QDMA Subsystem for PCI Express v1.0

Product Guide

Vivado Design Suite

PG302 (v1.0) April 17, 2018

XILINX
ALL PROGRAMMABLE™

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

- Supports PCIe Integrated Blocks 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.
- Support for both the AXI4 Memory Mapped and AXI4-Stream interfaces per queue.
- 2K queue sets
  - 2K H2C Descriptor rings.
  - 2K C2H Descriptor rings.
  - 2K C2H Write back rings.
- Supports Polling Mode (Status Descriptor Write Back).
- Interrupts
  - 2K MSI-X vectors.
  - Up to 8 MSI-X per function.
  - Interrupt coalescing.
- C2H Stream interrupt moderation.
- C2H AXI4-Stream Completion (CMPT) interrupt entry coalescence.
- Descriptor and DMA customization through user logic

    - Allows custom Descriptor format.
    - Traffic Management.
- Supports SR-IOV up to 4 Physical Functions (PF) and 252 Virtual Functions (VF)
    - Thin Hypervisor model.
    - Allows only privileged/Physical functions to program contexts and registers.
    - Function Level Reset (FLR) support.
    - Mailbox.

The 2018.1 version of this IP is marked as BETA. Not all features listed above are supported in the current release. For a list of features that are not supported, see Feature Support Roadmap. Some ports and the Descriptor Format will change in 2018.2. For more information, contact Xilinx Support.

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, Windows, DPDK Drivers
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. Standalone driver details can be found in AR 70928.
3. For the supported versions of the tools, see the Xilinx Design Tools: Release Notes Guide.

Overview

The Queue DMA (QDMA) subsystem provides PCI Express (PCIe) based high bandwidth and high packet rate by using the UltraScale+ Integrated Block for PCI Express in conjunction with an extensive DMA and bridge infrastructure. The primary mechanism for data transfer is bidirectional Endpoint ("Card") initiated transactions from Host to Card (H2C) and Card to Host (C2H). QDMA provides a secondary option of Host initiated memory accesses to AXI space on the card (Target Bridge – Master), and user logic initiated direct access to the Host (Target Bridge – Slave), including PCIe and AXI4 memory mapping. The primary user interfaces to the DMA engines are: AXI4 Memory Mapped (AXI-MM) and AXI4-Stream (AXI-ST); AXI4 Memory Mapped and AXI4-Lite for direct host or user logic initiated access; and several other functional interfaces. The user logic and software interact through the PCIe Base Address Registers (BAR) interfaces and AXI4-Lite Slave (AXI-L) register interfaces to set up and control the subsystem.

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 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.

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).

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.

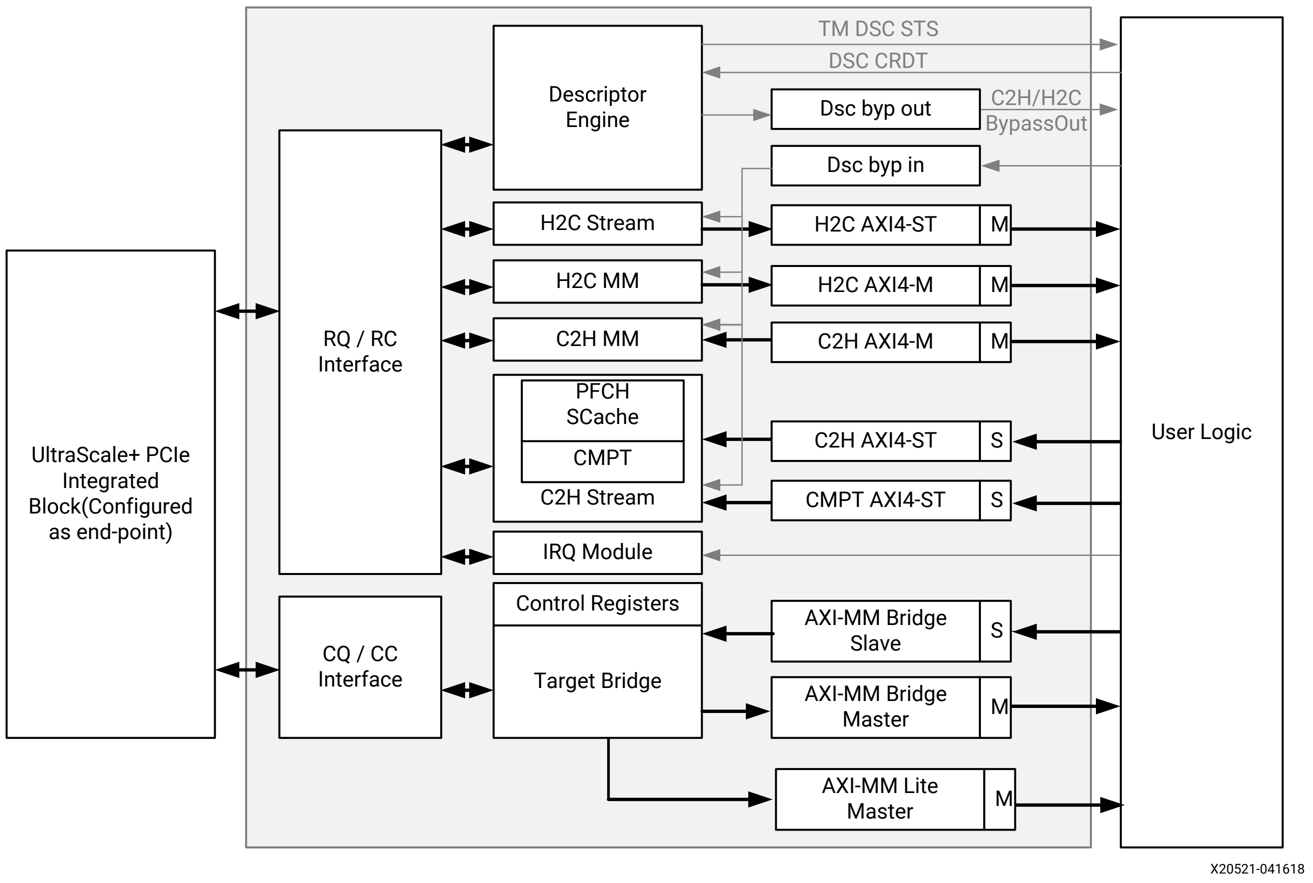
Table 1: **Glossary of Terms**

Acronym	Full Name
AXI-ST	AXI4-Stream
H2C	Host to Card
C2H	Card to Host
TXQ	Transfer Queue
RXQ	Receive Queue
TM	Traffic Manager
CMPT	C2H AXI4-Stream Completion
FLR	Function Level Reset
CQ	Completer Request
CC	Completer Completion
RQ	Requester Request
RC	Requester Completion
SRIOV	Single root input/output virtualization
PF	Physical function
VF	Virtual function
PIDX	Producer index pointer
CIDX	Consumer index pointer
CTXT	Context
HW	Hardware
SW	Software
QID	Queue Identification
CSR	Control/Status register
PFCH	Prefetch Block

QDMA Architecture

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

Figure 1: **QDMA Core**



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 theIntegrated Block for PCIe IP to determine where the request should be forwarded. The three 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 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 Integrated Block for PCIe® to prevent head of line blocking of posted requests. If Finite Completion Credits are not supported in the system or not configured in the Integrated Block for PCIe®PCIe IP, the DMA will meter non-posted transactions based on the PCIe Receive FIFO space. It is possible that non-posted transactions can be throttled by the number of outstanding PCIe tags.

Interrupt Module

IRQ module aggregates interrupts from various sources into the UltraScale+™ Integrated Block for PCIe® core interface. The interrupt sources are queue-based interrupts, user interrupts and error interrupts.

Queue-based interrupts and User interrupts are allowed on the PFs and VF, but Error interrupts are allowed only for PFs. If the SRIOV is not enabled, each PF has the choice of MSI-X or MSI. If the SRIOV is enabled, only MSI-X is supported on all functions.

With support for MSI-X, MSI 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 the MSI.

The UltraScale+™ Integrated Block for PCIe core offers eight interrupts per functions. The QDMA offers a novel way of aggregating interrupts from multiple queues on a given PCIe function to a single interrupt vector. Theoretically, all 2K queues can be mapped to single vectors. The QDMA offers 256 interrupt aggregation rings that can be flexibly allocated among 256 functions.

Descriptor Engine

H2C and C2H descriptors are fetched by the Descriptor Engine. The descriptor engine maintains per queue context where it tracks the software PIDX, CIDX, BADDR, queue configurations, etc. It uses a round-robin algorithm for fetching the descriptors. The descriptor engine has separate buffers for H2C and C2H, and ensures it never fetches available space. In addition, it has only one descriptor fetch outstanding per queue. It also reorders the out-of-order completions so that descriptors for the queues 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 the H2C or C2H engine. In internal mode, based on context setting, 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 completion of DMA operations. With the exception of C2H Stream, all modes use this mechanism to convey completion of DMA operations, which allows the software to reclaim the descriptors and free up any associated buffers. This is indicated by the CIDX field of the status descriptor.

RECOMMENDED: *If the queue is associated with interrupt aggregation, Xilinx recommends that you turn off this status descriptor, and instead get the DMA status from the interrupt aggregation ring.*

To put a limit on the number of fetched descriptors, turn on crediting on a per queue basis. In this mode, the descriptor engine fetches the descriptors for available credit, and the total descriptors fetched per queue is limited to the credit provided. The user logic can return the credit through the `dsc_crdt` interface.

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. This prioritizes and optimizes the descriptor storage, and implements a DMA descriptor pull mode.

H2C MM Engine

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

For Internal mode, the queue descriptor engine delivers memory mapped descriptors straight to H2C MM engine. The user logic can also inject the descriptor to H2C bypass interface to move data from host to card memory. This gives the ability to do interesting things such as mixing the 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 C2H AXI-M interface. The engine generates reads on the C2H AXI-M, splitting descriptors into multiple requests based on 4K boundaries. Once completion data for a read request is received, it generates a write on the Integrated Block for PCIe® interface. For source and destination addresses that are not aligned, the hardware will shift the data and split writes on the PCIe to obey MPS and prevent 4K alignment crossing. Each completed descriptor is checked to determine whether a writeback and/or interrupt is required.

For Internal mode, the queue descriptor engine delivers memory mapped descriptors straight to C2H MM engine. The user logic can also inject the descriptor to C2H bypass in interface to move data from host to card memory. This gives the ability to do interesting things such as mixing the control and DMA commands in the same queue. Control information can be sent to a control processor indicating the completion of DMA operation.

The PCIe Function number information will be provided by the AXI-MM `aruser` interface bus. A parity bus separate from the data and user bus is also provided for end-to-end parity support.

H2C Stream Engine

H2C engine moves data from host to H2C Stream interface. For internal mode, queue descriptors are delivered straight to the H2C engine. For a queue in bypass, the descriptors can be reformatted and fed to bypass-in interface. The engine is responsible for breaking up DMA reads to MPS size, guarantee the space for completions, and also makes sure completions are reordered to make for correct H2C Stream interface ordering.

It has the buffering for up to 256 DMA reads and up to 32 Kbytes of data. There is an aligner to zero align the PCIe completion data to the AXI-ST interface. This allows every descriptor to be random offset and random length. Total length of all descriptors should be less than 64 KB.

For internal mode queues, each descriptor defines a single packet to be transferred to the H2C AXI-ST interface. A packet straddling multiple descriptors is not allowed, due to the lack of per queue storage. Multi-descriptor packets can be implemented using the descriptor bypass mode, where descriptors are delivered to user logic, to be stored on a per queue basis. When it has enough descriptors to form the packet, the H2C DMA can be initiated by delivering those descriptors, non-interleaved, with other H2C ST packet descriptors, through bypass in interface. Also, in bypass in the interface, the user logic can control the generation of the status descriptor.

C2H Stream Engine

THe C2H streaming engine is responsible for DMA writing the streaming packet from the user logic to the descriptors in C2H descriptor queue specified by the QID associated with the packet. It allows user logic to send up to 28B of metadata along with packet, which will be placed into C2H AXI-Stream Completion (CMPT) queue entry.

C2H has two major blocks to accomplish C2H streaming DMA, Prefetch Block (PFCH) cache and CMPT. The PFCH and CMPT will have per queue context to performance of its function, which the software is expected to program.

PFCH cache has three main modes, on a per queue basis, called cache mode, simple bypass and cached bypass mode.

- In **simple bypass mode**, a queue fetched descriptor is sent to user logic. User logic is then responsible for delivering the packet and associated descriptor in bypass interface. The ordering of the descriptors in bypass interface and C2H stream interface be maintained across queue among simple bypass mode queues.
- In **cache mode** and **cached bypass mode**, the PFCH module offers storage for 512 descriptors, which can be used by up to 64 different queues. In this mode, it controls the descriptors to be fetched by managing the C2H descriptor queue credit on demand based on received packets in the pipeline. One could turn on the pre-fetch mode per queue basis and that causes the descriptors to be opportunistically pre-fetched so that descriptors are available before the packet shows up and reduce the latency. The size of the buffer is fixed for a queue (PFCH context) and it can scatter the packet up to 7 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 does not have the same restriction as simple bypass mode.

After the DMA write of the packet from the C2H stream interface is done, the CMPT packet from the CMPT interface is placed into the CMPT queue ID from the C2H stream interface. Queue state and configuration is stored in CMPT context per queue basis. It stores the base address, CIDX, PIDX, configurations in context. The software identifies the new CMPT entry being written based on the color bit or the based status descriptor. It also can be configured to generate the interrupt, status descriptor, 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.

CMPT has cache of 32 entries to coalesce the multiple smaller CMPT writes into 64B writes to improve the PCIe efficiency. At any time, it can simultaneously coalesce for 32 queues and any additional queue, needing to write CMPT entry, will cause the eviction of least recently used queue.

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 Integrated Block for PCIe®. The interface supports up to 32 outstanding reads and 32 outstanding writes. One or more PCIe BAR of any PF or VF can be mapped to master AXI-MM interface. This selection needs to be done at the point of configuring the IP. Function ID, bar ID (bar hit), VF group and VF group offset will be made available as part of `aruser` and `awuser` of the AXI-MM interface to help the user logic identify the source of memory access. Also, each host initiated access can be uniquely mapped to 64 bit AXI address space through PCIe to AXI BAR translation.

Bridge Master AXI4-Lite Interface

One or more PCIe BAR of any PF or VF can be mapped to the master AXI4-Lite interface. This selection needs to be done at the point of configuring the IP. 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. Also, each of the host initiated access can be uniquely mapped to 64 bit AXI address space. Though the user bits are non-standard, without `user` bits the user logic may not be able to identify the intended address space. One outstanding read and one outstanding write are supported on this interface.

PCIe to AXI BARs

For requests received in PCIe, a set of six 32-bit BARs for each Physical Function, one 32-bit BAR for EXPROM BAR on physical function, and six 32-bit BARs for Virtual Functions within the same Physical Function when SR-IOV is enabled are available. These BARs provide address translation to the AXI4 memory mapped spaced capability, interface routing, and AXI4 request attribute configuration. 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 AXI-MM Master interface.

The configurable AXI request attributes include:

- Address aperture to be translated: `attr_dma_pciebar2axibar_[0-5]_len`
- Translated address: `attr_dma_pciebar2axibar_[0-5]`

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.
- Allocate and Other Allocate are not supported.

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 Integrated Block for PCIe®. AXI to PCIe translation is supported through the AXI to PCIE BARs. The interface will split requests as necessary to obey PCIe MPS and 4K crossing requirements. Up to 32 outstanding read requests and up to 32 outstanding 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. The QDMA registers are virtualized for VFs and PFs. Example VFs and PFs can access different parts of the address space, and each has access to its own queues. To accommodate all modes, this interface provides a non-standard AXI4-Lite slave interface where the user logic can provide function ID, which gives the QDMA proper internal register access. One outstanding read request and one outstanding write request are supported.

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 attributes. The attributes include:

- Base address: `attr_dma_axibar_base_[0-5]`
- High address: `attr_dma_axibar_highaddr_[0-5]`
- Address size (32-bit or 64-bit): `attr_dma_axibar_as_[0-5]`
- Translated address: `attr_dma_axibar_[0-5]`

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

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 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 (slow path) is deferred to the physical function driver. The driver that does not have sufficient privileges needs to communicate with the privileged driver through mailbox 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 context and registers.
- Drivers are allowed to do pointer updates only for the queue allocated to them.
- IOMMU can be turned on to check that the DMA is done by PFs and VFs. The ARID comes from queue context programmed by 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 will be 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 incoming mailbox message outstanding.

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

QDMA Operations

Theory of Rings

Multi-queue DMA uses RDMA model queue pairs to allow RNIC implementation in the user logic. Each queue set consists of Host to Card (H2C) Ring, Card to Host (C2H) Ring, and a C2H Stream Completion (CMPT) Ring.

H2C and C2H rings are always written by the driver/software. The hardware is always read from the Queue set (QSet0 - QSet2047). H2C carries the descriptors for the DMA read operations. C2H carries the descriptors for the DMA write operations.

In all internal modes, H2C descriptors carry address and length, which is also called scatter gather descriptor. It supports 32 bits of metadata that can be passed from software to hardware along with every packet. The descriptor can be of memory mapped (carries host address and card address) or streaming (only host address). Through descriptor bypass, you can define the complex descriptor format where software is permitted the concept of immediate data and more metadata along with packet.

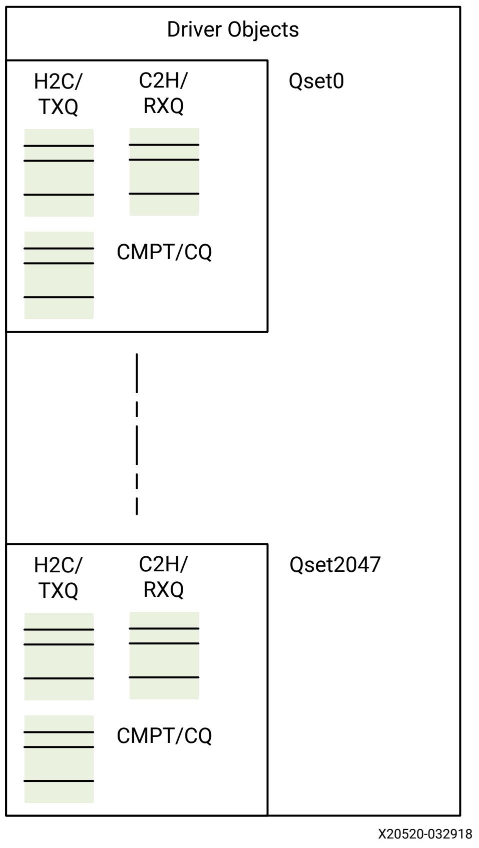
Memory Mapped descriptors for C2H queue consist of card address, host address and the length of transfer. In streaming mode except for simple bypass mode, descriptors carry only the host address and the buffer size of the descriptor passed by the driver, which is expected to be of fixed size for the whole queue.

The software advertises valid descriptors to H2C queue and C2H queue by writing its producer index (PIDX) to the hardware. H2C and C2H descriptors can be reclaimed upon completion of DMA operation through the status descriptor. The status descriptor is the last entry of the descriptor ring in a queue set. 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 C2H stream mode, C2H descriptors will be reclaimed based on the queue entry. Typically it carries one entry per C2H packet, indicating of one or more C2H descriptors consumed. The CMPT queue entry carries enough information for software to claim all the descriptors consumed. Through external logic it can be extended to carry other kinds of completions or information to host.

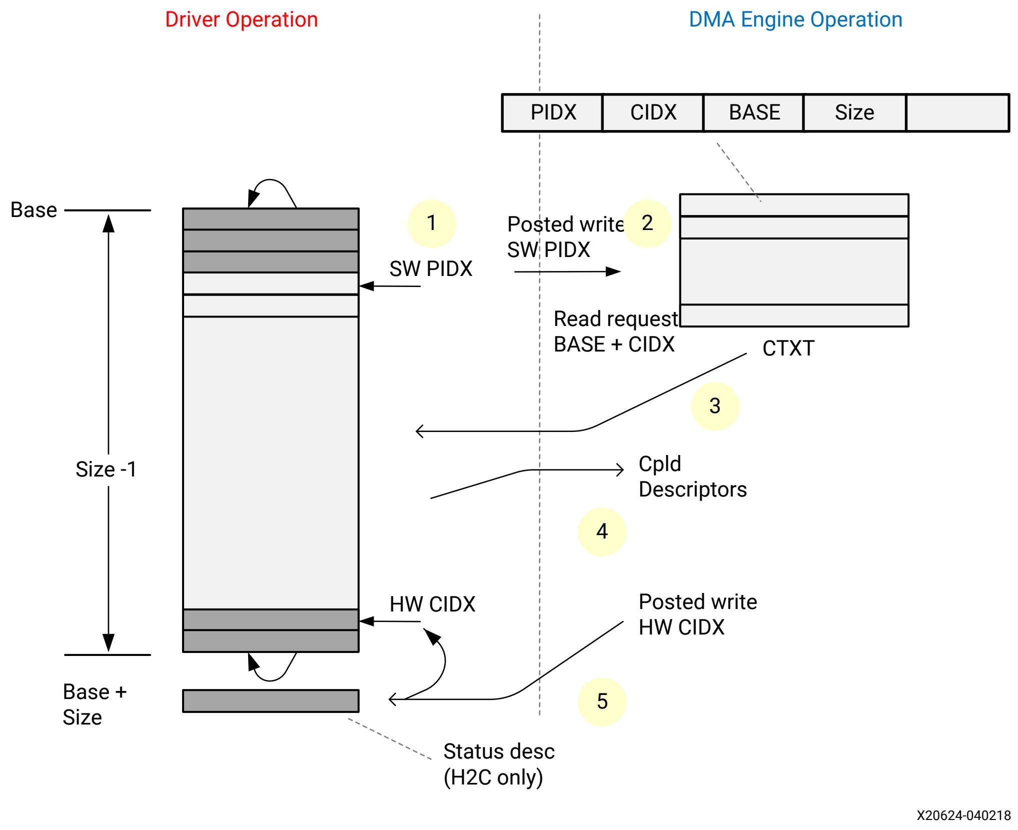
CMPT entry written to the ring can be detected by the driver using color bit in the descriptor or the status descriptor at the end of the CMPT queue ring. This CMPT descriptor can carry meta data for C2H stream packet and also it can be custom completion or immediate for user application. The CMPT queue can be detected by the driver using a color bit in the descriptor or last descriptor of the CMPT queue reserved as status descriptor. The CMPT queue supports two formats, internal or user. In internal format, it conveys metadata, color bit, descriptor format, error and packet length. In the user format, the length is the responsibility of the user logic.

Figure 2: **Theory of Rings**



H2C and C2H Circular Buffer Queues

Figure 3: **H2C and C2H Circular Buffer Queues**



The above figures shows the H2C and C2H fetch operation.

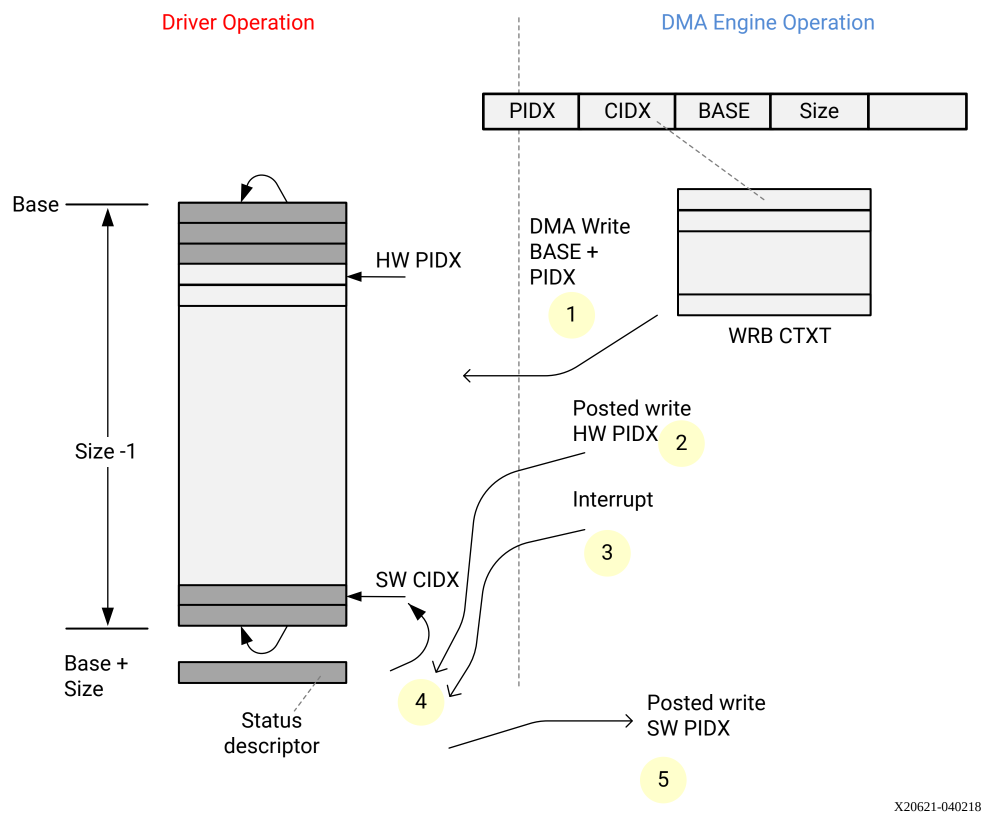
- For H2C, the application data gets written to a buffer, the software forms the descriptor and posts it to the copy of the Producer Index (PIDX) location in the descriptor ring. (For C2H, the driver forms the descriptor with the buffer for the hardware to the DMA packet.)
- The software sends the posted write for the associated Queue ID (QID) with its current PIDX value.
- Upon reception of PIDX, updates the hardware, which issues DMA read to address BASE +CIDX.
- The read completion from the host memory is delivered to the H2C Engine or C2H Engine. In case of bypass, it will be sent out.
- Whenever the descriptor processing is complete, only for the H2C Status Descriptor with CIDX will be written to allow the driver to reuse the descriptors and deallocate buffers.

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.

C2H DMA Write Back

Figure 4: **C2H DMA Write Back**



When C2H receives a packet from the user logic, it gets the DMA buffer from the Fetch Engine and the DMA writes the payload to one or more buffers. After that, write back operation begins.

Simple flow of DMA Write back queue operation.

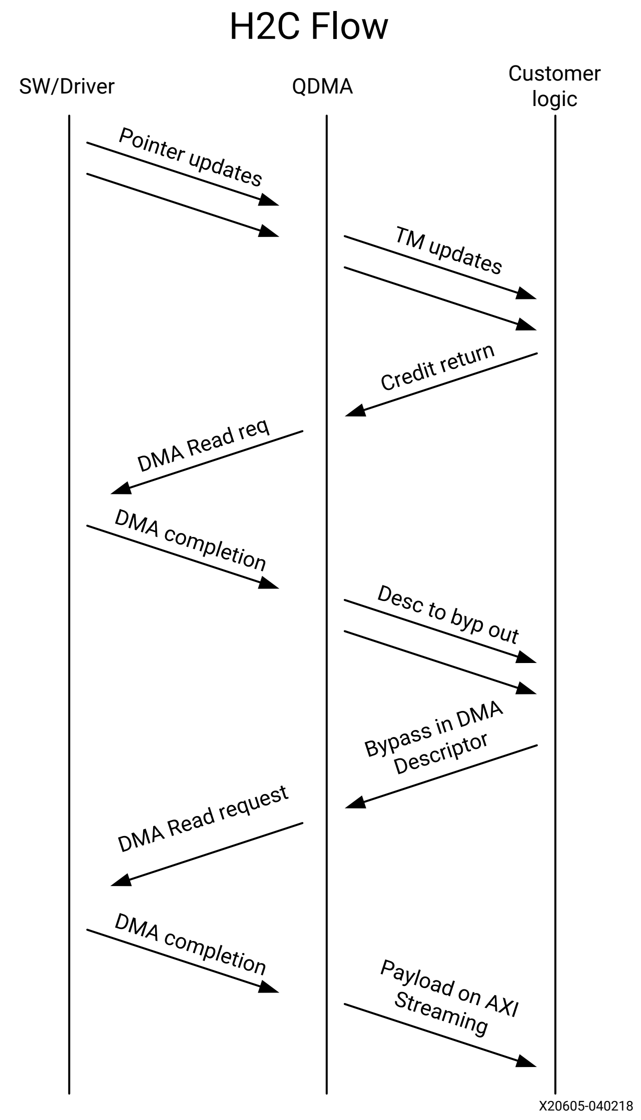
- The DMA write completion descriptor will be written to address BASE+PIDX.
- Posted write to status descriptor with PIDX.
- If in interrupt mode, generate the interrupt.
- The software identifies the new descriptor being written, updates the SW CIDX, and reads any buffers associated with the descriptor.
- Sends posted write back to queue DMA with SW CIDX. This allows the hardware to reuse the descriptors again.

Descriptor Bypass

The Descriptor bypass mode provides immediate data with efficient traffic management. It also provides address translation of descriptors. The C2H Descriptor bypass mode allows for custom write back format. The Descriptor bypass mode can be used with RoCE/iWARP Send queues. In Descriptor bypass mode, descriptors are pushed in from the soft logic and stored in-order in the descriptor buffer based on the channel to which the descriptor belongs. The descriptor bypass can be enabled on a per channel basis. The descriptor bypass is controllable through registers.

H2C Descriptor Bypass

Figure 5: **H2C Descriptor Bypass Flow**



When Descriptor Bypass for Read (H2C) is enabled, these descriptor bypass ports are present. For port descriptions, seeQDMA Descriptor Bypass Input Ports and QDMA Descriptor Bypass Output Ports.

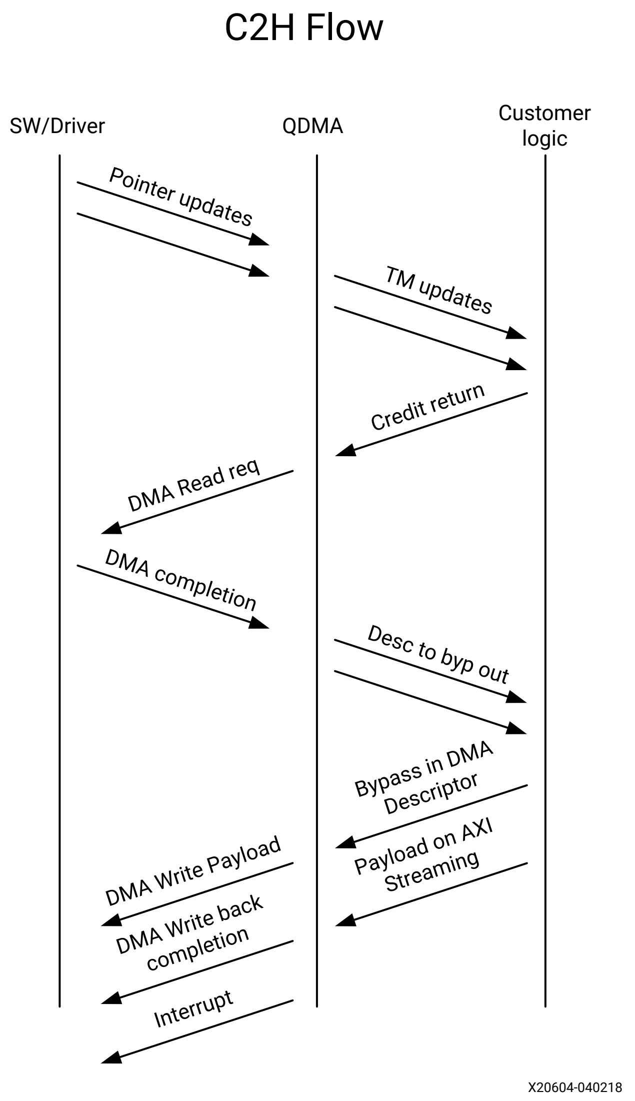
C2H Descriptor Bypass

The C2H Bypass mode supports two bypass input modes:

- **Simple mode**: Customer logic takes full responsibility of returning descriptor in sync with incoming packet.
- **Cache mode**: This mode makes use of cache in QDMA where up to 512 C2H descriptors can be cached for 64 queues.

The C2H descriptor bypass flow is as shown below.

Figure 6: **C2H Descriptor Bypass Flow**



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

C2H Stream

C2H Descriptor

The C2H descriptors can be from the Fetch Engine or C2H Bypass Input interfaces. The descriptors from the Fetch Engine are in cache mode. The PFCH block keeps the order of the descriptors. 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. For the cache mode, the PFCH block keeps the order of the descriptors.

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 PFCH block sends the credits for the descriptors.

The Descriptor Context has a `desc_byp` bit. When it is 1'b1, the Fetch Engine sends out the descriptors on the C2H Bypass Output interface. The user application convert it and loops it back to the QDMA Subsystem for PCIe on the C2H Bypass Input interface. When it is 1'b0, the Fetch Engine sends the descriptors to the PFCH block directly.

Three cases per queue basis are supported.

	c2h_byp_in	desc_ctxt.desc_byp	pfch_ctxt.bypass
Case 1	simple mode	1	1
Case 2	cache mode	1	0
Case 3	cache mode	0	0

For Case 1, the 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 Case 2, the 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 PFCH block sends the credits for the descriptors, and it keeps the order of the descriptors.

For Case 3, the Fetch Engine sends the descriptors to the PFCH block. The PFCH block 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.

C2H DMA Write Engine

The C2H DMA Write Engine block gets the C2H streaming data packet from the user application. It breaks the data packet into smaller TLP packets and sends them to the integrated block for PCIe.

C2H Completion

When the user application sends the C2H data packet to the DMA, it also sends the CMPT packet. 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 when the data format bit is 1'b0.

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 when the data format bit is 1'b1.

Name	Size	Index
User defined	61 bits-253 bits	[255:3]
rsvd	[2:1]	[2:1]
Data format	[0:0]	[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 128bits.

When the DMA write of the data packet is done, the QDMA writes the CMPT packet into the CMPT queue. Besides the user defined data, it also includes some other information, such as error, color, and the length onto the CMPT packet.

The following is the CMPT packet inside the CMPT queue in the User Format when the data format bit is 1'b1.

Name	Size	Index
User defined	61 bits-253 bits	[255:3]
err	1	[2:2]
color	1	[1:1]

Name	Size	Index
Data format	1	[0:0]

The following is the CMPT packet inside the CMPT queue in the Standard Format when the data format bit is 1’b0.

Name	Size	Index
User defined	44 bits-236 bits	[255:20]
Len	16	[19:4]
rsvd	1	[3:3]
err	1	[2:2]
color	1	[1:1]
Data format	1	[0:0]

C2H Interrupt Moderation

The QDMA Subsystem for PCIe provides a means to moderate the C2H completion interrupts. You can choose from one of the 7 modes to regulate the C2H completion interrupts. 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.

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_TIMER:** The QDMA Subsystem for PCIe maintains a timer for each QID running with this mode. The idea behind this mode is to interrupt the driver after a specific interval of time. These intervals of time can be configured by the driver. Every QID can be configured to use one of 16 programmable timer values.

- **TRIGGER_USER:** The QDMA Subsystem for PCIe provides the User 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 to generate an interrupt when the number of unread completion descriptors in the completion ring exceeds a certain threshold. This threshold is driver programmable on a per-queue basis. When this mode is set, the QDMA Subsystem for PCIe sends an interrupt whenever it determines that the number of unread completion descriptors in the completion queue has exceeded the threshold value. In this mode, the subsystem remains sensitive to any requests for interrupts sent by the user application along with the C2H packet.
- **TRIGGER_TIMER_COUNT:** This mode makes The QDMA Subsystem for PCIe generate interrupts when either sufficient time has passed or when the number of unread completion descriptors in the completion queue has exceeded the threshold value.
- **TRIGGER_USER_TIMER:** In this mode, the QDMA Subsystem for PCIe remains sensitive to any requests for interrupts sent by the user application along with the C2H packet in addition to interrupting the host if a certain amount of time has passed.
- **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 followings are the flow charts of different modes.

Figure 7: **Flowchart for EVERY Mode**

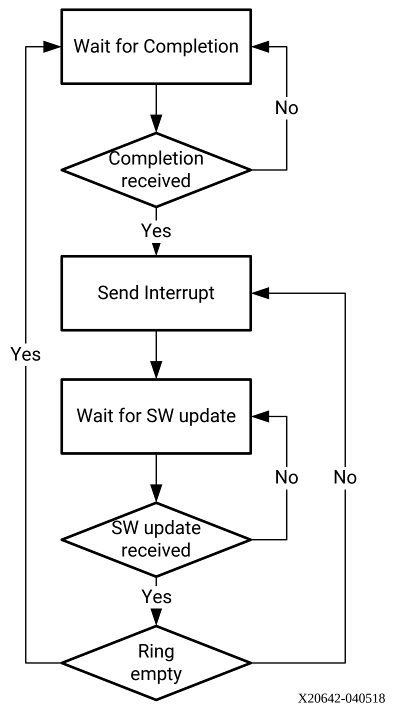


Figure 8: **Flowchart for TIMER Mode**

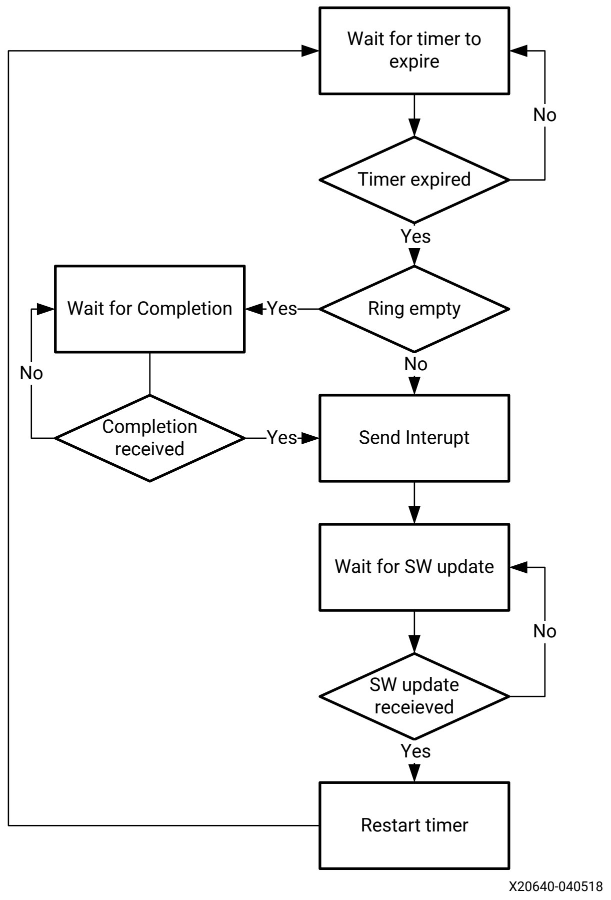


Figure 9: **Flowchart for USER Mode**

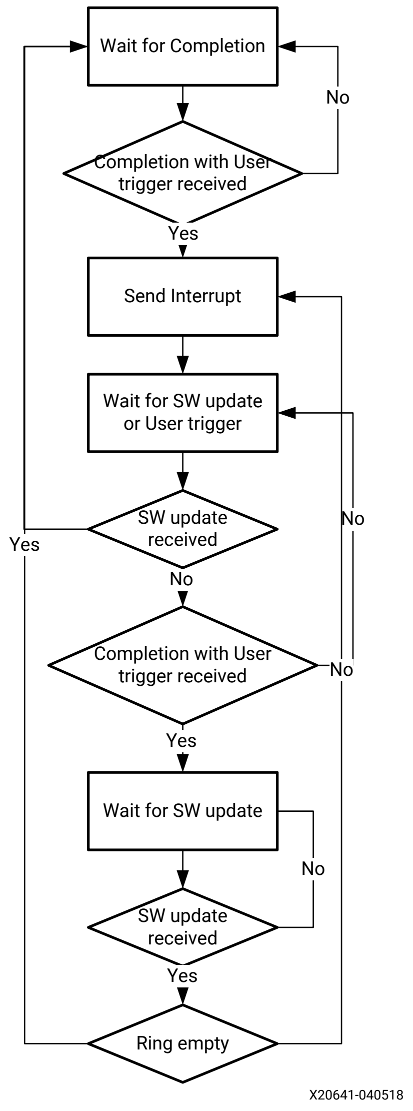


Figure 10: **Flowchart for USER_COUNT Mode**

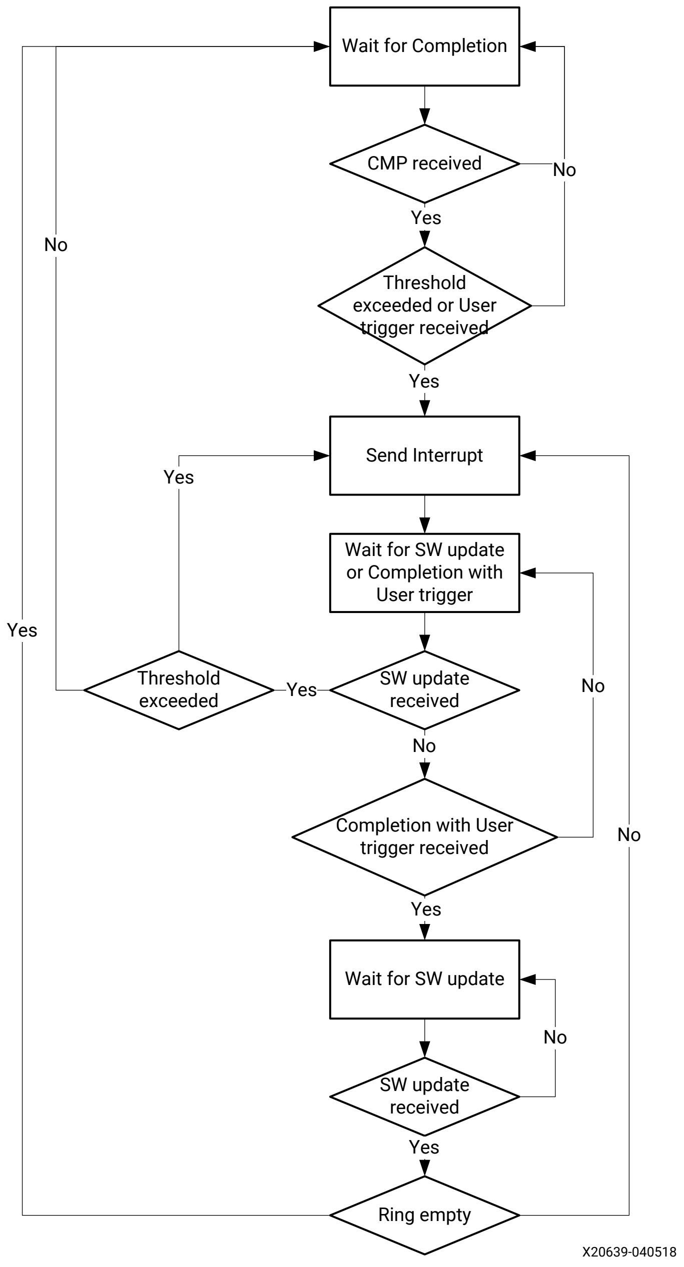


Figure 11: **Flowchart for TIMER_COUNT Mode**

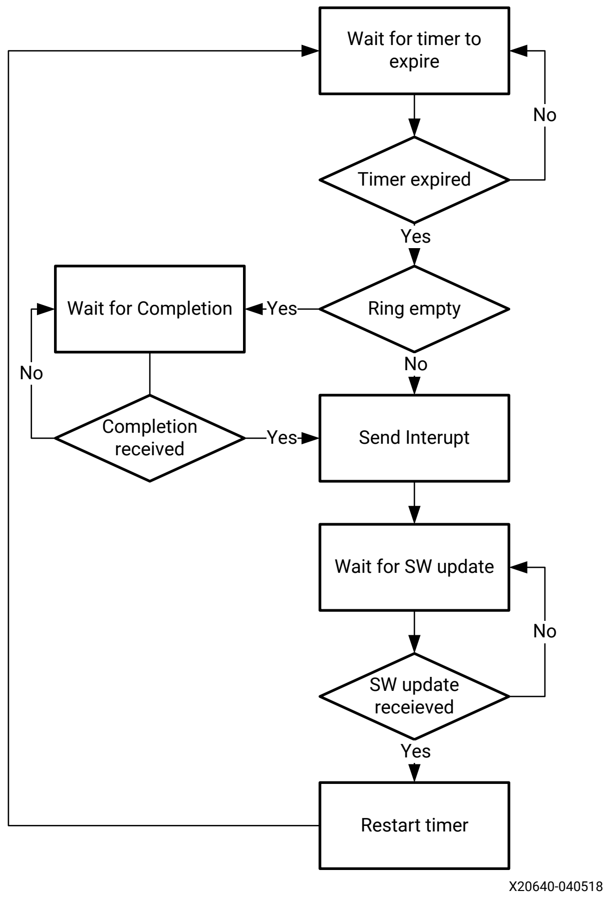
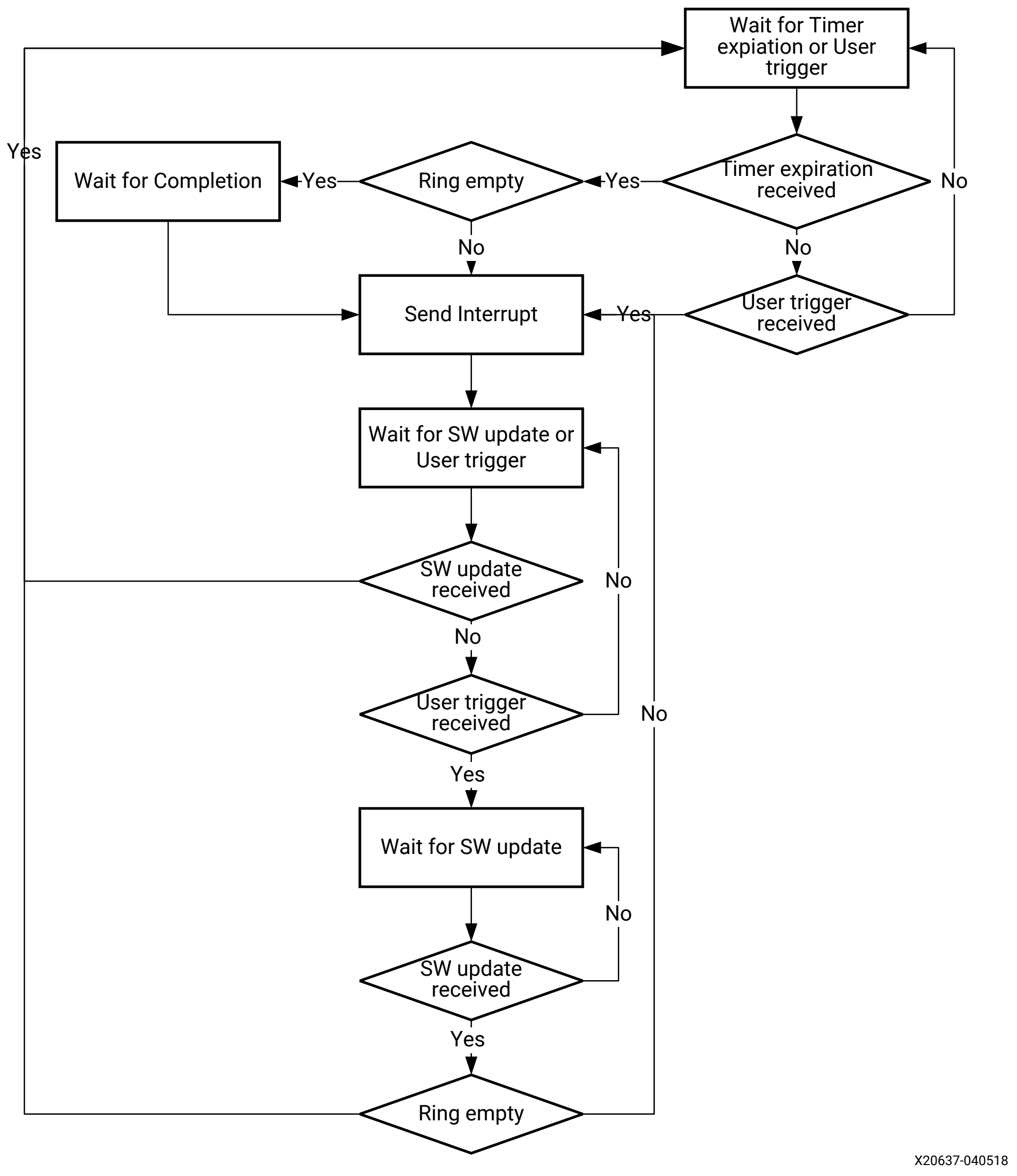
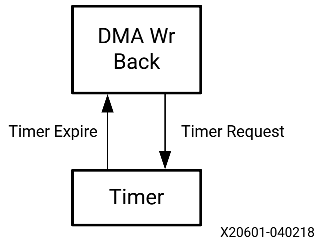


Figure 12: **Flowchart for USER_TIMER Mode**



C2H Timer

Figure 13: **C2H Timer**



The C2H timer is a trigger mode in the WRB context . It supports 2048 queues, and each queue has its own timer. When the timer expires, a timer expire signal is sent to the write back 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_TIMER_TICK defines the value for a timer tick. 16 QDMA_C2H_TIMER_CNT [7:0] registers have the timer counts based on the timer tick. The timer_ix is the index to the QDMA_C2H_TIMER_CNT registers.

Timer Quadrant

The Timer Quadrant allocates the timer injections into four quadrants. It stalls the reference timer when the previous quadrant still has active injections that are not yet expire. This guarantees the new timer injection do not conflict with the old timer injections.

C2H Registers	Access Type	Description
QDMA_C2H_INT_TIMER_TICK	RW	The value of a timer tick.
QDMA_C2H_TIMER_CNT	RW	16 registers. The bit [7:0] has the timer counts based on the timer tick.

SRIOV Support

The QDMA Subsystem for PCIe provides an optional feature to support the Single Root I/O Virtualization and Sharing (SR-IOV) based Virtualization.

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 the 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.

When the SR-IOV capability is enabled during the QDMA Subsystem for PCIe configuration, the subsystem allocates dedicated hardware resource which provides each VF independent DMA (optional), memory space and interrupts. The VF configuration space can be mapped to the virtual systems memory which enables direct access to the VF physical address including allowing the DMA access to the virtual system using Intel® Virtualization Technology for Directed I/O (VT-d). VT-d is responsible for I/O device assignments, the DMA, and Interrupt remapping. Moreover, in order to protect VMs from bad memory accesses, IOMMU must be enabled at the Host.

H2C Stream

The H2C engine is responsible for transferring data from the host and deliver it to the user logic. The H2C engine operates on the H2C descriptors. Each descriptor specifies the start address and the length of the data to be transferred to the user logic. The H2C 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 to hide the host read latency. The H2C engine implements a 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.

Based on the context that the H2C engine receives along with the descriptor, it could additionally be asked to send a status write back to the host once it is done issuing data to the user logic.

The H2C engine can be operated in two modes:

- **Internal mode**: In internal mode, after the descriptor is fetched from the host, it is fed straight to the H2C engine for processing. In internal mode, each descriptor transfers exactly one packet of data.
- **Bypass mode**: In bypass mode, after the descriptors are fetched from the host, they are sent to the user logic. The bypass logic stores these descriptors and then sends them back to the QDMA Subsystem for PCIe through the descriptor bypass-in interface. The descriptors are then fed to the H2C engine for processing.

The following are the advantages of using the bypass mode:

- The user logic can have a custom descriptor format.
- 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 Subsystem for PCIe when ready to sink all the data.
- Performs address translation.

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

Figure 14: **H2C Internal Mode Flow**

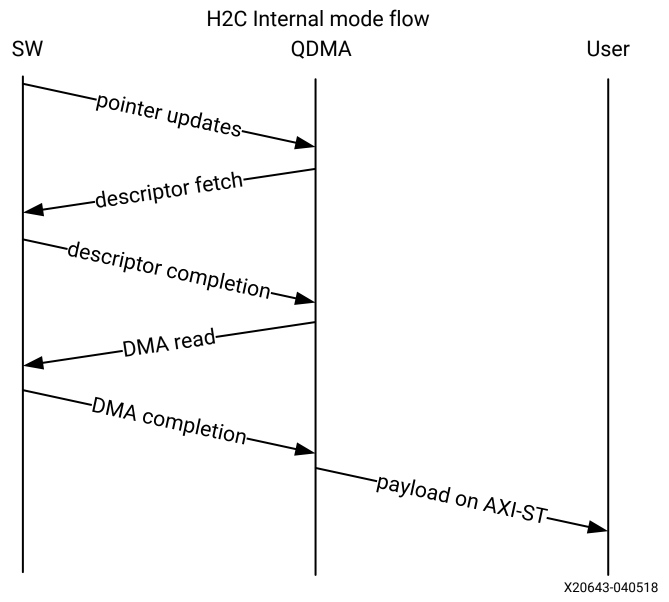
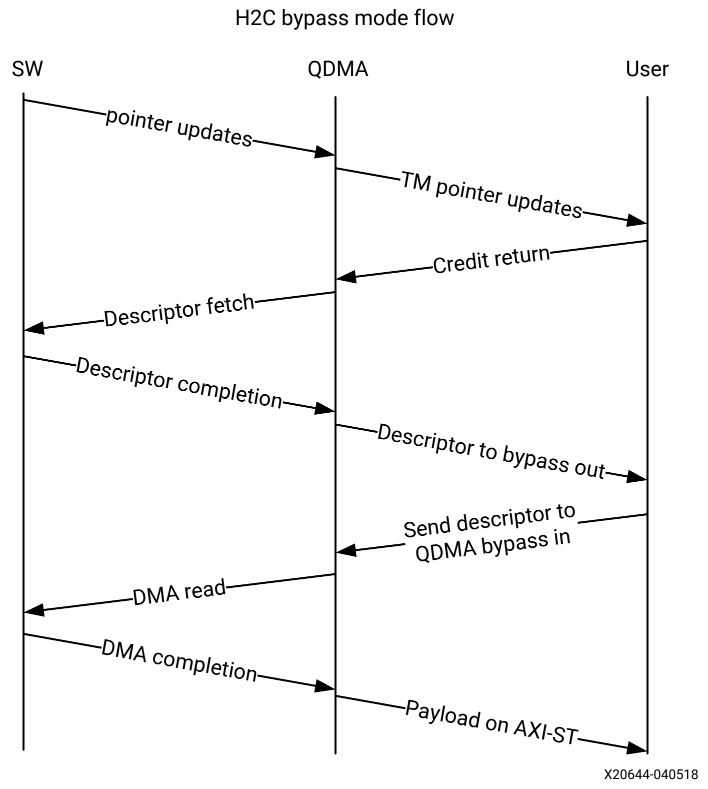


Figure 15: **H2C Bypass Mode Flow**



When using the H2C engine in internal mode, each descriptor transfers exactly one packet of data. The maximum length of the packet can be 64K-1 bytes. The descriptor format in internal mode is shown below:

Name	Size (b)	Index	Description
addr_h	32	[127:96]	Upper 32 bits of the address.
addr_l	32	[95:64]	Lower 32 bits of the address.
rsv1	16	[63:48]	Reserved
len	16	[47:32]	Length of data to the DMA.
rsv2	32	[31:0]	Reserved

When using the H2C engine in bypass mode, each packet can span over multiple descriptors. The maximum total length of the packet still needs to be 64K-1 bytes. In this mode, it is required that the user will send the batch of descriptors defining a packet to QDMA without interleaving with descriptors from other queues. The descriptor format used in this mode has two additional bits to specify whether a descriptor is the Start-Of-Packet or End-Of-Packet descriptor. The middle descriptors must have SOP=EOP=0. It is legal to have a descriptor with SOP=EOP=1. Shown below is the descriptor format in the bypass mode:

Name	Size (b)	Index	Description
addr_h	32	[127:96]	Upper 32 bits of the address.
addr_l	32	[95:64]	Lower 32 bits of the address.
rsv1	14	[63:50]	Reserved
eop	1	[49]	Descriptor marks end of packet.
sop	1	[48]	Descriptor marks start of packet.
len	16	[47:32]	Length of data to the DMA.
rsv2	32	[31:0]	Reserved

When feeding in the descriptor on the bypass in interface, the user logic can request that the QDMA Subsystem for PCIe sends 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 when the DMA is done. These behaviors can be controlled by using the `sdi` and `mrkr_req` bits in the bypass in interface. Refer to the QDMA Descriptor Bypass Input Ports description for details.

An alternative H2C descriptor format can be used. This is done by resetting the `use_stm_dsc_format` attribute. This alternative descriptor format has the same information as the ones described above except the fields are in different locations. The table below describe this other descriptor format.

Name	Ssize (b)	Index	Description
rsv2	34	[127:94]	Reserved

Name	Ssize (b)	Index	Description
eop	1	[94]	Descriptor marks end of packet.
sop	1	[93]	Descriptor marks start of packet.
rsv1	13	[92:80]	Reserved
len	16	[79:64]	Length of data to the DMA.
addr	64	[63:0]	64b address.

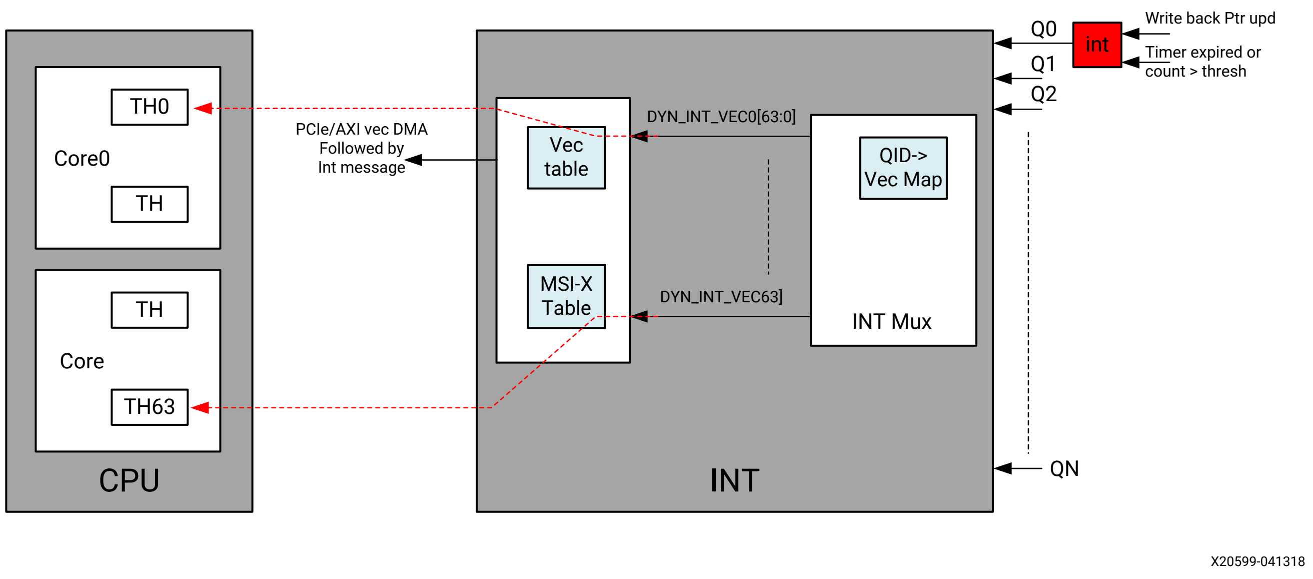
The H2C engine has a data aligner that aligns the data to 0B boundary before issuing it to the user logic. When in bypass mode, each descriptor defining the packet can have an arbitrarily aligned address and an arbitrarily aligned length. The aligner aligns and packs the data so that a continuous stream of data starting a 0B boundary is issued to the user logic.

QDMA Interrupts

The QDMA Subsystem for PCIe supports up to 256 MSI-X vectors, with up to 8 MSI-X vectors per function. Legacy interrupts are not supported in the subsystem. A single MSI-X vector can be used to support multiple queues. Each vector has an associated Interrupt Ring. The QID and status of queues requiring service are written into the Interrupt Ring. There can be at most one entry for each QID in the ring. When a PCIe MSI-X interrupt is received by the Host, the software reads the Interrupt Ring to determine which queues need service. Mapping of queues to vectors is programmable. It has independent table programming per PF. It supports all interrupt modes for non-SR-IOV, and MSI/MSI-X for SRIOV.

Queue-Based Interrupt Visualization

Figure 16: **Queue-Based Interrupt Visualization**



Interrupt Flow

When the H2C or C2H interrupt occur, the QID to vector RAM is read. The RAM has 2K entries to support up to 2K queues. It also includes two portions: one for H2C, and one for C2H. It maps the QID to the vector and indicates if it 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 used as the index of the Interrupt Context RAM.

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

Signals	Bits	Owner	Description
H2c_en_coal	1	Driver	1’b1: indirect interrupt mode. 1’b0: direct interrupt mode for H2C interrupt.
H2c_vector	8	Driver	Interrupt vector for the H2C interrupt.
C2h_en_coal	1	Driver	1’b1: indirect interrupt mode. 1’b0: direct interrupt mode for C2H interrupt.
C2h_vector	8	Driver	Interrupt vector for the C2H interrupt.

Direct Interrupt mode: For direct interrupt mode, the PCIe MSI-X message is out directly.

Indirect Interrupt mode: For indirect interrupt mode, it supports up to 256 Interrupt Rings and up to 256 functions. Each function can use one Interrupt Ring or multiple Interrupt Rings.

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

- Look up the QID to vector RAM.
- Look up the Interrupt Context RAM.
- Writes to the Interrupt Ring.
- Send out the PCIe MSI-X message.

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

Each entry of the Interrupt Context RAM has the following information for the associated Interrupt Ring.

Signals	Bits	Owner	Description
Pidx	12	DMA	Cumulative pointer of the Interrupt Ring entry written.

Signals	Bits	Owner	Description
Page_Size	3	Driver	Size of the Interrupt Ring. 0: 4KB 1: 8KB 2: 12KB 3: 16KB 4: 20KB 5: 24KB 6: 28KB 7: 32KB
Baddr_4k	52	Driver	Base Addr[63:12]. The Interrupt Ring in memory is 4K aligned.
Color	1	DMA	This bit inverts every time PIDX wraps around.
Int_st	1	DMA	Interrupt state. 0: WAIT_TRIGGER 1: ISR_RUNNING
Dbg_small_page_size	1	Driver	For debug purposes, supports some smaller ring sizes. {Dbg_small_page_size, Page_Size}: 4'b1000: 128B 4'b1001: 512B 4'b1010: 1KB
Vec	5	Driver	Interrupt Vector
Vld	1	Driver	Valid

After looking up the Interrupt Context RAM, it then writes to the Interrupt Ring. It also updates the Interrupt Context RAM with the new PIDX, color, and the interrupt state.

Each entry of the Interrupt Ring has 8B data with the following information.

Signals	Bits	Owner	Description
Pidx	12	DMA	Cumulative pointer of the Interrupt Ring entry written.
Page_Size	3	Driver	Size of the Interrupt Ring. 0: 4KB 1: 8KB 2: 12KB 3: 16KB 4: 20KB 5: 24KB 6: 28KB 7: 32KB
Baddr_4k	52	Driver	Base Addr[63:12]. The Interrupt Ring in memory is 4K aligned.
Color	1	DMA	This bit inverts every time PIDX wraps around.

Signals	Bits	Owner	Description
Int_st	1	DMA	Interrupt state. 0: WAIT_TRIGGER 1: ISR_RUNNING
Vec	6	Driver	Interrupt Vector
Vld	1	Driver	Valid

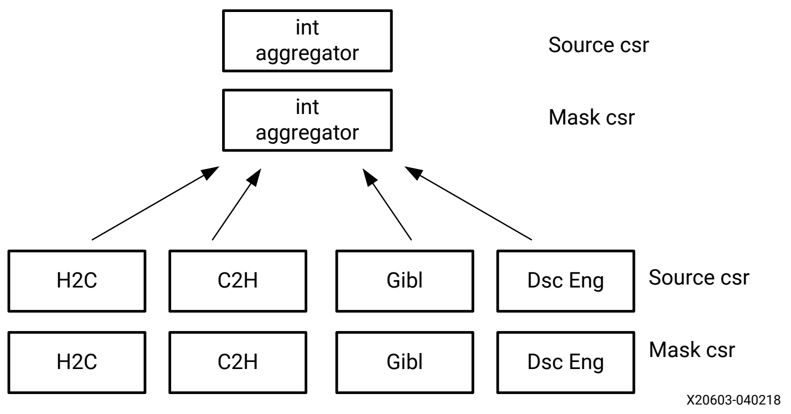
Finally, the QDMA Subsystem for PCIe sends out the PCIe MSI-X message using the interrupt vector from the Interrupt Context RAM.

When the PCIe MSI-X interrupt is received by the Host, the software reads the Interrupt Ring to determine which queues need service. After the software reads the Interrupt Ring, it will do a dynamic pointer update for the software CIDX to indicate the cumulative pointer that the software reads to. If the software CIDX is equal to the PIDX, this triggers a write to the Interrupt Ring on the interrupt state of that queue. If the software CIDX is not equal to the PIDX, it sends out another PCIe MSI-X message. Therefore, the software can read the Interrupt Ring again.

Asynchronous Internal 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 interrupt is configurable to any one of the PF.

Figure 17: **Asynchronous Interrupts**



In a Queue based scheme, interrupts are broadcast to all PFs and maintain status for each PF while in the async internal scheme. All async interrupts are configured to any one of the PFs.

The Source CSR registers are write 1 to clear type. Reset does not clear the status. They operate independent of the mask. The Mask CSR registers enables the associated source to participate in the interrupt.

Error Interrupt Handling

The Error Aggregator module aggregates all of the errors together. When the error occurs, it generates an Error Interrupt if the ARM bit is set. The ARM bit is set by the software and cleared by the hardware when the Error Interrupt is taken. The Interrupt Arbiter arbitrates the Error Interrupt with the other interrupts.

The Error Interrupt supports the direct interrupt mode and indirect interrupt mode. For the direct interrupt mode, it sends out the PCIe MSI-X message directly. For the indirect interrupt mode, it processes the interrupt with the following steps.

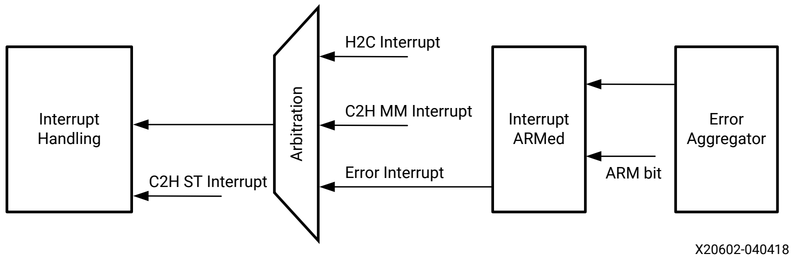
1. Reads the Error Interrupt register to get the vector.
2. Looks up the Interrupt Context RAM.
3. Writes to the Interrupt Ring.
4. Sends out the PCIe MSI-X message.

The following is the data in the error interrupt register.

Signals	Bits	Owner	Description
Err_int_arm	1	Driver	ARM bit; set by software and clear by hardware.
En_coal	1	Driver	1'b1: indirect interrupt mode 1'b0: direct interrupt mode
Vec	8	Driver	Interrupt Vector
Func	8	Driver	Function

The following shows how Error Interrupt handling.

Figure 18: **Error Interrupt Handling**



Function Level Reset

The FLR mechanism enables software to quiesce and reset Endpoint hardware with function-level granularity.

Errors

Linkdown

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

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 UltraScale+™ Devices 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.

Error Aggregator

There are Leaf Error Aggregators in different places. They log the errors and propagate them to a 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. We can 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).

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 enables the propagation of the corresponding error to the Central Error Aggregator. The error mask register does not affect the error logging to the error status register.

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

C2H Streaming Error

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

QDMA_C2H_ERR_MASK (0xAF4): 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): The QID of the first C2H streaming error.

QDMA_C2H MM Status (0x1040)

Table 2: **C2H MM Error Code (0x1058)**

Bit	Default	Access Type	Field	Description
[31:17]				Reserved
[16]			rdwr	Read or Write Error. 0: Read error 1: Write error
[15:0]			error_code	If Write Error, bit position: 2: RAM uncorrectable error 1: Unsupported request 0: Function level reset Other bits reserved If Read Error, bit position: 1: Slave error 0: Decode error

Table 3: **C2H0 MM Error Info (0x105C)**

Bit	Default	Access Type	Field	Description
[31:29]				Reserved
[28:17]			qid	Queue ID of the descriptor.
[16]			dir	Direction of descriptor.
[15:0]			cidx	Consumer index of the descriptor.

QDMA H2C0 MM Error

H2C0 MM Status (0x1240)

Table 4: **H2C0 MM Error Code (0x1258)**

Bit	Default	Access Type	Field	Description
[31:17]				Reserved
[16]			rdwr	Read or Write Error. 0: Read error 1: Write error
[15:0]			error_code	If Read Error, bit position: 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, bit position: 1: Slave error 0: Decode error

Table 5: **H2C0 MM Error Info (0x1258)**

Bit	Default	Access Type	Field	Description
[31:29]				Reserved
[28:17]			qid	Queue ID of the descriptor.
[16]			dir	Direction of descriptor.
[15:0]			cidx	Consumer index of the descriptor

TRQ Error

QDMA_GLBL_TRQ_ERR_STS (0x260): The error status register of the TRQ errors.

QDMA_GLBL_TRQ_ERR_MSK (0x264): The error mask register.

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

Descriptor Error

QDMA_GLBL_TRQ_ERR_STS (0x260): The error status register of the TRQ errors.

QDMA_GLBL_TRQ_ERR_MSK (0x264): The error mask register.

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

RAM Double Bit Error

Table 6: **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 writeback 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	Writeback 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.
[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.

Table 7: **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

RAM Single Error

Table 8: **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 writeback 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	Writeback 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.
[3:1]			reserved	
[0]			mi_h2c0_dat	H2C MM data buffer single bit ECC error.

Table 9: **QDMA_RAM_SBE_MSK_A(0xf0)**

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

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.

Applications

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

Feature Support Roadmap

The following is the feature roadmap for the QDMA Subsystem for PCIe.

Table 10: **Feature Support Roadmap**

Features	2018.1	Future Release
AXI-MM and AXI-ST only support	No	Yes
AXI-MM and AXI-ST both supported	Yes	
Interrupt coalescing	No	Yes
Descriptor Bypass in and out	No	Yes
Gen1/Gen2, 64/128 bit modes	Not tested	Will be tested in future release
PF Support	Only 2 PF	All 4 PF Support
VF Support	Driver supports only 8 VF	Up to 252 VFs supported
AXI-MM/AXI-ST Support for VFs	AXI-MM only	AXI-MM and AXI-ST both
Queues	All 2K queues	Queue number configurability
Vivado® IP integrator support	No	Yes
Linux Driver Support	Yes	
DPDK Driver Support	No	Yes

Table 10: **Feature Support Roadmap** *(cont'd)*

Features	2018.1	Future Release
Windows Driver Support	No	Yes
Timing	Not meeting with default options	Timing optimization in progress

Licensing and Ordering

This Xilinx® LogiCORE™ IP module is provided at no additional cost with the Xilinx® Vivado® 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.

Product Specification

Standards

This subsystem adheres to the following standard(s):

- ARM 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* (www.pcisig.com/specifications).

Minimum Device Requirements

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

Table 11: **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, x16	-2, -2L, -3

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 Figure 1 in QDMA Architecture.

Note: Some tables in this port list will change in 2018.2, including, but not limited to, the QDMA Descriptor Bypass Input/Output Ports (Ports not enabled in the IP), and the AXI4-Stream H2C/C2H Interfaces.

QDMA Global Ports

Table 12: **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.

AXI Bridge Master Ports

Table 13: **AXI4 Memory Mapped Master Bypass 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	3’h0
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	1’b0
m_axib_arcache[3:0]	O	4’h0
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] = bus number m_axib_aruser[18:16] = bar id m_axib_aruser[26:19] = vf offset m_axib_aruser[28:27] = vf id

Table 14: **AXI4 Memory Mapped Master Bypass 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_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 15: **AXI4 Memory Mapped Master Bypass 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.

Table 15: **AXI4 Memory Mapped Master Bypass Write Address Interface Port Descriptions** *(cont'd)*

Signal Name	I/O	Description
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	3’h0
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	1’b0
m_axib_awcache[3:0]	O	4’h0
m_axib_awuser[28:0]	O	Master write user bits. m_axib_awuser[7:0] = function number m_axib_awuser[15:8] = bus number m_axib_awuser[18:16] = bar id m_axib_awuser[26:19] = vf offset m_axib_awuser[28:27] = vf id

Table 16: **AXI4 Memory Mapped Master Bypass 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_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 17: **AXI4 Memory Mapped Master Bypass 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.
m_axib_bready	O	Master response ready.

AXI Bridge Slave Ports

Table 18: **AXI Bridge Slave 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
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	Reserved. Tie to GND.
s_axib_bid [C_S_AXI_ID_WIDTH-1:0]	O	Slave response ID
s_axib_bresp[1:0]	O	Slave write response
s_axib_bvalid	O	Slave write response valid
s_axib_bready	I	Slave response ready
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
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_rresp[1:0]	O	Slave read response

Table 18: **AXI Bridge Slave Port Descriptions** *(cont'd)*

Port Name	I/O	Description
s_axib_rlast	O	Slave read last
s_axib_rvalid	O	Slave read valid
s_axib_rready	I	Slave read ready
s_axib_ruser[C_S_AXI_DATA_WIDTH/8-1:0]	O	s_axib_aruser[7:0] indicates function number

AXI4-Lite Master Ports

Table 19: **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.
m_axil_awprot[2:0]	O	3'h0
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_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 20: **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_arprot[2:0]	O	3'h0
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 21: **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_awvalid	I	The assertion of this signal means there is a valid write request to the address on s_axil_awaddr.
s_axil_awprot[2:0]	I	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 22: **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_arprot[2:0]	I	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_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 Ports

Table 23: **AXI4 Memory Mapped Read Address Interface Signals**

Signal Name	Direction	Description
m_axi_araddr [63:0]	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 ARM AMBA AXI4-Stream Protocol Specification (ARM IHI 0051A).
m_axi_arlen[7:0]	O	Master read burst length.
m_axi_arsize[2:0]	O	Master read burst size.

Table 23: **AXI4 Memory Mapped Read Address Interface Signals** *(cont'd)*

Signal Name	Direction	Description
m_axi_arprot[2:0]	O	3’h0
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	1’b0
m_axi_arcache[3:0]	O	4’h0
m_axi_arburst[1:0]	O	Master read burst type.

Table 24: **AXI4 Memory Mapped Read Interface Signals**

Signal Name	Direction	Description
m_axi_rdata [255:0]	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	I	Parity ports for read interface. This port is enabled only in Propagate Parity mode.

Table 25: **AXI4 Memory Mapped Write Address Interface Signals**

Signal Name	Direction	Description
m_axi_awaddr[63:0]	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_awlen[7:0]	O	Master write address length.
m_axi_awsize[2:0]	O	Master write address size.
m_axi_awburst[1:0]	O	Master write address burst type.
m_axi_awprot[2:0]	O	3’h0
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	1’b0
m_axi_awcache[3:0]	O	4’h0

Table 26: **AXI4 Memory Mapped Write Interface Signals**

Signal Name	Direction	Description
m_axi_wdata[255:0]	O	Master write data.

Table 26: **AXI4 Memory Mapped Write Interface Signals** *(cont'd)*

Signal Name	Direction	Description
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	O	Parity ports for read interface. This port is enabled only in Propagate Parity mode.

Table 27: **AXI4 Memory Mapped 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 28: **AXI4-Stream H2C Port Descriptions**

Port Name	I/O	Description
m_axis_h2c_tdata [AXI_DATA_WIDTH-1:0]	O	
m_axis_h2c_dpar [AXI_DATA_WIDTH/8-1:0]	O	Odd parity calculated bit-per-byte over m_axis_h2c_tdata[255:0] m_axis_h2c_dpar [0] is parity calculated over m_axis_h2c_tdata[31:0] m_axis_h2c_dpar [1] is parity calculated over m_axis_h2c_tdata[63:32] and so on
m_axis_h2c_tuser [69:0]	O	Contains the following information: m_axis_h2c_tuser [10:0]: QID m_axis_h2c_tuser [11:11]: Channel ID m_axis_h2c_tuser [12]: SOP m_axis_h2c_tuser [13]: EOP m_axis_h2c_tuser [19:14]: LEB m_axis_h2c_tuser [25:20]: MEB m_axis_h2c_tuser [26]: WBC m_axis_h2c_tuser [34:27]: PCIe read error m_axis_h2c_tuser [37:35]: QDMA port ID m_axis_h2c_tuser [69:38]: bits[31:0] of the H2C descriptor which was used to fetch this data
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

AXI4-Stream C2H Ports

Table 29: **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
s_axis_c2h_dpar [AXI_DATA_WIDTH/8-1:0]	I	Data parity
s_axis_c2h_ctrl_len [15:0]	I	Length of the packet
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, we can safely do queue invalidation. When this bit is set, the imm_data bit has to be set too
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 30: **AXI4-Stream C2H Completion Port Descriptions**

Port Name	I/O	Description
s_axis_c2h_cmpt_tdata[127:0]	I	Completion data from the User. This contains information that is written to the completion ring in the host. This information includes length of the packet transferred in bytes, error, color bit and User data
s_axis_c2h_cmpt_size [1:0]	I	00: 8B completion 01: 16B completion 10: 32B completion 11: unknown
s_axis_c2h_cmpt_dpar [15:0]	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 Drop Ports

Table 31: **AXI-ST C2H Drops Port Descriptions**

Port Name	I/O	Description
axis_c2h_drop_valid	O	Valid
axis_c2h_drop_qid [10:0]	O	QID for which the packet was dropped
axis_c2h_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

Configuration Management Ports

Table 32: **Configuration Management Port Descriptions**

Port Name	I/O	Description
cfg_mgmt_addr [18:0]	I	Read/Write Address Configuration Space Dword-aligned address.
cfg_mgmt_write	I	Write Enable Asserted for a write operation. Active-High.
cfg_mgmt_write_data [31:0]	I	Write Data Write data is used to configure the Configuration and Management registers.
cfg_mgmt_byte_enable[3:0]	I	Byte Enable Byte enable for write data, where cfg_mgmt_byte_enable[0] corresponds to cfg_mgmt_write_data[7:0], and so on.
cfg_mgmt_read	I	Read Enable Asserted for a read operation. Active-High.
cfg_mgmt_read_data [31:0]	O	Read Data Out Read data provides the configuration of the Configuration and Management registers.
cfg_mgmt_read_write_done	O	Read/Write operation complete Asserted for 1 cycle when the operation is complete. Active-High.

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 33: **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. When neither user-implemented legacy or extended configuration space is enabled, receipt of a configuration read results in a one-cycle assertion of this signal, together with valid cfg_ext_register_number and cfg_ext_function_number. When user-implemented legacy, extended configuration space, or both are enabled, for the cfg_ext_register_number ranges, *0x10-0x1f* or *0x100-0x3ff* , respectively, this signal is asserted, until user logic presents cfg_ext_read_data and cfg_ext_read_data_valid. For cfg_ext_register_number ranges outside *0x10 - 0x1f* or *0x100 - 0x3ff* , receipt of a configuration read always results in a one-cycle assertion of this signal.
cfg_ext_write_received	O	1	Configuration Extend Write Received The core generates a one-cycle pulse on this output when it has received a configuration write request from the link.
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.
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.

FLR Ports

Table 34: **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 35: **QDMA H2C Descriptor Bypass Input Port Descriptions**

Port Name	I/O	Description
h2c_byp_in_dsc [255:0]	I	H2C Bypass In Descriptor[DJ1] The H2C descriptor provided by the User to QDMA. QDMA uses it to fetch data from the host
h2c_byp_in_sdi	I	H2C 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_cmpt_req	I	H2C 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_dsc_sz [1:0]	I	Descriptor size. 1: 16B H2C streaming, 2: 32B H2C MM
h2c_byp_in_st_mm	I	Indicates whether this is a streaming data descriptor or memory-mapped descriptor. 0: streaming; 1: memory-mapped
h2c_byp_in_qid [10:0]	I	The QID associated with the H2C descriptor ring
h2c_byp_in_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_func [7:0]	I	PCIe function ID
h2c_byp_in_cidx [15:0]	I	The CIDX that will be used for the status descriptor update and/or interrupt (coalescing mode). Generally the CIDX should be left unchanged from when it was received from the descriptor bypass output interface.

Table 35: **QDMA H2C Descriptor Bypass Input Port Descriptions** *(cont'd)*

Port Name	I/O	Description
h2c_byp_in_port_id [2:0]	I	QDMA port ID
h2c_byp_in_vld	I	Valid. High indicates descriptor is valid, one pulse for one descriptor.
h2c_byp_in_rdy	O	Ready to take in descriptor

Table 36: **QDMA C2H Descriptor Bypass Input Port Descriptions**

Port Name	I/O	Description
c2h_byp_in_dsc [255:0]	I	The C2H descriptor that the QDMA will use to send data to the host
c2h_byp_in_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 H2C ring context for this QID and armed by the driver
c2h_byp_in_cmpt_req	I	C2H 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
c2h_byp_in_dsc_sz [1:0]	I	Descriptor size. 0: 8B for C2H streaming; 2: 32B for C2H MM
c2h_byp_in_st_mm	I	Indicates whether this is a streaming data descriptor or memory-mapped descriptor. 0: streaming; 1: memory-mapped
c2h_byp_in_qid [10:0]	I	The QID associated with the C2H descriptor ring
c2h_byp_in_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_func [7:0]	I	PCIe function ID
c2h_byp_in_cidx [15:0]	I	The User must echo the CIDX from the descriptor that it received on the bypass-out interface
c2h_byp_in_port_id[2:0]	I	QDMA port ID
c2h_byp_in_vld	I	Valid. High indicates descriptor is valid, one pulse for one descriptor.
c2h_byp_in_rdy	O	Ready to take in descriptor

QDMA Descriptor Bypass Output Ports

Table 37: **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
h2c_byp_out_cmpt_rsp	O	Indicates completion status in response to h2c_byp_in_cmpt_req
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_qid [10:0]	O	The QID associated with the H2C descriptor ring

Table 37: **QDMA H2C Descriptor Bypass Output Port Descriptions** *(cont'd)*

Port Name	I/O	Description
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 38: **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
c2h_byp_out_cmpt_rsp	O	Indicates completion status in response to c2h_byp_in_cmpt_req
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_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
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.

QDMA Descriptor Complete Ports

Table 39: **QDMA TM Credit In Port Descriptions**

Port Name	I/O	Description
dsc_cmpt_dsc [255 : 0]	O	Completion status issued to the user application. This information is delivered in response to setting the h2c_byp_in_cmpt bit on the bypass-in interface when providing the descriptor to QDMA.
dsc_cmpt_dir	O	Indicates whether this completion status is for H2C or C2H descriptor. 0: H2C 1: C2H

Table 39: **QDMA TM Credit In Port Descriptions** *(cont'd)*

Port Name	I/O	Description
dsc_cmpt_st_mm	O	Indicates whether this is a streaming data descriptor or memory-mapped descriptor. 0: streaming 1: memory-mapped
dsc_cmpt_qid [10:0]	O	The QID associated with the descriptor ring for which this status update is being issued
dsc_cmpt_last	O	
dsc_cmpt_error	O	
dsc_cmpt_func [7:0]	O	PCIe function ID
dsc_cmpt_cidx [15:0]	O	
dsc_cmpt_port_id [2:0]	O	QDMA port ID
dsc_cmpt_vld	O	
dsc_cmpt_rdy	I	

QDMA Descriptor Credit Input Ports

Table 40: **QDMA Descriptor Credit Input Port Descriptions**

Port Name	I/O	Description
dsc_crdt_in_vld	I	Valid
dsc_crdt_in_rdy	O	Ready
dsc_crdt_in_dir	I	Indicates whether this completion status is 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 which this status update is being issued
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 41: **QDMA TM Credit Output Port Descriptions**

Port Name	I/O	Description
tm_dsc_sts_vld	O	Valid
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

Table 41: **QDMA TM Credit Output Port Descriptions** *(cont'd)*

Port Name	I/O	Description
tm_dsc_sts_mm	O	Indicates whether the status update is for a streaming or memory-mapped User. 0: streaming 1: memory-mapped
tm_dsc_sts_qid [10:0]	O	The QID of the ring
tm_dsc_sts_avl [7:0]	O	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_irq_arm	O	If set, it indicates to the User that the driver is ready to accept interrupts
tm_dsc_sts_port_id [2:0]	O	QDMA port ID
tm_dsc_sts_rdy	I	Ready. When this interface is not used, Ready must be tied-off to 1.

User Interrupts

Table 42: **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.
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

Table 43: **QDMA Address Register Space**

Target Name	Base (Hex)	Byte size (dec)	Notes
QDMA_TRQ_SEL_GLBL1 (0x00000)	00000000	256	QDMA Configuration CSR space

Table 43: **QDMA Address Register Space** *(cont'd)*

Target Name	Base (Hex)	Byte size (dec)	Notes
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	Functional 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_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_TRQ_EXT (0x2400)	00002400	16384	External register space
QDMA_TRQ_SEL_QUEUE_PF (0x6400)	00006400	32768	PF Direct QCSR (16B per Q, up to max of 2048 Qs per function)
PF Top Register Address	**0000E3FF**	**58368**	**PF BarAddr need to be 64KB align**
QDMA_TRQ_MSIX_VF (0x0000)	00000000	4096	Space for 32 MSIX vectors and PBA
QDMA_TRQ_EXT_VF (0x1000)	00001000	8192	External virtual function register space
QDMA_TRQ_SEL_QUEUE_VF (0x3000)	00003000	32768	VF Direct QCSR (16B per Q, up to max of 2048 Qs per function)
VF Top Register Address	**0000B000**	**45056**	**VF BarAddr alignment should be on largest VF bar window**

QDMA_TRQ_SEL_GLBL1 (0x00000)

Table 44: **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
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 Write Flush Timeout (0x60)	0x60	Config Write timeout control register

Config Block Identifier (0x00)

Table 45: **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 46: **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 47: **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 48: **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 49: **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 50: **Config Block MSI Enable (0x14)**

Bit	Default	Access Type	Field	Description
[0]	PCIe IP	RO	MSI_enable	MSI_en MSI Enable
[1]	PCIe IP	RO	MSIX_enable	MSI-X Enable

Config Block PCIE Data Width (0x18)

Table 51: **Config Block PCIE Data Width (0x18)**

Bit	Default	Access Type	Field	Description
[2:0]	C_DAT_WIDTH	RO	PCIe_AXI4_stream_width	pcie_width PCIe AXI4-Stream Width 0: 64 bits 1: 128 bits 2: 256 bits 3: 512 bits

Config PCIE Control (0x1C)

Table 52: **Config PCIE Control (0x1C)**

Bit	Default	Access Type	Field	Description
[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 53: **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. 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 54: **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
2:0	3'h5	RW	usr_max_read_request_prog	user_prg_read Maximum read request size issued to the user application. 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 Write Flush Timeout (0x60)

Table 55: **Config Write Flush Timeout (0x60)**

Bit	Default	Access Type	Field	Description
4:0	5'h0	RW	Write_flush_timeout	Write Flush Timeout Applies to AXI4-Stream C2H channels. This register specifies the number of clock cycles a channel waits for data before flushing the write data it already received from PCIe. This action closes the descriptor and generates a writeback. A value of 0 disables the timeout. The timeout value in clocks = 2 *value* .

QDMA_TRQ_SEL_GLBL2 (0x00100)

Table 56: **QDMA_TRQ_SEL_GLBL2 (0x00100) Register Space**

Register	Address	Description
QDMA_GLBL2_IDENTIFIER	0x100	Identifier 0x1FD3xxxx
QDMA_GLBL2_PF_BARLITE_INT	0x104	PF BAR information for internal DMA registers
QDMA_GLBL2_PF_VF_BARLITE_INT	0x108	VF BAR information for internal DMA registers

Table 56: **QDMA_TRQ_SEL_GLBL2 (0x00100) Register Space** *(cont'd)*

Register	Address	Description
QDMA_GLBL2_PF_BARLITE_EXT	0x10C	PF BAR information for External AXI-Lite Msater
QDMA_GLBL2_PF_VF_BARLITE_EXT	0x110	VF BAR information for External AXI-Lite Master
QDMA_GLBL2_CHANNEL_INST	0x114	
QDMA_GLBL2_CHANNEL_MDMA	0x118	
QDMA_GLBL2_CHANNEL_STRM	0x11C	
QDMA_GLBL2_MDMA_CAP	0x120	
QDMA_GLBL2_XDMA_CAP	0x124	
QDMA_GLBL2_PASID_CAP	0x128	
QDMA_GLBL2_FUNC_RET	0x12C	

QDMA_TRQ_SEL_GLBL (0x00200)

Table 57: **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_SCRATCH (0x244)	0x244	Reserved
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_RNG_SZ (0x204-0x240)

Table 58: **QDMA_GLBL_RNG_SZ (0x204-0x240)**

Bit	Default	Access Type	Field	Description
31:16	16’h0	NA		Reserved
15:0	16’h0	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 writeback 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

QDMA_GLBL_ERR_STAT (0X248)

Table 59: **QDMA_GLBL_ERR_STAT (0X248)**

Bit	Default	Access Type	Field	Description
[31:9]	0	NA	Reserved	Reserved

[8:0]	0	RW1C	error	Bit Error_location
				8 err_c2h_st;
				7 Reserved;
				6 err_c2h_mm_0;
				5 Reserved;
				4 err_h2c_mm_0;
				3 err_trq;
				2 err_dsc;
				1 err_ram_dbe;
				0 err_ram_sbe

QDMA_GLBL_ERR_MASK (0X24C)

Table 60: **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 61: **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

Table 61: **QDMA_GLBL_DSC_CFG (0x250)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[5:3]	0	RW	Max_dsc_fetch	Max nuber of descriptors to fetch in one request. 8 * 2^val // Max value is 6
[2:0]	0	RW	Wb_acc_int	Writeback accumulation 2^(val+1); Max 256;

Write back accumulation value is calculated as 2^(register bit [2:0]). Maximum accumulation is 256. Accumulation can be disabled via queue context

QDMA_GLBL_DSC_ERR_STS (0x254)

Table 62: **QDMA_GLBL_DSC_ERR_STS (0x254)**

Bit	Default	Access Type	Field	Description
[31]	0	RW1C	error	COR_ERR_CPLI_RAM_SBE
[30]	0	RW1C	error	COR_ERR_CPLD_RAM_SBE
[29]	0	RW1C	error	COR_ERR_HW_CTXT_RAM_SBE
[28]	0	RW1C	error	COR_ERR_CRD_RCV_RAM_SBE
[27]	0	RW1C	error	COR_ERR_SW_CTXT_RAM_SBE
[26]	0	RW1C	error	UNC_ERR_CPLI_RAM_DBE
[25]	0	RW1C	error	UNC_ERR_CPLD_RAM_DBE
[24]	0	RW1C	error	UNC_ERR_HW_CTXT_RAM_DBE
[23]	0	RW1C	error	UNC_ERR_CRD_RCV_RAM_DBE
[22]	0	RW1C	error	UNC_ERR_SW_CTXT_RAM_DBE
[20]	0	RW1C	error	UNC_ERR_DMA
[19]	0	RW1C	error	UNC_ERR_WR_FLR
[18]	0	RW1C	error	UNC_ERR_WR_UR
[17]	0	RW1C	error	UNC_ERR_DAT_PARITY
[16]	0	RW1C	error	UNC_ERR_DAT_POISON
[9]	0	RW1C	error	UNC_ERR_HDR_TIMEOUT
[5]	0	RW1C	error	UNC_ERR_HDR_FLR
[4]	0	RW1C	error	UNC_ERR_HDR_TAG
[3]	0	RW1C	error	UNC_ERR_HDR_ADDR
[2]	0	RW1C	error	UNC_ERR_HDR_PARAM
[1]	0	RW1C	error	UNC_ERR_HDR_UR_CA
[0]	0	RW1C	error	UNC_ERR_HDR_POISON

QDMA_GLBL_DSC_ERR_MSK (0x258)

Table 63: **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 64: **QDMA_GLBL_DSC_ERR_LOG0 (0x25C)**

Bit	Default	Access Type	Field	Description
[31:29]	0	RW	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 65: **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 1: PIDX update exceeded 255 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 66: **QDMA_GLBL_TRQ_ERR_STS (0x264)**

Bit	Default	Access Type	Field	Description
[31:1]	0	NA	Reserved	Reserved
[0]	0	RW1C	trq_err	Error valid

QDMA_GLBL_TRQ_ERR_MSK (0x268)

Table 67: **QDMA_GLBL_TRQ_ERR_MSK (0x268)**

Bit	Default	Access Type	Field	Description
[31:1]	0	NA	Reserved	Reserved
[0]	0	RW	mask	Mask enable

QDMA_GLBL_TRQ_ERR_LOG (0x26C)

Table 68: **QDMA_GLBL_TRQ_ERR_LOG (0x26C)**

Bit	Default	Access Type	Field	Description
[31:28]	0	NA	Reserved	Reserved
[27:24]	0	RW	Select	Register access space select 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_TRQ_SEL_FMAP (0x00400)

Table 69: **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 70: **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 71: **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 72: **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 73: **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 74: **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 75: **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.

Table 76: **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.

Table 77: **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.

Table 78: **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.

Table 79: **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 80: **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
				QDMA_CTXT_CMD_CLR = 0
				QDMA_CTXT_CMD_WR = 1
				QDMA_CTXT_CMD_RD = 2
				QDMA_CTXT_CMD_INV = 3

Table 80: **QDMA_IND_CTXT_CMD (0x824)** *(cont'd)*

Bit	Default	Access Type	Field	Description

[4:1]	0	RW	Sel	QDMA_CTXT_SELC_DEC_SW_C2H = 0
				QDMA_CTXT_SELC_DEC_SW_H2C = 1
				QDMA_CTXT_SELC_DEC_HW_C2H = 2
				QDMA_CTXT_SELC_DEC_HW_H2C = 3
				QDMA_CTXT_SELC_DEC_CR_C2H = 4
				QDMA_CTXT_SELC_DEC_CR_C2H = 5
				QDMA_CTXT_SELC_WRB = 6
				QDMA_CTXT_SELC_PFTCH = 7
				QDMA_CTXT_SELC_INT_COAL = 8

[0]	0	RW	busy	Write will be dropped when busy = 1 Read data is invalid when busy = 1

QDMA_TRQ_SEL_C2H (0x00A00)

Table 81: **QDMA_TRQ_SEL_C2H (0x00A00) Register Space**

Registers (Address)	Address	description
QDMA_C2H_TIMER_CNT[16] (0xA00-0xA3C)	0xA00-0xA3C	timer_count
QDMA_C2H_CNT_TH[16] (0xA40-0xA7C)	0xA40-0xA7C	threshold_count
QDMA_C2H_QID2VEC_MAP_QID (0xA80)	0xA80	The Queue ID index of the Qid2Vec RAM
QDMA_C2H_QID2VEC_MAP (0xA84)	0xA84	Map Queue ID to Vector
QDMA_C2H_STAT_S_AXIS_C2H_ACCEPTED (0XA88)	0xA88	Number of C2H packet accepted
QDMA_C2H_STAT_S_AXIS_WRB_ACCEPTED (0xA8C)	0xA8C	Number of C2H WRB packet accepted
QDMA_C2H_STAT_DESC_RSP_PKT_ACCEPTED (0xA90)	0xA90	Number of desc_rsp packet accepted from the Prefetch
QDMA_C2H_STAT_AXIS_PKG_CMP (0xA94)	0xA94	Number of axis packet completed from the C2H DMA Write Engine
QDMA_C2H_STAT_DESC_RSP_ACCEPTED (0xA98)	0xA98	Number of desc_rsp accepted including drop and error from the Prefetch
QDMA_C2H_STAT_DESC_RSP_CMP (0xA9C)	0xA9C	Number of desc_rsp completed including drop and error in the C2H DMA Write Engine
QDMA_C2H_STAT_WRQ_OUT (0xAA0)	0xAA0	Number of WRQ driven from the C2H DMA Write Engine
QDMA_C2H_STAT_WPL_REN_ACCEPTED (0xAA4)	0xAA4	Number of WPL REN accepted in the C2H DMA Write Engine
QDMA_C2H_STAT_TOTAL_WRQ_LEN (0xAA8)	0xAA8	Number of total WRQ length (including the empty packets) from the C2H DMA Write Engine
QDMA_C2H_STAT_TOTAL_WPL_LEN (0xAAC)	0xAAC	Number of total WPL length (including the empty packets) from the C2H DMA Write Engine

Table 81: **QDMA_TRQ_SEL_C2H (0x00A00) Register Space** *(cont'd)*

Registers (Address)	Address	description
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_C2H_ERR_INT (0B04)	0xB04	C2H error generated interrupt
QDMA_C2H_PFCH_CFG (0B08)	0xB08	Prefetch configuration
QDMA_C2H_INT_TIMER_TICK (0xB0C)	0xB0C	C2H interrupt timer tick
QDMA_C2H_STAT_DESC_RSP_DROP_ACCEPTED (0xB10)	0xB10	Number of dsc rsp with drop accepted
QDMA_C2H_STAT_DESC_RSP_ERR_ACCEPTED (0xB14)	0xB14	Number of dsc rsp with error accepted
QDMA_C2H_STAT_DESC_REQ (0xB18)	0xB18	Number of dsc request sent out from the C2H DMA Write Engine
QDMA_C2H_STAT_DEBUG_DMA_ENG_0 (0xB1C)	0xB1C	Debug registers 0
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 Prefetch error
QDMA_C2H_FIRST_ERR_QID (0xB30)	0xB30	The Qid of the first C2H error
QDMA_STAT_NUM_WRB_IN (0xB34)	0xB34	Number of WRB passed from DmaWrEnginre to Wrb block
QDMA_STAT_NUM_WRB_OUT (0xB38)	0xB38	Number of WRB(excluding STAT_DESC) passed from Wrb to WrbCoal block
QDMA_STAT_NUM_WRB_DRP (0xB3C)	0xB3C	Number of WRB dropped inside Wrb block
QDMA_STAT_NUM_STAT_DESC_OUT (0xB40)	0xB40	Number of STAT_DESC issued from Wrb to WrbCoal block
QDMA_STAT_NUM_DSC_CRDT_SENT (0xB44)	0xB44	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	Number of descriptors received from the fetch engine
QDMA_STAT_NUM_BYP_DSC_RCVD (0XB4C)	0xB4C	Number of descriptors received from the bypass path
QDMA_C2H_WRB_COAL_CFG (0xB50)	0xB50	C2H completion coalesce configuration

QDMA_C2H_TIMER_CNT[16] (0xA00-0xA3C)

Table 82: **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 writeback context to select its timer value using the timer count index field.

QDMA_C2H_CNT_TH[16] (0xA40-0xA7C)

Table 83: **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 writeback context to select its count threshold using the count threshold index field.

QDMA_C2H_QID2VEC_MAP_QID (0xA80)

Table 84: **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 85: **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 coalescing
[17:9]	0	RW	H2c_vector	H2C Vector
[8]	0	RW	C2h_en_coal	Enable the C2H interrupt coalescing
[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 coalescing ring index (max 256) if the interrupt coalescing is enabled.

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

QDMA_C2H_STAT_S_AXIS_C2H_ACCEPTED (0XA88)

Table 86: **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 87: **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 88: **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 89: **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 90: **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 91: **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 92: **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 93: **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 94: **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 95: **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 96: **QDMA_C2H_BUF_SZ[16] (0xAB0-0xAEC)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW	Size	C2H Buffer size for each descriptor in a given queue.

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 97: **QDMA_C2H_ERR_STAT (0xAF0)**

Bit	Default	Access Type	Field	Description
[31:13]	0	NA	Rsvd	Reserved
[12]	0	RW	wrb_inv_q_err	Flags the error if the SW updates an invalidated queue. When it does, the Write back pointer update.
[11]	0	RW	port_id_byp_in_mismatch	Port_id from the C2H packet and the Port_id from the bypass_in don't match
[10]	0	RW	port_id_ctxt_mismatch	Port_id from the C2H packet and the Port_id in the Prefetch context don't 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	timer_quad_cnt_err	The total amount of the timers in the Timer FIFOs should match with the total count from the timer quadrant
[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	Rsvd1	Reserved
[4]	0	RW	Rsvd2	Reserved
[3]	0	RW	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	RW	Rsvd3	Reserved
[1]	0	RW	len_mismatch	Flag the error if the total packet length doesn't match the signal from the s_axis_c2h_ctrl.len

Table 97: **QDMA_C2H_ERR_STAT (0xAF0)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[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 98: **QDMA_C2H_ERR_MASK (0xAF4)**

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

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

QDMA_C2H_FATAL_ERR_STAT (0xAF8)

Table 99: **QDMA_C2H_FATAL_ERR_STAT (0xAF8)**

Bit	Default	Access Type	Field	Description
[31:21]	0	RO	Reserved	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

Table 99: **QDMA_C2H_FATAL_ERR_STAT (0xAF8)** *(cont'd)*

Bit	Default	Access Type	Field	Description
[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 SW can write 1'b1 to clear the error if it wants to. The QDMA_C2H_FATAL_ERR_MASK register doesn't affect the error logging.

QDMA_C2H_FATAL_ERR_MASK (0xAFC)

Table 100: **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 101: **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_C2H_ERR_INT (0B04)

Table 102: **QDMA_C2H_ERR_INT (0B04)**

Bit	Default	Access Type	Field	Description
[31:11]	0	NA	Reserved	Reserved
[10]	0	RW	Err_int_arm	The SW sets the bit to arm the interrupt. The HW clears the bit when the interrupt is taken by the Interrupt Module.
[9]	0	RW	En_coal	1’b1: indirect interrupt; 1’b0: direct interrupt
[8]	0	RW	Vec	For the direct interrupt, this is the interrupt vector; For the indirect interrupt, this is the Interrupt Coalescing Context RAM index
[7:0]	0	RW	Func	Function

This register is for the error generated interrupt.

QDMA_C2H_PFCH_CFG (0B08)

Table 103: **QDMA_C2H_PFCH_CFG (0B08)**

Bit	Default	Access Type	Field	Description
[31:25]	0	RW	Evt_qcnt_th	Start Eviction when pfch qcnt >= evt_qcnt_th; The evc_qcnt_th should always be less than pfch_qcnt
[24:16]	0	RW	Pfch_qcnt	Max pfch qcnt allowed. Recommended value is <60
[15:8]	0	RW	Num_pfch	Controls number of entries prefetched in cache per queue. Recommended value is 8.
[7:0]	0	RW	Pfch_fl_th	Stop prefetch when FL Free count <= pfch_fl_th, minimum value is 16

QDMA_C2H_INT_TIMER_TICK (0xB0C)

Table 104: **QDMA_C2H_INT_TIMER_TICK (0xB0C)**

Bit	Default	Access Type	Field	Description
[31:0]	0	RW	Timer_tick	Value of a C2H timer tick

QDMA_C2H_STAT_DESC_RSP_DROP_ACCEPTED (0xB10)

Table 105: **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 106: **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 107: **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 108: **QDMA_C2H_STAT_DEBUG_DMA_ENG_0 (0xB1C)**

Bit	Default	Access Type	Field	Description
[31]	0	NA	Reserved	Reserved
[30:28]		RO	wrb_fifo_out_cnt	count of wrb fifo
[27:18]		RO	qid_fifo_out_cnt	count of qid fifo
[17:8]		RO	payload_fifo_out_cnt	count of payload fifo
[7:5]		RO	wrq_fifo_out_cnt	count of wrq fifo
[4]		RO	wrb_sm_cs	write back state machine
[3:0]		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 109: **QDMA_C2H_STAT_DEBUG_DMA_ENG_1 (0xB20)**

Bit	Default	Access Type	Field	Description
[31]	0	NA	Reserved	Reserved
[30]		RO	desc_rsp_last	desc_rsp_last signal
[29:20]		RO	payload_fifo_in_cnt	number of incoming entries to payload fifo
[19:10]		RO	payload_fifo_output_cnt	number of popup entries from payload fifo
[9:0]		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 110: **QDMA_C2H_STAT_DEBUG_DMA_ENG_2 (0xB24)**

Bit	Default	Access Type	Field	Description
[31:30]	0	NA	Reserved	Reserved
[29:20]		RO	wrb_fifo_in_cnt	number of incoming entries to wrb fifo
[19:10]		RO	wrb_fifo_output_cnt	number of popup entries from wrb fifo
[9:0]		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 111: **QDMA_C2H_STAT_DEBUG_DMA_ENG_3 (0xB28)**

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

QDMA_C2H_DBG_PFCH_ERR_CTXT (0xB2C)

Table 112: **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 113: **QDMA_C2H_FIRST_ERR_QID (0xB30)**

Bit	Default	Access Type	Field	Description
[31:20]		NA	Reserved	Reserved
[20:16]	0	RO	Err_type	4’b1111: NA 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’b1000: timer_quad_cnt_err 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
[15:12]		NA	Reserved	
[11:0]	0	RO	Qid	The Qid of the first C2H error

QDMA_STAT_NUM_WRB_IN (0xB34)

Table 114: **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 115: **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 116: **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 117: **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 118: **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 119: **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 120: **QDMA_STAT_NUM_BYP_DSC_RCVD (0XB4C)**

Bit	Default	Access Type	Field	Description
[31:11]	0	NA	Reserved	Reserved
[10:0]	0	RO	dsc_cnt	Number of descriptors received from the bypass path

QDMA_C2H_WRB_COAL_CFG (0xB50)

Table 121: **QDMA_C2H_WRB_COAL_CFG (0xB50)**

Bit	Default	Access Type	Field	Description
[31:26]	0	RW	max_buf_sz	To make the design see a certain value of the coalesce buffer. Used to hit coverage
[25:14]	0	RW	tick_val	coalesce buffer timer tick value
[13:2]	0	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 WRB in it
[0]	0	RW	done_glb_flush	coalesce buffer sets this bit when it flushes an entry

QDMA_TRQ_SEL_C2H_MM (0x1000)

Table 122: **QDMA_TRQ_SEL_C2H_MM (0x1000) Register Space**

Registers	Address	Description
C2H MM Control	0x04	Channel control bits
	0x08	Channel control bits W1S
	0x0C	Channel control bits C1S
C2H MM Status	0x40	Status bits
	0x44	Status clear
C2H Completed Descriptor Count	0x48	Completed Descriptor count

C2H MM Control

Table 123: **C2H Channel Control (0x04)**

Bit	Default	Access Type	Field	Description
31:28			Reserved	Reserved

Table 123: **C2H Channel Control (0x04)** *(cont'd)*

Bit	Default	Access Type	Field	Description
27	0x0	RW	c2h_stream_wrb_disable	Disables the metadata writeback for C2H AXI4-Stream. No effect if the channel is configured to use AXI Memory Mapped.
26	0x0	RW	pollmode_wrb_enable	pollmode_wb_enable Poll mode writeback enable. When this bit is set, the DMA writes back the completed descriptor count when a descriptor with the Completed bit set, is completed.
25	1’b0	RW	non_inc_mode	non_inc_mode Non-incrementing address mode. Applies to m_axi_araddr interface only.
23:19	5’h0	RW	desc_error	desc_error Set to all 1s (0x1F) to enable logging of Status.Desc_error and to stop the engine if the error is detected.
13:9	5’h0	RW	read_error	read_error Set to all 1s (0x1F) to enable logging of Status.Read_error and to stop the engine if the error is detected
6	1’b0	RW	idle_stopped	idle_stopped Set to 1 to enable logging of Status.Idle_stopped
5	1’b0	RW	invalid_length	invalid_length Set to 1 to enable logging of Status.Invalid_length
4	1’b0	RW	magic_stopped	magic_stopped Set to 1 to enable logging of Status.Magic_stopped
3	1’b0	RW	align_mismatch	align_mismatch Set to 1 to enable logging of Status.Align_mismatch
2	1’b0	RW	desc_completed	desc_completed Set to 1 to enable logging of Status.Descriptor_completed
1	1’b0	RW	desc_stopped	desc_stopped Set to 1 to enable logging of Status.Descriptor_stopped
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 124: **C2H Channel Control (0x08)**

Bit	Default	Access Type	Field	Description
		W1S		Control Bit descriptions are the same as in C2H Channel Control (0x04).

Table 125: **C2H Channel Control (0x0C)**

Bit	Default	Access Type	Field	Description
		W1C		Control Bit descriptions are the same as in C2H Channel Control (0x04).

C2H MM Status

Table 126: **C2H Channel Status (0x40)**

Bit	Default	Access Type	Field	Description
23:19	5'h0	RW1C	desc_error	descr_error[4:0] Reset (0) on setting the Control register Run bit. Bit position: 4:Unexpected completion 3: Header EP 2: Parity error 1: Completer abort 0: Unsupported request
13:9	5'h0	RW1C	read_error	read_error[4:0] Reset (0) on setting the Control register Run bit. Bit position: 4-2: Reserved 1: Slave error 0: Decode error
6	1'b0	RW1C	idle_stopped	idle_stopped Reset (0) on setting the Control register Run bit. Set when the engine is idle after resetting the Control register Run bit if the Control register ie_idle_stopped bit is set.
5	1'b0	RW1C	invalid_length	invalid_length Reset on setting the Control register Run bit. Set when the descriptor length is not a multiple of the data width of an AXI4-Stream channel and the Control register ie_invalid_length bit is set.
4	1'b0	RW1C	magic_stopped	magic_stopped Reset on setting the Control register Run bit. Set when the engine encounters a descriptor with invalid magic and stopped if the Control register ie_magic_stopped bit is set.
3	13'b0	RW1C	align_mismatch	align_mismatch Source and destination address on descriptor are not properly aligned to each other.
2	1'b0	RW1C	descriptor_completed	descriptor_completed Reset on setting the Control register Run bit. Set after the engine has completed a descriptor with the COMPLETE bit set if the Control register ie_descriptor_completed bit is set.

Table 126: **C2H Channel Status (0x40)** *(cont'd)*

Bit	Default	Access Type	Field	Description
1	1’b0	RW1C	descriptor_stopped	descriptor_stopped Reset on setting the Control register Run bit. Set after the engine completed a descriptor with the STOP bit set if the Control register ie_magic_stopped bit is set.
0	1’b0	RO	busy	busy Set if the SGDMA engine is busy. Zero when it is idle.

Table 127: **C2H Channel Status (0x44)**

Bit	Default	Access Type	Field	Description
23:1		RC		Status Bit descriptions are the same as in C2H Channel Status (0x40).

C2H Completed Descriptor Count

Table 128: **C2H Channel Completed Descriptor Count (0x48)**

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 (0x04).).

QDMA_TRQ_SEL_H2C_MM (0x1200)

Table 129: **QDMA_TRQ_SEL_H2C_MM (0x1200) Register Space**

Register	Address	Description
H2C MM Control	0x04	Channel control bits
	0x08	Channel control bits W1S
	0x0C	Channel control bits C1S
H2C MM Status	0x40	Status bits
	0x44	Status clear
H2C Completed Descriptor Count	0x48	Completed Descriptor count

H2C MM Control

Table 130: **H2C Channel Control (0x04)**

Bit	Default	Access Type	Field	Description
31:28			Reserved	Reserved
27	1'b0	RW	h2c_stream_wrb_disable	When set write back information for H2C in AXI-Stream mode is disabled, default write back is enabled.
26	0x0	RW	pollmode_wrb_enable	pollmode_wrb_enable Poll mode writeback enable. When this bit is set the DMA writes back the completed descriptor count when a descriptor with the Completed bit set, is completed.
25	1'b0	RW	non_inc_mode	non_inc_mode Non-incrementing address mode. Applies to m_axi_araddr interface only.
23:19	5'h0	RW	desc_error	desc_error Set to all 1s (0x1F) to enable logging of Status.Desc_error and to stop the engine if the error is detected.
18:14	5'h0	RW	write_error	write_error Set to all 1s (0x1F) to enable logging of Status.Write_error and to stop the engine if the error is detected.
13:9	5'h0	RW	read_error	read_error Set to all 1s (0x1F) to enable logging of Status.Read_error and to stop the engine if the error is detected.
8:7			Reserved	Reserved
6	1'b0	RW	idle_stopped	idle_stopped Set to 1 to enable logging of Status.Idle_stopped
5	1'b0	RW	invalid_length	invalid_length Set to 1 to enable logging of Status.Invalid_length
4	1'b0	RW	magic_stopped	magic_stopped Set to 1 to enable logging of Status.Magic_stopped
3	1'b0	RW	align_mismatch	align_mismatch Set to 1 to enable logging of Status.Align_mismatch
2	1'b0	RW	desc_completed	desc_completed Set to 1 to enable logging of Status.Descriptor_completed
1	1'b0	RW	desc_stopped	desc_stopped Set to 1 to enable logging of Status.Descriptor_stopped
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 131: **H2C Channel Control (0x08)**

Bit	Default	Access Type	Field	Description
26:0		W1S		Control Bit descriptions are the same as in H2C Channel Control (0x04).

Table 132: **H2C Channel Control (0x0C)**

Bit	Default	Access Type	Field	Description
26:0		W1C		Control Bit descriptions are the same as in H2C Channel Control (0x04).

H2C MM Status

Table 133: **H2C Channel Status (0x40)**

Bit	Default	Access Type	Field	Description
23:19	5’h0	RW1C	h2c_desc_err	descr_error[4:0] Reset (0) on setting the Control register Run bit. 4: Unexpected completion 3: Header EP 2: Parity error 1: Completer abort 0: Unsupported request
18:14	5’h0	RW1C	write_error	write_error[4:0] Reset (0) on setting the Control register Run bit. Bit position: 4-2: Reserved 1: Slave error 0: Decode error
13:9	5’h0	RW1C	read_error	read_error[4:0] Reset (0) on setting the Control register Run bit. Bit position 4: Unexpected completion 3: Header EP 2: Parity error 1: Completer abort 0: Unsupported request

Table 133: **H2C Channel Status (0x40)** *(cont'd)*

Bit	Default	Access Type	Field	Description
6	1’b0	RW1C	idle_stopped	idle_stopped Reset (0) on setting the Control register Run bit. Set when the engine is idle after resetting the Control register Run bit if the Control register ie_idle_stopped bit is set.
5	1’b0	RW1C	invalid_length	invalid_length Reset on setting the Control register Run bit. Set when the descriptor length is not a multiple of the data width of an AXI4-Stream channel and the Control register ie_invalid_length bit is set.
4	1’b0	RW1C	magic_stopped	magic_stopped Reset on setting the Control register Run bit. Set when the engine encounters a descriptor with invalid magic and stopped if the Control register ie_magic_stopped bit is set.
3	1’b0	RW1C	align_mismatch	align_mismatch Source and destination address on descriptor are not properly aligned to each other.
2	1’b0	RW1C	desc_completed	desc_completed Reset on setting the Control register Run bit. Set after the engine has completed a descriptor with the COMPLETE bit set if the Control register ie_descriptor_stopped bit is set.
1	1’b0	RW1C	desc_stopped	desc_stopped Reset on setting Control register Run bit. Set after the engine completed a descriptor with the STOP bit set if the Control register ie_descriptor_stopped bit is set.
0	1’b0	RO	busy	busy Set if the SGDMA engine is busy. Zero when it is idle.

Table 134: **H2C Channel Status (0x44)**

Bit	Default	Access Type	Field	Description
23:1		RC		Status Clear on Read. Bit description is the same as in H2C Channel Status (0x40). Bit 0 cannot be cleared.

H2C Completed Descriptor Count

Table 135: **H2C Channel Completed Descriptor Count (0x48)**

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 (0x04).

QDMA_TRQ_MSIX (0x1400)

QDMA_TRQ_EXT (0x2400)

Table 136: **QDMA_TRQ_EXT (0x2400) Register Space**

Register	Address	Description
Function Status Register (0x0)	0x00	Status bits
Function Command Register (0x04)	0x04	Command register bits
Target Function Register (0x0C)	0x0C	Function configuration register
PF Acknowledge Registers (0x20-0x3C)	0x20-0x3C	PF acknowledge
Incoming Message Memory (0x40-0x7C)	0x40-0x7C	Incoming message
Outgoing Message Memory (0x80-0xCC)	0x80-0xCC	Outgoing message
FLR Control/Status Register (0x100)	0x100	FLR control and status

Mailbox Addressing

PF addressing:

```
Addr = PF_Bar_offset + PF_Start_offset + CSR_addr
```

- PF_Start_offset = 0x2400

VF addressing:

```
Addr = VF_Bar_offset + VF_Start_offset + VF_offset + CSR_addr
```

- VF_Start_offset = 0x1000
- VF_offset = VFG_offset * VF_apperture_size
- VFG_offset is the function offset within the vfg.

- VF_apperture_size = 32KB (GUI option can be changed)

Function Status Register (0x0)

Table 137: **Function Status Register (0x0)**

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 (0x04)

Table 138: **Function Command Register (0x04)**

Bit	Default	Access Type	Field	Description
[31:3]	0	NA	Reserved	Reserved
2	0	RO	Reserved	Reserved

Table 138: **Function Command Register (0x04)** *(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.

Target Function Register (0x0C)

Table 139: **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.

PF Acknowledge Registers (0x20-0x3C)

Table 140: **PF Acknowledge Registers (0x20-0x3C)**

Register	Addr	Default	Access Type	Field	Width	Description
Ack0	0x20	0	RW		32	Acknowledgement from FN 31~0
Ack1	0x24	0	RW		32	Acknowledgement from FN 63~32
Ack2	0x28	0	RW		32	Acknowledgement from FN 95~64
Ack3	0x2c	0	RW		32	Acknowledgement from FN 127~96
Ack4	0x30	0	RW		32	Acknowledgement from FN 159~128
Ack5	0x34	0	RW		32	Acknowledgement from FN 191~160
Ack6	0x38	0	RW		32	Acknowledgement from FN 223~192
Ack7	0x3c	0	RW		32	Acknowledgement from FN 255~224

Incoming Message Memory (0x40-0x7C)

Table 141: **Incoming Message Memory (0x40-0x7C)**

Register	Addr	Default	Access Type	Field	Width	Description
i_msg_0	0x40	0	RW		32	Input message byte 3 ~ 0
i_msg_1	0x44	0	RW		32	Input message byte 7 ~ 4
i_msg_2	0x48	0	RW		32	Input message byte 11 ~ 8
i_msg_3	0x4c	0	RW		32	Input message byte 15 ~ 12
i_msg_4	0x50	0	RW		32	Input message byte 19 ~ 16
i_msg_5	0x54	0	RW		32	Input message byte 25 ~ 20
i_msg_6	0x58	0	RW		32	Input message byte 27 ~ 24
i_msg_7	0x5c	0	RW		32	Input message byte 31 ~ 28
i_msg_8	0x60	0	RW		32	Input message byte 35 ~ 32
i_msg_9	0x64	0	RW		32	Input message byte 39 ~ 36
i_msg_10	0x68	0	RW		32	Input message byte 43 ~ 40
i_msg_11	0x6c	0	RW		32	Input message byte 47 ~ 44
i_msg_12	0x70	0	RW		32	Input message byte 51 ~ 48
i_msg_13	0x74	0	RW		32	Input message byte 55 ~ 52
i_msg_14	0x78	0	RW		32	Input message byte 59 ~ 56
i_msg_15	0x7c	0	RW		32	Input message byte 63 ~ 60

Outgoing Message Memory (0x80-0xCC)

Table 142: **Outgoing Message Memory (0x80-0xCC)**

Register	Addr	Default	Access Type	Field	Width	Description
o_msg_0	0x80	0	RW		32	Output message byte 3 ~ 0
o_msg_1	0x84	0	RW		32	Output message byte 7 ~ 4
o_msg_2	0x88	0	RW		32	Output message byte 11 ~ 8
o_msg_3	0x8c	0	RW		32	Output message byte 15 ~ 12
o_msg_4	0xa0	0	RW		32	Output message byte 19 ~ 16
o_msg_5	0xa4	0	RW		32	Output message byte 25 ~ 20
o_msg_6	0xa8	0	RW		32	Output message byte 27 ~ 24
o_msg_7	0xac	0	RW		32	Output message byte 31 ~ 28
o_msg_8	0xb0	0	RW		32	Output message byte 35 ~ 32
o_msg_9	0xb4	0	RW		32	Output message byte 39 ~ 36
o_msg_10	0xb8	0	RW		32	Output message byte 43 ~ 40
o_msg_11	0xbc	0	RW		32	Output message byte 47 ~ 44
o_msg_12	0xc0	0	RW		32	Output message byte 51 ~ 48
o_msg_13	0xc4	0	RW		32	Output message byte 55 ~ 52

Table 142: **Outgoing Message Memory (0x80-0xCC)** *(cont'd)*

Register	Addr	Default	Access Type	Field	Width	Description
o_msg_14	0xc8	0	RW		32	Output message byte 59 ~ 56
o_msg_15	0xcc	0	RW		32	Output message byte 63 ~ 60

FLR Control/Status Register (0x100)

Table 143: **FLR Control/Status Register (0x100)**

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.

QDMA_TRQ_SEL_QUEUE_PF (0x6400)

Table 144: **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 145: **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 146: **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 147: **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 148: **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
[26:24]	0	RW	trigger_mode	Trigger mode WRB_TRIG_DIS, WRB_TRIG_ANY, WRB_TRIG_TIMER, WRB_TRIG_CNT, WRB_TRIG_COMBO, WRB_TRIG_USR
[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_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_TRQ_EXT_VF (0x1000)

VF functions can access External registers (Mali box and flr registers) with offset (0x1000). The description for this register space is the same as QDMA_TRQ_EXT (0x2400). There are some restrictions as noted in the register descriptions.

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
	0c3004 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
	0c3014 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

Context Structure Definition

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

The descriptor context is used by the descriptor engine.

Table 149: **Software Descriptor Context Structure Definition**

Bit	Bit Width	Field Name	Group	Description
[127:64]	64	dsc_base	dsc_base	Base address of Descriptor Ring
[63:61]	3	rsv	dsc_sts	Reserved for status
[60]	1	err_wb_sent	"	
[59:58]	2	err	"	Error status. Bit[1] dma, Bit[0] dsc
[57]	1	irq_no_last	"	No interrupt was sent and pidx/cidx was idle
[56]	1	irq_pnd	"	Interrupt pending
[55:54]	2	rsv0	dsc_ctrl	Reserved for control
[53]	1	irq_en	"	Interrupt enable
[52]	1	wbk_en	"	Writeback enable (Disable for C2H stream)
[51]	1	mm_chn	"	If 32B descriptor which MM channel to use
[50]	1	byp	"	Send to descriptor bypass out
[49:48]	2	dsc_sz	"	Descriptor size. 0: 8B, 1:16B; 2:32B; 3:rsv
[47:44]	4	rng_sz	"	Descriptor ring size index to ring size registers
[43:36]	8	fnc_id	"	Function ID
[35]	1	wbi_acc_en	"	Write back/Interrupt after accumulation
[34]	1	wbi_chk	"	Writeback/Interrupt after pending check
[33]	1	fcrd_en	"	Enable fetch credit
[32]	1	qen	"	valid
[31:17]	15	rsv	dsc_pidx	Reserved

Table 149: **Software Descriptor Context Structure Definition** *(cont'd)*

Bit	Bit Width	Field Name	Group	Description
[16]	1	irq_ack	"	Interrupt Ack
[15:0]	16	pidx	"	Producer Index

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

Table 150: **Hardware Descriptor Context Structure Definition**

Bit	Bit Width	Field Name	Description
[47:42]	6	rsvd	Reserved
[41]	1	idl_stp_b	Queue invalid and no descriptors pending
[40]	1	pnd	Descriptor pending
[39:32]	8	wb_acc	Writeback accumulator count
[31:16]	16	crd_use	credit use
[15:0]	16	cidx	Consumer Index

Credit Descriptor Context Structure

Table 151: **Credit Descriptor Context Structure Definition**

Bit	Bit Width	Field Name	Description
[31:16]	6	rsvd	Reserved
[15:0]	16	credt	Hardware Credit that is processed by DESC fetch engine

C2H CMPT Context Structure (0x6)

The writeback context is used by the writeback engine.

Table 152: **C2H Writeback Context Structure Defintion**

Bit	Bit Width	Field Name	Description
[127:123]	5	rsvd	Reserved
[122:121]	2	err	Error
[120]	1	valid	Context is valid
[119:104]	16	cidx	Initial Consumer Index
[103:88]	16	pidx	Initial Producer Index

Table 152: **C2H Writeback Context Structure Defintion** *(cont'd)*

Bit	Bit Width	Field Name	Description
[87:86]	2	desc_size	Descriptor Size: 8B:0 16B:1 32B:2 Unknown:3
[85:28]	58	baddr_64	Base address of Writeback ring – bit[63:6]
[27:24]	4	qsize_idx	Writeback ring size index to ring size registers
[23]	1	color	Initial color bit to be used on writeback
[22:21]	2	int_st	Interrupt State: ISR:0, TRIG:1, ARMED:2
[20:17]	4	timer_idx	Index to timer register to Wrb trigger timer
[16:13]	4	counter_idx	Index to counter register to Wrb on trigger counter
[12:5]	8	fnc_id	Function ID
[4:2]	3	trig_mode	Trigger Mode: Disable:0 Any:1 Timer:2 Counter:3 Combo:4 User:5
[1]	1	en_int	Cause Interrupt on Writeback
[0]	1	en_stat_desc	Cause Status Descriptor write on Writeback

C2H Prefetch Context Structure (0x7)

The prefetch context is used by the C2H prefetch engine which interact between descriptor fetch engine and DMA write engine to pair up the descriptor and its payload.

Table 153: **C2H Prefetch Context Structure Definition**

Bit	Bit Width	Field Name	Bit Location	Description
[45]	1	valid	[45]	Context is valid
[44:29]	16	sw_crdt	[44:29]	Software credit (RO)
[28:28]	1	pfch	[28:28]	Queue is in prefetch
[27:27]	1	pfch_en	[27:27]	Enable prefetch
[26:16]	11	rsv	[26:16]	Reserve
[15:13]	3	port_id	[15:13]	Port ID
[12:5]	8	fnc_id	[12:5]	Function ID
[4:1]	4	buf_size_idx	[4:1]	Buffer size index
[0]	1	bypass	[0]	C2H is in bypass mode

Interrupt Context Structure (0x8)

Table 154: **Interrupt Context Structure Definition**

Bit	Bit Width	Field Name	Description
[75:64]	12	pidx	Producer Index
[63:61]	3	page_size	Page size
[60:9]	52	baddr_4k	Base address of Interrupt ring – bit[63:12]
[8]	1	color	Color bit
[7]	1	int_st	Interrupt ISM Status: WAIT_TRIGGER:0, ISR_RUNNING:1
[6:1]	6	vec	Vector ID
[0]	1	valid	Valid

Queue Entry Structure

Note: Descriptor formats for AXI4-Stream H2C Descriptors and AXI4-Stream C2H Descriptors will change in 2018.2.

AXI4-Stream C2H Descriptor (8B)

Table 155: **AXI4-Stream C2H Descriptor Structure**

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

AXI4-Stream H2C Descriptor (16B)

Table 156: **AXI4-Stream H2C Descriptor Structure**

Bit	Bit Width	Field Name	Description
[127:95]	33	rsvd	Reserved
[94]	1	eop	End of Packet
[93]	1	sop	Start of Packet
[92]	1	dv	Descriptor Valid
[91:64]	28	lengthInByte	Read length in byte
[63:0]	64	addr	Source Address

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

Table 157: **AXI4 Memory Mapped Descriptor Structure for H2C and C2H**

Bit	Bit Width	Field Name	Description
[256:192]	64	rsvd1	Reserved
[191:64]	64	dst_addr	Destination Address
[127:95]	33	rsvd0	Reserved
[94]	1	eop	End of Packet
[93]	1	sop	Start of Packet
[92]	1	dv	Descriptor Valid
[91:64]	28	lengthInByte	Read length in byte
[63:0]	64	src_addr	Source Address

Writeback Structure

AXI4-Stream C2H Writeback Entry Structure

The writeback header is an entry in the writeback ring.

Table 158: **AXI4-Stream C2H Writeback Entry Structure**

Bit	Bit Width	Field Name	Description
[255:20] [127:20] [63:20]	236 bits 108 bits 44 bits	User defined	User defined bits for 32 Bytes settings. User defined bits for 16 Bytes settings. User defined bits for 8 Bytes settings.
[19:4]	16	Len	Total length for this transfer (Could be sum of multiple descriptor)
[3:3]	1	Reserved	Reserved
[2:2]	1	Err	Write back entry Error
[1:1]	1	Color	Cause Status Descriptor write on Writeback
[0:0]	1	Reserved	Reserved

AXI4-Stream C2H Writeback Status Structure

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

When **Writeback Status Descriptor** is enabled, the PIDX is used to indicate the currently available writeback to be processed.

Table 159: **AXI4-Stream C2H Writeback Status Structure**

Bit	Bit Width	Field Name	Description
[63:35]	29	Reserve	Reserved
[34:33]	2	int_state	Interrupt State: ISR:0, TRIG:1, ARMED:2
[32]	1	color	Color status bit
[31:16]	16	cidx	Consumer Index (RO)
[15:0]	16	pidx	Producer Index

AXI4-Stream H2C Writeback Status Structure

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

Table 160: **AXI4-Stream H2C Writeback Status 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)

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

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

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 an 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.

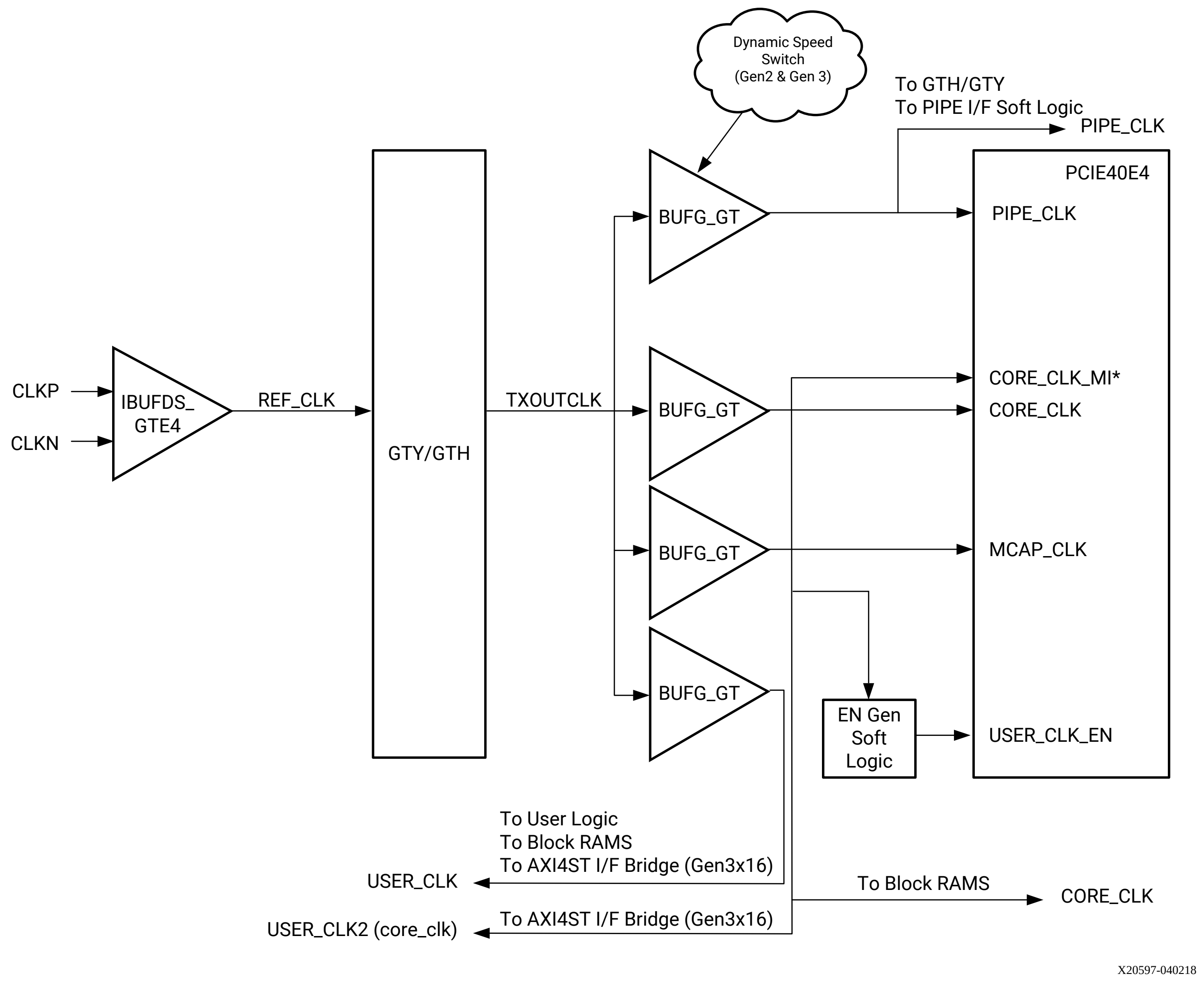
Related Information
Xilinx Resources

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 20: **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`.

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.

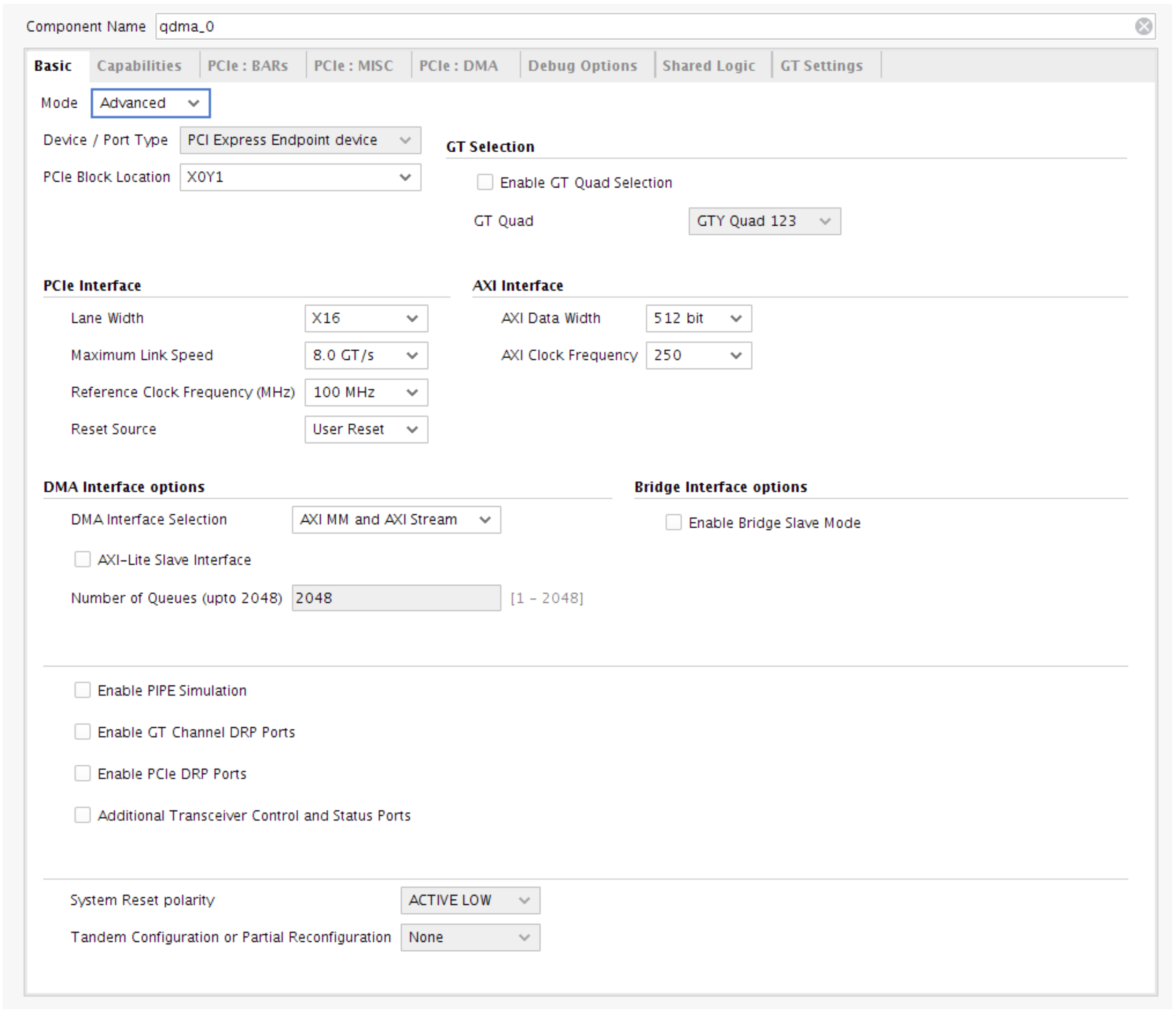
For details, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) and the *Vivado Design Suite User Guide: Getting Started* (UG910).

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 21: **Basic Tab**



Component Name: qdma_0

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	8.0 GT/s
Reference Clock Frequency (MHz)	100 MHz
Reset Source	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

System Reset polarity: ACTIVE LOW

Tandem Configuration or Partial Reconfiguration: None

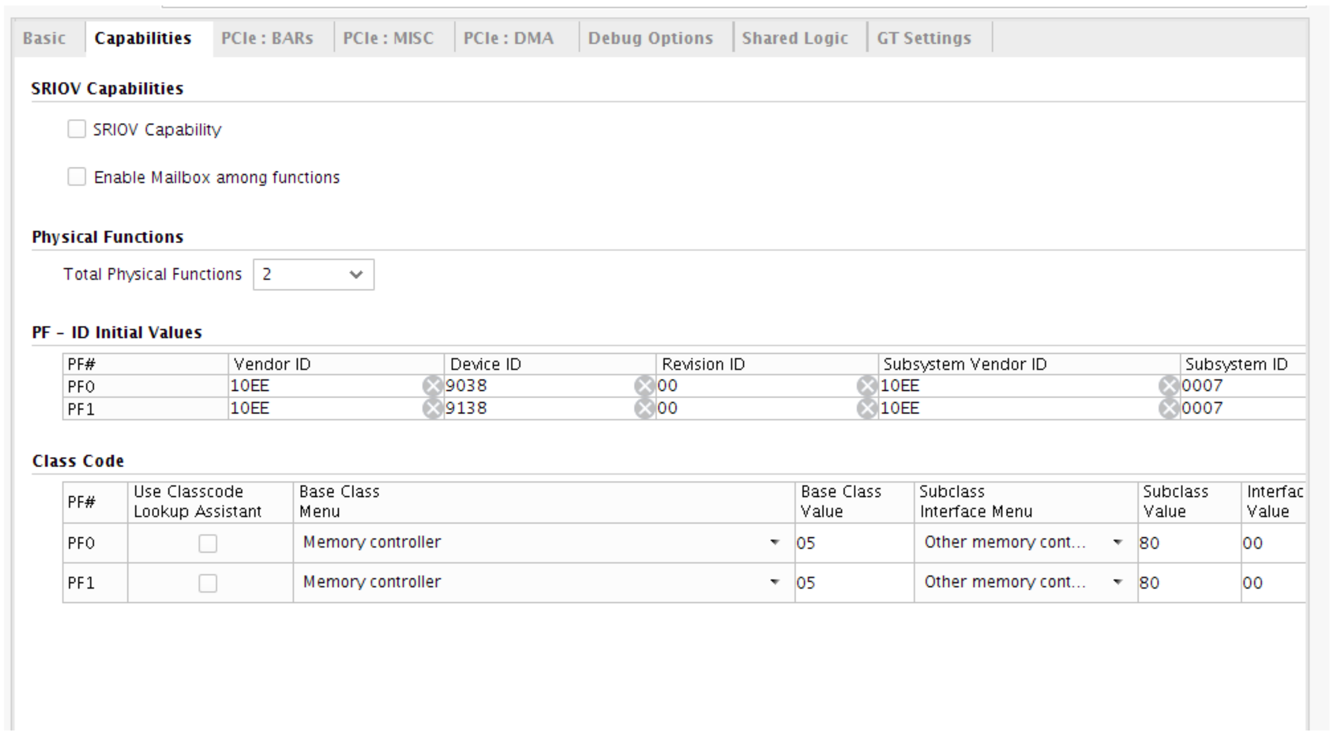
- **Mode:** Allows you to select the Basic or Advanced mode of the configuration of core.
- **Device /Port Type:** Only PCI Express® Endpoint device mode is supported.
- **GT Selection/Enable GT Quad Selection:** Select the Quad in which lane 0 is located.

- **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. Default option is Gen3.
- **Reference Clock Frequency:** The default is 100 MHz.
- **Reset Source:** You can choose one of:
    - **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.
- **Total Physical Functions:** you can choose between 1 or 2 physical function being present in the core.
- **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:** AXI4 Memory Mapped and AXI4-Stream.
- **AXI Lite Slave Interface:** Select to enable the AXI4-Lite 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.
- **System Reset polarity:** System Reset polarity can be selected to be active high or low.
- **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 22: **Capabilities Tab**



Basic | Capabilities | PCIe : BARs | PCIe : MISC | PCIe : DMA | Debug Options | Shared Logic | GT Settings

SRIOV Capabilities

- [ ] SRIOV Capability
- [ ] Enable Mailbox among functions

Physical Functions

Total Physical Functions: 2

PF - ID Initial Values

PF#	Vendor ID	Device ID	Revision ID	Subsystem Vendor ID	Subsystem ID
PF0	10EE	9038	00	10EE	0007
PF1	10EE	9138	00	10EE	0007

Class Code

PF#	Use Classcode Lookup Assistant	Base Class Menu	Base Class Value	Subclass Interface Menu	Subclass Value	Interfac Value
PF0	☐	Memory controller	05	Other memory cont...	80	00
PF1	☐	Memory controller	05	Other memory cont...	80	00

- **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** is enabled, **Enable Mailbox among functions** will be enabled by default.
- **Total Physical Functions:** A maximum of two 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, 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.

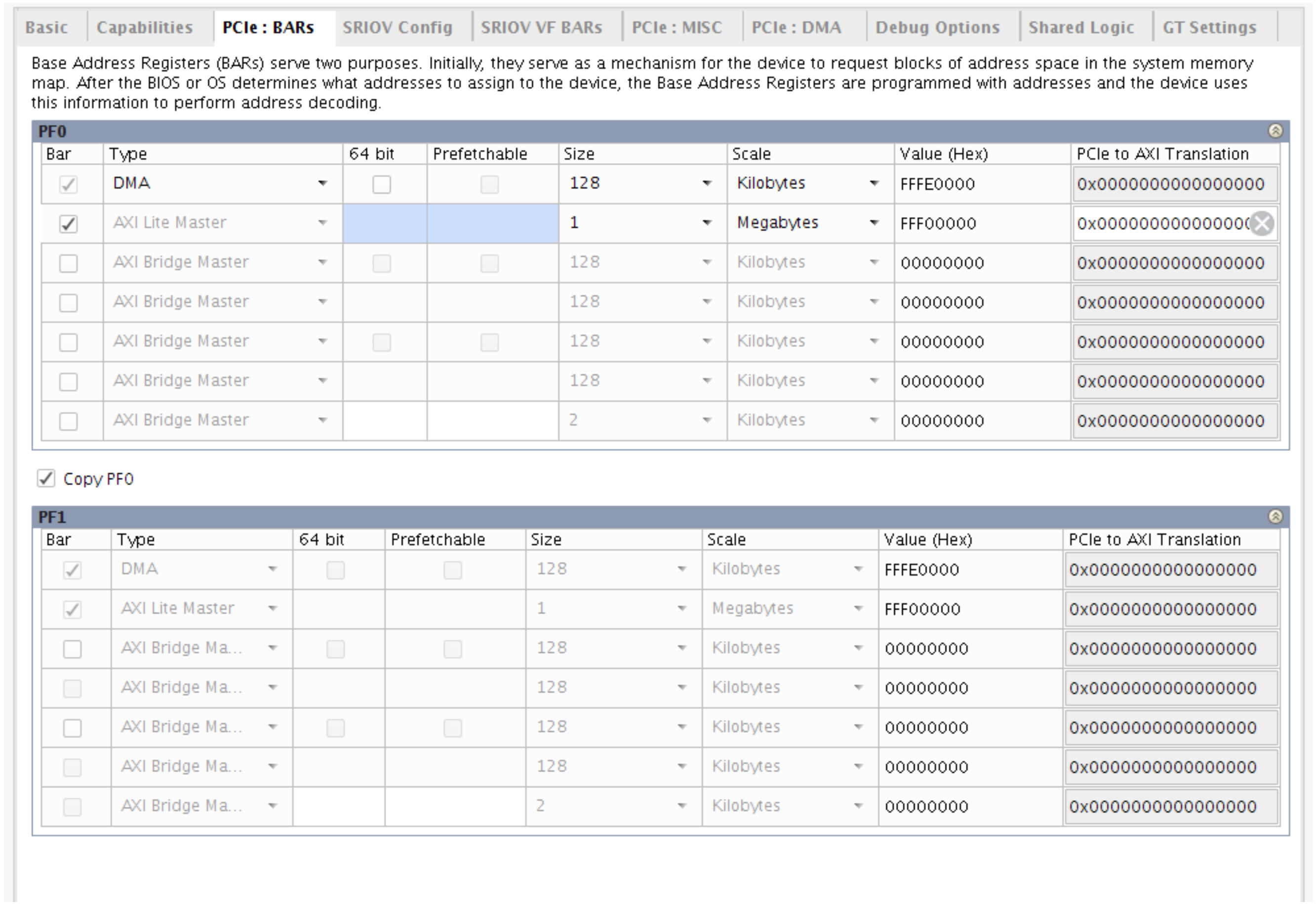
TIP: *It is always recommended that the link width, speed and Device Port type be changed first and then the Device ID value. Make sure the Device ID value is set correctly before generating the IP.*

- **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 23: **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	FFFE0000	0x0000000000000000
✓	AXI Lite Master			1	Megabytes	FFF00000	0x000000000000000
☐	AXI Bridge Master	☐	☐	128	Kilobytes	00000000	0x0000000000000000
☐	AXI Bridge Master			128	Kilobytes	00000000	0x0000000000000000
☐	AXI Bridge Master	☐	☐	128	Kilobytes	00000000	0x0000000000000000
☐	AXI Bridge Master			128	Kilobytes	00000000	0x0000000000000000
☐	AXI Bridge Master			2	Kilobytes	00000000	0x0000000000000000

✓ Copy PF0

PF1

Bar	Type	64 bit	Prefetchable	Size	Scale	Value (Hex)	PCIe to AXI Translation
✓	DMA	☐	☐	128	Kilobytes	FFFE0000	0x0000000000000000
✓	AXI Lite Master			1	Megabytes	FFF00000	0x0000000000000000
☐	AXI Bridge Ma...	☐	☐	128	Kilobytes	00000000	0x0000000000000000
☐	AXI Bridge Ma...			128	Kilobytes	00000000	0x0000000000000000
☐	AXI Bridge Ma...	☐	☐	128	Kilobytes	00000000	0x0000000000000000
☐	AXI Bridge Ma...			128	Kilobytes	00000000	0x0000000000000000
☐	AXI Bridge Ma...			2	Kilobytes	00000000	0x0000000000000000

- **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 shared 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.

- **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_TRQ_EXT (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.
- **Bypass AXI 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.
- **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.

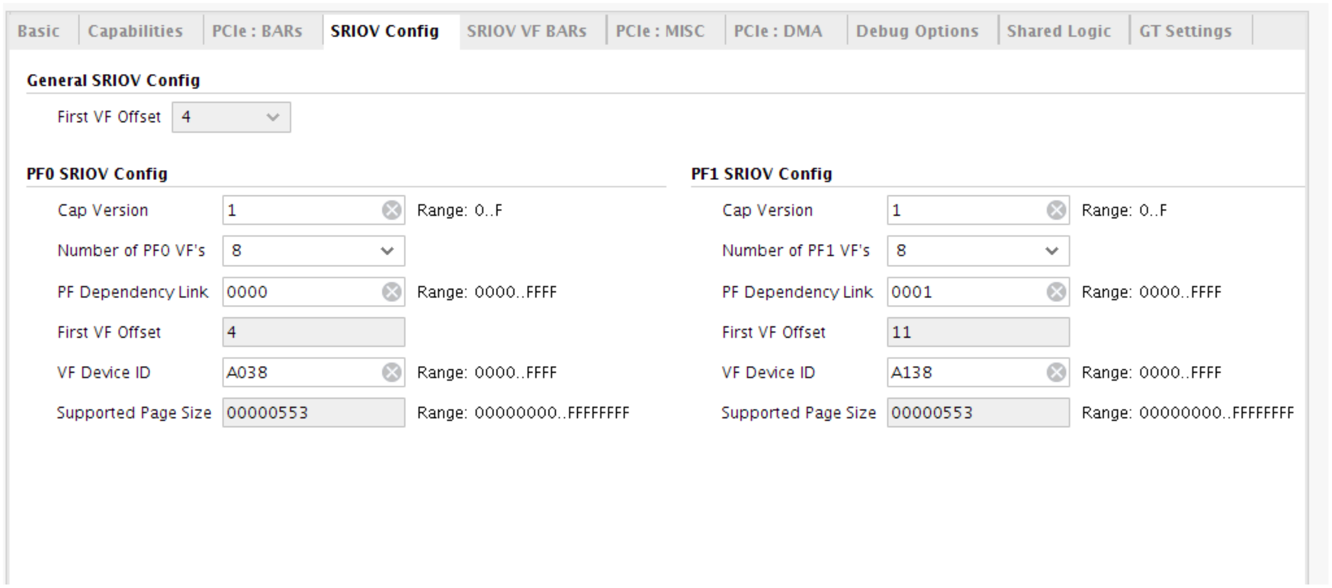
SRIOV Config Tab

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 will appear.

The SRIOV Config Tab is shown in the following figure.

Figure 24: **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:

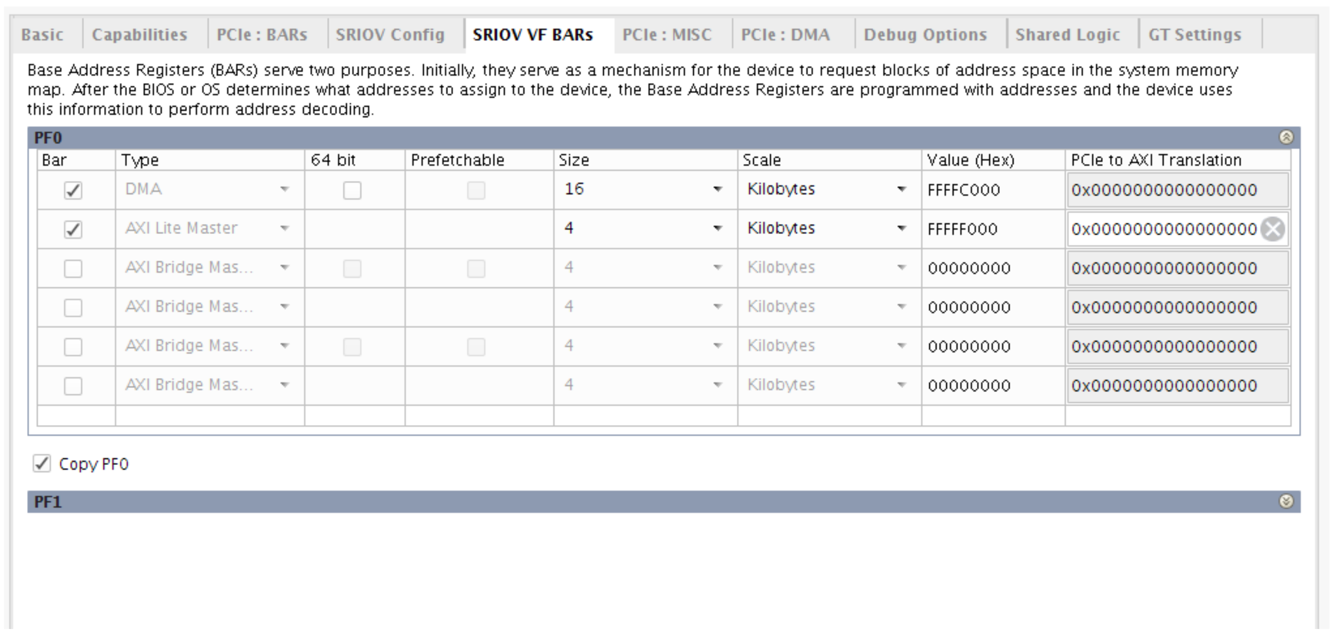
PF1_FIRST_VF_OFFSET = 63 + NUM_PF0_VFS

- **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 25: **SRIOV VF BARs Tab**



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	FFFFC000	0x0000000000000000
✓	AXI Lite Master			4	Kilobytes	FFFFF000	0x0000000000000000
☐	AXI Bridge Mas...	☐	☐	4	Kilobytes	00000000	0x0000000000000000
☐	AXI Bridge Mas...			4	Kilobytes	00000000	0x0000000000000000
☐	AXI Bridge Mas...	☐	☐	4	Kilobytes	00000000	0x0000000000000000
☐	AXI Bridge Mas...			4	Kilobytes	00000000	0x0000000000000000

✓ Copy PF0

PF1

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 Master Master:** Is fixed to BAR2 space.For all other bars user have option to select 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 AXI4-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.

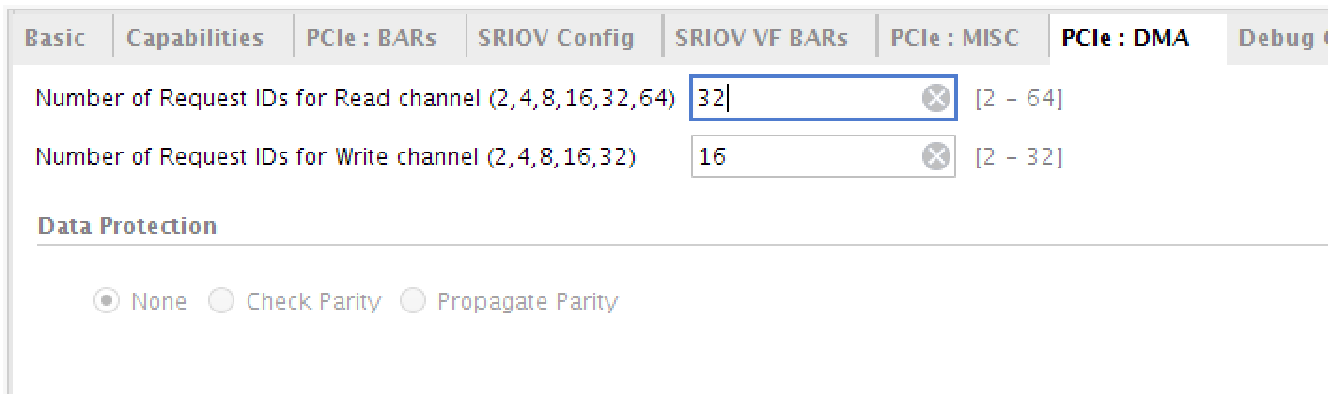
- **64-bit:**

  VF BARs can be either 64-bit or 32-bit:

  - 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

Figure 26: **PCIe MISC 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]

Data Protection

[✓ None] Check Parity Propagate Parity

- **MSI-X Capabilities:** MSI-X is enabled as default.The MSI-X settings for different physical functions can be set as required.
- **MSIx Table Settings:** Defines the MSI-X Table Structure.

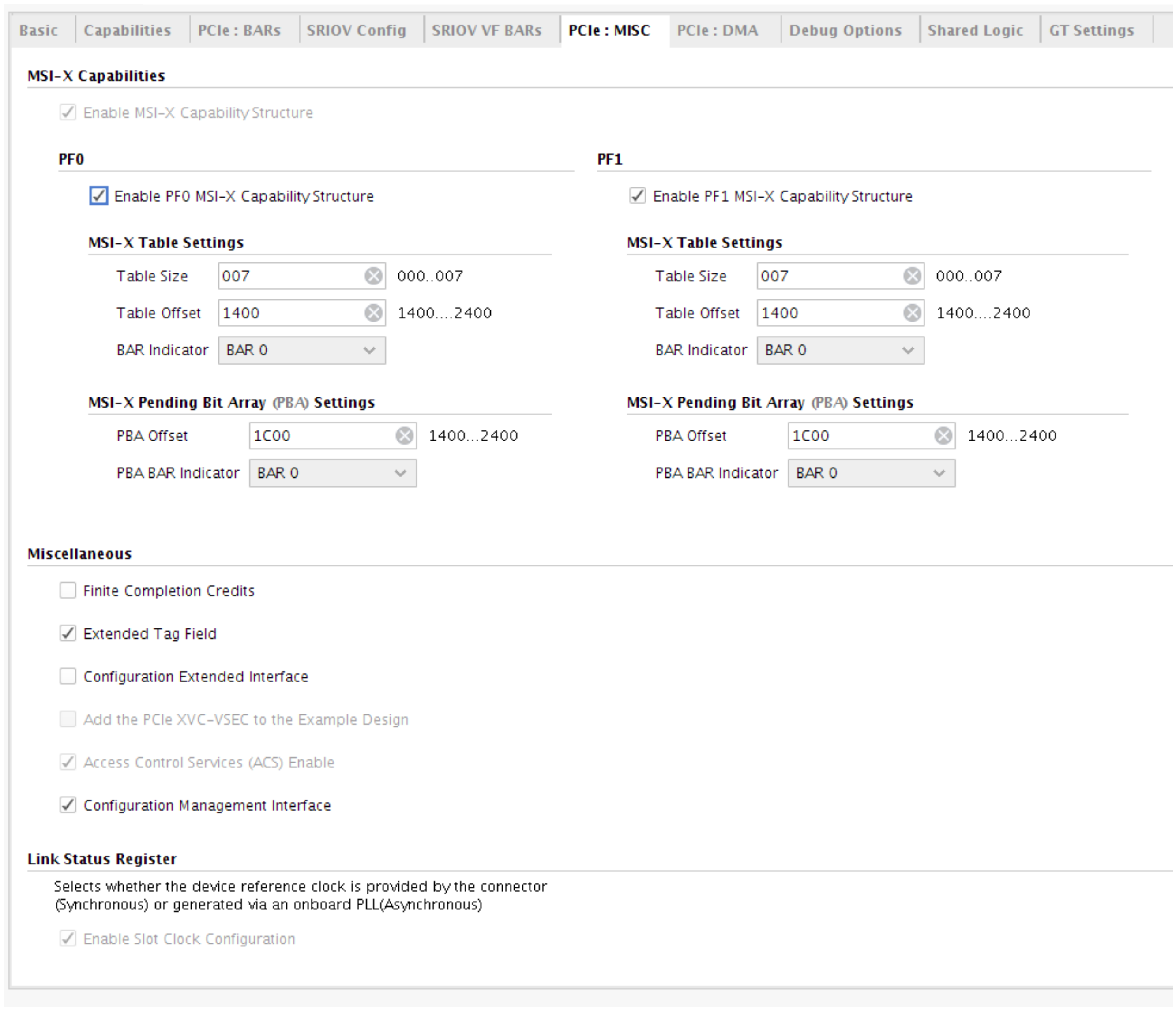
  - **Table Size:** Specifies the MSI-X Table size. The default is 8 (8 interrupt vectors per function).
  - **Table Offset:** Specifies the offset from the Base addres Register (BAR)in DMA configiration space used to map function in MSI-X Table onto memory space. Table space should be between 0x1400 to 0x2400.
  - **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 should be between 0x1400 to 0x2400.
  - **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:** 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:** 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 27: **PCIe DMA 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	1400	1400....2400
BAR Indicator	BAR 0	

MSI-X Pending Bit Array (PBA) Settings

PBA Offset	1C00	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	1400	1400....2400
BAR Indicator	BAR 0	

MSI-X Pending Bit Array (PBA) Settings

PBA Offset	1C00	1400...2400
PBA BAR Indicator	BAR 0	

Miscellaneous

- ☐ Finite Completion Credits
- ✓ 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

- **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.

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 164: **Common In/Out Commands and Endpoint PIPE Signals Mappings**

In Commands	Endpoint PIPE Signals Mapping	Out Commands	Endpoint PIPE Signals Mapping
common_commands_in[25:0]	not used	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
		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 165: **Input/Output Bus with Endpoint PIPE Signals Mapping**

Input Bus	Endpoint PIPE Signals Mapping	Output Bus	Endpoint PIPE Signals Mapping
pipe_rx_0_sigs[31:0]	pipe_rx0_data_gt	pipe_tx_0_sigs[31: 0]	pipe_tx0_data_gt
pipe_rx_0_sigs[33:32]	pipe_rx0_char_is_k_gt	pipe_tx_0_sigs[33:32]	pipe_tx0_char_is_k_gt
pipe_rx_0_sigs[34]	pipe_rx0_elec_idle_gt	pipe_tx_0_sigs[34]	pipe_tx0_elec_idle_gt
pipe_rx_0_sigs[35]	pipe_rx0_data_valid_gt	pipe_tx_0_sigs[35]	pipe_tx0_data_valid_gt
pipe_rx_0_sigs[36]	pipe_rx0_start_block_gt	pipe_tx_0_sigs[36]	pipe_tx0_start_block_gt
pipe_rx_0_sigs[38:37]	pipe_rx0_syncheader_gt	pipe_tx_0_sigs[38:37]	pipe_tx0_syncheader_gt
pipe_rx_0_sigs[83:39]	not used	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

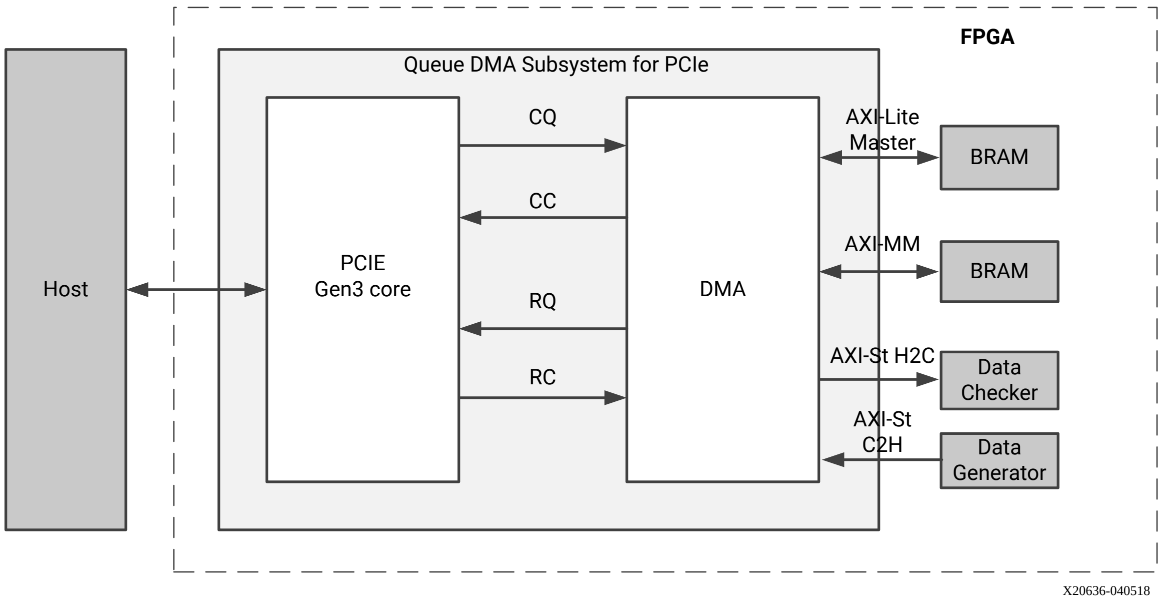
For details about synthesis and implementation, see the *Vivado Design Suite User Guide: Designing with IP* (UG896).

Chapter 6

Example Design

This chapter contains information about the AXI Memory Mapped and AXI Stream default example design provided in the Vivado Design Suite.

Figure 28: **Subsystem Example Design**



AXI4 Memory Mapped and AXI Stream 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 KB 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 below. Theese 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.

Table 166: **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
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

C2H_ST_QID (0x000)

Table 167: **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 168: **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 169: **C2H_CONTROL_REG (0x008)**

Bit	Default	Access Type	Field	Description
[31:2]	0	NA		Reserved
[1]	0	RW		Start AXI-St C2H transfer
[0]	0	NA		Reserved

H2C_CONTROL_REG (0x00C)

Table 170: **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 171: **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 172: **C2H_PACKET_COUNT (0x020)**

Bit	Default	Access Type	Field	Description
[31:10]	0	NA		Reserved
[9:0]	0	RW		AIX-St C2H number of packet to transfer

C2H_COMPLETION_DATA_0 (0x030)

Table 173: **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 174: **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 175: **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 176: **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 177: **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 178: **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 179: **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 180: **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 181: **C2H_COMPLETION_SIZE (0x050)**

Bit	Default	Access Type	Field	Description
[31:0]	0	NA		Reserved
[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 182: **SCRATCH_REG0 (0x060)**

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

SCRATCH_REG1 (0x064)

Table 183: **SCRATCH_REG1 (0x064)**

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

C2H_PACKETS_DROP (0x088)

Table 184: **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 185: **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.

Appendix A

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 186: **Subsystem Comparison**

	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 coalescing 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 (in 2018.1), Windows and DPDK (in a future release)

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

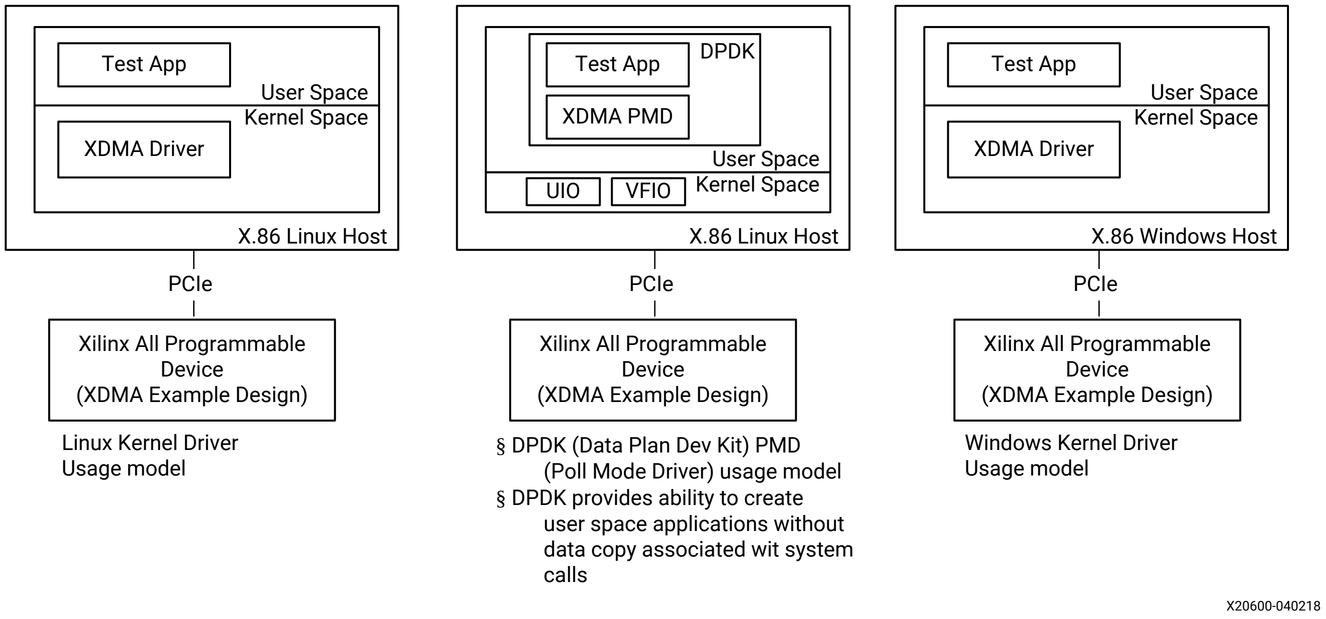
Ensure that all the timing constraints for the core were properly incorporated from the example design and that all constraints were met during implementation.

- 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 29: **Device Drivers**

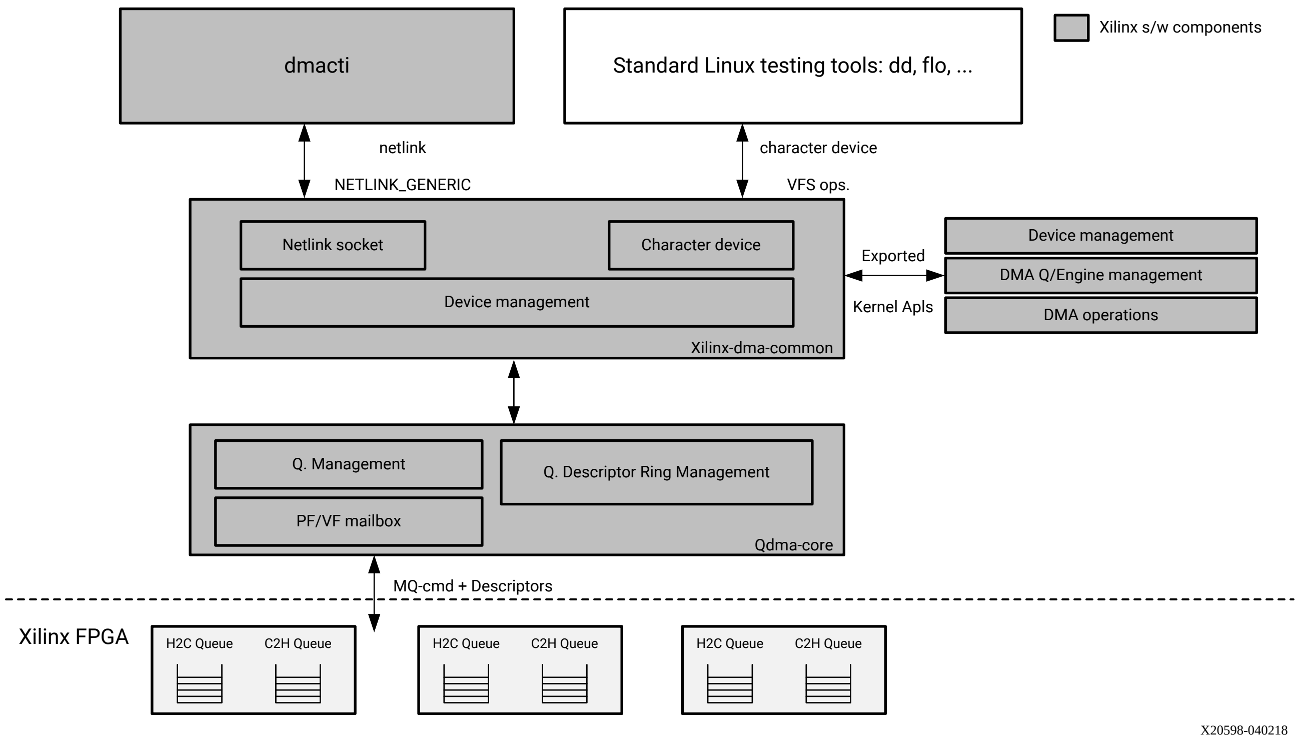

Test App
User Space
Kernel Space
XDMA Driver
X.86 Linux Host
PCIe
Xilinx All Programmable Device (XDMA Example Design)
Linux Kernel Driver Usage model
Test App
DPDK
XDMA PMD
User Space
UIO
VFIO
Kernel Space
X.86 Linux Host
PCIe
Xilinx All Programmable Device (XDMA Example Design)
§ DPDK (Data Plan Dev Kit) PMD (Poll Mode Driver) usage model
§ DPDK provides ability to create user space applications without data copy associated wit system calls
Test App
User Space
Kernel Space
XDMA Driver
X.86 Windows Host
PCIe
Xilinx All Programmable Device (XDMA Example Design)
Windows Kernel Driver Usage model
X20600-040218

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, we utilize Data Plan Dev Kit (DPDK) 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 30: **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
```

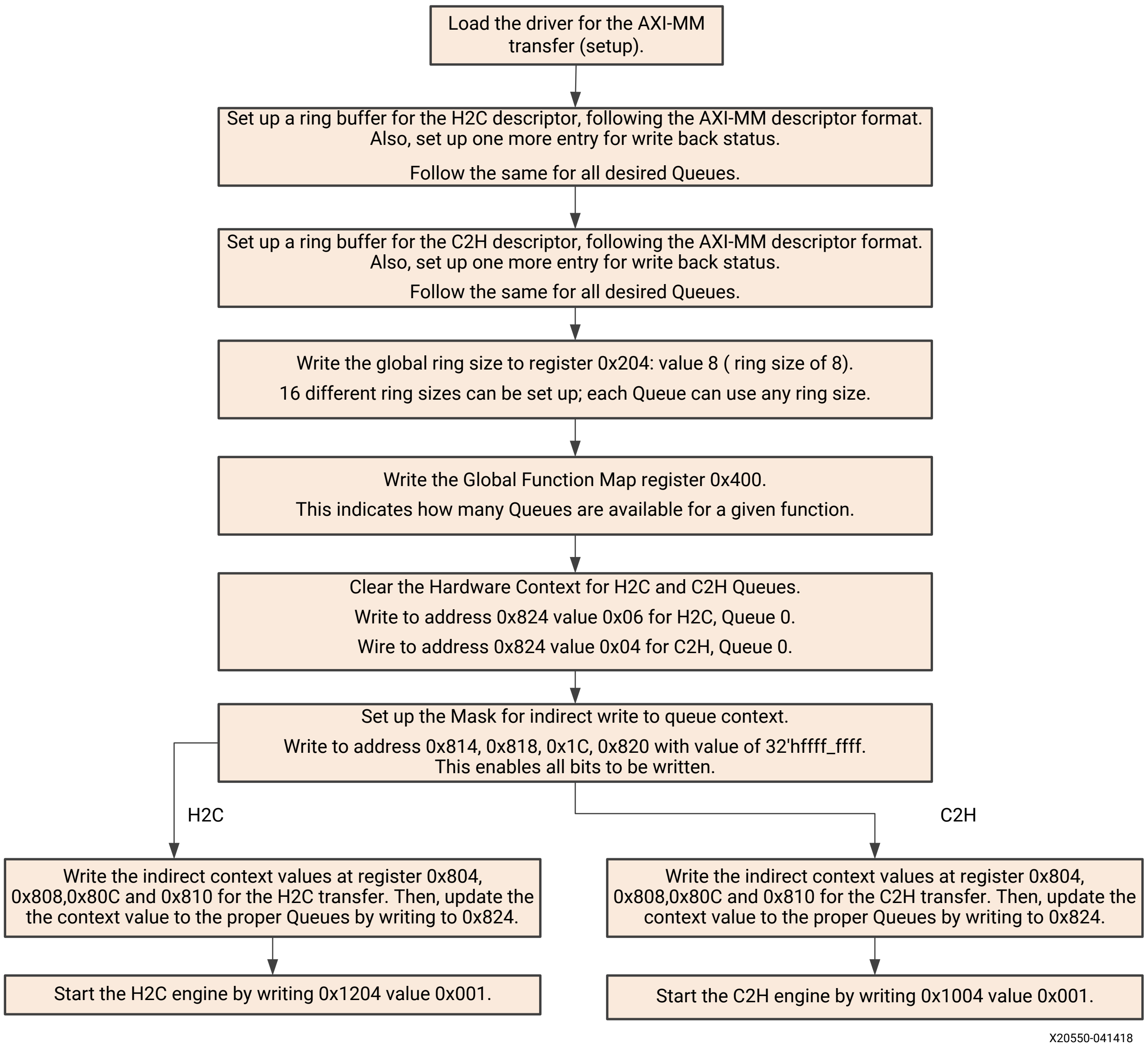
It releases the resources, which is allocated on the host.

> <*N*> is the QDMA function number obtained from *"dmactl dev list"*.
>
> <*id*> is the queue index.

Reference Software Driver Flow

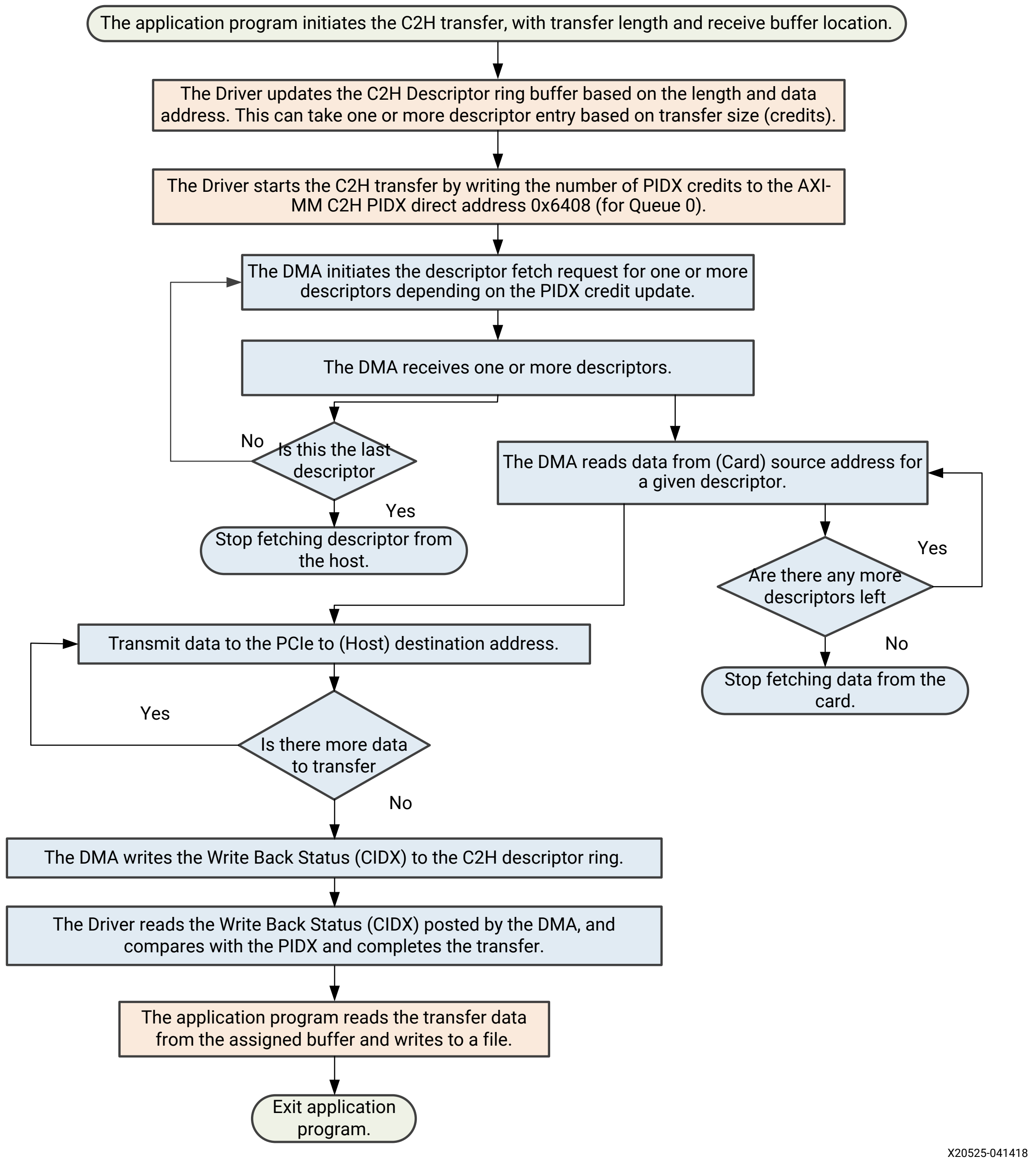
AXI4-Memory Map Flow Chart

Figure 31: **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

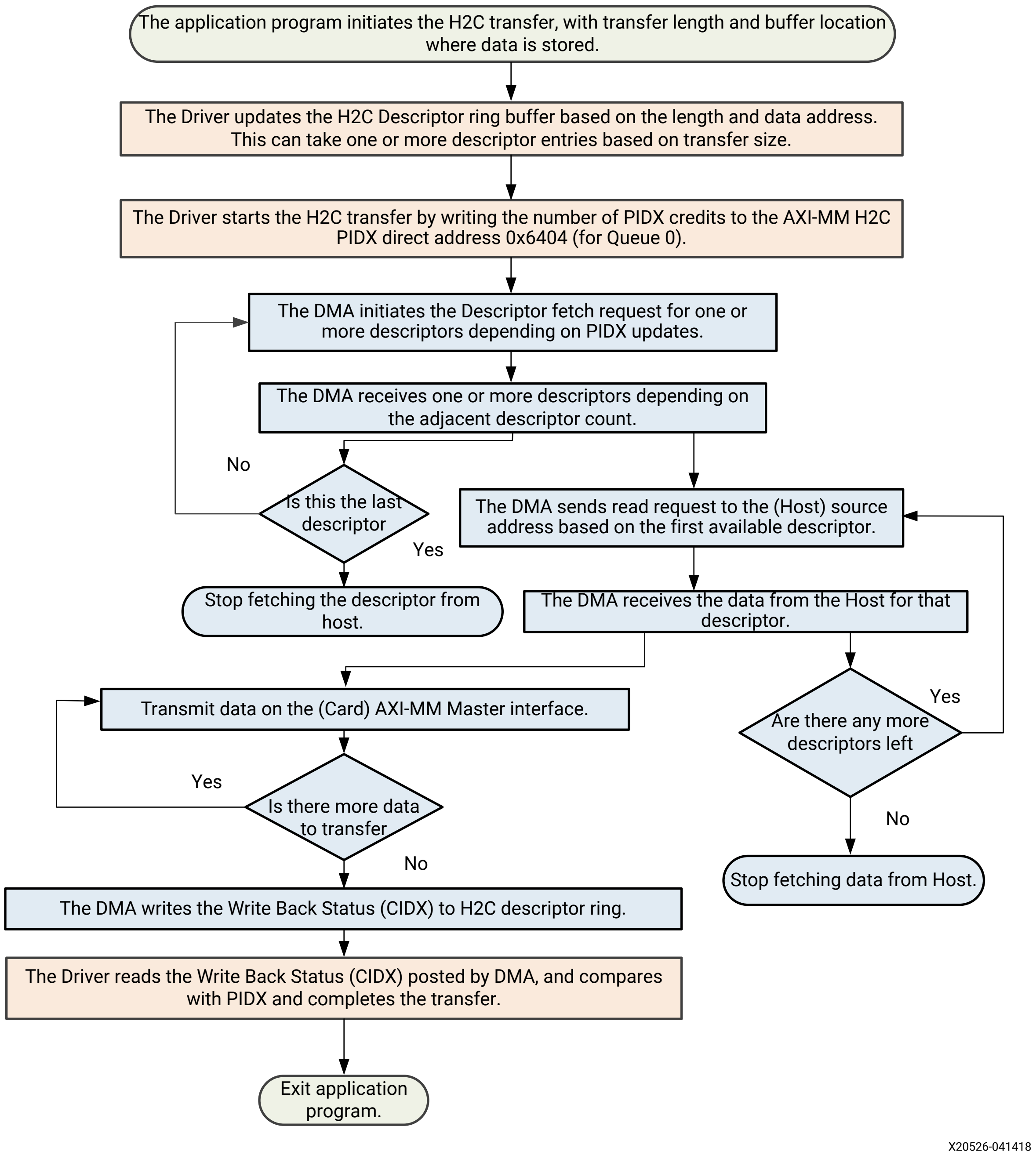
AXI4 Memory Mapped C2H Flow

Figure 32: **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

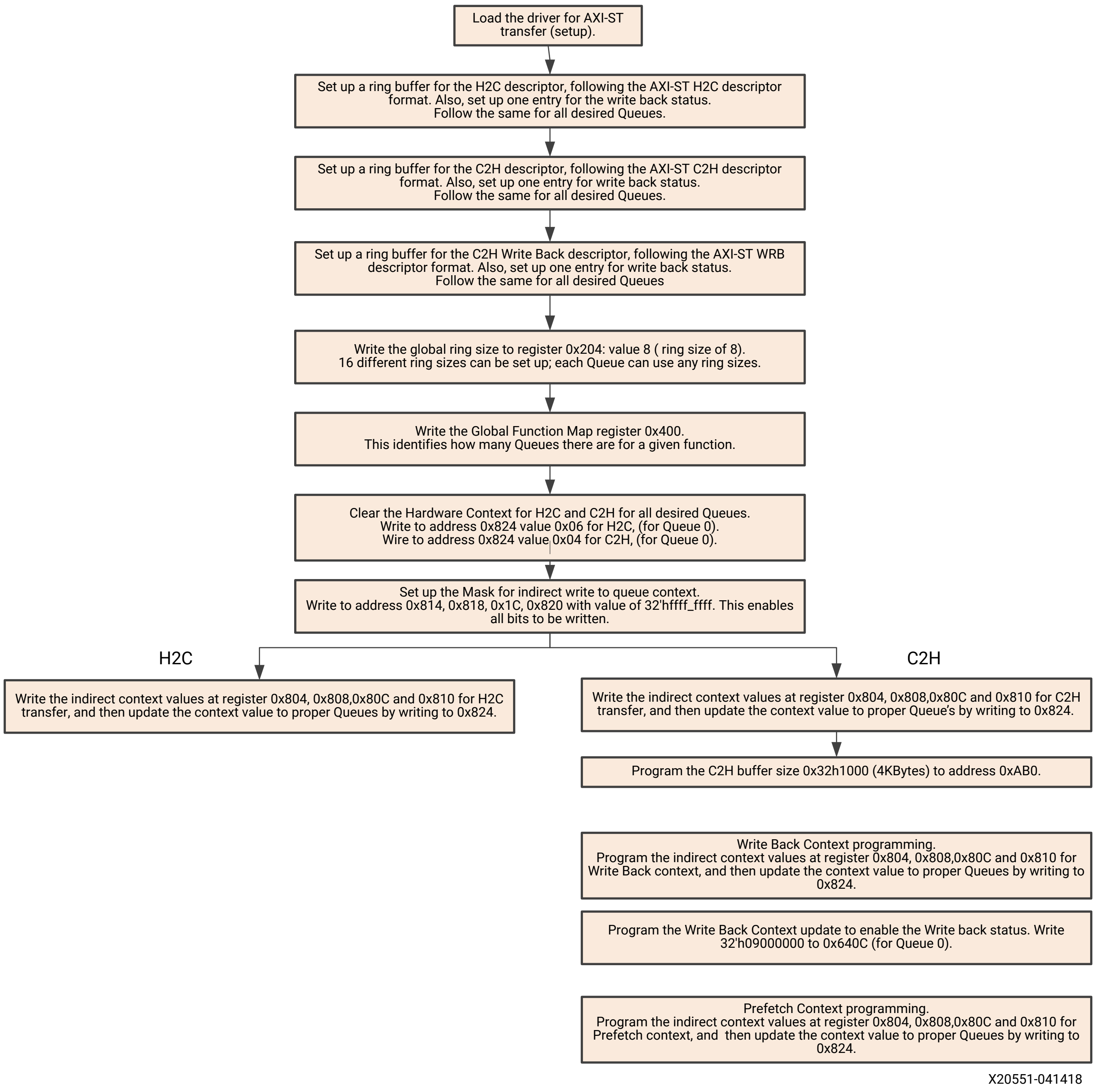
AXI4 Memory Mapped H2C Flow

Figure 33: **AXI4 Memory Mapped H2C Flow Diagram**


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
Stop fetching the descriptor from host.
The DMA sends read request to the (Host) source address based on the first available descriptor.
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

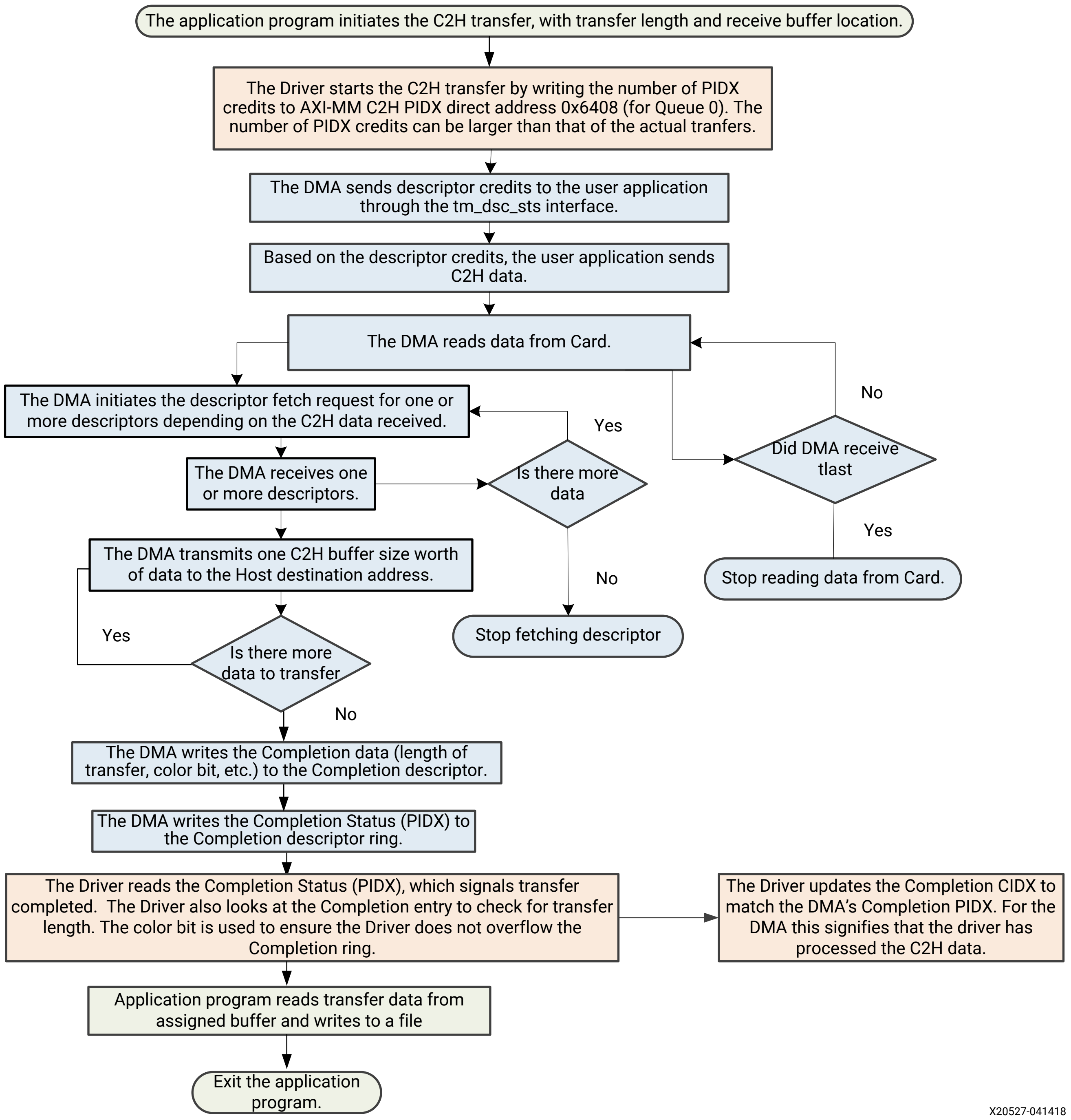
AXI4-Stream Flow Chart

Figure 34: **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

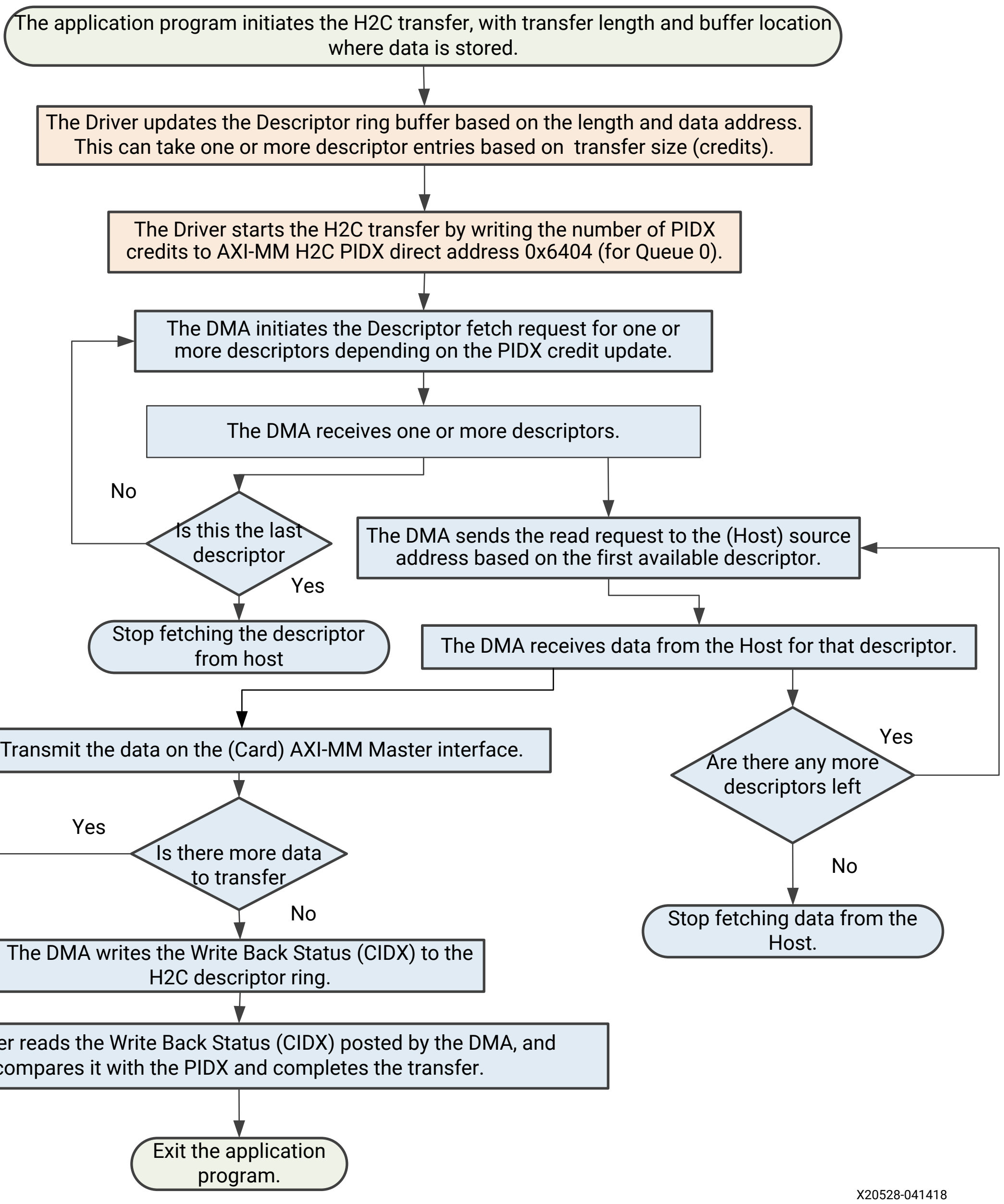
AXI4-Stream C2H Flow

Figure 35: **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 36: **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.
Yes
Are there any more descriptors left
No
Stop fetching data from the Host.
Transmit the data on the (Card) AXI-MM Master interface.
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. ARM 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)

Training Resources

1. Vivado Design Suite Hands-on Introductory Workshop
2. Vivado Design Suite Tool Flow

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
04/17/2018 v1.0	
Initial Xilinx release.	

Please Read: Important Legal Notices

AUTOMOTIVE APPLICATIONS DISCLAIMER

Copyright