** (fixed BAR0), **AXI Lite Master** (fixed to BAR1, if enabled), or **AXI Bridge Master** (fixed to BAR2, if enabled). All other BARs, you can select between AXI List Master and AXI Bridge Master. Expansion ROM can be enabled by selecting BAR6

  For 64-bit BAR (default section), **DMA** is BAR0 (fixed), **AXI Lite Master** is fixed to BAR2 (if enabled), and **AXI Bridge Master** is fixed to BAR4 (if enabled). Expansion ROM can be enabled by selection BAR6.

- **DMA:** DMA is fixed in BAR0 space and for all PFs. You can select **DMA Mailbox Management** rather than DMA; however, DMA Mailbox Management does not allow you to perform any DMA operations. After selecting the DMA Mailbox Management option, the host has access to the extended Mailbox space. For details about this space, see the QDMA_PF_MAILBOX (0x2400) register space.
- **AXI Lite Master:** Use this option to select or deselect the AXI Lite Master interface BAR space. The Size, scale and address translation are configurable.
- **AXI Bridge Master:** Use this option to select or deselect the AXI Bridge Master interface BAR space. The Size, scale and address translation are configurable.
  - **Size:** The available Size range depends on the 32-bit or 64-bit bar selected.
  - **Value:** The value assigned to the BAR based on the current selections.
- **Expansion ROM:**
- **Disabling Unused Resources:** For best results, disable unused base address registers to conserve system resources. A base address register is disabled by deselecting unused BARs in the Customize IP dialog box.

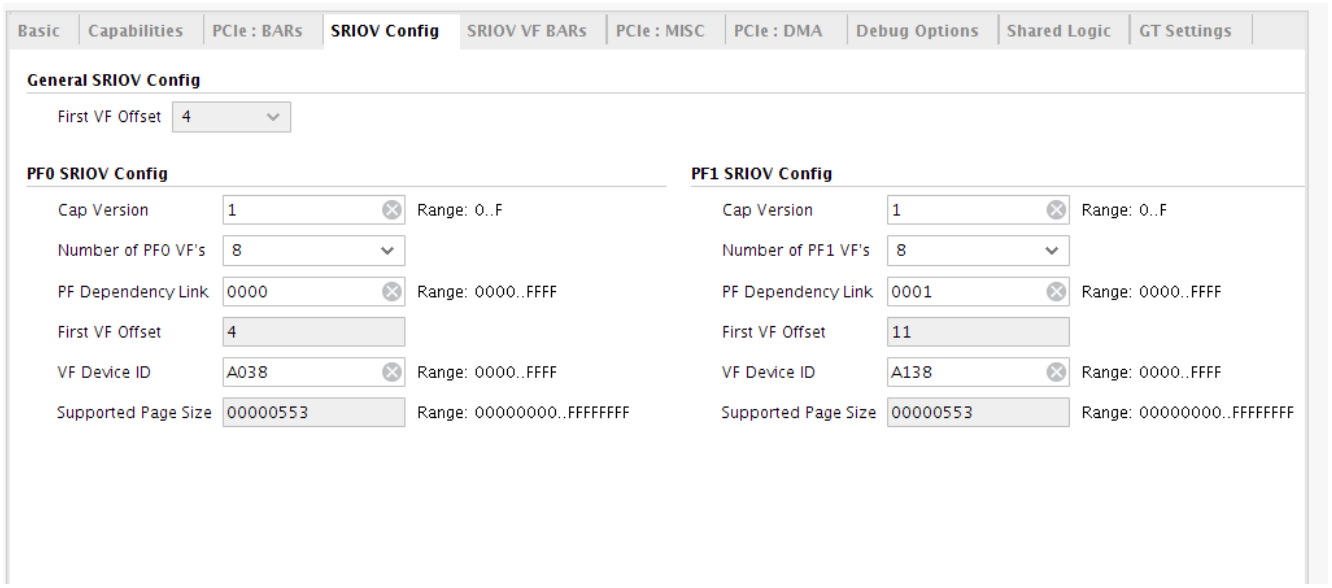
SRIOV Config Tab

The SRIOV Config tab allows you to specify the SR-IOV capability for a physical function (PF). The information is used to construct the SR-IOV capability structure. Virtual functions do not exist on power-on. It is the function of the system software to discover and enable VFs based on system capability. The VF support is discovered by scanning the SR-IOV capability structure for each PF.

***Note*:** When **SRIOV Capability** is selected in Capabilities Tab, the SRIOV Config tab appears.

The SRIOV Config Tab is shown in the following figure.

Figure 27: **SRIOV Config Tab**



General SRIOV Config

First VF Offset: 4

PF0 SRIOV Config

Cap Version	1	Range: 0..F
Number of PF0 VF's	8	
PF Dependency Link	0000	Range: 0000..FFFF
First VF Offset	4	
VF Device ID	A038	Range: 0000..FFFF
Supported Page Size	00000553	Range: 00000000..FFFFFFFF

PF1 SRIOV Config

Cap Version	1	Range: 0..F
Number of PF1 VF's	8	
PF Dependency Link	0001	Range: 0000..FFFF
First VF Offset	11	
VF Device ID	A138	Range: 0000..FFFF
Supported Page Size	00000553	Range: 00000000..FFFFFFFF

- **General SRIOV Config:** This value specifies the offset of the first PF with at least one enabled VF. When ARI is enabled, allowed value is 'd4 or 'd64, and the total number of VF in all PFs plus this field must not be greater than 256. When ARI is disabled, this field will be set to 1 to support 1PFplus 7VF non-ARI SRIOV configurations only.
- **Cap Version:** Indicates the 4-bit SR-IOV Capability version for the physical function.
- **Number of PFx VFs:** Indicates the number of virtual functions associated to the physical function. A total of 252 virtual functions are available that can be flexibly used across the four physical functions.
- **PFx Dependency Link:** Indicates the SR-IOV Functional Dependency Link for the physical function. The programming model for a device can have vendor-specific dependencies between sets of functions. The Function Dependency Link field is used to describe these dependencies.
- **First VF Offset:** Indicates the offset of the first virtual function (VF) for the physical function (PF). PF0 always resides at Offset 0, and PF1 always resides at Offset 1. Six virtual functions are available in the Gen3 Integrated Block for PCIe core and reside at the function number range 64–69. Virtual functions are mapped sequentially with VFs for PF0 taking precedence. For example, if PF0 has two virtual functions and PF1 has three, the following mapping occurs:

  The PFx_FIRST_VF_OFFSET is calculated by taking the first offset of the virtual function and subtracting that from the offset of the physical function.

  PFx_FIRST_VF_OFFSET = (PFx first VF offset - PFx offset)

  In the example above, the following offsets are used:

PF0_FIRST_VF_OFFSET = (64 - 0) = 64
PF1_FIRST_VF_OFFSET = (66 - 1) = 65

PF0 is always 64 assuming that PF0 has one or more virtual functions. The initial offset for PF1 is a function of how many VFs are attached to PF0 and is defined in the following pseudo code:

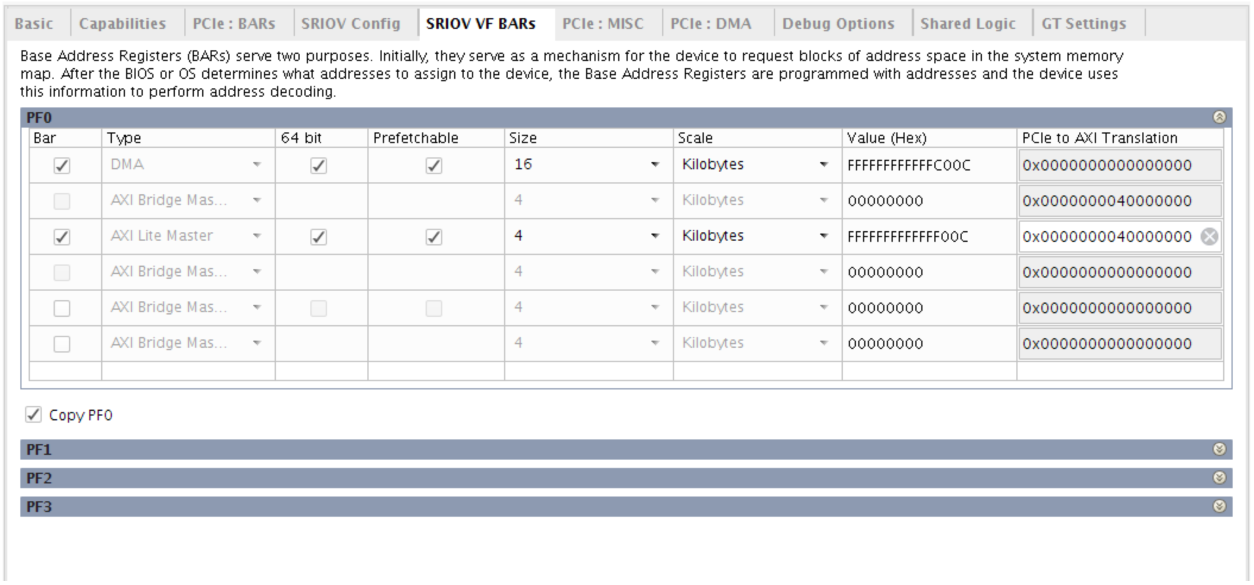
PF1_FIRST_VF_OFFSET = 63 + NUM_PF0_VFS

- **VF Device ID:** Indicates the 16-bit Device ID for all virtual functions associated with the physical function.
- **SRIOV Supported Page Size:** Indicates the page size supported by the physical function. This physical function supports a page size of 2n+12, if bit n of the 32-bit register is set.

SRIOV VF BARs Tab

The SRIOV VF BARs Tab is shown in the following figure.

Figure 28: **SRIOV VF BARs Tab**



Basic | Capabilities | PCIe : BARs | SRIOV Config | **SRIOV VF BARs** | PCIe : MISC | PCIe : DMA | Debug Options | Shared Logic | GT Settings

Base Address Registers (BARs) serve two purposes. Initially, they serve as a mechanism for the device to request blocks of address space in the system memory map. After the BIOS or OS determines what addresses to assign to the device, the Base Address Registers are programmed with addresses and the device uses this information to perform address decoding.

PF0

Bar	Type	64 bit	Prefetchable	Size	Scale	Value (Hex)	PCIe to AXI Translation
✓	DMA	✓	✓	16	Kilobytes	FFFFFFFFFFFFC00C	0x0000000000000000
	AXI Bridge Mas...			4	Kilobytes	00000000	0x0000000040000000
✓	AXI Lite Master	✓	✓	4	Kilobytes	FFFFFFFFFFFFF00C	0x0000000040000000
	AXI Bridge Mas...			4	Kilobytes	00000000	0x0000000000000000
	AXI Bridge Mas...			4	Kilobytes	00000000	0x0000000000000000
	AXI Bridge Mas...			4	Kilobytes	00000000	0x0000000000000000

✓ Copy PF0

PF1

PF2

PF3

The SRIOV VF BARs tab enables you to configure the base address registers (BARs) for all virtual function (VFs) within a virtual function group (VFG). All the VFs within the same VFG share the same BASE ADDRESS Registers (BARS) configurations. Each Virtual Function supports up to six 32-bit BARs or three 64-bit BARs. Virtual Function BARs can be configured without any dependency on the settings of the associated Physical Functions BARs,

- **BAR:** Select applicable BARs using the checkboxes.
- **Type:** Select the relevant option:

- **DMA:** Is fixed to BAR0 space.
- **AXI Lite Master:** Is fixed to BAR1 space.
- **AXI Bridge Master:** Is fixed to BAR2 space.For all other bars, select either AXI Lite Master or AXI Bridge Master.

***Note*:** The current IP supports at most one DMA BAR (or a management BAR given only mailbox is required) for one VF. The other bars can be configured as AXI Lite Master to access the assigned memory space through the AXI4-Lite bus. Virtual Function BARs do not support I/O space and must be configured to map to the appropriate memory space.

- **64-bit:**

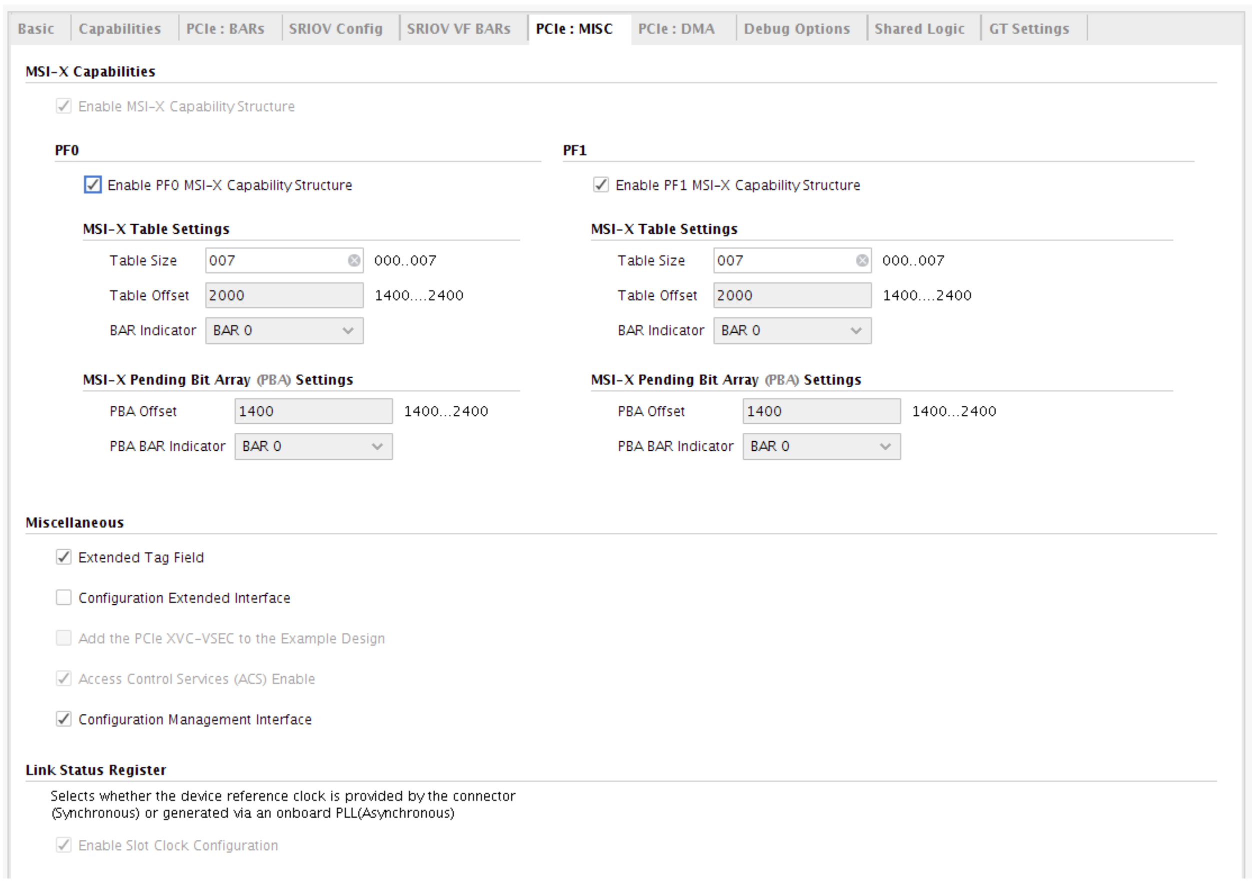
  VF BARs can be either 64-bit or 32-bit. The default is 64-bit BAR.

  - 64-bit addressing is supported for the DMA bar.
  - When a BAR is set as 64 bits, it uses the next BAR for the extended address space and makes the next BAR inaccessible.
  - No VF bar can be configured as **Prefetchable**.
- **Size:** The available Size range depends on the 32-bit or 64-bit bar selected. The Supported Page Sizes field indicates all the page sizes supported by the PF and, as required by the SR-IOV specification. Based on the Supported Page Size field, the system software sets the System Page Size field which is used to map the VF BAR memory addresses. Each VF BAR address is aligned to the system page boundary.
- **Value:** The value assigned to the BAR based on the current selections.

PCIe MISC Tab

The PCIe Miscellaneous Tab is shown in the following figure.

Figure 29: **PCIe MISC Tab**



MSI-X Capabilities

✓ Enable MSI-X Capability Structure

PF0

✓ Enable PF0 MSI-X Capability Structure

MSI-X Table Settings

Table Size	007	000..007
Table Offset	2000	1400....2400
BAR Indicator	BAR 0	

MSI-X Pending Bit Array (PBA) Settings

PBA Offset	1400	1400...2400
PBA BAR Indicator	BAR 0	

PF1

✓ Enable PF1 MSI-X Capability Structure

MSI-X Table Settings

Table Size	007	000..007
Table Offset	2000	1400....2400
BAR Indicator	BAR 0	

MSI-X Pending Bit Array (PBA) Settings

PBA Offset	1400	1400...2400
PBA BAR Indicator	BAR 0	

Miscellaneous

✓ Extended Tag Field

☐ Configuration Extended Interface

☐ Add the PCIe XVC-VSEC to the Example Design

✓ Access Control Services (ACS) Enable

✓ Configuration Management Interface

Link Status Register

Selects whether the device reference clock is provided by the connector (Synchronous) or generated via an onboard PLL(Asynchronous)

✓ Enable Slot Clock Configuration

- **MSI-X Capabilities:** MSI-X is enabled by default.The MSI-X settings for different physical functions can be set as required.
- **MSIx Table Settings:** Defines the MSI-X Table Structure.
  - **Table Size:** Specifies the MSI-X Table size. The default is 8 (8 interrupt vectors per function).
  - **Table Offset:** Specifies the offset from the Base addres Register (BAR) in DMA configiration space used to map function in MSI-X Table onto memory space. Table space is fixed at offset 0x2000.
  - **BAR Indicator:** Is fixed to DMA BAR which is BAR0.
- **MSI-X Pending Bit Array Settings:**
  - **PBA Offset:** Specifies the offset from the DMA BAR register that point so the base of MSI-X PDB. Table space is fixed at offset 0x1400.

- **PBA BAR Indicator:** Is fixed to DMA BAR which is BAR0.

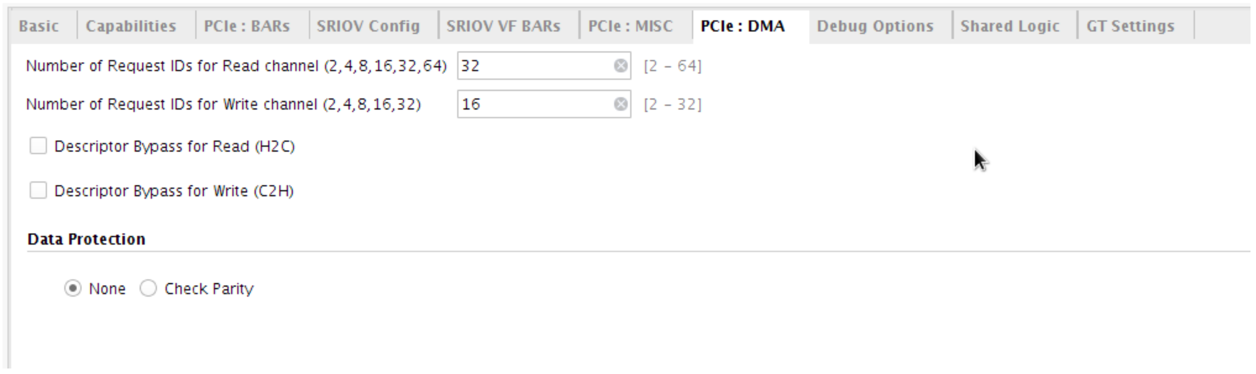
- **Finite Completion Credits:** In systems which support fine completion credits, this option can be enabled for better performance.
- **Extended Tag:** By default for UltraScale+™ devices the Extended Tab option gives 256 Tags. If Extended Tag option is not selected DMA will use 32 tags.
- **Configuration Extended Interface:** The PCIe extended interface can be selected for more configuration space. When Configuration Extednd Interface is selected user is responsible for adding logic to extend the interface to make it work properly.
- **Access Control Server (ACS) Enable:**

  ACS is selected by default.
- **Configuration Management Interface:** The PCIe Configuration Management interface can be enabled and brought to the top level when this option is selected.

PCIe DMA Tab

The PCIe DMA Tab is shown in the following figure.

Figure 30: **PCIe DMA Tab**



Number of Request IDs for Read channel (2,4,8,16,32,64) 32 [2 - 64]

Number of Request IDs for Write channel (2,4,8,16,32) 16 [2 - 32]

☐ Descriptor Bypass for Read (H2C)

☐ Descriptor Bypass for Write (C2H)

Data Protection

[✓ None] Check Parity

- **Number of Request IDs for Read channel:** Select the maximum number of outstanding request per channel. Select from 2 to 64.
- **Number of Request IDs for Write channel:** Select maximum number of outstanding request per channel. Select from 2 to 32.
- **Descriptor Bypass for Read (H2C):** This option enables the desciptor bypass output and input ports for H2C transfer. Note that only context setting determines if the descripor is sent out.
- **Descriptor Bypass for Write (C2H):** This option enables the descriptor bypass output and input ports for C2H transfer. Note that only context settings determine if the descriptor is sent out.

- **Data Protection:** Parity Checking: The default is no parity checking.When **Check Parity** is enabled, the QDMA checks for parity on read data from the PCIe and generates parity for write data to the PCIe.

User Parameters

This section does not apply to this subsystem.

Output Generation

For details, see the *Vivado Design Suite User Guide: Designing with IP* (UG896).

Constraining the Subsystem

Required Constraints

The QDMA Subsystem for PCIe requires the specification of timing and other physical implementation constraints to meet specified performance requirements for PCI Express®. These constraints are provided in a Xilinx Design Constraints (XDC) file. Pinouts and hierarchy names in the generated XDC correspond to the provided example design.

IMPORTANT!: *If the example design top file is not used, copy the IBUFDS_GTE4 instance for the reference clock, IBUF Instance for* `sys_rst` *and also the location and timing constraints associated with them into your local design top.*

To achieve consistent implementation results, an XDC containing these original, unmodified constraints must be used when a design is run through the Xilinx® tools. For additional details on the definition and use of an XDC or specific constraints, see *Vivado Design Suite User Guide: Using Constraints* (UG903).

Constraints provided with the Integrated Block for PCIe solution have been tested in hardware and provide consistent results. Constraints can be modified, but modifications should only be made with a thorough understanding of the effect of each constraint. Additionally, support is not provided for designs that deviate from the provided constraints.

Device, Package, and Speed Grade Selections

The device selection portion of the XDC informs the implementation tools which part, package, and speed grade to target for the design.

The device selection section always contains a part selection line, but can also contain part or package-specific options. An example part selection line follows:

CONFIG PART = xcvu9p-flgb2104-2-i

Clock Frequencies

For detailed information about clock requirements, see the *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213).

Clock Management

For detailed information about clock requirements, see the *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213).

Clock Placement

For detailed information about clock requirements, see the *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213).

Banking

This section is not applicable for this IP subsystem.

Transceiver Placement

This section is not applicable for this IP subsystem.

I/O Standard and Placement

This section is not applicable for this IP subsystem.

Relocating the Integrated Block Core

By default, the IP core-level constraints lock block RAMs, transceivers, and the PCIe block to the recommended location. To relocate these blocks, you must override the constraints for these blocks in the XDC constraint file. To do so:

1. Copy the constraints for the block that needs to be overwritten from the core-level XDC constraint file.
2. Place the constraints in the user XDC constraint file.
3. Update the constraints with the new location.

The user XDC constraints are usually scoped to the top-level of the design; therefore, ensure that the cells referred by the constraints are still valid after copying and pasting them. Typically, you need to update the module path with the full hierarchy name.

Note: If there are locations that need to be swapped (that is, the new location is currently being occupied by another module), there are two ways to do this:

- If there is a temporary location available, move the first module out of the way to a new temporary location first. Then, move the second module to the location that was occupied by the first module. Next, move the first module to the location of the second module. These steps can be done in XDC constraint file.
- If there is no other location available to be used as a temporary location, use the `reset_property` command from Tcl command window on the first module before relocating the second module to this location. The `reset_property` command cannot be done in the XDC constraint file and must be called from the Tcl command file or typed directly into the Tcl Console.

Simulation

For comprehensive information about Vivado® simulation components, as well as information about using supported third-party tools, see the *Vivado Design Suite User Guide: Logic Simulation* (UG900).

Basic Simulation

Simulation models for AXI-MM and AXI-ST options can be generated and simulated. The simple simulation model options enabled you to develop complex designs.

AXI-MM Mode

The example design for the AXI4 Memory Mapped (AXI-MM) mode has 512 KB block RAM on the user side, so data can be written to the block RAM and read from block RAM to the Host. After H2C transfer is started DMA reads data from the Host memory and writes to the block RAM. Then, the C2H transfer is started and the DMA reads data from the block RAM and writes to the Host memory. The original data is compared with the C2H write data. H2C and C2H are setup with one descriptor each, and the total transfer size is 128 bytes.

More detailed steps are described in Reference Software Driver Flow.

AXI-ST Mode

The example design for the AXI4-Stream (AXI_ST) mode has data checker to check the data from H2C transfer and has data genertator for C2H transfer.

After H2C transfer is started the DMA engine reads data from the Host memory and writes to the user side. Once the transfer is completed DMA updated Write Back status and generates Interrupt (if enabled). The data checker on the user side checks for a predefined data to be present, and the result is posted in a predefined address for the user to read.

After C2H transfer is started the data generator, the user side generates predefine data and associated control signals. The DMA transfers data to the Host, updates the write back status, and generates interrupt (if enabled).

H2C and C2H are setup with one descriptor each, and the total transfer size is 128 bytes.

More detailed steps are described in Reference Software Driver Flow.

PIPE Mode Simulation

The QDMA Subsystem for PCIe supports the PIPE mode simulation where the PIPE interface of the core is connected to the PIPE interface of the link partner. This mode increases the simulation speed.

Use the **Enable PIPE Simulation** option on the Basic tab of the Customize IP dialog box to enable PIPE mode simulation in the current Vivado® Design Suite solution example design, in either Endpoint mode or Root Port mode. The External PIPE Interface signals are generated at the core boundary for access to the external device. Enabling this feature also provides the necessary hooks to use third-party PCI Express® VIPs/BFMs instead of the Root Port model provided with the example design.

The tables below describe the PIPE bus signals available at the top level of the core and their corresponding mapping inside the EP core (`pcie_top`) PIPE signals.

Table 287: **In Commands and Endpoint PIPE Signal Mappings**

In Commands	Endpoint PIPE Signals Mapping
common_commands_in[25:0]	not used

Table 288: **Out Commands and Endpoint PIPE Signal Mappings**

Out Commands	Endpoint PIPE Signals Mapping
common_commands_out[0]	pipe_clk[1]
common_commands_out[2:1]	pipe_tx_rate_gt[2]
common_commands_out[3]	pipe_tx_rcvr_det_gt
common_commands_out[6:4]	pipe_tx_margin_gt
common_commands_out[7]	pipe_tx_swing_gt
common_commands_out[8]	pipe_tx_reset_gt
common_commands_out[9]	pipe_tx_deemph_gt

Table 288: **Out Commands and Endpoint PIPE Signal Mappings** *(cont'd)*

Out Commands	Endpoint PIPE Signals Mapping
common_commands_out[16:10]	not used[3]

Notes:

1. pipe_clk is an output clock based on the core configuration. For Gen1 rate, pipe_clk is 125 MHz. For Gen2 and Gen3, pipe_clk is 250 MHz
2. pipe_tx_rate_gt indicates the pipe rate (2’b00-Gen1, 2’b01-Gen2, and 2’b10-Gen3)
3. The functionality of this port has been deprecated and it can be left unconnected.

Table 289: **Input Bus With Endpoint PIPE Signal Mapping**

Input Bus	Endpoint PIPE Signal Mapping
pipe_rx_0_sigs[31:0]	pipe_rx0_data_gt
pipe_rx_0_sigs[33:32]	pipe_rx0_char_is_k_gt
pipe_rx_0_sigs[34]	pipe_rx0_elec_idle_gt
pipe_rx_0_sigs[35]	pipe_rx0_data_valid_gt
pipe_rx_0_sigs[36]	pipe_rx0_start_block_gt
pipe_rx_0_sigs[38:37]	pipe_rx0_syncheader_gt
pipe_rx_0_sigs[83:39]	not used

Table 290: **Output Bus with Endpoint PIPE Signal Mapping**

Output Bus	Endpoint PIPE Signals Mapping
pipe_tx_0_sigs[31: 0]	pipe_tx0_data_gt
pipe_tx_0_sigs[33:32]	pipe_tx0_char_is_k_gt
pipe_tx_0_sigs[34]	pipe_tx0_elec_idle_gt
pipe_tx_0_sigs[35]	pipe_tx0_data_valid_gt
pipe_tx_0_sigs[36]	pipe_tx0_start_block_gt
pipe_tx_0_sigs[38:37]	pipe_tx0_syncheader_gt
pipe_tx_0_sigs[39]	pipe_tx0_polarity_gt
pipe_tx_0_sigs[41:40]	pipe_tx0_powerdown_gt
pipe_tx_0_sigs[69:42]	not used[1]

Notes:

1. The functionality of this port has been deprecated and it can be left unconnected.

Synthesis and Implementation

For details about synthesis and implementation, see the *Vivado Design Suite User Guide: Designing with IP* (UG896).

Chapter 6

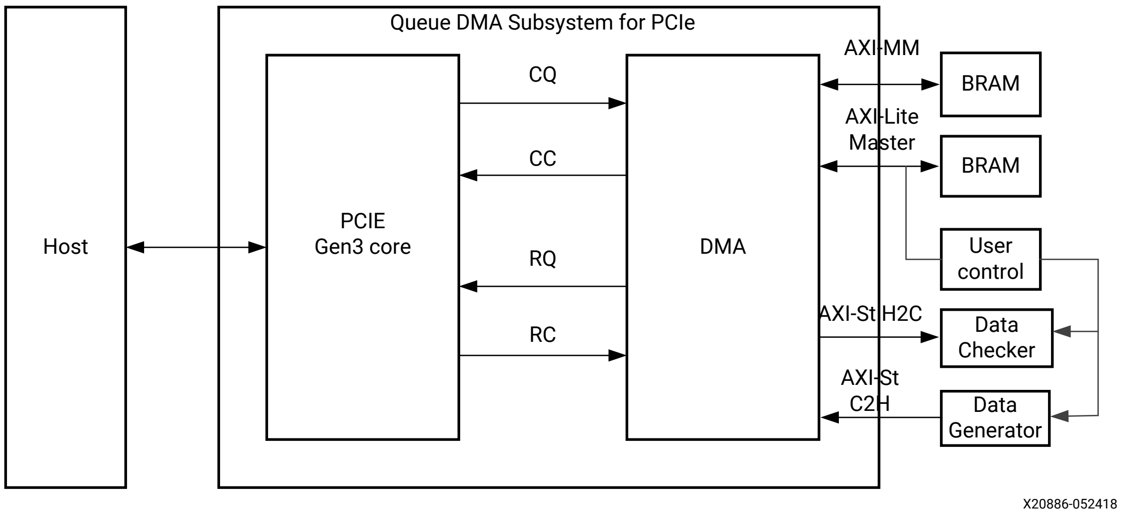
Example Design

This chapter contains information about the four example design provided in the Vivado® Design Suite.

AXI4 Memory Mapped and AXI4-Stream Default Example Design

The following is the block diagram of the AXI4 memory mapped and AXI4-Stream example design block.

Figure 31: **Default Example Design**



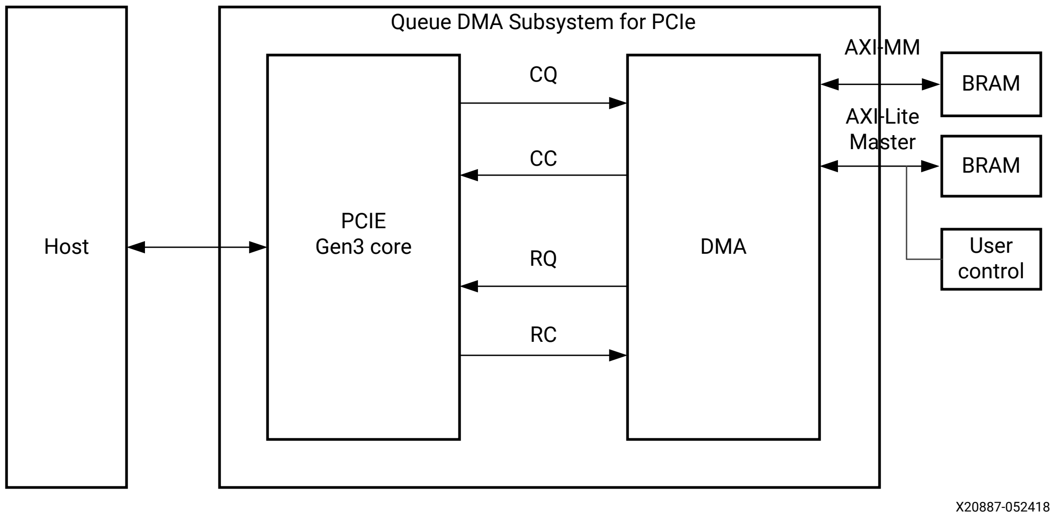
In order to test the AXI4-Stream and AXI4 Memory Mapped interface, there is some logic implemented in FPGA. When the example design is generated for QDMA Subsystem for PCIe, the modules that are generated are for testing purposes only. In the example design:

- The AXI4 MM interface is connected to the 512 KBytes block RAM.
- The AXI4-Stream interface is connected to custom data generator and data checker module.
- The data generator and checker works only with predefined pattern, which is a 16-bit incremental pattern starting with 0. This data file is included in driver package.

The pattern generator and checker can be controlled using the registers found in Example Design Registers. These registers can only be controlled through the AXI4-Lite Master interface. To test the QDMA Subsystem for PCIe's AXI4-Stream interface, ensure that the AXI4-Lite Master interface is present on BAR1 when using the example design.

AXI Memory Mapped Example Design

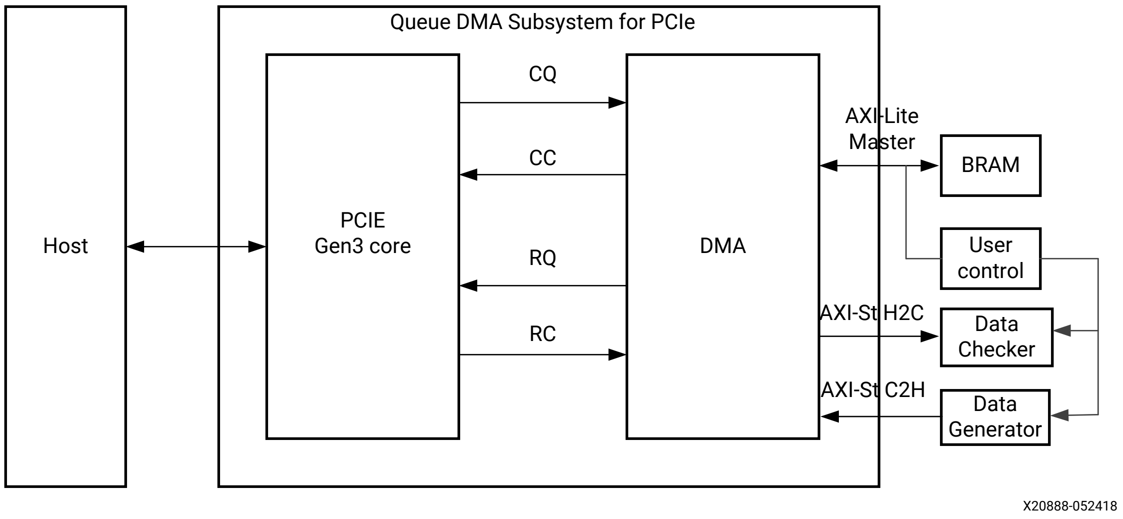
Figure 32: **AXI4 Memory Map Example Design**



The example design above is generated when you select **AXI-MM only** in the DMA Interface Selection option in the Basic Tab. In this mode, the AXI-MM interface is connected to a 512 KBytes block RAM. The diagram above shows where the BAR1 'AXI-Lite Master' is conneted to a 4 KBytes block RAM. H2C transfers read data from from Host and writes to block RAM. C2H transfers read data from block RAM and writes to Host memory.

AXI Stream Example Design

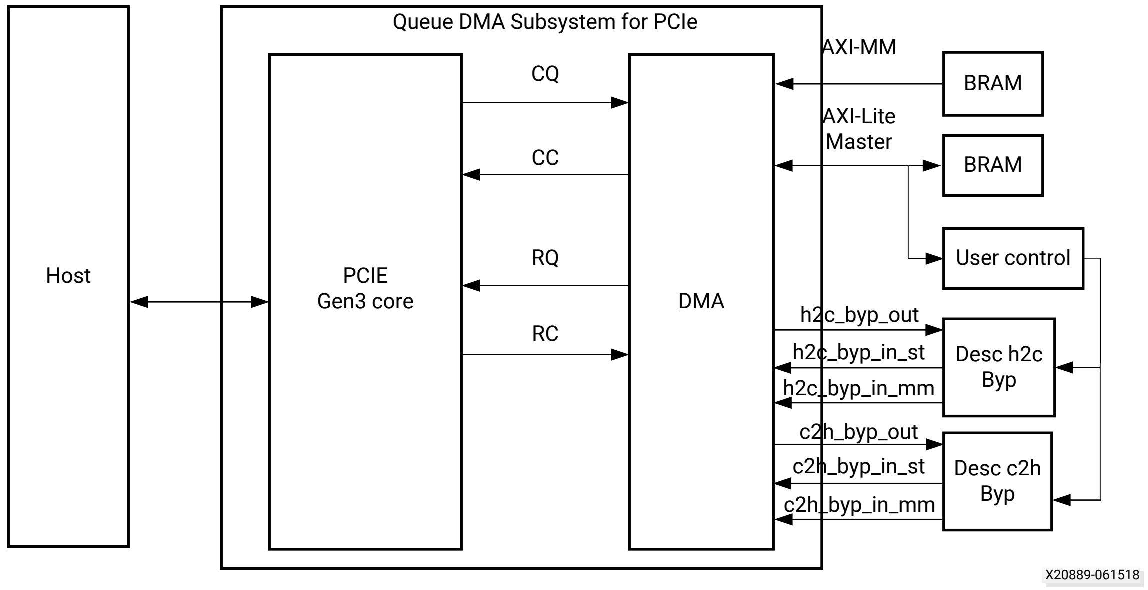
Figure 33: **AXI4-Stream Example Design**



The example design above is generated when you select the **AXI-ST only** option in the DMA Interface Selection option in the Basic Tab. In this mode, the AXI-ST H2C interface is connected to a data checker, and the AXI-ST C2H interface is connected to data generator. The diagram above shows where BAR1 'AXI-Lite Master' is connected to 4 KBytes block RAM and user control logic. The software can control data checker and data generator though the AXI4-Lite Master interface. The data generator and checker work only with a predefined pattern, which is a 16-bit incremental pattern starting with 0. This data file is included in the driver package.

Example Design with Descriptor Bypass In/Out Loopback

Figure 34: **AXI4 Memory Map and Descriptor Bypass Example Design**



The example design above is generated when you select the **Descriptor Bypass for Read (H2C)** and **Descriptor Bypass for Write (C2H)** options in the PCIe DMA Tab. These options can be selected with all three options 1) AXI-MM, 2) AXI-ST and 3) AXI-MM and AXI-ST options. The Descriptor Bypass in/out loopback is controlled by AXI4-Lite master by writing to the Example Design Register 0x90 bit 0 and 1.

To enable Descriptor bypass out, proper context programming needs to be done. Refer to Context Programming section.

Example Design Registers

Table 291: **Example Design Registers**

Registers	Address	Description
C2H_ST_QID (0x000)	0x000	AXI-ST C2H Queue id
C2H_ST_LEN (0x004)	0x004	AXI-ST C2H transfer length
C2H_CONTROL_REG (0x008)	0x008	AXI-ST C2H pattern generator control
H2C_CONTROL_REG (0x00C)	0x00C	AXI-ST H2C Control

Table 291: **Example Design Registers** *(cont'd)*

Registers	Address	Description
H2C_STATUS (0x010)	0x010	AXI-ST H2C Status
C2H_PACKET_COUNT (0x020)	0x020	AXI-ST C2H number of packets to transfer
C2H_COMPLETION_DATA_0 (0x030) to C2H_COMPLETION_DATA_7 (0x04C)	0x4C-0x030	AXI-ST C2H Write back data
C2H_COMPLETION_SIZE (0x050)	0x050	AXI-ST C2H Write data size.
SCRATCH_REG0 (0x060)	0x060	Scratch register 0
SCRATCH_REG1 (0x064)	0x064	Scratch register 1
C2H_PACKETS_DROP (0x088)	0x088	AXI-ST C2H Packets drop count
C2H_PACKETS_ACCEPTED (0x08C)	0x08C	AXI-ST C2H Packets accepted count
Descriptor Bypass (0x090)	0x090	C2H and H2C descriptor bypass loopback
User interrupt (0x094)	0x094	User interrupt, vector number, function number
User Interrupt Mask (0x098)	0x098	User interrupt mask
User Interrupt Vector (0x09C)	0x09C	User interrupt vector
DMA Control (0x0A0)	0x0A0	DMA control
VDM Message Read (0x0A4)	0x0A4	VDM message read

C2H_ST_QID (0x000)

Table 292: **C2H_ST_QID (0x000)**

Bit	Default	Access Type	Field	Description
[31:11]	0	NA		Reserved
[10:0]	0	RW	C2h_st_qid	AXI- Streaming C2h Queue id

C2H_ST_LEN (0x004)

Table 293: **C2H_ST_LEN (0x004)**

Bit	Default	Access Type	Field	Description
[31:16]	0	NA		Reserved
[15:0]	0	RW	C2h_st_len	AXI- Streaming C2h Queue id

C2H_CONTROL_REG (0x008)

Table 294: **C2H_CONTROL_REG (0x008)**

Bit	Default	Access Type	Field	Description
[31:3]	0	NA		Reserved
[2]	0	RW		Immediate data. When set data generator will send immedisate data.
[1]	0	RW		Start AXI-ST C2H transfer
[0]	0	RW		Streaming loop back. When set the data packet from H2C streamig port in Card side will be looked back to C2H streaming ports.

For Normal C2H stream packet tranfer, set address offset 0x08 to 0x2.

For C2H immediate data transfer, set address offset 0x8 to 0x6.

For C2H/H2C stream loopback, set address offset 0x8 to 0x1.

H2C_CONTROL_REG (0x00C)

Table 295: **H2C_CONTROL_REG (0x00C)**

Bit	Default	Access Type	Field	Description
[31:1]	0	NA		Reserved
[0]	0	RW		Clear match bit for H2C transfer

H2C_STATUS (0x010)

Table 296: **H2C_STATUS (0x010)**

Bit	Default	Access Type	Field	Description
[31:15]	0	NA		Reserved
[14:4]	0	R		H2C transfer Queue ID
[3:1]	0	NA		Reserved
[0]	0	R		H2C transfer match

C2H_PACKET_COUNT (0x020)

Table 297: **C2H_PACKET_COUNT (0x020)**

Bit	Default	Access Type	Field	Description
[31:10]	0	NA		Reserved
[9:0]	0	RW		AXI-ST C2H number of packet to transfer

C2H_COMPLETION_DATA_0 (0x030)

Table 298: **C2H_COMPLETION_DATA_0 (0x030)**

Bit	Default	Access Type	Field	Description
[31:0]	0	NA		AXI-ST C2H Completion Data [31:0]

C2H_COMPLETION_DATA_1 (0x034)

Table 299: **C2H_COMPLETION_DATA_1 (0x034)**

Bit	Default	Access Type	Field	Description
[31:0]	0	NA		AXI-ST C2H Completion Data [63:32]

C2H_COMPLETION_DATA_2 (0x038)

Table 300: **C2H_COMPLETION_DATA_2 (0x038)**

Bit	Default	Access Type	Field	Description
[31:0]	0	NA		AXI-ST C2H Completion Data [95:64]

C2H_COMPLETION_DATA_3 (0x03C)

Table 301: **C2H_COMPLETION_DATA_3 (0x03C)**

Bit	Default	Access Type	Field	Description
[31:0]	0	NA		AXI-ST C2H Completion Data [127:96]

C2H_COMPLETION_DATA_4 (0x040)

Table 302: **C2H_COMPLETION_DATA_4 (0x040)**

Bit	Default	Access Type	Field	Description
[31:0]	0	NA		AXI-ST C2H Completion Data [159:128]

C2H_COMPLETION_DATA_5 (0x044)

Table 303: **C2H_COMPLETION_DATA_5 (0x044)**

Bit	Default	Access Type	Field	Description
[31:0]	0	NA		AXI-ST C2H Completion Data [191:160]

C2H_COMPLETION_DATA_6 (0x048)

Table 304: **C2H_COMPLETION_DATA_6 (0x048)**

Bit	Default	Access Type	Field	Description
[31:0]	0	NA		AXI-ST C2H Completion Data [223:192]

C2H_COMPLETION_DATA_7 (0x04C)

Table 305: **C2H_COMPLETION_DATA_7 (0x04C)**

Bit	Default	Access Type	Field	Description
[31:0]	0	NA		AXI-ST C2H Completion Data [255:224]

C2H_COMPLETION_SIZE (0x050)

Table 306: **C2H_COMPLETION_SIZE (0x050)**

Bit	Default	Access Type	Field	Description
[31:0]	0	NA		Reserved

Table 306: **C2H_COMPLETION_SIZE (0x050)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[1:0]	0	RW		AXI-St C2H completion data size 00 : 8 Bytes 01 : 16 bytes 10: 32 Bytes 11 : Reserved

SCRATCH_REG0 (0x060)

Table 307: **SCRATCH_REG0 (0x060)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW		Scratch register

SCRATCH_REG1 (0x064)

Table 308: **SCRATCH_REG1 (0x064)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW		Scratch register

C2H_PACKETS_DROP (0x088)

Table 309: **C2H_PACKETS_DROP (0x088)**

Bit	Default	Access Type	Field	Description
[31:0]	0	R		AXI-ST C2H packet (descriptor) drop per transfer

Each AXI-ST C2H transfer can contain one or more descriptor depending on transfer size and C2H buffer size. This register represents how many of the descriptors were dropped in current transfer. This register will reset to 0 in beginning of transfer.

C2H_PACKETS_ACCEPTED (0x08C)

Table 310: **C2H_PACKETS_ACCEPTED (0x08C)**

Bit	Default	Access Type	Field	Description
[31:0]	0	R		AX-st C2H packet(descriptor) accepted per transfer

Each AXI-St C2H transfer can contain one or more descriptor depending on transfer size and C2H buffer size. This register represents how many of the descriptors were accepted in current transfer. This register will reset to 0 in beginning of transfer.

Descriptor Bypass (0x090)

Table 311: **Descriptor bypass (0x090)**

Bit	Default	Access Type	Field	Description
[31:2]	0	NA		Reserved
[1]	0	RW	c2h_dsc_bypass	C2H descriptor bypass loopback. When set C2H descriptor bypass out will be looked back to C2H descriptir bypss in ports.
[0]	0	RW	h2c_dsc_bypass	H2C descriptor bypass loopback. When set H2C descriptor bypass out will be looked back to H2C descriptor bypass in ports.

User interrupt (0x094)

Table 312: **User interrupt (0x094)**

Bit	Default	Access Type	Field	Description
[31:20]	0	NA		Reserved
[19:12]	0	RW	usr_irq_in_fun	User interrupt function number
[11:9]	0	NA		Reserved
[8:4]	0	RW	usr_irq_in_vec	User interrupt vector number
[3:1]	0	NA		Reserved
[0]	0	RW	usr_irq	User interrupt. When set example design will generate a user interrupt

To generate a user interrupt:

1. Write the function number at bits[19:12]. This corresponds to which function generates the `usr_irq_in_fnc` user interrupt.
2. Write MSI-X Vector number at bits [8:4]. This corresponds to which entry in MSI-X table is set up for `usr_irq_in_vec` user interrupt.
3. Write 1 to bit [0] to generate user interrupt. This bit clears itself after an ack, `usr_irq_out_ack`, from DMA is generated.

All three above setps can be done at same time (with a single write).

User Interrupt Mask (0x098)

Table 313: **User Interrupt Mask (0x098)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW		User Interrupt Mask

User Interrupt Vector (0x09C)

Table 314: **User Interrupt Vector (0x09C)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW		User Interrupt Vector

The `user_interrupt_mask[31:0]` and `user_interrupt_vector[31:0]` registers are provided as an example design for user interrupt aggregation that can generate a user interrupt for a function. The `user_interrupt_mask[31:0]` is anded (bitwire and) with `user_interrupt_vector[31:0]` and an user interrupt is generated. The `user_interrupt_vector[31:0]` is clear on read register.

To generate a user interrupt:

1. Write the function number at `user_interrupt[19:12]`. This corresponds to which function generates the `usr_irq_in_fnc` user interrupt.
2. Write the MSI-X Vector number at `user_interrupt[8:4]`. This corresponds to which entry in MSI-X table is set up for the `usr_irq_in_vec` user interrupt.
3. Write mask value in the `user_interrupt_mask[31:0]` register.
4. Write the interrupt vector value in the `user_interrupt_vector[31:0]` register.

This generates a user interrupt to the DMA block.

There are two way to generate user interrupt:

- Write to `user_interrupt[0]`, or
- Write to the `user_interrupt_vector[31:0]` register with mask set.

DMA Control (0x0A0)

Table 315: **DMA Control (0x0A0)**

Bit	Default	Access Type	Field	Description
[31:1]		NA		Reserved
[0]	0	RW	gen_qdma_reset	When soft_reset is set, generates a soft reset to the DMA block. This bit is cleared after 100 cycles.

Writing a 1 to `DMA_control[0]` generates a soft reset on `soft_reset_n`(active-Low). A reset is asserted for 100 cycles, and after that, the signal will be deasserted.

VDM Message Read (0x0A4)

Table 316: **VDM Message Read (0x0A4)**

Bit	Default	Access Type	Field	Description
[31:0]		RO		VDM message read

Vendor Defined Message (VDM) messages, `st_rx_msg_data`, are stored in fifo in the example design. A read to this register (0x0A4) will pop out one 32-bit message at a time.

Upgrading

Comparing With DMA/Bridge Subsystem for PCI Express

The table below describes the differences between the DMA/Bridge Subsystem for PCI Express® and QDMA Subsystem for PCI Express.

Table 317: **Comparing Subsystems**

	DMA/Bridge Subsystem	QDMA Subsystem
Configuration	Up to Gen3x16.	Up to Gen3x16.
Channels/Queues	4 H2C, 4 C2H channels with 1PF.	Up to 2K queues (All can be assigned to one PF or distributed amongst all 4).
SR-IOV	Not Supported.	Supported (4 PF, 252 VFs).
User Interface	Configured with AXI-MM OR AXI-ST, but not both.	Each queue will have a context which will tell whether it goes to a AXI4-Memory or AXI4-Stream.
User Interrupts	Up to 16 user interrupts.	Interrupt aggregation per function.
Device Support	Supported for 7 Series Gen2 to UltraScale+™ devices.	Only supported for UltraScale+ devices.
Interrupts	Legacy, MSI, MSI-X supported.	MSI-X Supported for PFs. Only MSI-X Supported for VFs.
Driver Support	Linux, Windows Example Drivers.	Linux, DPDK. Windows Driver (in a future release).

Appendix B

Debugging

This appendix includes details about resources available on the Xilinx Support website and debugging tools.

Finding Help on Xilinx.com

To help in the design and debug process when using the subsystem, the Xilinx Support web page contains key resources such as product documentation, release notes, answer records, information about known issues, and links for obtaining further product support.

Documentation

This product guide is the main document associated with the subsystem. This guide, along with documentation related to all products that aid in the design process, can be found on the Xilinx Support web page or by using the Xilinx® Documentation Navigator. Download the Xilinx Documentation Navigator from the Downloads page. For more information about this tool and the features available, open the online help after installation.

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

The Solution Center specific to the QDMA Subsystem for PCIe is the Xilinx Solution Center for PCI Express.

Answer Records

Answer Records include information about commonly encountered problems, helpful information on how to resolve these problems, and any known issues with a Xilinx product. Answer Records are created and maintained daily ensuring that users have access to the most accurate information available.

Answer Records for this subsystem can be located by using the Search Support box on the main Xilinx support web page. To maximize your search results, use keywords such as:

- Product name
- Tool message(s)
- Summary of the issue encountered

A filter search is available after results are returned to further target the results.

Master Answer Record for the Subsystem

AR 70927.

Technical Support

Xilinx provides technical support in the Xilinx Support web page for this LogiCORE™ IP product when used as described in the product documentation. Xilinx cannot guarantee timing, functionality, or support if you do any of the following:

- Implement the solution in devices that are not defined in the documentation.
- Customize the solution beyond that allowed in the product documentation.
- Change any section of the design labeled DO NOT MODIFY.

To contact Xilinx Technical Support, navigate to the Xilinx Support web page.

Debug Tools

There are many tools available to address QDMA Subsystem for PCIe design issues. It is important to know which tools are useful for debugging various situations.

Vivado Design Suite Debug Feature

The Vivado® Design Suite debug feature inserts logic analyzer and virtual I/O cores directly into your design. The debug feature also allows you to set trigger conditions to capture application and integrated block port signals in hardware. Captured signals can then be analyzed. This feature in the Vivado IDE is used for logic debugging and validation of a design running in Xilinx® devices.

The Vivado logic analyzer is used to interact with the logic debug LogiCORE IP cores, including:

- ILA 2.0 (and later versions)
- VIO 2.0 (and later versions)

See the *Vivado Design Suite User Guide: Programming and Debugging* (UG908).

Hardware Debug

Hardware issues can range from link bring-up to problems seen after hours of testing. This section provides debug steps for common issues. The Vivado® debug feature is a valuable resource to use in hardware debug. The signal names mentioned in the following individual sections can be probed using the debug feature for debugging the specific problems.

General Checks

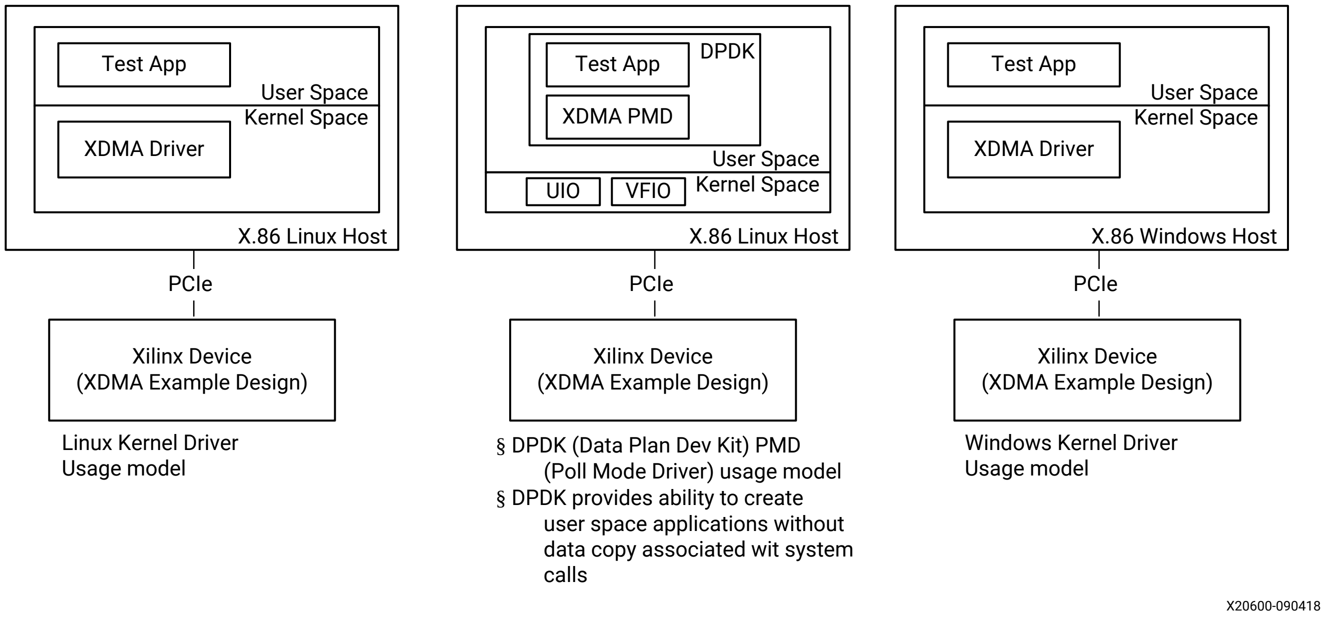
Ensure that all the timing constraints for the core were properly incorporated from the example design and that all constraints were met during implementation.

- Does it work in post-place and route timing simulation? If problems are seen in hardware but not in timing simulation, this could indicate a PCB issue. Ensure that all clock sources are active and clean.
- If using MMCMs in the design, ensure that all MMCMs have obtained lock by monitoring the locked port.
- If your outputs go to 0, check your licensing.

Application Software Development

Device Drivers

Figure 35: **Device Drivers**

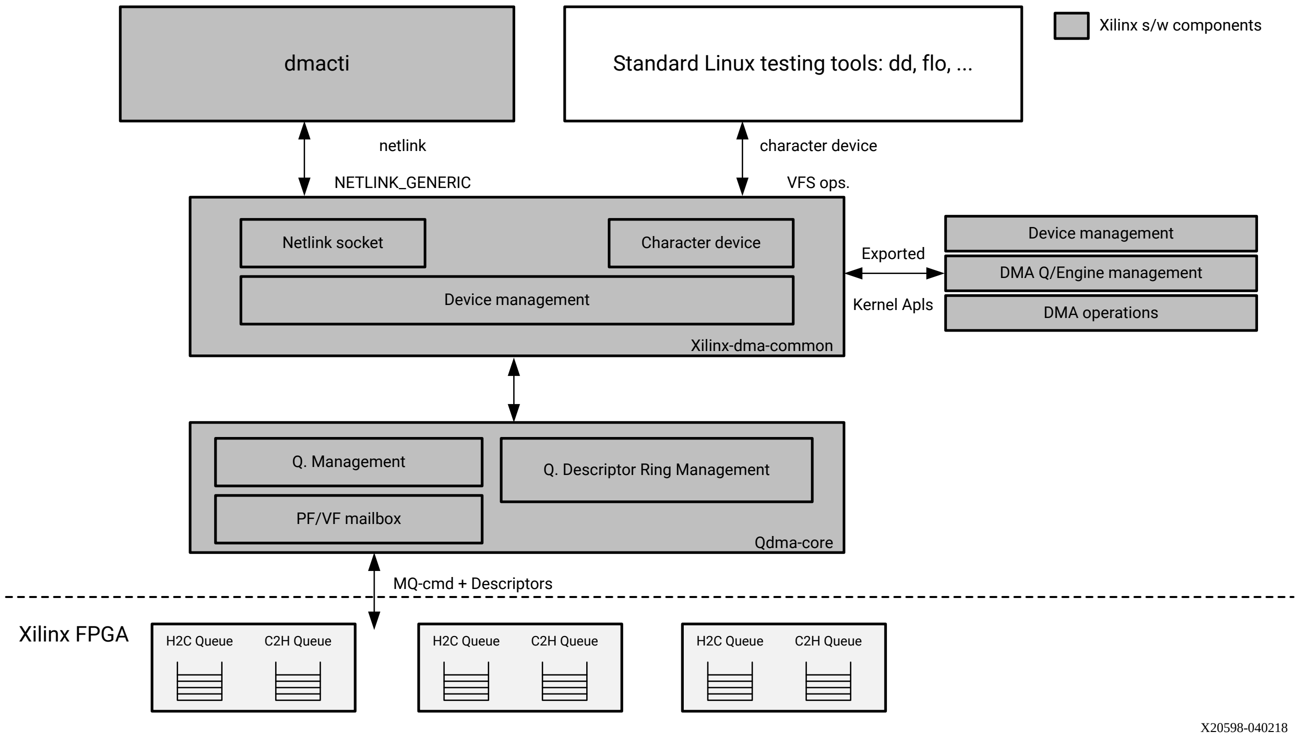


The above figure shows the usage model of Linux and Windows QDMA software drivers. The QDMA Subsystem for PCIe example design is implemented on a Xilinx® FPGA, which is connected to an X86 host through PCI Express.

- In the first use mode, the QDMA driver in kernel space runs on Linux, whereas the test application runs in user space.
- In the second use mode, the Data Plan Dev Kit (DPDK) is used to develop a QDMA Poll Mode Driver (PMD) running entirely in the user space, and use the UIO and VFIO kernel framework to communicate with the FPGA.
- In the third usage mode, the QDMA driver runs in kernel space on Windows, whereas the test application runs in the user space.

Linux DMA Software Architecture (PF/VF)

Figure 36: **Linux DMA Software Architecture**



The QDMA driver consists of the following three major components:

- **Device control tool**: Creates a netlink socket for PCIe device query, queue management, reading the context of a queue, etc.
- **DMA tool**: Is the user space application to initiate a DMA transaction. You can use standard Linux utility `dd` or `fio`, or use the example application in the driver package.
- **Kernel space driver**: Creates the descriptors and translates the user space function into low-level command to interact with the FPGA device.

Using the Driver

1. Download the driver from AR 70928.
2. Compile the driver.

```
make install
```

TIP: *Run make in the top level of the QDMA driver folder.*

3. Load the kernel driver module.

```
modprobe qdma
```

4. Manage the device.

```
dmactl dev list // list all QDMA function
```

5. Add a queue.

```
dmactl qdma<N> q add mode <mm|st> dir <h2c|c2h>
```

It allocates resources for setting up the queue. Each added queue will appear as a character device on the host, which can be opened to perform DMA transaction.

<*N*> is the QDMA function number obtained from "*./dmactl dev list*".
<*mm|st*> selects either memory mapped (*mm*) or streaming (*st*) mode.
<*mm|st*> selects either memory mapped (*mm*) or streaming (*st*) mode.

6. Start a queue.

```
dmactl qdma<N> q <id> start
```

It configures and sets up the queue on the FPGA. The queue is read to be used since then.

<*N*> is the QDMA function number obtained from "*./dmactl dev list*". <*id*> is the queue index.

7. Start DMA transaction.

```
cd ./tool
./dma_to_device -d <device> -a <address> -s <size> -o <offset> -c
<count> -f <file>
./dma_from_device -d <device> -a <address> -s <size> -o <offset> -c
<count> -f <file>
```

<*device*> is the name of the character device.
<*address*> is the start address on the AXI bus.
<*size*> is the size of a single DMA transfer in bytes.
<*offset*> is the page offset of a transfer.
<*count*> is the number of transfers.
<*file*> is the file name to dump all data transfer, which is optional.

8. Stop a queue.

```
dmactl qdma<N> q <id> stop
```

It removes a queue on the FPGA.

<*N*> is the QDMA function number obtained from "*dmactl dev list*".
<*id*> is the queue index.

9. Delete a queue.

```
dmactl qdma<N> q <id> del
```

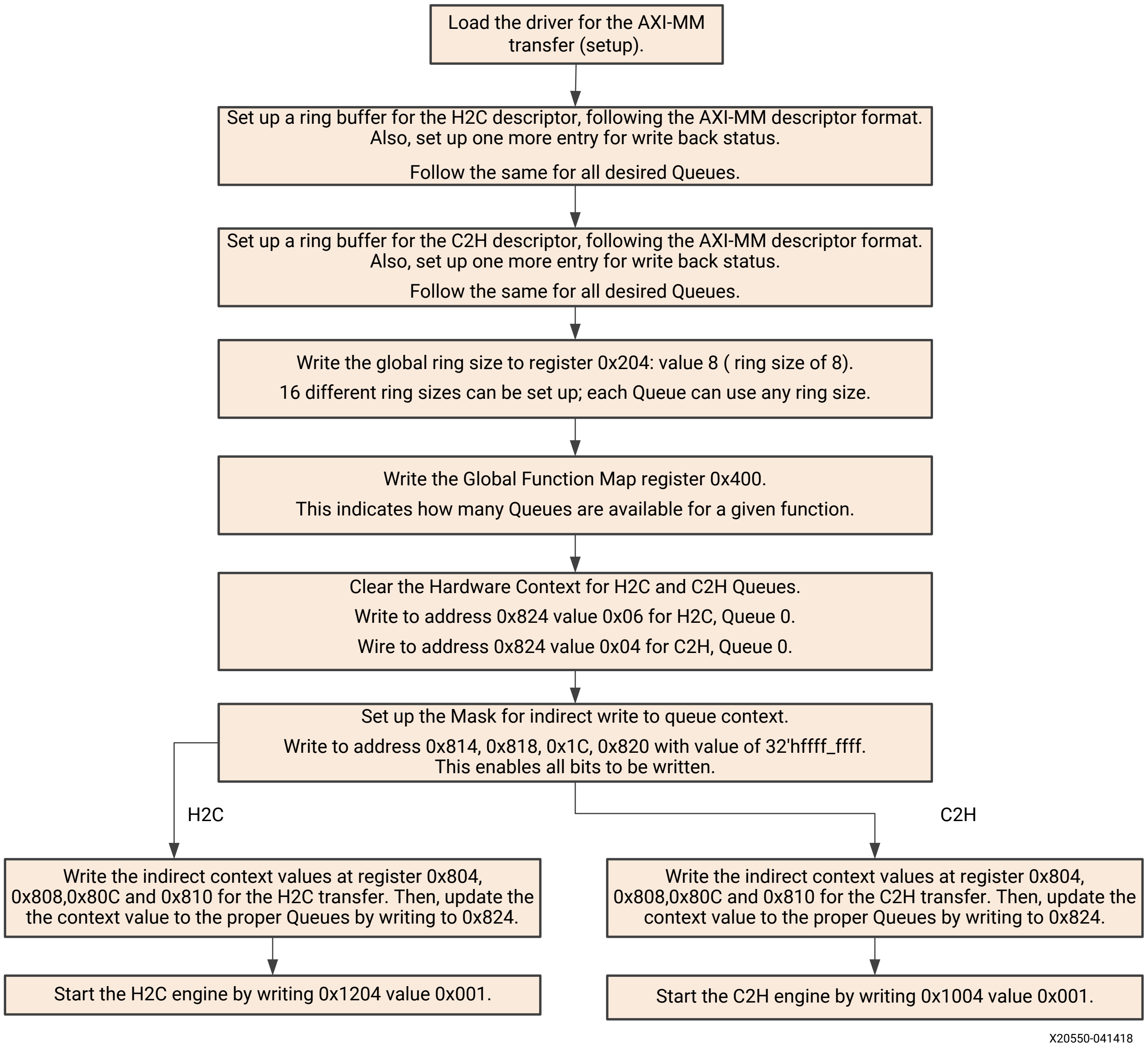
It releases the resources, which is allocated on the host.

<*N*> is the QDMA function number obtained from "*dmactl dev list*".
<*id*> is the queue index.

Reference Software Driver Flow

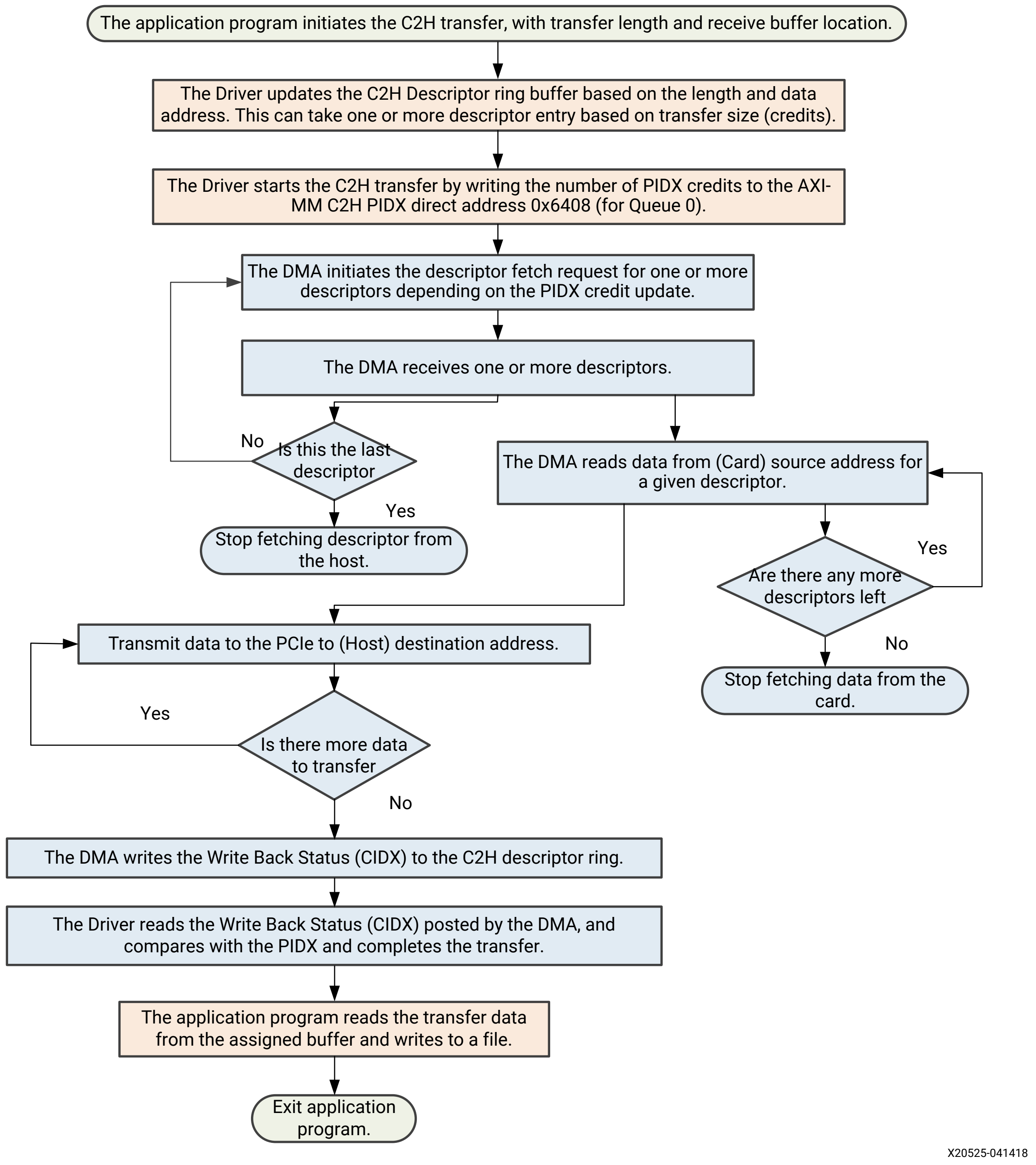
AXI4-Memory Map Flow Chart

Figure 37: **AXI4-Memory Map Flow Chart**


Load the driver for the AXI-MM transfer (setup).
Set up a ring buffer for the H2C descriptor, following the AXI-MM descriptor format. Also, set up one more entry for write back status. Follow the same for all desired Queues.
Set up a ring buffer for the C2H descriptor, following the AXI-MM descriptor format. Also, set up one more entry for write back status. Follow the same for all desired Queues.
Write the global ring size to register 0x204: value 8 (ring size of 8). 16 different ring sizes can be set up; each Queue can use any ring size.
Write the Global Function Map register 0x400. This indicates how many Queues are available for a given function.
Clear the Hardware Context for H2C and C2H Queues. Write to address 0x824 value 0x06 for H2C, Queue 0. Wire to address 0x824 value 0x04 for C2H, Queue 0.
Set up the Mask for indirect write to queue context. Write to address 0x814, 0x818, 0x1C, 0x820 with value of 32'hffff_ffff. This enables all bits to be written.
H2C
C2H
Write the indirect context values at register 0x804, 0x808,0x80C and 0x810 for the H2C transfer. Then, update the the context value to the proper Queues by writing to 0x824.
Write the indirect context values at register 0x804, 0x808,0x80C and 0x810 for the C2H transfer. Then, update the context value to the proper Queues by writing to 0x824.
Start the H2C engine by writing 0x1204 value 0x001.
Start the C2H engine by writing 0x1004 value 0x001.
X20550-041418

AXI4 Memory Mapped C2H Flow

Figure 38: **AXI4 Memory Mapped C2H Flow Diagram**


The application program initiates the C2H transfer, with transfer length and receive buffer location.
The Driver updates the C2H Descriptor ring buffer based on the length and data address. This can take one or more descriptor entry based on transfer size (credits).
The Driver starts the C2H transfer by writing the number of PIDX credits to the AXI-MM C2H PIDX direct address 0x6408 (for Queue 0).
The DMA initiates the descriptor fetch request for one or more descriptors depending on the PIDX credit update.
The DMA receives one or more descriptors.
No
Is this the last descriptor
Yes
Stop fetching descriptor from the host.
The DMA reads data from (Card) source address for a given descriptor.
Yes
Are there any more descriptors left
No
Stop fetching data from the card.
Transmit data to the PCIe to (Host) destination address.
Yes
Is there more data to transfer
No
The DMA writes the Write Back Status (CIDX) to the C2H descriptor ring.
The Driver reads the Write Back Status (CIDX) posted by the DMA, and compares with the PIDX and completes the transfer.
The application program reads the transfer data from the assigned buffer and writes to a file.
Exit application program.
X20525-041418

AXI4 Memory Mapped H2C Flow

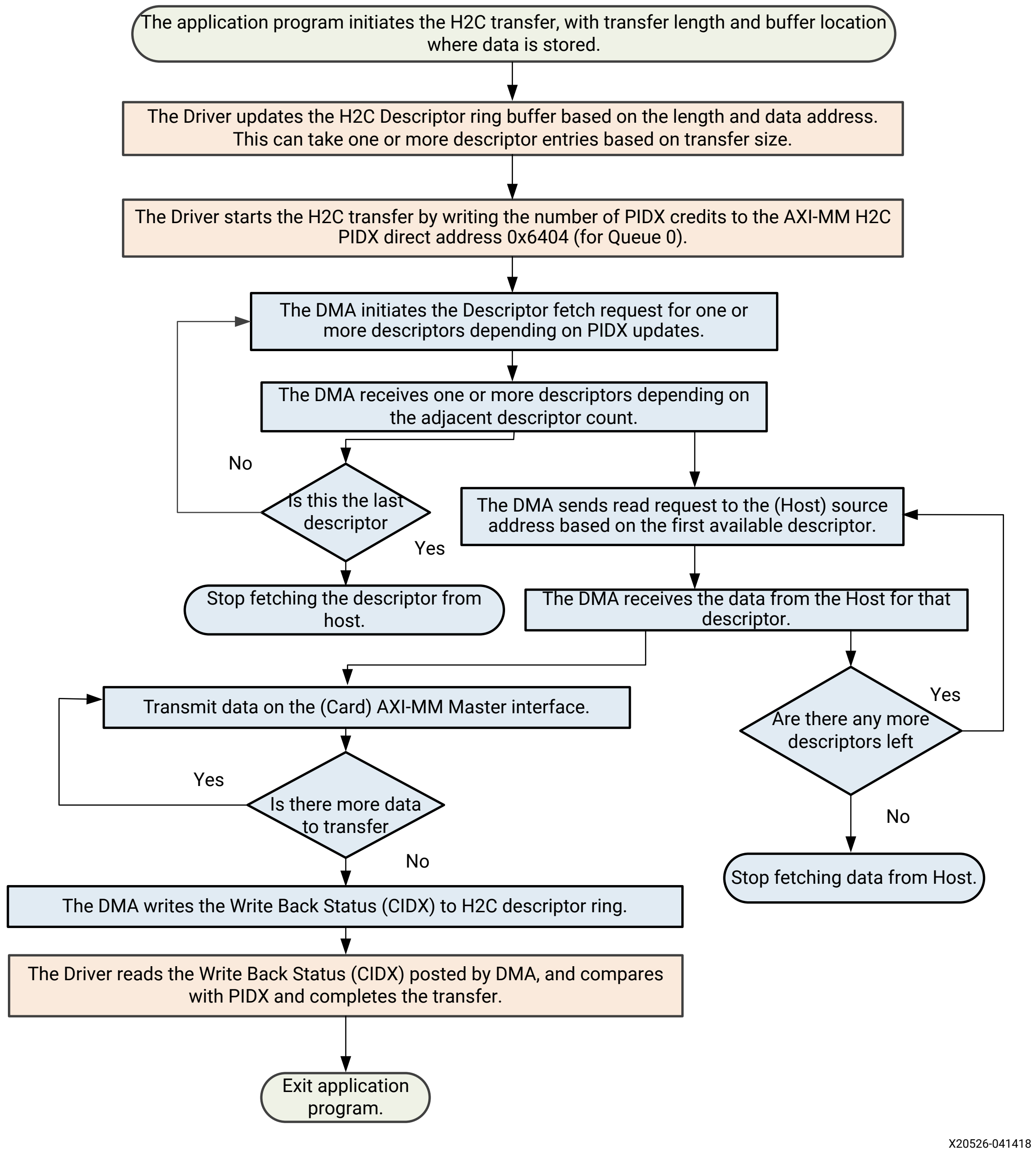
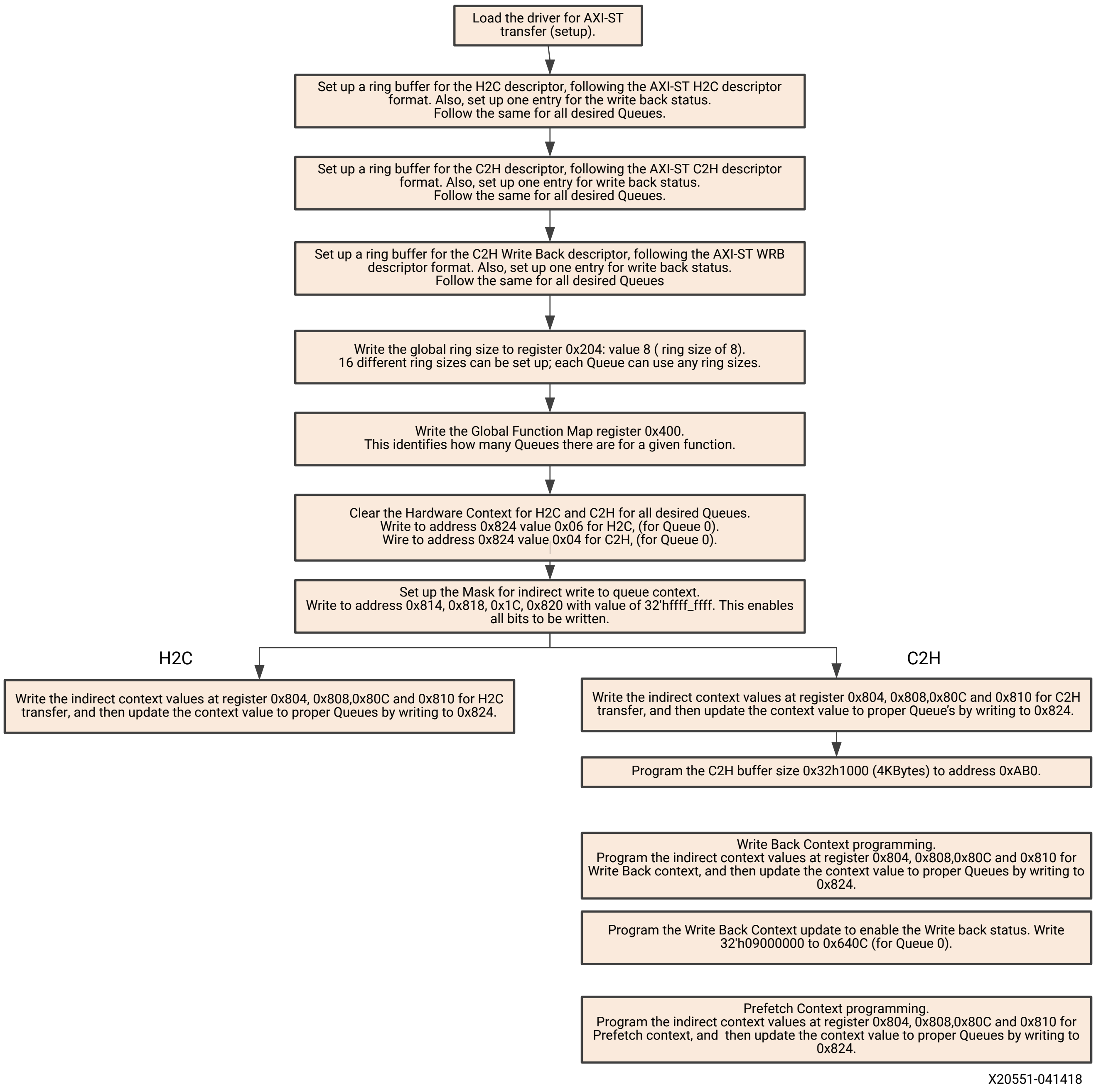

The application program initiates the H2C transfer, with transfer length and buffer location where data is stored.
The Driver updates the H2C Descriptor ring buffer based on the length and data address. This can take one or more descriptor entries based on transfer size.
The Driver starts the H2C transfer by writing the number of PIDX credits to the AXI-MM H2C PIDX direct address 0x6404 (for Queue 0).
The DMA initiates the Descriptor fetch request for one or more descriptors depending on PIDX updates.
The DMA receives one or more descriptors depending on the adjacent descriptor count.
No
Is this the last descriptor
Yes
The DMA sends read request to the (Host) source address based on the first available descriptor.
Stop fetching the descriptor from host.
The DMA receives the data from the Host for that descriptor.
Transmit data on the (Card) AXI-MM Master interface.
Yes
Are there any more descriptors left
Yes
Is there more data to transfer
No
No
Stop fetching data from Host.
The DMA writes the Write Back Status (CIDX) to H2C descriptor ring.
The Driver reads the Write Back Status (CIDX) posted by DMA, and compares with PIDX and completes the transfer.
Exit application program.
X20526-041418

Figure 39: **AXI4 Memory Mapped H2C Flow Diagram**

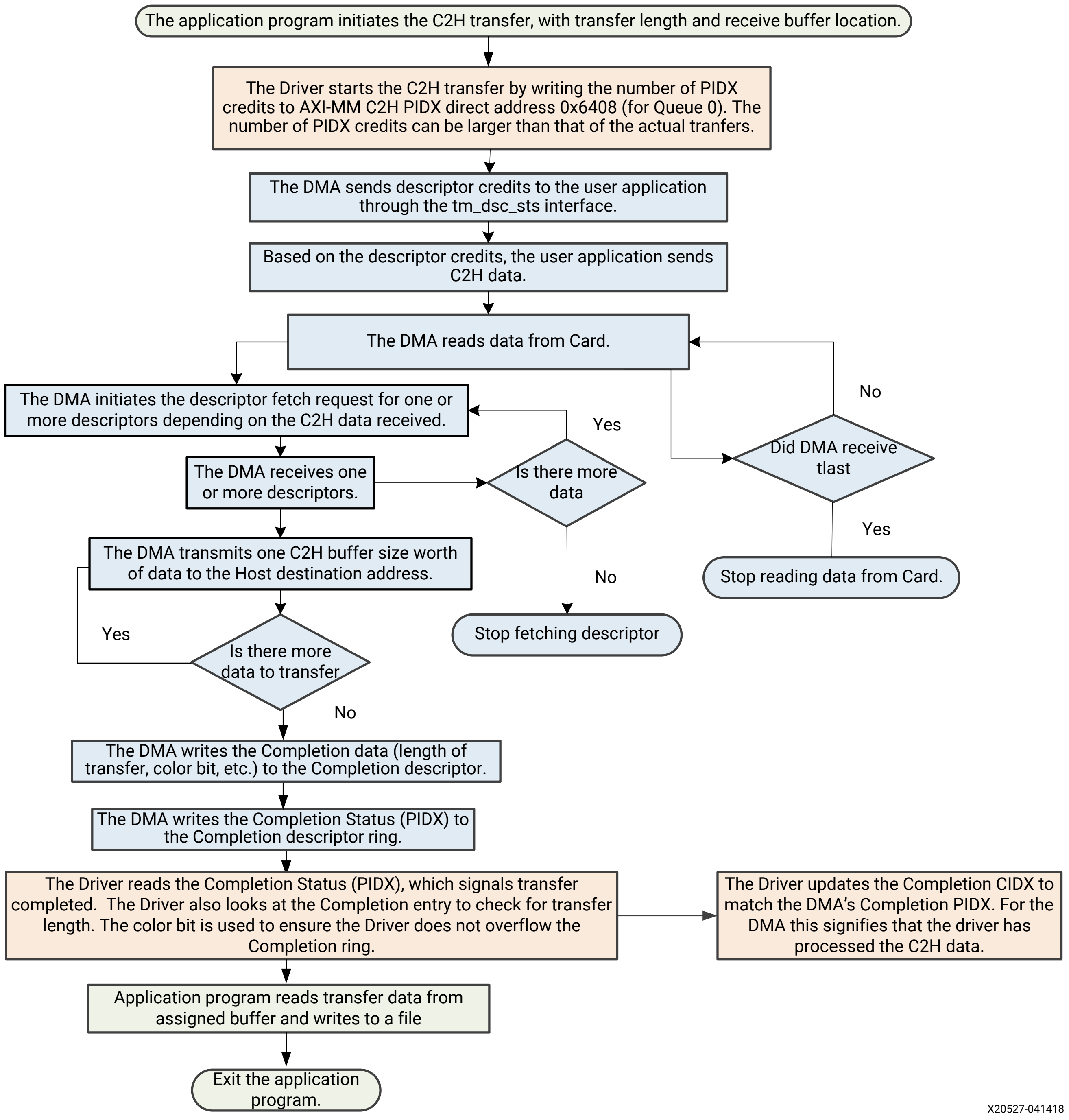
AXI4-Stream Flow Chart

Figure 40: **AXI4-Stream Flow Chart**


Load the driver for AXI-ST transfer (setup).
Set up a ring buffer for the H2C descriptor, following the AXI-ST H2C descriptor format. Also, set up one entry for the write back status. Follow the same for all desired Queues.
Set up a ring buffer for the C2H descriptor, following the AXI-ST C2H descriptor format. Also, set up one entry for write back status. Follow the same for all desired Queues.
Set up a ring buffer for the C2H Write Back descriptor, following the AXI-ST WRB descriptor format. Also, set up one entry for write back status. Follow the same for all desired Queues
Write the global ring size to register 0x204: value 8 (ring size of 8). 16 different ring sizes can be set up; each Queue can use any ring sizes.
Write the Global Function Map register 0x400. This identifies how many Queues there are for a given function.
Clear the Hardware Context for H2C and C2H for all desired Queues. Write to address 0x824 value 0x06 for H2C, (for Queue 0). Wire to address 0x824 value 0x04 for C2H, (for Queue 0).
Set up the Mask for indirect write to queue context. Write to address 0x814, 0x818, 0x1C, 0x820 with value of 32'hffff_ffff. This enables all bits to be written.
H2C
C2H
Write the indirect context values at register 0x804, 0x808,0x80C and 0x810 for H2C transfer, and then update the context value to proper Queues by writing to 0x824.
Write the indirect context values at register 0x804, 0x808,0x80C and 0x810 for C2H transfer, and then update the context value to proper Queue's by writing to 0x824.
Program the C2H buffer size 0x32h1000 (4KBytes) to address 0xAB0.
Write Back Context programming. Program the indirect context values at register 0x804, 0x808,0x80C and 0x810 for Write Back context, and then update the context value to proper Queues by writing to 0x824.
Program the Write Back Context update to enable the Write back status. Write 32'h09000000 to 0x640C (for Queue 0).
Prefetch Context programming. Program the indirect context values at register 0x804, 0x808,0x80C and 0x810 for Prefetch context, and then update the context value to proper Queues by writing to 0x824.
X20551-041418

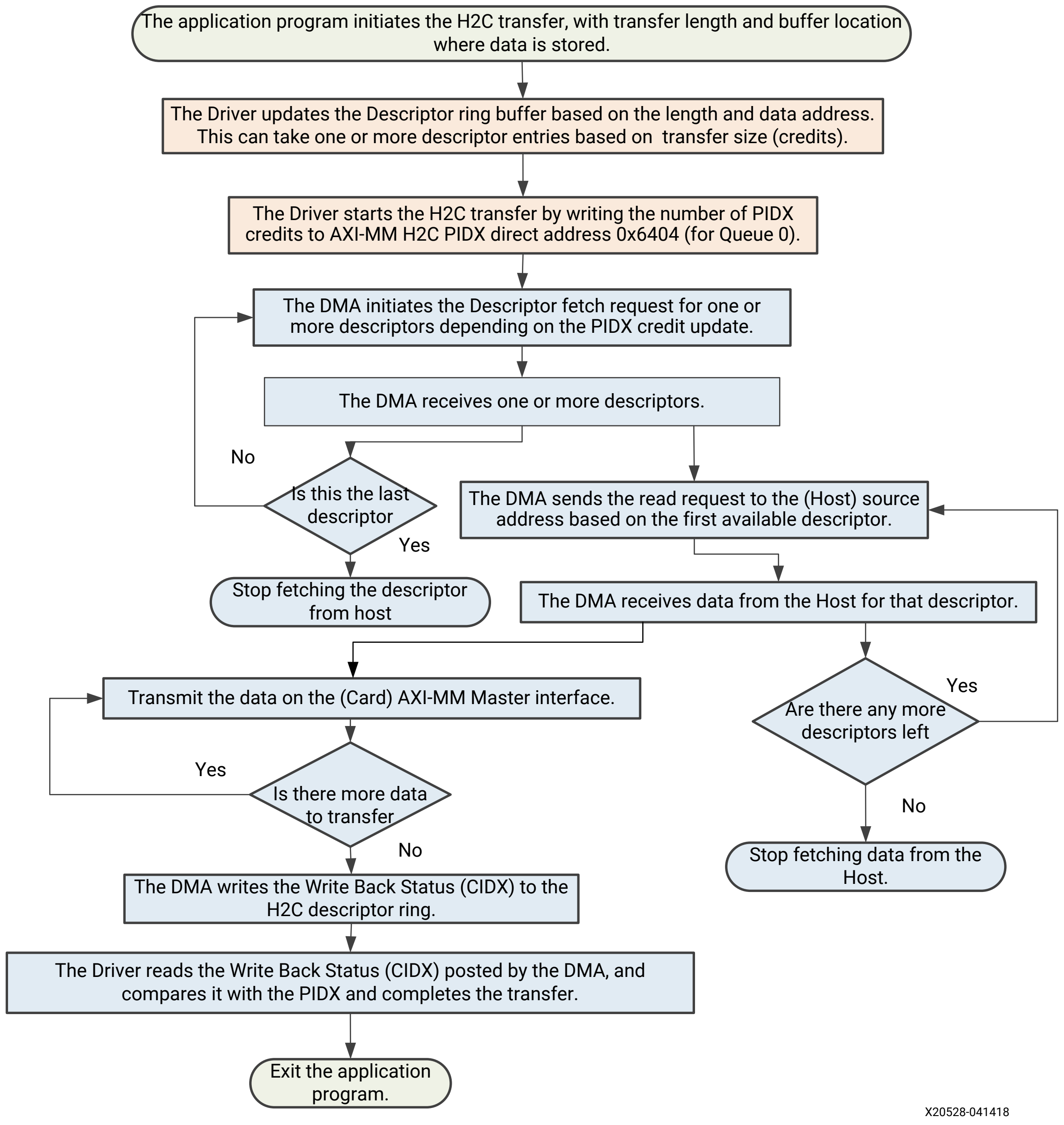
AXI4-Stream C2H Flow

Figure 41: **AXI4-Stream C2H Flow Diagram**


The application program initiates the C2H transfer, with transfer length and receive buffer location.
The Driver starts the C2H transfer by writing the number of PIDX credits to AXI-MM C2H PIDX direct address 0x6408 (for Queue 0). The number of PIDX credits can be larger than that of the actual tranfers.
The DMA sends descriptor credits to the user application through the tm_dsc_sts interface.
Based on the descriptor credits, the user application sends C2H data.
The DMA reads data from Card.
The DMA initiates the descriptor fetch request for one or more descriptors depending on the C2H data received.
The DMA receives one or more descriptors.
Is there more data
Yes
No
Stop fetching descriptor
Did DMA receive tlast
No
Yes
Stop reading data from Card.
The DMA transmits one C2H buffer size worth of data to the Host destination address.
Is there more data to transfer
Yes
No
The DMA writes the Completion data (length of transfer, color bit, etc.) to the Completion descriptor.
The DMA writes the Completion Status (PIDX) to the Completion descriptor ring.
The Driver reads the Completion Status (PIDX), which signals transfer completed. The Driver also looks at the Completion entry to check for transfer length. The color bit is used to ensure the Driver does not overflow the Completion ring.
The Driver updates the Completion CIDX to match the DMA's Completion PIDX. For the DMA this signifies that the driver has processed the C2H data.
Application program reads transfer data from assigned buffer and writes to a file
Exit the application program.
X20527-041418

AXI4-Stream H2C Flow

Figure 42: **AXI4-Stream H2C Flow Diagram**


The application program initiates the H2C transfer, with transfer length and buffer location where data is stored.
The Driver updates the Descriptor ring buffer based on the length and data address. This can take one or more descriptor entries based on transfer size (credits).
The Driver starts the H2C transfer by writing the number of PIDX credits to AXI-MM H2C PIDX direct address 0x6404 (for Queue 0).
The DMA initiates the Descriptor fetch request for one or more descriptors depending on the PIDX credit update.
The DMA receives one or more descriptors.
No
Is this the last descriptor
Yes
Stop fetching the descriptor from host
The DMA sends the read request to the (Host) source address based on the first available descriptor.
The DMA receives data from the Host for that descriptor.
Transmit the data on the (Card) AXI-MM Master interface.
Yes
Are there any more descriptors left
No
Stop fetching data from the Host.
Yes
Is there more data to transfer
No
The DMA writes the Write Back Status (CIDX) to the H2C descriptor ring.
The Driver reads the Write Back Status (CIDX) posted by the DMA, and compares it with the PIDX and completes the transfer.
Exit the application program.
X20528-041418

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open the Xilinx Documentation Navigator (DocNav):

- From the Vivado® IDE, select **Help** → **Documentation and Tutorials**.
- On Windows, select **Start** → **All Programs** → **Xilinx Design Tools** → **DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In the Xilinx Documentation Navigator, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

Note: For more information on Documentation Navigator, see the Documentation Navigator page on the Xilinx website.

References

These documents provide supplemental material useful with this product guide:

1. *AMBA AXI4-Stream Protocol Specification* (ARM IHI 0051A)
2. *PCI-SIG Specifications* (www.pcisig.com/specifications)
3. *Virtex-7 FPGA Integrated Block for PCI Express LogiCORE IP Product Guide* (PG023)
4. *7 Series FPGAs Integrated Block for PCI Express LogiCORE IP Product Guide* (PG054)
5. *UltraScale Devices Gen3 Integrated Block for PCI Express LogiCORE IP Product Guide* (PG156)
6. *AXI Bridge for PCI Express Gen3 Subsystem Product Guide* (PG194)
7. *DMA/Bridge Subsystem for PCI Express Product Guide* (PG195)
8. *UltraScale+ Devices Integrated Block for PCI Express LogiCORE IP Product Guide* (PG213)
9. *Vivado Design Suite: AXI Reference Guide* (UG1037)
10. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
11. *Vivado Design Suite User Guide: Designing with IP* (UG896)
12. *Vivado Design Suite User Guide: Getting Started* (UG910)
13. *Vivado Design Suite User Guide: Logic Simulation* (UG900)
14. *Vivado Design Suite User Guide: Using Constraints* (UG903)
15. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
09/04/2018 v2.0	
Port Descriptions	For tm_dsc_sts_rdy (VDM Ports) and st_rx_msg_rdy (QDMA Traffic Manager Credit Output Ports), emphasized that when this interface is not used, Ready must be tied-off to 1.
Register Space	Added a register to stall read requests from H2C Stream Engine if the amount of outstanding data exceeds a programmed threshold.
	Added a new C2H Completion interrupt trigger mode that includes user trigger, timer expiration, or count exceeding the threshold

Section	Revision Summary
06/22/2018 v2.0	
Overview chapter	Updated content throughout.
Port Descriptions section	Changed some table content, and some reorganization of the content.
Register Space section	Added Memory Map Register Space and AXI4-Lite Slave Register Space section.
Context Structure Definition section, and Queue Entry Structure section	Removed these sections, and moved content into the QDMA Operations section in the Overview chapter.
Design Flow Steps chapter	Updated descriptions for Basic Tab, Capabilities Tab, PCIe BARs Tab, PCIe Misc Tab, and PCIe DMA Tab.
Example Design chapter	Added two new example designs, and added example design registers.
04/17/2018 v1.0	
Initial Xilinx release.	

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright

© Copyright 2018 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.