1.EFC-2 Data Network

The data networks provide the connectivity for data transfer between the external DRAM, Buffer RAM and the various functional units and processors. There are two data networks in the EFC-2, one each for he DRAM and the Buffer RAM. This is a scalable balanced tree network, with independent arbitration at each network node.
The packet types supported by the data network are enumerated in the FSD.

1.1.Tree Network

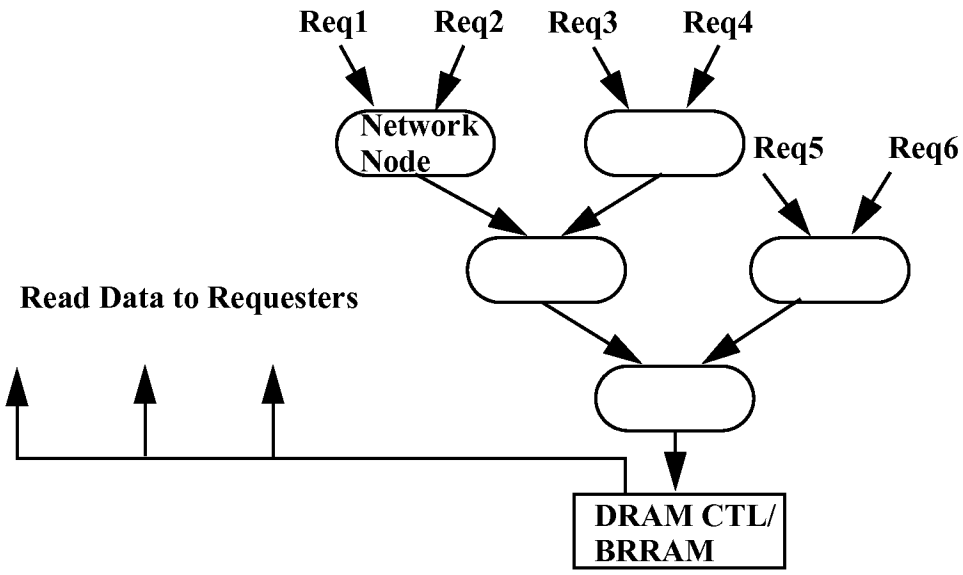


Fig1: Data Network

1.2.Data Network Node

The data network node is a two input data path element with a simple round robin arbiter to select between the inputs. The data inputs to the node are packets whose lengths vary from 64 to 256 bytes, in multiples of 64 bytes. The data path width is 256 bits with a valid and a start of frame indication.

Arbiter: The arbiter chooses between the two available inputs in a pure round robin fashion, switching inputs at packet boundaries. The packet length is not considered for arbitration. The functional units split up a data transfer message into packets of the biggest size - 256 bytes - while requesting a transfer. So, a simple round robin will be fair for overwhelming majority of the cases, except when there is a countinuous transfer of varing size packets, which is rare.

Polaris EX2015
Sandisk v. Polaris
IPR2025-00517
0001

The arbiter chooses a source based on the availability of data, the current source being serviced. It asserts XOFF based othese factors. Since the nodes are expected to be dispersed throughout the device, all signals in and out of a node are registered, due to which the arbiter makes a decision to switch sources a clock before the current packet terminates.

Flow Control: The node can receive flow control at any point from the downstream node and it is expected to honor it at the same clock. In other words the flow control acts as a pull interface. If "XOFF" is low for a given clock and there is data available, it is transferred out of the node. Similarly, the node generates an indpendant flow control to the up stream nodes. Thers is a small 2 deep FIFO in the node input to facilitate that. There are two XOFF's, one for each upstream node. These are asserted based on the source selected and available space in the input FIFO.

Pipeline Control: There is an ouput staging register in each node. This is to facilitate a truly distributed network, not constrained by physical placements. The node accepts data if the input FIFO is not full. If the FIFO has data and the output staging register is empty or the downstream node is ready to accept, data is transferred from the input FIFO to this register. The Valid flag in the staging register is the valid out for the node.

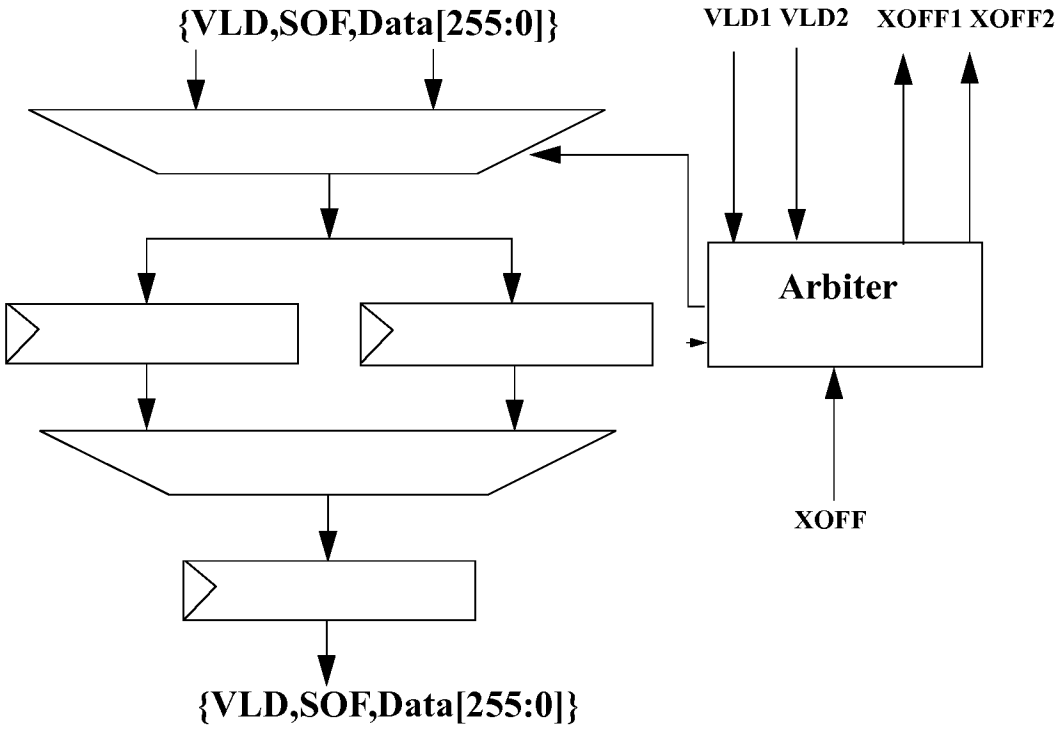


Fig2: Data Network Node

The data transfer from the destination to the source is through a bus. The source nodes have separate ports for accepting data from the DRAM and the Buffer RAM.

1.3.Network Interface

Table 1: Data Network Requester Write Interface

Signal Name	Direction	Description
Data [255:0]	OUT	Write data into the network.
Valid	OUT	Indicates valid data on the data bus
SOF	OUT	Start of Frame.
XOFF	IN	Pulls data from the requester when low.

The write interface of the functional modules to the data network shown in Table1.

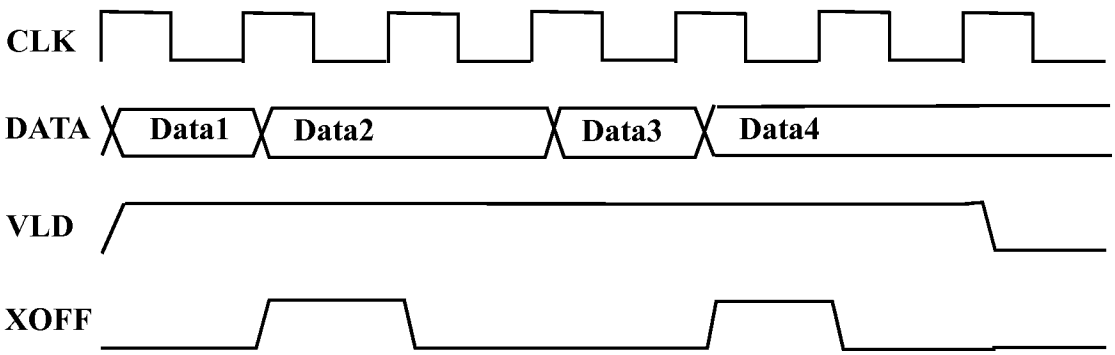


Fig3: Data Network Write Timing

Table 2: Data Network Requester Read Interface

Signal Name	Direction	Description
Data [255:0]	OUT	Write data into the network.
Valid	OUT	Indicates valid data on the data bus
SOF	OUT	Start of Frame.

Delaware NVMHCI DMA

1.1 Overview

The NVMHCI unit facilitates DMA of NVMHCI commands from the Host(s) to the BRAM/DRAM. Delaware supports 16 hosts, with 4 queues each for submission and completion status, per host. The arbitration between hosts is pure round robin. Within a host, the queues are selected through a weighted round robin algorithm, with the queues being allocated different credits.

Assumptions for Delaware
The NVMHCI hardware in Delaware does not parse the commands. This is left to the processor servicing the submission message. Each submission command is assumed to be 64 bytes long. The queue structure is assumed to be a contiguous circular memory buffer, aligned to a 4kB boundary.
Completion commands are assumed to be 16 bytes long.

1.1.1 Queue Structure

The NVMHCI queue structure for the command submission and completion queues, are illustrated in the following figure. There are 4 queues per port(host). The submission queues have fixed length messages of 64 bytes each. The queue base addresses are aligned to a 4KB boundary. Each submission queue can hold 64 messages.

The submission queues have credits allocated to them, so that they can be serviced in a weighted round robin manner. The units of the credits are messages. In other words, one credit corresponds to one message. ‘

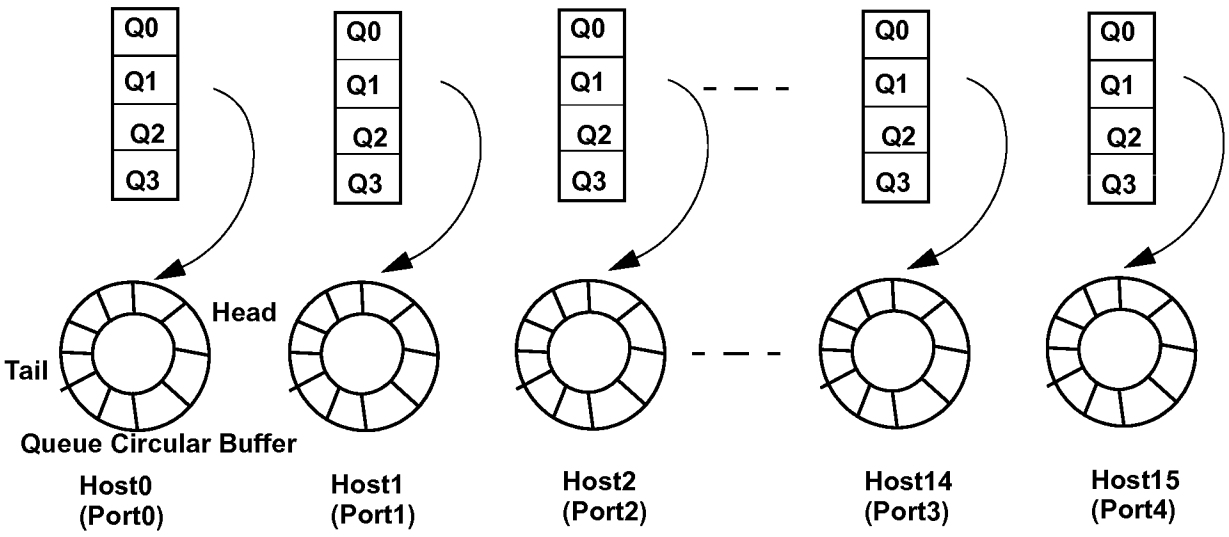


Figure 1.1 NVMHCI Queue Structures

The completion queue messages are 16 bytes each. The completion queue structure is similar to the submission queue. The processor in Delaware, which processes the submission commands determines the completion queue to post the completion message to and indicates that in the "status queue update" message.

1.1.2 Submission Queue Requests

The base address for each of the 64 submission queues is maintained in the respective submission queue base address registers by the host. The host maintains the submission queue tail pointer. The tail pointer points to the last message in the queue. The submission queue head pointer is maintained by the controller.

Since the queue sizes are fixed at 4kbytes, and the base addresses begin at a 4kB boundary, the queue pointers are only 6 bits wide. This is illustrated below.

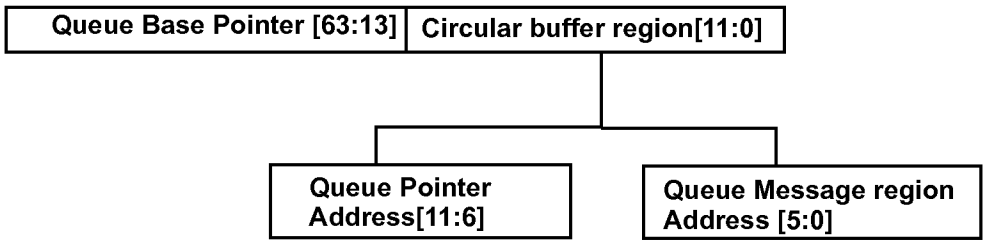


Figure 1.2 Queue Addressing

The queue selection algorithm is described in a subsequent section. The head pointer is incremented by one for each message read from the queue by the controller. The queue is empty when the head pointer equals the tail pointer. The queue is full when the head pointer is ahead of the tail pointer by one.

The queue selector selects the next eligible queue and passes the pointer to the next address to be read, to the PCIe request generator. The address is generated by concatenating the queue base address[63:12] with the tail pointer. Since, all the messages are 64 bytes, the LSB[5:0] bits are 0's.

A read request is generated only when the PCIe completion can accommodated in the completion FIFO. In response to status queue update message from the controller processors, PCIe write requests to the completion queue are issued. The completion write is posted to the head of the requested completion queue.

The PCIe completion FIFO is 4kB deep. Hence it can store 64 submission commands. As long as a completion for a given read request can be accommodated in the completion FIFO, a read request can be issued. The request generator keeps track of the empty space in the completion FIFO as well as the number of requests in flight.

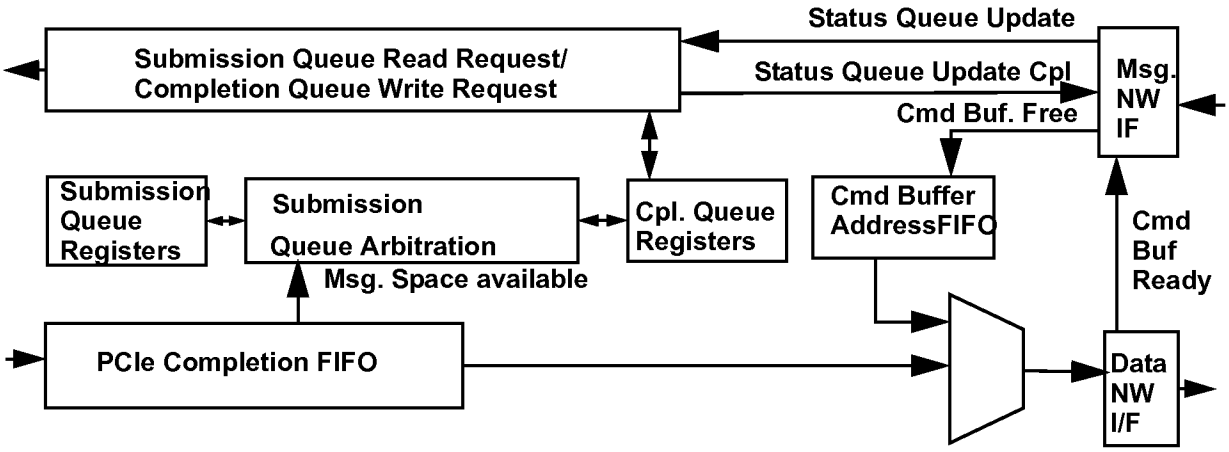


Figure 1.3 NVMHCI Module - Main Blocks

1.1.3 Completions from the Host

The completions from the host, in response to the submission queue read requests, are gathered in the completion FIFO, for transfer to the DRAM or BRAM. If a command buffer address is available, the command is send as a data network packet and written into the DRAM or BRAM, with request for a completion.

Command ready message is issued to the next available processor, once a completion for the Data Network Write has been received.

1.1.4 Queue Arbitration

The queue arbitration follows a two stage process. At first, the next available host is selected. If any host has active non empty queues, they are considered for the arbitration.

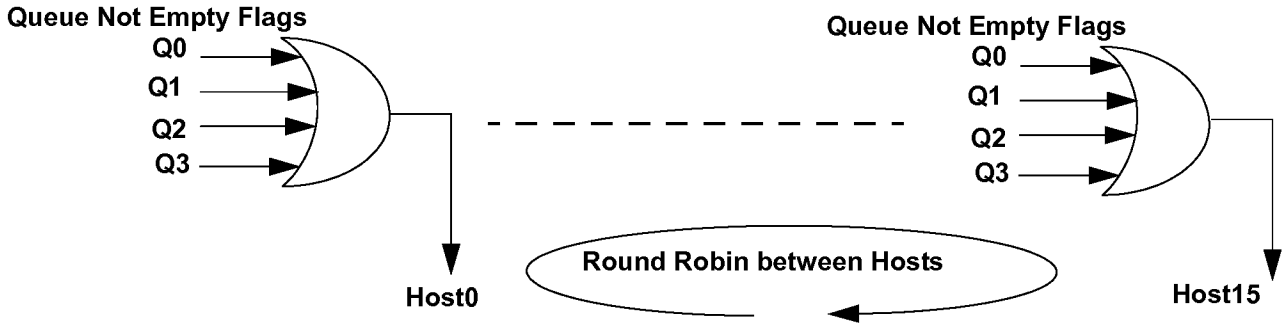


Figure 1.4 Host Selection

Once, a host has been selected, the next eligible queue has to selected, out of four queues. This is done with a weighted round robin arbiter. At the beginning of each round, the credits for each queue are loaded with the maximum value programmed by the host. At the start of the round, starting from Q0, the first non-empty queue with non-zero credit is selected. The head pointer for the queue is passed on to the request generator.

The queue length is calculated and the empty flag for the queue is updated, if the queue is empty. On each update to the submission queue tail pointer, the empty flag for the respective queue is set. The flag is reset, if the queue is empty on a dequeue from the submission queue. (update to the head pointer). So, while on a enqueue, the tail pointer is written, on a dequeue, the tail pointer has to be read and the queue length calculated, as well the head pointer updated with the next queue message location.

Once a queue has been serviced, the next queue is selected based on the same criteria, in a round robin manner. This continues till the credits drop to zero or one or more queues go empty. If either case is true for all the queues in a host, the weighted round robin round ends. The next round is starts with the current credits being updated to the maximum value, and the process repeats.

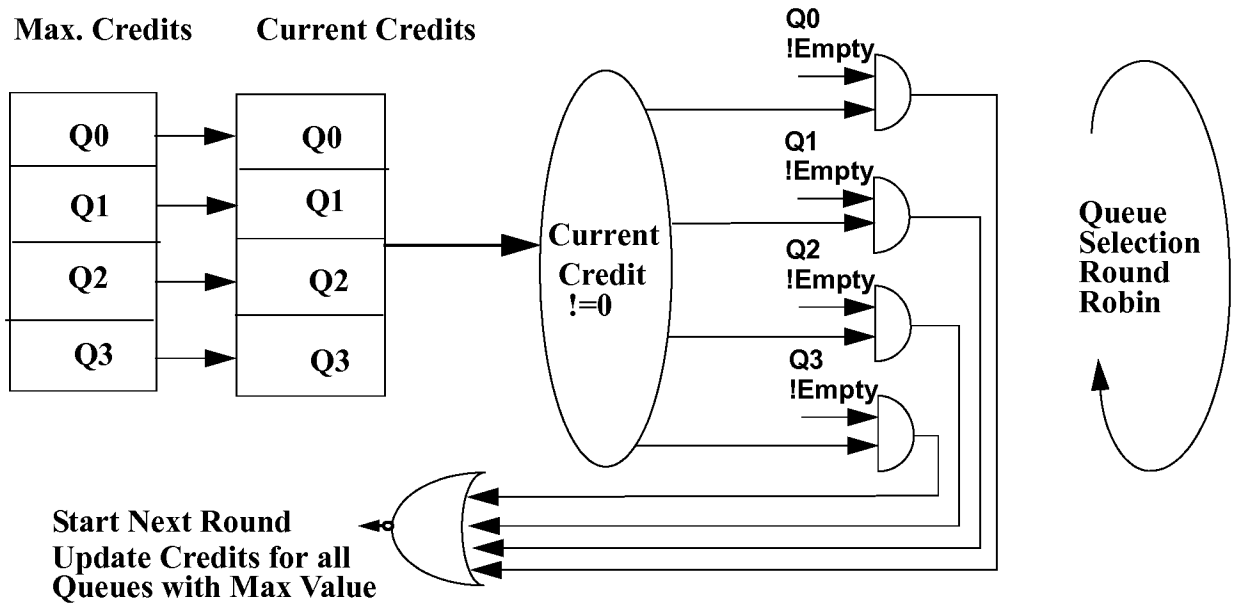


Figure 1.5 Queue Selection

-

1. Message Network

The message networks provide the connectivity for message transfer between the various functional units and processors. This is a tile network, with independent arbitration at each network node.
The packet types supported by the message network are enumerated in the FSD.

1.1.Message Network Node

The message network node is a five input data path element. Each node can tie to a functional unit or to the adjacent nodes in E,W,N or S. The data inputs to the node are packets whose lengths are 64 bits.

Arbiter: The node can accept or transmit data in E,W,N,S or to the functional unit or processor attached to it. Hence there is an arbiter for each of the five outputs. This is a simple round robin arbiter. Since the message network packet is one clock, the arbitration changes every clock, based on the data availability at each of the inputs.
The arbiter chooses a source based on the availability of data and the current source being serviced.

Flow Control: The node can receive flow control at any point from the adjacent nodes or functional unit and it is expected to honor it at the same clock. In other words the flow control acts as a pull interface. If "XOFF" is low for a given clock and there is data available, it is transferred out of the node, else the data is held in the output staging register for the port.
Similarly, on the input side, the node generates an independent flow control to the adjacent nodes and the functional unit. There is a small 2 deep FIFO in the node input to facilitate that and to ensure zero gaps between data transfers.

Pipeline Control: There is an output staging register in the node, per output. This is to facilitate a truly distributed network, not constrained by physical placements. The node accepts data if the input FIFO is not full. If the FIFO has data and the output staging register is empty data is transferred from the input FIFO to this register. The Valid flag in the staging register is the valid out for the node. The flow control from the adjacent nodes, help to pull data out of the staging register.

Route to Self: The message network node supports "route to self" feature for the processor complex attached to it. This means that the messages will be looped back to the source.

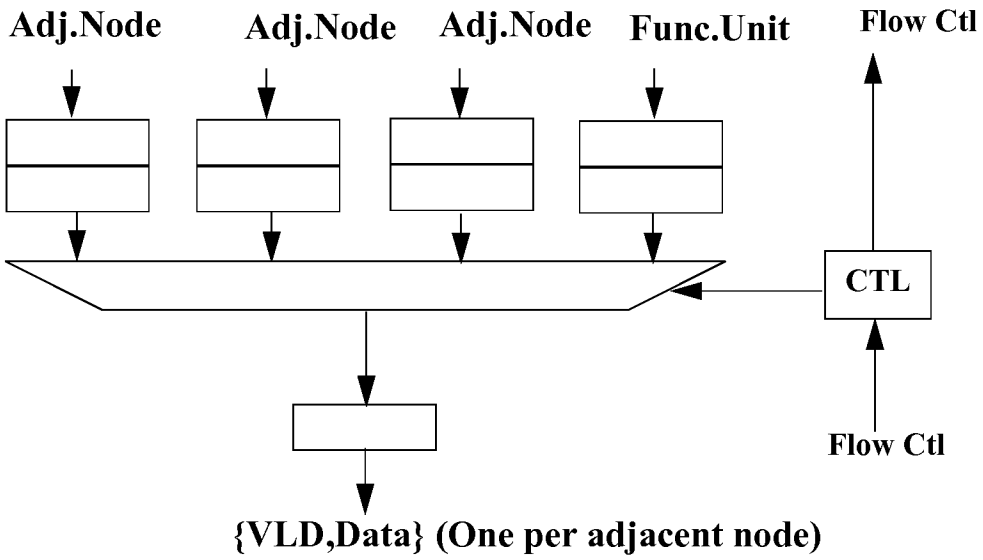


Fig2: Message Network Node

1.2.Network Interface

The following tables specify the interface to the message network as seen by a functional block or a processor complex.

Table 1: Message Network Input from the functional unit to the network

Signal Name	Direction (w.r.t func.blk)	Description
Data [‘MSG_NW_WIDTH-1:0]	OUT	Write data into the network from the functional unit.
Valid	OUT	Indicates valid data on the output data bus.
XOFF OUT	OUT	Flow Control from the functional unit to the network.

Table 2: Message Network output to the functional unit

Signal Name	Direction (w.r.t func.blk)	Description
Data [‘MSG_NW_WIDTH-1:0]	IN	Write data into the functional unit from the network
Valid	IN	Indicates valid data on the input data bus.
XOFF IN	IN	Flow Control from the network to the functional unit.

The Flow control timing for the message network is illustrated below.

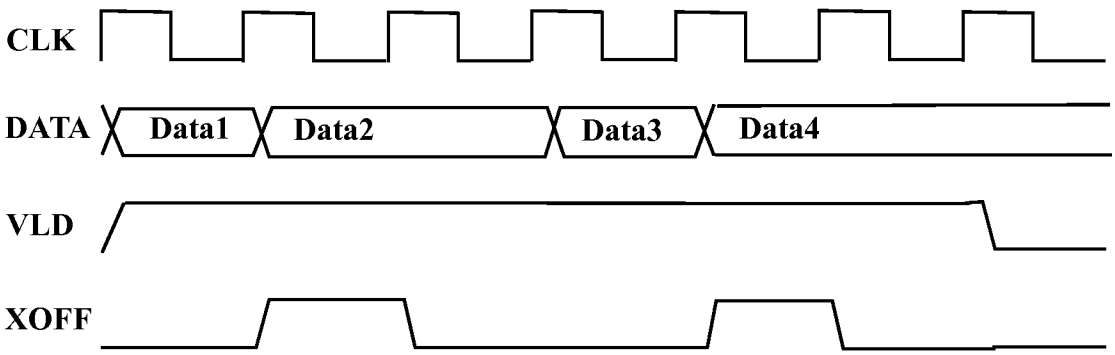


Fig3: Message Network Flow Control

Architecture

Delaware is the first device ever to publicly demonstrate a working implementation of enterprise NVMHCI. It is a processor based architecture for processing the eNVMHCI commands and managing the flash storage. It has hardware based blocks to control the host, memory and flash interfaces to achieve the needed performance levels and a scalable multi-processor architecture to get the required firmware performance. It has an interface transparent shared hardware and software architecture. It has a software partitioning and architecture and implementation that allows for scaling based on the resources and requirements.

The functional operation is described in delaware_hw_spec.pdf

The firmware operation is described in delaware_firmware-architecture.pdf

The enterprise NVMHCI architecture and implementation in Delaware is designed to achieve maximum throughput of submission and completion queues. It interfaces to the scalable data and message networks and provides the hardware operation needed to offload the command processing from the host and local processors.

The functional operation is described in delaware_hw_spec.pdf

The preferred embodiment is described in delaware_nvmhci_dma.pdf

The message network is a hop to hop routed message passing network architected and designed to provide message interconnect between on chip processors and functional blocks. It is a non-blocking network with directional routing and allows for flexible and scalable interconnect.

The functional operation is described in delaware_hw_spec.pdf

The preferred embodiment is described in message_network_node.pdf

The inter processor communication (IPC) protocol is the messaging protocol running on top of the message network. It the form of outbound, inbound and free queues it provide the means for processor to processor communication and processor to functional units communication. It also provides for configurable and non-blocking message passing.

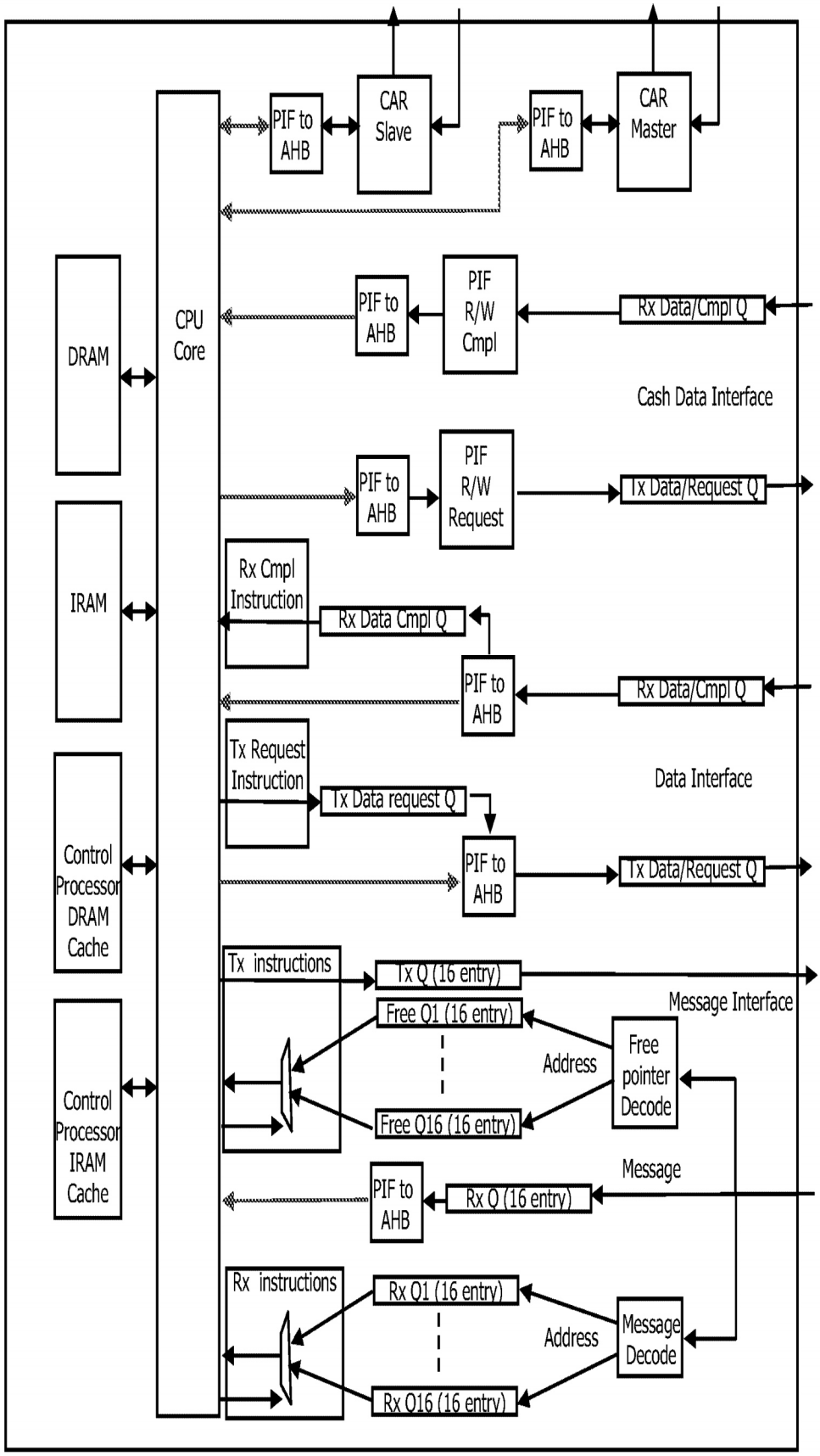
The functional operation is described in delaware_hw_spec.pdf

The Data network is a scalable data network architected and designed to provide a data interconnect between on chip processors and functional blocks.

The functional operation is described in delaware_hw_spec.pdf

The preferred embodiment is described in data_network.pdf

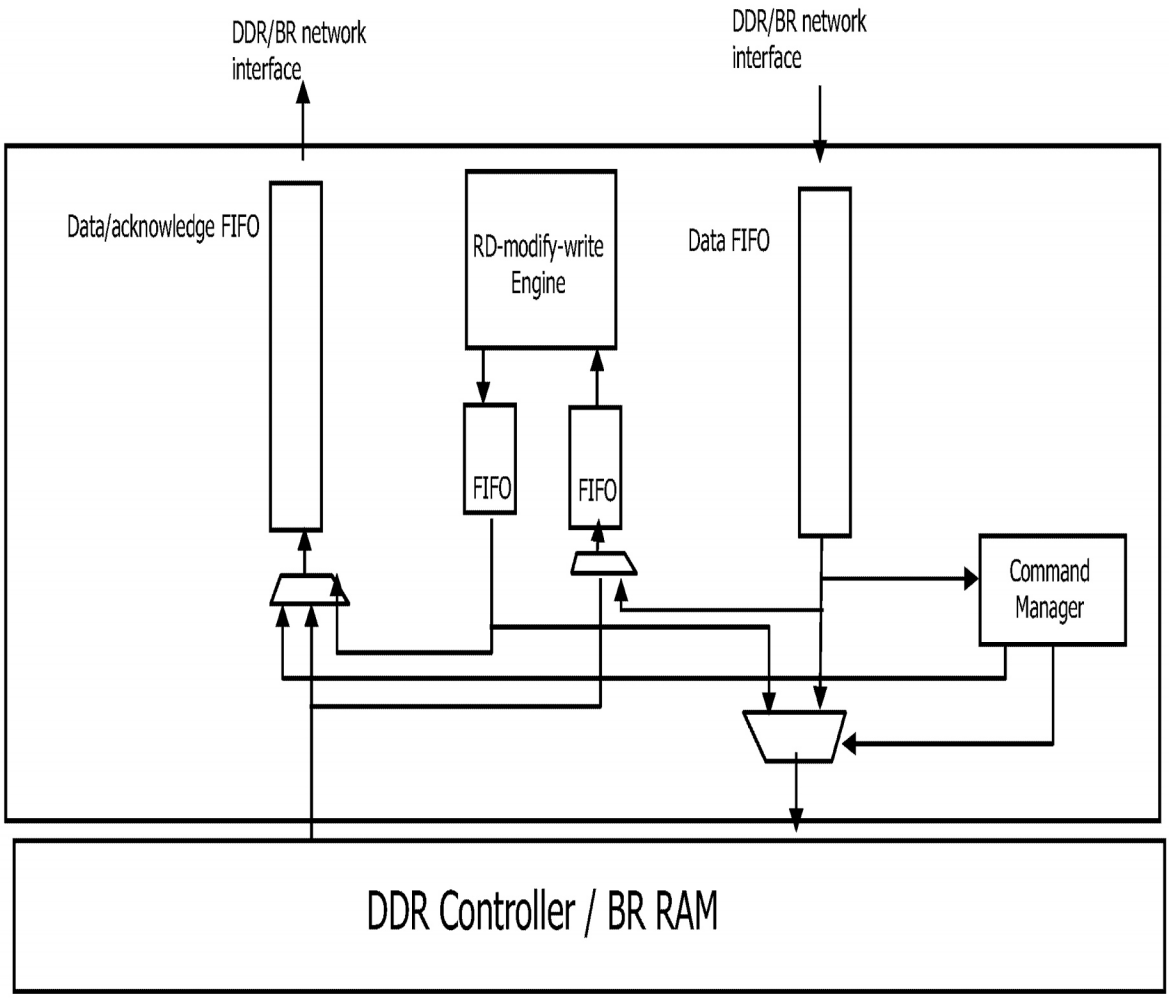
Processor block



The processor core is a third party product. However, the message network, data network and CAR connectivity architecture and implementation is an IDT product. The connectivity architecture provides for scalable efficient data and message networks interconnect. It is designed to minimize processor cycle overhead and provide direct and efficent access to the processor memory.

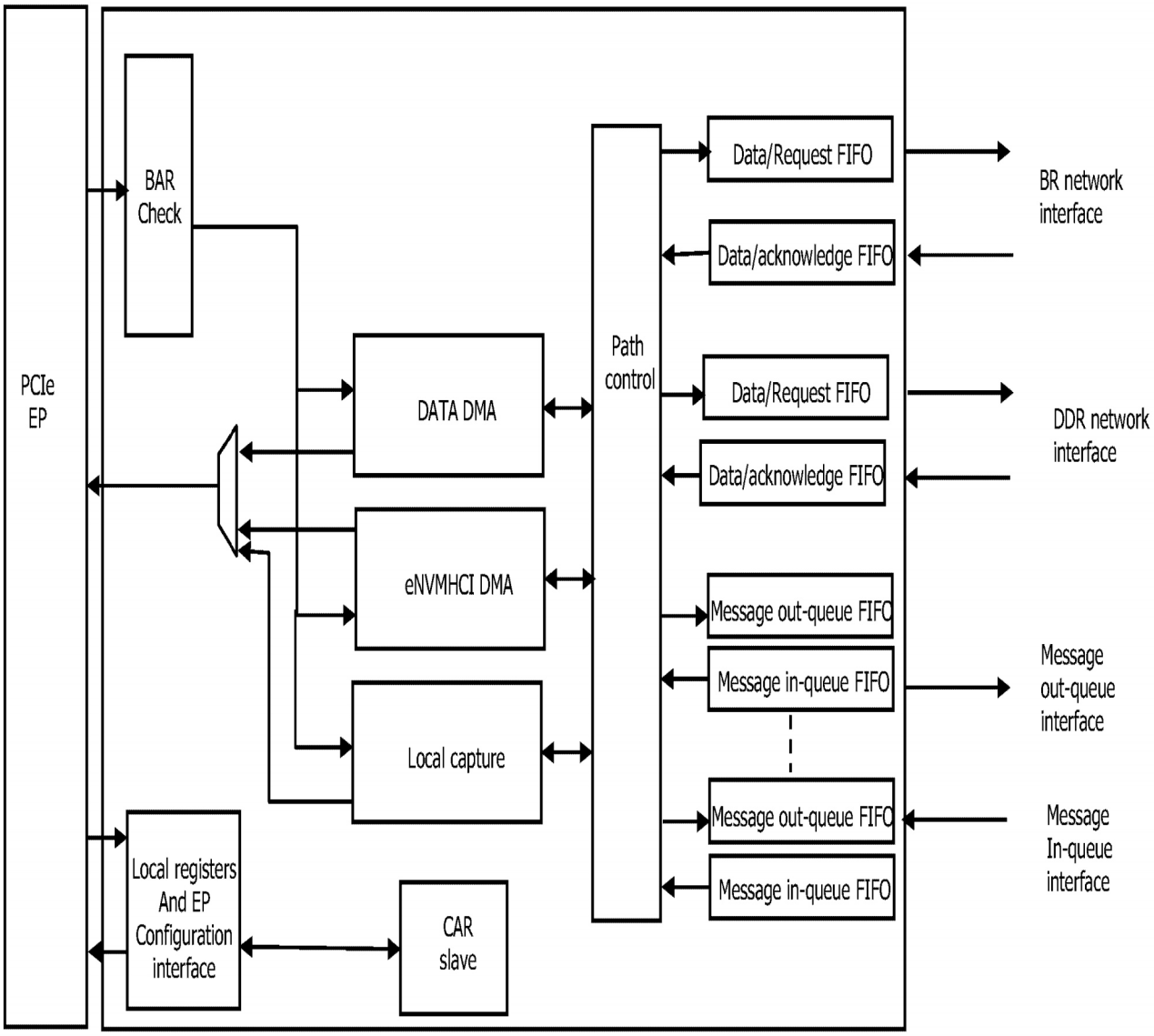
The functional operation is described in delaware_hw_spec.pdf

Memory network interface



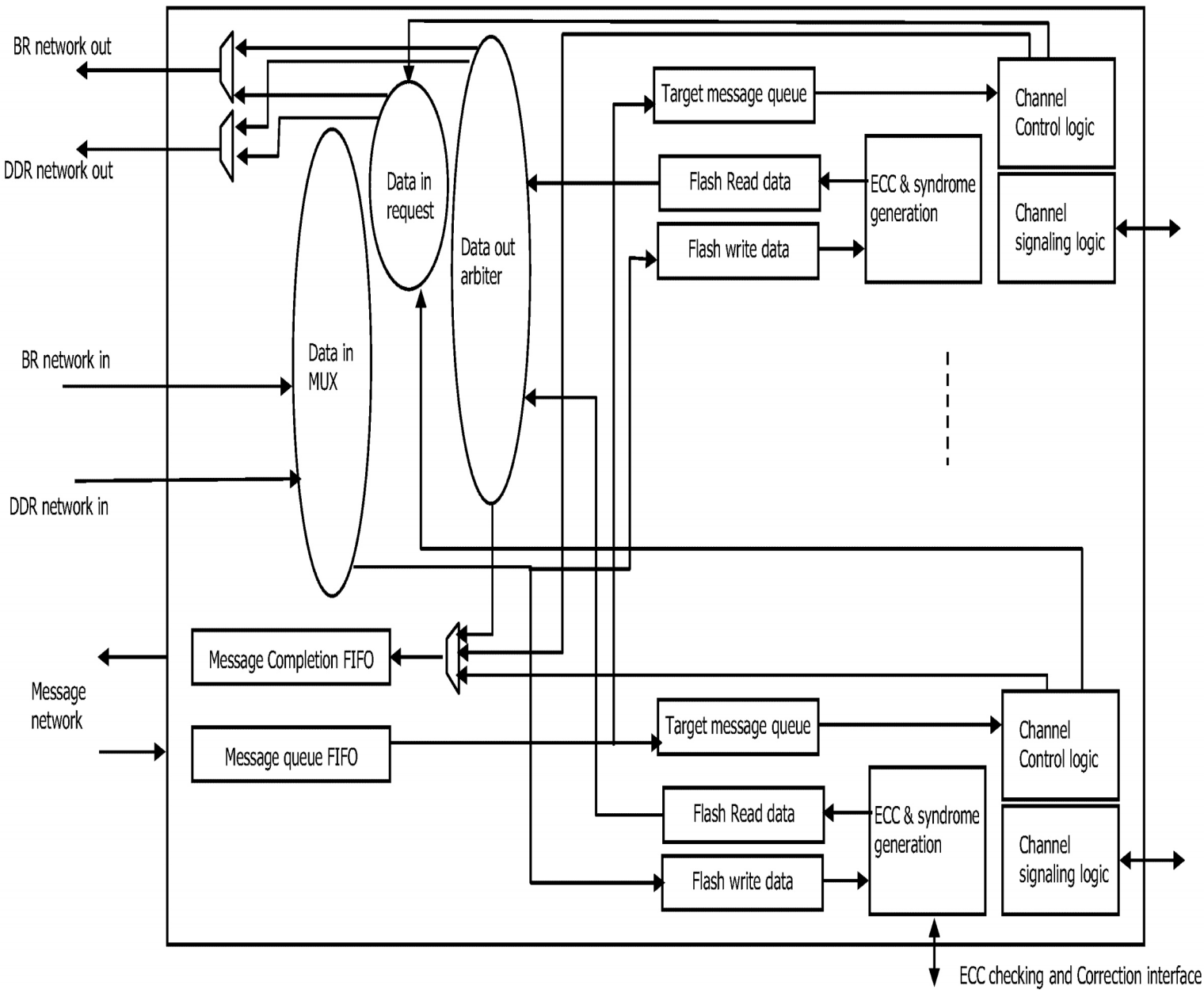
The Delaware Memory Network interface is used to interconnect processors and functional units to the Delaware memory buffers such as the external DRAM and internal SRAM. The Memory Network has small routing elements that route the request packets from the requestor to the Memory Controller. The Completions from the Memory Controller are broadcast to the requestors via a broadcast bus. There are six commands for the Memory Network: Rd, RdLock, RdXorWr, Wr, WrLn and Xfer.

The functional operation is described in delaware_hw_spec.pdf
The preferred embodiment is described in delaware_memory_interface.pdf



The DMA block provides the interface between the Host processor in a system and the Local processing and memory structures. It provides a packet receive and transit interface on the host side and data and message network connectivity on the device side. It can support high bandwidth throughput functionality to the host interface.

The functional operation is described in delaware_hw_spec.pdf



The Delaware flash controller is used to provide high bandwidth and efficient read and write operation to the flash devices. It is a scalable block with per channel logic for flash device interface and common logic for data and message network connectivity. It provides for a calibration capable and programable interface to the flash devices.

The functional operation is described in delaware_hw_spec.pdf



Delaware Enterprise Flash Controller Firmware Architecture Specification

August 9, 2010

25 Independence Blvd, Warren, New Jersey 07059
Telephone: (800) 345-7015 • (408) 284-8200 • FAX: (408) 284-2775
Printed in U.S.A.
©2007 Integrated Device Technology, Inc.

GENERAL DISCLAIMER

Integrated Device Technology, Inc. reserves the right to make changes to its products or specifications at any time, without notice, in order to improve design or performance and to supply the best possible product. IDT does not assume any responsibility for use of any circuitry described other than the circuitry embodied in an IDT product. The Company makes no representations that circuitry described herein is free from patent infringement or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent, patent rights or other rights, of Integrated Device Technology, Inc.

CODE DISCLAIMER

Code examples provided by IDT are for illustrative purposes only and should not be relied upon for developing applications. Any use of the code examples below is completely at your own risk. IDT MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE NONINFRINGEMENT, QUALITY, SAFETY OR SUITABILITY OF THE CODE, EITHER EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. FURTHER, IDT MAKES NO REPRESENTATIONS OR WARRANTIES AS TO THE TRUTH, ACCURACY OR COMPLETENESS OF ANY STATEMENTS, INFORMATION OR MATERIALS CONCERNING CODE EXAMPLES CONTAINED IN ANY IDT PUBLICATION OR PUBLIC DISCLOSURE OR THAT IS CONTAINED ON ANY IDT INTERNET SITE. IN NO EVENT WILL IDT BE LIABLE FOR ANY DIRECT, CONSEQUENTIAL, INCIDENTAL, INDIRECT, PUNITIVE OR SPECIAL DAMAGES, HOWEVER THEY MAY ARISE, AND EVEN IF IDT HAS BEEN PREVIOUSLY ADVISED ABOUT THE POSSIBILITY OF SUCH DAMAGES. The code examples also may be subject to United States export control laws and may be subject to the export or import laws of other countries and it is your responsibility to comply with any applicable laws or regulations.

LIFE SUPPORT POLICY

Integrated Device Technology's products are not authorized for use as critical components in life support devices or systems unless a specific written agreement pertaining to such intended use is executed between the manufacturer and an officer of IDT.

1. Life support devices or systems are devices or systems which (a) are intended for surgical implant into the body or (b) support or sustain life and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. A critical component is any components of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

IDT, the IDT logo, and Integrated Device Technology are trademarks or registered trademarks of Integrated Device Technology, Inc.

Numeric Representations

To represent numerical values, either decimal, binary, or hexadecimal formats will be used. The binary format is as follows: 0bDDD, where "D" represents either 0 or 1; the hexadecimal format is as follows: 0xDD, where "D" represents the hexadecimal digit(s); otherwise, it is decimal.

The compressed notation ABC[x|y|z]D refers to ABCxD, ABCyD, and ABCzD.

The compressed notation ABC[y:x]D refers to ABCxD, ABC(x+1)D, ABC(x+2)D,... ABCyD.

Data Units

The following data unit terminology is used in this document.

Term	Words	Bytes	Bits
Byte	1/2	1	8
Word	1	2	16
Doubleword (Dword)	2	4	32
Quadword (Qword)	4	8	64

Table 1 Data Unit Terminology

In quadwords, bit 63 is always the most significant bit and bit 0 is the least significant bit. In doublewords, bit 31 is always the most significant bit and bit 0 is the least significant bit. In words, bit 15 is always the most significant bit and bit 0 is the least significant bit. In bytes, bit 7 is always the most significant bit and bit 0 is the least significant bit.

The ordering of bytes within words is referred to as either "big endian" or "little endian." Big endian systems label byte zero as the most significant (leftmost) byte of a word. Little endian systems label byte zero as the least significant (rightmost) byte of a word. See Figure 1.

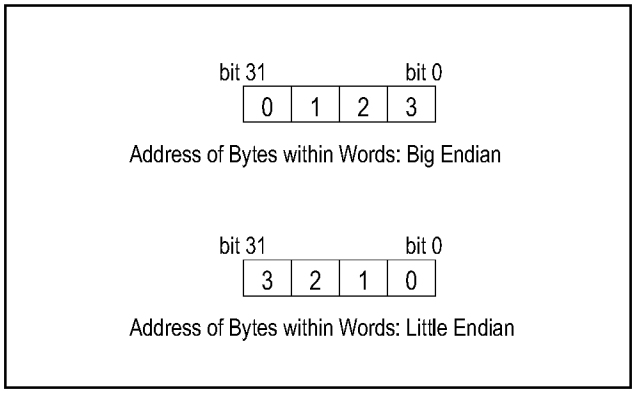


Figure 1 Example of Byte Ordering for "Big Endian" or "Little Endian" System Definition

Table Of Contents

Chapter 1 Overview

1.1 Hardware 1-1
1.1.1 CPU Architecture 1-3
1.1.1.1 Address Map 1-4
1.1.1.2 Interrupts 1-4
1.1.1.3 Exception Vectors 1-5
1.1.1.4 Reset Vectors 1-6
1.2 Firmware 1-6
1.2.1 Architectural Overview 1-7
1.2.2 Ware-leveling 1-7
1.2.3 Message Network 1-7
1.2.4 Memory Network 1-8
1.2.5 RTOS Support 1-8
1.2.6 Low Power Considerations 1-8

Chapter 2 Control Manager

2.1 Overview 2-1
2.2 NVRAM Control 2-1
2.2.1 Partitions 2-1
2.2.1.1 Superblock Partition 2-2
2.2.1.2 Bad Block Table Partition 2-2
2.2.1.3 File Partitions 2-2
2.2.1.4 Delaware NVRAM Memory Map 2-5
2.3 DRAM Control 2-5
2.3.1 Memory Map 2-6
2.4 Startup Sequence 2-6
2.4.1 Scanning for a valid boot block 2-7
2.4.2 Bad Block Scanning and Determination 2-8
2.5 Shutdown Sequence 2-8
2.6 Interrupts and Error Handling 2-9
2.6.1 Control Partition Error recovery 2-9

Chapter 3 PCIe Manager

3.1 Overview 3-1

Chapter 4 Data Manager

4.1 Overview 4-1
4.2 Architecture 4-1
4.3 Command Initiation 4-3
4.4 Releasing a Command State Structure 4-4
4.5 Requesting the Command 4-4
4.6 Obtaining the Command 4-4
4.7 Command Processing 4-4
4.8 Page Processing 4-5
4.8.1 Read Page Processing 4-6
4.8.2 Write Page Processing 4-7
4.9 Releasing a Page State Structure 4-7
4.10 Inbound Message Queues 4-7
4.10.1 Inbound Messages 4-7
4.11 Outbound Message Queues 4-8
4.11.1 Outbound Messages 4-8

Chapter 5 Write Manager

5.1 Overview 5-1
5.2 Architecture 5-1
5.3 Performance Considerations 5-2
5.4 Inbound Message Queues 5-2
5.4.1 Inbound Messages 5-2
5.5 Outbound Message Queues 5-3
5.5.1 Outbound Messages 5-3

Chapter 6 Wearleveling Manager

6.1 Overview 6-1
6.2 Wear Leveling 6-1
6.2.1 Priority based Algorithm 6-2
6.2.2 Block Information Management 6-4
6.2.3 FreeList and DirtyList Management 6-6
6.2.4 State Diagrams 6-10
6.2.5 Block State Machine 6-11
6.2.6 Obsolete State Machine 6-11
6.2.7 Reclaim State Machine 6-12
6.2.8 Allocation of Free Flash Block for Writing 6-13
6.2.9 Wear leveling across channel/target 6-14
6.3 Garbage Collecting 6-15
6.3.1 State Diagram 6-16
6.4 Bad Block Management 6-18
6.5 Inbound Message Queues 6-18
6.6 Inbound Outbound Messages 6-18

Chapter 7 Page Frame Manager

7.1 Overview 7-1
7.2 Architecture 7-1
7.2.1 Initialization 7-2
7.2.2 Page Frame Allocation 7-2
7.2.3 Page Frame Freeing 7-2
7.2.4 Statistics 7-3
7.3 Performance Considerations 7-3
7.4 Inbound Message Queues 7-3
7.4.1 Inbound Messages 7-3
7.5 Outbound Message Queues 7-3
7.5.1 Outbound Messages 7-3

Chapter 8 Inter-processor Communications

8.1 Overview 8-1
8.2 Firmware 8-1
8.2.1 Initialization 8-1
8.2.2 IPC Map Generator 8-2
8.2.3 Interrupts 8-3
8.2.4 Sending Messages 8-4
8.2.5 Receiving Messages 8-5
8.2.6 IPC Message Queue Library API 8-5
8.2.6.1 Library Files 8-5
8.2.6.2 Include Files 8-5
8.2.6.3 void IPCInit(const IPC_CONFIG *map) 8-5
8.2.6.4 unsigned int IPCOutboundGet(int qIndex) 8-6
8.2.6.5 void IPCOutboundPost(int qIndex, unsigned int handle) 8-6
8.2.6.6 void *IPCInboundGet(int qIndex, unsigned int *handle) 8-6
8.2.6.7 void IPCInboundFree(unsigned int handle) 8-6
8.2.6.8 int IPCOutboundIntMask(unsigned int mask) 8-6
8.2.6.9 int IPCInboundIntMask(unsigned int mask) 8-6
8.2.6.10void IPCWriteData(int qIndex, unsigned int handle, unsigned int data) 8-7
8.2.6.11void *IPCInboundGetAny(unsigned int mask, unsigned int *handle, int *qIndex) 8-7

Chapter 9 Memory Network

9.1 Overview 9-1
9.2 Memory Network API 9-1
9.2.1 void memInit(void) 9-1
9.2.2 void memMoveOutPosted(u32 source, reg64 dest, int len) 9-1
9.2.3 void memAddPosted(reg64 addr, u32 value) 9-1
9.2.4 void memXorPosted(reg64 addr, u32 value) 9-1
9.2.5 void memXferPosted(reg64 source, reg64 dest, int len) 9-2
9.2.6 int memMoveOut(u32 source, reg64 dest, int len) 9-2
9.2.7 int MemAdd(reg64 addr, u32 value) 9-2
9.2.8 int memXor(reg64 addr, u32 value) 9-2
9.2.9 int memXfer(reg64 source, reg64 dest, u32 len) 9-3

9.2.10 int memMoveIn(reg64 source, u32 dest, int len) ... 9-3
9.2.11 int memMoveInAndLock(reg64 source, u32 dest, int len, int lockBit) ... 9-3
9.2.12 void memFreeTag(u32 tag) ... 9-4

Chapter 10 CRC Library

10.1 Overview ... 10-1
10.1.1 crc crcFast(unsigned char const message[], int nBytes) ... 10-1

Chapter 11 Software Managed FIFO's

11.1 Overview ... 11-1
11.2 FIFO Library API ... 11-1
11.2.1 Library Files ... 11-1
11.2.2 Include Files ... 11-2
11.2.3 void fifoInit(DATA_FIFO *fifo, void **storage, int maxEntries) ... 11-2
11.2.4 int fifoPush(DATA_FIFO *fifo, void *data) ... 11-2
11.2.5 void *fifoPop(DATA_FIFO *fifo) ... 11-2
11.2.6 void *fifoPeek(DATA_FIFO *fifo) ... 11-2
11.2.7 int fifoIsEmpty(DATA_FIFO *fifo) ... 11-2
11.3 FIFO Data Structure ... 11-3

Chapter 12 IPC Messages

12.1 Overview ... 12-1
12.2 Functional Unit Messages ... 12-1
12.2.1 CmdBufFree ... 12-1
12.2.2 CmdReady ... 12-1
12.2.3 DMACpl ... 12-1
12.2.4 EgressDMA ... 12-1
12.2.5 EgressTLP ... 12-1
12.2.6 EgressTLPCpl ... 12-1
12.2.7 FlashErase ... 12-1
12.2.8 FlashOpCpl ... 12-1
12.2.9 FlashRead ... 12-2
12.2.10 FlashWrite ... 12-2
12.2.11 IngressDMA ... 12-2
12.2.12 IngressTLP ... 12-2
12.2.13 StsQueueUpdate ... 12-2
12.2.14 StsQueueUpdateCpl ... 12-2
12.3 Inter-Manager Messages ... 12-2
12.3.1 Block Open Request ... 12-2
12.3.2 Block Open Reply ... 12-3
12.3.3 Block Close ... 12-4
12.3.4 Sync Request ... 12-4
12.3.5 Sync Reply ... 12-5
12.3.6 Page Write ... 12-5
12.3.7 Obsolete Page ... 12-5
12.3.8 Page Frame Init ... 12-6

12.3.9 Page Frame Alloc Request 12-6
12.3.10 Page Frame Free 12-7
12.3.11 Page Frame Alloc Reply 12-8
12.3.12 Power Down Request 12-8
12.3.13 Power Down Reply 12-9
12.3.14 Power Up Request 12-9
12.3.15 Power Up Reply 12-10
12.3.16 Reclaim Block Request 12-11
12.3.17 Reclaim Block Reply 12-11
12.3.18 XON 12-12
12.3.19 New Command 12-13
12.4 WRITE_REQUEST 12-14
12.5 FLUSH_ALL 12-15
12.6 FLUSH_ALL_REPLY 12-15
12.7 LOG_ENTRY 12-16
12.7.1 Page Map Log Entry 12-17
12.7.2 Erase Count Log Entry 12-17
12.7.3 Bad Block Log Entry 12-18
12.7.4 Block State Log Entry 12-18
12.7.5 Block Valid Page Log Entry 12-19
12.7.6 Next Block Log Entry 12-19
12.8 BLOCK_FREE_REQUEST 12-19
12.9 PAGE_WRITE 12-20
12.10 PAUSE_OPERATION 12-20
12.11 OPERATION_PAUSED 12-21
12.12 RESUME_OPERATION 12-21
12.13 INIT_DATA 12-22
12.14 INIT_DATA_COMPLETE 12-22
12.15 INITIATE_SNAPSHOT 12-23
12.16 SNAPSHOT_COMPLETE 12-23
12.17 BAD_BLOCK 12-24

Chapter 13 Data Structures

13.1 Overview 13-1
13.2 NVMHCI Data Structures 13-1
13.2.1 Command Header 13-1
13.2.2 Physical Region Descriptor Table Entry 13-1
13.2.3 Physical Region Descriptor Index Table Entry 13-1
13.3 List Head Structure 13-2
13.4 Memory Network Tag Structure 13-2
13.5 Page Cache Header 13-2
13.6 Partition Table 13-3
13.6.1 Partition Types 13-3
13.7 Super Block 13-3

13.7.1 Super Block Header 13-4
13.8 File Header 13-4
13.9 Logical to Physical Map 13-4
13.10 Meta Data Map 13-5
13.11 Log Page Header 13-5
13.12 Snapshot Page Header 13-5

Chapter 14 64-Bit Calculations

14.1 Overview 14-1
14.2 reg64 MlpToR64(unsigned int) 14-1
14.3 unsigned int R64ToMlp(reg64) 14-1
14.4 reg64 AddR64Ar(reg64, int) 14-1
14.5 reg64 AddR64UAr(reg64, unsigned int) 14-2
14.6 reg64 AddR64R64(reg64, reg64) 14-2
14.7 reg64 SubR64R64(reg64, reg64) 14-2

Chapter 15 eNVMHCI

15.1 Overview 15-1
15.2 Vendor Specific Commands 15-1
15.2.1 Physical Read 15-1

Chapter 16 References

List Of Figures

Figure 1.1 High Level Architectural Block Diagram 1-1
Figure 1.2 108Mini Block Diagram 1-3
Figure 1.3 108Mini Address Map 1-4
Figure 1.4 XTOS/XTHAL Interrupt Example 1-5
Figure 1.5 Example Processor Complex with Processor IDs 1-6
Figure 2.1 Superblock Partition Organization 2-2
Figure 2.2 Bad Block Table Organization 2-2
Figure 2.3 Partition Containing a File 2-3
Figure 2.4 Page Write Error 2-4
Figure 2.5 Super Block Partition Layout 2-7
Figure 2.6 Factory Bad Block Scanning Algorithm 2-8
Figure 4.1 Command State Structure 4-2
Figure 4.2 Page State Structure 4-2
Figure 4.3 Data Structure Relationships 4-3
Figure 4.4 Inbound Message Buffer Structure 4-3
Figure 4.5 Memory Network Tag Structure 4-4
Figure 4.6 Memory Network Request Types 4-4
Figure 4.7 Pseudo Code to Process Page Requests 4-6
Figure 5.1 Block State Structure 5-1
Figure 5.2 Inbound Message Buffer Structure 5-1
Figure 6.1 Wear Levelling Queue Structures 6-2
Figure 6.2 Setting the Priority 6-3
Figure 6.3 Priority Lookup Table's relationship to the DirtyList 6-7
Figure 6.4 Adding a new entry 6-8
Figure 6.5 Moving an entry in the DirtyList 6-9
Figure 6.1 Block Open/Close State MachineDiagram 6-11
Figure 6.6 Obsolete Page State Machine Disgram 6-12
Figure 6.7 Reclaim Block State Machine Diagram 6-13
Figure 6.8 Allocation of next block to write across channels and targets 6-14
Figure 6.9 Garbage Collecting State Diagram 6-17
Figure 7.1 Page Frame Buffer Base Addresses in DDR and Buffer RAM 7-1
Figure 7.2 Page Alloc Data Structure 7-1
Figure 8.1 IPC Operation 8-1
Figure 8.2 IPC Config Structure 8-2
Figure 8.3 IPC Configuration Map 8-2
Figure 8.4 IPC Manager Config File 8-3
Figure 8.5 IPC Configuration Map Text File 8-3
Figure 8.6 Outbound IPC Usage Example 8-4
Figure 8.7 Inbound IPC Usage Example 8-5
Figure 8.8 Message Dispatching Example 8-7
Figure 11.1 Software Managed FIFO Usage Example 11-1
Figure 12.1 Block Open Request Message Format 12-2
Figure 12.2 Block Open Reply Message Format 12-3
Figure 12.3 Block Close Message Format 12-4
Figure 12.4 Sync Reply Message Format 12-4
Figure 12.5 Sync Reply Message Format 12-5
Figure 12.6 Obsolete Page Message Format 12-5
Figure 12.7 Page Frame Init Message Format 12-6
Figure 12.8 Page Frame Alloc Message Format 12-7

Figure 12.9 Page Frame Free Message Format 12-7
Figure 12.10 Page Frame Alloc Reply Message Format 12-8
Figure 12.11 Power Down Request Message Format 12-9
Figure 12.12 Power Down Reply Message Format 12-9
Figure 12.13 Power Up Request Message Format 12-10
Figure 12.14 Power Up Reply Message Format 12-10
Figure 12.15 Reclaim Block Request Message Format 12-11
Figure 12.16 Reclaim Block Reply Message Format 12-12
Figure 12.17 XON Message Format 12-12
Figure 12.18 New Command Message Format 12-13
Figure 12.19 NVMHCI_CMD_DONE Message Format 12-14
Figure 12.20 WRITE_REQUEST Message Format 12-14
Figure 12.21 FLUSH_ALL Message Format 12-15
Figure 12.22 FLUSH_ALL_REPLY Message Format 12-16
Figure 12.23 Page Mapping Table Log Entry Format 12-17
Figure 12.24 Erase Count Log Entry Format 12-17
Figure 12.25 Bad Block Log Entry Format 12-18
Figure 12.26 Block State Log Entry Format 12-18
Figure 12.27 Block Valid Page Log Entry Fields 12-19
Figure 12.28 Next Block Log Entry Fields 12-19
Figure 12.29 BLOCK_FREE_REQUEST Message Format 12-20
Figure 12.30 PAGE_WRITE Message Format 12-20
Figure 12.31 PAUSE_OPERATION Message Format 12-21
Figure 12.32 OPERATION_PAUSED Message Format 12-21
Figure 12.33 RESUME_OPERATION Message Format 12-21
Figure 12.34 INIT_DATA Message Format 12-22
Figure 12.35 INIT_DATA_COMPLETE Message Format 12-22
Figure 12.36 INITIATE_SNAPSHOT Message Format 12-23
Figure 12.37 SNAPSHOT_COMPLETE Message Format 12-23
Figure 12.38 BAD_BLOCK Message Format 12-24

List Of Tables

Table 1.1 IRQ Mappings 1-4
Table 1.2 108Mini Exception Vectors 1-6
Table 2.1 Partition Table Types 2-1
Table 2.3 Partition Table Types used in DRAM 2-6
Table 2.5 Factory Bad Block Marker 2-8
Table 6.1 Write Manager 6-18
Table 6.2 Wearleveling Manager Issued and Received Message 6-19
Table 12.1 Block Open Request Message Fields 12-3
Table 12.2 Block Open Reply Message Fields 12-3
Table 12.3 Block Close Message Fields 12-4
Table 12.4 Sync Request Message Fields 12-4
Table 12.5 Sync Reply Message Fields 12-5
Table 12.6 Obsolete Page Message Fields 12-6
Table 12.7 Page Frame Init Message Fields 12-6
Table 12.8 Page Frame Alloc Message Fields 12-7
Table 12.9 Page Frame Free Message Fields 12-7
Table 12.10 Page Frame Alloc Reply Message Fields 12-8
Table 12.11 Power Down Request Message Fields 12-9
Table 12.12 Power Down Reply Message Fields 12-9
Table 12.13 Power Up Request Message Fields 12-10
Table 12.14 Power Up Reply Message Fields 12-11
Table 12.15 Reclaim Block Request Message Fields 12-11
Table 12.16 Reclaim Block Reply Message Fields 12-12
Table 12.17 XON Message Fields 12-12
Table 12.18 New Command Message Fields 12-13
Table 12.19 NVMHCI_CMD_DONE Message Fields 12-14
Table 12.20 WRITE_REQUEST Message Fields 12-14
Table 12.21 FLUSH_ALL Message Fields 12-15
Table 12.22 FLUSH_ALL_REPLY Message Fields 12-16
Table 12.23 LOG_ENTRY Message Format 12-16
Table 12.24 LOG_ENTRY Message Fields 12-16
Table 12.25 Page Mapping Table Log Entry Fields 12-17
Table 12.26 Erase Count Log Entry Fields 12-17
Table 12.27 Bad Block Log Entry Fields 12-18
Table 12.28 Block State Log Entry Fields 12-18
Table 12.29 Block Valid Page Log Entry Fields 12-19
Table 12.30 BLOCK_FREE_REQUEST Message Fields 12-20
Table 12.31 PAGE_WRITE Message Fields 12-20
Table 12.32 PAUSE_OPERATION Message Fields 12-21
Table 12.33 OPERATION_PAUSED Message Fields 12-21
Table 12.34 RESUME_OPERATION Message Fields 12-21
Table 12.35 INIT_DATA Message Fields 12-22
Table 12.36 INIT_DATA_COMPLETE Message Fields 12-22
Table 12.37 INITIATE_SNAPSHOT Message Fields 12-23
Table 12.38 SNAPSHOT_COMPLETE Message Fields 12-24
Table 12.39 BAD_BLOCK Message Fields 12-24

Overview

1.1 Hardware

The Delaware FPGA prototype is a programmable PCI Express (PCIe) NAND Flash controller. The hardware architecture provides a high performance, symmetric, and simple to program platform for controlling NAND Flash and interfacing to a host via PCIe. Unlike other high performance NAND Flash controllers that hardwire algorithms in order to achieve high performance, all Delaware controller operations are under firmware control enabling OEMs to differentiate product offerings by deploying proprietary Flash management and controller algorithms.

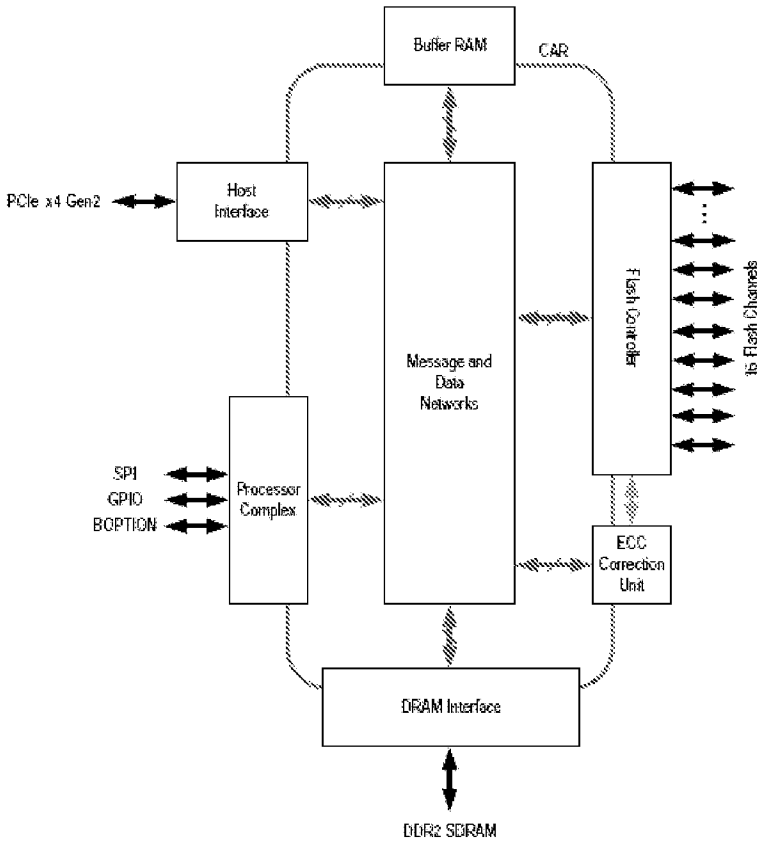


Figure 1.1 High Level Architectural Block Diagram

The architecture consists of six functional units.

- *Host Interface*
  - x4 PCIe Gen2 interface
  - Ingress and Egress DMA
  - Queueing and dispatching of NVMHCI commands
  - Reception and transmission of PCIe TLPs associated with memory mapped I/O and configuration transactions

- *Flash Controller - Interface to 16 ONFI 2.1 Flash channels*
  - Computes and inserts ECC parity
  - Checks ECC syndrome
  - Orchestrates ECC correction using ECC correction unit
- *ECC Correction Unit*
  - Performs ECC correction
- *Buffer RAM*
  - High speed on-chip SRAM for temporary storage of Flash data and controller data structures
- *DRAM Interface*
  - An interface to high speed external DDR SDRAM for temporary storage of Flash data and controller data structures
- *Processor Complex*
  - A pool of programmable processors (Tensilica cores) used to execute Flash management and controller algorithms.
    - Except for the control processor, all processors cores are identical. The archi-tecture enables the processors to be used as a pool of processors, a pipeline of processors, or as a combination of the two approaches.
    - The control processor has additional capabilities and is responsible for booting the system and providing overall control and management.

Functional units communicate using one of five interconnects.

- *Message Network (MSGN)*
  - The MSGN provides routing of short messages between functional units and processors. These messages are used to orchestrate operations. Bulk data transfers use one of the data networks.
- *Buffer RAM Data Network (BRDN)*
  - The BRDN provides bulk data transfer between functional units and buffer RAM.
- *DRAM Data Network (DRDN)*
  - The BRDN provides bulk data transfer between functional units and the DRAM interface.
- *Common Access Ring (CAR)*
  - The CAR provides access to functional unit control and status registers. This is a relatively low performance interface intended for initialization, debug, statistics collection and error recovery.
- *Global Interrupts*
  - Interrupts provide a mechanism for the signalling of events to processors. There are three types of global interrupts.
    - Attention Interrupts
    - Each function unit and processor has the ability to generate an attention interrupt. These interrupts are used to indicate that an event has occurred within the functional unit that requires special processing. All attention interrupts are routed to the control processor.
    - Processor Attention Interrupts
    - A processor attention interrupt is routed from the control processor to each of the other processors. This interrupt is used by the control processor to signal to a specific processor that special processing is required by that processor.
    - Power Down Interrupt
    - A power fail interrupt is generated by the assertion of the Power Down (PWRDN) signal. The power fail interrupt is routed to all of the proces-sors.
- *Each functional unit and processor has a node ID. The node ID uniquely identifies a func-tional unit or*

processor and is used for routing in the message and data networks.

- *Buffer RAM and DRAM controlled by the DRAM* interface together are referred to as controller memory.*
  - Controller memory is used to hold controller data structures and Flash data.
  - Except for a performance difference, controller memory is completely symmetric. Anything that may be stored or any operation that may be performed in one memory (e.g., buffer RAM) may be stored and performed in the other memory (i.e., DRAM).
  - Controller memory is accessed by functional units using a flat 37-bit address. This address is called a controller memory address. The most significant bit (i.e., bit 37) of the controller memory address is used to select buffer RAM or DRAM. The remaining bits form a byte address in the selected memory.
    - Although the FPGA prototype implements supports far less memory, the controller architecture supports a maximum of 64GB of DRAM.
  - A 64B quantity is referred to as a memory line. Controller memory is optimized for memory line accesses. Although memory is byte addressable, highest performance is achieved with memory line accesses.

- *Host memory refers to memory addressable through the PCIe interface. While host memory is typically physically located in the root of the PCIe hierarchy, there is nothing to prevent this memory from being located within any other PCIe function.*

1.1.1 CPU Architecture

The Tensilica 108Mini is a 32-bit little-endian, cache-less, 5-stage pipeline, RISC architecture CPU core which includes 32-bits of GPIO, 22 interrupts, exception handing and a memory protection unit. The figure below depicts a simplified block diagram of the 108Mini.

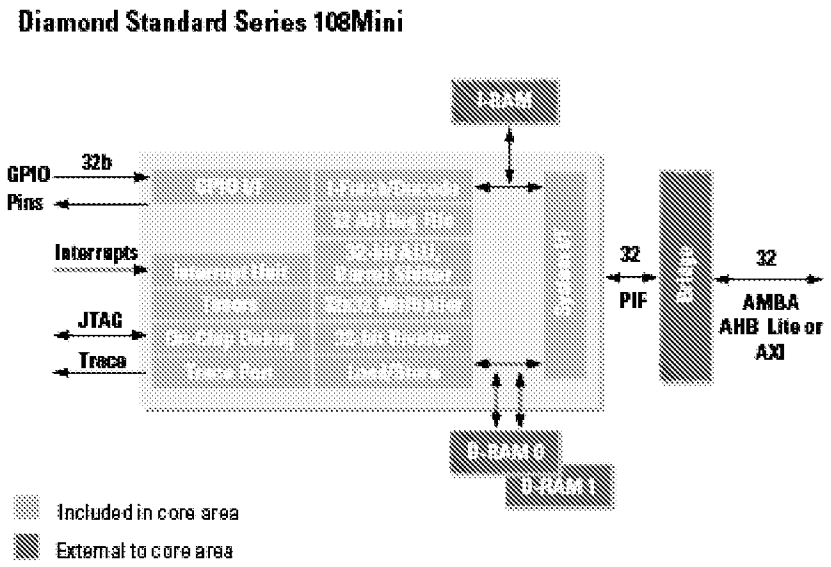


Figure 1.2 108Mini Block Diagram

Each 108Mini will have access to 64KB of local instruction memory (starting at 0x7ffa0000) and 64KB of local data memory (starting at 0x7ff80000). Local data and instruction memories are extremely efficient, having an access time of only one cycle. See the Address Map section for more details relating to the layout of the 108Mini address space.

Custom instructions will be created to handle the most time consuming operations of the firmware. These instructions are described in detail in the Processor Complex section of the Delaware FSD.

1.1.1.1 Address Map

The 108Mini is capable of accessing 4GB of address space. The figure below presents a high level overview of this address space. Custom linker support packages (LSP) has been created to support this memory map and should be used for all firmware development. The custom LSPs are located in the Delaware CVS repository under delaware/firmware/lsp.

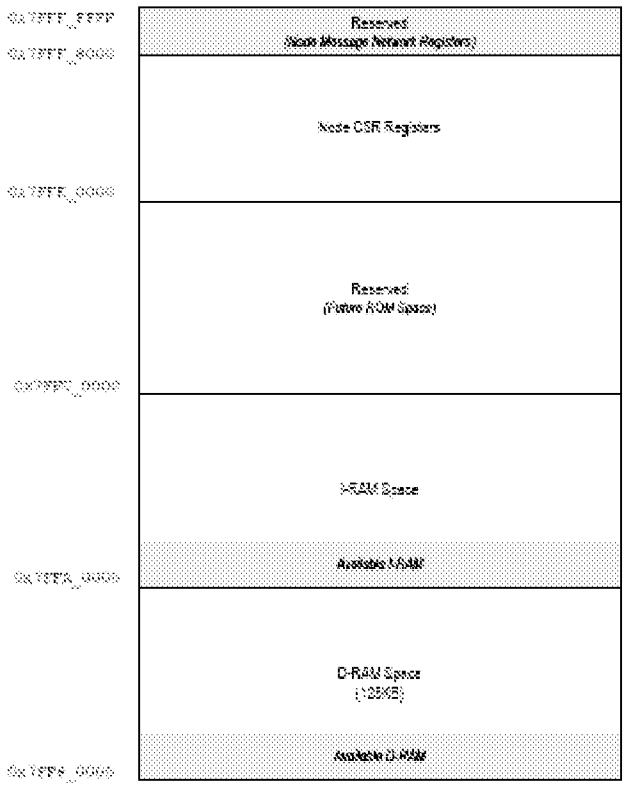


Figure 1.3 108Mini Address Map

1.1.1.2 Interrupts

The Delaware core is configured to handle 12 IRQ signals ranging in priority from 1 to 6. The priority level of an IRQ directly relates to the CPU's interrupt level as specified by the *psintlevel* register. When the *psintlevel* register is set to 0, all interrupts are enabled, when set to 1, only interrupts above priority 1 are enabled, etc. The table below lists each IRQ, it's sensitivity level, priority and assigned function.

IRQ #	Edge/Level	Priority	Function	Notes
0	-	1	Software	
1	-	1	Timer 0	
2	-	1	WriteErr	
3	Level	2	Outbound IPC	
4	Level	2	Inbound IPC	
5	Level	2	Memory Net-work	
6	Level	3	Attention	
7	-	3	Timer 1	

Table 1.1 IRQ Mappings

IRQ #	Edge/Level	Priority	Function	Notes
8	Level	3	Reserved (fatal error)	
9	-	3	Software	
10	Edge	4	Power Down	
11	-	6	NMI	Power Down NMI

Table 1.1 IRQ Mappings

Software support for interrupt handling will be provided by the Tensilica XTOS and XTHAL libraries. Figure 1.4 shows an example of how to use the Tensilica XTOS and XTHAL library calls to setup a timer interrupt and how to place the processer into a low power state waiting for an interrupt to occur.

_xtos_set_interrupt_handler() is used to attach a function to the interrupt for timer 0. Compare register 0 is then set to the current value of the *ccount* register plus a timeout value. Finally, interrupts are unmasked for timer 0. When timer 0 generates an interrupt, timerISR() is called. This function sets the new compare value for timer 0 and increments the *count* variable. The main loop of the program executes the "waiti 0" instruction. This instruction puts the processor into a state where it waits for an interrupt. When an interrupt occurs, the timer ISR is executed, upon exit from the ISR, execution continues from the instruction following the *waiti* instruction. Which in this case is a call to printf(). Refer to the Low Power Considerations section for more details relating to the *waiti* instruction.

```
#include <stdio.h>
#include <xtensa/hal.h>
#include <xtensa/xtruntime.h>

#define TIMEOUT             0x10000
#define TIMER0_INT_MASK     (1<< XCHAL_TIMER0_INTERRUPT)
static int                  count = 0;

void timerISR(void)
{
       xthal_set_ccompare(0, xthal_get_ccompare(0) + TIMEOUT);
       count++;
}

int main(int argc, char *argv[])
{
       _xtos_set_interrupt_handler(XCHAL_TIMER0_INTERRUPT, timerISR);
       xthal_set_ccompare(0, xthal_get_ccount() + TIMEOUT);
       _xtos_ints_on(TIMER0_INT_MASK);

       for(;;)
       {
              asm("waiti    0");
              printf("%d\n", count);
       }
       return(0);
}
```

Figure 1.4 XTOS/XTHAL Interrupt Example

1.1.1.3 Exception Vectors

The 108Mini supports eleven exception vectors. Two are assigned as reset vectors, seven are assigned to interrupt request priorities (1-7), one is assigned to Register Window Overflow, and one is assigned as a Kernel exception vector.

Exception Address	Size	Description
0x7FFA_0000	0x180	Window Register Overflow Vector
0x7FFA_0188	0x38	Level 2 Interrupt Vector
0x7FFA_01c8	0x38	Level 3 Interrupt Vector
0x7FFA_0208	0x38	Level 4 Interrupt Vector
0x7FFA_0248	0x38	Level 5 Interrupt Vector
0x7FFA_0288	0x1C	Level 6 Interrupt Vector
0x7FFA_0300	0x38	Kernel (Stacked) Exception Vector
0x7FFA_0340	0x38	User (Program) Exception Vector
0x7FFA_03C0	0x40	Double Exception
0x7FFA_0400	0x300	Reset Vector

Table 1.2 108Mini Exception Vectors

The management of these vectors will be provided by the Tensilica XTOS and XTHAL libraries. These libraries will also provide the default exception handlers.

1.1.1.4 Reset Vectors

As can be seen from Table 1.2, the 108Mini supports two individual reset vectors. The *System ROM Reset Vector* and the *Alternate Reset Vector*. The default configuration would be for the CPU to begin executing code at the *System ROM Reset Vector* after a reset, however, if the *StatVectorSel* input pin to the core is driven high (*SimStaticVectorSelect* simulator parameter is set to 1), the core will begin to execute code at the *Alternate Reset Vector* after a reset. This functionality is useful in multi-processor systems like the Delaware device where a single CPU core, which boots from the *System ROM Reset Vector*, may be used to load code directly into the instruction memory of other processors in the system before releasing their reset signals causing them to boot from instruction memory at the *Alternate Reset Vector*.

1.2 Firmware

As described in the Hardware section, the processor complex will contain multiple embedded processors capable of passing information to each other and to hardware blocks via a network of message queues. Each processor will have a firmware image which will be loaded into the processors instruction/data memory during the system initialization process. Full details of this initialization process are defined in the Initialization Sequence section of this document.

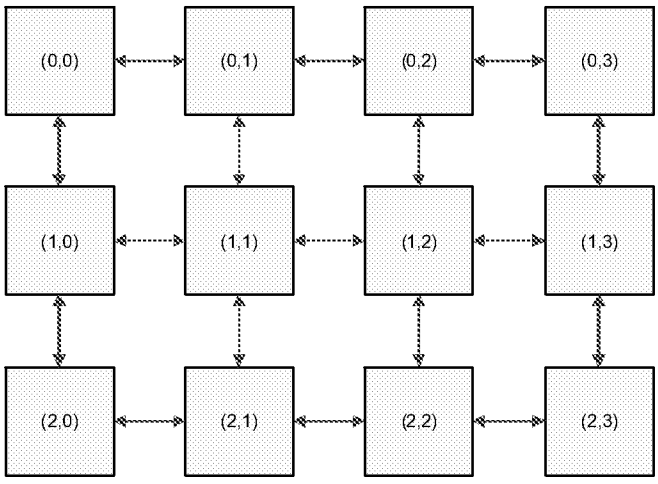


Figure 1.5 Example Processor Complex with Processor IDs

Each processor will have a unique processor ID which will be hard coded into its PRID register. This ID will not only uniquely identify each processor but also its location within the processor complex. The format of the PRID register is described below.

- *Bits 7:4 - Row location within the processor complex.*
- *Bits 3:0 - Column location within the processor complex.*

To provide a high degree of portability and maintainability the Delaware firmware will be written utilizing the 'C' programing language (although there may be a need to implement some functionality at the assembly level). The firmware is located in the Delaware CVS repository under delaware/firmware/EFC.

1.2.1 Architectural Overview

The processors within the processor complex will be configured into 5 different classes, each class of processor is responsible for a perticular set of functionality. The processor classes are:

- *PCIe Manager - Responsible for the management of the PCIe endpoint and for the generation of PCIe TLP's*
- *Control Manager - Responsible for the initialization of the system, and the loading and saving of the system state information to flash.*
- *Data Manager - Responsible for managing read/write requests from the host.*
- *Wearleveling Manager - Responsible for managing the wearleveling aspects of the flash.*
- *Write Manager - Responsible for assigning physical flash page addresses for host write commands.*
- *Page Frame Manager - Manages the allocation and freeing of page frames in buffer RAM and DDR memory.*

With the exception of the Data Manager there will be only one processor per class. In order to obtain the required performance it will be necessary to implement multiple Data Managers. As host commands are received by the host processor they will be farmed out to multiple Data Managers. Even though there are multiple Data Managers, it is expected that they will all execute the same firmware image.

Each class of processor has a section within this specification that describes its functionality.

1.2.2 Ware-leveling

Due to the limited number of times that a flash device may be erased (between 10,000 and 100,000 times) before the integrity of the data written to the device cannot be guaranteed, it will be necessary to implement a ware-leveling algorithm. This algorithm will determine the next flash block to be written and manage the movement of data within the flash devices in order to prolong their life and ensure that they ware out at approximately the same rate. More details on the ware-leveling algorithm may be found in the Ware Leveling section of this document.

1.2.3 Message Network

Processors within the processor complex communicate with each other and with hardware blocks via the message network. Each processor has 15 inbound and 15 outbound message queues indexed as 1-15, inbound message queue index 0 is utilized by the memory network. If a processor is to receive messages from another processor it is the responsibility of the destination processor to populate the appropriate outbound message queue on the source processor with message buffers that lie within the destination processors local data memory. To send a message the source processor first obtains a message buffer pointer on the destination processor by reading the

appropriate outbound message queue. Values are then written to the message buffer utilizing customized instructions, the message buffer pointer is then written to the appropriate inbound message queue on the destination processor.

1.2.4 Memory Network

The processors within the processor complex do not have direct access to the Buffer RAM or the DDR memory. Instead, variable sized blocks of memory may be copied between Buffer RAM or DDR memory and a processors local data RAM utilizing customized instructions. Because the operation of moving data in to or out of a processors local data RAM is non-blocking, a method to signal that the data movement completed is required. Inbound message queue 0 on each processor will be used to receive data transfer completions. Each completion will contain a tag in the range of 1-32 which was allocated by the data move instruction, thus the firmware will be able to keep track of each memory move request.

1.2.5 RTOS Support

Should a Real Time Operating System be required by the firmware, the MicroC/OS-II real-time kernel has been ported to the Xtensa architecture, the source code for which is available in the Huron CVS repository under huron/software/Firmware/uCOS_II. An example application is also available under huron/software/Firmware/uCOSII_Demo.

1.2.6 Low Power Considerations

In general, the more instructions a CPU core executes, the more power it consumes. In order to conserve energy while idling, all firmware should make use of the Xtensa *waiti* instruction. This instruction halts the CPU until an interrupt is received. This will reduce the power consumption of the CPU cores and thus help to reduce the temperature of the device. The *waiti* instruction accepts a single parameter which is a value in the range of 0 - 15 that represents the interrupt level at which the instruction should be executed. With an interrupt level of 0, all interrupts will wake the core. If an interrupt level of 1 is specified, only interrupts with a priority higher than 1 will wake the core. For more information relating to the *waiti* instruction, refer to the Xtensa Instruction Set Architecture Reference Manual from Tensilica.

Control Manager

2.1 Overview

The Control Manager is responsible for the initialization of the system, and the loading and saving of the system state. In the future it may be modified to load firmware for the processors out of the flash memory, and to do logging for the system in case of a surprise shutdown. To perform these functions, it controls a section of NVRAM to save system state information, and a section of DRAM to store this information when the system is running so that other processors will have access to it.

A smaller responsibility for the control processor is to handle errors and interrupts for Delaware.

2.2 NVRAM Control

The NVRAM is split up into different partitions to handle different functions. It uses this data to save and restore the state of the memory.

2.2.1 Partitions

A single NVM device array may support up to 16 partitions, as determined by the size of the partition table array defined within the super block data structure.

The following partition types are currently defined:

Partition ID	Partition Name
0x00000000	Unused
0xDEADBEEF	Super Block Partition
0xBADBBADB	Bad Block Table Partition
0xBAD0A550	Firmware Partition (currently unused)
0xFEEDFACE	Block State Partition
0xDEAFBABE	Logical to Physical Mapping Partition
0xABFAB000	User Data Partition

Table 2.1 Partition Table Types

The Control processor has control over every partition save the user data partition. That is controlled by the wear leveling manager. There is no wear leveling performed on NVRAM dedicated to the control processor. The control processor will write in a circular fashion in each partition to do wear leveling without excessive computation. All partitions under the control processor are also called the **control partitions**.

The partitions shown in blue contain structures that are contained in a single page. The superblock contains basic device information, as well as the partitioning table. The ID's listed in Figure 2.1 are used in the partitioning table for identification. The bad block table partition contains the bad blocks in the control partitions only.

The partitions shown in green are file partitions, these partitions contain a single file. The block state partition contains bad block information, as well as access counts and other information used by the wear leveler. The logical to physical mapping partition contains the information needed to map from the logical address that the host sees to the physical address used by the flash controller.

2.2.1.1 Superblock Partition

The Super Block Partition constitutes up to the first two blocks of the control partitions, and is where the super-block structure resides. Firmware attempts to locate the Super Block by reading a page at a time from the Super Block Partitions looking for the Super Block Partition ID in the first DWord of the page data. Since there may be multiple copies of the Super Block stored in the Super Block Partition, a 32-bit version number, located in the second DWord of the page data and incremented each time a Super Block data structure is written, is used to determine which copy of the Super Block is the latest. Assuming that both the partition ID and the version number requirements are met the Super Block is deemed to be valid.

Since a flash page cannot be re-written without first erasing the block it belongs to, each time the Super Block is updated it will need to be written to the next available page in the Super Block Partition. When the Super Block is written to the first page of a block, the block must first be erased. This will allow for a history of Super Blocks to be maintained within the partition. If a failure occurs when writing the Super Block, the Super Block version number will be incremented, the block where the write failure occurred will be marked as bad in the bad block partition, and an attempt will be made to write the Super Block to the first page of the next block in the Super Block Partition.

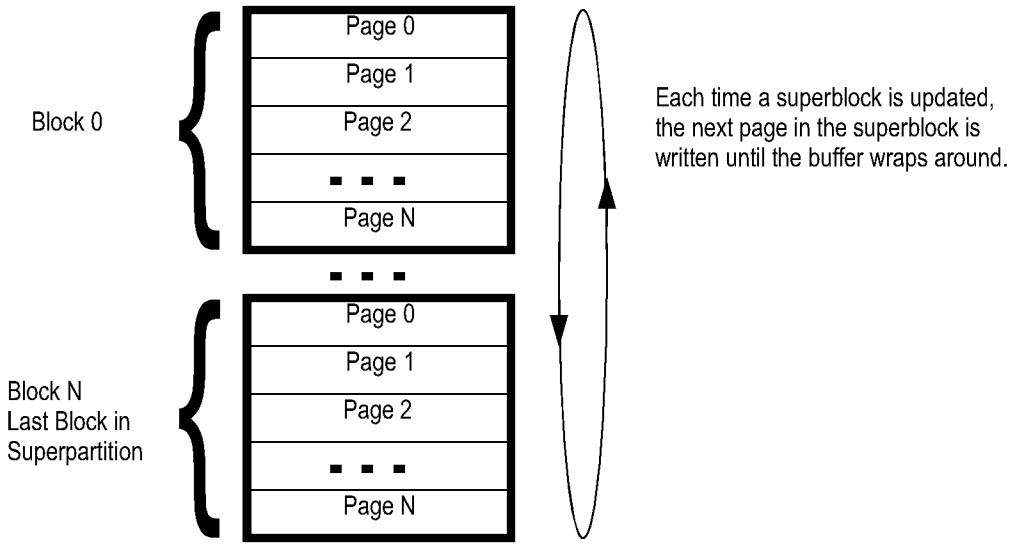


Figure 2.1 Superblock Partition Organization

2.2.1.2 Bad Block Table Partition

The bad block table partition consists of a single page. (this could be expanded on in the future) Each byte in the page corresponds to a block maintained by the control processor. (i.e. the control partition) This allows the control processor to quickly determine if a block is bad, or if a group of contiguous blocks are bad. When a new block is found to be bad, a new page is written, and the superblock is updated to point to it. This page will not be written very often as there are a maximum of 4K blocks that it can control with the current selected NVRAM.

.

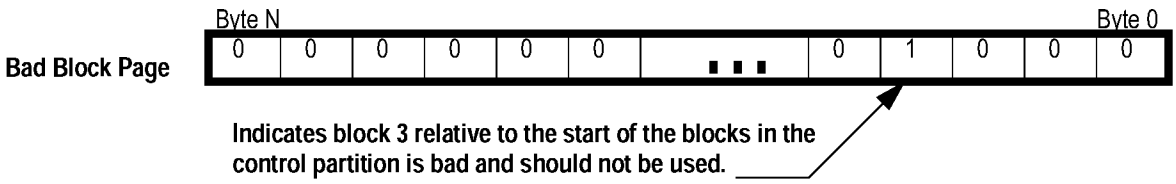


Figure 2.2 Bad Block Table Organization

2.2.1.3 File Partitions

The Firmware, block state, and logical to physical mapping partitions are stored as a single file within a partition.

The file to be written to this type of partition will be broken into page sized chunks, each chunk will begin with a File Header structure. The very first time that a file is written to the partition it will be written starting at the first page of the first block of the partition, as determined by the partition table. Before writing to the first page of each block, the block must first be erased. Once the first page of the file has been successfully written the head pointer in the partition table will be updated to point to this page. Each time a page is written its file header will be updated to point to the previous good page (a good page is a page that was written with no errors). Once the last page of the file has been written the partition table tail pointer is updated to point to the last page that constitutes the file. The figure below depicts the state of the partition after a file has been written with no errors.

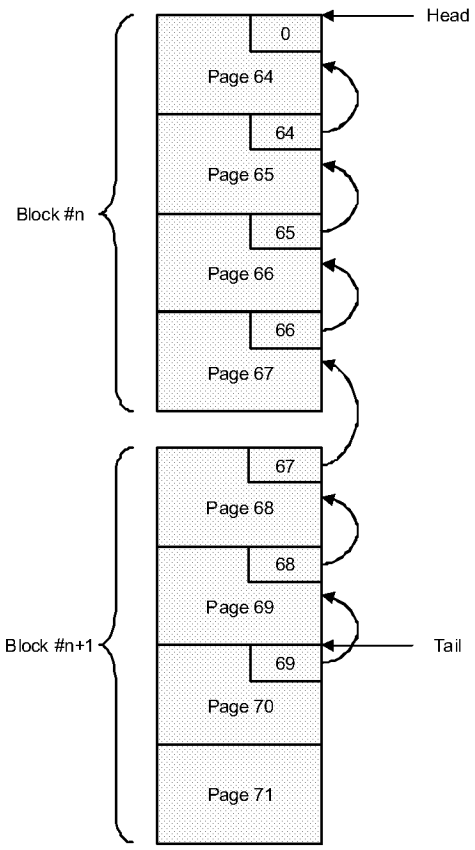


Figure 2.3 Partition Containing a File

Reading the file back from the partition is a simple case of reading the page pointed to by the tail pointer in the partition table. Once this page has been read, the information in its File Header structure will point to the previous page of the file, which is then read, and so on and so forth. This process continues until a File Header is located that points to page zero, this marks the beginning of the file. Due to the fact that when the file is read, it is done so in reverse page order, the File Header of each page contains an *offset* field that specifies the offset within the file where the data contained within each page resides. This enables the pages to be read in reverse order but for the image in memory to be organized correctly.

If an error occurs while writing a page to a block within a partition, the block number is added to the bad block table, any pages already written to the block will be moved to the next good block and writing of the file will continue. The figure below depicts that state of the partition if a page write error was detected while writing page 70. Pages 68 and 69 in block #n+1 will be moved to pages 72 and 73 in block #n+2, which will be followed by the page of data that reported the error. In this case, the pointers to the previous pages in the File Header must be updated to maintain the correct file structure before the moved pages are written to their new block.

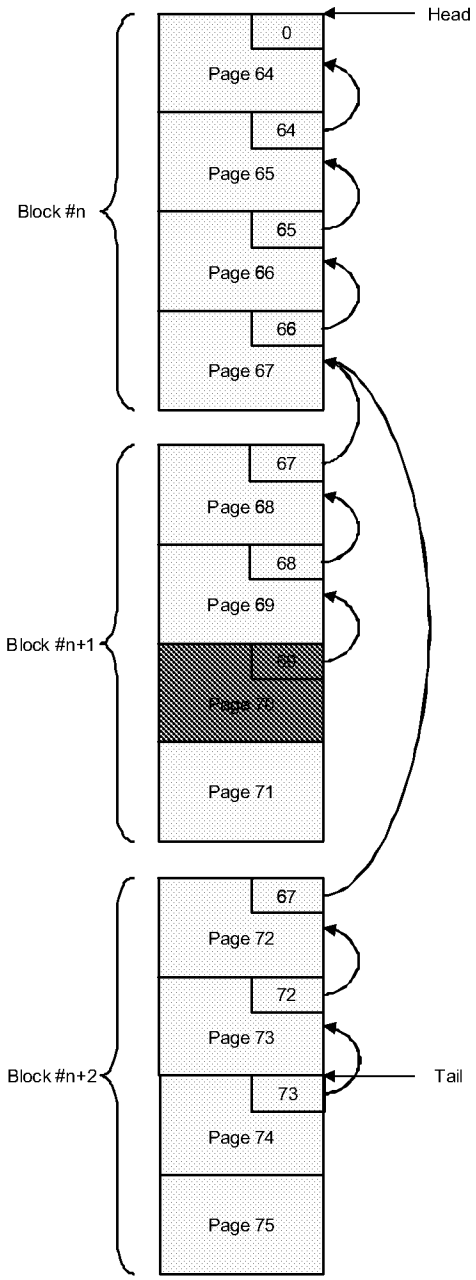


Figure 2.4 Page Write Error

The head page of a file must always be located on the first page of the next good block following the tail pointer. This allows the wraparound condition, which would occur if the file was too big to fit into the partition, to be checked for. Finally, because information is updated in the partition table each time the file is written to the partition, and due to the fact that the partition table is located in the Super Block, the Super Block must be saved every time a file is updated.

2.2.1.4 Delaware NVRAM Memory Map

Once the partitions are understood, the memory map can be produced based on the amount of storage required for each partition. Different memory options will require different partition sizes. The following is the memory available for the Delaware system:

NVRAM Used: Micron MT29F128G
Blocks: 32 K
Pages/Block 128
Page Size (bytes: 4 K

We are using 16 channels with 2 RAMs each totalling for the system:
Blocks: 1 M
Pages: 128 M
Total capacity: 500 M

Partition Requirements:
Superblock: 1 bock minimum
Bad Block Table: 1 block minimum
Block State: 64 bytes/block = 64 Mbytes minimum. (128 blocks)
Logical to Physical Map: 4 bytes/page = 500 Mbytes minimum (1 K blocks)

The minimum requirements assume no bad blocks in that partition. Also, the closer to the minimum we are, the shorter the lifetime of the product due to the unreliability of NVRAM after 10,000 writes. We also want to be able to keep a history for previous files saved. Therefore, the partitions will quadrouple the minimum for redundancy and reliability.

Address (bytes)	NVRAM Address Map
0x0000000000 - 0x00001fffff	Superblock Partition 4 blocks
0x0000200000 - 0x00003fffff	Bad Block Table Partition 4 blocks
0x0000400000 - 0x000fffffff	Block State Partition 512 block(-8 to make it line up better)
0x0010000000 - 0x007fffffff	Logical to Physical Map Partition 4 K blocks (-512 to makeit line up better)
0x0080000000 - 0x7fffffffff	User data partition

Table 2.2 NVRAM Memory Map

In the future we could make 1+1 redundant control partitions on a different channel for channel based redundancy, but that will not be done in the first revision of the product.

2.3 DRAM Control

The DRAM has a capacity of 1 GB, and holds much of the same information that is stored in the control partitions of NVRAM. It is the "live" copy when the controller is up and running. When the controller is shutdown the DRAM is copied back to the NVRAM partitions for storage until the next boot up.

The system is setup so that the low portion of the DRAM may be used for instruction fetch space for the processors. (see TBD for details) Therefore, the control partition that is copied over to the DRAM will be placed as high as possible in the address range. Like the NVRAM memory map, this information is stored in the superpartition table. An additional partition ID is created for this table for the data cache to and from the NVRAM:

Partition ID	Partition Name
0x00000000	Unused
0xDEADBEEF	Super Block Partition
0xFEEDFACE	Block State Partition
0xDEAFBABE	Logical to Physical Mapping Partition
0xABADCAFE	Data Cache Partition

Table 2.3 Partition Table Types used in DRAM

2.3.1 Memory Map

Address	DRAM partition
0x00000000 - 0x00000fff	Superblock / Global information (read only please)
0x00001000 - 0x1bffffff	Expansion Instruction RAM / General purpose cache
0x1c000000 - 0x1fffffff	Block State Information (handled by the wear level manager)
0x20000000 - 0x3fffffff	Logical to physical map (handled by the page manager)

Table 2.4 DRAM memory map

2.4 Startup Sequence

Upon de-assertion of the system reset, the control manager will begin executing instructions out of it's pre-loaded instruction RAM. As Delaware is an FPGA, the instruction RAM was loaded with the Xilinx bit stream. Therefore, in this embodiment of the design the firmware for the control manager may not be updated save by changing the eprom holding the Xilinx code. The definition of a flash firmware partition will be maintained, however, in the event that a eprom boot loader or upgrade feature is desired at a later point in time.

The control processor will then scan the flash to determine if it has been initialized previously. If so, it will load the global information into the DRAM for other processors to use, and then indicate to the other processors through the sync message that the memory has completed loading and is good to use. If the memory has not been previously initialized, it will write a bad block table and superblock, and then indicate to the other managers that they must initialize the block state and logical to physical information. The outline below demonstrates the steps taken at startup time.

1. Reset de-asserted, control processor begins the boot process
2. All other processors power up in the "halt" state.
3. Initialize message and data network connectivity to the FCC, BRAM, and DRAM.
4. Query FCC to see what RAMs/channels are installed.
5. Run FCC initialization sequence to get proper RAM timing.
6. Scan superblock partition for latest valid superblock. If none is found:
   - Create superblock. Set the partition pointers for the Logical to physical, the block status, and firmware partition pointers to null as they are not yet valid.
   - Scan the control partition for the bad blocks, and write out the bad block table.
   - The control processor deals with physical addresses only, so it addresses the FCC directly.

7. Copy all valid information (non null partitions) to DRAM. If the logical to physical map partition is not valid, then zero out that section of DRAM.
8. If firmware partitions is valid (not null) copy to instruction RAM of target processors.
9. Start up other processors.
10. Initialize processor interfaces that the control processor interfaces with.
11. Send Power Up Message to other processors to bring them out of a quiessed state.
12. Write the superblock to indicate that the system state is open. This is used to detect an improper shutdown.
13. If this is the first initialization, power down all processors after then come up and save the system snapshot, and bring them back on again.
14. wait for interrupt.
    (Alternatively, I could start a watchdog timer to prevent the tensilica "time bomb" from going off.)

The actual data structures used are described in section TBD

2.4.1 Scanning for a valid boot block

Firmware attempts to locate the super block by reading a page at a time from the super block partition in sequential order looking for the Super Block Partition ID in the first DWord of the page data. Since there may be multiple copies of the super block stored in the Super Block Partition, a 32-bit version number, located in the second DWord of the page data and incremented each time a super block data structure is written, is used to determine which copy of the super block is the latest. Assuming that both the partition ID and the version number requirements are met the super block is deemed to be valid.

Since a flash page cannot be re-written without first erasing the block it belongs to, each time the super block is updated it will need to be written to the next available page in the Super Block Partition. When the super block is written to the first page of a block, the block must first be erased. This will allow for a history of Super Blocks to be maintained within the partition.

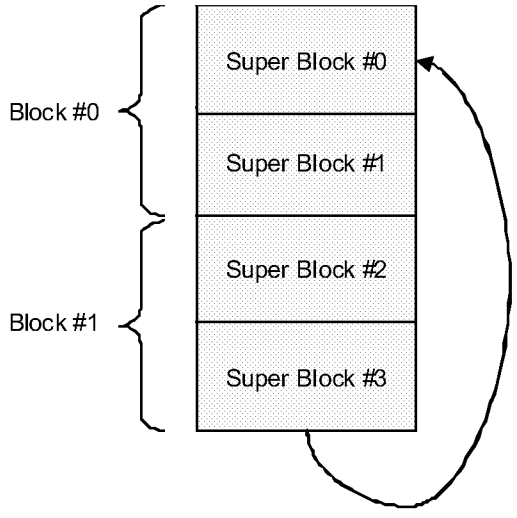


Figure 2.5 Super Block Partition Layout

2.4.2 Bad Block Scanning and Determination

NAND flash devices are shipped from the factory erased. The factory identifies invalid blocks before shipping by programming data other than FFh into a well known location as shown in Table 2.5. The factory marked bad blocks are discovered and their addresses are saved in the Bad Block Partition during initial system initialization. The bad blocks must not be used to store any data to avoid any data corruption.

In order to identify all the Bad Blocks during the initial system initialization, all blocks in the system must be scanned the first time the firmware runs to determine if a block is bad or not. The control processor will scan all blocks in the control partitions, and the wear level manager will scan the rest. The scanning algorithm is shown in Figure 2.6. In this example, ONFI 2.0 devices are used.

```
for (block = 0; block < TOTAL_BLOCKS; block++)
{
    Get_First_Page_Spare_Data(&data[0]);
    if (data[0] == '\0')
    {
        Log_Bad_Block(block);
    }
}
Sv_Bad_Block_To_Bad_Block_Partition()
```

Figure 2.6 Factory Bad Block Scanning Algorithm

Vendor	Bad Block Marker
Micron	First spare byte in the first page of the each block is 00h
Samsung	First spare byte in first or second page of a block is non-FFh
ONFI 1.0, 2.0	At least one byte in the spare area of the first or last page of a block is 00h
Numonvx	First spare byte in the last page of a block is non-FFh

Table 2.5 Factory Bad Block Marker

2.5 Shutdown Sequence

The shutdown sequence is used to park the drive in a good state before the power is removed. Mostly it just copies data from DRAM to non volatile storage.

1. Power down interrupt, or time bomb timer interrupt comes in. Process begins.
2. Control processor sends a message to all other processors to indicate that they are to finish critical work, flush any cached information to NVRAM, and then stop.
3. When all processors have stopped, the control processor copies the appropriate data from the DRAM to the NVRAM under it's control one block at a time.
   - The memory (if any) to be copied to NVRAM is indicated with a message from the page manager and the wear level manager.
   - The bad block table is used to prevent any faulty blocks from being used.
   - If at any time a bad block is found during this process, the bad block table is updated, and the next good block is used.

4. Superblock is updated to indicate a proper shutdown was completed.
5. Upstream interrupt is generated to signal to the host that it is ok to power down. (how? Could be set an LED indication as well?)

2.6 Interrupts and Error Handling

The Control processor is the aggregation point for all other processors in the system. If another processor indicates an interrupt, the control processor will record an interrupt vector and forward that on to the host.

2.6.1 Control Partition Error recovery

In the case of improper shutdown or data corruption, several mechanisms are employed to recover the state of the control partition to the extent possible. An error condition is signaled when on power up:

- There is no valid superblock.
  The firmware will determine if this is due to a factory erased condition. If the NVRAM is determined to be factory erased then this is not an error condition, and initialization will proceed.
- The state of the last valid superblock indicates the shutdown was not complete.
- The last superblock has a CRC or other sanity check error condition.

If any of these error conditions are present when the system comes out of reset, the control manager will place the chip in an error state. This will prevent reads and writes from going through, but will allow the acceptance of NVNHCI commands. Commands must come through from the host driver to direct the control manager on what steps to take next. Possible commands are:

1. Factory erase and initialize. Bad blocks will be re-determined with a fresh bad block scan, and all other information will be lost/reformatted.
2. Re-format from last known good state. The last known good state will be used for bad block determination and statistics. The blocks will be erased, and the address mapping reformatted.

PCIe Manager

3.1 Overview

The PCIe Manager is responsible for managing the PCIe endpoint configuration and memory mapped I/O space, and the interface to the host, i.e. eNVMHCI.

Data Manager

4.1 Overview

The Data Manager manages eNVMHCI read and write requests from the host. Command requests are received as CmdReady messages from the PCIe Manager. The CmdReady message consists of bits 36:5 of a 37 bit address that points to an eNVMHCI command buffer located in buffer RAM. The Data Manager first requests that the new command is copied from buffer RAM to local data RAM so that it can be processed. Processing of the command begins once the completion is received signaling that the command has been copied. The command is then converted from a request consisting of a starting sector and a sector count to a starting flash page and a page count. The Data Manager then issues requests to lock the Page Table Entry (PTE) associated with each page and to allocate a page frame buffer. Once the completions to these requests have been received page request processing may continue. This is where the data flows for read and write requests differ. In the case of a page read, a FlashRead message is sent to the Flash Controller (FC). In the case of a page write, an IngressDMA message is sent to the IDMA to transfer the write data from host memory to the page frame buffer associated with the page. Once the DMACpl message is received signaling that the DMA transfer has completed, a FlashWrite message is sent to the Write Manager. In both cases a FlashOpCpl message is sent to the Data Manager from the FC to signal that the flash operation completed. At this point, for page write requests, the operation is complete and the PTE for the page is unlocked. For page read requests, the data placed in the page frame buffer associated with the page must be transferred to host memory. The Data Manager sends an EgressDMA message to the EDMA in order to copy the data from the page frame buffer to host memory. A DMACpl message is received by the Data Manager when the transfer completes, this signals the end of the page read request and the PTE entry associated with the page is unlocked. Once all of the page requests associated with a command have completed, the command itself is completed by sending a StsQueueUpdate message to the PCIe Manager.

If the FC reported a non-fatal error in the completion to a write request the Data Manager may re-issue the request up to *n* times before completing the page write request. If after *n* retries the request still fails, an error will be reported to the PCIe Manager upon completion of the command.

- What errors can the Flash Controller report???
- Fatal write errors - Channel/target could not be accessed???
- Non-fatal write errors - Page write error???
- Fatal read errors - Uncorrectable ECC errors, channel/target could not be accessed???
- Non-fatal read errors - ???

4.2 Architecture

In order to utilize as many CPU cycles as possible, all requests issued to other managers and functional blocks will be non-blocking. Replies to these requests will be received asynchronously at some later point in time via the inbound message queues.

To facilitate this, the Data Manager will consist of two state machines. One will be used to keep track of individual eNVMHCI commands and the other the flash page requests that constitute the command. The Data Manager will be capable of managing multiple page requests and multiple eNVMHCI commands at a time. Each eNVMHCI command and page request will utilize a Command State Structure or a Page State Structure, respectivly. The number of in-flight commands and page requests is limited only by the number of state structures allocated to each state machine.

The basic premise is this, certain structures and message buffers are treated as resources. Resources may be allocated and released. If a resource can not be allocated the a request is saved to the head of the resources pending list. When a resource is released, its pending list is checked to see pending requests need processing.

```
typedef struct {
        LIST_HEAD           list;
        CMD_STATE           state;
        CMD_STATUS          status;
        NVM_COMMAND         command;
        u64                 commandBufferAddr;
        u32                 currentPage;
        u32                 totalPages;
        u32                 completedPages;
        u32                 numSectors;
        u64                 startSector;
} COMMAND_STATE;
```

Figure 4.1 Command State Structure

The *list* field in the Command State Structure is used to enqueue the structure to different lists at runtime. The *command* field defines a buffer where the eNVMHCI command is placed when it is copied from buffer RAM. The *commandBufferAddr* field contains the 64-bit address of the eNVMHCI command in buffer RAM. The *state* and *status* fields are used by the state machine to keep track of command processing, the *currentPage*, *totalPages*, and *completedPages* fields are used to keep track of page operation requests and completions, and the *numSectors* and *startSector* are in page operation requests and valid sector bitmask generation.

```
typedef struct {
        LIST_HEAD           list;
        PG_STATE            state;
        PG_STATUS           status;
        u32                 retries;
        u32                 used;
        u32                 index;
        u32                 validSectors;
        u32                 offset;
        u32                 size;
        u32                 logicalPageAddr;
        L2P                 pageTableEntry;
        u32                 pageFrameAddr;
        u32                 metadataAddr;
        u32                 operation;
        COMMAND_STATE       *commandState;
} PAGE_STATE;
```

Figure 4.2 Page State Structure

The *list* field in the Page State Structure is used to enqueue the structure to different lists at runtime. The *state* and *status* fields are used by the state machine to keep track of the exact state of the page. The *index* field holds the index of a page request so that completions may be associated with a PRDI entry. The *validSectors* field is a 32-bit bitmask that is used to determine which sectors within the page are valid. The *logicalPageAddr* field is used to maintain a copy of the logical page address. The *pageTableEntry* field is used to maintain a copy of the page lookup table entry for this page. The *pageFrameField* is a 64-bit pointer to the page frame buffer for this page. The *metadataAddr* field is a 64-bit pointer to the meta-data buffer for this page. The *operation* field is used to specify if this is a read or write operation, etc, and the *commandState* field is a pointer back to the Command State Structure for this page operation.

The Data Manager will declare two static arrays, one for the Command State Structures and one for the Page State Structures. There must be at least as many Page State Structures as Command State Structures, if not more. The arrays are initialized to 0 and each array element is added to either the free command list or the free page list. The Data Manager then polls its inbound queues waiting for messages from other managers. Figure 4.3 depicts the relationships between the data structures utilized by the Data Manager.

- What about dynamic allocation of structures? Can we allocate some upfront and then add more if we run out, need to test the memory allocation routines from Tensilica?

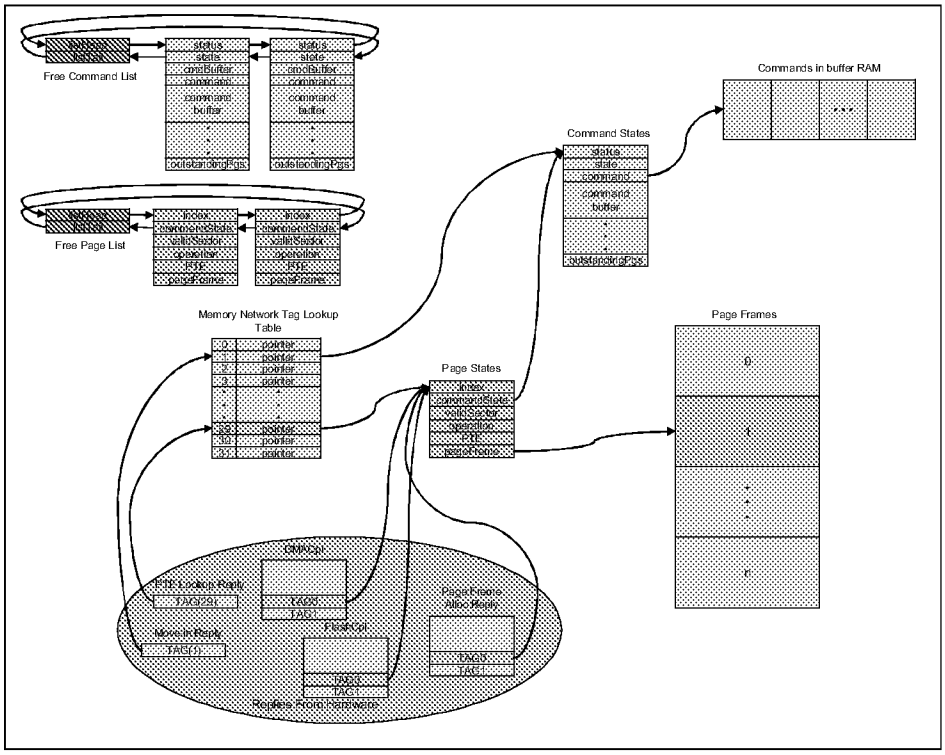


Figure 4.3 Data Structure Relationships

4.3 Command Initiation

An eNVMHCI command is initiated when a CmdReady messages is sent to the Data Manager by the Host Controller. The CmdReady message contains bits 36:5 of a 37 bit address that points to the location of the eNVMHCI command in buffer RAM. The Data Manager obtains a Command State Structure from the free command list. If the free command list is empty, the inbound message queue from the Host Controller should be masked and the CmdReady message should be queued at the head of a command state structure pending list. Otherwise the CmdReady message is freed and command processing continues.

To avoid the copying of data and the dynamic allocation of memory when requests are added to the pending list, each inbound message buffers is contained within a larger structure which includes a **LIST_HEAD** structure. The **LIST_HEAD** structure associated with the message buffer is accessed using the *container_of* 'C' macro.

```
typedef struct {
        LIST_HEAD           list;
        u8                  buffer[MESSAGE_SIZE];
} MESSAGE;
```

Figure 4.4 Inbound Message Buffer Structure

4.4 Releasing a Command State Structure

Each time a Command State Structure is released (added to the free list), the command state structure pending list is checked to see if there are any pending requests. If there are pending requests, a request is removed from the tail of the pending list and the Command Initiation procedure commences. At this point, if the pending command list is empty, the inbound queue from the Host Controller should be unmasked.

4.5 Requesting the Command

Once a Command State Structure has been obtained, the eNVMHCI command must be copied from buffer RAM to local data RAM. A *move-in* instruction is executed to request that data be copied from buffer RAM to local data RAM. The *move-in* instruction returns a tag, in the range of 0-31, which is used to index into an array of Memory Network Tag Structures. The *type* field in the Memory Network Tag Structure indexed should be set to **COMMAND_STATE** and the *tag* field set to point to the Command State Structure for the command. The command state in the Command State Structure should then be changed to **WAIT_FOR_COMMAND**. The Data Manager then resumes polling the inbound message queues for messages from the other managers.

```
typedef struct {
        MEM_REQ_TYPE        type;
        void                *tag;
} MEM_TAG;
```

Figure 4.5 Memory Network Tag Structure

The *type* field in the Memory Network Tag Structure is used to specify the type of the structure that the *tag* field points to. Figure 4.6 lists the supported tag types.

```
typedef enum {
        LOOKUP_ENTRY,
        MOVE_COMMAND
} MEM_REQ_TYPE;
```

Figure 4.6 Memory Network Request Types

4.6 Obtaining the Command

At some later point in time the Data Processor will receive a completion to the *move-in* instruction via inbound message queue 0. The 32-bit value read from inbound message queue 0 will contain a status, and a tag, in the range of 0-31, which should be used to index back into the array of Memory Network Tag Structures. Based on the *type* field in the Memory Network Tag Structure the Data Manager knows that the *tag* field is a pointer to a Command State Structure. If the status reports that the *move-in* instruction failed, the *move-in* instruction is executed again and a new tag obtained. The information from the previous Memory Network Tag Structure should be moved to the Memory Network Tag Structure indexed by the new tag. The Data Manager then continues to poll its inbound message queues. If the status reports that the *move-in* instruction completed successfully, the *state* field in the Command State Structure should be changed to **PROCESSING_COMMAND**, the buffer RAM command buffer is released by sending a CmdBufFree message to the Host Controller, and command processing commences.

4.7 Command Processing

The first step in command processing is to initialize the following Command State Structure fields with appropriate values.

- *startSector - Starting sector number from eNVMHCI command*

- *numSectors - Sector count from eNVHMCI command*
- *currentPage - Current page index being processed, initialized to 0*
- *totalPages - Populated with the total page count, determined once all page requests have been issued*
- *completedPages - Initialized to 0, incremented each time a page request is completed*

Once the Command State Structure is initialized, Page Processing may begin.

4.8 Page Processing

For each page in the eNVMHCI command the Data Manager will attempt to obtain a freePage State Structure. If no Page State Structures are available, the Command State Structure for the command will be queued to the page state structure pending list and the Data Manager will once again begin polling its inbound message queues. If a Page State Structure was obtained, the Data Manager issues a request for the page lookup table entry, sets the *state* field in the Page State Structure to **WAITING_FOR_MESSAGE**, then issues a request for a page frame. The request for the page lookup table entry cannot fail since the instruction will block if no tags are available. The request for a page frame may fail if there are no message buffers available to the Page Frame Manager. In which case the Page State Structure will be queued at the head of the page frame pending request list, and the interrupt for the free buffer queue from the Page Frame Manager will be enabled. When a buffer to the Page Frame Manager becomes available an interrupt will be generated, the ISR will check to see if the interrupt was generated by the Page Frame Manager releasing buffers, and if so, will remove a Page State Structure from the tail of the page frame pending request list and issue a Page Frame Alloc Request message to the Page Frame Manager. This process is repeated until there are no more pending page frame requests, in which case the interrupt from the Page Frame Manager free queue is masked, or there are no more buffers available to the Page Frame Manager.

When a request is made for the page lookup table entry, the tag returned from the *move-in-lock* instruction is used to index into the Memory Network Tag Structure array, the *type* field is set to **LOOKUP_ENTRY** and the *tag* field is set to the address of the Page State Structure. When the memory network reply is received the tag in the reply is used to lookup the appropriate entry in the Memory Network Tag Structure array. If the lock was obtained, the **LOCK_OBTAINED** bit is set in the *status* field of the Page State Structure.

When a Page Frame Alloc Reply message is received and a page frame was allocated, *TAG0* of the message points to the Page State Structure, the Data Manager uses this pointer to set the **PAGE_FRAME_OBTAINED** bit in the *status* field. If a page frame could not be allocated the page frame allocation message is retried, if a message buffer could not be obtained, the Page State Structure will be queued at the head of the page frame pending request list, and the interrupt for the free buffer queue from the Page Frame Manager will be enabled.

When a Page Frame Alloc Reply message or a memory network completion is received the status field in the Page State Structure is compared to (**PAGE_FRAME_OBTAINED | LOCK_OBTAINED**), if the comparison is true the page is ready for the next stage in processing, if the comparison is false, the page is still waiting for a lock or a page frame. What happens next depends on whether this is a read or write page request, each operation is described in the sections that follow.

```
u32 logPage;
u32 validSectors;
u32 startBit;
u32 numBits;

while(cmdState->numSectors)
{
        obtain_page_state_structure();
        if(no_page_state_structure)
        {
                queue_command_state_structure_on_pending_list();
                return;
        }
```

```
                startBit = cmdState->startSector & SECTORS_PER_PAGE_MASK;
                numBits   =   __minu(cmdState->numSectors,   SECTORS_PER_PAGE   -
startBit);

                validSectors = generateBitMask(startBit, numBits);
                index = cmdState->currentPage++;
                logPage = sector2page(cmdState->startSector);

                initialize_page_state_structure();

                request_PTE(); // this cannot fail
                request_page_frame();
                if(request_failed)
                {
                        queue_page_state_structure_on_free_buffer_list();
                        unmask_free_message_irq();
                        return;
                }

                cmdState->numSectors -= numBits;
                cmdState->startSector += numBits;
        }
        cmdState->totalPages = cmdState->currentPage;
```

Figure 4.7 Pseudo Code to Process Page Requests

Some of the pseudo code functions utilized in Figure 4.7 such as generateBitMask() may need be optimized to custom instructions.

4.8.1 Read Page Processing

When a page lookup table entry has been locked and a page frame obtained for a page read operation, the Data Manager must then change the *state* field of the Page State Structure to **WAIT_FOR_FC** and issue a FlashRead message to the Flash Controller in order to read the page data from flash and write it to the page frame/ meta-data buffers supplied in the FlashRead message. The *TAG0* field in the FlashRead message will be set to the address of the Page State Structure so that it can be located when the FlashOpCpl message is received.

If a message buffer to the Flash Controller could not be obtained, the Page State Structure for the page request is queued at the head of the flash controller pending queue and the interrupt for the free queue to the Flash Controller is unmasked. When buffers to the Flash Controller are available an interrupt will be generated. The ISR will remove queued flash requests from the tail of the flash controller pending list and issue FlashRead/FlashWrite messages to the Flash Controller until there are no more pending requests for no more message buffers. In the case of no more pending requests the interrupt is masked.

When the FlashOpCpl message is received, the *TAG0* field is used to access the Page State Structure. From the Page State Structure the Command State Structure can be accessed which contains a copy of the eNVMHCI command. The eNVMHCI command will contain the PRD/PRDI entries required to transfer the data from the page frame buffer to host memory.

To perform the transfer of data to host memory the Data Manager issues an EgressDMA message and the state field in the Page State Structure is changed to **WAIT_FOR_DMA**. The *TAG0* field in the EgressDMA message is set to the address of the Page State Structure so that it can be located when the DMACpl message is received. The case of no message buffers being available to send the message to the EDMA is handled in the same manner as when a message buffer could not be obtained to send a message to the Flash Controller.

Assuming a successful completion is received from the EDMA, the page lookup table will be unlocked, the page frame will be released, the *completedPages* field in the Command State Structure will be incremented, and the Page State Structure will be released. If the *totalPages* field is non-zero it will be compared to the *completedPages* field. If they are equal, all pages for this command have been completed, the Command State Structure is released and a StsQueueUpdate message will be sent to the PCIe Manager.

If the EDMA reported an error the request will be retried.

4.8.2 Write Page Processing

Write page processing is very similar to read page processing, except that data is moved from host memory to the page frame/meta-data buffers and then a request is sent to the Flash Controller to write the flash page. All other error processing and resource management is handled the same.

4.9 Releasing a Page State Structure

Each time a Page State Structure is released (added to the free list), the page state structure pending list is checked to see if there are commands pending on Page State Structures. If there are pending commands, a Command State Structure is removed from the tail of the list, Page Processing then continues for this command.

4.10 Inbound Message Queues

The Data Manager requires inbound message queues to receive messages from the following managers and functional blocks.

- *PCIe Manager*
- *Ingress DMA (IDMA)*
- *Egress DMA (EDMA)*
- *Flash Controller*
- *Page Frame Manager*
- *Memory Network*

4.10.1 Inbound Messages

The following messages are received from other managers and functional blocks by the Data Manager. The exact format of these message is described in the IPC Messages section of this document.

- *CmdReady - Received from the PCIe Manager when a new command is available for processing in the buffer RAM.*
- *DMACpl - Received from either the IDMA or the EDMA to signal completion of a DMA transfer.*
- *FlashOpCpl - Received from the FC to signal completion of a flash operation.*
- *MemNtkwCpl - Received from the memory network to signal completion of a data move-in/move-out instruction.*
- *Page Frame Alloc Reply - Sent from the Page Frame Manager in response to a Page Frame Alloc Request.*
- *StsQueueUpdateCpl - Received from the PCIe Manager to signal success or failure of a StsQueueUp-*

date message.

4.11 Outbound Message Queues

The Data Manager will be required to send messages to the following managers and functional blocks.

- *Ingress DMA (IDMA)*
- *Egress DMA (EDMA)*
- *Flash Controller*
- *Write Manager*
- *PCIe Manager*
- *Wearleveling Manager*
- *Page Frame Manager*

4.11.1 Outbound Messages

The following messages are sent to other managers and functional blocks by the Data Manager. The exact format of these message is described in the IPC Messages section of this document.

- *IngressDMA - Sent to the IDMA to request that data be moved from host memory to the buffer RAM or DDR memory.*
- *EgressDMA - Sent to the EDMA to request that data be moved from the buffer RAM or DDR memory to host memory.*
- *FlashRead - Sent to the FC to request that a flash page be read.*
- *Page Write - Sent to the Write Manager to request that a flash page be written.*
- *CmdBufFree - Sent to the buffer RAM to release a command buffer.*
- *Page Frame Alloc Request - Sent to the Page Frame Manager to request that a page frame be allocated.*
- *StsQueueUpdate - Sent to the PCIe Manager to signal completion of a command.*

Write Manager

5.1 Overview

The Write Manager keeps track of an open flash block for each flash target. These blocks are requested from the Wearleveling Manager as needed by means of Block Open Request and Block Open Reply messages. When Page Write messages are received from other managers, the Write Manager assigns a physical flash page address to the request from one of its open blocks, forwards the request on to the flash controller, and sends an Obsolete Page message to the Wearleveling Manager informing it of the physical flash page that was written and the physical page, if any, that was obsoleted by this write. As open blocks are filled, the Write Manager sends a message to the Wearleveling Manager to close the block and request a new one for the specified target. The maximum number of open blocks maintained by the Write Manager is equal to the maximum number of supported flash targets.

5.2 Architecture

The Write Manager maintains an array of Block State Structures, one array element per flash target. Initially, all block state structures will have their *state* fields set to **IDLE**. While in the **IDLE** state the block state structures do not contain valid block numbers and their pending operations list is empty. During initialization the Write Manager will send a Block Open Request message to the Wearleveling Manager for each supported target to request a new block, and set the *state* field in each Block State Structure to **REQUESTING**.

```
typedef struct {
        BLK_STATE           state;
        u32                 skipValue;
        u32                 skipCount;
        u32                 blockAddr;
        u32                 pageAddr;
        LIST_HEAD           pending;
} BLOCK_STATE;
```

Figure 5.1 Block State Structure

The Write Manager uses an index counter, which is initialized to zero, to determine which flash block (array element) should be used for the next flash write operation. Each time a flash write is issue to the Write Manager, the index counter is incremented, if the index counter >= the number of flash targets it is reset back to zero. This ensures that data is spread evenly across all flash targets.

If a Page Write request is received and the selected block state is **REQUESTING**, a request for a new block has been issued for this block but the response has not yet been received. In this case the request should be added to the head of the pending list for the selected block. When a Block Open Reply is received, the block state is changed to **OPEN** and any requests that are pending for the block will be issued from the tail of the pending list.

To avoid the copying of data and the dynamic allocation of memory when requests are added to the pending list of the Block State Structure, each inbound message buffers is contained within a larger structure which includes a **LIST_HEAD** structure. The **LIST_HEAD** structure associated with the message buffer is accessed using the *container_of* 'C' macro.

```
typedef struct {
        u8                  buffer[MESSAGE_SIZE];
        LIST_HEAD           list;
} MESSAGE;
```

Figure 5.2 Inbound Message Buffer Structure

When creating a FlashWrite message, the physical flash page address is created by logically OR'ing the *blockAddr* and the *pageAddr* fields from the Block State Structure, this value should then be written to the *FlashAddr* and *TAG1* fields of the message. All other fields in the FlashWrite message should be copied from the received Page Write message. The Write Manager must also send an Obsolete Page message to the Wearleveling Manager to flag that the new page was written and that a page was possibly obsoleted by this write operation. The *TAG1* field of the Page Write message is used to communicate the obsoleted page address and a status used to determine if the page is being obsoleted due to a page write error. If the *TAG1* field is 0xffffffff, there is no obsolete page. If the TAG1 field != 0xffffffff, bits 30:0 represent the address of the flash page being obsoleted and bit 31, if set, specifies that the page is being obsoleted because of a page write failure.

- Need to add the node ID and queue index where the flash controller should send the completion to the FlashWrite message.

The *pageAddr* field in the Block State Structure is then incremented, if *pageAddr* >= the number of pages per block, the block state is changed to **REQUESTING** and a Block Open Request is sent to the Wearleveling Manager to close the current block and to open a new block.

The Wearleveling Manager is able to specify a skip count in the Block Open Reply that it sends. The skip count is used to force the Write Manager to skip a particular block every n counts. For example, if the Wearleveling Manager specifies a skip count of 2, the block will be skipped after every two write requests. This helps the Wearleveling Manager to balance flash write requests across all flash devices. A value of 0 in the skip count specifies that the block should not be skipped, and a value of 63 specifies that the block should always be skipped.

Block states:

- *IDLE*
- *OPEN*
- *REQUESTING*

5.3 Performance Considerations

Under heavy traffic the Write Manager will be issuing requests to open new blocks in bursts due to the spraying algorithm used to keep all targets busy. To avoid a stall, the Wearleveling Manager must be able to handle bursts of block open requests. The number of requests in each burst will be equal to the number of supported targets.

To avoid impacting the write performance, it may be necessary to double buffer each target block structure. When a block is closed and a new one opened the Write Manager will then be able to use the buffered target block structure instead of queuing requests until the Block Open Reply is received. This will double the number of open blocks at any one time.

5.4 Inbound Message Queues

The Write Manager requires inbound message queues to receive messages from the following managers and functional blocks.

- *Data Manager*
- *Wearleveling Manager*

5.4.1 Inbound Messages

The following messages are received from other managers and functional blocks by the Write Manager. The exact format of these message is described in the IPC Messages section of this document.

- *Page Write - Received from the Data Manager or the Wearleveling Manager when a flash page is to be written.*

- *Block Open Reply - Received from the Wearleveling Manager in response to a Block Open Request.*

5.5 Outbound Message Queues

The Write Manager will be required to send messages to the following managers and functional blocks.

- *Flash Controller*
- *Wearleveling Manager*

5.5.1 Outbound Messages

The following messages are sent to other managers and functional blocks by the Write Manager. The exact format of these message is described in the IPC Messages section of this document.

- *FlashWrite - Sent to the FC to request that a flash page be written.*
- *Block Open Request - Sent to the Wearleveling Manager to open a new flash block.*
- *Block Close - Sent to the Wearleveling Manager to close a block when it is full.*
- *Obsolete Page - Sent to the Wearleveling Manager to obsolete a page.*

Wearleveling Manager

6.1 Overview

NAND flash memory is organized as units of blocks. There are multiple pages in a block. A block is an erase-able unit. Data is accessed in the unit of a page. For the Micron MT29F128G08 flash device, the page size is 4 KBytes, there are 128 pages in a block and there are 32K blocks. There are 16 flash channels and 2 flash devices in Delaware for a total of 512TBytes of storage capacity.

A logical to physical mapping table is used to map logical block address as request by the host to the physical block address of the flash devices. Data is not updated in place but written to a new page. Old data is invalidated and becomes obsolete. After data are written to the flash for some time, some of the pages within a block may contain obsolete data. The obsolete data must be cleaned up at regular interval. Pages that contain valid data within a block are moved to new Flash block such that the block can be erased. After erasure, the block is ready to be written again. The process to reclaim the obsolete pages is called garbage collection.

A block (SLC flash) can be typically erased 100,000 times before it becomes unreliable and cannot be used to store any more data. Garbage collection prefers to reclaim those blocks with the least number of pages that contain valid data to reduce the data copy traffic to a minimum. This causes some of the flash blocks to be erased more frequent than others. The overall lifespan of a flash device could be shorten because of uneven wear on the flash device. Wear leveling refers to the process to prevent blocks from being unevenly erased (or worn) so as to prolong the life of a flash device.

Delaware supports 16 flash channels and 8 targets per channel. Wear leveling has to ensure even wear on all targets with a channel and across channels. It has to manage the allocation of next block to be used for data storage, decide when to invoke garbage collection and which block to be reclaimed next in order to wear all the devices as evenly as possible. It also needs to minimize the number of pages that have to be moved in order to reclaim flash blocks.

The Wearleveling Manager keeps the state of each block in the system in order to manage the flash devices. When a bad block is discovered during the initial bad block scan or during normal usage, the state of the block is marked as bad. When a block is marked as bad, it is never be used again.

The Wearleveling manager is responsible for the wear leveling, garbage collection and bad block management.

6.2 Wear Leveling

During the initial system initialization, all flash blocks are erased and ready to be programmed. They are queued in the FreeList. A free flash block is allocated from the FreeList for the first write request. This block is known as the ActiveBlock. Page must be written from page 0 to the highest page number in ascending order. i.e page within a block must be written in the order of page 0, 1, 2, 3, ...etc. When all the pages are written in a block, the block is added to the DirtyList and a new ActiveBlock is allocated from the FreeList.

When the number of entries in the FreeList falls below a low water mark threshold, garbage collector is activated to reclaim dirty blocks and put them back to the FreeList. The garbage collector does not stop until the number of entries in the FreeList reaches a high water mark threshold.

The queue structure is shown in Figure 6.1. The Queue Manager manages the FreeList and DirtyList queues. It adds entries into the FreeList after a block has been erased and ready to be programmed again. When all the pages of the ActiveBlock are written, the dirty block is added to the DirtyList. When a new ActiveBlock is needed, the Allocator removes an entry from the head of the FreeList and this entry becomes the new ActiveBlock. Whenever the Garbage Collector is activated, it removes the first entry from the DirtyList and reclaim the dirty pages. It first copies all valid pages in this dirty block into new pages in the ActiveBlock. It then erases the dirty block and adds it back to the FreeList. Additional free block is created. It should be noted that the garbage collector may need to clean up multiple dirty entries in order to reclaim a single free block. For instance, if all the pages in the dirty entry

are valid pages, then a new ActiveBlock is required to store all the valid pages during the page copy. A new free block is created after the erasure but a new ActiveBlock is used for store the valid pages. The net gain is zero as no new additional free block is created.

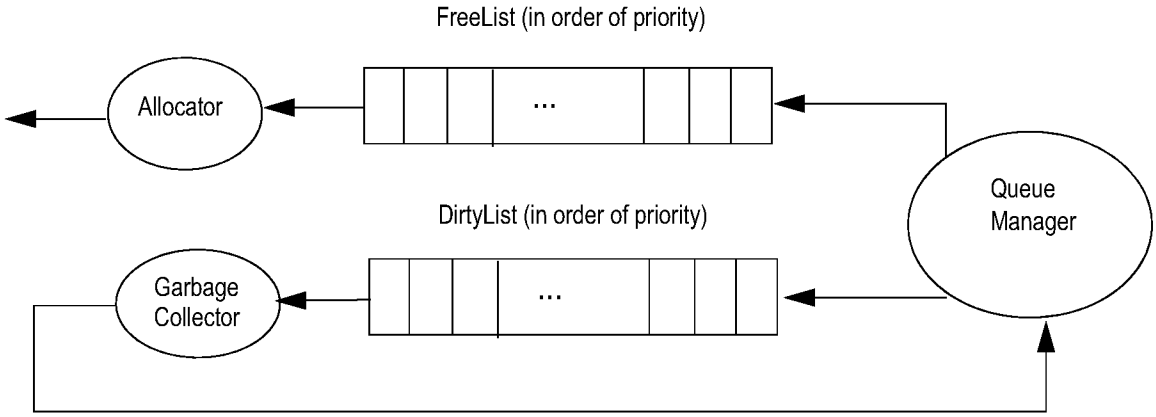


Figure 6.1 Wear Levelling Queue Structures

6.2.1 Priority based Algorithm

The entries in the DirtyList are in the order of priority. The lowest priority is at the head of the list and the highest priority is at the tail of the list. The priority is a function of the number of valid pages in a block and the number of erase count for the block. How the priority is calculated is shown in Figure 6.2. The block with the lowest number of erase count and the least number of valid pages is going to be acted upon by the garbage collector first. The block with the lowest number of erase count is picked first such that all blocks are erased evenly and reach their maximum number of erasures at about the same time. When there are less number of valid pages to copy, less flash bandwidth is used and hence the higher the system performance. The priority is designed in such a way to pick the block with the least number of pages to copy first. The erase count of a block is allowed to derivate from the average. When the erase count is falling behind from the average further and further, the priority becomes lower and eventually is at the head of the DirtyList even though all pages within the block may be valid.

Priority = # of Valid Pages + Erase Weigth

D = distance from average erase count

Erase Weight

= 0	for 0 <= D <= 15
= 1/8 D	for 16 <= D <= 31
= 1/4 D	for 32 <= D <= 47
= 3/8 D	for 48 <= D <= 63
= 1/2 D	for 64 <= D <= 79
= 5/8 D	for 80 <= D <= 95
= 3/4 D	for 96 <= D <= 111
= 7/8 D	for 112 <= D <= 127
= D	for 128 <= D

Figure 6.2 Setting the Priority

The algorithm supports static as well as dynamic wear leveling. Static data is never overwritten and the page that stores the static data is never obsoleted. All pages in a block that stores static(aka cold) data are validated. When the pages in this static block is reclaimed by the garbage collector, all pages have to be copied to a new ActiveBlock. Since the priority is based on both the number of valid pages and the erase weight, the static block reaches the head of the DirtyList eventually when other blocks are erased more often i.e. the static block's erase weight derivates from the average erase count more and more. For dynamic data, pages get overwritten all the time. A block that contains dynamic (aka hot) data have some or all it pages obsoleted. The block that contains the hottest data reaches the head of the DirtyList more often such that the least number pages have to be copied before the block is erased. When a hot block is erased more and more, its erase count is getting bigger and bigger than the average erase count.

The wear leveling is activated whenever there is a write request to a flash device. This is necessary as the number of valid pages within a block may change. The priority of a block has to be adjusted and the position of the block in the DirtyList must be also adjusted.

The garbage collector is not activated unless the number of entries in the FreeList falls below a low water mark threshold (LoTH). This is to avoid unnecessary data copy and delay the block erasure as late as possible. The garbage collector stops when the number of entries in the FreeList reaches a high water mark threshold (HiTH).

Only 80% of all the flash pages are advertised as external storage and 20% of the pages are reserved as spare. This allows a reasonable system performance when the flash devices are almost full. Without any spare pages, garbage collector has to copy 127 pages within a block to a local memory buffer, erases the block, and write back 128 pages (127 old pages and 1 new page) to the block new block for every single page write request.

Let $Q^{FreeList}$ $Q^{DirtyList}$ be the Queue than contains free flash block and dirty block respectively. A free flash block is a block that has been erased and be ready to be written into. A dirty block is a block that has all its pages written. A dirty block may contain anywhere between 0 and 128 valid pages of data. The entries in the DirtyList queues are sorted in the accenting order of priority. The priority is function of the erase count and the number of valid pages within a block. How the priority is determined is shown in Figure 6.2. Let function H(Q) return the first entry (head) of the queue, N(Q) returns the number of entries in the queue, AB be the ActiveBlock and function P(AB) is the next page of AB where next page is written into. AB is not in the any one of the $Q^{FreeList}$ $Q^{DirtyList}$.

The wear leveling algorithm:

- *Wear leveling is activated for every write request.*
- *Write data to P(AB)*
- *Adjust number of valid pages in ActiveBlock*
- *if (this is an update)*

  obsolete old page

  adjust number of valid page and priority

  adjust position of block in the DirtyList based on new priority
- *if (there is no more free page in the ActiveBlock)*

  Calculate priority of ActiveBlock

  Add AB to $Q^{DirtyList}$ - sorted by priority

  AB = $H(Q^{FreeList})$

  if ($N(Q^{FreeList}) < LoTH$)

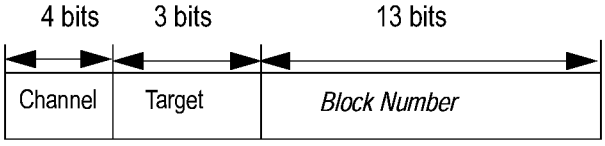
  Activate Garbage Collector

Garbage Collector:

- *Remove* $H(Q^{DirtyList})$
- *Copy data from all valid pages to local buffer*
- *Issue Write Request to save all valid pages*
- *Erase block*
- *Increment erase count and adjust priority*
- *Add block to* $Q^{FreeList}$
- *Repeats until* $(N(Q^{FreeList}) > HiTH)$

6.2.2 Block Information Management

A single data structure BLOCK_INFO is used to store the block information of each block in the flash device. The number of BLOCK_INFO entries matches the number of physical blocks in the system. The channel number, target number within a channel and the block number within a target make up the physical block number as shown below.



The BLOCK_INFO structure is shown below:

```
typedef struct {
      struct list_head    blockList;
                          // for queueing
      unsigned short      state               : 4,
                          // state of the current block
                          // i.e. erased, bad, active ...etc
                          reserved0           : 4,
                          pageNum             : 8;
                          // If this is an active block, pageNum
                          // is the page to write to on the next
                          // write request. After the page write,
                          // it should be incremented to to next page.
                          // When the last page has been written, this
                          // block should be then queued in the
                          // DirtyList. A new block should be allocated
                          // from the FreeList and becomes the
                          // new ActiveBlock
      unsigned short      validPageNum;
                          // How many valid Page in this block
      unsigned int        eraseCount;
                          // number of time this block has been erased
      unsigned int        validPageBit[4];
                          // bit is set to 1 when a page contains valid data
                          // There are 128 bits to support 128 pages
      unsigned int        numPagesMoved;
                          // Keep track of how many pages have to be moved
                          // before erasure in Garbage Collector. This is
                          // for managment purpose only

} BLOCK_INFO;

// Allocate one BLOCK_INFO per physical block
// phyical block is used as an index to locate the
// corresponding block data
// The phyical block number can also be calculated based
// on the pointer to a flash_block_data.

BLOCK_INFO    flash_block_data[FLASH_BLOCK_NUM];
```

An array of BLOCK_INFO flash_block_data is declared to allocate enough space to have one BLOCK_INFO per physical block in the system. During initial initialization, the BLOCK_INFO structures of all user data blocks are queued in the FreeList. There are one FreeList and one DirtyList per target. There are a total of 256 FreeList queues and 256 DirtyList queues. There are flash blocks reserved for other purpose such as storage for the firmware image and bad block information. Those special blocks are not managed by the wear leveling and hence not queued to the FreeList nor the DirtyList. All bad blocks are not used and not queued to the FreeList.

Since flash_block_data is organized as an array, the physical block number is used as an index to this array to locate the BLOCK_INFO for a particular physical block.

6.2.3 FreeList and DirtyList Management

There are only two operations that are performed on the FreeList; add or remove an entry. An entry is always added to the tail of the FreeList and is always removed from the head of the FreeList. In the steady state, the number of entries in the FreeList is small (HiTH is around 8).

The DirtyList contain flash_block_data in ascending order of the priority. In order to reduce the processing time to sort the entries, a priority based table lookup method is used. The actual implementation is not a single list. There are multiple lists; one per priority.

There are three operations that are performed on the DirtyList; add, remove and re-arrange an entry. The entry must be sorted in the order of priority. The priority of a block in the DirtyList changes when a page is obsoleted or when the average erase count has changed. The priority becomes negative when the erase count of the block is lower than the average erase count. In order to have a priority index starting 0, 128 is added to the priority before using it as an index to the priority table.

There are 512 entries in the priority lookup table and hence supporting a total of 512 different priorities. Simulation result indicates that the difference between the highest and the lowest erase count is around 200. The number of valid pages in a block is between 0 and 128. The range of the priority is less than 512. If the priority is less than -128, then 0 is used as the lookup index. If the priority is higher than 384, 511 is used as the lookup index. The entry in the priority lookup table at index N is a pointer to the head of the entries that has the priority N (actual priority is N - 128 as 128 is added to the priority to get the index). A NULL pointer is stored in the entry N if there is no entry that has the priority N.

An example of the priority lookup table and its relationship to the DirtyList is shown in Figure 6.3. In this example, there are two entries with the priority 1, three entries with the priority 3, one entry with priority 487 and the last there entries have the priorities of 488. The entries in sorted in assenting order of the priority. The priority table entry contains the head and tail pointers to the flash_block_data entries with the corresponding priority. There are 512 double linked list queues to make up the DirtyList. To reduce the searching time for the lowest priority in the DirtyList, the lowest Priority is stored and updated and the lowest priority changes.

```
// priority table is an array of 512 list_head entries
// pointing to flash_block_data
// We also remember the lowest priority in the DirtyList.

struct list_head    priorityLookupTable[512];
int                 lowestPriority = 0;

// To find out how many entries are there in the DirtyList
// We have to walk the priorityTable

int    numOfEntries = 0;
entry = priorityLookupTable[lowestPriority];
for (priority = lowestPriority; priority < 512; priority++) {
       entry = priorityLookupTable[priority];
       while ( entry is not the last entry) {
              numOfEntries++
              entry = entry->next;
       }
}
```

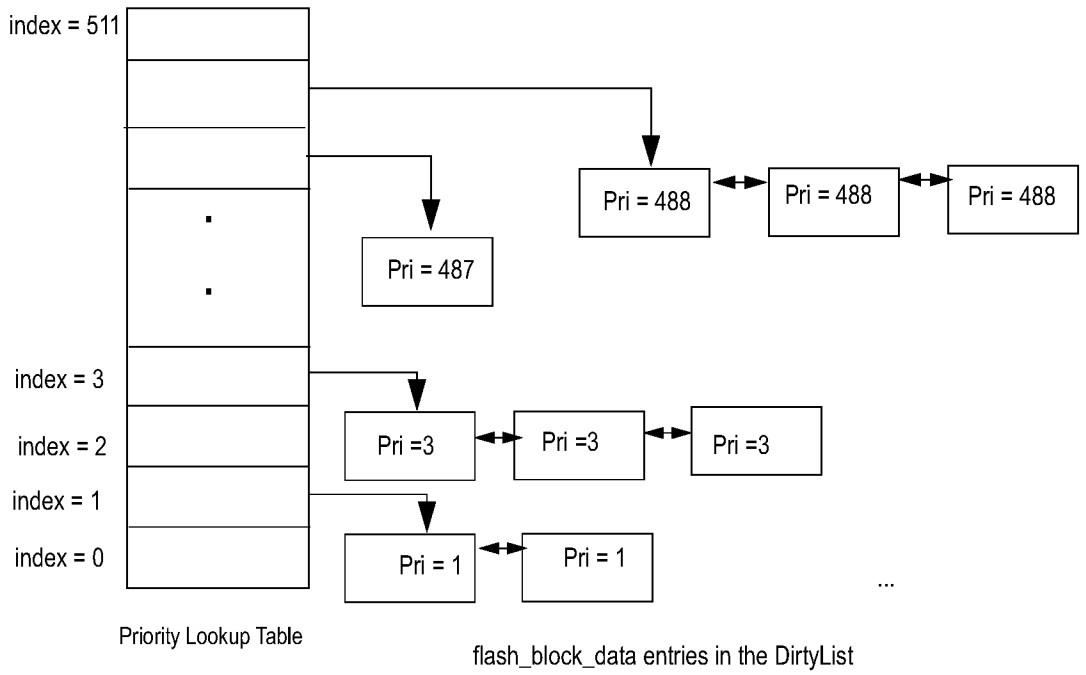


Figure 6.3 Priority Lookup Table's relationship to the DirtyList

The procedure to add an entry to the DirtyList:

```
AddEntry(BLOCK_INFO *new_block)
{
        index = GetIndex(new_block);
        current_block = priorityLookupTable[index];
        list_add_tail(new_block, current_block);
}
```

Figure 6.4 show an example before and after an entry is added to DirtyList. This example assumes that the priority for the new entry is 3 and there are already existing entries with priority 3 in the DirtyList. As shown in the example, it takes very little processor cycles to add an entry in the DirtyList. All it takes is to calculate the priority of the entry in order to get an index to the priority lookup table. The new entry is then added to the tail to the list with the corresponding priority.

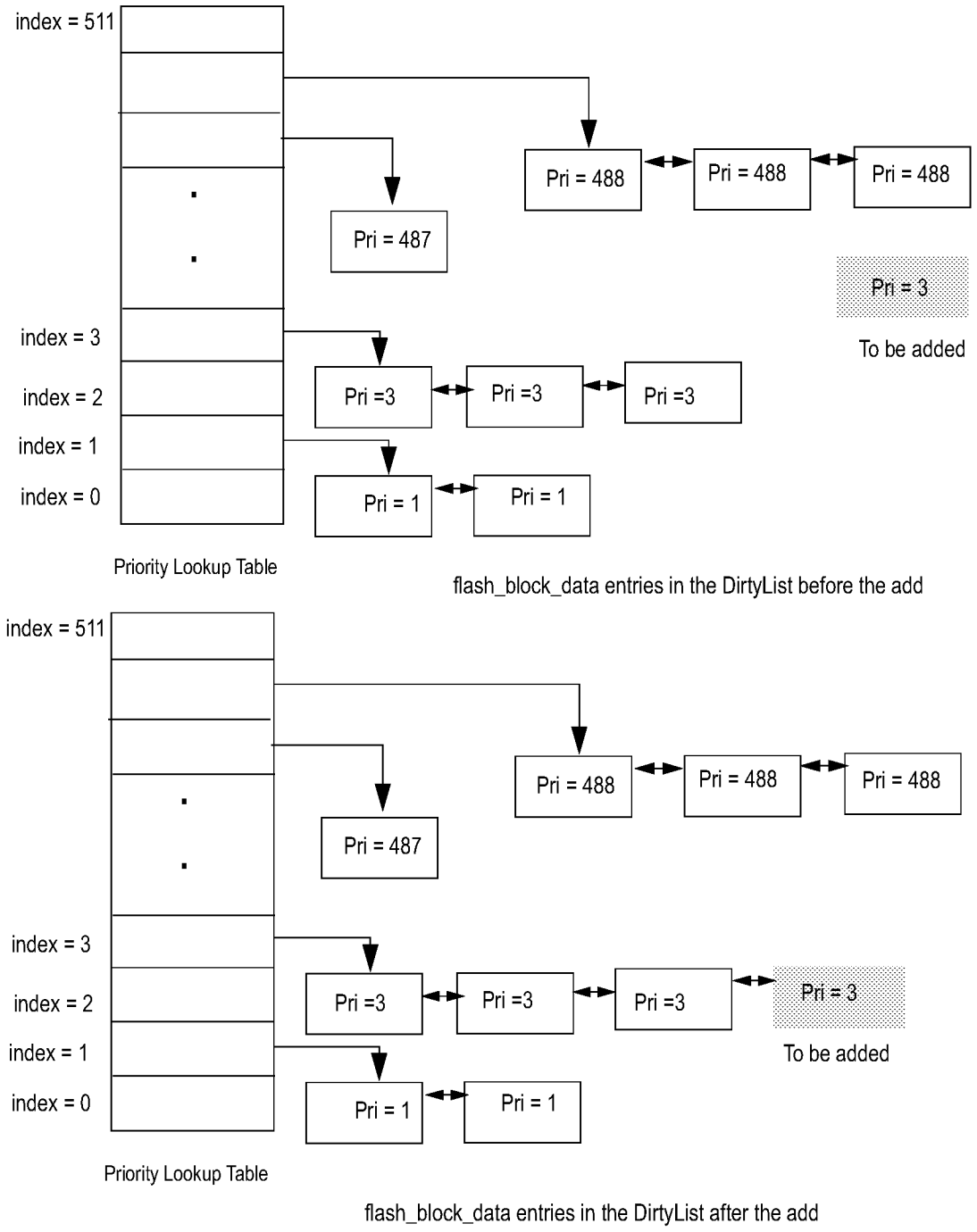


Figure 6.4 Adding a new entry

The procedure to remove an entry from the DirtyList:

```
RemoveEntry(BLOCK_INFO *new_block)
```

```
{
        //  All we need to is to remove the entry from the
        //  DirtyList.
        list_delete(new_block);
}
```

Removing an entry from the DirtyList is very simple and takes very few processor cycles.

When the priority changes in a block, the entry has to be re-arranged in the DirtyList. There are two cases where the priority changes when an entry is in the DirtyList; when a page becomes obsoleted or when the average erase count has changed. When the average erase count goes up by 1, the priority of a block may remain the same, goes up or down by 1, 2, 4, 8 or higher depending upon its own erase count distance from the average erase count. The priority for all blocks with the DirtyList have to be recalculated and hence re-sorted. This takes up too much CPU cycles. In general, all the entries are more or less in the right order after the average erase count has increased by 1. There is not much to lose for not resorting the entries in the DirtyList. No action is required when the average erase count changes. When a page of a block is obsoleted some time later, the entry is then sorted and be inserted into the new priority list.

Note: What happen to a block that has all static data ?

The other event that triggers the change of priority is when a page is obsoleted. This happens when a page is updated. A single page is updated and hence the priority is reduced by 1. The affected block in the DirtlyList has to be moved up the list.

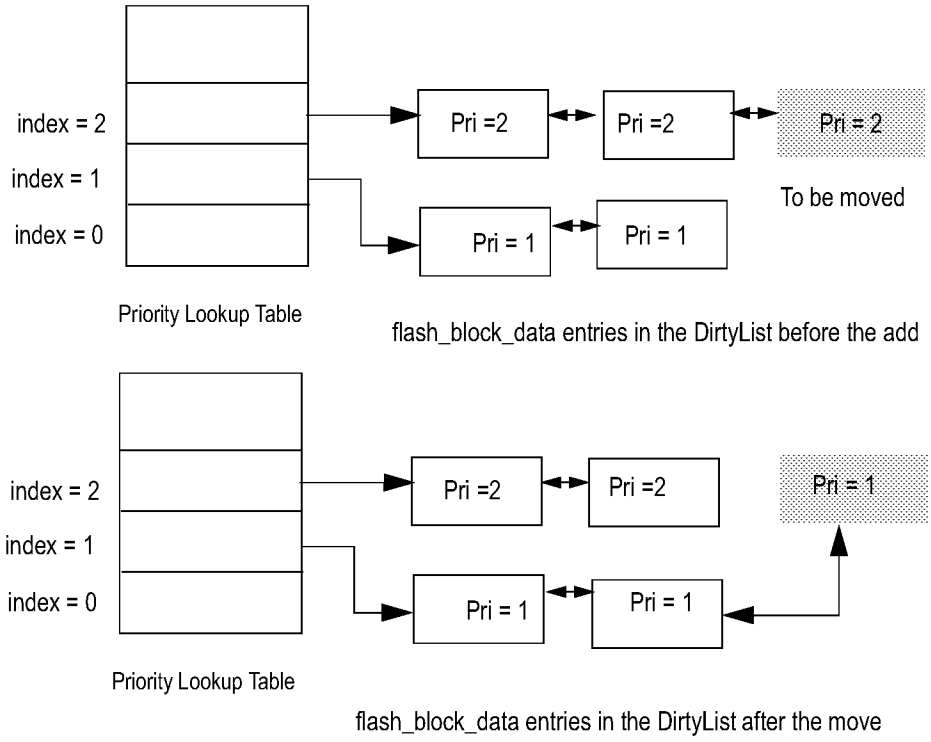


Figure 6.5 Moving an entry in the DirtyList

An example on how an entry is moved is shown in Figure 6.5. In the example, a page has been obsoleted and the priority is going to be changed from 2 to 1. There are three entries in the DirtyList with the same priority 2. There are two entries in the priority 1. The procedure to move the entry in this example

- *Remove the entry from the DirtyList*
- *Locate the head of the priority list which has a priority of 1 using the priority lookup table*
- *Add the entry to the tail*

```
MoveEntry(BLOCK_INFO *new_block)
{
        //  remove the entry from the
        //  DirtyList.
        list_delete(new_block);
        index = GetIndex(new_block);
        current_block = priorityLookupTable[index];
        list_add_tail(new_block, current_block);
}
```

6.2.4 State Diagrams

There are three separate state machines running concurrently. The Block state machine handles the block open/close request fromthe Write Manager. The Obsolete state machine handles the Obsolete page request from the Write Manager. The Recliam state machine handles the garbage collection request to the Garbage Collection Manager.

6.2.5 Block State Machine

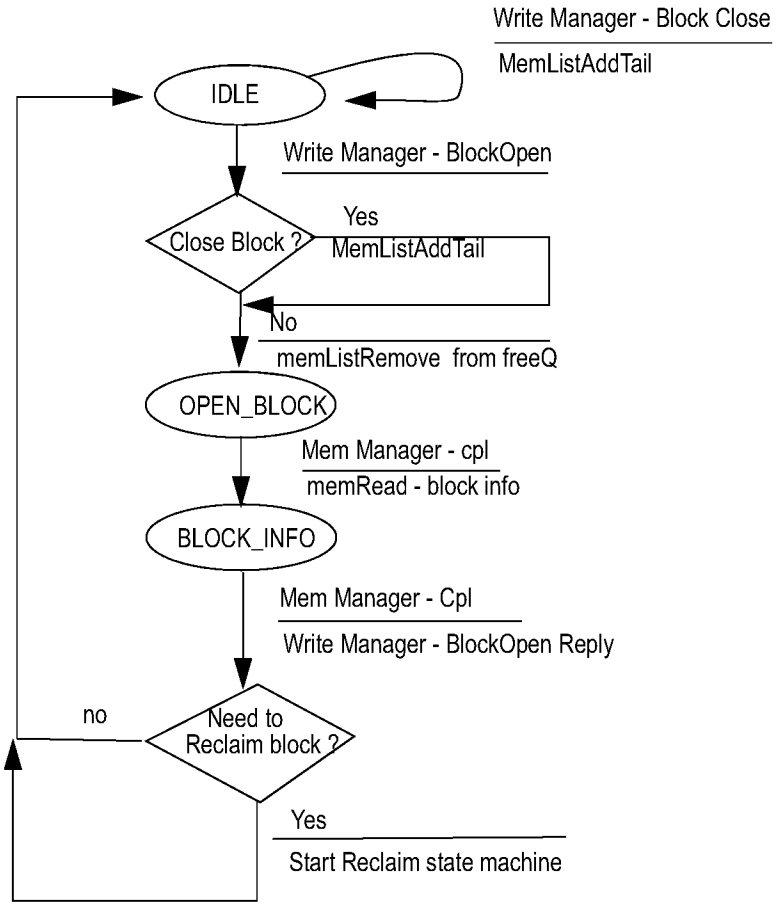


Figure 6.1 Block Open/Close State MachineDiagram

6.2.6 Obsolete State Machine

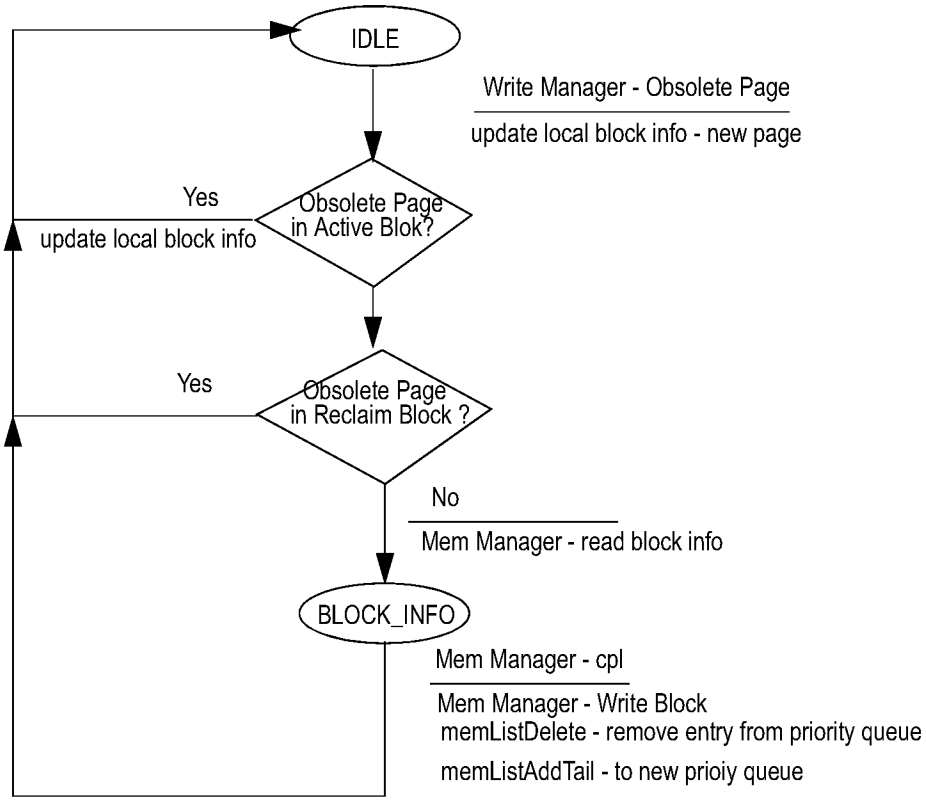


Figure 6.6 Obsolete Page State Machine Disgram

6.2.7 Reclaim State Machine

Where more blocks need to be reclaimed at the end, we don't start another reclaim immediately. A request is made to start another reclaim cycle instead. This allows other target to have a chance to start the reclaim process.

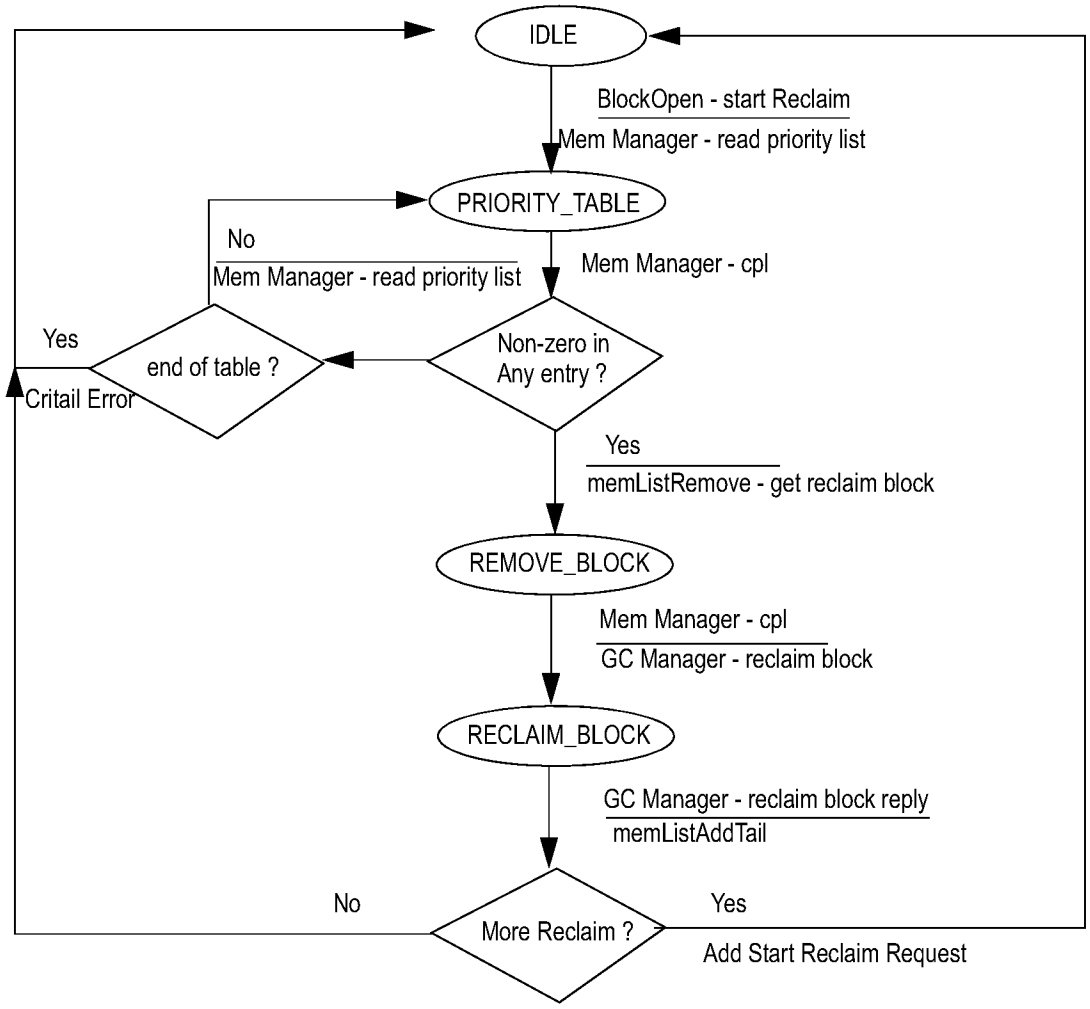


Figure 6.7 Reclaim Block State Machine Diagram

6.2.8 Allocation of Free Flash Block for Writing

There is a FreeList and a DirtyList per target. In a system with 16 channels and 8 targets per channel, there are a total of 128 FreeLists and 128 DirtyLists. In order to get the best write performance, write request is distributed evenly to all targets. The write manage keeps an array of "open" block where data are written to. A single free flash block is allocated from the Wearleveling manager for each one of the index in the array. The size of the array is 128. The Write Manager uses a round robin algorithm to pick the next page to write to such that all targets are written to in parallel. Write manager starts writing to the "open" block at index 0. The index is incremented by 1 on subsequent write request. The index wraps around to 0 when it reaches a value of 127.

```
//
//     16 channel and 8 targets per channel
//
```

```
//     index to the array = channel * 8 + target
//     i.e. for a channel of 2 and target of 3, the
//     index the freeBlock = 19.
BLOCK_INFO    freeBlock[16 * 8];


AllocateBlock(int index)
{
       BLOCK_INFO    block;

       return list_remove(&freeBlock[index]);
}
```

The same algorithm is also shown in Figure 6.8. The first block to be used for the first page write request is channel 0 target 0. The next block to be used for the next write request is channel 0 target 1 and the next one is channel 0 target 2 and so on. When the last block channel 15 target 7 is used, it repeats itself and starts from channel 0 target 0 again.

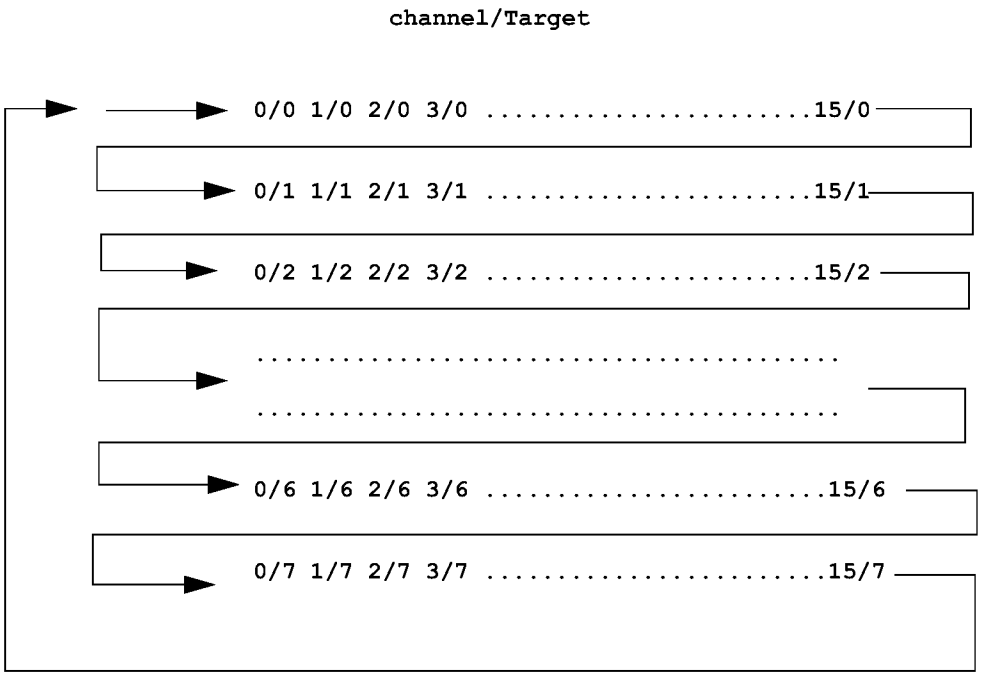


Figure 6.8 Allocation of next block to write across channels and targets

6.2.9 Wear leveling across channel/target

Since the write request is distributed evenly across all channels and targets, data should be written evenly to all target flash devices. Depending upon the traffic pattern, block cross different targets may not be erased evenly. The priority based wear leveling works within a target only. In order to perform wear leveling across channel/target, the

average erase count of all the targets is maintained. When the average erase count of a target is higher than a high water market threshold, the target is skipped every other write request. when the average erase count falls below a low water threshold, the target can be written on every write request.

6.3 Garbage Collecting

Garbage collector is activated when the number of entries in the FreeList drops below a low water mark threshold (LoTH). Garbage collector removes flash_block_data from the head of DirtyList to reclaim obsolete pages. The garbage collector stops reclaiming blocks when the number of entries in the FreeList reaches a high water mark threshold (HiTH). Entries are removed from the Dirtylist using the following algorithm:

```
GarbageCollector()
{
        //
        //      Locate the head for the DirtyList
        //      It is the first entry in the lowest priority list
        //

        list = priorityLookupTable[lowestPriority];
        while (QueueSize(FreeList) < HiTH){
                if (!QueueEmpty(list)) {
                        entry = list_head(list);
                        list_delete(entry);
                        ReclaimBlock(entry);
                }
                else {
                        //
                        //      no more entry in this priority. Search for
                        //      the next higher priority
                        for (index = lowestPriority; index < 512; index++) {
                                list = priorityLookupTable[index];
                                if (!QueueEmpty(list)) {
                                        lowestPriority = index;
                                        break;
                                }
                        }
                        //
                        // no more entry in the DirtyList
                        // we are done. Note: This should not happen unless
                        // the high water mark threshold is not set correctly.
                        if (index == 512) {
                                return
                        }
                }
        }
}


ReclaimBlock(entry)
{
        unsigned        logicalPageNum;
        unsigned        phyPageNum;
        char            *bufferRam;
```

```
                //
                //      Need to copy all valid pages within
                //      this block to new pages
                //
                while (entry->validPageNum--) {
                        //
                        //      Get the current page number to be
                        //      copied out.
                        //      Allocate a buffer from buffer ram such
                        //      that we can read the page data and metadata
                        //      to this buffer.
                        //      Once we have the page data in the buffer ram,
                        //      We can read the logical page number from the
                        //      metadata area.
                        page = NextPage(entry);
                        bufferRam = BufferRamAlloc();
                        ReadPageRequest(page, bufferRam);
                        logicalPageNum = GetLogicalPageNum(bufferRam);
                        //
                        //      We have to lock the logical to physical page
                        //      entry such that no more change can be made
                        //      to this page until the page has been written
                        //      to the new page.
                        LockEntry(logicalPageNum);
                        phyPageNum = GetPhyPageNum(logicalPageNum);
                        if (PhyPageNum == page) {
                                //
                                //      The phyical page number is still the same
                                //      as before. The page has not been accessed
                                //      since our last read request. We ask the
                                //      write manager to save this page to a
                                //      new page. The write manager will unlock
                                //      the entry after the write is completed.
                                WritePageRequest(logicalPageNum, bufferRam);
                        }
                        else {
                                //
                                //      If the phyical page number is different,
                                //      the page has been updated with new data.
                                //      This page is obsoleted and there is no
                                //      need to save this page. We can just
                                //      unlock the entry and move on.
                                UnlockEntry(logicalPageNum);
                        }
                EraseBlock(entry);
        }
```

6.3.1 State Diagram

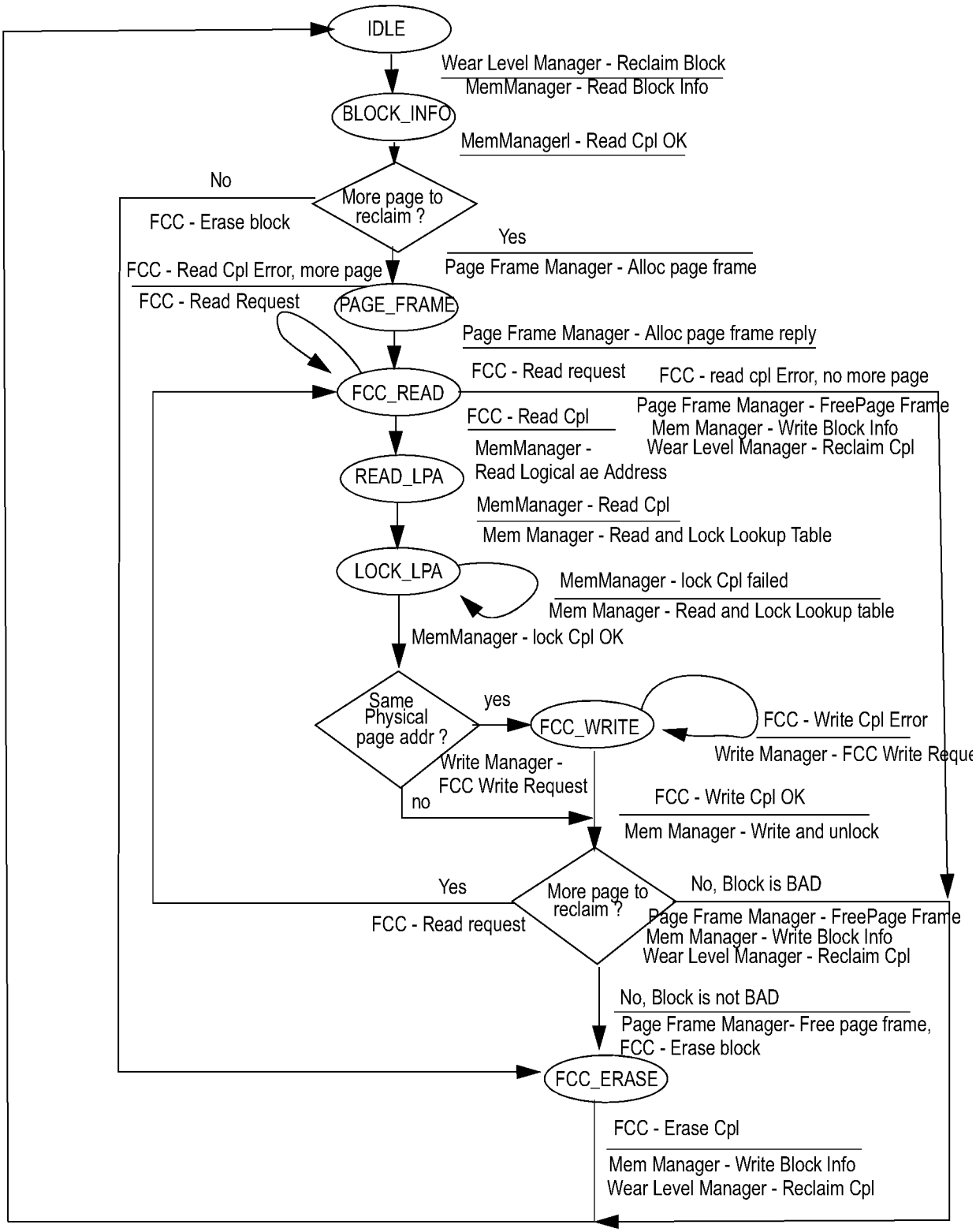


Figure 6.9 Garbage Collecting State Diagram

6.4 Bad Block Management

NAND flash devices are shipped from the factory erased. The factory identifies invalid blocks before shipping by programming data other than FFh into a well known location. The Micron MT29F128G08 Flash device has the first spare byte in the first page of the each block programmed to be 00h to indicate that the block is bad. The factory marked bad blocks are discovered during initial system initialization and the blocks are marked as BAD. The bad blocks must not be used to store any data to avoid any data corruption. The bad blocks are not added to the freeList. Only the user partitions are scanned for bad block during the first time initialization.

A bad block is developed when there is a programmed error or erase error during normal usage. When there is an error during erasure, this block is marked as BAD and not added back to the freeList. This block is never be reused again. When there is a program error during a page write request, the block is marked as BAD. The block is not retired immediately. The next page is used for the next write request. When all the pages are written, this block is subjected to the normal wear leveling management. When garbage collection kicks in to clean up this block, all valid pages are copied from this block to a new block. Since this bad is marked as BAD, this block is not going to be erased. This block is never to be reused again.

6.5 Inbound Message Queues

The Wearleveling manager requires inbound message queues to receive messages from the following managers and functional blocks.

Table 6.1 Write Manager

- *Flash Controller*
- *Page Frame Manager*
- *Memory Network*
- *Control Manager*

6.6 Inbound Outbound Messages

The Wearleveling Manager sends request and response messages to the Write Manager, Page Frame Manager and the Flash Controller. A list of the messages that the Wearleveling Manager issues to and receives from other managers/functional unit are shown in Table 6.2.

Issued Messages \ Received Messages	Control Manager	Write Manager	Wearleveling Manager	GarbageCollection Manager	Page Frame Manager	Flash Controller
Control Manager	N/A	N/A	TBD	TBD	N/A	N/A
Write Manager	N/A	N/A	Requests BlockOpenRequest BlockClose ObsoletePage	N/A	N/A	N/A
Wearleveling Manager	Requests TBD Completions TBD	Completions BlockOpenReply	N/A	Requests ReclaimBlockRequest	N/A	N/A
GarbageCllection Manager		Requests PageWrite	Completions ReclaimBlockReply	N/A	Requests PageFrameAllocRequest PageFrameFree	Requests FlashRead FlashErase
Page Frame Manager	N/A	N/A	N/A	Completions PageFrameAllocReply	N/A	N/A
Flash Controller	N/A	N/A	N/A	Completions FlashOpCpl	N/A	N/A

Table 6.2 Wearleveling Manager Issued and Received Message

Page Frame Manager

7.1 Overview

A page frame is a memory buffer where the flash controller writes data to in the case of a flash read operation and reads data from in the case of a flash write. A page frame consists of two buffers, one for the flash page data and one for the flash page meta-data. Page frames may exist in buffer RAM or DDR memory. All managers that issue flash read or write operations must first allocate a page frame. The address of the page frame is then passed to the flash controller in a FlashRead or FlashWrite message. The Page Frame Manager is responsible for initializing, allocating and freeing page frames in response to messages received from other managers.

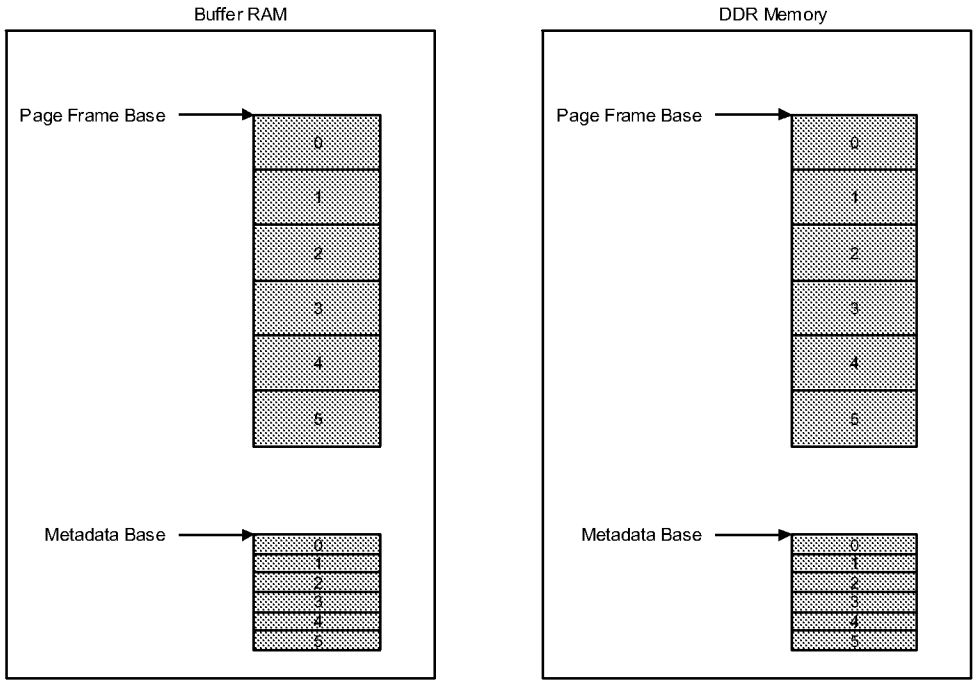


Figure 7.1 Page Frame Buffer Base Addresses in DDR and Buffer RAM

7.2 Architecture

The Page Frame Manager defines two static arrays of Page Alloc Data Structures, one for maintaining page frame and meta-data buffers located in buffer RAM, and one for maintaining page frame and meta-data buffers located in DDR memory.

```
typedef struct {
      LIST_HEAD          list;
      u32                pageFrameAddr;
      u32                metadataAddr;
} PAGE_ALLOC;
```

Figure 7.2 Page Alloc Data Structure

The *pageFrameAddr* and *metadataAddr* fields in the Page Alloc Data Structure hold bits 36:5 of a 37 bit address used to access the page frame and meta-data buffers. All page frame and meta-data buffers should be 64B aligned.

The Page Frame Manager maintains two lists:

- *BRAMPages*
- *DDRPages*

The *BRAMPages* list consists of free page frames that reside in buffer RAM and the *DDRPages* list consists of free page frames that reside in DDR memory.

7.2.1 Initialization

- How do we know when the super block has been loaded so that a manager can read it from DDR??? Discussed using SYNC message with Control Manager (Jack)

Until a Page Frame Init message is received, the Page Frame Manager will respond to all Page Frame Alloc Request messages with a Page Frame Alloc Reply message that contains 0 in the *COUNT* field and all Page Frame Free messages will be ignored. A Page Frame Init message consists of the following information:

- *Base address of page frame memory in buffer RAM*
- *Size of the buffer RAM memory allocated for page frames*
- *Base address of page frame memory in DDR*
- *Size of DDR memory allocated for page frames*

With this information the Page Frame Manager first programs the flash controllers base address pointers and then initializes the buffer RAM and DDR **PAGE_ALLOC** arrays. As each array element is initialized it is added to the appropriate free list. The size of the page frame and meta-data buffers will be obtained from the Super Block, or hard coded in #defines.

7.2.2 Page Frame Allocation

One-to-n page frames are allocated when the Page Frame Manager receives a Page Frame Alloc Request from another manager. The *COUNT* field in the received messages specifies how many page frames are being requested and the *M* field specifies from which memory pool the page frames should be allocated, buffer RAM, DDR or "don't care". In the case where the *M* field is set to "don't care" the Page Frame Manager will always attempt to allocate page frames from the buffer RAM before DDR. For the number of page frames requested, the Page Frame Manager removes a **PAGE_ALLOC** structure from the appropriate list, and generates the Page Frame Alloc Reply message. If the number of page frames requested exceeds the number of free page frames the Page Frame Manager will return the actual number of page frames that were allocated in the *COUNT* field of the reply message, this value will be zero if no page frames are available.

7.2.3 Page Frame Freeing

When a page frame is no longer needed a Page Frame Free message should be sent to the Page Frame Manager in order to release the buffer. The *COUNT* field in the Page Frame Free message specifies how may buffers are being freed by this request. For each page frame being released the Page Frame Manager calculates the index into the appropriate **PAGE_ALLOC** array and adds the indexed array element to the appropriate free list. The topmost bit in the page frame address is used to determine if the page frame resides in buffer RAM or DDR memory. The **PAGE_ALLOC** array index is calculated with the following algorithm:

```
index = (page_frame_address - page_frame_base) >> page_frame_shift
```

This assumes that page frames are always $2^{page\_frame\_shift}$ bytes in size.

7.2.4 Statistics

With the algorithms described above it will be possible to maintain page frame statistics, the number of times a page frame was allocated etc.

7.3 Performance Considerations

Under heavy loads the Page Frame Manager may become a system bottleneck, care must be taken to optimize the Page Frame Manager code for maximum performance.

7.4 Inbound Message Queues

The Page Frame Manager requires inbound message queues to receive messages from the following managers and functional blocks.

- *Wearleveling Manager*
- *Data Managers*
- *Control Manager*

7.4.1 Inbound Messages

The following messages are received from other managers and functional blocks by the Page Frame Manager. The exact format of these message is described in the IPC Messages section of this document.

- *Page Frame Init - Message sent to the Page Frame Manager containing the buffer RAM and DDR page frame configuration information.*
- *Page Frame Alloc Request - Message sent to the Page Frame Manager to request one or more page frame buffers.*
- *Page Frame Free - Message sent to the Page Frame Manager to free one or more page frames that are no longer needed.*

7.5 Outbound Message Queues

The Page Frame Manager will be required to send messages to the following managers and functional blocks.

- *Wearleveling Manager*
- *Data Managers*
- *Control Manager*

7.5.1 Outbound Messages

The following messages are sent to other managers and functional blocks by the Page Frame Manager. The exact format of these message is described in the IPC Messages section of this document.

- *Page Frame Alloc Reply - Message sent from the Page Frame Manager in response to a Page Frame*

Alloc Request message.

Inter-processor Communications

8.1 Overview

Each processor is able to send and receive messages from all of the other processors and hardware blocks in the system by means of the Inter-processor Communications (IPC) network. Each IPC is a unidirectional pipe that consists of an outbound free queue, an inbound full queue, and message data memory. During the IPC initialization process, the message data memory is divided into equal sized message buffers. The address of each of these buffers is then written to the IPC's outbound free queue, see #1 in the figure below.

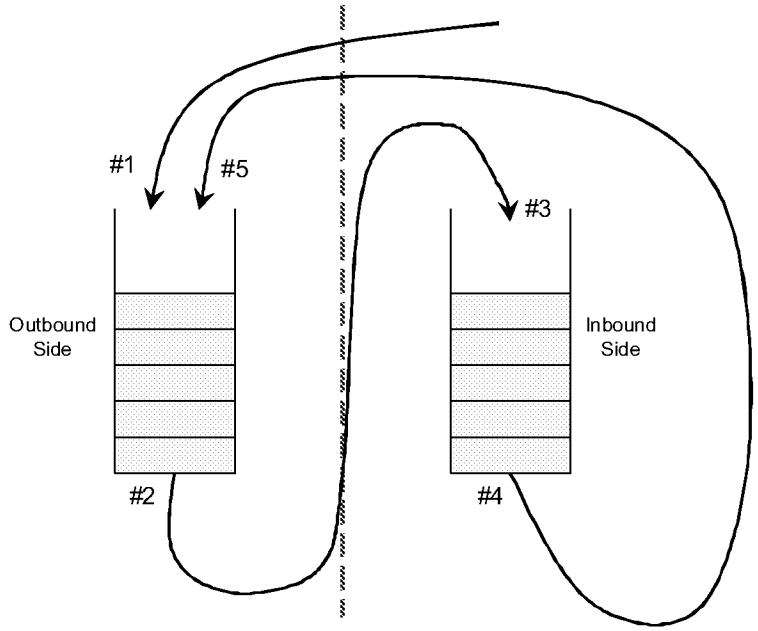


Figure 8.1 IPC Operation

When a message is to be transmitted, the address of a free message buffer is first obtained by reading the outbound free queue, see #2 in the figure above. The message buffer is then filled with the message data before the message is sent by writing it to the outbound side post queue (#3). The inbound side receives a message by reading a message buffer address from the inbound full queue (#4). The inbound message is then processed and freed back to the outbound side of the IPC by writing the address of the message buffer to the inbound free queue (#5).

8.2 Firmware

The IPC firmware will be built as a static library that can be linked with any application that requires such functionality. The following sections describe the functionality of the IPC library.

8.2.1 Initialization

Before an IPC can be used the outbound side of the IPC must be linked to the inbound side of an IPC on a different node. In order to simplify this process, the IPC_CONFIG structure, shown below, is used to describe the relationship between all of the inbound and outbound IPC's.

```
typedef struct {
        unsigned int isHardware          : 2;
        unsigned int depth               : 8;
        unsigned int outboundNodeID      : 8;
        unsigned int outboundIPCIndex    : 4;
        unsigned int inboundNodeID       : 8;
        unsigned int inboundIPCIndex     : 4;
        unsigned int messageAddr;
        unsigned int messageSize;
} IPC_CONFIG;
```

Figure 8.2 IPC Config Structure

The *depth* field is used to specify how many message pointers the IPC may hold. The *outboundNodeID* field specifies the node ID where the outbound side of the IPC physically resides and the *outboundIPCIndex* field specifies the index of the outbound side of the IPC on that node. Likewise, the *inboundNodeID* and *inboundIPCIndex* fields are used to specify the physical location of the inbound side of the IPC. The *messageAddr* field specifies the start address of the message buffer memory associated with this IPC and *messageSize* specifies the size, in bytes, of each message. The *isHardware* field is used to specify that the destination node is a hardware block which causes slightly different handling during initialization. A typical IPC configuration map is show below.

```
static const IPC_CONFIG    map[] = {
        { 0, 4, 0x00, 0, 0x01, 0, (unsigned int) messageBuffer, IPC_BUFF_SIZE
},
        { 0, 4, 0x01, 0, 0x02, 0, (unsigned int) 0, 0 },
        { 0, 4, 0x02, 0, 0x03, 0, (unsigned int) 0, 0 },
        { 0, 4, 0x03, 0, 0x10, 0, (unsigned int) 0, 0 },
        { 0, 4, 0x10, 0, 0x11, 0, (unsigned int) 0, 0 },
        { 0, 4, 0x11, 0, 0x12, 0, (unsigned int) 0, 0 },
        { 0, 4, 0x12, 0, 0x13, 0, (unsigned int) 0, 0 },
        { 0, 4, 0x13, 0, 0x20, 0, (unsigned int) 0, 0 },
        { 0, 4, 0x20, 0, 0x21, 0, (unsigned int) 0, 0 },
        { 0, 4, 0x21, 0, 0x22, 0, (unsigned int) 0, 0 },
        { 0, 4, 0x22, 0, 0x23, 0, (unsigned int) 0, 0 },
        { 0, 4, 0x23, 0, 0x00, 0, (unsigned int) 0, 0 },
        // End of the IPC map
        { 0, 0x00, 0x00, 0, 0, 0x00000000, 0 },
};
```

Figure 8.3 IPC Configuration Map

Initialization of the IPC network is performed by each CPU calling void IPCInit(const IPC_CONFIG *map) and passing a pointer to the IPC configuration map.

Due to the complexity of the IPC Configuration Map, a native linux application (IPCMapGenerator) has been developed to generate an IPC Configuration Map from simple text files. This application also generates the #defines that should be used by cores that require access to the IPC network. The generated IPC Configuration Map is located in *lib/ipcmap.c* and generated *#defines* reside in *include/ipcmap.h* within the Delaware firmware.

8.2.2 IPC Map Generator

The IPC Map Generator is used to auto generate the IPC Configuration Map and #defines that should be utilized by cores that require access to the IPC network. The IPC Map Generator utilizes two text files to generate the this information. The first text file (managers.txt) is used to give simple names to each manager in the system,

to specify if the manager is a hardware functional unit or a CPU core, and to specify if a destination managers outbound queue index should match with the source managers queue index. An example managers.txt file is shown in Figure 8.4.

```
# Manager Name0/1 = CPU/HW Match outbound
IDMA          1
PFMGR         0             1
core02
core03
PCIE
DMGR
WMGR
FCC           1
EDMA          1
core21
core22
core23
```

Figure 8.4 IPC Manager Config File

The first entry in the file is used to specify the manager name. This text must match one of the definitions in *include/nodeid.h* within the Delaware firmware. The second entry is used to specify if the manager is a hardware functional unit or a CPU core. A value of 1 equals hardware functional unit, a value of 0 or no value equals a CPU core.

The second file is used to configure the IPC Configuration Map. An example of an IPC configuration text file is shown in Figure 8.5.

```
#<source node><destination node><queue depth><message size><message mem start>
DMGR            IDMA              16            24            0x3ffe0000
DMGR            EDMA              16            24            0x3ffe0000
DMGR            FCC               16            24            0x3ffe0000
WMGR            FCC               16            24
FCC             WMGR              16            12            &msgBuffers
FCC             DMGR              16            12            &msgBuffers
IDMA            DMGR              16            12
EDMA            DMGR              16            12
PCIE            DMGR              16            12
```

Figure 8.5 IPC Configuration Map Text File

The first field is the name of the source node, the one that will be sending messages. The second field is the name of the destination node, the one that will be receiving messages. Both of these names must match names configures in the IPC Manager Config File. The next value specifies the depth of the queue in entries, which is followed by the size, in bytes, of each message in the queue. The final value is either the name of a variable that will be used for the message buffers, or a physical memory address in the case of hardware functional blocks. If the last entry is blank a previously assigned variable or memory address is used, adjusted by the size of the previously declared queues.

8.2.3 Interrupts

If enabled, the outbound free queue and the inbound full queue have the capability of generating an interrupt to the processor when they contain valid message pointers. Interrupts for each inbound/outbound message queue may be masked/unmasked by calling int IPCInboundIntMask(unsigned int mask) or int IPCOutboundInt-Mask(unsigned int mask), respectivly.

8.2.4 Sending Messages

In order to send a message, a message buffer pointer must first be obtained by calling unsigned int IPCOutboundGet(int qIndex). Assuming a message buffer pointer was available, the message buffer will be populated and then sent to the inbound side of the IPC by calling void IPCOutboundPost(int qIndex, unsigned int handle).

If the software managed queue associated with this IPC currently contains queued message pointers or if the outbound IPC free queue is empty, unsigned int IPCOutboundGet(int qIndex) will return a software allocated message buffer pointer (if a software message buffer could not be allocated, *IPC_EMPTY* will be returned). This message buffer will be populated in the usual way before being sent to the inbound side of the IPC by calling void IPCOutboundPost(int qIndex, unsigned int handle).

When void IPCOutboundPost(int qIndex, unsigned int handle) is called, it checks to see if the *buffer* parameter points to a regular buffer or a software allocated buffer. If *buffer* points to a software allocated buffer the buffer pointer is queued and the outbound IPC message queue interrupt is enabled. If *buffer* points to a regular buffer, the buffer pointer is posted immediately to the inbound side of the IPC message queue.

```
int sendMsg(int ipc_ob)
{
       unsigned int              buffer;

       /* Get a message buffer pointer */
       if((buffer = IPCOutboundGet(ipc_ob)) != IPC_EMPTY)
       {
              unsigned int temp = buffer;
              temp = IPCWriteData(ipc_ob, temp, 0x55);
              temp = IPCWriteData(ipc_ob, temp, 0xaa);
              IPCOutboundPost(ipc_ob, buffer);
       }
       else
              printf("Error getting buffer!\n");

       return(0);
}
```

Figure 8.6 Outbound IPC Usage Example

8.2.5 Receiving Messages

A message is received by calling void *IPCInboundGet(int qIndex, unsigned int *handle). If a message is available, a pointer to the message buffer will be returned. If no messages are available a value of *IPC_EMPTY* will be returned. Once a received message has been processed the message buffer pointer should be freed by calling void IPCInboundFree(unsigned int handle).

```
int recvMsg(unsigned int ipc_ib)
{
       unsigned int              *buffer;
       unsigned int              handle;

       /* Get a message buffer pointer */
       if((buffer = IPCInboundGet(ipc_ib, &handle)) != (void  *) IPC_EMPTY)
       {
              /*
               * Process message here
               */
              IPCInboundFree(handle);
       }
       else
              printf("No messages available!\n");

       return(0);
}
```

Figure 8.7 Inbound IPC Usage Example

8.2.6 IPC Message Queue Library API

The following sections document the IPC message queue library API. For the most up to date documentation please refer to the IPC message queue source code located in the Delaware CVS repository under *delaware/firmware/EFC/lib*.

8.2.6.1 Library Files

The following libraries should be linked to any application that utilizes IPC message queues.

- *libDelaware.a*

8.2.6.2 Include Files

The following include files should be included in order to use the IPC message queue API.

- *ipc.h*
- *ipcmap.h*

8.2.6.3 void IPCInit(const IPC_CONFIG *map)

Initialize the inter-processor communication network based on the configuration map passed in via the map parameter.

Parameters:

map Pointer to the IPC configuration map.

Returns:

0 Initialization successful.

-1 The IPC configuration map contains duplicate entries.

8.2.6.4 unsigned int IPCOutboundGet(int qIndex)

Get the next available message buffer pointer from the outbound IPC free queue.

Parameters:

qIndex	Index of the outbound free queue.

Returns:

Handle to a message buffer on success, else IPC_EMPTY.

8.2.6.5 void IPCOutboundPost(int qIndex, unsigned int handle)

Send a message buffer pointer to the inbound side of the IPC message queue.

Parameters:

qIndex	Index of the outbound message queue.
buffer	Handle to the message buffer to be posted.

Returns:

N/A

8.2.6.6 void *IPCInboundGet(int qIndex, unsigned int *handle)

Receive the next message buffer pointer queued at the inbound side of the IPC message queue.

Parameters:

qIndex	Index of the inbound message queue.
handle	Pointer to a handle variable.

Returns:

Pointer to a message buffer on success, else IPC_EMPTY.

8.2.6.7 void IPCInboundFree(unsigned int handle)

Free a message buffer pointer that was previously received by calling void *IPCInboundGet(int qIndex, unsigned int *handle).

Parameters:

buffer	Handle to the message buffer being freed.

Returns:

N/A

8.2.6.8 int IPCOutboundIntMask(unsigned int mask)

Mask or unmask outbound IPC interrupts. Each bit in mask corresponds to an outbound free queue.

Parameters:

mask	Bit mask, bits set to 1 mask interrupts, 0 unmask interrupts.

Returns:

Previous mask value.

8.2.6.9 int IPCInboundIntMask(unsigned int mask)

Mask or unmask inbound IPC interrupts. Each bit in mask corresponds to an inbound queue.

Parameters:

mask	Bit mask, bits set to 1 mask interrupts, 0 unmask interrupts.

Returns:

Previous mask value.

8.2.6.10 void IPCWriteData(int qIndex, unsigned int handle, unsigned int data)

Write 4 bytes of data to the address and outbound queue specified.

Parameters:

qIndex	Index of the outbound queue from where *addr* was obtained.
addr	Handle to the destination node address where *data* should be written.
data	Data to be written.

Returns:

Handle to the next 4 bytes to be written on the destination node.

Figure 8.8 Message Dispatching Example

8.2.6.11 void *IPCInboundGetAny(unsigned int mask, unsigned int *handle, int *qIndex)

Receive the next message buffer pointer from any one of the unmasked inbound message queues.

Parameters:

mask	Bitmask of inbound queues to include in the read.
handle	Pointer to a handle variable.
qIndex	Index of the inbound message queue.

Returns:

Pointer to a message buffer on success, else IPC_EMPTY.

Memory Network

9.1 Overview

The processors within the processor complex do not have direct access to the Buffer RAM or the DDR memory. Instead, variable sized blocks of memory may be copied between Buffer RAM or DDR memory and a processors local data RAM via the Memory Network. Each processor core includes custom instructions to allow posted (blocking) and non-posted (non-blocking) requests to be issued to the Memory Network. Instructions that issue posted requests to the Memory Network always return 0xff, while instructions that issue non-posted requests return a tag value in the range of 1-32. The completion of a non-posted request is signaled via inbound IPC message queue 0. This completion contains the same tag (in bits 7:0) that was returned when the non-posted request was issued. Firmware will use this tag to match Memory Network requests with their corresponding completions. The exact format of this completion is described in the Processor Complex section of the Delaware FSD.

Firmware wishing to utilize the Memory Network API should include include/memNetwork.h.

9.2 Memory Network API

The following sections describe the API that should be used to issue requests to the Memory Network.

9.2.1 void memInit(void)

This function should be called before calling any other Memory Network functions to initialize the Memory Network and to add tags to the free pool.

9.2.2 void memMoveOutPosted(u32 source, reg64 dest, int len)

Transfer the specified number of bytes from *source* in local data RAM to *dest* in buffer RAM or DDR memory.

Parameters:

source	Source address of transfer in local data RAM.
dest	Destination address of transfer in buffer RAM or DDR memory.
len	Number of bytes to be transferred. Must be a multiple of 4.

Returns:

N/A

9.2.3 void memAddPosted(reg64 addr, u32 value)

Add *value* to the data at *addr* in buffer RAM or DDR memory.

Parameters:

addr	Specifies the DWord aligned address of the data that *value* will be added to.
value	Value to add to the data at *addr*.

Returns:

N/A

9.2.4 void memXorPosted(reg64 addr, u32 value)

XOR *value* to the data at *addr* in buffer RAM or DDR memory.

Parameters:

addr	Specifies the DWord aligned address of the data that *value* will be XOR'ed with.
value	Value to XOR to the data at *addr*.

Returns:

N/A

9.2.5 void memXferPosted(reg64 source, reg64 dest, int len)

Transfer the number of bytes specified by *len* between buffer RAM and DDR memory. *source* and *dest* must be DWord aligned and may not address the same memory type.

Parameters:

source	Source address of transfer.
dest	Destination address of transfer.
len	Number of bytes to be transferred. Must be a multiple of 4.

Returns:

N/A

9.2.6 int memMoveOut(u32 source, reg64 dest, int len)

Transfer the specified number of bytes from *source* in local data RAM to *dest* in buffer RAM or DDR memory. This request is non-posted, this means that a completion will be sent to inbound IPC message queue 0.

Parameters:

source	Source address of transfer in local data RAM.
dest	Destination address of transfer in buffer RAM or DDR memory.
len	Number of bytes to be transferred. Must be a multiple of 4.

Returns:

0xff - Hardware busy. The request should be re-issued by firmware.

1-32 - Request was accepted by the hardware and a tag returned.

9.2.7 int MemAdd(reg64 addr, u32 value)

Add *value* to the data at *addr* in buffer RAM or DDR memory. This request is non-posted, this means that a completion will be sent to inbound IPC message queue 0.

Parameters:

addr	Specifies the DWord aligned address of the data that *value* will be added to.
value	Value to add to the data at *addr*.

Returns:

0xff - Hardware busy. The request should be re-issued by firmware.

1-32 - Request was accepted by the hardware and a tag returned.

9.2.8 int memXor(reg64 addr, u32 value)

XOR *value* to the data at *addr* in buffer RAM or DDR memory. This request is non-posted, this means that a completion will be sent to inbound IPC message queue 0.

Parameters:

addr	Specifies the DWord aligned address of the data that *value* will be XOR'ed with.
value	Value to XOR to the data at *addr*.

Returns:

0xff - Hardware busy. The request should be re-issued by firmware.

1-32 - Request was accepted by the hardware and a tag returned.

9.2.9 int memXfer(reg64 source, reg64 dest, u32 len)

Transfer the number of bytes specified by *len* between buffer RAM and DDR memory. *source* and *dest* must be DWord aligned and may not address the same memory type. This request is non-posted, this means that a completion will be sent to inbound IPC message queue 0.

Parameters:

source	Source address of transfer.
dest	Destination address of transfer.
len	Number of bytes to be transferred. Must be a multiple of 4.

Returns:

0xff - Hardware busy. The request should be re-issued by firmware.

1-32 - Request was accepted by the hardware and a tag returned.

9.2.10 int memMoveIn(reg64 source, u32 dest, int len)

Transfer the specified number of bytes from *source* in buffer RAM or DDR memory to *dest* in local data RAM. This request is non-posted, this means that a completion will be sent to inbound IPC message queue 0.

Parameters:

source	Source address of transfer in buffer RAM or DDR memory.
dest	Destination address of transfer in local data RAM.
len	Number of bytes to be transferred. Must be a multiple of 4.

Returns:

0xff - Hardware busy. The request should be re-issued by firmware.

1-32 - Request was accepted by the hardware and a tag returned.

9.2.11 int memMoveInAndLock(reg64 source, u32 dest, int len, int lockBit)

Transfer the specified number of bytes from *source* in buffer RAM or DDR memory to *dest* in local data RAM. The bit specified by *lockBit* is set within the data to prevent access from any other processor. If the lock was obtained the data transferred reflects the previous state of the lock bit. If the lock could not be obtained, an exception will be signaled in the completion and no data will be transferred.

This request is non-posted, this means that a completion for this request will be sent to inbound IPC message queue 0.

Parameters:

source	Source address of transfer in buffer RAM or DDR memory.
dest	Destination address of transfer in local data RAM.
len	Number of bytes to be transferred. Must be a multiple of 4.
lockBit	Specifies the position of the lock bit within the data being transferred (0-2047).

Returns:

0xff - Hardware busy. The request should be re-issued by firmware.

1-32 - Request was accepted by the hardware and a tag returned.

9.2.12 void memFreeTag(u32 tag)

This function is used to free a tag, that was returned from one of the non-posted Memory Network API calls, when it is no longer needed by the firmware. It returns the specified tag value, in the range of 1-32, to the pool of tags that may be used by the hardware. During initialization, firmware should call this function 32 times with tag values of 1-32 in order to fill the hardware's free tag pool.

Parameters:

tag	Tag number to be returned to the free tag pool.

Returns:

N/A

CRC Library

10.1 Overview

Under certain circumstances it may be necessary to calculate the CRC32 for a given data set. For this reason the Huron firmware includes a CRC library which may be linked with any of the firmware to provide this functionality.

10.1.1 crc crcFast(unsigned char const message[], int nBytes)

Function to calculate a CRC32 over the specified data.

Parameters:

message	Pointer to the beginning of the data for which the CRC should be calculated.
nBytes	Size, in bytes, of the data pointed to by *message*.

Returns:

CRC32 Calculated over the specified data.

Software Managed FIFO's

11.1 Overview

The software managed FIFO library allows software to easily maintain and manage FIFO's that used to queue pointer sized pieces of information. In the case of the 108Mini, a pointer is 32-bits in size. One restriction to the usage model of the FIFO library is that a value of -1 may not be pushed into a FIFO since the value -1 is used to denote that a FIFO contains no data.

```
#define FIFO_ENTRIES              0x20

/* FIFO data storage */
static void *fifoStorage[FIFO_ENTRIES];
static DATA_FIFO    myFIFO;

int main(int argc, char *argv[])
{
       int n;
       void *data;

       /* Initialize the FIFO */
       fifoInit(&myFIFO, fifoStorage, FIFO_ENTRIES);

       /* Fill the FIFO */
       for(n=0; ; n++)
       {
              if(fifoPush(&myFIFO, (void *) n))
                     break;
       }

       /*Empty the FIFO */
       while((data = fifoPop(&myFIFO)) != (void *) -1)
              printf("Popped 0x%x\n", (int) data);

       return(0);
}
```

Figure 11.1 Software Managed FIFO Usage Example

11.2 FIFO Library API

The following sections document the software managed FIFO library API. For the most up to date documentation please refer to the software managed FIFO source code located in the Huron CVS repository under *huron/software/Firmware/EFC/libFIFO*.

11.2.1 Library Files

The following static libraries will be created as a result of building the FIFO library. These libraries should be linked to any application that utilizes the software managed FIFO queues.

- *libDelaware.a*

11.2.2 Include Files

The following include files should be included in order to use the software managed FIFO API.

- *fifo.h*

11.2.3 void fifoInit(DATA_FIFO *fifo, void **storage, int maxEntries)

Initialize a software managed FIFO queue and assign data storage to it.

Parameters:

fifo	A pointer to a DATA_FIFO structure for this FIFO.
storage	A pointer to an array of pointers to use as storage for the FIFO data.
maxEntries	The maximum number of entries the storage memory will hold.

Returns:

N/A

11.2.4 int fifoPush(DATA_FIFO *fifo, void *data)

Push a pointer sized piece of data into the FIFO.

Parameters:

fifo	A pointer to the DATA_FIFO structure for this FIFO.
data	Pointer sized data to be pushed into the FIFO.

Returns:

0 The operation failed due to the FIFO being full.
1 The operation was successful.

11.2.5 void *fifoPop(DATA_FIFO *fifo)

Pop a pointer sized piece of data from the FIFO.

Parameters:

fifo	A pointer to the DATA_FIFO structure for this FIFO.

Returns:

The pointer sized data popped from the FIFO on success, (void *) -1 if the FIFO is empty.

11.2.6 void *fifoPeek(DATA_FIFO *fifo)

Peek at the data located at the head of the FIFO.

Parameters:

fifo	A pointer to the DATA_FIFO structure for this FIFO.

Returns:

The pointer sized data located at the head of the FIFO on success, (void *) -1 if the FIFO is empty.

11.2.7 int fifoIsEmpty(DATA_FIFO *fifo)

Check to see if the specified FIFO contains data.

Parameters:

fifo	A pointer to the DATA_FIFO structure for this FIFO.

Returns:

0 The specified FIFO contains data.
1 The specified FIFO is empty.

11.3 FIFO Data Structure

A data structure of the following type is required for FIFO queue defined. The structure contains a pointer to the head, tail, base and limit of the FIFO data storage memory, the maximum number of entries the FIFO can hold and the current number of entries in the FIFO.

```
typedef struct {
      void  **head;
      void  **tail;
      void  **base;
      void  **limit;
      int   size;
      int   entries;
} DATA_FIFO;
```

IPC Messages

12.1 Overview

The following sections describe the Inter Processor Communication messages used by the CPU's to communicate with each other and with hardware blocks such as the I/E DMA engines. A high level description of the messages used to communicate with hardware blocks will be included below, but the actual format for these messages will be described in the FSD.

12.2 Functional Unit Messages

The messages in this section are used to communicate with the functional units of the system and are described in full in the FSD.

12.2.1 CmdBufFree

Message sent to the buffer RAM manager to release the memory that was previously allocated for an eNVMHCI command.

12.2.2 CmdReady

Message sent to one of the Data Managers to inform it of a new eNVMHCI command. This message will include the address of the command in buffer RAM.

12.2.3 DMACpl

Message sent from one of the DMA engines to signal completion of a DMA transfer.

12.2.4 EgressDMA

Message sent to the Egress DMA engine to initiate a transfer of data from buffer RAM or DDR memory to host memory.

12.2.5 EgressTLP

Message sent to the host controller hardware requesting that it send a TLP to the host.

12.2.6 EgressTLPCpl

Message sent in response to an EgressTPL message.

12.2.7 FlashErase

Message sent to the flash controller to request the erasure of a flash block.

12.2.8 FlashOpCpl

Message received from thet flash controller to signal the completion of a flash operation (read/write/erase).

12.2.9 FlashRead

Message sent to the flash controller to read a flash page.

12.2.10 FlashWrite

Message send to the flash controller to write a flash page.

12.2.11 IngressDMA

Message sent to the Ingress DMA engine to initiate a transfer of data from host memory to the buffer RAM or DDR memory.

12.2.12 IngressTLP

Message sent by the host controller hardware when a TLP is received by the host.

12.2.13 StsQueueUpdate

Message sent to the Host Controller signaling that a command has completed and a command status should be written to the host and an interrupt generated.

12.2.14 StsQueueUpdateCpl

Message received from the Host Controller when the status of a completed command has been written to host memory.

12.3 Inter-Manager Messages

The following messages are used by managers within the processor complex.

12.3.1 Block Open Request

This message is sent to the Wearleveling Manager to open a new flash block and optionally close a flash block. If the T (index valid) bit is set, the Wearleveling Manager will attempt to allocate a free block for the index specified by the INDEX field. If the T bit is not set, the Wearleveling Manager may allocate a block from any index. If the B bit is set, the BLOCK_NUMBER field contains the number of a block that should be closed. This removes the need for a separate Block Close message.

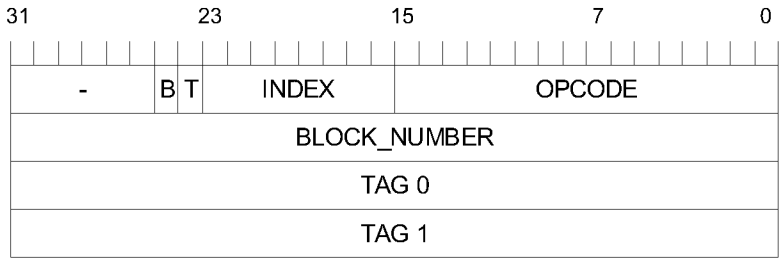


Figure 12.1 Block Open Request Message Format

Field Name	Value	Description
OPCODE	0x0000	Message opcode.
INDEX	0-255	If the T bit is set, this field specifies the 8-bit index number for which a block is being requested.
T	-	Index valid bit, specifies that the INDEX field is valid.
B	-	Block number valid bit, specifies that the BLOCK_NUMBER field is valid.
BLOCK_NUMBER	-	Number of the block that is being closed by this request.
TAG 0	-	Transparent data passed back in Block Open Reply
TAG 1	-	Transparent data passed back in Block Open Reply

Table 12.1 Block Open Request Message Fields

12.3.2 Block Open Reply

This message is sent from the Wearleveling Manager in response to a Block Open Request message.

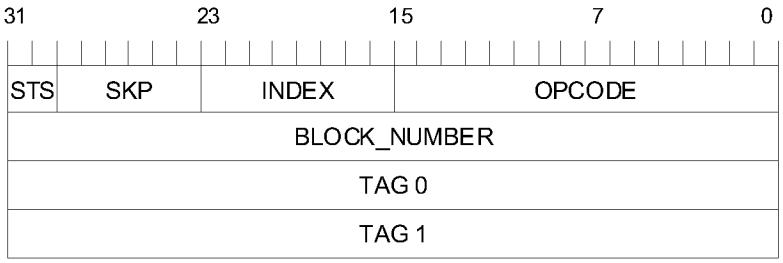


Figure 12.2 Block Open Reply Message Format

Field Name	Value	Description
OPCODE	0x0001	Message opcode.
INDEX	-	Specifies the index that the allocated block belongs to.
SKP	-	Skip count, skip this block every SKP requests. 000000b = don't skip, 000001b = skip every other, 000010b = skip every two, 111111b = always skip.
STS	-	Status of the block request: 0 - Success (erased block allocated). 1 - Block could not be allocated. 2 - XOFF, block allocated for garbage collection use only. This block should not be used to service write page requests from the Data Managers until an XON message is received.
BLOCK_NUMBER	-	Physical block number. Undefined if STATUS is non-zero.
TAG 0	-	Transparent data from the Block Open Request message.

Table 12.2 Block Open Reply Message Fields

Field Name	Value	Description
TAG 1	-	Transparent data from the Block Open Request message.

Table 12.2 Block Open Reply Message Fields

12.3.3 Block Close

This message is sent to the Wearleveling Manager to close an NVM block that was opened with the Block Open Request message.

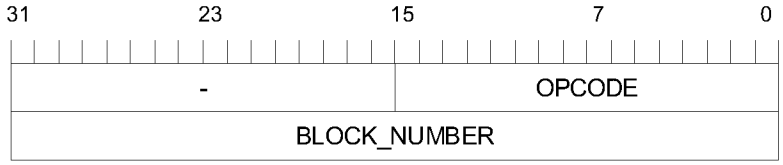


Figure 12.3 Block Close Message Format

Field Name	Value	Description
OPCODE	0x0002	Message opcode.
BLOCK_NUMBER	-	Block number of the block being closed.

Table 12.3 Block Close Message Fields

12.3.4 Sync Request

A Sync Request message is sent from one processor to another in order to synchronize them. When the destination processor receives the Sync Request it should respond with a Sync Reply message.

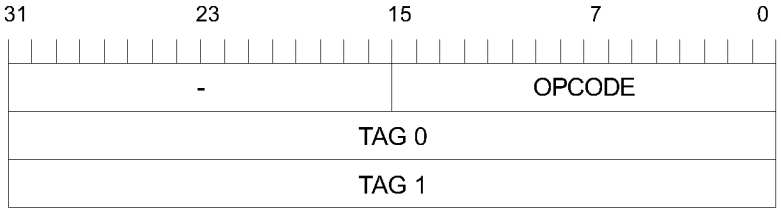


Figure 12.4 Sync Reply Message Format

Field Name	Value	Description
OPCODE	0x0003	Message opcode.
TAG 0	-	Transparent data returned in the Sync Reply message.
TAG 1	-	Transparent data returned in the Sync Reply message.

Table 12.4 Sync Request Message Fields

12.3.5 Sync Reply

This message is sent in response to a Sync Request message.

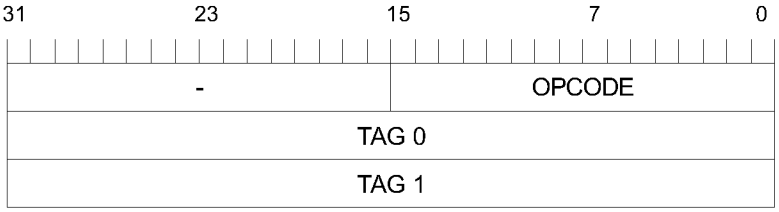


Figure 12.5 Sync Reply Message Format

Field Name	Value	Description
OPCODE	0x0004	Message opcode.
TAG 0	-	Transparent data from the Sync Request message.
TAG 1	-	Transparent data from the Sync Request message.

Table 12.5 Sync Reply Message Fields

12.3.6 Page Write

This message is sent from the Data Manager to the Write Manager to request data be written to the specified flash page. The format of this message is identical to the FlashWrite message except that when received from the Data Manager, the **FlashAddr** field will contain the current physical page address *(jack: should this be logical?)* if the page is being overwritten or 0xffffffff if this is the first time a page is being written. This field will be overwritten by the physical page address allocated by the Write Manager before this message is forwarded to the flash controller. The TAG1 field will be used to determine if this message is being sent due to a previous failure reported by the FCC. A non-zero value in the TAG1 field indicates that this message was sent due to a previous failure reported by the FCC.

12.3.7 Obsolete Page

Message sent to the Wearleveling Manager stating that a page is being written and one is possibly being obsoleted.

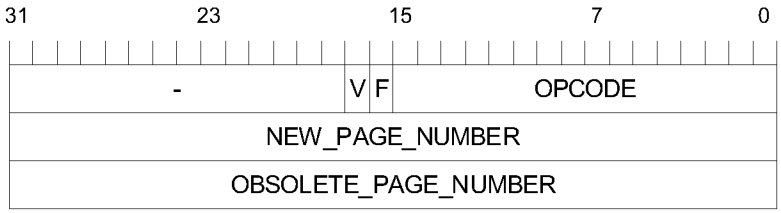


Figure 12.6 Obsolete Page Message Format

Field Name	Value	Description
OPCODE	0x0005	Message opcode.
V	-	Valid bit. This bit specifies that the OBSOLETE_PAGE_NUMBER field is valid.
F	-	The page specified by OBSOLETE_PAGE_NUMBER is being obsoleted due to a failure.
NEW_PAGE_NUMBER	-	The number of the new page being written.
OBSOLETE_PAGE_NUMBER	-	The number of the page being obsoleted.

Table 12.6 Obsolete Page Message Fields

12.3.8 Page Frame Init

Message sent to the Page Frame Manager which it uses to initialize the free page frame lists.

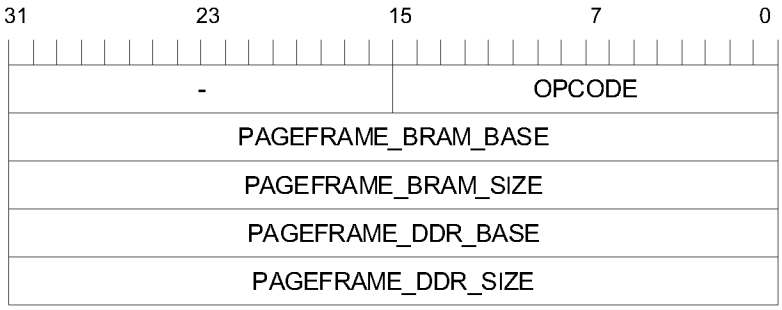


Figure 12.7 Page Frame Init Message Format

Field Name	Value	Description
OPCODE	0x0006	Message opcode.
PAGEFRAME_BRAM_BASE	-	Bits 36:5 of a 37 bit address that specifies the base address of the page frame buffer array located in buffer RAM.
PAGEFRAME_BRAM_SIZE	-	Size in bytes of the memory pointed to by PAGEFRAME_BRAM_BASE.
PAGEFRAME_DDR_BASE	-	Bits 36:5 of a 37 bit address that specifies the base address of the page frame buffer array located in DDR memory.
PAGEFRAME_DDR_SIZE	-	Size in bytes of the memory pointed to by PAGEFRAME_DDR_BASE.

Table 12.7 Page Frame Init Message Fields

12.3.9 Page Frame Alloc Request

Message sent to the Page Frame Manager to request one or more page frame buffers.

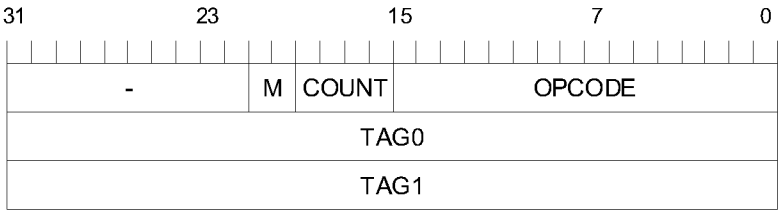


Figure 12.8 Page Frame Alloc Message Format

Field Name	Value	Description
OPCODE	0x0007	Message opcode.
COUNT	-	Number of page frame buffers being requested in the range of 1-15.
M	-	Used to specify which memory pool the page frame should be allocated from: 0 - Buffer RAM 1 - DDR Memory 2 - Don't care (Page Frame Manager may allocate from any pool)
TAG 0	-	Tag value to be passed back in the Page Frame Alloc Reply.
TAG 1	-	Tag value to be passed back in the Page Frame Alloc Reply.

Table 12.8 Page Frame Alloc Message Fields

12.3.10 Page Frame Free

Message sent to the Page Frame Manager to free one or more page frames that are no longer needed.

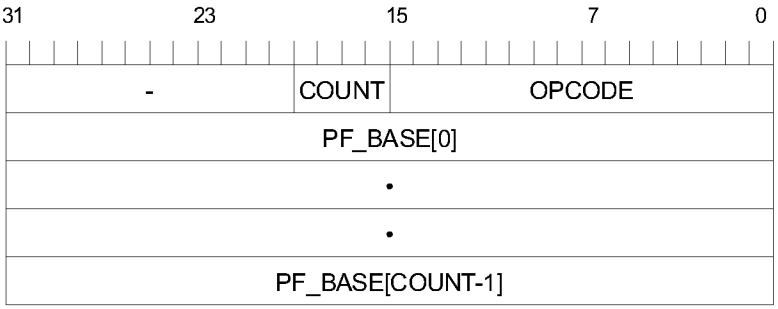


Figure 12.9 Page Frame Free Message Format

Field Name	Value	Description
OPCODE	0x0008	Message opcode.

Table 12.9 Page Frame Free Message Fields

Field Name	Value	Description
COUNT	-	Number of page frame buffers being freed, should be in the range of 1-15.
PF_BASE[n]	-	Bits 36:5 of the 37 bit address that points to the page frame to be freed.

Table 12.9 Page Frame Free Message Fields

12.3.11 Page Frame Alloc Reply

Message sent from the Page Frame Manager in response to a Page Frame Alloc Request message. This reply may contain 0-COUNT page frame buffer addresses.

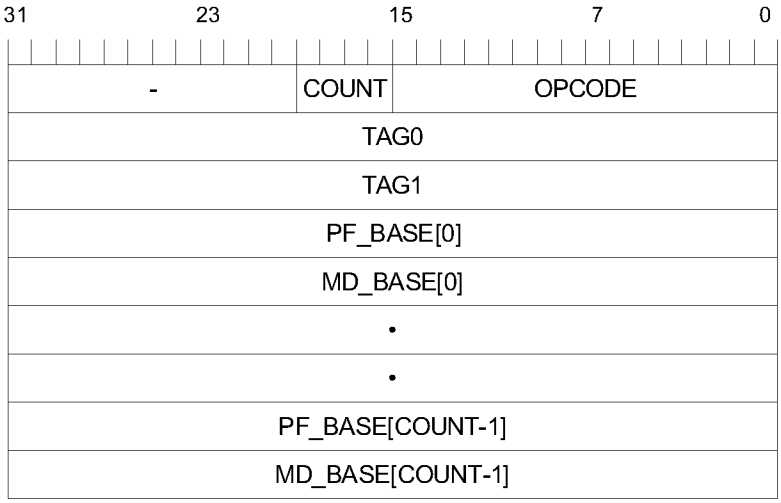


Figure 12.10 Page Frame Alloc Reply Message Format

Field Name	Value	Description
OPCODE	0x0009	Message opcode.
COUNT	-	Number of page frame buffers being returned, in the range of 0-15.
TAG 0	-	Tag value from Page Frame Alloc Request.
TAG 1	-	Tag value from Page Frame Alloc Request.
PF_BASE[n]	-	Bits 36:5 of a 37 bit address that points to the page frame buffer.
MD_BASE[n]	-	Bits 36:5 of a 37 bit address that points to the meta-data buffer.

Table 12.10 Page Frame Alloc Reply Message Fields

12.3.12 Power Down Request

Message sent all processors from the control processor to do any operations required to get to a quiessed state. This is used for taking a snapshot, as well as saving the system state for power down.

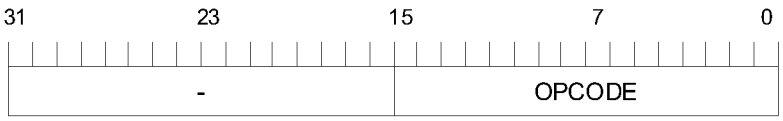


Figure 12.11 Power Down Request Message Format

Field Name	Value	Description
OPCODE	0x000b	Message opcode.

Table 12.11 Power Down Request Message Fields

12.3.13 Power Down Complete

This message is sent in response to a Power Down Request to signal that the system power down is complete.

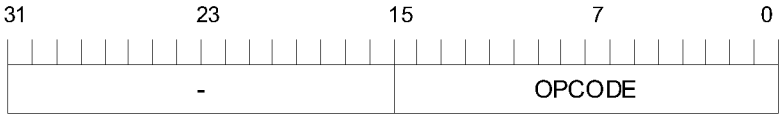


Figure 12.12 Power Down Reply Message Format

Field Name	Value	Description
OPCODE	0x000c	Message opcode.

Table 12.12 Power Down Reply Message Fields

12.3.14 Power Up Request

Message sent to all processors from the control processor to do any operations required to remove itself from the quiessed state and move to an active state.

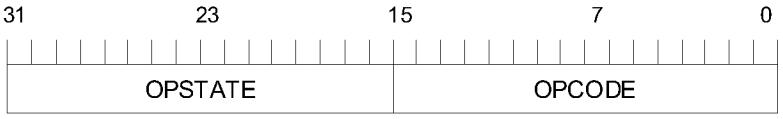


Figure 12.13 Power Up Request Message Format

Field Name	Value	Description
OPCODE	0x000d	Message opcode.

Table 12.13 Power Up Request Message Fields

Field Name	Value	Description
OPSTATE	-	This data specifies the operating state we are to power up in: **0: First Initialization Powerup.** Create necessary data structures in DRAM partitions. **1: Normal mode.** DRAM has been loaded from previous saved state. **2: Error.** An error occured during power up. Do not accept read or write commands. Do allow NVMHCI commands to allow the user to take action. **3: Reformat:** Error recovery. Use bad block state iformation, but erase all blocks and re-initialize data structures.

Table 12.13 Power Up Request Message Fields

12.3.15 Power Up Complete

This message is sent in response to a Power Down Request to signal that the system power up is complete.

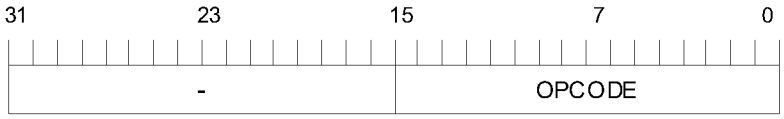


Figure 12.14 Power Up Reply Message Format

Field Name	Value	Description
OPCODE	0x000e	Message opcode.

Table 12.14 Power Up Reply Message Fields

12.3.16 Reclaim Block Request

Message sent from the Wear Level Manager to the Garbage Collection Manager to reclaim a block. The Garbage Collection Manager copies all valid pages from this block and requests the Write Manager to write those page data back to the flash memory. The block is then erased. After the erasure, the block info data is updated and a response is then sent back to the Wear Level Manger. There can only be one outstanding Reclaim Block Request per target.

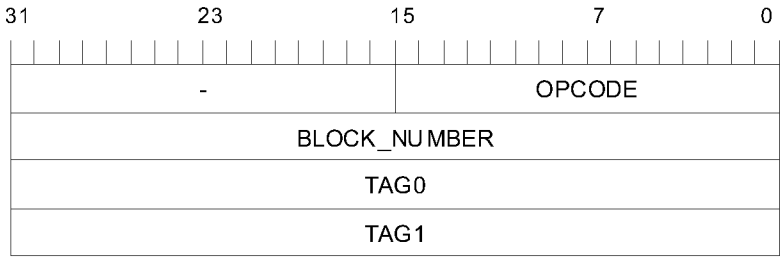


Figure 12.15 Reclaim Block Request Message Format

Field Name	Value	Description
OPCODE	0x0010	Message opcode.
BLOCK_NUMBER	-	The flash block that needs to be reclaimed
TAG 0	-	Transparent data passed back in Reclaim Block Reply
TAG 1	-	Transparent data passed back in Reclaim Block Reply

Table 12.15 Reclaim Block Request Message Fields

12.3.17 Reclaim Block Reply

Message sent from the Garbage Collection Manger to the Wear Level Manager in response to the Reclaim Block Reply. If the reclaim is successful, the block is erased and all block info has been updated. The Wear Level Manager can just queue this block into the free queue and the block is ready to be programmed again.

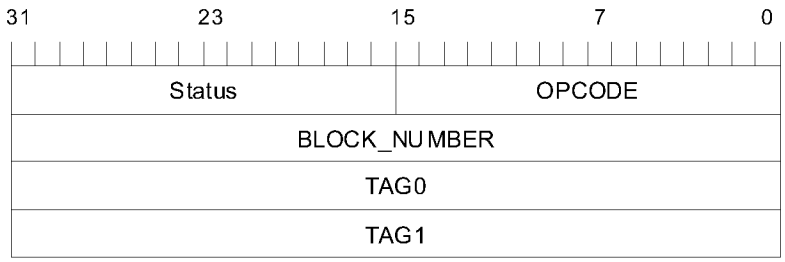


Figure 12.16 Reclaim Block Reply Message Format

Field Name	Value	Description
OPCODE	0x0011	Message opcode.
Status		Status: 0: Success, the block has been erased and can be put back to the free queue. 1: Block is a BAD block. Do not use anymore. All valid page has been moved to other pages. 2: Read error on certain pages. Block should be retired as BAD block. 3: Unknown block number.
BLOCK_NUMBER	-	The flash block that has been reclaimed
TAG 0	-	Transparent data passed back in Reclaim Block Reply
TAG 1	-	Transparent data passed back in Reclaim Block Reply

Table 12.16 Reclaim Block Reply Message Fields

12.3.18 XON

Message sent to the Write Manager requesting that it resume the servicing of Data Manager page write requests for the specified target index.

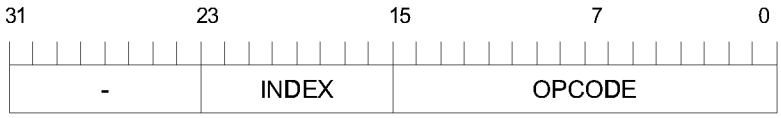


Figure 12.17 XON Message Format

Field Name	Value	Description
OPCODE	0x0012	Message opcode.
INDEX	-	Index of the target the XON message relates to.

Table 12.17 XON Message Fields

12.3.19 New Command

This message is sent from the Data Manager to one of the other managers to request that they process an eNVMHCI command located in buffer RAM. The receiving manager is responsible for freeing the command buffer once it has processed the command.

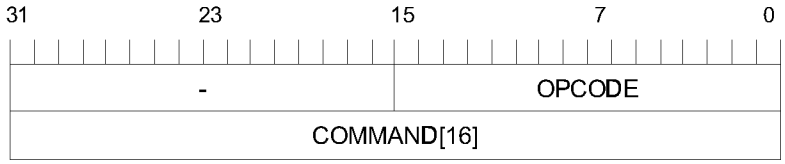


Figure 12.18 New Command Message Format

Field Name	Value	Description
OPCODE	0x0013	Message opcode.
COMMAND	-	Dwords that constitute the eNVMHCI command.

Table 12.18 New Command Message Fields

12.3.20 List Remove Posted

This message is sent to the manager that is responsible for linked list management to remove an entry from a linked list.

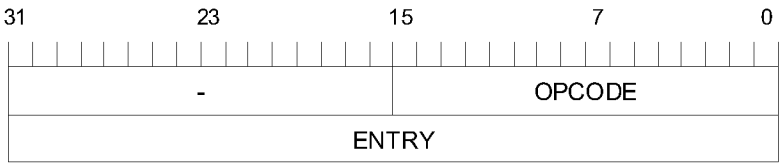


Figure 12.19 List Remove Posted Message Format

Field Name	Value	Description
OPCODE	0x0014	Message opcode.
ENTRY	-	32-bit address of entry to be removed.

Table 12.19 List Remove Posted Message Fields

12.3.21 List Add Head Posted

This message is sent to the manager that is responsible for linked list management to add an entry to the head of a linked list.

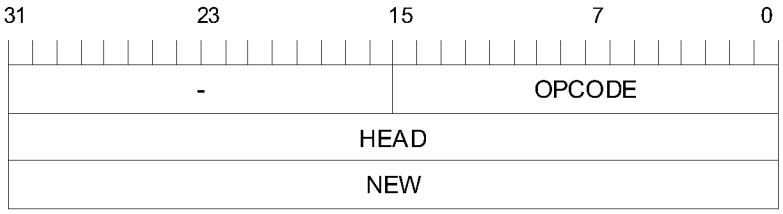


Figure 12.20 List Add Head Posted Message Format

Field Name	Value	Description
OPCODE	0x0015	Message opcode.
HEAD	-	32-bit address of the head of the list.
NEW	-	32-bit address of the new entry to be added to the head of the list.

Table 12.20 List Add Head Posted Message Fields

12.3.22 List Add Tail Posted

This message is sent to the manager that is responsible for linked list management to add an entry to the tail of a linked list.

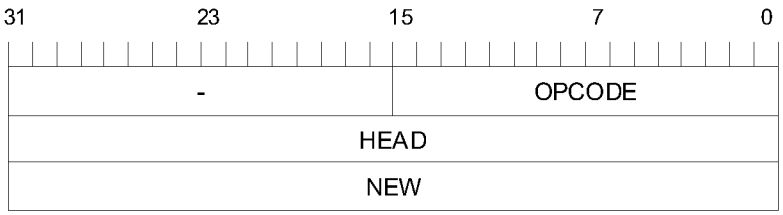


Figure 12.21 List Add Tail Posted Message Format

Field Name	Value	Description
OPCODE	0x0016	Message opcode.
HEAD	-	32-bit address of the head of the list.
NEW	-	32-bit address of the new entry to be added to the tail of the list.

Table 12.21 List Add Tail Posted Message Fields

12.3.23 List Remove Head

This message is sent to the manager that is responsible for linked list management to remove an entry from the head of a linked list. This message is non-posted meaning that a List Remove Reply is sent to the requester upon completion of the operation.

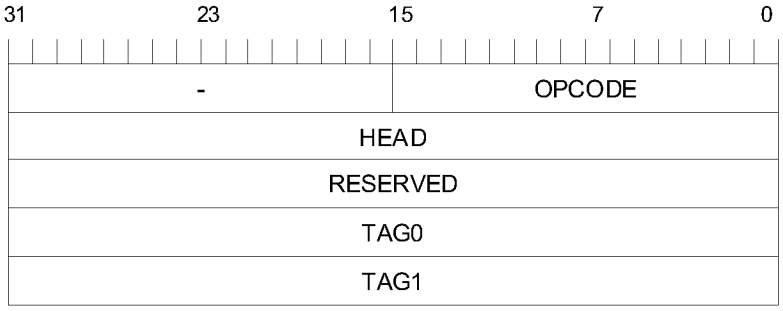


Figure 12.22 List Remove Head Message Format

Field Name	Value	Description
OPCODE	0x0017	Message opcode.
HEAD	-	32-bit address of the head of the list.
RESERVED	-	Reserved.
TAG0	-	32-bits of transparent data copied to List Remove Reply message.
TAG1	-	32-bits of transparent data copied to List Remove Reply message.

Table 12.22 List Remove Head Message Fields

12.3.24 List Remove Tail

This message is sent to the manager that is responsible for linked list management to remove an entry from the tail of a linked list. This message is non-posted meaning that a List Remove Reply is sent to the requester upon completion of the operation.

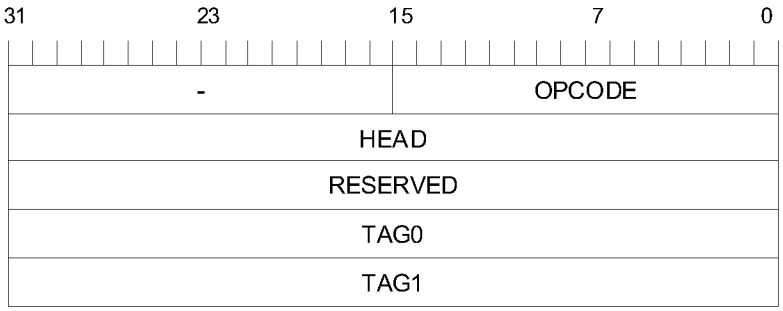


Figure 12.23 List Remove Tail Message Format

Field Name	Value	Description
OPCODE	0x0018	Message opcode.
HEAD	-	32-bit address of the head of the list.
RESERVED	-	Reserved.
TAG0	-	32-bits of transparent data copied to List Remove Reply message.
TAG1	-	32-bits of transparent data copied to List Remove Reply message.

Table 12.23 List Remove Tail Message Fields

12.3.25 List Remove Reply

Message sent by the manager responsible for linked list management in response to a List Remove Head or List Remove Tail message.

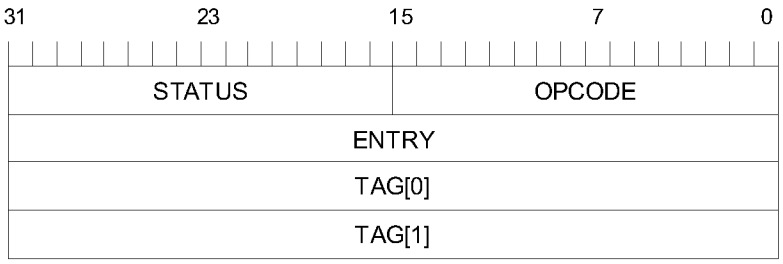


Figure 12.24 List Remove Reply Message Format

Field Name	Value	Description
OPCODE	0x0019	Message opcode.
STATUS	0 - Success 1 - List is empty	Completion status of the operation.
ENTRY	-	32-bit address of the entry removed from the list.
TAG0	-	32-bits of transparent data copied to List Remove Reply message.
TAG1	-	32-bits of transparent data copied to List Remove Reply message.

Table 12.24 List Remove Reply Message Fields

****

****

EVERYTHING BELOW THIS POINT IS FROM HURON AN NOT REQUIRED BUT NOT REMOVED YET. DO NOT REMOVE!!!

****

****

This message is sent from the Interface Manager to the PCIe Manager to inform it of the completion of the NVMHCI commands specified by the *CMD_ISSUE* field. Bits that are set in the *CMD_ISSUE* field will be cleared in the NVMHCI Command Issue register by the PCIe Manager and if the I field is set an interrupt will be generated to the host computer.

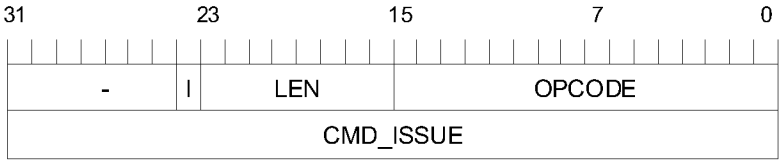


Figure 12.25 NVMHCI_CMD_DONE Message Format

Field Name	Value	Description
OPCODE		Message opcode.
LEN	0x02	Length of message in DWORDS.
I	-	Set if the PCIe Manager should interrupt the host after clearing the appropriate bits in the NVMHCI Command Issue register.
CMD_ISSUE	-	Bit vector representing the entries in the command list for which processing is complete. The PCIe Manager will clear the corresponding bits in the NVMHCI Command Issue register.

Table 12.25 NVMHCI_CMD_DONE Message Fields

12.4 WRITE_REQUEST

This message is sent by the Interface Manager to the Write Manager to request that the sectors specified by *SECTOR_VEC* in the page frame pointed to by *PAGE_FRAME_ADDR* be written to the flash or cache.

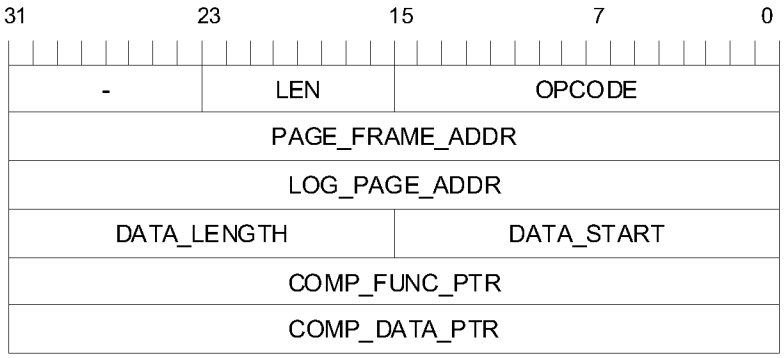


Figure 12.26 WRITE_REQUEST Message Format

Field Name	Value	Description
OPCODE		Message opcode.
LEN	0x06	Length of message in DWords.
PAGE_FRAME_ADDR	-	Address in Buffer RAM of the page that contains data to be written.
LOG_PAGE_ADDR	-	Logical address of the page being written.

Table 12.26 WRITE_REQUEST Message Fields

Field Name	Value	Description
DATA_START	-	Start offset of the data within the NVM page to be written.
DATA_LENGTH	-	Length of the data within the NVM page to be written.
COMP_FUNC_PTR	-	Completion function pointer passed on to the WRITE_COMPLETION message. This entry may be NULL.
COMP_DATA_PTR	-	Completion data pointer, passed on to the WRITE_COMPLETION message. This entry may be NULL.

Table 12.26 WRITE_REQUEST Message Fields

12.5 FLUSH_ALL

This message is sent to the Data Buffer Manager requesting that the read cache be cleared and any cached writes be flushed to flash.

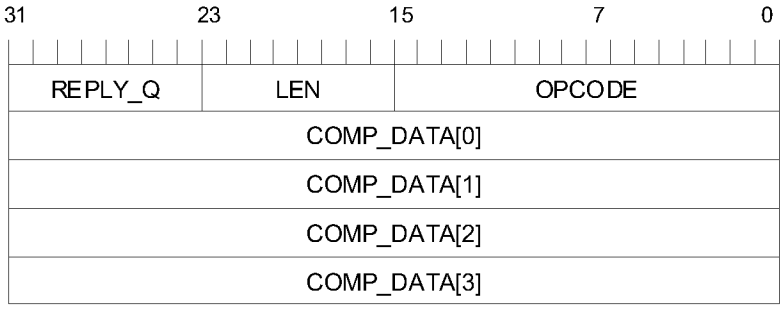


Figure 12.27 FLUSH_ALL Message Format

Field Name	Value	Description
OPCODE	FLUSH_ALL	Message opcode.
LEN	0x01 - 0x05	Length of message in DWords. Ranges from 1 - 5 DWords based on the amount of completion data the message is carrying.
REPLY_Q	-	ID of the message queue where the reply to this message should be sent.
COMP_DATA[0-3]	-	Up to 4 DWords of completion data. This data is appended to the *FLUSH_ALL_REPLY* message.

Table 12.27 FLUSH_ALL Message Fields

12.6 FLUSH_ALL_REPLY

This message is sent from the Data Buffer Manager to the queue specified in the *REPLY_Q* field of the *FLUSH_ALL* message after the Data Buffer Manager has invalidated the read cache and flushed any cached writes to flash.

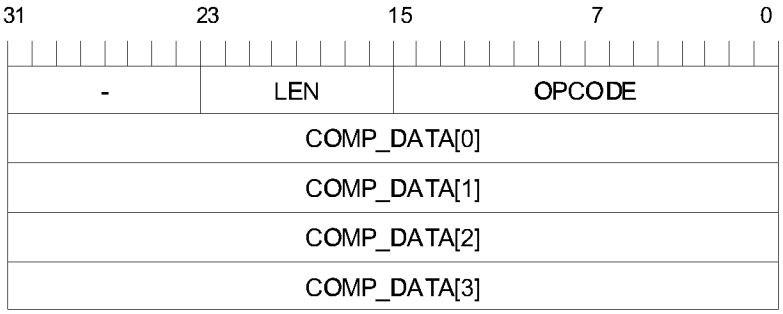


Figure 12.28 FLUSH_ALL_REPLY Message Format

Field Name	Value	Description
OPCODE	FLUSH_ALL_REPLY	Message opcode.
LEN	0x01 - 0x05	Length of message in DWords. Ranges from 1 - 5 DWords based on the amount of completion data the message is carrying.
COMP_DATA[0-3]	-	Up to 4 DWords of completion data.

Table 12.28 FLUSH_ALL_REPLY Message Fields

12.7 LOG_ENTRY

This message is sent to the Logging Manager to request that the log entry contained within the LOG_DATA[0-3] fields be added to the currently open log.

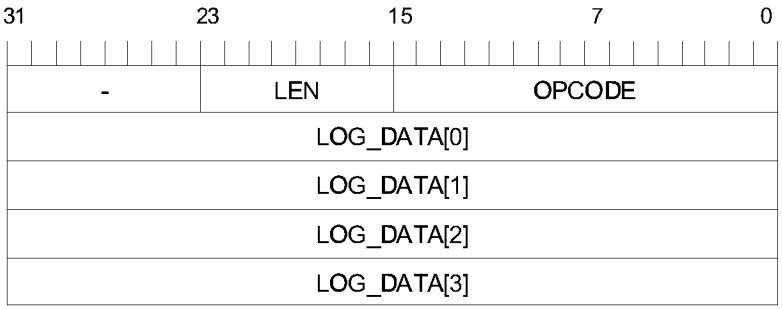


Table 12.29 LOG_ENTRY Message Format

Field Name	Value	Description
OPCODE	LOG_ENTRY	Message opcode.
LEN	0x02-0x05	Length of message in DWords.
LOG_DATA	-	Up to 4 DWords of log entry information.

Table 12.30 LOG_ENTRY Message Fields

12.7.1 Page Map Log Entry

This type of log entry is used to log a change to the logical to physical page mapping table.

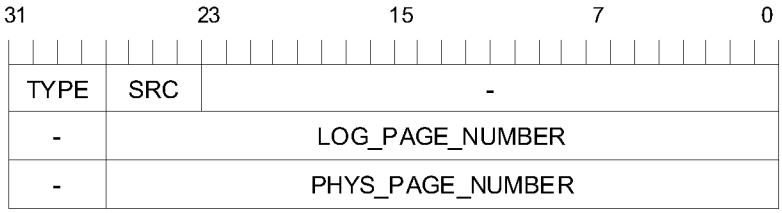


Figure 12.29 Page Mapping Table Log Entry Format

Field Name	Value	Description
TYPE	0x0	Page mapping table log entry.
SRC	-	ID of the manager that initiated this log message.
LOG_PAGE_NUMBER	-	Logical page address that maps to PHYS_PAGE_NUMBER.
PHYS_PAGE_NUMBER	-	Physical page address that LOG_PAGE_NUMBER maps to.

Table 12.31 Page Mapping Table Log Entry Fields

12.7.2 Erase Count Log Entry

This type of log entry is used to log changes to the NVM block erase counts.

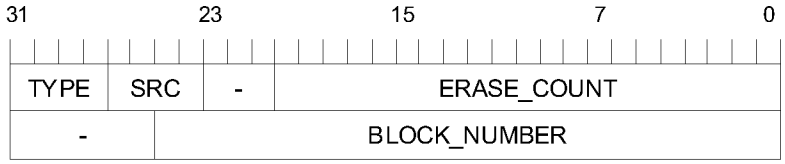


Figure 12.30 Erase Count Log Entry Format

Field Name	Value	Description
TYPE	0x2	Erase count log entry.
SRC	-	ID of the manager that initiated this log message.
BLOCK_NUMBER	-	Physical block number.
ERASE_COUNT	-	Number of times the specified block has been erased.

Table 12.32 Erase Count Log Entry Fields

12.7.3 Bad Block Log Entry

This type of log entry is used to log changes (additions) to the bad block table.

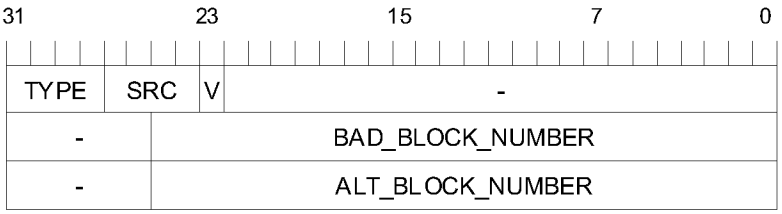


Figure 12.31 Bad Block Log Entry Format

Field Name	Value	Description
TYPE	0x3	Bad block log entry.
SRC	-	ID of the manager that initiated this log message.
V	-	This bad block log entry includes an alternate block number.
BAD_BLOCK_NUMBER	-	Physical block number.
ALT_BLOCK_NUMBER	-	The block number of the block that should be used instead of the bad block.

Table 12.33 Bad Block Log Entry Fields

12.7.4 Block State Log Entry

This type of log entry is used to log changes to a blocks state.

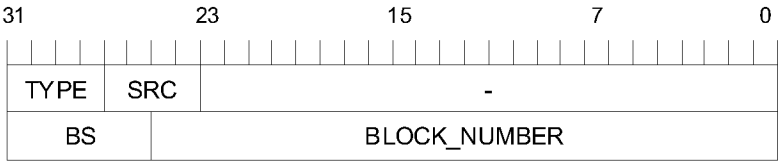


Figure 12.32 Block State Log Entry Format

Field Name	Value	Description
TYPE	0x4	Block state log entry.
SRC	-	ID of the manager that initiated this log message.
BS	-	The new state of the block.
BLOCK_NUMBER	-	Physical block number.

Table 12.34 Block State Log Entry Fields

12.7.5 Block Valid Page Log Entry

This message is sent from the Wearleveling Manager to the Logging Manager to log the fact that the valid page bit vector for the specified block has changed.

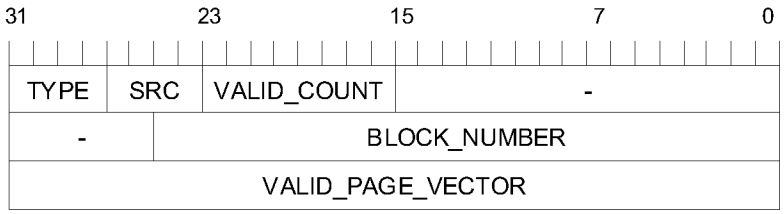


Figure 12.33 Block Valid Page Log Entry Fields

Field Name	Value	Description
TYPE	0x4	Block state log entry.
SRC	-	ID of the manager that initiated this log message.
VALID_COUNT	-	The number of valid pages in this block.
BLOCK_NUMBER	-	Physical block number.
VALID_PAGE_VECTOR	-	Bit vector representing which pages within the block are valid.

Table 12.35 Block Valid Page Log Entry Fields

12.7.6 Next Block Log Entry

This should be the last log entry written to the last page of incremental log data within a log block. This log entry signifies that this is the last entry in this page, but more log pages are located in the block specified by the BLOCK_NUMBER field.

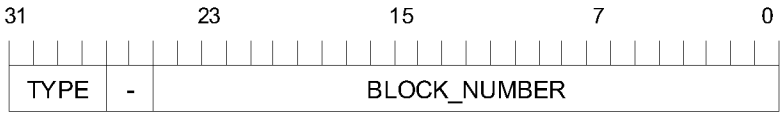


Figure 12.34 Next Block Log Entry Fields

12.8 BLOCK_FREE_REQUEST

Message sent to the Wearleveling Manager stating that the specified block may be erased.

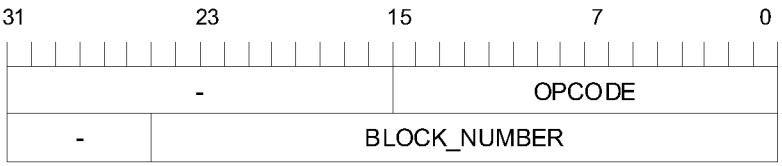


Figure 12.35 BLOCK_FREE_REQUEST Message Format

Field Name	Value	Description
OPCODE	BLOCK_FREE_REQUEST	Message opcode.
BLOCK_NUMBER	-	Block number to be erased..

Table 12.36 BLOCK_FREE_REQUEST Message Fields

12.9 PAGE_WRITE

Message sent to the Wearleveling Manager stating that a page is being written and one is possibly being obsoleted.

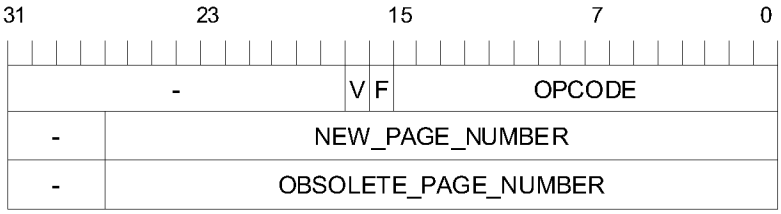


Figure 12.36 PAGE_WRITE Message Format

Field Name	Value	Description
OPCODE	PAGE_WRITE	Message opcode.
V	-	Valid bit. This bit specifies that the OBSOLETE_PAGE_NUMBER field is valid.
F	-	Specifies that the write to the new page failed.
NEW_PAGE_NUMBER	-	The number of the new page being written.
OBSOLETE_PAGE_NUMBER	-	The page number being obsoleted.

Table 12.37 PAGE_WRITE Message Fields

12.10 PAUSE_OPERATION

Message sent to the Wearleveling Manager requesting that it pause all wear leveling operations.

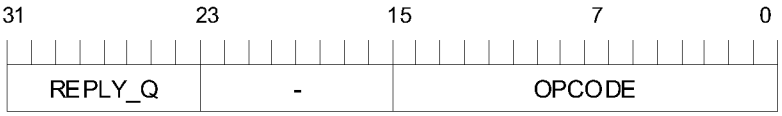


Figure 12.37 PAUSE_OPERATION Message Format

Field Name	Value	Description
OPCODE	PAUSE_OPERATION	Message opcode.
REPLY_Q	-	Index of the IPC message queue where the reply to this message should be sent.

Table 12.38 PAUSE_OPERATION Message Fields

12.11 OPERATION_PAUSED

Message sent from the Wearleveling Manager once wear leveling has been paused due to a PAUSE_OPERATION message being received.

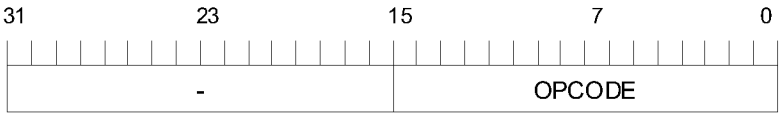


Figure 12.38 OPERATION_PAUSED Message Format

Field Name	Value	Description
OPCODE	OPERATION_PAUSED	Message opcode.

Table 12.39 OPERATION_PAUSED Message Fields

12.12 RESUME_OPERATION

Message sent to the Wearleveling Manager to resume wear leveling operations.

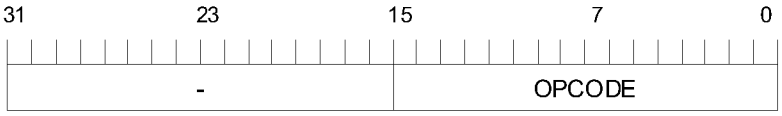


Figure 12.39 RESUME_OPERATION Message Format

Field Name	Value	Description
OPCODE	RESUME_OPERATION	Message opcode.

Table 12.40 RESUME_OPERATION Message Fields

12.13 INIT_DATA

Message sent to a mannager informing it to initialize its data structures. Upon completion of the data structure initialization an INIT_DATA_COMPLETE message should be sent to the source of the INIT_DATA message.

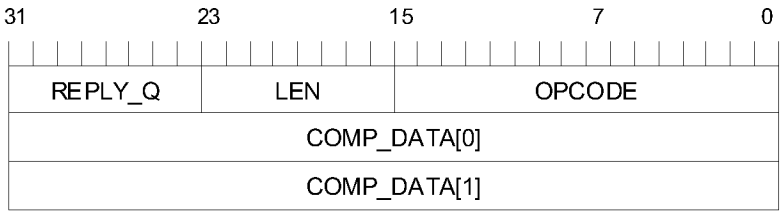


Figure 12.40 INIT_DATA Message Format

Field Name	Value	Description
OPCODE	INIT_DATA	Message opcode.
LEN	3	Length of this message in DWords.
REPLY_Q	-	Index of the outbound IPC message queue where the INIT_DATA_COMPLETE message should be sent.
COMP_DATA[0]	-	Completion data sent to requester in completion message.
COMP_DATA[1]	-	Completion data sent to requester in completion message.

Table 12.41 INIT_DATA Message Fields

12.14 INIT_DATA_COMPLETE

Message sent to the requester in response to an INIT_DATA message.

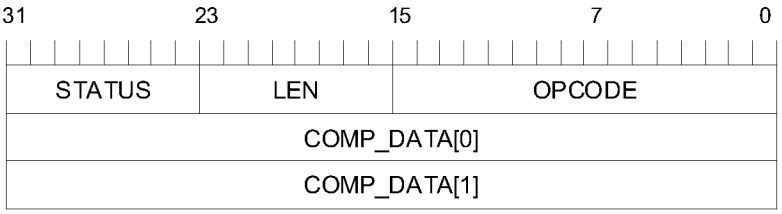


Figure 12.41 INIT_DATA_COMPLETE Message Format

Field Name	Value	Description
OPCODE	INIT_DATA_COMPLETE	Message opcode.
LEN	3	Length of this message in DWords.
STATUS	-	Status of the data initialize process.
COMP_DATA[0]	-	Completion data from INIT_DATA message.

Table 12.42 INIT_DATA_COMPLETE Message Fields

Field Name	Value	Description
COMP_DATA[1]	-	Completion data from INIT_DATA message.

Table 12.42 INIT_DATA_COMPLETE Message Fields

12.15 INITIATE_SNAPSHOT

Message sent to the Logging Manager in order to initiate the creation of a system snapshot.

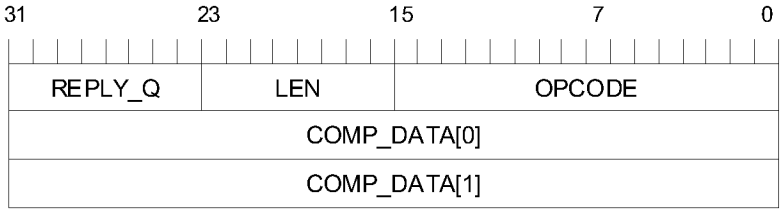


Figure 12.42 INITIATE_SNAPSHOT Message Format

Field Name	Value	Description
OPCODE	INITIATE_SNAPSHOT	Message opcode.
LEN	3	Length of this message in DWords.
REPLY_Q	-	Index of the outbound IPC message queue where the SNAPSHOT_COMPLETE message should be sent.
COMP_DATA[0]	-	Completion data sent to requester in completion message.
COMP_DATA[1]	-	Completion data sent to requester in completion message.

Table 12.43 INITIATE_SNAPSHOT Message Fields

12.16 SNAPSHOT_COMPLETE

Message sent from the Logging Manager to the snapshot requester upon completion of the snapshot.

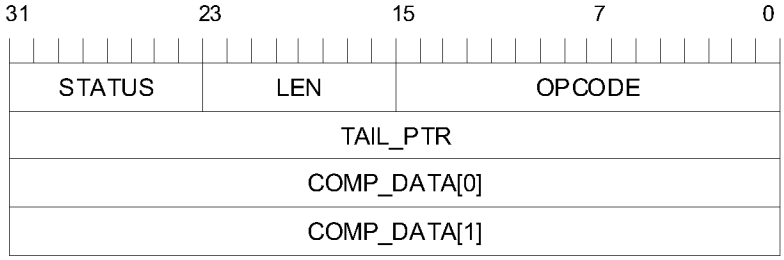


Figure 12.43 SNAPSHOT_COMPLETE Message Format

Field Name	Value	Description
OPCODE	SNAPSHOT_COMPLETE	Message opcode.
LEN	4	Length of this message in DWords.
STATUS	-	Status of the data initialize process.
TAIL_PTR	-	Page number of the last page in the snapshot image.
COMP_DATA[0]	-	Completion data from INITIATE_SNAPSHOT message.
COMP_DATA[1]	-	Completion data from INITIATE_SNAPSHOT message.

Table 12.44 SNAPSHOT_COMPLETE Message Fields

12.17 BAD_BLOCK

Message sent to the Wearleveling Manager in order to flag a block as bad. An alternate block, used in place of the bad block may be specified.

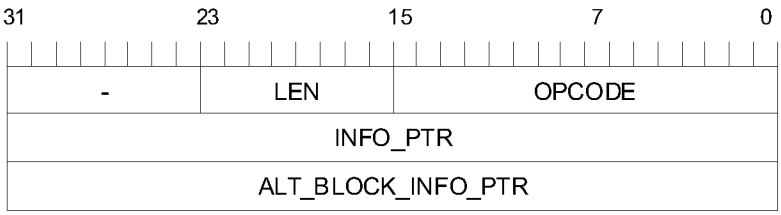


Figure 12.44 BAD_BLOCK Message Format

Field Name	Value	Description
OPCODE	BAD_BLOCK	Message opcode.
LEN	3	Length of this message in DWords.
INFO_PTR	-	Pointer to the bad blocks info structure.
ALT_BLOCK_INFO_PTR	-	Pointer to an alternate blocks info structure.

Table 12.45 BAD_BLOCK Message Fields

Data Structures

13.1 Overview

13.2 NVMHCI Data Structures

The data structures described in the following sections are defined in the Non-Volatile Memory Host Controller Interface (NVMHCI) 1.0 specification.

13.2.1 Command Header

NVMHCI command header structure.

```
typedef struct {
      unsigned short      commandLength       : 4,
                          reserved0           : 1,
                          interrupt           : 1,
                          write               : 1,
                          dataTransfer        : 2,
                          reserver1           : 7;
      unsigned short      PRDTL;
      unsigned int        PDRIOffset;
      unsigned int        CTBALo;
      unsigned int        CTBAHi;
      unsigned int        metadataOffset;
      unsigned int        reserved2;
      unsigned int        status;
      unsigned int        reserved3;
} NVM_COMMAND_HDR;
```

13.2.2 Physical Region Descriptor Table Entry

NVMHCI PRD table entry structure.

```
typedef struct {
      unsigned int        dataAddressLo;
      unsigned int        dataAddressHi;
      unsigned int        reserved0;
      unsigned short      byteCount;
      unsigned short      reserved1;
} NVM_PRD;
```

13.2.3 Physical Region Descriptor Index Table Entry

NVMHCI PRDI table entry structure.

```
typedef struct {
      unsigned short PRDEntry;
      unsigned short offset;
} NVM_PRDI;
```

13.3 List Head Structure

The list head structure is used to create doubly linked lists of structures. Typically this structure will be embedded with another structure.

```
typedef struct list_head {
      struct list_head    *prev;
      struct list_head    *next;
} LIST_HEAD;
```

13.4 Memory Network Tag Structure

This structure is used to keep track of memory network requests issued by a processor.

```
typedef struct {
      MEM_TAG_TYPE        type;
      void                *tag;
} MEM_TAG;

typedef enum {
      PAGE_STATE,
      COMMAND_STATE
} MEM_TAG_TYPE;
```

An array of this structure will be defined. The size of the array is defined by the maximum number of tags that the memory network instructions can issue. When a memory network request is issued, a tag is returned, in the range of 0-31 for example, this tag will be used to index the network state structure array and save a pointer (tag) and a pointer type (tagType) value. When a completion message is received for the memory network request it contains the tag issued when the memory network request was issued. This tag may then be used to lookup the pointer and pointer type which will be used to complete the request.

13.5 Page Cache Header

Each cached NVM page has a PAGE_CACHE_HDR structure associated with it. The *sectorVector* field is a bit vector specifying which sectors within the page cache contain valid data. When reading data from the page cache, if the result of the SECTOR_VECTOR from the read request AND'ed with the *sectorValid* field is equal to the SECTOR_VECTOR, all of the requested sectors are located within the page cache. Otherwise, some of the sectors are located in the page cache and some are located in the NVM device at the address specified by the *physPageAddr* field. To support the lookup of a cache entries PPE, the *logPageAddr* field contains the logical address of the associated page. The *writeQueue* field, if set, specifies that the cache entry is located within a Page Write Cache list. The actual page data begins at *sectorData[0]*. The size of this data is dependant on the size of the NVM device page and meta-data.

The *prev* and *next* fields are used to manipulate the page cache header within a doubly linked list.

```
typedef struct page_cache_hdr {
```

```
        struct page_cache_hdr      *prev;
        struct page_cache_hdr      *next;
        unsigned int               sectorVector; // Upto 32 sectors/page
        unsigned int               writeQueue    : 1;
        unsigned int               logPageAddr;
        unsigned int               physPageAddr;
        unsigned char              sectorData[0];
} PAGE_CACHE_HDR;
```

13.6 Partition Table

The Partition Table data structure, which is embedded within the Super Block, is used to describe the physical partitioning of the flash memory. The partition table contains the type, and the start and end blocks that constitute the partition. If the partition contains a file, the head and tail pointers for the file are also maintained with this data structure. The start and end addresses are the page number for NVRAM, and the upper 32 of 37 bits for BRAM and DRAM.

```
typedef struct {
    PARTITION_TYPE          partitionType;
    unsigned int            partitionStart;
    unsigned int            partitionEnd;
    unsigned int            head;
    unsigned int            tail;
} PARTITION_TABLE;
```

13.6.1 Partition Types

The partition types are defined by the **PARTITION_TYPE** enum.

```
typedef enum {
    NULL_PARTITION                    = 0x00000000,
    SUPER_BLOCK_PARTITION             = 0xdeadbeef,
    BAD_BLOCK_TABLE_PARTITION         = 0xbadbbadb,
    FIRMWARE_PARTITION                = 0xbad0a550,
    BLOCK_STATE_PARTITION             = 0xfeedface,
    LOGICAL_TO_PHYSICAL_MAP_PARTITION = 0xdeafbabe,
    USER_DATA_PARTITION               = 0xabfab000,
    SUBMIT_Q_PARTITION                = 0xfeeadded,
    DATA_CACHE_PARTITION              = 0xabadcafe
} PARTITION_TYPE;
```

13.7 Super Block

The Super Block is the key to the system. Located within the first 4 NVM blocks, the Super Block contains information critical to the operation of the system and for this reason special care should be taken when handling it. The Super Block begins with a Super Block Header the data from which is used to locate the latest Super Block This structure is copied to address 0x0 of the DRAM after initialization for general use. Since this structure is bigger than a single moveIn command can accomodate, it is split up into sub structures that may be moved in/out one at a time.

```
typedef struct {
      SUPER_BLOCK_HEADER      header;
      unsigned int        reserved[52]; // used to pad out to an even 256B
      PARTITION_TABLE     DRAMpartitionTable[16];  // takes up 256B
      PARTITION_TABLE     BRAMpartitionTable[16];  // takes up 256B
      PARTITION_TABLE     NVRAMpartitionTable[16]; // takes up 256B
} SUPER_BLOCK;
```

13.7.1 Super Block Header

The data within the Super Block Header is used to locate the most recent version of the Super Block.

```
typedef struct {
    PARTITION_TYPE          partitionType;
    unsigned int            versionNumber;
    unsigned int            myPageNum;      // set to current page #
                                            // for a sanity check
    BLOCK_STATE             currentState;  // superblock documents a
                                          // open or closed save state
    unsigned int           channels;      // number of flash channels
    unsigned int            targetsPerChannel;
    NVRAM_TYPE              flashType;
    unsigned int            pageSizeShift; // in terms of power of 2
    unsigned int            pagesPerBlock;
    unsigned int            metadataSize;   // in bytes
    unsigned int            totalSize;      // in pages
    unsigned int            advertizedSize; // in pages
} SUPER_BLOCK_HEADER;
```

13.8 File Header

When a file is written to a partition, a File Header structure is placed at the beginning of each page that constitute the file being written. The File Header contains information that will later be used to retrieve the file. This is made to be 64 bytes so that the DRAM accesses putting the file back together will always be aligned.

```
typedef struct {
    PARTITION_TYPE          partitionType;
    unsigned int            prevGoodPage;
    unsigned int            offset;          // Upper 32 bits of Addr
    unsigned int            length;          // In bytes
    unsigned int            reserved[12];    // Align to 64B
} FILE_HEADER;
```

13.9 Logical to Physical Map

These 4 bytes are used for each entry into the logical to physical map table.

```
typedef struct {
    unsigned int            physicalAddress : 28,
```

```
                                    location : 2,
                                    locked : 1,
                                    valid : 1;
} L2P_MAP;
```

13.10 Meta Data Map

This structure is used to organize the metadata and protect the software from changes when this structure changes in hardware. There are 256 bytes reserved for each meta data block, although the current NVRAM only has 224 bytes. The hardware is using some for ECC, and the firmware is reserving some for CRC, but quite a bit is left over for customer use.

The logical address and sequence number is there so that the logical to physical map table may be re-created in the event of a catastrophic failure.

```
typedef struct {
    unsigned int          crcData[8];
    unsigned int          generalUseData[34];
    unsigned int          eccData[14];
    unsigned int          reservedData[8];
} META_DATA_MAP;

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ OBSOLETE BELOW ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//    v     v     v     v     v     v     v     v     v     v     v     v     v
```

13.11 Log Page Header

Each page of log information is preceded by a Log Page Header structure. The *id* field in the header contains a static value 'LPH' which identifies this page as containing log data. A non-zero value in the *snapshot* field is used to denote that this log page contains snapshot data rather than incremental log data. The *snapshotEnd* field, if set, denotes the end of the snapshot data.

```
typedef struct {
      unsigned int        id                  : 24,  // 'LPH'
                          snapshot            : 1,
                          snapshotEnd         : 1;
} LOG_PAGE_HEADER;
```

13.12 Snapshot Page Header

The Snapshot Page Header consists of a Log Page Header followed the destination *offset*, in DRAM, and the *length* of the data contained within the log entry.

```
typedef struct {
      LOG_PAGE_HEADER     header;       // Log Page Header
      unsigned int        nextBlock;    // Physical addr of next block
      unsigned int        offsetLo;     // DRAM destination offset
      unsigned int        offsetHi;     // DRAM destination offset
      unsigned int        length;       // Length of data that follows
```

```
        unsigned char       snapshotData[0];// Snapshot data
} SNAPSHOT_PAGE_HEADER;
```

64-Bit Calculations

14.1 Overview

In certain circumstances it may be necessary to perform simple 64-bit calculations such as in the case of calculating the host address for a DMA transfer. This is currently possible on the 32-bit Tensilica core by specifying variables of the 'long long' C type. But, although possible, these types of calculations are quite slow, requiring multiple cycles for a simple addition of a 64 and 32 value.

To facilitate faster 64-bit calculations the customized Tensilica core used for Delaware includes a 64-bit register file that contains 4x64-bit general purpose registers. In order for the firmware to make use of this register file it must declare variables using the 'reg64' custom C type, see below.

```
reg64    this_is_a_64_bit_variable;
```

The format of this custom type is synonymous with 'unsigned long long'. Initializing a reg64 variable to a know value is possible via the following C code.

```
#define VALUE = 0x1234LL

reg64 t = VALUE;
```

Because this is a custom C type, it does require some special handling. For example, variables of this type may not be passed directly into C functions, a pointer to the variable must be passed instead, and a variable of this type may not be incremented or decremented using ++ or --, instead a macro generated by the Tensilica tools should be used, e.g.

```
this_is_a_64_bit_variable=AddR64Ar(this_is_a_64_bit_variable, 1);
```

The following sections describe the macros that are available for manipulating the 64-bit registers. All macros are defined in a Tensilica generated header file called <xtensa/tie/IDT_TIE_rev17.h>. Although the revision number may change for this header file, the base name should remain the same.

14.2 reg64 MIpToR64(unsigned int)

This macro takes bits 36:5 of a 37 bit address as the input parameter and returns the expanded 64-bit address. This macro performs the same operations as the following C code.

```
unsigned int a;

unsigned long long addr;

addr = (unsigned long long) a << 5;
```

14.3 unsigned int R64ToMIp(reg64)

This macro takes a 64 bit value as an input and converts it to a 32-bit value by shifting it to the right by 5. This macro performs the same operations as the following C code.

```
unsigned int a;

unsigned long long addr;

a = (unsigned int) (addr >> 5);
```

14.4 reg64 AddR64Ar(reg64, int)

Add a 32-bit signed value to a 64-bit value and return the result.

14.5 reg64 AddR64UAr(reg64, unsigned int)

Add a 32-bit unsigned value to a 64-bit value and return the result.

14.6 reg64 AddR64R64(reg64, reg64)

Add a 64-bit value to a 64-bit value and return the result.

14.7 reg64 SubR64R64(reg64, reg64)

Subtract a 64-bit value from a 64-bit value and return the result.

eNVMHCI

15.1 Overview

How is eNVMHCI supported?

Functionality is split between the Host Interface (Submission Queue and Status Queue managers) and the PCIe Manager which handles the generation of MSI-X TLP's.

15.2 Vendor Specific Commands

The firmware should support the following vendor specific commands.

15.2.1 Physical Read

The physical read command is a way in which the host can issue a read to any physical page of NV memory. This read will operate the same as a normal read except for the fact that a translation from logical to physical page number will not be performed.

This will allow the host computer to interrogate the control partition in NV memory.

References

[1] Non-Volatile Memory Host Controller Interface (NVMHCI) 1.0, April 14, 2008, http://www.intel.com/standards/nvmhci/index.htm.

Revision History

- *Draft*
  -



Delaware
Enterprise Flash Controller
FPGA Prototype

May 22, 2010

180 Mount Airy Road, Basking Ridge, New Jersey 07920
Telephone: (908) 766-4950
Printed in U.S.A.

GENERAL DISCLAIMER

Integrated Device Technology, Inc. reserves the right to make changes to its products or specifications at any time, without notice, in order to improve design or performance and to supply the best possible product. IDT does not assume any responsibility for use of any circuitry described other than the circuitry embodied in an IDT product. The Company makes no representations that circuitry described herein is free from patent infringement or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent, patent rights or other rights, of Integrated Device Technology, Inc.

CODE DISCLAIMER

Code examples provided by IDT are for illustrative purposes only and should not be relied upon for developing applications. Any use of the code examples below is completely at your own risk. IDT MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE NONINFRINGEMENT, QUALITY, SAFETY OR SUITABILITY OF THE CODE, EITHER EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. FURTHER, IDT MAKES NO REPRESENTATIONS OR WARRANTIES AS TO THE TRUTH, ACCURACY OR COMPLETENESS OF ANY STATEMENTS, INFORMATION OR MATERIALS CONCERNING CODE EXAMPLES CONTAINED IN ANY IDT PUBLICATION OR PUBLIC DISCLOSURE OR THAT IS CONTAINED ON ANY IDT INTERNET SITE. IN NO EVENT WILL IDT BE LIABLE FOR ANY DIRECT, CONSEQUENTIAL, INCIDENTAL, INDIRECT, PUNITIVE OR SPECIAL DAMAGES,

IDT, the IDT logo, and Integrated Device Technology are trademarks or registered trademarks of Integrated Device Technology, Inc.

About This Manual

Signal Nomenclature

To avoid confusion when dealing with a mixture of "active-low" and "active-high" signals, the terms assertion and negation are used. The term assert or assertion is used to indicate that a signal is active or true, independent of whether that level is represented by a high or low voltage. The term negate or negation is used to indicate that a signal is inactive or false.

To define the active polarity of a signal, a suffix will be used. Signals ending with an 'N' should be interpreted as being active, or asserted, when at a logic zero (low) level. All other signals (including clocks, buses and select lines) will be interpreted as being active, or asserted when at a logic one (high) level.

To define buses, the most significant bit (MSB) will be on the left and least significant bit (LSB) will be on the right. No leading zeros will be included.

Throughout this manual, when describing signal transitions, the following terminology is used. Rising edge indicates a low-to-high (0 to 1) transition. Falling edge indicates a high-to-low (1 to 0) transition. These terms are illustrated in Figure 1.

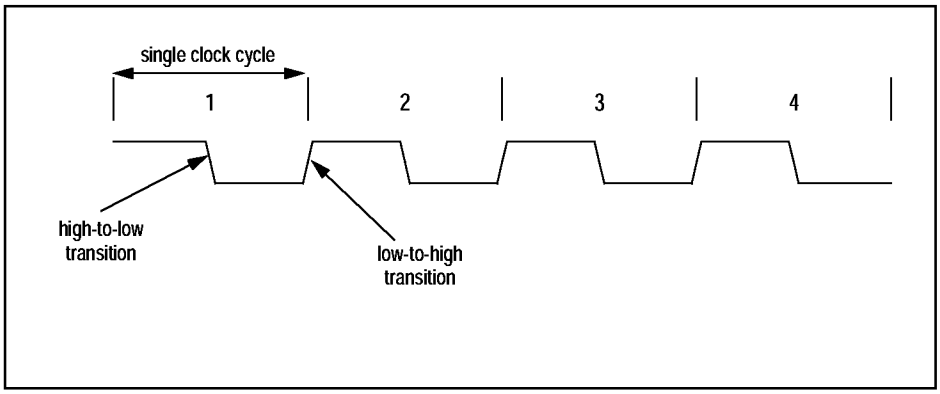


Figure 1 Signal Transitions

Numeric Representations

To represent numerical values, either decimal, binary, or hexadecimal formats will be used. The binary format is as follows: 0bDDD, where "D" represents either 0 or 1; the hexadecimal format is as follows: 0xDD, where "D" represents the hexadecimal digit(s); otherwise, it is decimal.

The compressed notation ABC[x|y|z]D refers to ABCxD, ABCyD, and ABCzD.

The compressed notation ABC[y:x]D refers to ABCxD, ABC(x+1)D, ABC(x+2)D,... ABCyD if x < y and ABCyD, ABC(y+1)D, ABC(y+2)D,... ABCxD if y <x.

Data Units

The following data unit terminology is used in this document.

Term	Words	Bytes	Bits
Byte	1/2	1	8
Word	1	2	16
Doubleword (Dword)	2	4	32
Quadword (Qword)	4	8	64

Table 1 Data Unit Terminology

In quadwords, bit 63 is always the most significant bit and bit 0 is the least significant bit. In doublewords, bit 31 is always the most significant bit and bit 0 is the least significant bit. In words, bit 15 is always the most significant bit and bit 0 is the least significant bit. In bytes, bit 7 is always the most significant bit and bit 0 is the least significant bit.

The ordering of bytes within words is referred to as either "big endian" or "little endian." Big endian systems label byte zero as the most significant (leftmost) byte of a word. Little endian systems label byte zero as the least significant (rightmost) byte of a word. See Figure 2.

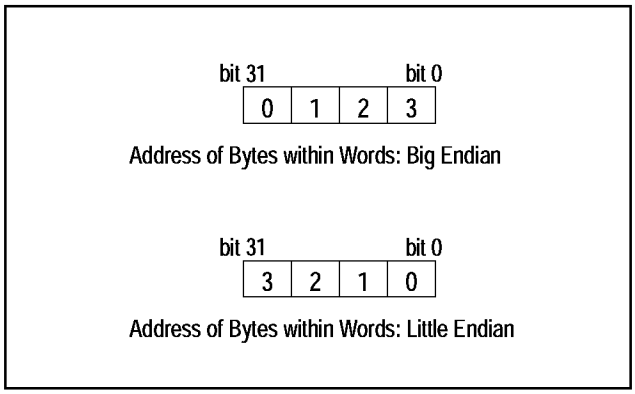


Figure 2 Example of Byte Ordering for "Big Endian" or "Little Endian" System Definition

Register Terminology

Software in the context of this register terminology refers to modifications made by any programmable agent (e.g., PCIe root configuration writes).

Type	Abbreviation	Description
Hardware Initialized	HWINIT	Register bits are initialized by hardware mechanisms such as pin strapping or serial EEPROM.
Read Only and Clear	RC	Software can read the register/bits with this attribute. Reading the value will automatically cause the register/bit to be reset to zero. Writing to an RC location has no effect.
Read Clear and Write	RCW	Software can read the register/bits with this attribute. Reading the value will automatically cause the register/bits to be reset to zero. Writes cause the register/bits to be modified.

Table 2 Register Terminology (Sheet 1 of 2)

Type	Abbreviation	Description
Reserved	Reserved	The value read from a reserved register/bit is undefined. Thus, software must deal correctly with fields that are reserved. On reads, software must use appropriate masks to extract the defined bits and not rely on reserved bits being any particular value. On writes, software must ensure that the values of reserved bit positions are preserved. That is, the values of reserved bit positions must first be read, merged with the new values for other bit positions and then written back.
Read Only	RO	Software can only read registers/bits with this attribute. Contents are hardwired to a constant value or are status bits that may be set and cleared by hardware. Writing to a RO location has no effect.
Read and Write	RW	Software can both read and write bits with this attribute.
Read and Write Clear	RW1C	Software can read and write to registers/bits with this attribute. However, writing a value of zero to a bit with this attribute has no effect. A RW1C bit can only be set to a value of 1 by a hardware event. To clear a RW1C bit (i.e., change its value to zero) a value of one must be written to the location. An RW1C bit is never cleared by hardware.
Read and Write when Unlocked	RWL	Software can read the register/bits with this attribute. Writing to register/bits with this attribute will only cause the value to be modified if the register is unlocked.

Table 2 Register Terminology (Sheet 2 of 2)

Table of Contents

About This Manual

Signal Nomenclature i
Numeric Representations i
Data Units i
Register Terminology ii

Chapter 1 Overview

1.1 Introduction 1-1
1.2 Features 1-1

Chapter 2 Architectural Overview

2.1 Introduction 2-1
2.2 Functional Operation 2-5
2.2.1 Host Read Overview 2-7
2.2.2 Host Write Overview 2-8

Chapter 3 System Configuration

3.1 FPGA Board 3-1
3.2 FPGA Partitioning 3-2
3.3 Message Network 3-5
3.4 Buffer RAM Data Network 3-6
3.5 DRAM Data Network 3-7

Chapter 4 Memory and Hardware Data Structures

4.1 Introduction 4-1
4.2 Memory 4-1
4.2.1 Controller Memory 4-1
4.2.2 Node Address Space 4-1
4.2.3 CAR Address Space 4-4
4.2.4 Global Memory Space 4-5
4.3 Controller Data Structures 4-5
4.3.1 Memory Line 4-5
4.3.2 Memory Line Pointer 4-6
4.3.3 Page Frame 4-6

Chapter 5 Memory Network

5.1 Introduction 5-1
5.2 Ordering Rules 5-3
5.3 Memory Network Packets 5-4
5.4 Transactions 5-5

5.4.1 Completion (Cpl) ... 5-6
5.4.2 Read (Rd) ... 5-7
5.4.3 Read and Lock (RdLock) ... 5-8
5.4.4 Read XOR and Write (RdXorWr) ... 5-8
5.4.5 Write (Wr) ... 5-10
5.4.6 Write Line (WrLine) ... 5-10
5.4.7 Transfer (Xfer) ... 5-11

Chapter 6 Message Network

6.1 Introduction ... 6-1
6.2 Message Network Packets and Messages ... 6-3
6.3 Message Queues ... 6-4

Chapter 7 Buffer RAM

7.1 Introduction ... 7-1

Chapter 8 Host Interface

8.1 Introduction ... 8-1
8.2 Message Network Interface ... 8-1
8.3 Ingress DMA ... 8-3
8.4 Egress DMA ... 8-3
8.5 PCIe Processor Interface ... 8-3
8.6 NVMHCI Submission Queue Manager ... 8-3
8.7 NVMHCI Completion Queue Manager ... 8-7
8.8 PCIe Registers ... 8-12
8.8.1 PCIe Configuration Space ... 8-12
8.8.2 PCIe BAR 0 ... 8-12
8.9 CAR Registers ... 8-12
8.9.1 General Host Interface Registers ... 8-16
8.9.2 NVMHCI Submission Queue Manager ... 8-18
8.9.3 NVMHCI Completion Queue Manager ... 8-19
8.9.4 PCIe Processor Interface ... 8-20
8.9.5 NVMHCI Submission Queue Manager ... 8-20
8.9.6 Message Network Interface Registers ... 8-20

Chapter 9 PCI Express Interface

9.1 Introduction ... 9-1
9.2 CAR Registers ... 9-1
9.2.1 PCIe Interface Registers ... 9-2

Chapter 10 Flash Controller

10.1 Introduction ... 10-1
10.2 Flash Timing ... 10-1
10.3 Flash Devices and Addressing ... 10-1
10.4 Page Frame Addressing ... 10-2
10.5 Flash Controller Request Messages ... 10-3
10.5.1 FlashRead ... 10-3
10.5.2 FlashWrite ... 10-4
10.5.3 FlashErase ... 10-5
10.6 Flash Calibration ... 10-6
10.7 Message Network Interface ... 10-7
10.8 CAR Registers ... 10-8
10.8.1 Flash Configuration Registers ... 10-9
10.8.2 Page Frame Region Base Registers ... 10-16
10.8.3 Message Network Interface Registers ... 10-17

Chapter 11 Error Correction Unit

11.1 Introduction ... 11-1

Chapter 12 Processor Complex

12.1 Introduction ... 12-1
12.2 Processor Configuration ... 12-1
12.3 Interrupts ... 12-4
12.4 GPIO Ports ... 12-4
12.5 Processor Address Map ... 12-6
12.6 64-bit Register File and Instructions ... 12-7
12.6.1 64-bit Instructions ... 12-9
12.6.2 Reg64 Type Prototypes ... 12-13
12.7 Message Network Interface ... 12-14
12.7.1 Outbound Message Queues ... 12-15
12.7.2 Inbound Message Network Interface ... 12-18
12.7.3 Message Queue Helper Instructions ... 12-20
12.8 Memory Network Interface ... 12-20
12.8.1 Memory Network Instructions ... 12-23
12.9 CAR Registers ... 12-27
12.9.1 Debug Registers ... 12-27

Chapter 13 DRAM Interface

13.1 Introduction ... 13-1

Chapter 14 Functional Unit Messages

14.1 Introduction 14-1
14.2 CmdReady 14-1
14.3 DmaCpl 14-1
14.4 EgressDma 14-2
14.5 EgressTlp 14-3
14.6 FlashErase 14-3
14.7 FlashOpCpl 14-4
14.8 FlashRead 14-5
14.9 FlashWrite 14-6
14.10 IngressDma 14-7
14.11 IngressTlp 14-8
14.12 CplQueueUpdate 14-8
14.13 CplQueueUpdateCpl 14-9

Revision History

List of Tables

Table 2.1 Functional Unit Messages 2-6
Table 2.2 Functional Unit Issued and Received Messages 2-6
Table 3.1 FPGA Q 3-2
Table 3.2 FPGA A 3-2
Table 3.3 FPGA B 3-2
Table 3.4 FPGA C 3-3
Table 3.5 FPGA F 3-3
Table 3.6 FPGA E 3-4
Table 4.1 Message Queue Node Addresses 4-3
Table 5.1 Request Packet Header Field Bit Allocation 5-4
Table 5.2 Request Packet Type Field Encoding 5-4
Table 5.3 Completion Packet Header Field Bit Allocation 5-5
Table 5.4 Memory Network Requests 5-5
Table 5.5 Completion (Cpl) Header Fields 5-6
Table 5.6 Read (Rd) Header Fields 5-7
Table 5.7 Read and Lock (RdLock) Header Fields 5-8
Table 5.8 Read XOR and Write (RdXorWr) Header Fields 5-8
Table 5.9 Read Add and Write (RdAddWr) Header Fields 5-9
Table 5.10 Write (Wr) Header Fields 5-10
Table 5.11 Write Line (WrLine) Header Fields 5-10
Table 5.12 Transfer (Xfer) Header Fields 5-11
Table 6.1 Sending a Three DWord Message 6-6
Table 6.2 Freeing a Message Buffer 6-7
Table 8.1 PCIe Non Posted Request Tag Allocation 8-1
Table 8.2 Host Interface Outbound Message Queues 8-1
Table 8.3 Host Interface Inbound Message Queues 8-2
Table 8.4 Host Interface CSR Address Map 8-12
Table 9.1 PCIe Interface CSR Address Map 9-1
Table 10.1 Flash Timing Register Configuration (125 MHz or less) 10-1
Table 10.2 Flash Controller Outbound Message Queues 10-7
Table 10.3 Flash Controller Inbound Message Queues 10-8
Table 10.4 Flash Controller CSR Address Map 10-8
Table 12.1 Delaware Tensilica Processor Core Configuration 12-1
Table 12.2 Processor Interrupts 12-4
Table 12.3 GPIO Input Port 12-4
Table 12.4 GPIO Output Port 12-5
Table 12.5 64-bit Register File Instructions 12-8
Table 12.6 64-bit Register File Prototypes 12-8
Table 12.7 Outbound Message Queue State 12-15
Table 12.8 Outbound Message Queue Instruction Summary 12-15
Table 12.9 Message Network Packet Generated by a SendMsg Instruction 12-16
Table 12.10 Message Network Packet Generated by a PostMsg Instruction 12-16
Table 12.11 Outbound Message Queue State 12-18
Table 12.12 Outbound Message Queue Instruction Summary 12-18
Table 12.13 Message Network Packet Generated by a PostMsg Instruction 12-19
Table 12.14 Memory Network Interface Instruction Summary 12-20
Table 12.15 Memory Network Interface Message Buffer Pointer Field Description 12-22
Table 12.16 Flash Controller CSR Address Map 12-27
Table 14.1 CmdReady Message Format 14-1

Table 14.2 DmaCpl Message Format 14-1
Table 14.3 EgressDMA Message Format 14-2
Table 14.4 EgressTlp Message Format 14-3
Table 14.5 FlashErase Message Format 14-3
Table 14.6 FlashOpCpl Message Format 14-4
Table 14.7 FlashRead Message Format 14-5
Table 14.8 FlashWrite Message Format 14-6
Table 14.9 IngressDMA Message Format 14-7
Table 14.10 IngressTlp Message Format 14-8
Table 14.11 CplQueueUpdate Message Format 14-8
Table 14.12 CplQueueUpdateCpl Message Format 14-9

List of Figures

Figure 2.1 High Level Architecture 2-2
Figure 2.2 Architectural Block Diagram 2-4
Figure 3.1 Delaware FPGA Board 3-1
Figure 3.2 Message Network 3-5
Figure 3.3 Buffer RAM Data Network 3-6
Figure 3.4 DRAM Data Network 3-7
Figure 4.1 Controller Memory Map 4-1
Figure 4.2 Node Address Space 4-2
Figure 4.3 CAR Address Space 4-4
Figure 4.4 Global Memory Space 4-5
Figure 4.5 Flash Page Frame 4-6
Figure 4.6 Page Frame Layout in Controller Memory 4-7
Figure 5.1 Memory Network 5-1
Figure 5.2 Memory Network Request and Completion Routing 5-2
Figure 6.1 Message Network Node 6-1
Figure 6.2 Simplified Message Network Node Router Architecture 6-2
Figure 6.3 Node Router Controller Routing Algorithm 6-2
Figure 6.4 Example Message Network Packet Routing 6-3
Figure 6.5 Message Network Packet Format 6-3
Figure 6.6 Conceptual Model of a Message Queue 6-4
Figure 6.7 Message Buffer Pointer Format 6-5
Figure 6.8 Conceptual Model of a Message Queue with Two Source Nodes 6-8
Figure 8.1 Host Interface Inbound and Outbound Message Queues 8-1
Figure 8.2 Host Interface Node Address Map 8-3
Figure 8.3 Submission Queue 8-3
Figure 8.4 NVMHCI Submission Queue Manager 8-5
Figure 8.5 Completion Queue 8-7
Figure 8.6 Completion Format 8-8
Figure 8.7 NVMHCI Completion Queue Manager 8-9
Figure 10.1 Flash Address Format 10-1
Figure 10.2 Example Page Frame Layout in Buffer RAM 10-2
Figure 10.3 Flash Controller Inbound and Outbound Message Queues 10-7
Figure 10.4 Host Interface Node Address Map 10-8
Figure 11.1 Page Frame Layout In Memory 11-1
Figure 11.2 Page Frame Layout in a Flash Page 11-1
Figure 12.1 Processor Address Map 12-6
Figure 12.2 Local Node Processor Address Map 12-7
Figure 12.3 Memory Network Interface Message Buffer Pointer Format 12-22



Overview

1.1 Introduction

1.2 Features

Architectural Overview

2.1 Introduction

- The Delaware FPGA prototype is a programmable PCI Express (PCIe) NAND Flash controller. The hardware architecture provides a high performance, symmetric, and simple to program platform for controlling NAND Flash and interfacing to a host via PCIe. Unlike other high performance NAND Flash controllers that hardwire algorithms in order to achieve high performance, all Delaware controller operations are under firmware control enabling OEMs to differentiate product offerings by deploying proprietary Flash management and controller algorithms.
- Figure 2.1 shows a high level block diagram of Delaware. The architecture consists of six functional units.
  - Host Interface
    - x4 PCIe Gen2 interface
    - Ingress and Egress DMA
    - Queueing and dispatching of NVMHCI commands
    - Reception and transmission of PCIe TLPs associated with memory mapped I/O and configuration transactions
  - Flash Controller
    - Interface to 16 ONFI 2.1 Flash channels
    - Computes and inserts ECC parity
    - Checks ECC syndrome
    - Orchestrates ECC correction using ECC correction unit
  - ECC Correction Unit
    - Performs ECC correction
  - Buffer RAM
    - High speed on-chip SRAM for temporary storage of Flash data and controller data structures
  - DRAM Interface
    - An interface to high speed external DDR SDRAM for temporary storage of Flash data and controller data structures
  - Processor Complex
    - A pool of programmable processors (Tensilica cores) used to execute Flash management and controller algorithms.
      - Except for the control processor, all processors cores are identical. The architecture enables the processors to be used as a pool or processors, a pipeline of processors, or as a combination of the two approaches.
      - The control processor has additional capabilities and is responsible for booting the system and providing overall control and management.

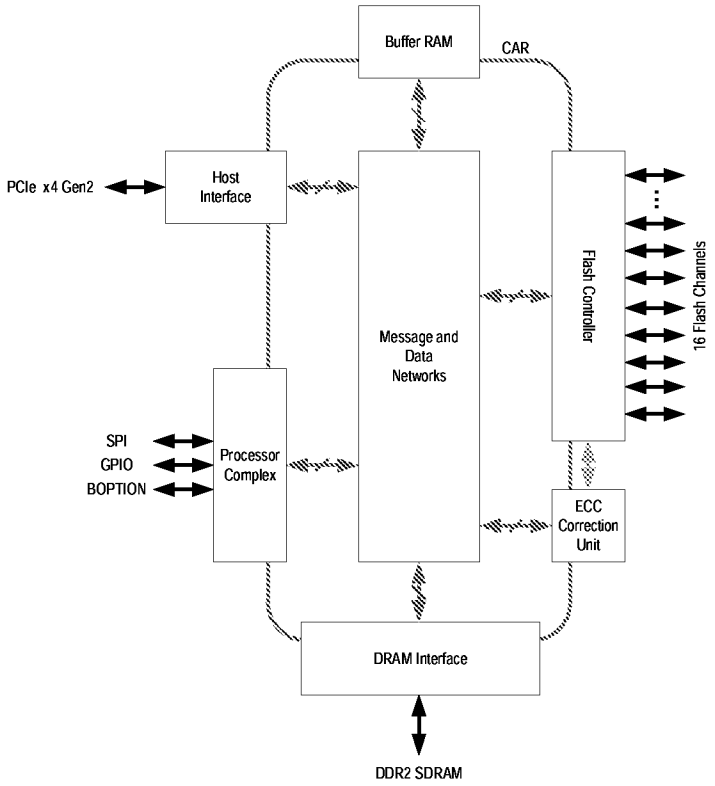


Figure 2.1 High Level Architecture

- Functional units communicate using one of five interconnects.
  - Message Network (MSGN)
    - The MSGN provides routing of short messages between functional units and processors. These messages are used to orchestrate operations. Bulk data transfers use one of the data networks.
  - Buffer RAM Data Network (BRDN)
    - The BRDN provides bulk data transfer between functional units and buffer RAM.
  - DRAM Data Network (DRDN)
    - The DRDN provides bulk data transfer between functional units and the DRAM interface.
  - Common Access Ring (CAR)
    - The CAR provides access to functional unit control and status registers. This is a relatively low performance interface intended for initialization, debug, statistics collection and error recovery.
  - Global Interrupts
    - Interrupts provide a mechanism for the signalling of events to processors. There are three types of global interrupts.
      - Attention Interrupts
        - Each function unit and processor has the ability to generate an attention interrupt. These interrupts are used to indicate that an event has occurred within the functional unit that requires special processing. All attention interrupts are routed to the control processor.

- Processor Attention Interrupts
  - A processor attention interrupt is routed from the control processor to each of the other processors. This interrupt is used by the control processor to signal to a specific processor that special processing is required by that processor.
- Power Down Interrupt
  - A power fail interrupt is generated by the assertion of the Power Down (PWRDN) signal. The power fail interrupt is routed to all of the processors.

- Each functional unit and processor has a node ID. The node ID uniquely identifies a functional unit or processor and is used for routing in the message and data networks.
- Buffer RAM and DRAM controlled by the DRAM interface together are referred to as controller memory.
  - Controller memory is used to hold controller data structures and Flash data.
  - Except for a performance difference, controller memory is completely symmetric. Anything that may be stored or any operation that may be performed in one memory (e.g., buffer RAM) may be stored and performed in the other memory (i.e., DRAM).
  - Controller memory is accessed by functional units using a flat 37-bit address. This address is called a controller memory address. The most significant bit (i.e., bit 37) of the controller memory address is used to select buffer RAM or DRAM. The remaining bits form a byte address in the selected memory.
    - Although the FPGA prototype implements supports far less memory, the controller architecture supports a maximum of 64GB of DRAM.
  - A 64B quantity is referred to as a memory line. Controller memory is optimized for memory line accesses. Although memory is byte addressable, highest performance is achieved with memory line accesses.
- Host memory refers to memory addressable through the PCIe interface. While host memory is typically physically located in the root of the PCIe hierarchy, there is nothing to prevent this memory from being located within any other PCIe function.
- Figure 2.2 shows the Delaware architecture in more detail.

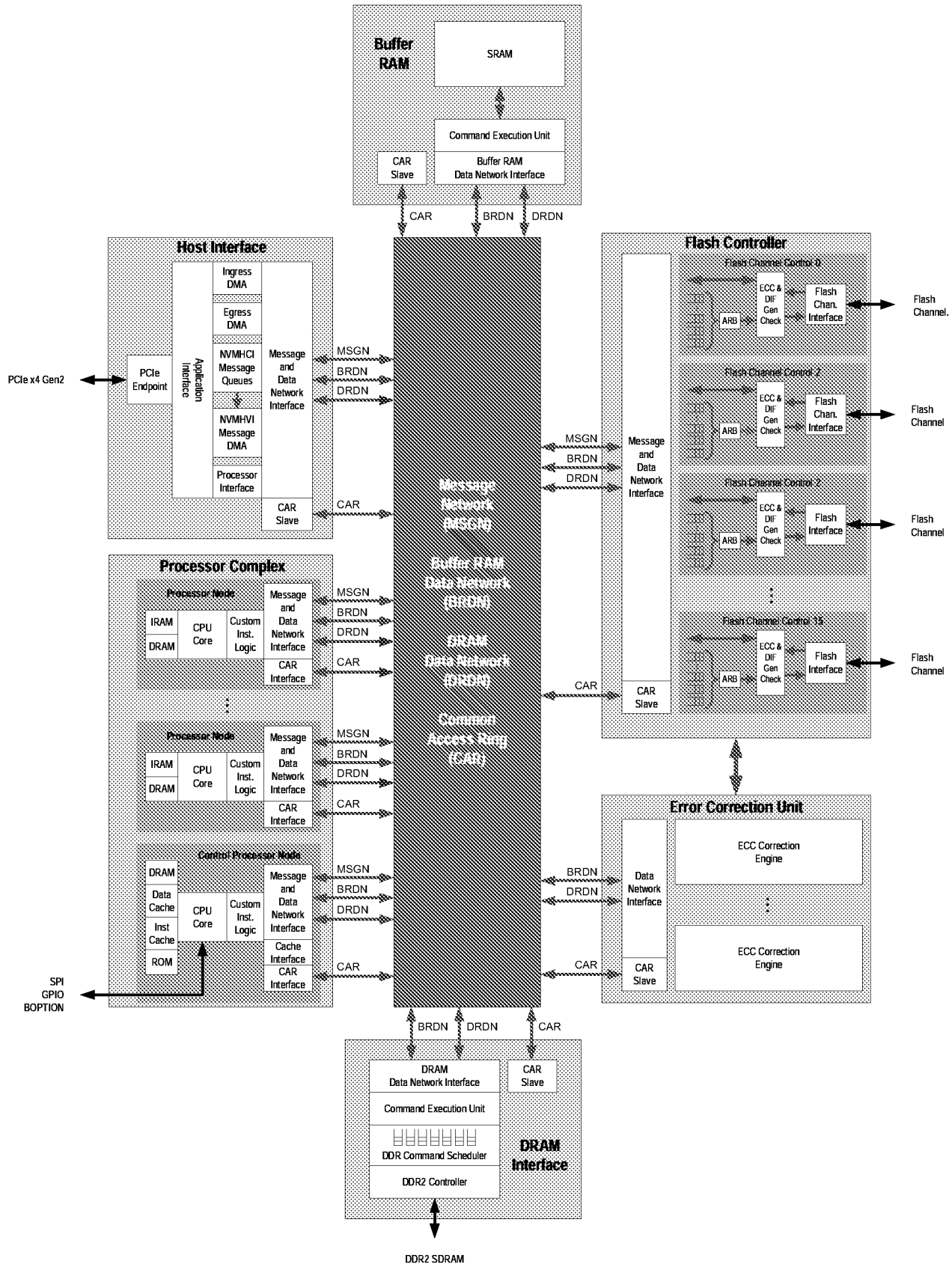


Figure 2.2 Architectural Block Diagram

2.2 Functional Operation

- The Delaware architecture consists of a two components: generic hardwired functional units that primarily operate on data and programmable processors that orchestrate operations.
  - Hardwired functions
    - Host interface
    - DMA operations between controller memory and host memory
    - NVMHCI command queuing, arbitration, and transfer
    - NAND Flash read, write and erase operations
    - ECC parity generation, syndrome checking and error correction
  - Functions under firmware control
    - NVMHCI command interpretation
    - Flash read, write, and erase operation initiation
    - DMA initiation
    - Wear leveling algorithm
    - Garbage collection algorithm
    - Flash management algorithms (e.g., bad block determination/management)
    - Flash Translation Layer (FTL) processing and data structures
    - Abrupt shutdown / power failure handing
    - Flash meta data usage
- The data interface between hardwired functional units and firmware running on processors is controller memory.
  - Controller memory is used to hold Flash data structures used by processors and Flash data.
  - Hardwired functional units never directly access Flash data structures in controller memory.
  - While processors may access Flash data in controller memory (e.g., to access data structures required for Flash management in the meta data portion of a Flash page frame), these accesses should be minimized since it is not the intent for processors to perform data path processing. Using processors for data path processing may have a negative impact on controller performance.
- The control interface between hardwired functional units and firmware running on processors is through messages. These messages as transferred between processors and functional units via the message network.
  - Table 2.1 lists functional unit messages. These are messages that are transferred between a hardwired functional unit and a processor. The definition, format and behavior of these messages are fixed in hardware and defined in Chapter 14.
  - In addition to functional unit messages (i.e., messages exchanged between hardwired functional units and processors) processors may exchange messages among themselves. The definition, format and behavior of these messages if defined by the firmware running on the controller and are outside the scope of this specification.

Message Class	Message	Description
Ingress and Egress DMA	EgressDma	DMA data from controller memory to host memory
	IngressDma	DMA data from host memory to controller memory
	DmaCpl	Completion associated with a previously issued IngressDMA or EgressDMA request
TLP Transmission	EgressTlp	Emit a TLP on the PCIe interface
TLP Reception	IngressTlp	A TLP was receive don the PCIe interface that requires software processing
Flash Operations	FlashErase	Erase a Flash block
	FlashRead	Read data from Flash and transfer to controller memory
	FlashWrite	Write data from controller memory to Flash
	FlashOpCpl	Completion associated with a previously issued Flash operation request
NVMHCI Command	CmdReady	NVMHCI command has been transferred from host memory to controller memory and is ready for processing
	CplQueueUpdate	Insert a completion queue entry contained in the message in the next free completion queue entry location in PCIe memory
	CplQueueUpdateCpl	Completion associated with a previously issued CplQueueUpdate request

Table 2.1 Functional Unit Messages

- Functional unit messages may be categorized as either requests or completions.
  - A request message may be issued by a hardwired functional unit or processor and indicates that a certain operation should be performed.
  - The message network contains a flow control mechanism that is used to back pressure requests. However, in some cases it is necessary for the agent issuing the request to know that the processing of a request has completed and whether any errors occurred during this processing. This information is contained within completion messages.
- Table 2.2 shows the messages exchanged between functional units. The host interface and Flash controller are hardwired functional units while the processor complex contains processors on which firmware is run.

Issued Messages \ Received Messages	Host Interface	Processor Complex	Flash Controller
Host Interface		Requests CmdReady IngressTlp Completions DmaCpl CplQueueUpdateCpl	
Processor Complex	Requests EgressDma IngressDma EgressTlp CplQueueUpdate		Requests FlashRead FlashWrite FlashErase
Flash Controller		Completions FlashOpCpl	

Table 2.2 Functional Unit Issued and Received Messages

2.2.1 Host Read Overview

- This section provides a high level overview of a host read operation. The intent of this overview is not to describe an implementation of a host read since the specific operations performed are under firmware control, but to illustrate how the controller operates.
  - The description assumes a single processor performs all of the actions associated with processing of a host read command. In order to achieve high performance, actual firmware may employ multiple processors.
- The host launches an ENVMHCI read command by creating a command in a submission queue and writing an updated tail pointer value to the corresponding submission queue tail doorbell register. This signals to the controller that a new command is ready to be executed.
- The host interface arbitrates among submission queues with ready commands. The controller selects a submission queue and transfers a command from the head of the queue in PCIe memory to the controller. It then sends a CmdReady message containing the command contents to a free processor.
- A processor receives a CmdReady message and initiates operations required to service the read.
  - The processor allocates a Flash page frame in controller memory.
  - The processor issues a FlashRead request message to the Flash controller. The message contains the physical address of the Flash page to be read and the address of the page frame where the data should be transferred.
  - The Flash controller performs the read operation, orchestrates and required ECC corrections, and places the Flash data in the page frame. When this operation completes, the Flash controller issues a FlashOp completion message back to the processor.
  - The processor issues an EgressDMA request message to the host interface to transfer the data in the Flash page frame to host memory.
  - When the egress DMA operation completes, the host interface issues a DmaCpl completion message back to the processor. The processor then deallocates the Flash page frame.
  - The above operations repeat until all of the requested data in the ENVMHCI command has been transferred.
- The processor signals that processing associated with an ENVMHCI command has completed by enqueueing a completion entry in the completion queue associated with the submission queue on which the ENVMHCI command arrived.
  - The processor sends a CplQueueUpdate message to the host interface containing the completion entry contents to be enqueued and the completion queue ID.
  - The host interface checks the status of the completion queue.
    - If the completion queue is not full, then the host interface writes the completion entry in the next free location in the completion queue and sends a CplQueueUpdateCpl message back to the processor indicating that the operation completed.
    - If the completion queue is full, then the host interface send a CplQueueUpdateCpl message back to the processor indicating that the operation must be retried. When a retry is signalled, the processor waits and reissues the CplQueueUpdate message. The process repeats until the completion entry is successfully enqueued in the completion queue.
  - The processor issues an interrupt to the host indicating that the completion queue has been updated.

2.2.2 Host Write Overview

- This section provides a high level overview of a host write operation. The intent of this overview is not to describe an implementation of a host write since the specific operations performed are under firmware control, but to illustrate how the controller operates.
  - The description assumes a single processor performs all of the actions associated with processing of a host write command. In order to achieve high performance, actual firmware may employ multiple processors.
  - The description assumes a extremely simple block reclamation algorithm. Actual firmware will likely employ a much more sophisticated algorithm.
- The host launches an ENVMHCI write command by creating a command in a submission queue and writing an updated tail pointer value to the corresponding submission queue tail doorbell register. This signals to the controller that a new command is ready to be executed.
- The host interface arbitrates among submission queues with ready commands. The controller selects a submission queue and transfers a command from the head of the queue in PCIe memory to the controller. It then sends a CmdReady message containing the command contents to a free processor.
- A processor receives a CmdReady message and initiates operations required to service the write.
  - The processor allocates a Flash page frame in controller memory.
  - The processor issues a IngressDMA request message to the host interface to transfer the data in host memory to the Flash page frame.
  - When the ingress DMA operation completes, the host interface issues a DmaCpl completion message back to the processor.
  - The processor allocates a Flash page to hold the contents of the Flash page frame. If necessary, the processor performs block reclamation.
    - The processor selects a Flash block for reclamation. If the Flash block has active pages, then the processor allocates Flash page frames in controller memory to hold the active page data, and transfers the contents of the Flash pages to the Flash page frames. It does this by issuing FlashRead request messages to the Flash controller and waiting for corresponding FlashOpCpl completion messages.
    - Once all of the active pages have been saved in controller memory, the processor erases the Flash block. It does this by issuing a FlashErase request message to the Flash interface. When the Flash erase completes, the flash interface issues a FlashOpCpl completion message to the processor.
    - The processor when writes the active pages saved in controller memory back to the erased Flash block. It does this by issuing a FlashWrite request message to the Flash controller with the physical address of the Flash page to be written and the address of the page frame containing active page data. When the Flash write completes, the flash interface issues a FlashOpCpl message back to the processor.
  - The processor issues a FlashWrite request message to the Flash controller. The message contains the physical address of the Flash page to be written and the address of the page frame that holds the data.
  - The Flash controller computes ECC parity bits, inserts them in the meta data and performs the write operation. When this operation completes, the Flash controller issues a FlashOp completion message back to the processor.
  - The above operations repeat until all of the requested data in the NVMHCI command has been transferred.

- The processor signals that processing associated with an ENVMHCI command has completed by enqueueing a completion entry in the completion queue associated with the submission queue on which the ENVMHCI command arrived.
  - The processor sends a CplQueueUpdate message to the host interface containing the completion entry contents to be enqueued and the completion queue ID.
  - The host interface checks the status of the completion queue.
    - If the completion queue is not full, then the host interface writes the completion entry in the next free location in the completion queue and sends a CplQueueUpdateCpl message back to the processor indicating that the operation completed.
    - If the completion queue is full, then the host interface send a CplQueueUpdateCpl message back to the processor indicating that the operation must be retried. When a retry is signalled, the processor waits and reissues the CplQueueUpdate message. The process repeats until the completion entry is successfully enqueued in the completion queue.
  - The processor issues an interrupt to the host indicating that the completion queue has been updated.

System Configuration

3.1 FPGA Board

- The Delaware FPGA prototype utilizes a DN9000K10PCIe-8T FPGA board. A block diagram of the board is shown in Figure 3.1.
  - The board contains six FPGAs. Five Xilinx LX330 FPGAs and one LX70T.
  - FPGA A contains an SODIMM connection that is used to interface to the Tensilica processor probe via a processor JTAG connection.
  - FPGA B contains an so DIMM connection that interfaces to an external DDR3 SODIMM.
  - FPGA C contains two SODIMM connections. These connections are used to interface to custom IDT NAND Flash modules. Each SODIMM supports four ONFI 2.1 channels with eight targets each.

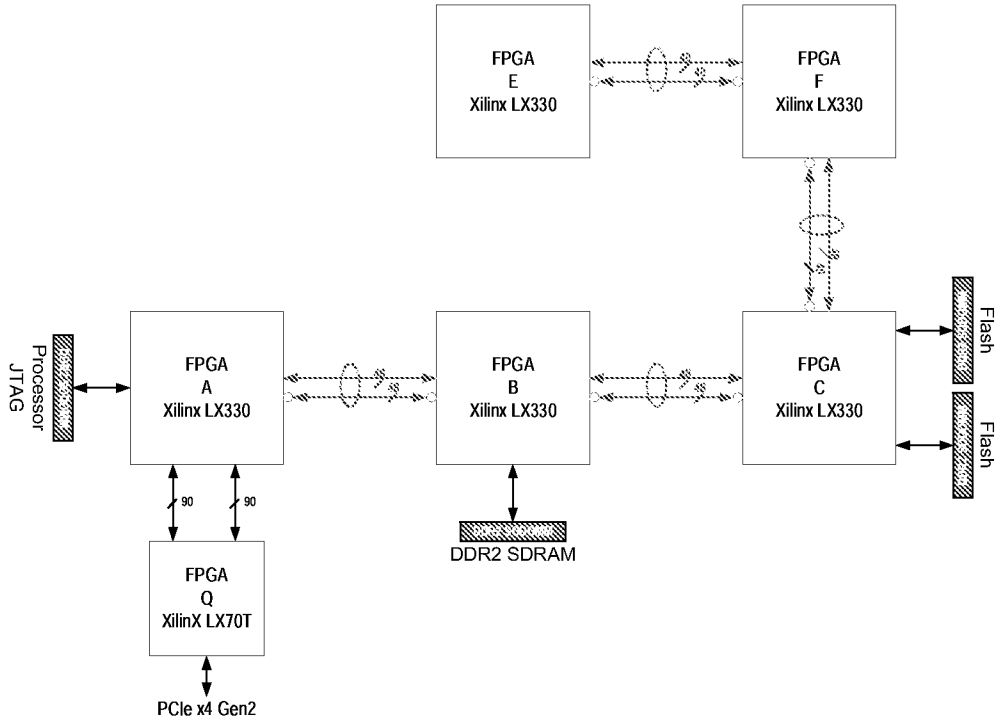


Figure 3.1 Delaware FPGA Board

3.2 FPGA Partitioning

Module	Node ID	Message Network	Buffer RAM Data Network	DRAM Data Network	CAR	Processor JTAG	JTAG	Global Interrupt
PCIe Interface	(15,0)				X			
CAR Slave Debug Module	(15,8)				X			
CAR Master Debug Module					X			

Table 3.1 FPGA Q

Module	Node ID	Message Network	Buffer RAM Data Network	DRAM Data Network	CAR	Processor JTAG	JTAG	Global Interrupt
Host Interface	(0,0)	X	X	X	X			
Processor Node 64KB I-RAM and 64KB D-RAM	(0,1)	X	X	X	X	X		
Processor Node (optional) 64KB I-RAM and 64KB D-RAM	(1,1)	X	X	X	X	X		
Message Network Debug Node	(1,0)	X			X			
CAR Slave Debug Module	(14,8)				X			
DRAM Data Network Debug Node	(14,9)			X	X			
Buffer RAM Data Network Debug Node	(14,10)		X		X			

Table 3.2 FPGA A

Module	Node ID	Message Network	Buffer RAM Data Network	DRAM Data Network	CAR	Processor JTAG	JTAG	Global Interrupt
Buffer RAM	(13,0)		X		X			
DRAM Interface	(13,1)			X	X			
Processor Node 64KB I-RAM and 64KB D-RAM	(0,2)	X	X	X	X	X		
Processor Node 64KB I-RAM and 64KB D-RAM	(0,3)	X	X	X	X	X		
Message Network Debug Node	(1,2)	X			X			

Table 3.3 FPGA B

Module	Node ID	Message Network	Buffer RAM Data Network	DRAM Data Network	CAR	Processor JTAG	JTAG	Global Interrupt
CAR Slave Debug Module	(13, 8)				X			
DRAM Data Network Debug Node	(13,9)			X	X			
Buffer RAM Data Network Debug Node	(13,10)		X		X			

Table 3.3 FPGA B

Module	Node ID	Message Network	Buffer RAM Data Network	DRAM Data Network	CAR	Processor JTAG	JTAG	Global Interrupt
Flash Controller	(0,4)	X	X	X	X			
ECC Correction Unit	(12,0)		X	X	X			
Processor Node 64KB I-RAM and 64KB D-RAM	(0,5)	X	X	X	X	X		
Processor Node (optional) 64KB I-RAM and 64KB D-RAM	(1,4)	X	X	X	X	X		
Processor Node (optional) 64KB I-RAM and 64KB D-RAM	(1,5)	X	X	X	X	X		
Message Network Debug Node	(2,4)	X			X			
CAR Slave Debug Module	(12, 8)				X			
DRAM Data Network Debug Node	(12,9)			X	X			
Buffer RAM Data Network Debug Node	(12,10)		X		X			

Table 3.4 FPGA C

Module	Node ID	Message Network	Buffer RAM Data Network	DRAM Data Network	CAR	Processor JTAG	JTAG	Global Interrupt
Processor Node 32KB I-RAM and 32KB D-RAM	(0,6)	X	X	X	X	X		
Processor Node 32KB I-RAM and 32KB D-RAM	(0,7)	X	X	X	X	X		
Processor Node (optional) 32KB I-RAM and 32KB D-RAM	(1,6)	X	X	X	X	X		

Table 3.5 FPGA F

Module	Node ID	Message Network	Buffer RAM Data Network	DRAM Data Network	CAR	Processor JTAG	JTAG	Global Interrupt
Processor Node (optional) 32KB I-RAM and 32KB D-RAM	(1,7)	X	X	X	X	X		
Message Network Debug Node	(2,7)	X			X			
CAR Slave Debug Module	(11, 8)				X			
DRAM Data Network Debug Node	(11,9)			X	X			
Buffer RAM Data Network Debug Node	(11,10)		X		X			

Table 3.5 FPGA F

Module	Node ID	Message Network	Buffer RAM Data Network	DRAM Data Network	CAR	Processor JTAG	JTAG	Global Interrupt
Processor Node 32KB I-RAM and 32KB D-RAM	(0,8)	X	X	X	X	X		
Processor Node 32KB I-RAM and 32KB D-RAM	(0,9)	X	X	X	X	X		
Processor Node (optional) 32KB I-RAM and 32KB D-RAM	(1,8)	X	X	X	X	X		
Processor Node 32KB I-RAM and 32KB D-RAM	(1,9)	X	X	X	X	X		
Message Network Debug Node	(2,9)	X			X			
CAR Slave Debug Module	(10, 8)				X			
DRAM Data Network Debug Node	(10,9)			X	X			
Buffer RAM Data Network Debug Node	(10,10)		X		X			

Table 3.6 FPGA E

3.3 Message Network

- Figure 3.2 shows the message network implemented in the Delaware FPGA prototype.
  - Blue arrows represent normal connections between node routers
  - Green arrows represent inter-FPGA connections
  - Red arrows represent "terminate East" or "terminate West" connections

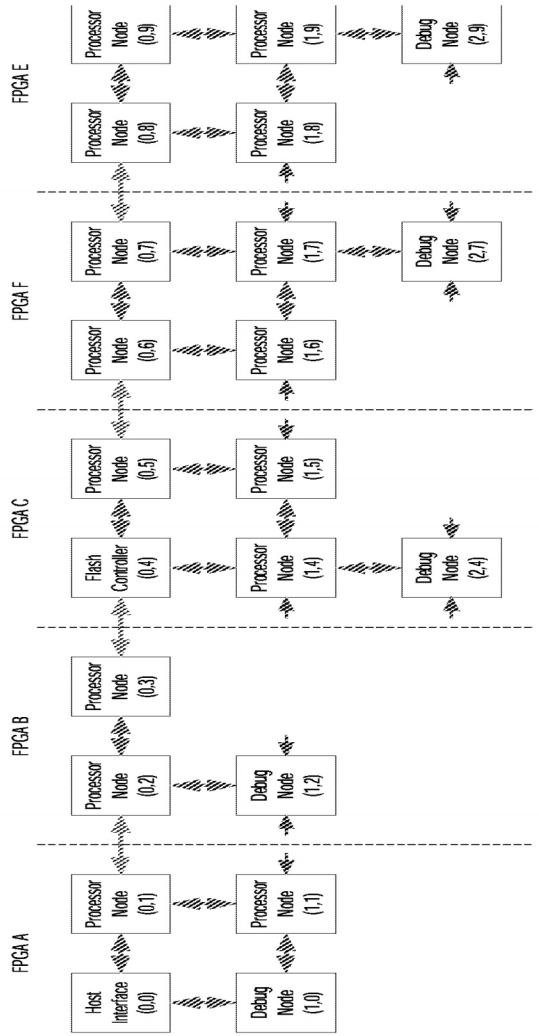


Figure 3.2 Message Network

3.4 Buffer RAM Data Network

- Figure 3.3 shows the configuration of the buffer RAM data network. The request path is a tree as shown while the completion path is a broadcast bus. Blue elements represent routing elements, blue lines represent intra-FPGA connections while orange lines represent inter-FPGA connections.

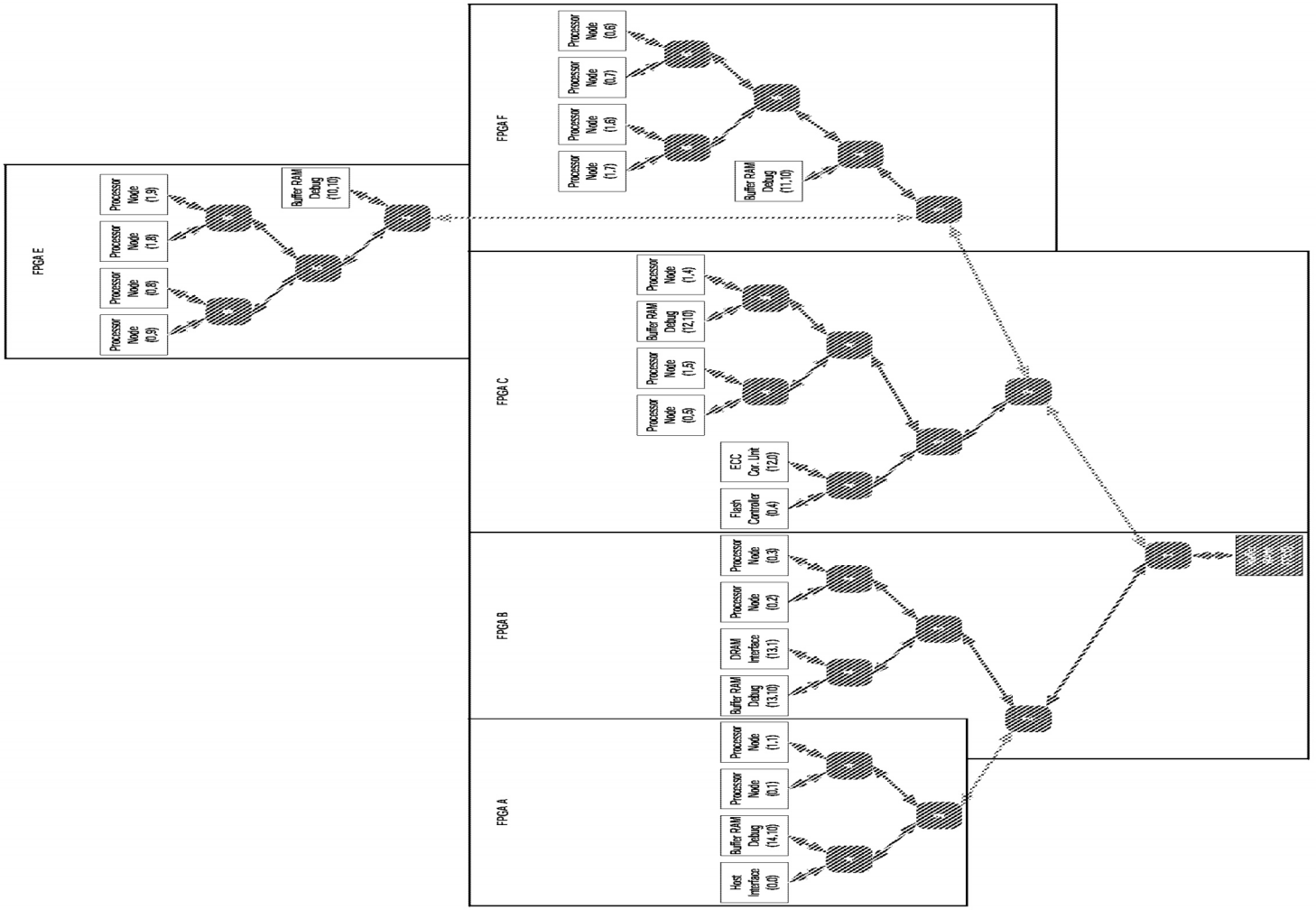


Figure 3.3 Buffer RAM Data Network

3.5 DRAM Data Network

- Figure 3.4 shows the configuration of the DRAM data network. The request path is a tree as shown while the completion path is a broadcast bus. Blue elements represent routing elements, blue lines represent intra-FPGA connections while orange lines represent inter-FPGA connections.

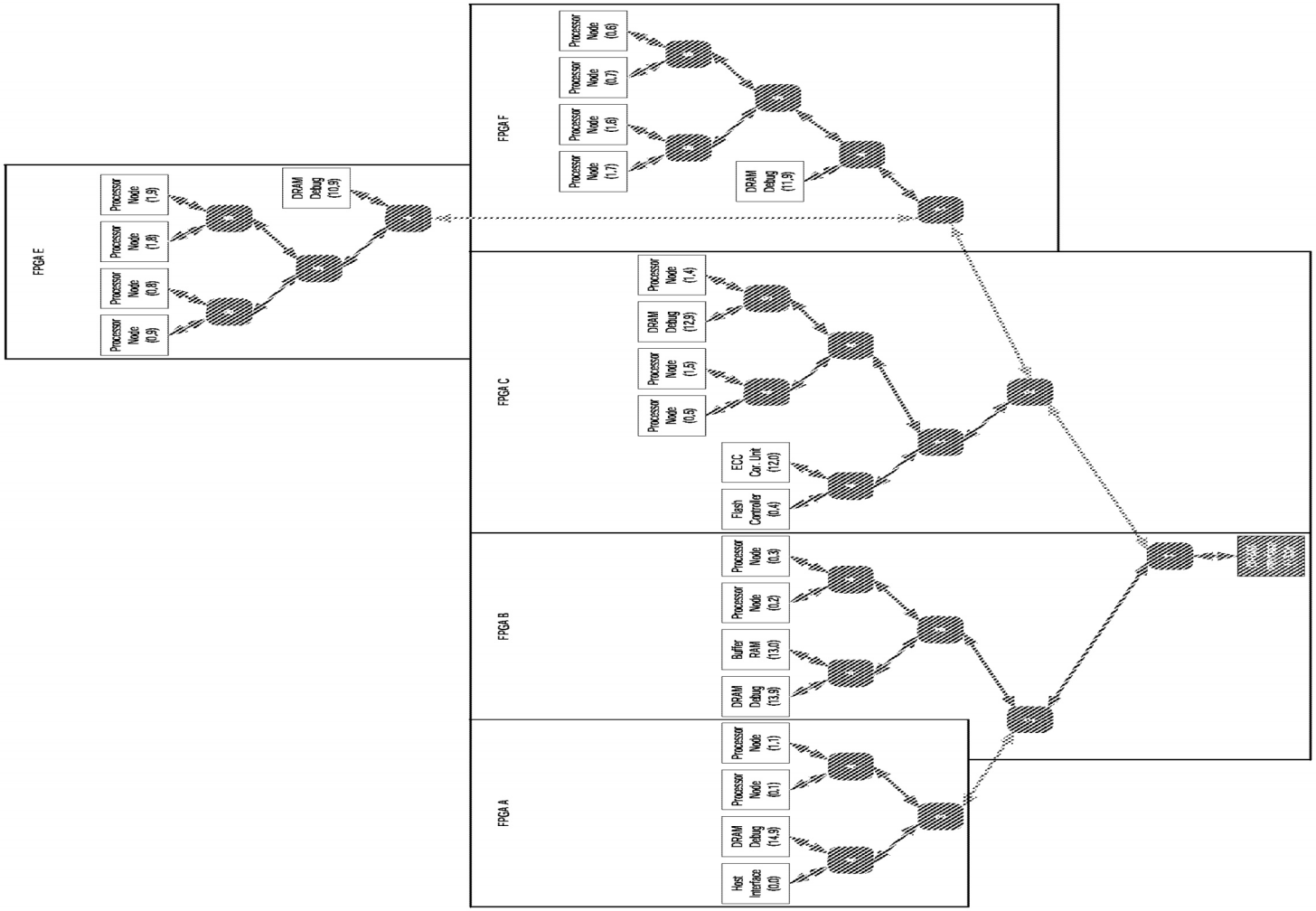


Figure 3.4 DRAM Data Network

Memory and Hardware Data Structures

4.1 Introduction

- This section describes the memory regions and the controller data structures used by Delaware hardware.

4.2 Memory

- Delaware implements three distinct memory regions: controller memory, node memory and the Common Access Ring (CAR).

4.2.1 Controller Memory

- Controller memory is used to store controller data structures and consists of buffer RAM and external DRAM managed by the DRAM controller.

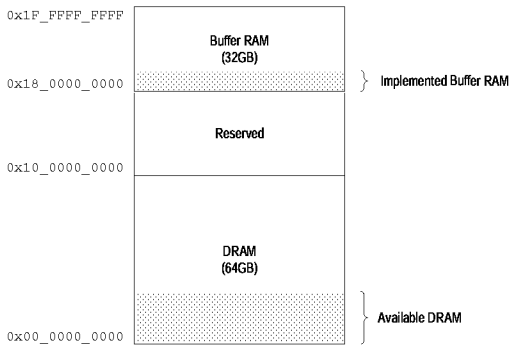


Figure 4.1 Controller Memory Map

- Controller memory has a flat 37-bit byte address. The memory map for controller memory is shown in Figure 4.1.
  - The upper address bit (i.e., address bit 37) indicates if the address is accessing DRAM or buffer RAM.
    - 0 = DRAM
    - 1 = buffer RAM
  - Architecturally Delaware supports up to 64 GB of DRAM; however, the amount of actual DRAM is implementation specific.
    - The Delaware FPGA prototype board implements 1 GB of DRAM.
  - Architecturally Delaware supports up to 32 GB of buffer RAM; however, the amount of actual memory is implementation specific.
    - The Delaware FPGA prototype implements tbd byte of buffer RAM.

4.2.2 Node Address Space

- As shown in Figure 4.2, associated with each node is a 512KB node address space.

- The bottom 384 KB of the node address space is accessible via the message network (i.e., 0x0_0000 through 0x5_FFFF). This space is typically used to hold message queue memory (see Section 6.3. "Message Queues" on page 6-4).
  - Whether or not in this space is CAR accessible is node dependent. Unless otherwise stated in the node description, this space is not CAR accessible.
- 0x6_0000 through 0x7_FEFF is accessible via the CAR and typically holds node specific CSR registers.
- 0x7_FF00 through 0x7_FFFF holds message network receive and free queues. These queues are accessible via both the CAR and message network.

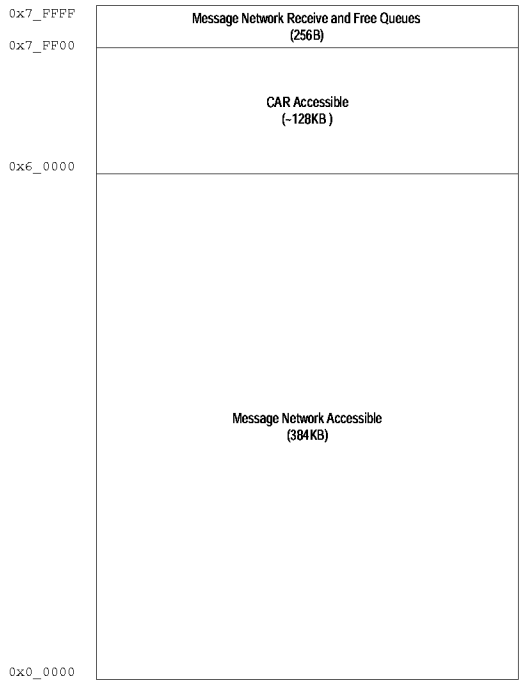


Figure 4.2 Node Address Space

- A node may architecturally support up to 32 outbound message queues and 32 inbound message queues (the number of supported inbound and outbound message queues supported by a node need not be equal).
- Associated with each outbound message queue is free queue address and associated with each inbound message queue is a receive queue address.
  - Writing a value to a queue address enqueues the value into the corresponding queue. This mechanism is used by message network packets to enqueue values into the receive and free queues (i.e., the node address is the message network packet points to a queue address).
  - A node need not implement all the inbound and outbound message queues supported by the architecture. Writing a value to an address associated with an unimplemented queue produces undefined results.
  - Queue addresses only support write operations. Reading from a queue address produces undefined results.
  - Although queue addresses are shown using a byte addresses, only DWord writes are supported. Write operations to a queue address using anything other than a DWord access produces undefined results.

Node Address	Description
0x7_FFFC	Inbound Message Queue 31 Receive Queue
0x7_FFF8	Inbound Message Queue 30 Receive Queue
0x7_FFF4	Inbound Message Queue 29 Receive Queue
0x7_FFF0	Inbound Message Queue 28 Receive Queue
0x7_FFEC	Inbound Message Queue 27 Receive Queue
0x7_FFE8	Inbound Message Queue 26 Receive Queue
0x7_FFE4	Inbound Message Queue 25 Receive Queue
0x7_FFE0	Inbound Message Queue 24 Receive Queue
0x7_FFDC	Inbound Message Queue 23 Receive Queue
0x7_FFD8	Inbound Message Queue 22 Receive Queue
0x7_FFD4	Inbound Message Queue 21 Receive Queue
0x7_FFD0	Inbound Message Queue 20 Receive Queue
0x7_FFCC	Inbound Message Queue 19 Receive Queue
0x7_FFC8	Inbound Message Queue 18 Receive Queue
0x7_FFC4	Inbound Message Queue 17 Receive Queue
0x7_FFC0	Inbound Message Queue 16 Receive Queue
0x7_FFBC	Inbound Message Queue 15 Receive Queue
0x7_FFB8	Inbound Message Queue 14 Receive Queue
0x7_FFB4	Inbound Message Queue 13 Receive Queue
0x7_FFB0	Inbound Message Queue 12 Receive Queue
0x7_FFAC	Inbound Message Queue 11 Receive Queue
0x7_FFA8	Inbound Message Queue 10 Receive Queue
0x7_FFA4	Inbound Message Queue 9 Receive Queue
0x7_FFA0	Inbound Message Queue 8 Receive Queue
0x7_FF9C	Inbound Message Queue 7 Receive Queue
0x7_FF98	Inbound Message Queue 6 Receive Queue
0x7_FF94	Inbound Message Queue 5 Receive Queue
0x7_FF90	Inbound Message Queue 4 Receive Queue
0x7_FF8C	Inbound Message Queue 3 Receive Queue
0x7_FF88	Inbound Message Queue 2 Receive Queue
0x7_FF84	Inbound Message Queue 1 Receive Queue
0x7_FF80	Inbound Message Queue 0 Receive Queue
0x7_FF7C	Outbound Message Queue 31 Free Queue
0x7_FF78	Outbound Message Queue 30 Free Queue
0x7_FF74	Outbound Message Queue 29 Free Queue
0x7_FF70	Outbound Message Queue 28 Free Queue
0x7_FF6C	Outbound Message Queue 27 Free Queue
0x7_FF68	Outbound Message Queue 26 Free Queue
0x7_FF64	Outbound Message Queue 25 Free Queue
0x7_FF60	Outbound Message Queue 24 Free Queue
0x7_FF5C	Outbound Message Queue 23 Free Queue
0x7_FF58	Outbound Message Queue 22 Free Queue
0x7_FF54	Outbound Message Queue 21 Free Queue
0x7_FF50	Outbound Message Queue 20 Free Queue
0x7_FF4C	Outbound Message Queue 19 Free Queue
0x7_FF48	Outbound Message Queue 18 Free Queue
0x7_FF44	Outbound Message Queue 17 Free Queue
0x7_FF40	Outbound Message Queue 16 Free Queue
0x7_FF3C	Outbound Message Queue 15 Free Queue

Table 4.1 Message Queue Node Addresses

Node Address	Description
0x7_FF38	Outbound Message Queue 14 Free Queue
0x7_FF34	Outbound Message Queue 13 Free Queue
0x7_FF30	Outbound Message Queue 12 Free Queue
0x7_FF2C	Outbound Message Queue 11 Free Queue
0x7_FF28	Outbound Message Queue 10 Free Queue
0x7_FF24	Outbound Message Queue 9 Free Queue
0x7_FF20	Outbound Message Queue 8 Free Queue
0x7_FF1C	Outbound Message Queue 7 Free Queue
0x7_FF18	Outbound Message Queue 6 Free Queue
0x7_FF14	Outbound Message Queue 5 Free Queue
0x7_FF10	Outbound Message Queue 4 Free Queue
0x7_FF0C	Outbound Message Queue 3 Free Queue
0x7_FF08	Outbound Message Queue 2 Free Queue
0x7_FF04	Outbound Message Queue 1 Free Queue
0x7_FF00	Outbound Message Queue 0 Free Queue

Table 4.1 Message Queue Node Addresses

4.2.3 CAR Address Space

- The Common Access Ring (CAR) is used to access functional unit control and status registers.
- Figure 4.3 show the 32-bit CAR address space.
  - Associated with each node is a 512KB node address space. A node is selected by placing its 8-bit Node ID in address bits 19 through 26.
  - CAR address space associated with a node has exactly the same memory map as the node's node address space described in Section 4.2.2, "Node Address Space" on page 4-1. The only exception is that the address space 0x7_8000 through 0x7_FFFF (message network receive and free queues) is inaccessible through CAR.

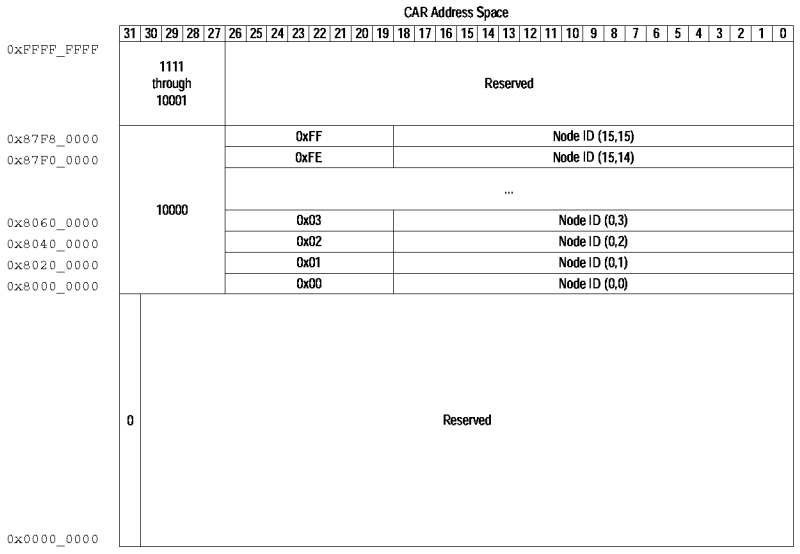


Figure 4.3 CAR Address Space

4.2.4 Global Memory Space

- This memory is provided for reference only. It is not necessary to read or understand this section in order to design or program the device.
- Although not entirely accessible through any single interface (i.e., memory network, message network or CAR), controller memory, node address space and CAR address space together form a single global address space. This layout of this global address is shown in Figure 4.4.

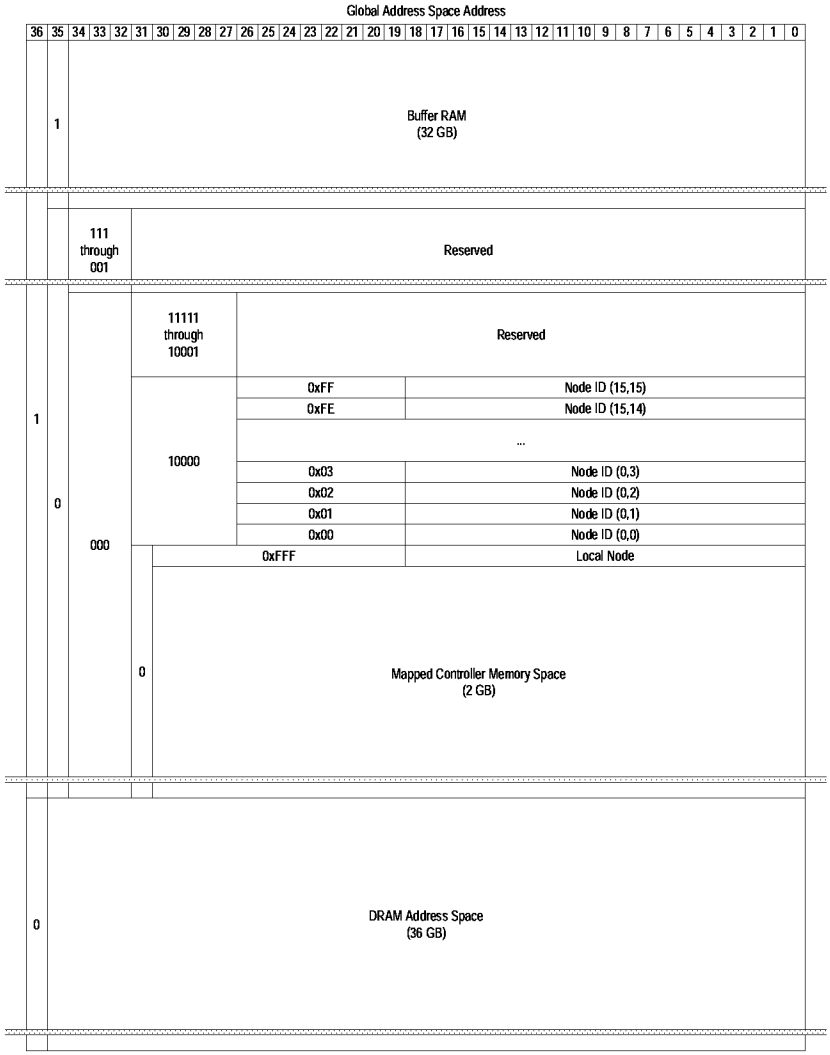


Figure 4.4 Global Memory Space

4.3 Controller Data Structures

4.3.1 Memory Line

- A memory line is a 64B quantity that is 64B aligned in controller memory.
- Controller memory is optimized for memory line transfers. When possible data structures in controller memory should be memory line aligned.

4.3.2 Memory Line Pointer

- Since a memory line must be 64B aligned in controller memory, the bottom six bits of a controller (byte) address of a memory line must be zero.
- A controller byte address is 37-bits.
- A memory line address is a 32-bit version of a controller address formed by shifting the controller address right five bits. Bit zero of a memory line address is always zero since a memory line is 64B aligned and the controller address is shift right five bits (and not six bits).
  - MemoryLine = (ControllerAddress) >> 5

4.3.3 Page Frame

- The contents of a Flash page in memory is referred to as a page frame.

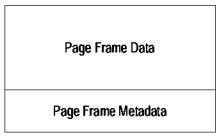


Figure 4.5 Flash Page Frame

- Figure 4.5 illustrates a page frame. A page frame consists of a page frame data and a page frame metadata region.
  - The page frame data region holds the data contents of a Flash page.
    - The page frame data region must start on a 4KB boundary (i.e., the bottom 12-bits of the byte address must be zero).
    - Delaware supports only a Flash page size of 4KB.
    - Princeton supports Flash page sizes of 4KB, 8KB and 16KB.
  - The page frame metadata region holds the metadata contents of a Flash page.
    - The page frame metadata region must start on a 256B aligned boundary (i.e., the bottom 8-bits of the byte address must be zero).
    - Delaware supports 224B of metadata, but the page frame metadata region is 256B (i.e., a power of two). The additional 32B are unused by functional units and may be used by the firmware. These 32B however are not written or read from Flash.

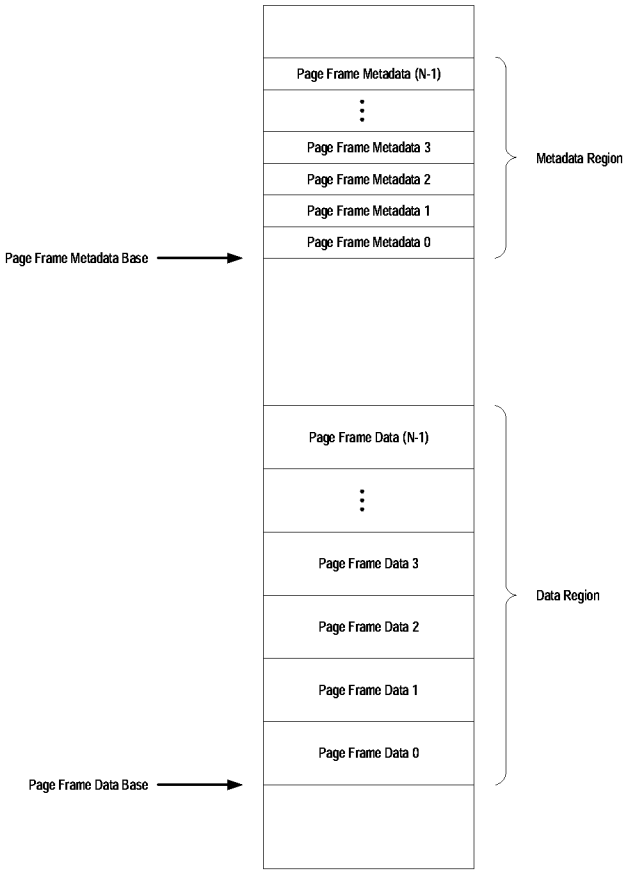


Figure 4.6 Page Frame Layout in Controller Memory

- Figure 4.6 shows the layout of page frames in controller memory. Each of the two memories that together form controller memory (i.e., Buffer RAM and DRAM) have their own version of the page frame layout shown in Figure 4.6. A page frame must be entirely contained in Buffer RAM or DRAM.
  - A page frame's data and metadata are split into two contiguous regions. The data region stores page frame data while the metadata region stores page frame metadata.
    - Page frame data and metadata regions are each a power of two in size. This organization both minimizes wasted space and simplifies page indexing.
    - The page frame data region must start on a 4KB boundary and the page frame meta data region must start on a 256B boundary.
  - Firmware, the flash controller, and the ECC correction unit need to know the start of these regions. No other functional unit understands page frames.
    - The Buffer RAM Page Frame Data Base (BRPFDB) and Buffer RAM Page Frame Metadata Base (BRPFMB) registers hold the base address pointers for page frames in buffer ram.
    - The DRAM Page Frame Data Base (DRPFDB) and DRAM Page Frame Metadata Base (DRPFMB) registers hold the base address pointers for page frames in DRAM.
    - Note that the registers hold truncated versions of these addresses and that the values held in these registers need to be shifted left eight or 12 bits to form a byte address.

- Given a page frame number, the address of the data and metadata regions in controller memory may be computed as follows.
  - DataRegion = PageFrameDataBase +(PageNumber << 12)
  - MetadataRegion = PageFrameMetadataBase + (PageNumber << 8)
- Page frame information is passed in FlashRead and FlashWrite messages to the Flash controller in the memory address (Addr) field. This field contains the upper 37-bits of the controller address that points to the start of the data region (i.e., DataRegion). The start of the page metadata region may be determined as follows.
  - PageNumber = (DataRegion - PageFrameDataBase) >> 12
  - MetadataRegion = PageFrameMetadataBase + (PageNumber << 8)

Memory Network

5.1 Introduction

- The Delaware memory network is implemented using two data networks: a Buffer RAM Data Network (BRDN) and a Data RAM Data Network (DRDN). The data networks are used to interconnect processors and functional units to the buffer RAM and DRAM interface which together form controller memory.
  - Although there are two memory networks, software is isolated from this implementation and views controller memory as a flat 37-bit address space. The address space is described in Section 4.2.1, "Controller Memory" on page 4-1.
  - Both memory networks operate in the same manner; therefore, the remainder of the chapter describes a generic memory network.

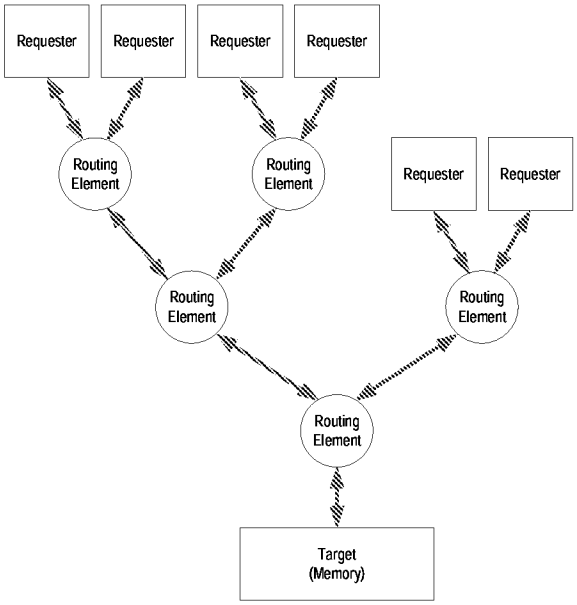


Figure 5.1 Memory Network

- As shown in Figure 5.1 the memory network may be viewed as a tree that routes requests from a requester to the target (i.e., memory) and routes completions from the target to the requester.

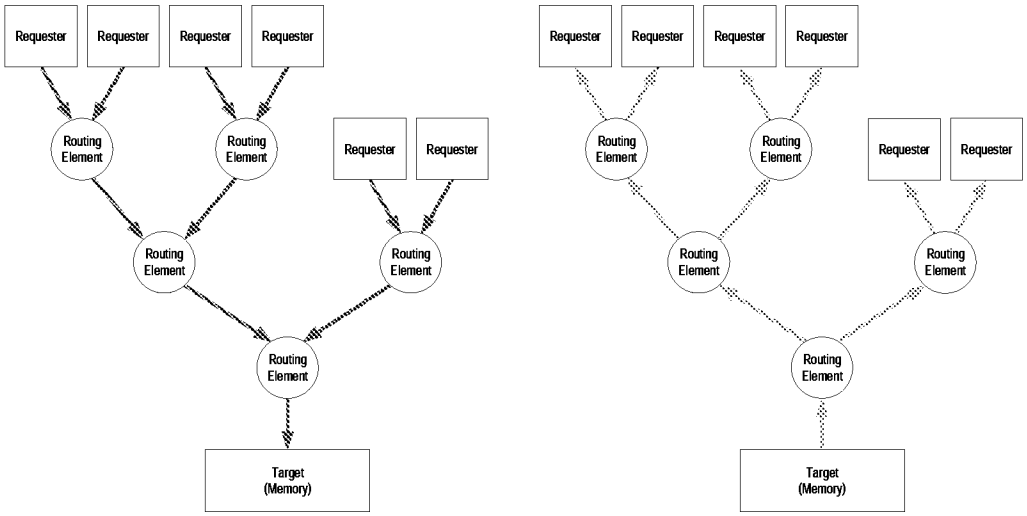


Figure 5.2 Memory Network Request and Completion Routing

- The routing of requests from a requester to a target is shown in Figure 5.2(a).
  - A request consists of a single variable length packet consisting of a header followed by up to four data lines.
    - A request header is 256-bits and its format is described in Section 5.3, "Memory Network Packets" on page 5-4.
    - A memory line is 512-bits (64B) and its format is described in Section 4.3.1, "Memory Line" on page 4-5.
  - Request routing elements route memory network requests to the target. It has two inputs and one output.
    - The routing element arbitrates between requests on its inputs and forwards the complete winning request on its output.
    - Request elements use fair arbitration to arbitrate among requests on its inputs. The request tree should be balanced since this gives each requester fair access to the target. As shown in example in Figure 5.2(a), in some cases it is not possible to build a balanced tree; however, the architecture of the Flash controller is such that target memory bandwidth exceeds the average bandwidth of all requesters. Therefore, arbitration is used simply as a mechanism to smooth bursty requests and not as mechanism for QoS.
- The routing of completions from a target to a requester is shown in Figure 5.2(b).
  - A completion consists of a single variable length packet consisting of a header followed by up to four data lines.
    - A completion header is 256-bits and its format is described in Section 5.3, "Memory Network Packets" on page 5-4.
    - A memory line is 512-bits (64B) and its format is described in Section 4.3.1, "Memory Line" on page 4-5.
  - Completion routing elements route memory network completions from a target back to the requester. It has one input and two outputs.

- Unlike the requester routing elements that need to arbitrate among multiple requester, the completion routing elements serve only a single target. Therefore, no arbitration is required and the completion routing element simply forwards the completion from the target to the appropriate output.
- In the Delaware FPGA prototype completions are broadcast from the target to all requesters. Requesters accept only those completions for which they were the original requester (as determined by the requester ID field). Therefore, completion routing elements may be viewed simply as a wire connection that forms a broadcast tree.

- A memory network requester may not back pressure a completion for any reason; therefore, a requester must ensure that it has allocated sufficient buffering to accept an entire completion prior to issuing a corresponding request.

5.2 Ordering Rules

- Memory network requests and completion are routed as atomic units (i.e., portions of a request from different requesters are never interleaved).
- Requests issued from different requesters may be routed to the target (i.e., memory) in any order.
- There are no ordering guarantees among requests issued to different targets (i.e., Buffer RAM and DRAM). These requests may be routed and may be completed in any order.
- There are no ordering guarantees among requests issued from different requesters. These requests may be routed and completed in any order.
- Except as noted below, requests issued from the same requester may be completed in any order as long as data dependencies are preserved.
- The following requests do not require a completion for functional operation and may be issued in a posted manner (i.e., with the Acknowledge bit cleared in the header causing no completion to be generated for the corresponding request).
  - Write(Wr)
  - Write Line (WrLine)
  - Read XOR and Write (RdXorWr)
  - Read Add and Write (RdAddWr)
  - Transfer (Xfer).
- Issuing one of the above requests in a non-posted manner (i.e., with the Acknowledge bit set in the header) creates an ordering barrier as described below.
  - The reception of a completion to a non-posted Write (Wr) or Write Line (WrLine) request indicates that all prior Wr and WrLine requests from that requester have been globally completed.
  - The reception of a completion to a non-posted Read XOR and Write (RdXorWr) request indicates that all prior RdXorWr requests from that requester have been globally completed.
  - The reception of a completion to a non-posted Read Add and Write (RdAddWr) request indicates that all prior RdAddWr requests from that requester have been globally completed.
  - The reception of a completion to a non-posted Transfer (Xfer) request indicates that all prior Xfer requests from that requester have been globally completed.

- The above ordering rules are independent. For example, a completion to a non-posted Wr does not mean that all prior RdXorWR requests from that requester have been globally completed.
- An operation is globally completed if the side effect of the operation (e.g., memory write value) is visible to all other requesters.
- Implementation notes
  - In the Delaware FPGA prototype all Buffer RAM and DRAM operations are performed in order (i.e., in the order in which they are received by the corresponding target).
  - In future ASIC implementations the Buffer RAM or DRAM controller may aggressively reorder operations. Such implementations are free to do so as long as the above memory ordering rules are satisfied.
  - Firmware should be written with comprehension of these ordering rules to ensure correct execution on any implementation.

5.3 Memory Network Packets

Bit Range	Field Size	Rd	RdLock	RdXorWr	RdAddWr	Wr	WrLine	Xfer
2:0	3	Length	Length	Length	Length	Length	Length	Length
6:3	4	Type	Type	Type	Type	Type	Type	Type
36:7	30	Address	Address	Address	Address	Address	Address	Address
44:37	8	RequesterID	RequesterID	RequesterID	RequesterID	RequesterID	RequesterID	RequesterID
52:45	8	RequestTag	RequestTag	RequestTag	RequestTag	RequestTag	RequestTag	RequestTag
53	1	Acknowledge	Acknowledge	Acknowledge	Acknowledge	Acknowledge	Acknowledge	Acknowledge
55:54	2	ReqLength	ReqLength	ReqLength	reserved	reserved	reserved	ReqLength
59:56	4	reserved	LockBit	reserved	AddDWord	WordEnable (64-bits)	ByteEnable (64-bits)	WordEnable (64-bits)
66:60	7	reserved	LockBit	reserved	reserved			
87:67	21	reserved	reserved	reserved	reserved			
119:88	32	reserved	reserved	reserved	AddValue			
149:120	30	reserved						TAddress
191:150	42	reserved						
255:192	64	ReqExtTag	ReqExtTag	ReqExtTag	ReqExtTag	ReqExtTag	ReqExtTag	ReqExtTag

Table 5.1 Request Packet Header Field Bit Allocation

Type Field Value	Request Command
0	Rd
1	RdLock
2	RdXorWr (Non-Atomic)

Table 5.2 Request Packet Type Field Encoding

Type Field Value	Request Command
3	Wr
4	WrLine
5	Xfer
6	RdAddWr
7	RdXorWr (Atomic)

Table 5.2 Request Packet Type Field Encoding

Bit Range	Field Size	Cpl
2:0	3	Length
3	1	MalformedReq
4	1	CorrrectedErr
5	1	UncorrectedErr
6	1	LockGranted
36:7	30	reserved
44:37	8	RequesterID
52:45	8	RequesteTag
191:53	139	reserved
255:192	64	ReqExtTag

Table 5.3 Completion Packet Header Field Bit Allocation

5.4 Transactions

Request	Description
Rd	Read. Read one or more memory lines.
RdLock	Read and Lock. Test lock bit. If the lock bit is cleared, then atomically set the lock bit and return data lines from memory.
RdXorWr	Read XOR and Write. Read one or more lines from memory, XOR it with the data contained in the request, and write it back to memory. This process is executed as an atomic operation.
RdAddrWr	Read Add and Write. Read one line from memory, add a 32-bit value to one of the DWords in that memory line, write the memory line back to memory. This process is executed as an atomic operation.
Wr	Write. Write one or more memory lines to memory.
WrLine	Write Memory Line. Write memory line with data contained in the request. Unlike a Write request, a WriteLine request contains byte enables.

Table 5.4 Memory Network Requests

Request	Description
Xfer	Transfer. Read one or more memory lines from memory and use this data to form a Write request on the other data network.

Table 5.4 Memory Network Requests

- If a malformed requester or an uncorrected error is detected, then a completion is generated for the request regardless of the setting of the Acknowledge bit in the request.
- Memory network requests contain a Requester Tag (RequesterTag) field. In non-posted transactions this field is passed unmodified from the request to the corresponding completion. Functional units that support multiple outstanding requests may use this field to map completions to corresponding requests. The manner in which this is done is functional unit specific.
    - As described in Section 5.2, "Ordering Rules" on page 5-3 the architecture allows requests issued from the same requester to be completed in any order as long as data dependencies are preserved. While this is true in general, targets in the Delaware FPGA prototype (i.e., Buffer RAM and DRAM) always complete operation from a requester in the order in which they were issued. A functional unit in the FPGA prototype may use ordering guarantee instead of the RequesterTag field (i.e., use of the RequesterTag field is optional in the FPGA prototype).

5.4.1 Completion (Cpl)

Field	Size (bits)	Description
Length	3	Packet Length. This field contains the number of data lines contained in the completion. Valid values are zero through four. 0 - Header only 1 - Header and one data line 2 - Header and two data lines 3 - Header and three data lines 4 - Header and four data lines others - reserved
RequesterID	8	Requester ID. This field encodes the ID of the corresponding request and is used as a routing tab for the completion.
RequesterTag	8	Requester Tag. This field contains the tag associated with the corresponding request.
MalformedRequest	1	Malformed Request. This bit is set if the completion corresponds to a malformed request.
CorrectedErr	1	Corrected Error. This bit is set if an error was detected and corrected during the processing of the corresponding request.
UncorrectedErr	1	Uncorrected Error. This bit is set if an error was detected in the processing of the corresponding request that could not be corrected.
LockGranted	1	Lock Granted. This bit is set if the corresponding request requested a lock and the lock was granted.

Table 5.5 Completion (Cpl) Header Fields

Field	Size (bits)	Description
ReqExtTag	64	Requester Extended Tag. This field is viewed as opaque data by the target memory and is passed unmodified from the request to the completion. This field is optionally populated by the requester and contains information regarding how the corresponding completion, when it is received by the requester, should be processed. The purpose of this field is to minimize state information maintained at the requester, thus, the format of this field is requester specific.

Table 5.5 Completion (Cpl) Header Fields

5.4.2 Read (Rd)

Field	Size (bits)	Description
Length	3	Packet Length. This field contains the number of data lines contained in the request. The only valid value of this field for this request is zero. 0 - Header only
Type	4	Request Type. The value in this field corresponds to the corresponding request.
Address	30	Address. This field contains the starting memory line address associated with the request. The byte address is formed by shifting the value in this field left by six bits.
ReqLength	2	Request Length. This field encodes the number of data lines to be read from memory. The number of requested data lines corresponds to the value in this field plus one (i.e., zero corresponds to one data line, one corresponds to two data lines, and so on).
RequesterID	8	Requester ID. This field encodes the ID of the corresponding request and is used as a routing tag for the completion.
RequesterTag	8	Requester Tag. This field contains the tag associated with the corresponding request.
Acknowledge	1	Acknowledge. When this bit is set a completion is generated for the corresponding request. This bit must always be set in Read requests.
ReqExtTag	64	Requester Extended Tag. This field is viewed as opaque data by the target memory and is passed unmodified from the request to the completion. This field is optionally populated by the requester and contains information regarding how the corresponding completion, when it is received by the requester, should be processed. The purpose of this field is to minimize state information maintained at the requester, thus, the format of this field is requester specific.

Table 5.6 Read (Rd) Header Fields

5.4.3 Read and Lock (RdLock)

Field	Size (bits)	Description
Length	3	Packet Length. This field contains the number of data lines contained in the request. The only valid value of this field for this request is zero. 0 - Header only
Type	4	Request Type. The value in this field corresponds to the corresponding request.
Address	30	Address. This field contains the starting memory line address associated with the request. The byte address is formed by shifting the value in this field left by six bits.
ReqLength	2	Request Length. This field encodes the number of data lines to be read from memory. The number of requested data lines corresponds to the value in this field plus one (i.e., zero corresponds to one data line, one corresponds to two data lines, and so on).
RequesterID	8	Requester ID. This field encodes the ID of the corresponding request and is used as a routing tag for the completion.
RequesterTag	8	Requester Tag. This field contains the tag associated with the corresponding request.
Acknowledge	1	Acknowledge. When this bit is set a completion is generated for the corresponding request. This bit must always be set in Read and Lock requests.
LockBit	11	Lock Bit Position.This field encodes the bit position of the lock bit. Up to four memory lines may be requested to be read from memory; however the lock bit must be in the first memory line. This field encodes the bit number of the lock bit in the first memory line. Valid values are 0 through 511.
ReqExtTag	64	Requester Extended Tag. This field is viewed as opaque data by the target memory and is passed unmodified from the request to the completion. This field is optionally populated by the requester and contains information regarding how the corresponding completion, when it is received by the requester, should be processed. The purpose of this field is to minimize state information maintained at the requester, thus, the format of this field is requester specific.

Table 5.7 Read and Lock (RdLock) Header Fields

5.4.4 Read XOR and Write (RdXorWr)

- There is an atomic and non-atomic version of this request. Both functionally operate in the same manner.

Field	Size (bits)	Description
Length	3	Packet Length. This field contains the number of data lines contained in the request. The only valid value of this field for this request is one. 1 - Header and one data line others - reserved
Type	4	Request Type. The value in this field corresponds to the corresponding request.
Address	30	Address. This field contains the starting memory line address associated with the request. The byte address is formed by shifting the value in this field left by six bits.

Table 5.8 Read XOR and Write (RdXorWr) Header Fields

Field	Size (bits)	Description
ReqLength	2	Request Length. This field encodes the number of data lines to be read from memory. The number of requested data lines corresponds to the value in this field plus one (i.e., zero corresponds to one data line, one corresponds to two data lines, and so on). The value of this field must correspond to the number of data lines contained in the request (e.g., if the Length field value corresponds to a header and two data lines, then the Request Length field must be set to a the value that corresponds to two data lines).
RequesterID	8	Requester ID. This field encodes the ID of the corresponding request and is used as a routing tag for the completion.
RequesterTag	8	Requester Tag. This field contains the tag associated with the corresponding request.
Acknowledge	1	Acknowledge. When this bit is set a completion is generated for the corresponding request.
ReqExtTag	64	Requester Extended Tag. This field is viewed as opaque data by the target memory and is passed unmodified from the request to the completion. This field is optionally populated by the requester and contains information regarding how the corresponding completion, when it is received by the requester, should be processed. The purpose of this field is to minimize state information maintained at the requester, thus, the format of this field is requester specific.

Table 5.8 Read XOR and Write (RdXorWr) Header Fields

Read Add and Write (RdAddWr)

Field	Size (bits)	Description
Length	3	Packet Length. This field contains the number of data lines contained in the request. The only valid value of this field for this request is zero. 0 - Header only
Type	4	Request Type. The value in this field corresponds to the corresponding request.
Address	30	Address. This field contains the starting memory line address associated with the request. The byte address is formed by shifting the value in this field left by six bits.
RequesterID	8	Requester ID. This field encodes the ID of the corresponding request and is used as a routing tag for the completion.
RequesterTag	8	Requester Tag. This field contains the tag associated with the corresponding request.
Acknowledge	1	Acknowledge. When this bit is set a completion is generated for the corresponding request.
AddDWord	4	This field indicates the DWord within the memory line on which the add should be performed. DWords within memory lines have a Little Endian byte ordering.
AddValue	32	This field contains the 32-bit value that is added to the DWord in the memory line selected by the AddDWord field.
ReqExtTag	64	Requester Extended Tag. This field is viewed as opaque data by the target memory and is passed unmodified from the request to the completion. This field is optionally populated by the requester and contains information regarding how the corresponding completion, when it is received by the requester, should be processed. The purpose of this field is to minimize state information maintained at the requester, thus, the format of this field is requester specific.

Table 5.9 Read Add and Write (RdAddWr) Header Fields

- Completions generated in response to this request (i.e., when the Acknowledge bit is set in the request) only contain a header. The data value is never returned.

5.4.5 Write (Wr)

Field	Size (bits)	Description
Length	3	Packet Length. This field contains the number of data lines contained in the request. Valid values are zero through four. 0 - Header only 1 - Header and one data line 2 - Header and two data lines 3 - Header and three data lines 4 - Header and four data lines others - reserved
Type	4	Request Type. The value in this field corresponds to the corresponding request.
Address	30	Address. This field contains the starting memory line address associated with the request. The byte address is formed by shifting the value in this field left by six bits.
RequesterID	8	Requester ID. This field encodes the ID of the corresponding request and is used as a routing tag for the completion.
RequesterTag	8	Requester Tag. This field contains the tag associated with the corresponding request.
Acknowledge	1	Acknowledge. When this bit is set a completion is generated for the corresponding request.
WordEnable	64	Word Enable. This field encodes the DWords that are two be written. A Write request may write up to four memory lines to memory. Each memory line consists of 64B or 16 DWords. Each bit in this field corresponds to a DWord (i.e., bit zero corresponds to DWord zero in memory line 0). When a bit is set, the corresponding DWord is written to memory.
ReqExtTag	64	Requester Extended Tag. This field is viewed as opaque data by the target memory and is passed unmodified from the request to the completion. This field is optionally populated by the requester and contains information regarding how the corresponding completion, when it is received by the requester, should be processed. The purpose of this field is to minimize state information maintained at the requester, thus, the format of this field is requester specific.

Table 5.10 Write (Wr) Header Fields

5.4.6 Write Line (WrLine)

Field	Size (bits)	Description
Length	3	Packet Length. This field contains the number of data lines contained in the request. The only valid value of this field for this request is one. 1 - Header and one data line others - reserved
Type	4	Request Type. The value in this field corresponds to the corresponding request.
Address	30	Address. This field contains the starting memory line address associated with the request. The byte address is formed by shifting the value in this field left by six bits.

Table 5.11 Write Line (WrLine) Header Fields

Field	Size (bits)	Description
RequesterID	8	Requester ID. This field encodes the ID of the corresponding request and is used as a routing tag for the completion.
RequesterTag	8	Requester Tag. This field contains the tag associated with the corresponding request.
Acknowledge	1	Acknowledge. When this bit is set a completion is generated for the corresponding request.
ByteEnable	64	Byte Enable. This field encodes the bytes that are two be written. A Write Line request contains exactly one memory line consisting of 64B. Each bit in this field corresponds to a byte. When a bit in this field is set, the corresponding byte is written.
ReqExtTag	64	Requester Extended Tag. This field is viewed as opaque data by the target memory and is passed unmodified from the request to the completion. This field is optionally populated by the requester and contains information regarding how the corresponding completion, when it is received by the requester, should be processed. The purpose of this field is to minimize state information maintained at the requester, thus, the format of this field is requester specific.

Table 5.11 Write Line (WrLine) Header Fields

5.4.7 Transfer (Xfer)

Field	Size (bits)	Description
Length	3	Packet Length. This field contains the number of data lines contained in the request. The only valid value of this field for this request is zero. 0 - Header only
Type	4	Request Type. The value in this field corresponds to the corresponding request.
Address	30	Address. This field contains the starting memory line address associated with the request. The byte address is formed by shifting the value in this field left by six bits.
ReqLength	2	Request Length. This field encodes the number of data lines to be read from memory. The number of requested data lines corresponds to the value in this field plus one (i.e., zero corresponds to one data line, one corresponds to two data lines, and so on).
RequesterID	8	Requester ID. This field encodes the ID of the corresponding request and is used as a routing tag for the completion.
RequesterTag	8	Requester Tag. This field contains the tag associated with the corresponding request.
Acknowledge	1	Acknowledge. When this bit is set a completion is generated for the corresponding request.
WordEnable	64	Word Enable. This field encodes the DWords that are two be written in the transfer operation. A Write request may write up to four memory lines to memory. Each memory line consists of 64B or 16 DWords. Each bit in this field corresponds to a DWord (i.e., bit zero corresponds to DWord zero in memory line 0). When a bit is set, the corresponding DWord is written in the transfer operation.
TAddress	30	Transfer Address. This field contains the starting memory line transfer address associated with the request. The byte address is formed by shifting the value in this field left by six bits.

Table 5.12 Transfer (Xfer) Header Fields

Field	Size (bits)	Description
ReqExtTag	64	Requester Extended Tag. This field is viewed as opaque data by the target memory and is passed unmodified from the request to the completion. This field is optionally populated by the requester and contains information regarding how the corresponding completion, when it is received by the requester, should be processed. The purpose of this field is to minimize state information maintained at the requester, thus, the format of this field is requester specific.

Table 5.12 Transfer (Xfer) Header Fields

Message Network

6.1 Introduction

- The message network is used to transfer control messages between functional units and processors. These messages are the primary means that firmware uses to control data path operations.
  - Message network messages consist of one or more message network packets.
  - Each packet that makes up a message is routed independently.
  - The format of functional unit messages are fixed by hardware and is described in Chapter 14. The format of messages between processors is under firmware control.
- The message network is a 2D mesh network consisting of nodes. (i.e., hardwired functional units or processors with a message network interface). Each node has a unique node ID that is used for packet routing.
  - The node ID is 8-bits in size and consists of two sub-fields: a 4-bit row ID and a 4-bit column ID.
  - The node in row one column two has a node ID of 0x12 and may be written in row and column format as (1,2).

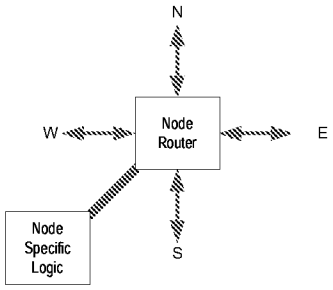


Figure 6.1 Message Network Node

- A node in the message network is shown in Figure 6.1 and consists of a node router and node specific logic. Node specific logic may be a hardwired functional unit or a processor. The node router is responsible for routing message network packets between node specific logic and the message network and for packets simply passing through the node router on their way to their ultimate destination.

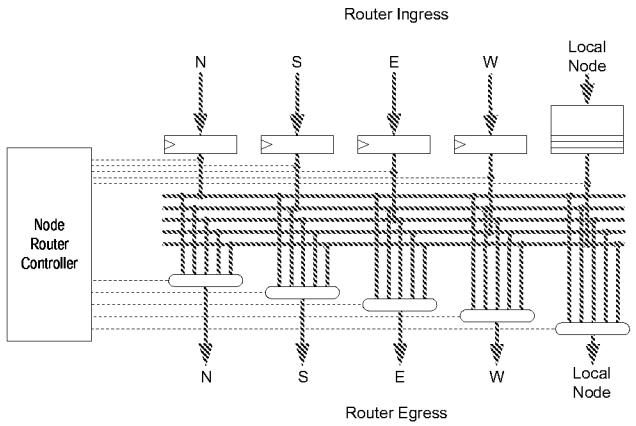


Figure 6.2 Simplified Message Network Node Router Architecture

- Figure 6.2 presents a simplified view of a node router. Packets from other nodes (i.e., North, South, East and West directions) are registered. Local node packets to be transmitted may be queued in a small output queue. The node router controller in the node router examines routing information in each packet and enables the corresponding output multiplexor(s).
  - Packets are registered in an input register.
  - Routing decisions are made in one clock cycle and the node router is able to route back-to-back packets.
  - The latency through a node router is one clock cycle.
  - Although not sown, a back pressure mechanism exists.
  - Two packets may be routed in a single cycle. For example, a packet from node specific logic going North may be routed in the same cycle as a packet from the West going East.

```
// Inputs
//
//      PktRow - destination node row (horizontal)
//      PktColumn - destination node column (vertical)
//
//      NodeRow - current node row ID
//      NodeColumn - current node column ID
//

rowOffset = PktRow - NodeRow
columnOffset = PktColumn - NodeColumn

If (columnOffset < 0) {
    output West
} else if (columnOffset > 0) {
    output East
} else if (rowOffset < 0) {
    output North
} else if (rowOffset > 0) {
    output South
} else {
    local Node
}
```

Figure 6.3 Node Router Controller Routing Algorithm

- The message network utilizes dimension-order routing. Packets are first routed horizontally (i.e., row) and then vertically (i.e., column).

- A simplified version of the algorithm used by the node router controller to route packets is shown in Figure 6.3.
- The following are differences between the simplified algorithm and the actual node controller routing algorithm.
  - The actual node router controller utilizes a fairness algorithm to prevent starvation.
  - The actual node router controller attempts to route multiple packets in a single clock cycle when possible.

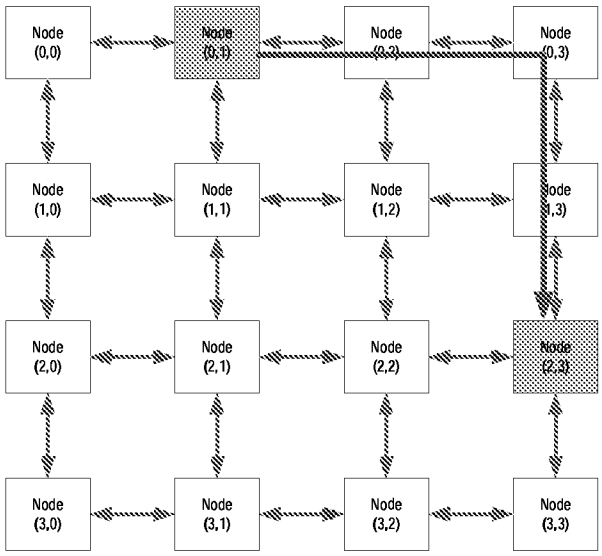


Figure 6.4 Example Message Network Packet Routing

- Figure 6.4 illustrates the routing of a message network packet from node (1,0) to node (3,2).
- Deadlock is avoided in the message network by guaranteeing that routed packets may always be accepted by the destination. This ensures that no dependencies exist.
- Message network packets that traverse the same route are delivered in order (i.e., a later packet is never allowed to bypass an earlier packet). Message network packets that traverse different routes may be delivered in any order.

6.2 Message Network Packets and Messages

- The message network routes message network packets between nodes. These packets are used to construct messages.

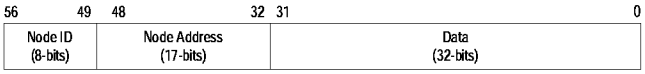


Figure 6.5 Message Network Packet Format

- The format of a message network packet is shown in Figure 6.5 and contains three fields that together occupy 57-bits.
  - The node ID is described in Section 6.1. It uniquely identifies a node, specifies the node's location (i.e., row and column) in the message network 2D mesh, and is used to route message network packets.

- The node address specifies the DWord address within the destination node where data contents of the packet should be written. The 17-bit DWord node address field allows a node to expose up to 512KB of address space on the message network.
- The data field contains 32-bits of data used to transfer messages.

- Message network packets received at the ultimate destination node are processed by writing the data field value contained in the packet to the node address contained in the packet.
  - There are no byte enables in message network packets so the entire 32-bits of the data field are always written to the node address.
  - The node address may specify the address of a memory or memory mapped peripheral (e.g., FIFO) within the destination node.

6.3 Message Queues

- Message queues build on top of message network packets to provide unidirectional message communication between nodes.
  - The originator of a message (i.e., the sender of a message) is referred to as the source node. The destination of a message (i.e., the receiver of a message) is referred to as the destination node.
  - A message queue consists of an outbound message queue and an inbound message queue. The outbound message queue is located in the source node while the inbound message queue is located in the destination node.
  - A message queue may be used to transfer messages from one or more source nodes to a single destination node (i.e., one to one or many to one). A single message queue is not intended to be used to send messages to multiple destination nodes (i.e., one to many or many to many).
  - Bidirectional message communication between nodes is accomplished through the use of two message queues (i.e., one in each direction).

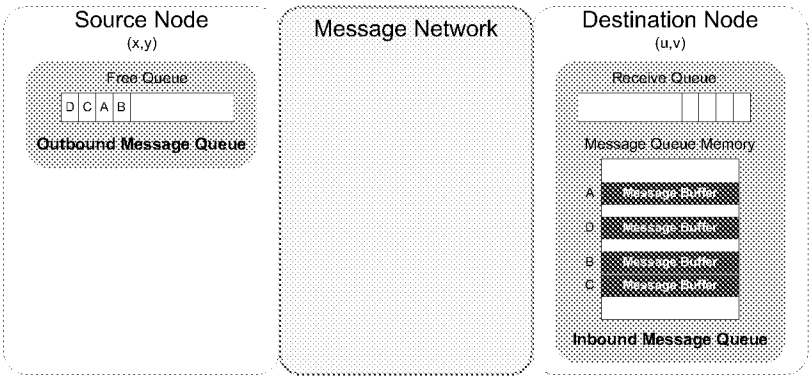


Figure 6.6 Conceptual Model of a Message Queue

- A conceptual model of a message queue with a single source node is shown in Figure 6.6.
  - The destination node contains the inbound message queue. The inbound message queue consists of message queue memory and a receive queue.
    - Message queue memory is used to store received messages and is addressable via the node address.

        - Message queue memory may be implemented as a special memory associated with one message queue, as a memory shared among all inbound message queues associated with a node, or as a general purpose memory in the node.
    - The receive queue is used to store message buffer pointers associated with received messages that have been posted (i.e., written to message queue memory and are available for processing by the destination node).
  - The source node contains the outbound message queue. The outbound message queue consists simply of a free queue.
    - The free queue is used to store pointers to message queue memory associated with the inbound message queue (i.e., message buffer pointers). These free buffers may be used to send messages.
  - During system initialization a mapping is created between an outbound message queue in a source node and the inbound message queue in a destination node. This mapping defines the message queue (i.e., one or more outbound message queues mapped to a single inbound message queue). The manner in which this mapping is configured is node specific.
- Figure 6.6 shows a message queue that supports up to four outstanding messages.
  - Each possible outstanding message has a preallocated fixed sized buffer referred to as a message buffer. The message buffer size must be equal to the maximum sized message associated with that message queue.
  - Each DWord of a message buffer has a unique node address. The starting address of a message buffer is called the message buffer's address. The addresses for the four message buffers shown in Figure 6.6 are A, B, C and D.
    - Message buffer addresses need not be contiguous or sequential in message queue memory.



Figure 6.7 Message Buffer Pointer Format

- The format of a message buffer pointer is shown in Figure 6.7.
  - The node address is a 19-bit byte address that points to a message buffer.
    - For message buffer pointers in the receive queue, this field points to the message buffer in the destination node's message queue memory that contains the posted message.
    - For message buffer pointers in the free queue, this field points to a message buffer in the destination node's message queue memory that is free and may be used to send a new message.
  - The queue ID field specifies a message queue ID.
    - For message buffer pointers in the receive queue (i.e., posted messages), this field specifies the outbound message queue ID in the source node that was used to send the message. This field may be used by the destination node to free the message buffer (i.e., enqueue the message buffer pointer in the free queue of the outbound message queue).
      - The free queue associated with an outbound message queue has a fixed node address that depends on the queue ID.

- For message buffer pointers in the free queue, this field is not used and its contents are undefined.

° The node ID specifies a message network node ID.

- For message buffer pointers in the receive queue (i.e., posted messages), this field specifies the node ID of the source node that contains the outbound message queue that was used to send the message. This field may be used by the destination node to free the message (i.e., enqueue the message buffer pointer in the free queue of the outbound message queue).
- For message buffer pointers in the free queue, this field is not used and its contents are undefined.

• A source node (outbound message queue) sends a message to a destination node (inbound message queue) by performing the following operations.

° Obtain a pointer to a free message buffer by dequeueing a message buffer pointer from the free queue.

° Writing the message contents to the free message buffer by transmitting a series of message network packets. The node address of these packets points to the destination node address in the message buffer while the data value contains the message data.

° Posting the message by generating a message network packet that writes a message buffer pointer to the receive queue of the inbound message queue.

• A destination node (inbound message queue) determines that a message has been received by the fact that its receive queue is non-empty.

° The destination node processes the received message by dequeueing the message buffer pointer from the receive queue. This pointer tells the destination node where in message queue memory the message is located.

- In addition to pointing to the message buffer containing the message, the message buffer pointer contains the node ID and outbound message queue ID that was used to send the message. This information may be used by the destination node to free the message buffer once processing of the message has completed.

° When the destination node completes processing a message, it frees the message buffer associated with the message by generating a message network packet that writes a message buffer pointer to the free queue of the outbound message queue.

- The message buffer pointer in the free queue points to the free message buffer in the destination node.

• During initialization the free queue in all outbound message queues is initialized with message buffer pointers to message buffers that may be used to transmit messages.

° The free queue in the example shown in Figure 6.6 is initialized with D, C, A and B (the node ID and queue ID fields are not used in a free queue and are not shown in the figure).

° The order in which message buffer pointers are enqueued in the free queue determines the order in they will be used. In this example D will be used first, followed by C and so on.

Packet Number	Node ID	Node Address	Data
1	(u,v)	D	DWord 0
2	(u,v)	D + 1	DWord 1

Table 6.1 Sending a Three DWord Message

Packet Number	Node ID	Node Address	Data
3	(u,v)	D + 2	DWord 2
4	(u,v)	Receive Queue	{(x,y), QID, D}

Table 6.1 Sending a Three DWord Message

- Table 6.1 shows the packets generated by an outbound message queue to send a three DWord message.
  - The first three packets are used to transfer message data while the last packet is used to transfer the message buffer pointer (i.e., post the message).
    - The node ID in all message network packets is the node ID of the destination node (i.e., (u,v)).
    - The address of the first packet is the message buffer address (i.e., D). DWord zero of the message is written to this address. Subsequent DWords of the message are written to subsequent addresses.
  - The final packet contains a destination node address that corresponds to the receive queue of the inbound message queue and a data value that corresponds to the message buffer pointer. This packet is used to post the message.
    - The node address in the message buffer pointer points to D (i.e., the message buffer used for the message).
    - The queue ID in the message buffer pointer contains the ID of the outbound message queue (i.e., in this case it is QID).
    - The node ID in the message buffer pointer contains the ID of the source node that contains the outbound message queue (i.e., in this case (x,y)).

Packet Number	Node ID	Node Address	Data
1	(x,y)	QID Node Address	D

Table 6.2 Freeing a Message Buffer

- Table 6.2 shows the packet generated by the destination node when processing of the received message has completed (i.e., when the message buffer is freed).
  - The node ID in this packet corresponds to the node ID of the source node (i.e., (x,y)).
  - The address contains the address of the free queue associated with the outbound message queue (i.e., QID node address).
  - The data field contains the message buffer pointer. Only the node address field is used in the message buffer pointer and this address points to the message buffer being freed (i.e., D).

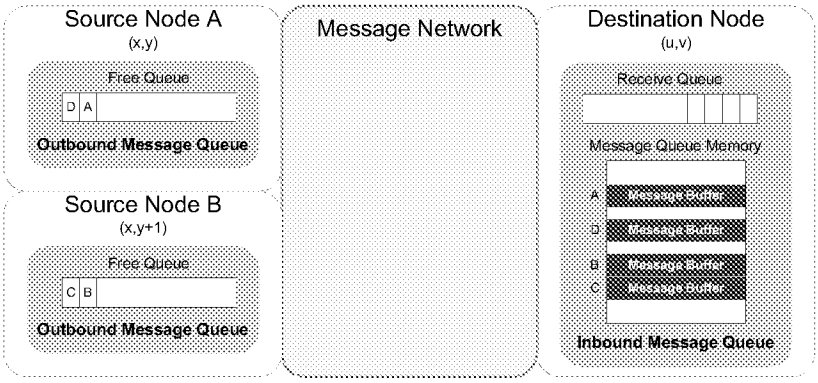


Figure 6.8 Conceptual Model of a Message Queue with Two Source Nodes

- Table 6.8 illustrates a message queue with two source nodes.
  - Both source node A and B can send messages to the destination node.
  - The message buffers used by source nodes A and B are preallocated. Source node A is allocated two buffers and can have at most two outstanding messages regardless of the state of source node B.
  - Messages posted by both source nodes A and B are enqueued in the receive queue. Therefore, the receive queue is appropriate for applications where head of line blocking of messages is not an issue (e.g., egress DMA operations).



Buffer RAM

7.1 Introduction

- The buffer RAM implements 1Mb (i.e., 128KB) of high speed SRAM.

Host Interface

8.1 Introduction

Tag Bit(s)	Function
7:5	Unused (must be zero)
4	Determines Requester 0 = Ingress DMA 1 = NVMHCI Submission Queue Manager
3:0	Request ID (supports up to 16 outstanding requests per requester)

Table 8.1 PCIe Non Posted Request Tag Allocation

8.2 Message Network Interface

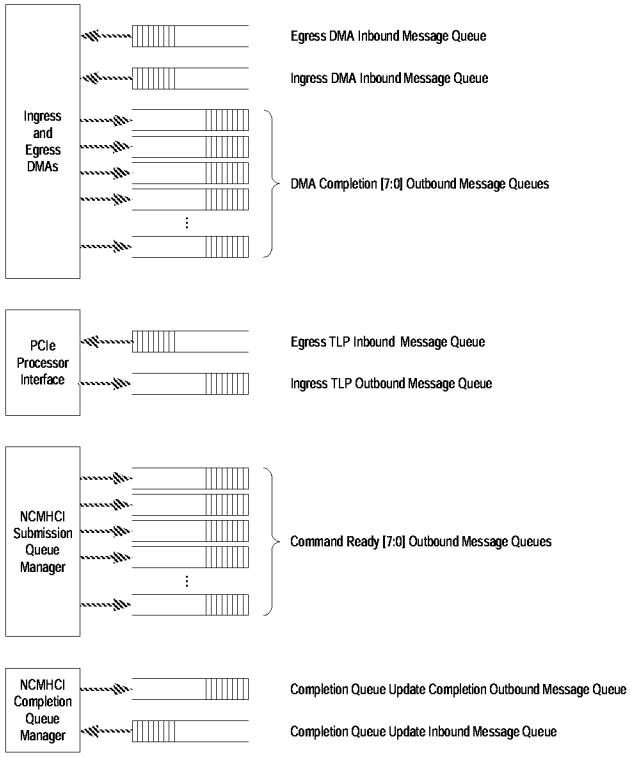


Figure 8.1 Host Interface Inbound and Outbound Message Queues

Queue ID	Function	Name	Queue Depth[1]
0	PCIe Proc. Interface	Ingress TLP Outbound Message Queue	8

Table 8.2 Host Interface Outbound Message Queues

Queue ID	Function	Name	Queue Depth[1]
1	NVMHVI Completion Queue Manager	Completion Queue Update Completion Outbound Message Queue	8
8	Ingress and Egress DMAs	DMA Completion 0 Outbound Message Queue	8
9		DMA Completion 1 Outbound Message Queue	8
10		DMA Completion 2 Outbound Message Queue	8
11		DMA Completion 3 Outbound Message Queue	8
12		DMA Completion 4 Outbound Message Queue	8
13		DMA Completion 5 Outbound Message Queue	8
14		DMA Completion 6 Outbound Message Queue	8
15		DMA Completion 7 Outbound Message Queue	8
16	NVMHVI Submission Queue Manager	Command Ready 0 Outbound Message Queue	8
17		Command Ready 1 Outbound Message Queue	8
18		Command Ready 2 Outbound Message Queue	8
19		Command Ready 3 Outbound Message Queue	8
20		Command Ready 4 Outbound Message Queue	8
21		Command Ready 5 Outbound Message Queue	8
22		Command Ready 6 Outbound Message Queue	8
23		Command Ready 7 Outbound Message Queue	8

Table 8.2 Host Interface Outbound Message Queues

1. This corresponds to the maximum number of supported entries in the free queue

Queue ID	Function	Name	Queue Depth[1]
0	PCIe Proc. Interface	Egress TLP Inbound Message Queue	16
1	NVMHVI Completion Queue Manager	Completion Queue Update Inbound Message Queue	8
2	Ingress and Egress DMAs	Ingress DMA Inbound Message Queue	64
3		Egress DMA Inbound Message Queue	64

Table 8.3 Host Interface Inbound Message Queues

1. This corresponds to the maximum number of supported entries in the receive queue

- Figure 8.2 shows the node address map for the host interface. Associated with each of the four inbound message queues is a dedicated 2KB message queue memory. Messages buffers for an inbound message queue must be allocated from its specific message queue memory. Allocating a message buffer from a different memory produces undefined results.

- Send the CplQueueUpdateCpl to the same Node ID that sent the CplQueueUpdate message. The Receive Queue ID should be the same as the Transmit Queue ID.

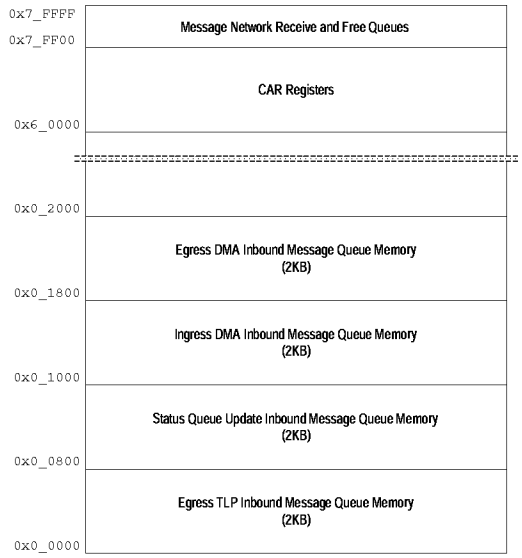


Figure 8.2 Host Interface Node Address Map

8.3 Ingress DMA

8.4 Egress DMA

8.5 PCIe Processor Interface

•

8.6 NVMHCI Submission Queue Manager

- Delaware supports 33 submission queue engines.
  - ° Each submission queue is managed by a submission queue engine
  - ° The 33 submission queue engines along with shared logic and buffers form the NVMHCVI submission queue manager.

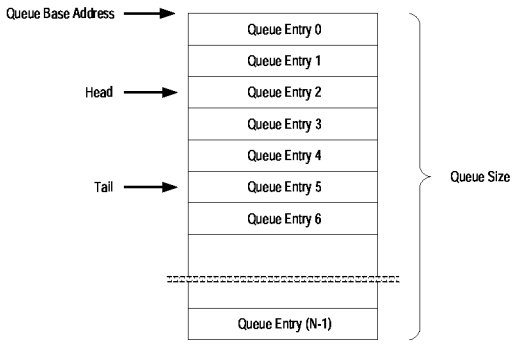


Figure 8.3 Submission Queue

- Figure 8.3 illustrates an NVMCHI submission queue.

- A submission queue is a FIFO of queue entries implemented as a circular buffer in PCIe memory. Each entry of a submission queue corresponds to a 64B NVMHCI command.
- The 64-bit PCIe base address of the submission queue is specified by the corresponding Submission Queue Engine Lower and Upper Base Address (SQExLBASE and SQExUBASE) registers.
- The number of entries in a submission queue (i.e., its size) is determined of the Size (SIZE) field in the Submission Queue Engine Control (SQExCTL) register.
  - The submission queue occupies the PCIe address range from the base address SQEx[U:L]BASE to SQEx[U:L]BASE + CmdSize * SQExCTL.SIZE - 1 where CmdSize corresponds to the size of an NVMHCI command in bytes (i.e., 64B)
- Associated with the submission queue is a head and a tail pointer.
  - The head pointer points to the entry at the head of the FIFO when the FIFO is non-empty (i.e., the oldest enqueued entry).
  - The tail pointer points to the entry following the tail of the FIFO when the FIFO is non-empty (i.e., location where the next entry will be written).
  - The queue is empty when the head pointer equals the tail pointer.
  - The queue is full when the head pointer equals the tail pointer plus one modulo the queue size

- Submission queue alignment rules.
  - A submission queue must start on a 4KB aligned boundary in PCIe memory. It may be completely located below the 4GB boundary, above the 4GB boundary, or may straddle the 4GB boundary.
  - All submission queue entries are 64B in size (i.e., one NVMHCI command).
  - Submission queue entries are always 64B aligned and hence never cross a 4KB boundary.
- A software driver enqueues entries in the submission queue (i.e., manages the tail pointer) while the submission queue engine dequeues entries from the submission queue (i.e., manages the head pointer).
  - A software driver enqueues an entry in the submission queue by performing the following actions.
    - Writing an NVMHCI command into the PCIe memory corresponding to the queue entry pointed to by the tail pointer.
    - Incrementing the tail pointer modulo the queue size (i.e., When the tail pointer goes past the last entry in the queue, it is reset to point to the start of the queue).
    - Communicating the new tail pointer to the submission queue engine by writing its value to the corresponding Submission Queue [16:0] Tail Doorbell (SQxTDBL) register in PCIe BAR0 memory space.
      - SQxTDBL is implemented in hardware by the host interface.
  - A software driver may enqueue multiple entries at once by incrementing the tail pointer by a value greater than one.
  - The submission queue engine manages the head pointer in hardware.

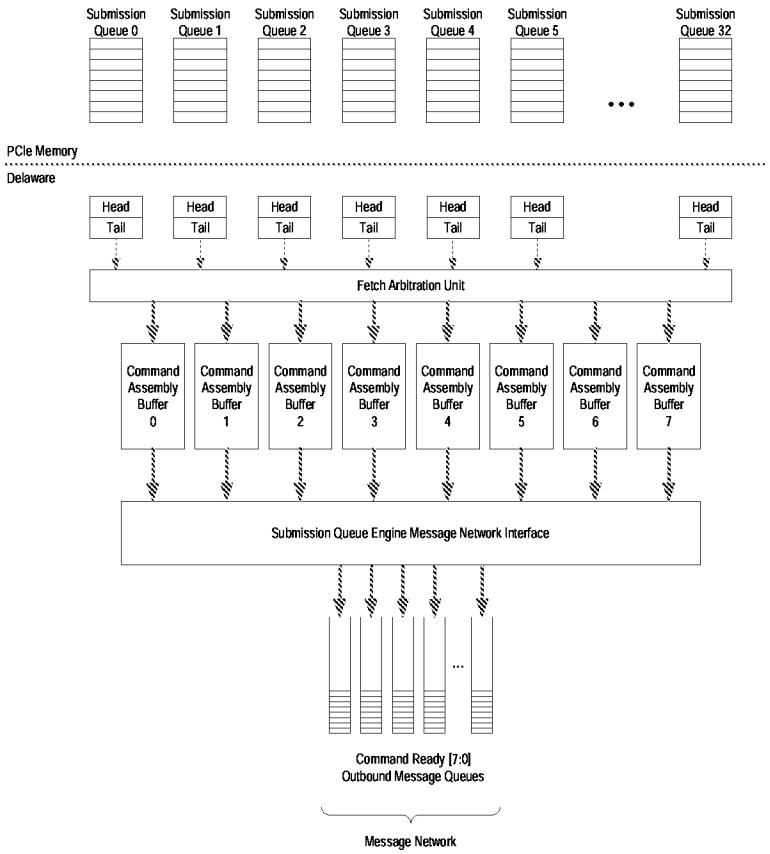


Figure 8.4 NVMHCI Submission Queue Manager

- Figure 8.4 shows a block diagram of the NVMHCI submission queue manger.
- Associated with each submission queue engine are hardware registers containing the head and tail pointers associated with that submission queue.
  - The head pointer is managed by the submission queue engine.
  - The tail pointer contains the last value written to the SQxTDBL.
- A submission queue is enabled by setting the Enable (EN) bit in the Submission Queue Engine Control (SQExCTL) register.
  - When a submission queue engine is disable, it does not fetch commands from the submission queue.
  - When a submission queue engine is enabled (i.e., the EN transition from a zero to a one), the head and tail pointers are reset to zero.
  - When a submission queue engine is disabled after being enabled, the submission queue engine stops fetching new commands from the submission queue (i.e., issuing PCIe memory read requests). The EN bit remains set until all PCIe requests associated with the submission queue engine have completed and all CmdReady messages for outstanding commands have been sent on the message network. The head and tail pointers remain unmodified.

- The number of commands in a submission queue (i.e., submission queue entries) that are ready to be processed may be computed as shown below. When numEntries is one or greater, the submission queue has a command ready to be processed.

```
if (head < tail) {
    numEntries = tail - head
} else {
    numEntries = tail - head + queueSize
}
```

- The NVMHCI submission queue manager contains eight command assembly buffers. The role of the fetch arbitration unit is to map submission queue engines with ready commands to command assembly buffers.
  - ° When a submission queue engine detects that a command is ready to be processed, in presents a request to the fetch arbitration unit.
  - ° The fetch arbitration unit selects a ready submission queue and maps its to a free command assembly buffer. This arbitration is performed in a fair manner (i.e., a ready submission queue engine is not selected again until all other ready submission queue engines have been serviced).
- Once a submission queue engine has been mapped to a command assembly buffer, a PCIe read request is issued to read the one command pointed to by the head of that submission queue. The eight command assembly buffers allow up to eight commands to be transferred in parallel and these commands may be associated with the same or different submission queues.
  - ° The PCIe header fields of the read request have the following values.
    - Requester ID: value in Requester ID (RID) field in the PCIe Control (PCIECTL) register.
    - Tag: A tag value is statically assigned to each command assembly buffer and its value is equal to the number of the command assembly buffer ORed with 0x10 (i.e., buffer 0 uses Tag 0x10, buffer 1 uses Tag 0x11, and so on).
    - Length: 16 (corresponds to 64B - the size of a command).
    - 1st DW BE and Last DW BE: 0xF
    - AT: zero
    - TC: zero
    - Attr: 0x2 (relaxed ordering and snoop caches)
      - Bits are required in the ASIC to control these attributes, but this should be fine for the FPGA.
    - EP: 0 (not poisoned)
    - TD: 0 (no ECRC)
    - Address: starting address of the submission queue entry containing the command to be fetched
      - If the upper 32-bits of the PCIe address are zero, then a 3 DWord PCIe header is used.
      - If the upper 32-bits of the PCIe address are non-zero, then a 4 DWord PCIe header is used.
- Once all of the completions associated with a command fetch have been received, the head value for that submission queue engine is incremented.

- The current head value associated with a submission queue is reported in completion queue entries; therefore, the head value must not be updated until all of the data associated with the command has been read from memory. Premature updating of the head value results in a race condition that may cause a new command to overwrite command being transferred before it is read from memory.
- A submission queue engine is able to simultaneously utilize multiple command assembly buffers to transfer multiple ready commands in parallel.

- The function of the submission queue engine message network interface it to transfer complete commands from the command assembly buffer to a processor using the message network.
  - When a complete command is available in a command assembly buffer, the command is transferred on the message network using one of the eight command ready outbound message queues.
  - Outbound message queues are allocated in a round robin manner with full queues and queues that are not enabled being skipped.
  - The format of the message is described in Section 14.2, "CmdReady" on page 14-1.
    - The first DWord contains the ID of the submission queue engine from which the command was fetched.
    - The subsequent 16 DWords contain the contents of the actual command.

8.7 NVMHCI Completion Queue Manager

- Delaware supports 33 completion queues.
  - Multiple submission queues may be mapped to a single completion queue.
  - Each completion queue is managed by a completion queue engine.
  - The 33 completion queue engines along with shared logic and buffers form the NVMHCVI completion queue manager.

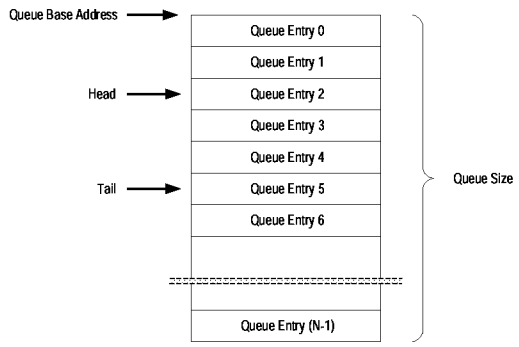


Figure 8.5 Completion Queue

- Figure 8.3 illustrates an NVMCHI completion queue. A completion queue is similar to a submission queue except that Delaware enqueues entries while the software driver dequeues entries.
  - A completion queue is a FIFO of queue entries implemented as a circular buffer in PCIe memory. Each entry of a completion queue corresponds to an 16B NVMHCI completion queue entry.

- The 64-bit PCIe base address of the completion queue is specified by the corresponding Completion Queue Engine Lower and Upper Base Address (CQExLBASE and CQExUBASE) registers.
- The number of entries in a completion queue (i.e., its size) is determined of the Size (SIZE) field in the Completion Queue Engine Control (CQExCTL) register.
  - The completion queue occupies the PCIe address range from the base address CQEx[U:L]BASE to CQEx[U:L]BASE + CmpEntry * CQExCTL.SIZE - 1 where CmpEntry corresponds to the size of an NVMHCI completion queue entry in bytes (i.e., 16B)
- Associated with the completion queue is a head and a tail pointer.
  - The head pointer points to the entry at the head of the FIFO when the FIFO is non-empty (i.e., the oldest enqueued entry).
  - The tail pointer points to the entry following the tail of the FIFO when the FIFO is non-empty (i.e., location where the next entry will be written).
  - The queue is empty when the head pointer equals the tail pointer.
  - The queue is full when the head pointer equals the tail pointer plus one modulo the queue size

- Completion queue alignment rules.
  - A completion queue must start on a 4KB aligned boundary in PCIe memory. It may be completely located below the 4GB boundary, above the 4GB boundary, or may straddle the 4GB boundary.
  - All completion queue entries are 16B in size (i.e., one NVMHCI completion entry).
  - Completion queue entries are always 16B aligned and hence never cross a 4KB boundary.

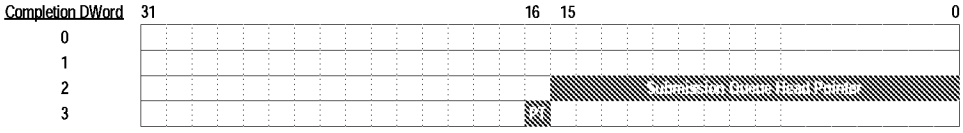


Figure 8.6 Completion Format

- The format of a completion in a completion queue is shown in Figure 8.6.
  - Completion DWord 0 corresponds to the 32-bit value that is written in the first DWord PCIe address of the completion entry.
  - Completion DWord 3 corresponds to the 32-bit value that is written to the last DWord PCIe address of the completion entry.
  - Light blue fields correspond to fields managed by Delaware firmware. See the NVMHCI specification for a definition of these fields.
  - Dark blue fields correspond to fields that are managed by the NVMHCI completion queue manager as described later in this section.

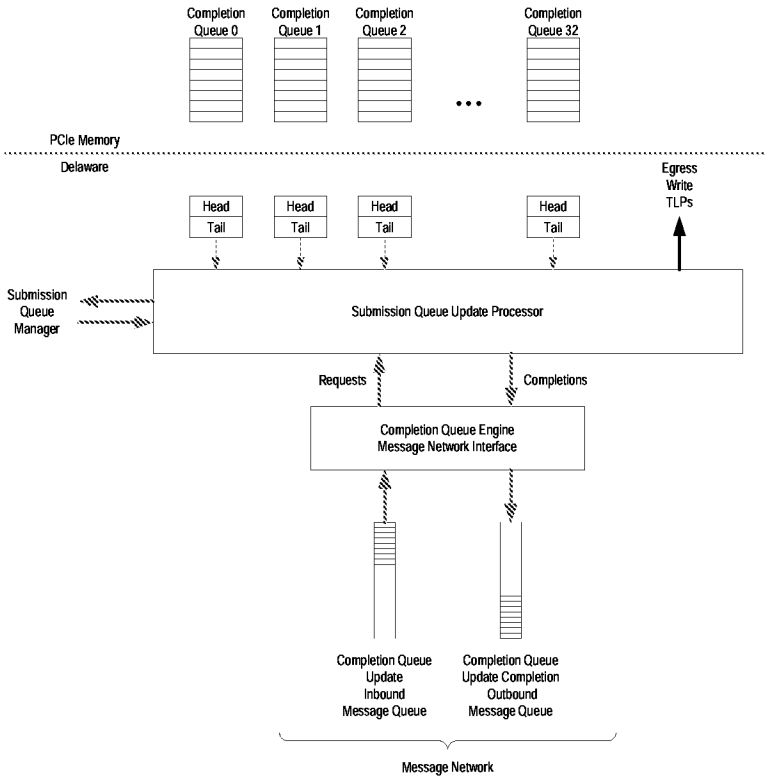


Figure 8.7 NVMHCI Completion Queue Manager

- Figure 8.7 shows a block diagram of the NVMHCI completion queue manger. The function of the NVMHCI completion queue manger is to receive Completion Queue Update (CplQueueUpdate) request messages on the message network and enqueue the completion contained in the message in the corresponding completion queue in PCIe memory.
- Associated with each completion queue engine are hardware registers containing the head and tail pointers associated with that completion queue and a phase tag state.
  - The tail pointer is managed by the completion queue engine.
  - The head pointer contains the last value written to the CQxHDBL and is managed by a software driver.
  - The phase tag allows the driver to easily determine which completion entries were updated in the completion queue.
    - The phase tag state in the completion queue manager is initialized to one.
    - The driver initializes the phase tag in all completion entries to zero.
    - When the completion queue manager writes a completion entry, the phase tag is set to one (if this feature is enabled in hardware).
    - The state of the phase tag in the completion queue manager is inverted each time the tail pointer wraps (i.e., the tail pointer moves from the last queue entry to the first queue entry).
- A completion engine is enabled by setting the Enable (EN) bit in the Completion Queue Engine Control (SQExCTL) register.
  - When a completion queue engine is enabled (i.e., the state of the EN bit transitions from a zero to a one), the following actions are performed.
    - The value of the head pointer is reset to zero.

- The value of the tail pointer is reset to zero.
- The phase tag state is set to one.

- The number of full completion queue entries in a completion queue may be computed as shown below. When numEntries = (CQExCTL.SIZE - 1) where CQExCTL.SIZE is the size field of the completion queue (i.e., number of entries in the queue), then the queue is full.
  - The completion queue full condition is head = (tail + 1) mod CQExCTL.size. Therefore, the maximum number of full completion queue entries is one less than the size of the queue.

```
if (head < tail) {
    numEntries = tail - head
} else {
    numEntries = tail - head + queueSize
}
```

- The NVMHCI completion queue manger receives and processes CplQueueUpdate messages received on the completion queue update inbound message queue. The message is dequeued from the head of the inbound message queue and is passed to the completion queue update processor. Each message contains the following information.
  - The completion queue engine ID with which the update is associated.
  - The two DWord value of the completion that should be enqueued in the completion queue.
- The submission queue update processor performs the following operations.
  - If the completion queue engine specified by the Completion Queue Engine ID (CQE) field in the message is disabled, then a completion message (i.e., CmpQueueUpdateCpl) is generated with the status disabled (i.e., the Status field in the CmpQueueUpdateCpl message set to disabled). No further processing is performed. The message is transmitted on the message network using the outbound message queue.
  - The status of the completion queue engine specified by the Completion Queue Engine ID (CQE) field in the message is checked to determine if it is full. If the corresponding completion queue is full, a completion message (i.e., CmpQueueUpdateCpl) is generated with the status full (i.e., the Status field in the CmpQueueUpdateCpl message set to full). No further processing is performed. The message is transmitted on the message network using the outbound message queue.
  - The Insert Submission Queue Head Value (InsertHead) bit is examined in the message. If the bit is set, then the contents of bits zero through 15 in the Completion DWord 2 (CmpDWord2) field are replaced with the current head value associated with the submission queue engine specified by the Submission Queue Engine ID (SQE) field (see Figure 8.6).
  - The Insert Phase Tag (InsertPhaseTag) bit is examined in the message. If the bit is set, then the current state of the phase tag associated with the submission queue engine specified by the Submission Queue Engine ID (SQE) field is inserted into DWord 3 (CmpDWord3) field bit 16 (see Figure 8.6).
  - A PCIe write request is generated to write the contents of the Completion DWord (CmpDWord0 through CmpDWord3) fields to the completion queue entry pointed to by the tail.
    - The PCIe header fields of the write request have the following values.
      - Requester ID: value in Requester ID (RID) field in the PCIe Control (PCIECTL) register.
      - Tag: 0

- Length: 4 (corresponds to 16B - the size of a completion queue entry).
- 1st DW BE and Last DW BE: 0xF
- AT: zero
- TC: zero
- Attr: 0x0 (no relaxed ordering and snoop caches)
- EP: 0 (not poisoned)
- TD: 0 (no ECRC)
- Address: starting address of the completion queue entry
  - If the upper 32-bits of the PCIe address are zero, then a 3 DWord PCIe header is used.
  - If the upper 32-bits of the PCIe address are non-zero, then a 4 DWord PCIe header is used.

° The tail pointer in incremented modulo the size of the completion queue. If the tail pointer wraps from the last entry to the first entry, then the phase tag bit associated with the completion queue engine is inverted.

° A completion message (i.e., CmpQueueUpdateCpl) is generated with the status success (i.e., the Status field in the CmpQueueUpdateCpl message set to success). No further processing is performed. The message is transmitted on the message network using the outbound message queue.

- If the completion queue manager is unable to transmit the completion message (i.e., CmpQueueUpdateCpl) due to the free queue associated with the outbound message queue being empty, then processing stalls.
- This message is generated only after the PCIe submission queue write request has been transmitted or enqueued in the PCIe interface such that there is no possibility that a subsequent TLP from an Egress TLP (EgressTLP) message request may pass it.

• The following malformed request checks are performed on received CmpQueueUpdate messages. If a malformed check is satisfied, then the message is malformed. The message is not sent to the submission queue update processor and a completion message (i.e., CmpQueueUpdateCpl) is generated with the status malformed (i.e., the Status field in the CmpQueueUpdateCpl message set to malformed).

° The Completion Queue Engine (CQE) field in the CplQueueUpdate message targets a disabled completion queue engine (i.e., a completion queue engine whose EN bit is zero).

° The Completion Queue Engine (CQE) field in the CplQueueUpdate message targets an unimplemented completion queue engine (i.e., a completion queue engine whose ID is greater than 32).

8.8 PCIe Registers

8.8.1 PCIe Configuration Space

8.8.2 PCIe BAR 0

SQxTDBL - Submission Queue [16:0] Tail Doorbell

Bit Field	Field Name	Type	Default Value	Description
14:0	SQT	RW	0x0	**Submission Queue Tail** Writing to this register indicates to the corresponding submission queue engine that the tail pointer has been updated and the value of the new tail pointer. When read this register returns the previous value written.
15	Reserved	RO	0x0	

CQxHDBL - Completion Queue [16:0] Head Doorbell

Bit Field	Field Name	Type	Default Value	Description
14:0	CQH	RW	0x0	**Completion Queue Head** Writing to this register indicates to the corresponding completion queue engine that the head pointer has been updated and the value of the new head pointer. When read this register returns the previous value written.
15	Reserved	RO	0x0	

8.9 CAR Registers

- Table 8.4 shows the host interface node address map. The CAR address is formed using the node address as described in Section 4.2.3, "CAR Address Space" on page 4-4.

Node Address	Register Mnemonic	Register Description
0x6_0000	PCIECTL	"PCIECTL - PCI Express Control" on page 16
0x6_0004	PCIEERR	"PCIEERR - PCI Express Error Status" on page 17
0x6_0008	MEMLBAR	"MEMLBAR - Memory Lower Base Address" on page 17
0x6_000C	MEMUBAR	"MEMUBAR - Memory Upper Base Address" on page 18
0x6_0200	ITLPQCTL	"ITLPQCTL - Ingress TLP Outbound Message Queue Control" on page 20
0x6_0220	DMACQCTL0	"DMACQCTL[7:0] - DMA Completion Outbound Message Queue Control [7:0]" on page 20
0x6_0224	DMACQCTL1	"DMACQCTL[7:0] - DMA Completion Outbound Message Queue Control [7:0]" on page 20
0x6_0228	DMACQCTL2	"DMACQCTL[7:0] - DMA Completion Outbound Message Queue Control [7:0]" on page 20
0x6_022C	DMACQCTL3	"DMACQCTL[7:0] - DMA Completion Outbound Message Queue Control [7:0]" on page 20
0x6_0230	DMACQCTL4	"DMACQCTL[7:0] - DMA Completion Outbound Message Queue Control [7:0]" on page 20

Table 8.4 Host Interface CSR Address Map

Node Address	Register Mnemonic	Register Description
0x6_0234	DMACQCTL5	"DMACQCTL[7:0] - DMA Completion Outbound Message Queue Control [7:0]" on page 20
0x6_0238	DMACQCTL6	"DMACQCTL[7:0] - DMA Completion Outbound Message Queue Control [7:0]" on page 20
0x6_023C	DMACQCTL7	"DMACQCTL[7:0] - DMA Completion Outbound Message Queue Control [7:0]" on page 20
0x6_0240	CMDRQCTL0	"CMDRQCTL[7:0] - Command Ready Outbound Message Queue Control [7:0]" on page 21
0x6_0244	CMDRQCTL1	"CMDRQCTL[7:0] - Command Ready Outbound Message Queue Control [7:0]" on page 21
0x6_0248	CMDRQCTL2	"CMDRQCTL[7:0] - Command Ready Outbound Message Queue Control [7:0]" on page 21
0x6_024C	CMDRQCTL3	"CMDRQCTL[7:0] - Command Ready Outbound Message Queue Control [7:0]" on page 21
0x6_0250	CMDRQCTL4	"CMDRQCTL[7:0] - Command Ready Outbound Message Queue Control [7:0]" on page 21
0x6_0254	CMDRQCTL5	"CMDRQCTL[7:0] - Command Ready Outbound Message Queue Control [7:0]" on page 21
0x6_0258	CMDRQCTL6	"CMDRQCTL[7:0] - Command Ready Outbound Message Queue Control [7:0]" on page 21
0x6_025C	CMDRQCTL7	"CMDRQCTL[7:0] - Command Ready Outbound Message Queue Control [7:0]" on page 21
0x6_1000	SQE0CTL	"SQE[16:0]CTL - Submission Queue Engine 16:0 Control" on page 18
0x6_1008	SQE0LBASE	"SQE[16:0]LBASE - Submission Queue Engine 16:0 Lower Base Address" on page 18
0x6_100C	SQE0UBASE	"SQE[16:0]UBASE - Submission Queue Engine 16:0 Upper Base Address" on page 18
0x6_1010	SQE0TDBL	"SQE[16:0]TDBL - Submission Queue [16:0] Tail Doorbell" on page 19
0x6_1020	SQE1CTL	"SQE[16:0]CTL - Submission Queue Engine 16:0 Control" on page 18
0x6_1028	SQE1LBASE	"SQE[16:0]LBASE - Submission Queue Engine 16:0 Lower Base Address" on page 18
0x6_102C	SQE1UBASE	"SQE[16:0]UBASE - Submission Queue Engine 16:0 Upper Base Address" on page 18
0x6_1030	SQE1TDBL	"SQE[16:0]TDBL - Submission Queue [16:0] Tail Doorbell" on page 19
0x6_1040	SQE2CTL	"SQE[16:0]CTL - Submission Queue Engine 16:0 Control" on page 18
0x6_1048	SQE2LBASE	"SQE[16:0]LBASE - Submission Queue Engine 16:0 Lower Base Address" on page 18
0x6_104C	SQE2UBASE	"SQE[16:0]UBASE - Submission Queue Engine 16:0 Upper Base Address" on page 18
0x6_1050	SQE2TDBL	"SQE[16:0]TDBL - Submission Queue [16:0] Tail Doorbell" on page 19
0x6_1060	SQE3CTL	"SQE[16:0]CTL - Submission Queue Engine 16:0 Control" on page 18
0x6_1068	SQE3LBASE	"SQE[16:0]LBASE - Submission Queue Engine 16:0 Lower Base Address" on page 18
0x6_106C	SQE3UBASE	"SQE[16:0]UBASE - Submission Queue Engine 16:0 Upper Base Address" on page 18
0x6_1070	SQE3TDBL	"SQE[16:0]TDBL - Submission Queue [16:0] Tail Doorbell" on page 19
0x6_1080	SQE4CTL	"SQE[16:0]CTL - Submission Queue Engine 16:0 Control" on page 18
0x6_1088	SQE4LBASE	"SQE[16:0]LBASE - Submission Queue Engine 16:0 Lower Base Address" on page 18
0x6_108C	SQE4UBASE	"SQE[16:0]UBASE - Submission Queue Engine 16:0 Upper Base Address" on page 18
0x6_1090	SQE4TDBL	"SQE[16:0]TDBL - Submission Queue [16:0] Tail Doorbell" on page 19
0x6_10A0	SQE5CTL	"SQE[16:0]CTL - Submission Queue Engine 16:0 Control" on page 18
0x6_10A8	SQE5LBASE	"SQE[16:0]LBASE - Submission Queue Engine 16:0 Lower Base Address" on page 18
0x6_10AC	SQE5UBASE	"SQE[16:0]UBASE - Submission Queue Engine 16:0 Upper Base Address" on page 18
0x6_10B0	SQE5TDBL	"SQE[16:0]TDBL - Submission Queue [16:0] Tail Doorbell" on page 19
0x6_10C0	SQE6CTL	"SQE[16:0]CTL - Submission Queue Engine 16:0 Control" on page 18
0x6_10C8	SQE6LBASE	"SQE[16:0]LBASE - Submission Queue Engine 16:0 Lower Base Address" on page 18
0x6_10CC	SQE6UBASE	"SQE[16:0]UBASE - Submission Queue Engine 16:0 Upper Base Address" on page 18
0x6_10D0	SQE6TDBL	"SQE[16:0]TDBL - Submission Queue [16:0] Tail Doorbell" on page 19
0x6_10E0	SQE7CTL	"SQE[16:0]CTL - Submission Queue Engine 16:0 Control" on page 18
0x6_10E8	SQE7LBASE	"SQE[16:0]LBASE - Submission Queue Engine 16:0 Lower Base Address" on page 18
0x6_10EC	SQE7UBASE	"SQE[16:0]UBASE - Submission Queue Engine 16:0 Upper Base Address" on page 18
0x6_10F0	SQE7TDBL	"SQE[16:0]TDBL - Submission Queue [16:0] Tail Doorbell" on page 19
0x6_1100	SQE8CTL	"SQE[16:0]CTL - Submission Queue Engine 16:0 Control" on page 18

Table 8.4 Host Interface CSR Address Map

Node Address	Register Mnemonic	Register Description
0x6_1108	SQE8LBASE	"SQE[16:0]LBASE - Submission Queue Engine 16:0 Lower Base Address" on page 18
0x6_110C	SQE8UBASE	"SQE[16:0]UBASE - Submission Queue Engine 16:0 Upper Base Address" on page 18
0x6_1110	SQE8TDBL	"SQE[16:0]TDBL - Submission Queue [16:0] Tail Doorbell" on page 19
0x6_1120	SQE9CTL	"SQE[16:0]CTL - Submission Queue Engine 16:0 Control" on page 18
0x6_1128	SQE9LBASE	"SQE[16:0]LBASE - Submission Queue Engine 16:0 Lower Base Address" on page 18
0x6_112C	SQE9UBASE	"SQE[16:0]UBASE - Submission Queue Engine 16:0 Upper Base Address" on page 18
0x6_1130	SQE9TDBL	"SQE[16:0]TDBL - Submission Queue [16:0] Tail Doorbell" on page 19
0x6_1140	SQE10CTL	"SQE[16:0]CTL - Submission Queue Engine 16:0 Control" on page 18
0x6_1148	SQE10LBASE	"SQE[16:0]LBASE - Submission Queue Engine 16:0 Lower Base Address" on page 18
0x6_114C	SQE10UBASE	"SQE[16:0]UBASE - Submission Queue Engine 16:0 Upper Base Address" on page 18
0x6_1150	SQE10TDBL	"SQE[16:0]TDBL - Submission Queue [16:0] Tail Doorbell" on page 19
0x6_1160	SQE11CTL	"SQE[16:0]CTL - Submission Queue Engine 16:0 Control" on page 18
0x6_1168	SQE11LBASE	"SQE[16:0]LBASE - Submission Queue Engine 16:0 Lower Base Address" on page 18
0x6_116C	SQE11UBASE	"SQE[16:0]UBASE - Submission Queue Engine 16:0 Upper Base Address" on page 18
0x6_1170	SQE11TDBL	"SQE[16:0]TDBL - Submission Queue [16:0] Tail Doorbell" on page 19
0x6_1180	SQE12CTL	"SQE[16:0]CTL - Submission Queue Engine 16:0 Control" on page 18
0x6_1188	SQE12LBASE	"SQE[16:0]LBASE - Submission Queue Engine 16:0 Lower Base Address" on page 18
0x6_118C	SQE12UBASE	"SQE[16:0]UBASE - Submission Queue Engine 16:0 Upper Base Address" on page 18
0x6_1190	SQE12TDBL	"SQE[16:0]TDBL - Submission Queue [16:0] Tail Doorbell" on page 19
0x6_11A0	SQE13CTL	"SQE[16:0]CTL - Submission Queue Engine 16:0 Control" on page 18
0x6_11A8	SQE13LBASE	"SQE[16:0]LBASE - Submission Queue Engine 16:0 Lower Base Address" on page 18
0x6_11AC	SQE13UBASE	"SQE[16:0]UBASE - Submission Queue Engine 16:0 Upper Base Address" on page 18
0x6_11B0	SQE13TDBL	"SQE[16:0]TDBL - Submission Queue [16:0] Tail Doorbell" on page 19
0x6_11C0	SQE14CTL	"SQE[16:0]CTL - Submission Queue Engine 16:0 Control" on page 18
0x6_11C8	SQE14LBASE	"SQE[16:0]LBASE - Submission Queue Engine 16:0 Lower Base Address" on page 18
0x6_11CC	SQE14UBASE	"SQE[16:0]UBASE - Submission Queue Engine 16:0 Upper Base Address" on page 18
0x6_11D0	SQE14TDBL	"SQE[16:0]TDBL - Submission Queue [16:0] Tail Doorbell" on page 19
0x6_11E0	SQE15CTL	"SQE[16:0]CTL - Submission Queue Engine 16:0 Control" on page 18
0x6_11E8	SQE15LBASE	"SQE[16:0]LBASE - Submission Queue Engine 16:0 Lower Base Address" on page 18
0x6_11EC	SQE15UBASE	"SQE[16:0]UBASE - Submission Queue Engine 16:0 Upper Base Address" on page 18
0x6_11F0	SQE15TDBL	"SQE[16:0]TDBL - Submission Queue [16:0] Tail Doorbell" on page 19
0x6_1200	SQE16CTL	"SQE[16:0]CTL - Submission Queue Engine 16:0 Control" on page 18
0x6_1208	SQE16LBASE	"SQE[16:0]LBASE - Submission Queue Engine 16:0 Lower Base Address" on page 18
0x6_120C	SQE16UBASE	"SQE[16:0]UBASE - Submission Queue Engine 16:0 Upper Base Address" on page 18
0x6_1210	SQE16TDBL	"SQE[16:0]TDBL - Submission Queue [16:0] Tail Doorbell" on page 19
0x6_5000	CQE0CTL	"CQE[16:0]CTL - Completion Queue Engine 16:0 Control" on page 19
0x6_5008	CQE0LBASE	"CQE[16:0]LBASE - Completion Queue Engine 16:0 Lower Base Address" on page 19
0x6_500C	CQE0UBASE	"CQE[16:0]UBASE - Completion Queue Engine 16:0 Upper Base Address" on page 19
0x6_5010	CQE0HDBL	"CQE[16:0]HDBL - Completion Queue [16:0] Head Doorbell" on page 20
0x6_5020	CQE1CTL	"CQE[16:0]CTL - Completion Queue Engine 16:0 Control" on page 19
0x6_5028	CQE1LBASE	"CQE[16:0]LBASE - Completion Queue Engine 16:0 Lower Base Address" on page 19
0x6_502C	CQE1UBASE	"CQE[16:0]UBASE - Completion Queue Engine 16:0 Upper Base Address" on page 19
0x6_5030	CQE1HDBL	"CQE[16:0]HDBL - Completion Queue [16:0] Head Doorbell" on page 20
0x6_5040	CQE2CTL	"CQE[16:0]CTL - Completion Queue Engine 16:0 Control" on page 19

Table 8.4 Host Interface CSR Address Map

Node Address	Register Mnemonic	Register Description
0x6_5048	CQE2LBASE	"CQE[16:0]LBASE - Completion Queue Engine 16:0 Lower Base Address" on page 19
0x6_504C	CQE2UBASE	"CQE[16:0]UBASE - Completion Queue Engine 16:0 Upper Base Address" on page 19
0x6_5050	CQE2HDBL	"CQE[16:0]HDBL - Completion Queue [16:0] Head Doorbell" on page 20
0x6_5060	CQE3CTL	"CQE[16:0]CTL - Completion Queue Engine 16:0 Control" on page 19
0x6_5068	CQE3LBASE	"CQE[16:0]LBASE - Completion Queue Engine 16:0 Lower Base Address" on page 19
0x6_506C	CQE3UBASE	"CQE[16:0]UBASE - Completion Queue Engine 16:0 Upper Base Address" on page 19
0x6_5070	CQE3HDBL	"CQE[16:0]HDBL - Completion Queue [16:0] Head Doorbell" on page 20
0x6_5080	CQE4CTL	"CQE[16:0]CTL - Completion Queue Engine 16:0 Control" on page 19
0x6_5088	CQE4LBASE	"CQE[16:0]LBASE - Completion Queue Engine 16:0 Lower Base Address" on page 19
0x6_508C	CQE4UBASE	"CQE[16:0]UBASE - Completion Queue Engine 16:0 Upper Base Address" on page 19
0x6_5090	CQE4HDBL	"CQE[16:0]HDBL - Completion Queue [16:0] Head Doorbell" on page 20
0x6_50A0	CQE5CTL	"CQE[16:0]CTL - Completion Queue Engine 16:0 Control" on page 19
0x6_50A8	CQE5LBASE	"CQE[16:0]LBASE - Completion Queue Engine 16:0 Lower Base Address" on page 19
0x6_50AC	CQE5UBASE	"CQE[16:0]UBASE - Completion Queue Engine 16:0 Upper Base Address" on page 19
0x6_50B0	CQE5HDBL	"CQE[16:0]HDBL - Completion Queue [16:0] Head Doorbell" on page 20
0x6_50C0	CQE6CTL	"CQE[16:0]CTL - Completion Queue Engine 16:0 Control" on page 19
0x6_50C8	CQE6LBASE	"CQE[16:0]LBASE - Completion Queue Engine 16:0 Lower Base Address" on page 19
0x6_50CC	CQE6UBASE	"CQE[16:0]UBASE - Completion Queue Engine 16:0 Upper Base Address" on page 19
0x6_50D0	CQE6HDBL	"CQE[16:0]HDBL - Completion Queue [16:0] Head Doorbell" on page 20
0x6_50E0	CQE7CTL	"CQE[16:0]CTL - Completion Queue Engine 16:0 Control" on page 19
0x6_50E8	CQE7LBASE	"CQE[16:0]LBASE - Completion Queue Engine 16:0 Lower Base Address" on page 19
0x6_50EC	CQE7UBASE	"CQE[16:0]UBASE - Completion Queue Engine 16:0 Upper Base Address" on page 19
0x6_50F0	CQE7HDBL	"CQE[16:0]HDBL - Completion Queue [16:0] Head Doorbell" on page 20
0x6_5100	CQE8CTL	"CQE[16:0]CTL - Completion Queue Engine 16:0 Control" on page 19
0x6_5108	CQE8LBASE	"CQE[16:0]LBASE - Completion Queue Engine 16:0 Lower Base Address" on page 19
0x6_510C	CQE8UBASE	"CQE[16:0]UBASE - Completion Queue Engine 16:0 Upper Base Address" on page 19
0x6_5110	CQE8HDBL	"CQE[16:0]HDBL - Completion Queue [16:0] Head Doorbell" on page 20
0x6_5120	CQE9CTL	"CQE[16:0]CTL - Completion Queue Engine 16:0 Control" on page 19
0x6_5128	CQE9LBASE	"CQE[16:0]LBASE - Completion Queue Engine 16:0 Lower Base Address" on page 19
0x6_512C	CQE9UBASE	"CQE[16:0]UBASE - Completion Queue Engine 16:0 Upper Base Address" on page 19
0x6_5130	CQE9HDBL	"CQE[16:0]HDBL - Completion Queue [16:0] Head Doorbell" on page 20
0x6_5140	CQE10CTL	"CQE[16:0]CTL - Completion Queue Engine 16:0 Control" on page 19
0x6_5148	CQE10LBASE	"CQE[16:0]LBASE - Completion Queue Engine 16:0 Lower Base Address" on page 19
0x6_514C	CQE10UBASE	"CQE[16:0]UBASE - Completion Queue Engine 16:0 Upper Base Address" on page 19
0x6_5150	CQE10HDBL	"CQE[16:0]HDBL - Completion Queue [16:0] Head Doorbell" on page 20
0x6_5160	CQE11CTL	"CQE[16:0]CTL - Completion Queue Engine 16:0 Control" on page 19
0x6_5168	CQE11LBASE	"CQE[16:0]LBASE - Completion Queue Engine 16:0 Lower Base Address" on page 19
0x6_516C	CQE11UBASE	"CQE[16:0]UBASE - Completion Queue Engine 16:0 Upper Base Address" on page 19
0x6_5170	CQE11HDBL	"CQE[16:0]HDBL - Completion Queue [16:0] Head Doorbell" on page 20
0x6_5180	CQE12CTL	"CQE[16:0]CTL - Completion Queue Engine 16:0 Control" on page 19
0x6_5188	CQE12LBASE	"CQE[16:0]LBASE - Completion Queue Engine 16:0 Lower Base Address" on page 19
0x6_518C	CQE12UBASE	"CQE[16:0]UBASE - Completion Queue Engine 16:0 Upper Base Address" on page 19
0x6_5190	CQE12HDBL	"CQE[16:0]HDBL - Completion Queue [16:0] Head Doorbell" on page 20
0x6_51A0	CQE13CTL	"CQE[16:0]CTL - Completion Queue Engine 16:0 Control" on page 19

Table 8.4 Host Interface CSR Address Map

Node Address	Register Mnemonic	Register Description
0x6_51A8	CQE13LBASE	"CQE[16:0]LBASE - Completion Queue Engine 16:0 Lower Base Address" on page 19
0x6_51AC	CQE13UBASE	"CQE[16:0]UBASE - Completion Queue Engine 16:0 Upper Base Address" on page 19
0x6_51B0	CQE13HDBL	"CQE[16:0]HDBL - Completion Queue [16:0] Head Doorbell" on page 20
0x6_51C0	CQE14CTL	"CQE[16:0]CTL - Completion Queue Engine 16:0 Control" on page 19
0x6_51C8	CQE14LBASE	"CQE[16:0]LBASE - Completion Queue Engine 16:0 Lower Base Address" on page 19
0x6_51CC	CQE14UBASE	"CQE[16:0]UBASE - Completion Queue Engine 16:0 Upper Base Address" on page 19
0x6_51D0	CQE14HDBL	"CQE[16:0]HDBL - Completion Queue [16:0] Head Doorbell" on page 20
0x6_51E0	CQE15CTL	"CQE[16:0]CTL - Completion Queue Engine 16:0 Control" on page 19
0x6_51E8	CQE15LBASE	"CQE[16:0]LBASE - Completion Queue Engine 16:0 Lower Base Address" on page 19
0x6_51EC	CQE15UBASE	"CQE[16:0]UBASE - Completion Queue Engine 16:0 Upper Base Address" on page 19
0x6_51F0	CQE15HDBL	"CQE[16:0]HDBL - Completion Queue [16:0] Head Doorbell" on page 20
0x6_5200	CQE16CTL	"CQE[16:0]CTL - Completion Queue Engine 16:0 Control" on page 19
0x6_5208	CQE16LBASE	"CQE[16:0]LBASE - Completion Queue Engine 16:0 Lower Base Address" on page 19
0x6_520C	CQE16UBASE	"CQE[16:0]UBASE - Completion Queue Engine 16:0 Upper Base Address" on page 19
0x6_5210	CQE16HDBL	"CQE[16:0]HDBL - Completion Queue [16:0] Head Doorbell" on page 20

Table 8.4 Host Interface CSR Address Map

8.9.1 General Host Interface Registers

PCIECTL - PCI Express Control

Bit Field	Field Name	Type	Default Value	Description
0	BDISCARD	RW	0x0	**Discard Vendor Defined Broadcast Messages** When this bit is set, vendor defined Type 1 broadcast messages received on the upstream port are silently discarded. Silently discarding a TLP means that flow control credits are returned, TLP contents are discarded, and no error bits are set.
3:1	MRRS	RW	0x0	**Maximum Read Request Size** This field sets the maximum host interface PCIe read request size. The design supports all PCIe values; however, it is expected that this field will always be set to 4KB so that is the only value that is verified/validated. *0 (128B) 128B* *1 (246B) 256B* *2 (512B) 512B* *3 (1024B) 1024B* *4 (2048B) 2048B* *5 (4096B) 4096B* *others reserved*
19:4	RID	RW	0x0	**Requester ID** This field contains the requester ID value that used by requests issued by the host interface.

Bit Field	Field Name	Type	Default Value	Description
22:20	MPS	RO	0x0	**Maximum Payload Size** This field sets the maximum TLP payload size. As a receiver, the function must handle TLPs as large as the set value. As a transmitter, the function must not generate TLPs exceeding this value. *0 (s128) 128B* *1 (s246) 256B* *2 (s512) 512B* *3 (s1024) 1024B* *4 (s2048) 2048B* *5 (s4096) 4096B* *others reserved* Delaware supports only an MPS of 128B; therefore, this field is hardwired to zero.
31:23	Reserved	RO	0x0	

PCIEERR - PCI Express Error Status

Bit Field	Field Name	Type	Default Value	Description
0	UCP	RW1C	0x0	**Unexpected Completion** This bit is set when the host interface receives an unexpected completion from the PCIe host interface.
31:1	Reserved	RO	0x0	

MEMLBAR - Memory Lower Base Address

Bit Field	Field Name	Type	Default Value	Description
1:0	REGPROC	RW	0x0	**MMIO Register Processing** This field controls how memory mapped I/O registers implemented by the host interface hardware are processed *0 (discard) Write accesses are silently discarded. Reads are passed to a processor to be serviced via an IngressTlp message.* *1 (hardware) Writes are processed by the host interface hardware. Reads are passed to a processor to be serviced via an IngressTlp message.* *2 (processor) Writes and reads are passed to a processor to be serviced via an IngressTlp message.*
12:2	Reserved	RO	0x0	
31:13	LBASE	RW	0x0	**Lower Base Address** This field contains bits 13 through 31 of the 64-bit PCIe base address of PCIe memory mapped NNVMHCI registers.

MEMUBAR - Memory Upper Base Address

Bit Field	Field Name	Type	Default Value	Description
31:0	UBASE	RW	0x0	**Upper Base Address** This field contains bits 32 through 63 of the 64-bit PCIe base address of PCIe memory mapped NNVMHCI registers.

8.9.2 NVMHCI Submission Queue Manager

SQE[16:0]CTL - Submission Queue Engine 16:0 Control

Bit Field	Field Name	Type	Default Value	Description
0	EN	RW	0x0	**Enable** When this bit is set, the corresponding submission queue is enabled. When a zero is written to this bit, the submission queue engine stops fetching new commands on PCIe. The bit remains set until all outstanding requests on PCIe have been completed and corresponding CmdReady messages have been issued on the message network.
15:1	SIZE	RW	0x0	**Size** This field contains the size of the corresponding submission queue in units of 64B commands.
31:16	Reserved	RO	0x0	

SQE[16:0]LBASE - Submission Queue Engine 16:0 Lower Base Address

Bit Field	Field Name	Type	Default Value	Description
11:0	Reserved	RO	0x0	
31:12	LBASE	RW	0x0	**Lower Base Address** This field contains bits 12 through 31 of the 64-bit PCIe base address of the corresponding submission queue. Since the submission queues must be 4KB aligned, bits zero through 11 are implicitly zero.

SQE[16:0]UBASE - Submission Queue Engine 16:0 Upper Base Address

Bit Field	Field Name	Type	Default Value	Description
31:0	UBASE	RW	0x0	**Upper Base Address** This field contains bits 32 through 63 of the 64-bit PCIe base address of the corresponding submission queue.

SQE[16:0]TDBL - Submission Queue [16:0] Tail Doorbell

Bit Field	Field Name	Type	Default Value	Description
14:0	SQT	RW	0x0	**Submission Queue Tail** This field aliases to the SQT field in the corresponding SQxTDBL register in PCIe BAR 0.
31:15	Reserved	RO	0x0	

8.9.3 NVMHCI Completion Queue Manager

CQE[16:0]CTL - Completion Queue Engine 16:0 Control

Bit Field	Field Name	Type	Default Value	Description
0	EN	RW	0x0	**Enable** When this bit is set, the corresponding completion queue is enabled. When a zero is written to this bit, the completion queue engine stops updating completion queue entries. The bit remains set until the completion queue engine has been disabled..
15:1	SIZE	RW	0x0	**Size** This field contains the size of the corresponding completion queue in units of 8B completion queue entries.
31:16	Reserved	RO	0x0	

CQE[16:0]LBASE - Completion Queue Engine 16:0 Lower Base Address

Bit Field	Field Name	Type	Default Value	Description
11:0	Reserved	RO	0x0	
31:12	LBASE	RW	0x0	**Lower Base Address** This field contains bits 12 through 31 of the 64-bit PCIe base address of the corresponding completion queue. Since the completion queues must be 4KB aligned, bits zero through 11 are implicitly zero.

CQE[16:0]UBASE - Completion Queue Engine 16:0 Upper Base Address

Bit Field	Field Name	Type	Default Value	Description
31:0	UBASE	RW	0x0	**Upper Base Address** This field contains bits 32 through 63 of the 64-bit PCIe base address of the corresponding completion queue.

CQE[16:0]HDBL - Completion Queue [16:0] Head Doorbell

Bit Field	Field Name	Type	Default Value	Description
14:0	CQH	RW	0x0	**Completion Queue Head** This field aliases to the CQH field in the corresponding CQxHDBL register in PCIe BAR 0.
31:15	Reserved	RO	0x0	

8.9.4 PCIe Processor Interface

8.9.5 NVMHCI Submission Queue Manager

8.9.6 Message Network Interface Registers

ITLPQCTL - Ingress TLP Outbound Message Queue Control

Bit Field	Field Name	Type	Default Value	Description
0	Reserved	RO	0x0	
8:1	NODEID	RW	0x0	**Inbound Message Queue Node ID** This field contains the Node ID of the inbound message queue interface (i.e., the destination node) with which this outbound message queue is associated.
13:9	RCVQ	RW	0x0	**Receive Queue** This field contains the Receive Queue ID within the inbound message queue interface with which this outbound message queue is associated.
31:14	Reserved	RO	0x0	

DMACQCTL[7:0] - DMA Completion Outbound Message Queue Control [7:0]

Bit Field	Field Name	Type	Default Value	Description
0	EN	RW	0x0	**Enable** When this bit is set the outbound message queue is enabled and the NODEID field value in this register is valid. When this bit is cleared, the outbound message queue is disabled.
8:1	NODEID	RW	0x0	**Inbound Message Queue Node ID** This field contains the Node ID of the inbound message queue interface (i.e., the destination node) with which this outbound message queue is associated.
13:9	RCVQ	RW	0x0	**Receive Queue** This field contains the Receive Queue ID within the inbound message queue interface with which this outbound message queue is associated.
31:14	Reserved	RO	0x0	

CMDRQCTL[7:0] - Command Ready Outbound Message Queue Control [7:0]

Bit Field	Field Name	Type	Default Value	Description
0	EN	RW	0x0	**Enable** When this bit is set the outbound message queue is enabled and the NODEID field value in this register is valid. When this bit is cleared, the outbound message queue is disabled.
8:1	NODEID	RW	0x0	**Inbound Message Queue Node ID** This field contains the Node ID of the inbound message queue interface (i.e., the destination node) with which this outbound message queue is associated.
13:9	RCVQ	RW	0x0	**Receive Queue** This field contains the Receive Queue ID within the inbound message queue interface with which this outbound message queue is associated.
31:14	Reserved	RO	0x0	

PCI Express Interface

9.1 Introduction

9.2 CAR Registers

- Table 9.1 shows the PCIe interface node address map. The CAR address is formed using the node address as described in Section 4.2.3, "CAR Address Space" on page 4-4.
- PCIe Interface CSR registers are CAR accessible.

Node Address	Register Mnemonic	Register Description
0x0_0000	PCIEITST	"PCIEITST - PCI Express Interface Test" on page 2
0x0_0004	PCIEIRST	"PCIEIRST - PCI Express Interface Reset" on page 2
0x0_0008	PCIEIMCTL	"PCIEIMCTL - PCI Express Interface Mode Control" on page 2
0x0_000C	PCIEIMSTS	"PCIEIMSTS - PCI Express Interface Mode Status" on page 2
0x0_0010	PCIEILTSSM	"PCIEILTSSM - PCI Express Interface LTSSM Status" on page 3
0x0_0014	PCIEIRXDET	"PCIEIRXDET - PCI Express Interface RX Detect Status" on page 3
0x0_0018	PCIEIRXIDLE	"PCIEIRXIDLE - PCI Express Interface RX idle Status" on page 3
0x0_001C	PCIEIERR	"PCIEIERR - PCI Express Interface Error Status" on page 3
0x0_0020	NWCFGCONT0	"NWCFGCONST0 - Northwest Logic Configuration Constant 0" on page 4
0x0_0024	NWCFGCTRL0	"NWCFGCTRL0 - Northwest Logic Configuration Control 0" on page 4
0x0_0030	PCIEITXACTL	"PCIEITXACTL - PCI Express Interface Transmit Analog Control" on page 4
0x0_0034	PCIEILWSTS	"PCIEILWSTS - PCI Express Interface Link Width Status" on page 4
0x0_0038	PCIEIMPS	"PCIEIMPS - PCI Express Interface Maximum Payload Size Control" on page 4
0x0_0040	NWPCIESTS0	"NWPCIESTS0 - Northwest Logic PCIe Status 0" on page 5
0x0_0044	NWPCIESTS1	"NWPCIESTS1 - Northwest Logic PCIe Status 1" on page 5
0x0_0048	NWPCIESTS2	"NWPCIESTS2 - Northwest Logic PCIe Status 2" on page 5
0x0_004C	NWPCIESTS3	"NWPCIESTS3 - Northwest Logic PCIe Status 3" on page 5
0x0_0050	NWPCIESTS4	"NWPCIESTS4 - Northwest Logic PCIe Status 4" on page 5
0x0_0054	NWPCIESTS5	"NWPCIESTS5 - Northwest Logic PCIe Status 5" on page 5
0x0_0058	NWPCIESTS6	"NWPCIESTS6 - Northwest Logic PCIe Status 6" on page 5
0x0_005C	NWPCIESTS7	"NWPCIESTS7 - Northwest Logic PCIe Status 7" on page 5
0x0_0060	PCIEIRXTLPS	"PCIEIRXTLPS - PCI Express Interface Receive TLP Status" on page 6
0x0_0064	PCIEITXTLPS	"PCIEITXTLPS - PCI Express Interface Transmit TLP Status" on page 6
0x0_0070	PCIEICRSCTL	"PCIEICRSCTL - PCI Express Interface CRS Control" on page 6

Table 9.1 PCIe Interface CSR Address Map

9.2.1 PCIe Interface Registers

PCIEITST - PCI Express Interface Test

Bit Field	Field Name	Type	Default Value	Description
31:0	TEST	RW	0x0	**Test** This is a read/write test register used to verify CAR connectivity. It has no functional effect on the operation of the PCIe interface.

PCIEIRST - PCI Express Interface Reset

Bit Field	Field Name	Type	Default Value	Description
0	RST	RW	0x1	**PCIe Interface Reset** The value of this field controls the reset input to the PCIe interface logic. This field should not be modified during normal operation.
31:1	Reserved	RO	0x0	

PCIEIMCTL - PCI Express Interface Mode Control

Bit Field	Field Name	Type	Default Value	Description
0	Mode	RW	0x1	**PCIe Interface Mode Control** This field controls the operating mode of the PCIe interface. This field should only be modified when the PCIe interface is reset (i.e., the RST bit in the PCIEIRST register is cleared). *0 (gen1) PCIe Gen1 (2.5 GT/s)* *1 (gen2) PCIe Gen2 (5 GT/s)*
31:1	Reserved	RO	0x0	

PCIEIMSTS - PCI Express Interface Mode Status

Bit Field	Field Name	Type	Default Value	Description
0	Mode	RO	0x1	**PCIe Interface Mode Status** This field contains the negotiated link mode (i.e., Gen 1 or Gen 2) when the link is in L0. *0 (gen1) PCIe Gen1 (2.5 GT/s)* *1 (gen2) PCIe Gen2 (5 GT/s)*
31:1	Reserved	RO	0x0	

PCIEILTSSM - PCI Express Interface LTSSM Status

Bit Field	Field Name	Type	Default Value	Description
5:0	LTSSM	RO	0x0	**PCIe Interface LTSSM Status** This field contains the current PCIe interface LTSSM status. *0* *(Detect) Detect LTSSM state* *1* *(Polling) Polling LTSSM state* *2* *(Config) Configuration LTSSM state* *3* *(L0) L0 LTSSM state* *4* *(Recovery) Recovery LTSSM state*
31:6	Reserved	RO	0x0	

PCIEIRXDET - PCI Express Interface RX Detect Status

Bit Field	Field Name	Type	Default Value	Description
3:0	RCV	RO	0x0	**Receiver Detected** Each bit in this field represents a PCIe lane (i.e., bit 0 represents lane 0, bit 1 represents lane 1, and so on). A bit in this field is set if a receiver is detected on the corresponding lane.
31:4	Reserved	RO	0x0	

PCIEIRXIDLE - PCI Express Interface RX Idle Status

Bit Field	Field Name	Type	Default Value	Description
3:0	IDLE	RO	0x0	**Electrical Idle** Each bit in this field represents a PCIe lane (i.e., bit 0 represents lane 0, bit 1 represents lane 1, and so on). A bit in this field is set if an electrical idle is detected on the corresponding lane.
31:4	Reserved	RO	0x0	

PCIEIERR - PCI Express Interface Error Status

Bit Field	Field Name	Type	Default Value	Description
4:0	ERR	RW	0x0	**Error** This field encodes various errors. This needs to be verified. *0* *(BuffOvr) Phy Rx Buffer Overflow* *1* *(BuffUnd) Phy Rx Buffer Underflow* *2* *(Dis) Phy Rx Disparity Error* *3* *(8b10b) Phy Rx 8b10b Error* *4* *(Malformed) Rx Malformed TLP*
31:5	Reserved	RO	0x0	

NWCFGCONST0 - Northwest Logic Configuration Constant 0

Bit Field	Field Name	Type	Default Value	Description
31:0	Reserved	RO	0x0	**See RTL**

NWCFGCTRL0 - Northwest Logic Configuration Control 0

Bit Field	Field Name	Type	Default Value	Description
31:0	Reserved	RO	0x0	**See RTL**

PCIEITXACTL - PCI Express Interface Transmit Analog Control

Bit Field	Field Name	Type	Default Value	Description
31:0	Reserved	RO	0x0	**See RTL**

PCIEILWSTS - PCI Express Interface Link Width Status

Bit Field	Field Name	Type	Default Value	Description
3:0	LW	RO	0x0	**Link Width** This field contains the negotiated link width when the link is in L0. *1 (x1) x1 Link Width* *2 (x2) x2 Link Width* *4 (x4) x4 Link Width* *others reserved*
31:4	Reserved	RO	0x0	

PCIEIMPS - PCI Express Interface Maximum Payload Size Control

Bit Field	Field Name	Type	Default Value	Description
2:0	MPS	RW	0x0	**Maximum Payload Size** This field sets the maximum TLP payload size. *0 (s128) 128B* *1 (s246) 256B* *2 (s512) 512B* *3 (s1024) 1024B* *4 (s2048) 2048B* *5 (s4096) 4096B* *others reserved* Delaware supports only an MPS of 128B; therefore, this field should not be set to a value other than zero.
31:3	Reserved	RO	0x0	

NWPCIESTS0 - Northwest Logic PCIe Status 0

Bit Field	Field Name	Type	Default Value	Description
31:0	Reserved	RO	0x0	See RTL

NWPCIESTS1 - Northwest Logic PCIe Status 1

Bit Field	Field Name	Type	Default Value	Description
31:0	Reserved	RO	0x0	See RTL

NWPCIESTS2 - Northwest Logic PCIe Status 2

Bit Field	Field Name	Type	Default Value	Description
31:0	Reserved	RO	0x0	See RTL

NWPCIESTS3 - Northwest Logic PCIe Status 3

Bit Field	Field Name	Type	Default Value	Description
31:0	Reserved	RO	0x0	See RTL

NWPCIESTS4 - Northwest Logic PCIe Status 4

Bit Field	Field Name	Type	Default Value	Description
31:0	Reserved	RO	0x0	See RTL

NWPCIESTS5 - Northwest Logic PCIe Status 5

Bit Field	Field Name	Type	Default Value	Description
31:0	Reserved	RO	0x0	See RTL

NWPCIESTS6 - Northwest Logic PCIe Status 6

Bit Field	Field Name	Type	Default Value	Description
31:0	Reserved	RO	0x0	See RTL

NWPCIESTS7 - Northwest Logic PCIe Status 7

Bit Field	Field Name	Type	Default Value	Description
31:0	Reserved	RO	0x0	See RTL

PCIEIRXTLPS - PCI Express Interface Receive TLP Status

Bit Field	Field Name	Type	Default Value	Description
31:0	RXCNT	RO	0x0	**Receive TLP Count** This field contains the total number of TLPs that have been received since the PCIe interface reset was negated. The value in this field is not reliable since it is updated in the PCIe clock domain and read in the system clock domain. If the value read from this register is the same for two consecutive reads, then there is reason to believe that this is the true value.

PCIEITXTLPS - PCI Express Interface Transmit TLP Status

Bit Field	Field Name	Type	Default Value	Description
31:0	TXCNT	RO	0x0	**Transmit TLP Count** This field contains the total number of TLPs that have been transmitted since the PCIe interface reset was negated. The value in this field is not reliable since it is updated in the PCIe clock domain and read in the system clock domain. If the value read from this register is the same for two consecutive reads, then there is reason to believe that this is the true value.

PCIEICRSCTL - PCI Express Interface CRS Control

Bit Field	Field Name	Type	Default Value	Description
0	CRS	RW	0x1	**Configuration Retry Status Control** When this bit is set, all non-posted TLPs received by the PCIe interface from the PCIe link are automatically terminated by the PCIe interface with a completion with status Configuration Request Retry Status (CRS). In addition, all other received TLPs are dropped and the behavior of egress (i.e., transmit) TLPs is undefined. *0 (disabled) Automatic CSR disabled* *1 (enabled) Automatic CSR enabled*
31:1	Reserved	RO	0x0	

Flash Controller

10.1 Introduction

10.2 Flash Timing

- The Flash controller is enabled by setting the Enable (EN) bit in the Flash Controller Control (FCCTL) register.
  - Firmware must wait a minimum of 100uS before enabling the Flash controller. This ensures that 100uS time value in step 4 of the Device Initialization section of the Micron data sheet is satisfied.
- The register field values shown in Table 10.1 must be programmed before the Flash controller is enabled. These values are appropriate for an IDT Flash module operating at a frequency of 125 MHz or less.
- Prior to using a Flash channel, the channel must be calibrated. See Section 10.6, "Flash Calibration" on page 10-6.

Register Name	Register Field	Value	Description
FTPROR	TPROR	0x1E848	Set device reset time to 1ms
FTR	TR	0xC35	Set read page operation time to 25us
FTPROG	TPROG	0x7A12	Set program page operation time to 250us
FTFEAT	TFEAT	0x7D	Set busy time for SET FEATURES and SET FEATURE operation to 1us.
FTBERS	TBERS	0x1E848	Set block erase operation time to 1ms
FINITDELAY	INITDELAY	0x7530	Set Flash initialization delay to 100us
FTRWPOLL	TRDPOLL	0x7D	Set read polling time to 1us
FTRWPOLL	TWRPOLL	0x186A	Set write/program polling time to 50us
FTERPOLL	TERPOLL	0x1E848	Set erase polling time to 1ms

Table 10.1 Flash Timing Register Configuration (125 MHz or less)

10.3 Flash Devices and Addressing

- Delaware is designed to support Micron 34nm SLC Flash.
  - IDT Flash module supports the following Flash devices.
    - MT29F64G08AMCBBH2 provides one LUN and 2GB per target.
    - MT29F128G08AUCBBH3 provides two LUNs and 4GB two LUNs.

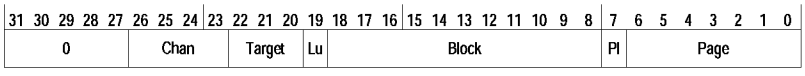


Figure 10.1 Flash Address Format

- The Flash Address (FlashAddr) field in FlashRead, FlashWrite and FlashErase messages specify the Flash page or block on which to operate. All of the messages use the same address format. The Page field is ignored by FlashErase which operates on Flash blocks. The Format of the flash address field is shown in Figure 10.1.
  - The Page field specifies the Flash page. There are 128 pages per block.
  - The Pl field specifies the Flash plane. There are two planes per devices.
  - The Block field specifies the Flash block. There are 2048 blocks per plane.
  - The Lu field specifies the Flash LUN. Delaware supports SLC Flash devices with one or two LUNs.
  - The Target field specifies the Flash target. Delaware supports 8 targets per channel.
  - The Chan field specifies the Flash channel. Delaware supports 16 channels.

10.4 Page Frame Addressing

- Page frames are stored in controller memory as described in Section 4.3.3. "Page Frame" on page 4-6. This section provides a concrete example of this addressing.

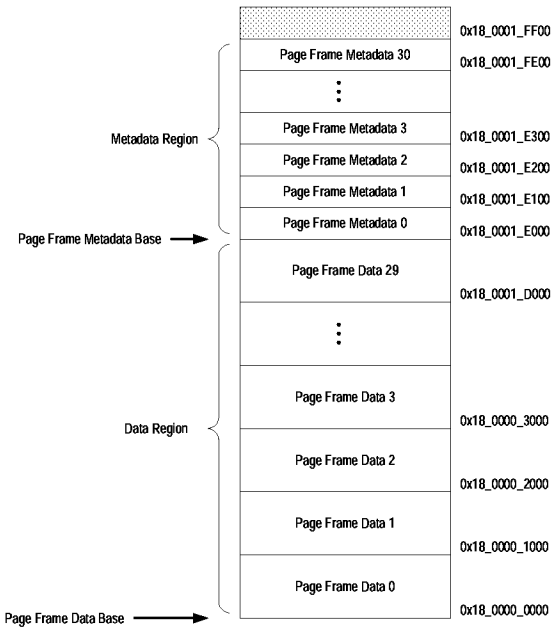


Figure 10.2 Example Page Frame Layout in Buffer RAM

- Delaware support a 1Mb buffer RAM (i.e., 128KB) and a page frame with a 4KB data region and a 256B metadata region (i.e., 224B of metadata that occupy a 256B region). Assume that the entire buffer RAM is allocated to page frames.
  - A page frame occupies 4352B; therefore, a 128KB buffer RAM can hold 30 pages.
  - The layout of the 30 pages in buffer RAM is shown in Figure 10.2.
- The page frame region base registers should be initialized as follows for the layout in this example.
  - The page frame data base is 0x18_0000_0000. Therefore, the Base Address (BASE-ADDR) field in the Buffer RAM Page Frame Data Base (BRPFDB) should be initialized with 0x180_0000 (i.e., the byte address shifted right 12-bits).

- The page frame metadata base is 0x18_0001_E000. Therefore, the Base Address (BASEADDR) field in the Buffer RAM Page Frame Metadata Base (BRPFMB) should be initialized with 0x1800_01E0 (i.e., the byte address shifted right 8-bits).

- Assume that the Flash controller received a FlashRead or a FlashWrite request message for an operation associated with page frame one. Then the Memory Address (Address) field in this message request would have the value of 0xC000_0080 (i.e., the value 0x18_0000_1000 shifted right five bits). This address corresponds to the data region associated with the page frame.
- The Flash controller uses the address field to determine the page number being access using the equations described in Section 4.3.3, "Page Frame" on page 4-6.
  - The following example computations assume that the values have been shifted by the appropriate amount required to form a byte address.
  - The page number may be computed as follows using the value in the Address field of the message request and the value in the BRPFDB register.
    - PageNumber = (DataRegion - PageFrameDataBase) >> 12
    - PageNumber = (0x18_0000_1000 - 0x18_0000_0000) >> 12
    - PageNumber = 1
  - Using the page number computed above and the value in the BRPFMB register, the start of the page frame metadata region may be computed as follows.
    - MetadataRegion = PageFrameMetadataBase + (PageNumber << 8)
    - MetadataRegion = 0x18_0001_E000 +(1 << 8)
    - MetadataRegion = 0x18_0001_E100

10.5 Flash Controller Request Messages

- In response to each Flash controller request message a completion (i.e., FlashOpCpl) message is generated.
- The following malformed request checks are performed on received messages. In addition to these common checks, request message specific checks may be performed.
  - The FlashOp field encodes an operation that is not implemented by the Flash controller.
- When a malformed request is detected, a completion message is generated with the status field set to malformed. The Flash controller does not process the message.

10.5.1 FlashRead

- The Flash Read (FlashRead) request message causes a Flash page (both data and metadata) to be read from Flash, detected errors to be corrected by the ECC correction engine, and the contents of the Flash page to be written to a page frame in controller memory.
- The format of a FlashRead request message is shown in Section 14.8, "FlashRead" on page 14-5.
- Request message fields
  - The FlashOp field encodes the operation to perform (i.e., read).
  - The EccOff bit allows ECC correction to be disabled. When correction is disabled the page contents read from Flash are written unmodified to the page frame.
  - The OutMsgQ field specifies the number of the flash completion outbound message queue that should be used to transmit the FlashOpCpl message.

- The Permute field is not used by Delaware and the contents of this field are ignored.
- The Mask field is not used by Delaware and the contents of this field are ignored.
- The FlashAddr field contains the Flash page address to be read. The format of the address is shown in Figure 10.1.
- The Addr field specifies the address of the page frame data region where the data read from Flash should be written. The metadata region is computed as described in Section 4.3.3. "Page Frame" on page 4-6.
- Tag0 and Tag1 contain opaque data that is passed from the request to the corresponding FlashOpCpl message.

- Errors
  - Malformed Request
    - There is no FlashRead specific error checking
  - Error
    - The read operation to the Flash did not complete (i.e., status polling limit exceeded)
    - The ECC correction unit detected more bits in error than could be corrected
- Completion message (FlashOpCpl) fields
  - The Tag0 and Tag1 fields contain the value from the corresponding request.
  - The Status field reports any detected errors.
  - The EccErrors field contains the total number of bits that were detected in error by the ECC correction unit.
  - The ChunkEccErrors field contains the number of bits that were detected in error by the ECC correction unit in the ECC chunk with the most errors.
  - The Retries field contains the number of times the Flash device was polled before the command was completed.
  - The FlashAddr field contains the contents of the corresponding field in the request message.
  - The Addr field contains the contents of the corresponding field in the request message.

10.5.2 FlashWrite

- The Flash Read (FlashWrite) request message causes a page frame (both data and metadata) to be read from controller memory, ECC parity bits to be computed, and the contents of the page frame and ECC to be written to a Flash Page.
- The format of a FlashWrite request message is shown in Section 14.9. "FlashWrite" on page 14-6.
- Request message fields
  - The FlashOp field encodes the operation to perform (i.e., write).
  - The EccOff bit allows ECC correction to be disabled. When correction is disabled the page contents are written to a Flash page unmodified (i.e., the ECC parity bits are not inserted).
  - The OutMsgQ field specifies the number of the flash completion outbound message queue that should be used to transmit the FlashOpCpl message.
  - The Permute field is not used by Delaware and the contents of this field are ignored.

- The Mask field is not used by the Flash write message request.
- The FlashAddr field contains the Flash page address to be written. The format of the address is shown in Figure 10.1.
- The Addr field specifies the address of the page frame data region containing data to be written. The metadata region is computed as described in Section 4.3.3. "Page Frame" on page 4-6.
- Tag0 and Tag1 contain opaque data that is passed from the request to the corresponding FlashOpCpl message.

- Errors
  - Malformed Request
    - There is no FlashWrite specific error checking
  - Error
    - The write operation to the Flash did not complete (i.e., status polling limit exceeded)
- Completion message (FlashOpCpl) fields
  - The Tag0 and Tag1 fields contain the value from the corresponding request.
  - The Status field reports any detected errors.
  - The EccErrors field is not used by the Flash write message request and its contents are undefined.
  - The ChunkEccErrors field is not used by the Flash write message request and its contents are undefined.
  - The Retries field contains the number of times the Flash device was polled before the command was completed.
  - The FlashAddr field contains the contents of the corresponding field in the request message.
  - The Addr field contains the contents of the corresponding field in the request message.

10.5.3 FlashErase

- The Flash Erase (FlashErase) request message causes a Flash block to be erased.
- The format of a FlashErase request message is shown in Section 14.6. "FlashErase" on page 14-3.
- Request message fields
  - The FlashOp field encodes the operation to perform (i.e., erase).
  - The EccOff bit is not used by the Flash erase message request.
  - The OutMsgQ field specifies the number of the flash completion outbound message queue that should be used to transmit the FlashOpCpl message.
  - The Permute field is not used by the Flash erase message request.
  - The Mask field is not used by the Flash erase message request.
  - The FlashAddr field contains the Flash block address to be erased. The format of the address is shown in Figure 10.1. The Page field is ignored.
  - The Addr field is not used by the Flash erase message request.

- Tag0 and Tag1 contain opaque data that is passed from the request to the corresponding FlashOpCpl message.

- Errors
  - Malformed Request
    - There is no FlashErase specific error checking
  - Error
    - The erase operation to the Flash did not complete (i.e., status polling limit exceeded)

- Completion message (FlashOpCpl) fields
  - The Tag0 and Tag1 fields contain the value from the corresponding request.
  - The Status field reports any detected errors.
  - The EccErrors field is not used by the Flash erase message request and its contents are undefined.
  - The ChunkEccErrors field is not used by the Flash erase message request and its contents are undefined.
  - The Retries field contains the number of times the Flash device was polled before the command was completed.
  - The FlashAddr field contains the contents of the corresponding field in the request message.
  - The Addr field is not used by the Flash erase message request and its contents are undefined.

10.6 Flash Calibration

- Prior to using a Flash channel for Flash read operations, the channel must be calibrated. Calibration adjusts Flash channel sampling parameters so that data received in a DDR manner from a Flash target is sampled correctly by the Delaware FPGA.
- Flash calibration uses the ONFI Read Parameter Page (ECh) command. The Read Parameter Page command retrieves a data structure from a Flash target that describes its organization, features, timing, and other behavioral parameters. This data structure contains three mandatory copies of a parameter page data structure.
  - Bytes 254 and 255 of each parameter page data structure contains a CRC-16 that may be used to verify the integrity of the parameter page.
  - See the Open NAND Flash Interface Specification (ONFi) Revision 2.2 for additional details regarding the Read Parameter Page command that a the definition of the parameter page data structure.
- When the Calibration Mode (CM) bit is set in the Flash Controller Control (FCCTL) register, the Flash controller enters a calibration mode. While in this mode, FlashRead message requests received by the Flash controller are processed by issuing Read Parameter page commands to Flash instead of normal Flash read commands. Data written to controller memory in the processing of the FlashRead message request is the value returned by the Read Parameter Page and sampled by the Delaware FPGA.
- Flash Calibration is performed in the following manner.
  - Enter calibration mode by setting CM bit in the FCCTL register.
  - Issue a FlashRead operation to a target
  - Check data retuned by the Read Parameter Page operation.

- Check the CRC of the three mandatory copies of the parameter page data structure.
  - If the CRC for one of the three mandatory parameter page data structures is correct, then Flash calibration is correct.
  - If CRC for none of the three mandatory parameter page data structures is valid, then Flash calibration is incorrect.

° Flash calibration is modified by modifying parameters in the Flash Timing Synchronous 2 (FTSYNC2) register, the above process is repeated, and the optimal calibration values are selected so that data is sampled in the middle of the data valid eye.

° The above process is repeated for all Flash channels. This process may be enhanced from the simple flow presented above so that multiple Flash operations to different channels are overlapped to reduce calibration time.

10.7 Message Network Interface

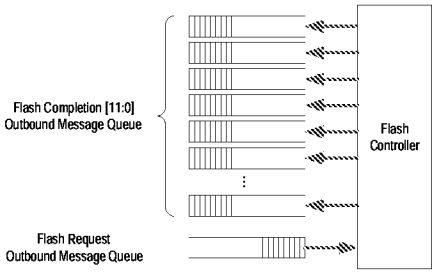


Figure 10.3 Flash Controller Inbound and Outbound Message Queues

Queue ID	Function	Name	Queue Depth [1]
0	Flash Completions	Flash Completion 0 Outbound Message Queue	8
1		Flash Completion 1 Outbound Message Queue	8
2		Flash Completion 2 Outbound Message Queue	8
3		Flash Completion 3 Outbound Message Queue	8
4		Flash Completion 4 Outbound Message Queue	8
5		Flash Completion 5 Outbound Message Queue	8
6		Flash Completion 6 Outbound Message Queue	8
7		Flash Completion 7 Outbound Message Queue	8
8		Flash Completion 8 Outbound Message Queue	8
9		Flash Completion 9 Outbound Message Queue	8
10		Flash Completion 10 Outbound Message Queue	8
11		Flash Completion 11 Outbound Message Queue	8

Table 10.2 Flash Controller Outbound Message Queues

1. This corresponds to the maximum number of supported entries in the free queue

Queue ID	Function	Name	Queue Depth [1]
0	Flash Requests	Flash Request Inbound Message Queue	128

Table 10.3 Flash Controller Inbound Message Queues

1. This corresponds to the maximum number of supported entries in the receive queue

- Figure 10.4 shows the node address map for the Flash controller. Associated with the inbound message queues is a dedicated 4KB message queue memory. Messages buffers for an inbound messages must be allocated from this memory.

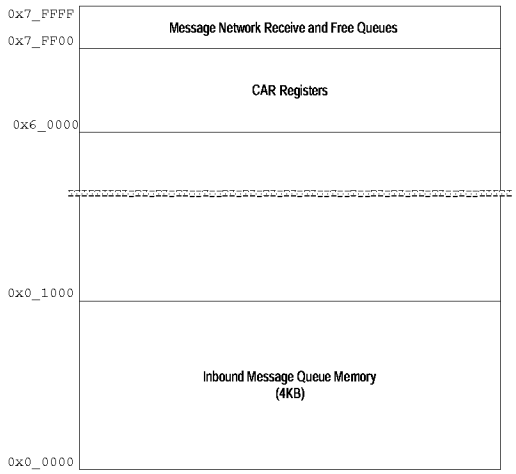


Figure 10.4 Host Interface Node Address Map

10.8 CAR Registers

- Table 10.4 shows the Flash controller node address map. The CAR address is formed using the node address as described in Section 4.2.3, "CAR Address Space" on page 4-4.

Node Address	Register Mnemonic	Register Description
0x6_0000	FCCTL	"FCCTL - Flash Controller Control" on page 9
0x6_0004	FINITDELAY	"FINITDELAY - Flash Initialization Delay" on page 10
0x6_0008	FTM	"FTM - Flash Timing Mode" on page 11
0x6_000C	FTR	"FTR - Flash Timing Read Page" on page 11
0x6_0010	FTPROG	"FTPROG - Flash Timing Program Page" on page 11
0x6_0014	FTFEAT	"FTFEAT - Flash Timing Busy Time for Set Feature" on page 11
0x6_0018	FTBERS	"FTBERS - Flash Timing Block Erase Time" on page 12
0x6_001C	FTPOR	"FTPOR - Flash Timing Power-On Reset Time" on page 12
0x6_0020	FTRWPOLL	"FTRWPOLL - Flash Timing Read and Write Polling Time" on page 12
0x6_0024	FTERPOLL	"FTERPOLL - Flash Timing Erase Polling Time" on page 12
0x6_0028	FTASYNC0	"FTASYNC0 - Flash Timing Asynchronous 0" on page 13
0x6_002C	FTASYNC1	"FTASYNC1 - Flash Timing Asynchronous 1" on page 13
0x6_0030	FTSYNC0	"FTSYNC0 - Flash Timing Synchronous 0" on page 14
0x6_0034	FTSYNC1	"FTSYNC1 - Flash Timing Synchronous 1" on page 15

Table 10.4 Flash Controller CSR Address Map

Node Address	Register Mnemonic	Register Description
0x6_0040	CH0FTSYNC2	"CH[7:0]FTSYNC2 - Channel [7:0] Flash Timing Synchronous 2" on page 15
0x6_0044	CH1FTSYNC2	"CH[7:0]FTSYNC2 - Channel [7:0] Flash Timing Synchronous 2" on page 15
0x6_0048	CH2FTSYNC2	"CH[7:0]FTSYNC2 - Channel [7:0] Flash Timing Synchronous 2" on page 15
0x6_004C	CH3FTSYNC2	"CH[7:0]FTSYNC2 - Channel [7:0] Flash Timing Synchronous 2" on page 15
0x6_0050	CH4FTSYNC2	"CH[7:0]FTSYNC2 - Channel [7:0] Flash Timing Synchronous 2" on page 15
0x6_0054	CH5FTSYNC2	"CH[7:0]FTSYNC2 - Channel [7:0] Flash Timing Synchronous 2" on page 15
0x6_0058	CH6FTSYNC2	"CH[7:0]FTSYNC2 - Channel [7:0] Flash Timing Synchronous 2" on page 15
0x6_005C	CH7FTSYNC2	"CH[7:0]FTSYNC2 - Channel [7:0] Flash Timing Synchronous 2" on page 15
0x6_0100	BRPFDB	"BRPFDB - Buffer RAM Page Frame Data Base" on page 16
0x6_0104	BRPFMB	"BRPFMB - Buffer RAM Page Frame Metadata Base" on page 16
0x6_0108	DRPFDB	"DRPFDB - DRAM Page Frame Data Base" on page 17
0x6_010C	DRPFMB	"DRPFMB - DRAM Page Frame Metadata Base" on page 17
0x6_0200	FCCPQCTL[0]	"FCPQCTL[11:0] - Flash Completion Outbound Message Queue Control [11:0]" on page 17
0x6_0204	FCCPQCTL[1]	"FCPQCTL[11:0] - Flash Completion Outbound Message Queue Control [11:0]" on page 17
0x6_0208	FCCPQCTL[2]	"FCPQCTL[11:0] - Flash Completion Outbound Message Queue Control [11:0]" on page 17
0x6_020C	FCCPQCTL[3]	"FCPQCTL[11:0] - Flash Completion Outbound Message Queue Control [11:0]" on page 17
0x6_0210	FCCPQCTL[4]	"FCPQCTL[11:0] - Flash Completion Outbound Message Queue Control [11:0]" on page 17
0x6_0214	FCCPQCTL[5]	"FCPQCTL[11:0] - Flash Completion Outbound Message Queue Control [11:0]" on page 17
0x6_0218	FCCPQCTL[6]	"FCPQCTL[11:0] - Flash Completion Outbound Message Queue Control [11:0]" on page 17
0x6_021C	FCCPQCTL[7]	"FCPQCTL[11:0] - Flash Completion Outbound Message Queue Control [11:0]" on page 17
0x6_0220	FCCPQCTL[8]	"FCPQCTL[11:0] - Flash Completion Outbound Message Queue Control [11:0]" on page 17
0x6_0224	FCCPQCTL[9]	"FCPQCTL[11:0] - Flash Completion Outbound Message Queue Control [11:0]" on page 17
0x6_0228	FCCPQCTL[10]	"FCPQCTL[11:0] - Flash Completion Outbound Message Queue Control [11:0]" on page 17
0x6_022C	FCCPQCTL[11]	"FCPQCTL[11:0] - Flash Completion Outbound Message Queue Control [11:0]" on page 17

Table 10.4 Flash Controller CSR Address Map

10.8.1 Flash Configuration Registers

FCCTL - Flash Controller Control

Bit Field	Field Name	Type	Default Value	Description
0	EN	RW	0x0	**Enable** Setting this bit enables the Flash controller. When the Flash controller is disabled, it does not process received Flash controller messages (i.e., they remain queued in the message queue). Once the Flash controller is enabled following a reset in the FPGA prototype it cannot be disabled (i.e., once the EN bit transitions from zero to one, it cannot be transitioned back to zero). Violating this rule produces undefined behavior.
1	WP	RW	0x1	**Write Protect** This bit controls the state of all Flash write protect output signals. When this bit is set, the Flash write protect signals are driven low. When this bit is cleared, the Flash write protect signals are driven high.

Bit Field	Field Name	Type	Default Value	Description
2	CM	RW	0x1	**Calibration Mode** This bit is intended for use during Flash calibration. When this bit is set, FlashRead message requests received by the Flash Controller cause a Read Parameter Page Flash command to be issued to Flash instead of a normal Flash read command. This is a global bit that affects all Flash channels and targets. The Address field in the FlashRead message request is used to select the specific Flash channel/target in the same manner that it does for a normal Flash read command; however, the address itself is not used in the Read Parameter Page command. Data returned by the Read Parameter Page Flash command is stored in controller memory in the same manner as data returned by a Flash read command. Firmware may transition this bit from one to zero and zero to one multiple times while the Flash controller is enabled (i.e., EN bit is set); however, the Flash controller must be idle (i.e., not executing Flash message requests) when the state of this bit is modified. Modifying the state of this bit while the Flash controller is not idle produces undefined results. Issuing any message request other than FlashRead while this bit is set produces undefined results.
31:4	Reserved	RO	0x0	

FINITDELAY - Flash Initialization Delay

Bit Field	Field Name	Type	Default Value	Description
17:0	INITDELAY	RW	0x7530	**Initialization Delay** This field specifies the amount of time that the Flash controller waits before initializing Flash channels in units of Flash controller clock cycles. If the Flash controller is enabled (i.e., the EN bit is set in the FCCTL register) prior to the expiration of this timer, then initialization of the Flash channels is delayed until the timer expires. There is no need for software to delay initialization of the Flash controller. The default value of this field corresponds to a delay of 150 uS when a 200 MHz Flash controller clock is used. There is no reason for software to ever modify this field. The behavior of the Flash controller when this field is modified is undefined.
31:18	Reserved	RO	0x0	

FTM - Flash Timing Mode

Bit Field	Field Name	Type	Default Value	Description
3:0	TMODE	RW	0x1	**Timing Mode** This field specifies the timing mode that the Flash devices are configured to use. Valid values for this field are one through five. See Table 9 on page 58 of the Micron data sheet.
31:4	Reserved	RO	0x0	

FTR - Flash Timing Read Page

Bit Field	Field Name	Type	Default Value	Description
14:0	TR	RW	0x0	**Read Page Operation Time** This field specifies the read page operation time (tR) in units of Flash controller clock cycles. See Miicron data sheet Table 42 on page 124. The value in this field should correspond to 25 us.
31:15	Reserved	RO	0x0	

FTPROG - Flash Timing Program Page

Bit Field	Field Name	Type	Default Value	Description
17:0	TPROG	RW	0x0	**Page Program Operation Time** This field specifies the page program operation time (tPROG) in units of Flash controller clock cycles. See Miicron data sheet Table 42 on page 124. The value in this field should correspond to 230 to 500 us.
31:18	Reserved	RO	0x0	

FTFEAT - Flash Timing Busy Time for Set Feature

Bit Field	Field Name	Type	Default Value	Description
8:0	TFEAT	RW	0x0	**Busy Time for Set Feature** This field specifies the busy time for a SET FEATURES and GET FEATURES (i.e., tFEAT) as well as the busy time for an interface change from asynchronous to synchronous mode (i.e., tITC) in units of Flash controller clock cycles. See Miicron data sheet Table 42 on page 124. This field should be set to the maximum of tFEAT and tITC. The value in this field should correspond 1 us.
31:9	Reserved	RO	0x0	

FTBERS - Flash Timing Block Erase Time

Bit Field	Field Name	Type	Default Value	Description
19:0	TBERS	RW	0x0	**Block Erase Operation Time** This field specifies the busy time for an erase block operation (tBERS) in units of Flash controller clock cycles. See Miicron data sheet Table 42 on page 124. The value in this field should correspond to 0.7 to 3 ms.
31:20	Reserved	RO	0x0	

FTPOR - Flash Timing Power-On Reset Time

Bit Field	Field Name	Type	Default Value	Description
19:0	TPOR	RW	0x0	**Power On Reset Time** This field specifies the power-on reset time (tPOR) in units of Flash controller clock cycles. See Miicron data sheet Table 42 on page 124. The value in this field should correspond to 1 ms.
31:20	Reserved	RO	0x0	

FTRWPOLL - Flash Timing Read and Write Polling Time

Bit Field	Field Name	Type	Default Value	Description
15:0	TRDPOLL	RW	0x0	**Read Polling Time** This field specifies read polling time in units of Flash controller clock cycles. See Miicron data sheet Table 42 on page 124. The value in this field should correspond to 1 us.
31:16	TWRPOLL	RW	0x0	**Write Polling Time** This field specifies write/program polling time in units of Flash controller clock cycles. See Miicron data sheet Table 42 on page 124. The value in this field should correspond to 50 us.

FTERPOLL - Flash Timing Erase Polling Time

Bit Field	Field Name	Type	Default Value	Description
19:0	TERPOLL	RW	0x0	**Erase Polling Time** This field specifies erase polling time in units of Flash controller clock cycles. See Miicron data sheet Table 42 on page 124. The value in this field should correspond to 1 ms.
31:20	Reserved	RO	0x0	

FTASYNC0 - Flash Timing Asynchronous 0

Bit Field	Field Name	Type	Default Value	Description
3:0	TCDH	RW	0x3	**Command and Data Hold Time** This field specifies the command and data hold time in units of Flash controller clock cycles when the Flash is in asynchronous mode. See Miicron data sheet Table 40 on page 119. This field should be set to the maximum of the following Flash parameters: tALH, tCH, tCLH, tDH and tWH.
7:5	TCDS	RW	0x7	**Command and Data Setup Time** This field specifies the command and data setup time and write pulse width in units of Flash controller clock cycles when the Flash is in asynchronous mode. See Miicron data sheet Table 40 on page 119. This field should be set to the maximum of the following Flash parameters: tALS, tCLS, tDS and tWP.
12:8	TCE	RW	0x9	**CE Setup Time** This field specifies the CE# setup time in units of Flash controller clock cycles when the Flash is in asynchronous mode. See Miicron data sheet Table 40 on page 119. The value in this field should be set to tCS - tTCDS where tTCDS is the time value selected by the corresponding field in this register.
16:13	TCRED	RW	0x4	**Command to Read Enable Delay** This field specifies the command to read enable delay in units of Flash controller clock cycles when the Flash is in asynchronous mode. See Miicron data sheet Table 40 on page 119. This field should be set to the maximum of the following Flash parameters: tAR and tCLR - 2 * Flash control clock period.
24:17	TADL	RW	0x19	**Address to Data Load Time** This field specifies the address to data load time in units of Flash controller clock cycles when the Flash is in asynchronous mode. See Miicron data sheet Table 40 on page 119. This field should be set to the tADL Flash parameter.
31:25	Reserved	RO	0x0	

FTASYNC1 - Flash Timing Asynchronous 1

Bit Field	Field Name	Type	Default Value	Description
7:0	TWHR	RW	0xF	**Write Enable High to Read Enable Low.** This field specifies the WE# high to RE# low (tWHR) time in units of Flash controller clock cycles when the Flash is in asynchronous mode. See Miicron data sheet Table 40 on page 119.
15:8	TRHW	RW	0x19	**Read High to Write Enable Low.** This field specifies theRE# high to WE# low (tRHW) time in units of Flash controller clock cycles when the Flash is in asynchronous mode. See Miicron data sheet Table 40 on page 119.

Bit Field	Field Name	Type	Default Value	Description
23:16	TRP	RW	0x6	**Read Enable Pulse Width.** This field specifies the RE# pulse width (tRP) time in units of Flash controller clock cycles when the Flash is in asynchronous mode. See Miicron data sheet Table 40 on page 119.
31:24	TREH	RW	0x3	**Read Enable High Hold Time.** This field specifies the RE# high hold time (tREH) time in units of Flash controller clock cycles when the Flash is in asynchronous mode. See Miicron data sheet Table 40 on page 119.

FTSYNC0 - Flash Timing Synchronous 0

Bit Field	Field Name	Type	Default Value	Description
3:0	TCDH	RW	0x1	**Command and Data Hold Time** This field specifies the command and data hold time in units of Flash controller clock cycles. See Miicron data sheet Table 41 on page 121. This field should be set to the maximum of the following Flash parameters: tCALH, tCAH, and tCH.
7:4	TCDS	RW	0x1	**Command and Data Setup Time** This field specifies the command and data setup time in units of Flash controller clock cycles when the Flash is in synchronous mode. See Miicron data sheet Table 41 on page 121. This field should be set to the maximum of the following Flash parameters: tCALS, and tCAS.
12:8	TCE	RW	0x4	**CE Setup Time** This field specifies the CE# setup time in units of Flash controller clock cycles when the Flash is in synchronous mode. See Miicron data sheet Table 41 on page 121. The value in this field should be set to tCS - tTCDS where tTCDS is the time value selected by the corresponding field in this register.
16:13	TCAD	RW	0x4	**Command, Address, and Data Delay** This field specifies the command, address and data delay in units of Flash controller clock cycles when the Flash is in synchronous mode. See Miicron data sheet Table 41 on page 121. This field should be set so that the Flash parameter tCAD > (TCAD - 2) * FlashControllerClockPeriod.
24:17	TADL	RW	0xD	**Address to Data Load Time** This field specifies the address to data load time in units of Flash controller clock cycles when the Flash is in synchronous mode. See Miicron data sheet Table 41 on page 121. This field should be set to the tADL Flash parameter.
31:25	Reserved	RO	0x0	

FTSYNC1 - Flash Timing Synchronous 1

Bit Field	Field Name	Type	Default Value	Description
7:0	TWHR	RW	0xA	**Command Cycle to Data Output** This field specifies the command cycle to data output time in units of Flash controller clock cycles. See Miicron data sheet Table 41 on page 121. This field should be set to the tWHR Flash parameter.
15:8	TRHW	RW	0x11	**Data Output to Command Address or Data Input** This field specifies the data output to command, address or data input time in units of Flash controller clock cycles when the Flash is in synchronous mode. See Miicron data sheet Table 41 on page 121. This field should be set to the tRHW Flash parameter.
31:16	Reserved	RO	0x0	

CH[7:0]FTSYNC2 - Channel [7:0] Flash Timing Synchronous 2

Bit Field	Field Name	Type	Default Value	Description
0	Reserved	RO	0x0	
1	EDGE	RW	0x0	**DQ and DQS Sampling Edge** This bit controls the Flash controller clock edge used to sample the DQ and DQS signals for the corresponding Flash channel. *0 (pos) Sample on the positive edge* *1 (neg) Sample on the negative edge*
2	DATATDIR	RW	0x0	**Data Tap Direction** This field controls the direction that the selected DQ data delay tap moves for the corresponding Flash channel when a one is written to the DATATSHFT field in this register. *0 (dec) Decrement delay tap* *1 (inc) Increment delay tap*
3	STRBTDIR	RW	0x0	**Strobe Tap Direction** This field controls the direction that the selected DQS strobe delay tap moves for the corresponding Flash channel when a one is written to the STRBTSHFT field in this register. *0 (dec) Decrement delay tap* *1 (inc) Increment delay tap*
7:4	Reserved	RO	0x0	
8	DATATSHFT	RW	0x0	**Data Tap Shift** Writing a one into this field shifts the DQ data delay tap for the corresponding Flash channel one position in the direction specified by the DATATDIR field in this register. The delay line has 64 tap settings numbered zero through 63. Incrementing or decrementing the tap past an end causes it to wrap to the other end.
15:9	Reserved	RO	0x0	

Bit Field	Field Name	Type	Default Value	Description
16	STRBTSHFT	RW	0x0	**Strobe Tap Shift** Writing a one into this field shifts the DQS strobe delay tap for the corresponding Flash channel one position in the direction specified by the STRBTDIR field in this register. The delay line has 64 tap settings numbered zero through 63. Incrementing or decrementing the tap past an end causes it to wrap to the other end.
31:17	Reserved	RO	0x0	

10.8.2 Page Frame Region Base Registers

BRPFDB - Buffer RAM Page Frame Data Base

Bit Field	Field Name	Type	Default Value	Description
24:0	BASEADDR	RW	0x0	**Base Address** This field contains the upper 25-bits of the 37-bit controller memory address that points to the start of the page frame data region in buffer RAM. The page frame data base address must be 4KB aligned. Therefore the bottom 12-bits are implicitly zero.
31:25	Reserved	RO	0x0	

BRPFMB - Buffer RAM Page Frame Metadata Base

Bit Field	Field Name	Type	Default Value	Description
28:0	BASEADDR	RW	0x0	**Base Address** This field contains the upper 29-bits of the 37-bit controller memory address that points to the start of the page frame metadata region in buffer RAM. The page frame metadata base address must be 256B aligned. Therefore the bottom 8-bits are implicitly zero.
31:29	Reserved	RO	0x0	

DRPFDB - DRAM Page Frame Data Base

Bit Field	Field Name	Type	Default Value	Description
24:0	BASEADDR	RW	0x0	**Base Address** This field contains the upper 25-bits of the 37-bit controller memory address that points to the start of the page frame data region in buffer RAM. The page frame data base address must be 4KB aligned. Therefore the bottom 12-bits are implicitly zero.
31:25	Reserved	RO	0x0	

DRPFMB - DRAM Page Frame Metadata Base

Bit Field	Field Name	Type	Default Value	Description
28:0	BASEADDR	RW	0x0	**Base Address** This field contains the upper 29-bits of the 37-bit controller memory address that points to the start of the page frame metadata region in buffer RAM. The page frame metadata base address must be 256B aligned. Therefore the bottom 8-bits are implicitly zero.
31:29	Reserved	RO	0x0	

10.8.3 Message Network Interface Registers

FCPQCTL[11:0] - Flash Completion Outbound Message Queue Control [11:0]

Bit Field	Field Name	Type	Default Value	Description
0	Reserved	RO	0x0	
8:1	NODEID	RW	0x0	**Inbound Message Queue Node ID** This field contains the Node ID of the inbound message queue interface (i.e., the destination node) with which this outbound message queue is associated.
13:9	RCVQ	RW	0x0	**Receive Queue** This field contains the Receive Queue ID within the inbound message queue interface with which this outbound message queue is associated.
31:14	Reserved	RO	0x0	

Error Correction Unit

11.1 Introduction

- Delaware Flash has a 4KB data page size and 224B of metadata. The page data plus metadata is split into four equal 1080B chunks.
- The ECC correction unit corrects up to 8-bits out of 1080B using a BCH ECC code.
- The ECC code requires 14B of parity per chunk.

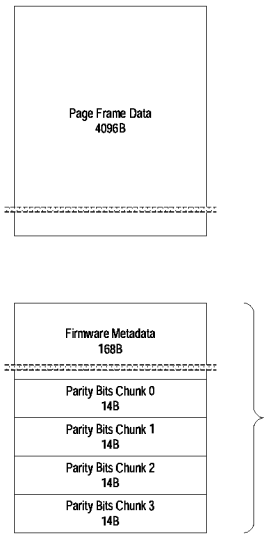


Figure 11.1 Page Frame Layout In Memory

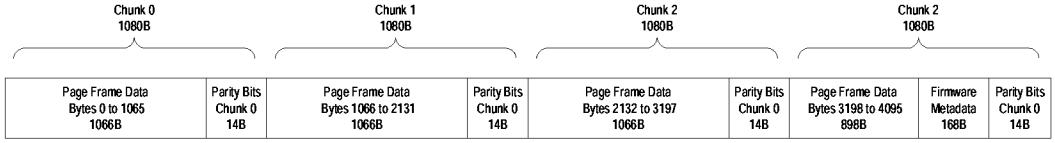


Figure 11.2 Page Frame Layout in a Flash Page

- Add note that when ECC correction is enabled, then the ECC of a page read from Flash may not be correct.

Processor Complex

12.1 Introduction

12.2 Processor Configuration

Feature	IDT 108Mini (baseline)	Delaware Processor
Xtensa ISA version	LX2.1	LX3
16-bit MAC with 40 bit Accumulator	no	no
MUL16	yes	no
MUL32	Fully Pipelined	Fully Pipelined
32 bit integer divider	yes	yes
MAC16	no	yes
Floating Point (coprocessor id 0)	no	no
CLAMPS	no	no
NSA/NSAU	yes	yes
MIN/MAX and MINU/MAXU	yes	yes
SEXT	yes	no
Boolean Registers	no	no
Number of Coprocessors (NCP)	0	0
Enable Density Instructions	yes	yes
Enable Processor ID	yes	yes
Zero-overhead loop instructions	no	yes
Synchronize instruction	yes	no
Conditional store synchronize instruction	yes	no
Count of Load/Store units	1	1
Max instruction width (bytes)	3	3
Pipeline length	5	5
Vectra LX DSP coprocessor instruction family	no	no
HiFi2 Audio Engine DSP coprocessor instruction family	no	no
Thread Pointer	no	no
Exception option	yes	yes
Interrupts enabled ?	yes	yes
Interrupt count	22	12
Int 0 type / priority level	ExtLevel / 1	Software / 1
Int 1 type / priority level	ExtLevel / 1	Timer / 1
Int 2 type / priority level	ExtLevel / 1	Write Error / 1
Int 3 type / priority level	ExtLevel / 1	ExtLevel / 2
Int 4 type / priority level	ExtLevel / 1	ExtLevel / 2
Int 5 type / priority level	ExtLevel / 1	ExtLevel / 2
Int 6 type / priority level	Timer / 1	ExtLevel / 3
Int 7 type / priority level	Software / 1	Timer / 3

Table 12.1 Delaware Tensilica Processor Core Configuration

Feature	IDT 108Mini (baseline)	Delaware Processor
Int 8 type / priority level	ExtLevel / 2	ExtLevel / 3
Int 9 type / priority level	ExtLevel / 3	Software / 3
Int 10 type / priority level	Timer / 3	ExtEdge / 4
Int 11 type / priority level	Software / 3	NMI / 6
Int 12 type / priority level	ExtLevel / 4	
Int 13 type / priority level	Timer / 5	
Int 14 type / priority level	NMI / 7	
Int 15 type / priority level	ExtEdge / 1	
Int 16 type / priority level	ExtEdge / 1	
Int 17 type / priority level	ExtEdge / 1	
Int 18 type / priority level	ExtEdge / 1	
Int 19 type / priority level	ExtEdge / 1	
Int 20 type / priority level	ExtEdge / 1	
Int 21 type / priority level	ExtEdge / 3	
High Priority Interrupts	yes	yes
Interrupt Level count	6	5
Medium Level Interrupts	yes	yes
Highest Medium Interrupt Level	3	3
Timer count	yes	yes
Timer count	3	2
Timer 0	6	1
Timer 1	10	3
Timer 2	13	
Byte ordering (endianness)	Big Endian	Little Endian
Number of physical AR registers. 16 forces Call0 ABI	32	32
Miscellaneous Special Register count	2	2
Generate exception on unaligned load/store address	exception	exception
Enable Processor Interface (PIF)	yes	yes
Configured Bus		AHBLite
Write buffer entries	4	16
Enable PIF Write Responses	no	no
Prioritize Load Before Store	no	yes
Widths of Cache and Memory Interfaces		
Width of Instruction Fetch Interface	32	32
Width of Data Memory/Cache interface	32	32
Width of PIF interface	32	32
Width of Interface to instruction cache	0	32
Instruction Cache	Not Selected	Not Selected
Instruction Cache Way Count	Not Selected	Not Selected
Instruction Cache Bytes (Size)	Not Selected	Not Selected
Instruction Cache Line Bytes	Not Selected	Not Selected
Instruction Cache Memory Error Detection	Not Selected	Not Selected
Instruction Cache Memory Error Enable	Not Selected	Not Selected
Data Cache	Not Selected	Not Selected
Data Cache Way Count	Not Selected	Not Selected
Data Cache Bytes (Size)	Not Selected	Not Selected

Table 12.1 Delaware Tensilica Processor Core Configuration

Feature	IDT 108Mini (baseline)	Delaware Processor
Data Cache Line Bytes	Not Selected	Not Selected
Data Cache Is Writeback	Not Selected	Not Selected
Data Cache Memory Error Detection	Not Selected	Not Selected
Data Cache Memory Error Enable	Not Selected	Not Selected
Debug	yes	yes
Data address breakpoint registers	2	2
Instruction address breakpoint registers	2	2
Debug interrupt level	6	5
Trace port (address trace and pipeline status)	yes	no
Add data trace	no	no
On Chip Debug(OCD)	yes	yes
Use array of 4 Debug Instruction Registers (DIRs)	yes	yes
External Debug Interrupt	yes	yes
Full scan	yes	no
Make latches transparent	no	no
MemoryProtection/MMU	Region Protection	Region Protection
System RAM start address / size	0x60000000 / 64M	0x40000000 / 1048064K
System ROM start address / size	0x50000000 / 16M	Not Selected
Inbound PIF request buffer depth	2	2
Local Memory		
Instruction RAM start address / size	0x40000000 / 128K [inbound PIF] [busy]	0x7FFA0000 / 64K [inbound PIF] [busy]
Instruction ROM start address / size	Not selected	Not selected
Data RAM [0] start address / size	0x3ffe0000 / 128K [inbound PIF] [busy]	0x7FF80000 / 64K [inbound PIF] [busy]
Data RAM [1] start address / size	0x3ffc0000 / 128K [inbound PIF] [busy]	Not Selected
Data ROM start address / size	Not selected	Not selected
XLMI start address / size	Not selected	Not selected
Vector configuration		
Reset Vector start address / size	0x50000000 / 0x300	0x7FFA0400 / 0x300
Kernel (Stacked) Exception Vector start address / size	0x40000300 / 0x38	0x7FFA0300 / 0x38
User (Program) Exception Vector start address / size	0x40000340 / 0x38	0x7FFA0340 / 0x38
Double Exception Vector start address / size	0x400003c0 / 0x40	0x7FFA03c0 / 0x40
Window Register Overflow Vector start address / size	0x40000000 / 0x178	0x7FFA0000 / 0x180
Level 2 Interrupt Vector start address / size	0x40000180 / 0x38	0x7FFA0188 / 0x38
Level 3 Interrupt Vector start address / size	0x400001c0 / 0x38	0x7FFA01c8 / 0x38
Level 4 Interrupt Vector start address / size	0x40000200 / 0x38	0x7FFA0208 / 0x38
Level 5 Interrupt Vector start address / size	0x40000240 / 0x38	0x7FFA0248 / 0x38
Level 6 Interrupt Vector start address / size	0x40000280 / 0x38	0x7FFA0288 / 0x38 (NMI)
Level 7 Interrupt Vector start address / size	0x400002c0 / 0x38	Not selected
Relocatable Vectors	yes	Not selected
Selected Static Vector set	Primary	Not selected
Primary Static Vector Group Base Address	0x50000000	Not selected
Alternate Static Vector Group Base Address	0x40000400	Not selected
Alternate Reset Vector Address	0x40000400	Not selected
Default Dynamic Vector Group Base Address (VECBASE)	0x40000000	Not selected

Table 12.1 Delaware Tensilica Processor Core Configuration

Feature	IDT 108Mini (baseline)	Delaware Processor
Target & CAD options		FPGA
Functional Unit Clock Gating	yes	no
Global Clock Gating	yes	no
Register file implementation block	Flip-flops	Flip-flops
Reset All Flops	yes	yes
Software Target Options		
Xtensa Tools should use Extended L32R	no	no
Software ABI	windowed	windowed
C Libraries	newlib	newlib
Miscellaneous Features		
32 bit input and output ports	yes	yes
32 bit input and output queue interfaces	no	no

Table 12.1 Delaware Tensilica Processor Core Configuration

12.3 Interrupts

Interrupt	Type	Level	Priority	Description
0	Software	1	Low	Software Interrupt
1	Timer	1	Medium	Timer 0 Interrupt
2	Write Error	1	Medium	Write Error Interrupt
3	External Level	2	Medium	Outbound Message Queue Interrupt
4	External Level	2	Medium	Inbound Message Queue Interrupt
5	External Level	2	Medium	Memory Network Interrupt
6	External Level	3	Medium	Attention Interrupt
7	Timer	3	Medium	Timer 1 Interrupt
8	External Level	3	Medium	reserved (other fatal error)
9	Software	3	Medium	Software Interrupt
10	External Edge	4	High	Power Down Interrupt
11	External Edge	NMI / 6	High	Power Down Non-Maskable Interrupt

Table 12.2 Processor Interrupts

12.4 GPIO Ports

GPIO Bit	Control Processor	Processor Node	Description
0		X	Control Processor Attention Interrupt
1	X		Processor Node (x,y) Attention Interrupt
2	X		Processor Node (x,y) Attention Interrupt
3	X		Processor Node (x,y) Attention Interrupt
4	X		Processor Node (x,y) Attention Interrupt
5	X		Processor Node (x,y) Attention Interrupt
6	X		Processor Node (x,y) Attention Interrupt

Table 12.3 GPIO Input Port

GPIO Bit	Control Processor	Processor Node	Description
7	X		Processor Node (x,y) Attention Interrupt
8	X		Processor Node (x,y) Attention Interrupt
9	X		Processor Node (x,y) Attention Interrupt
10	X		Processor Node (x,y) Attention Interrupt
11	X		Processor Node (x,y) Attention Interrupt
12			reserved
13			reserved
14			reserved
15			reserved
16			reserved
17			reserved
18	X		Error Correction Unit Attention Interrupt
19	X		Buffer RAM Attention Interrupt
20	X		DRAM Interface Attention Interrupt
21	X		Host Interface Attention Interrupt
22	X		Flash Controller Attention Interrupt
23	X		SPI Data Input (SPI_DI) Pin
24	X		Bond Option 0 (BOPTION[0]) Pin
25	X		Bond Option 1 (BOPTION[1]) Pin
26	X		GPIO 0 (GPIO[0]) Pin
27	X		GPIO 1 (GPIO[1]) Pin
28	X		GPIO 2 (GPIO[2]) Pin
29	X		GPIO 3 (GPIO[3]) Pin
30	X		Boot Mode 0 (BMODE[0]) Pin
31	X		Boot Mode 1 (BMODE[1]) Pin

Table 12.3 GPIO Input Port

GPIO Bit	Control Processor	Processor Node	Description
0			
1			
2			
3			
4			
5			
6			
7			
8	X		SPI Clock (SPI_CLK) Pin
9	X		SPI Data Output (SPI_DO) Pin
10	X		SPI ROM Chip Select (SPI__ROM_CSN) Pin
11	X		SPI Chip Select 0 (SPI_CSN0) Pin
12	X		SPI Chip Select 1 (SPI_CSN1) Pin
13	X		SPI Chip Select 2 (SPI_CSN2) Pin
14			

Table 12.4 GPIO Output Port

GPIO Bit	Control Processor	Processor Node	Description
15			
16			
17			
18			
19			
20	X		GPIO 0 (GPIO[0]) Pin Output Control
21	X		GPIO 1 (GPIO[1]) Pin Output Control
22	X		GPIO 2 (GPIO[2]) Pin Output Control
23	X		GPIO 3 (GPIO[3]) Pin Output Control
24	X		Boot Mode 0 (BMODE[0]) Pin Output Control
25	X		Boot Mode 1 (BMODE[1]) Pin Output Control
26	X		GPIO 0 (GPIO[0]) Pin Output Value
27	X		GPIO 1 (GPIO[1]) Pin Output Value
28	X		GPIO 2 (GPIO[2]) Pin Output Value
29	X		GPIO 3 (GPIO[3]) Pin Output Value
30	X		Boot Mode 0 (BMODE[0]) Pin Output Value
31	X		Boot Mode 1 (BMODE[1]) Pin Output Value

Table 12.4 GPIO Output Port

12.5 Processor Address Map

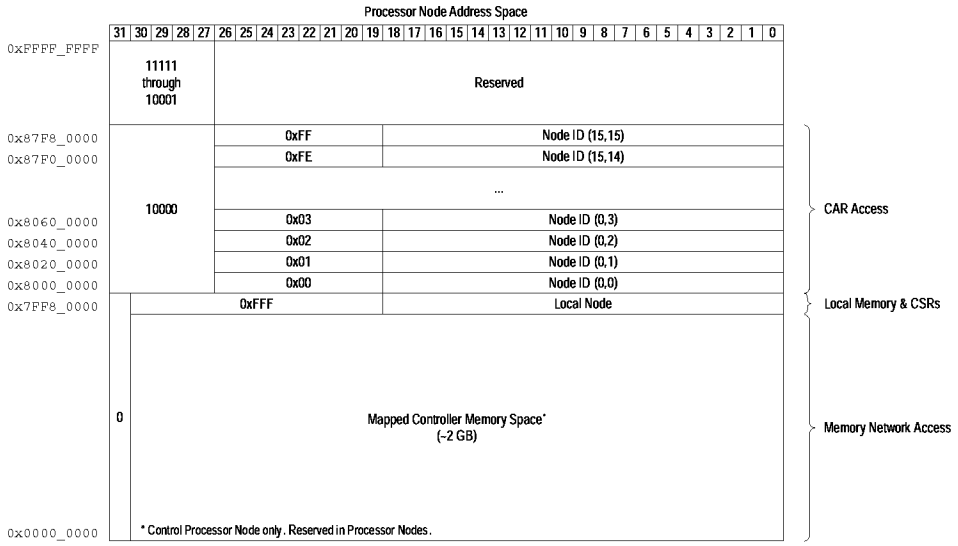


Figure 12.1 Processor Address Map

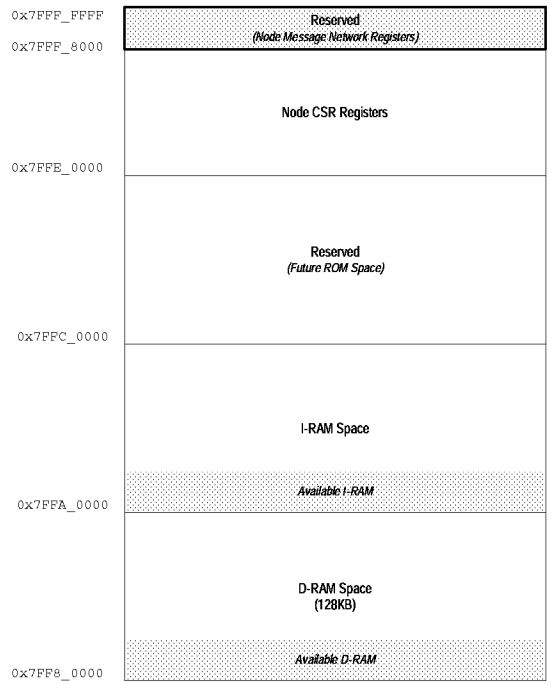


Figure 12.2 Local Node Processor Address Map

- I-RAM and D-RAM is referred to as local node processor memory.

12.6 64-bit Register File and Instructions

- A node processor implements the Tensilica LX3 32-bit instructions set with the 64-bit extensions outlined in this section.
- In addition to the architectural registers defined in the LX3 architecture, a node processor has a four entry 64-bit register file. A C data type (ctype) is defined for the register file so that the Tensilica C/C++ compiler may automatically perform memory loads/stores and register allocation.
  - R64 refers to the 64-bit register file (R64[0] is register zero, R64[1] is register one, and so on).
  - The C data type for a 64-bit register is reg64. A re64 must have DWord alignment in memory.
  - TIE declaration:
    - regfile r64 64 4 R64
    - ctype reg64 64 32 r64
  - Automatic type conversion is performed by the Tensilica C/C++ complier for the built-in types below.
    - int32 (C data type: int)
    - uint32 (C data type: unsigned int)
    - *int32 (C data type: pointer to int)
    - *uint32 (C data type: pointer to unsigned int)
    - *int64 (C data type: pointer to long long)

- *uint64 (C data type: pointer to unsigned long long)

Instruction	Description
Memory Loads	
L_R64_U32	Memory load of an unsigned 32-bit value into an R64 register
L_R64_S32	Memory load of a signed 32-bit value into an R64 register
L_R64U_32	Memory load of a 32-bit value into upper 32-bits of an R64 register
Memory Stores	
S_R64L_32	Memory store of the lower 32-bits of an R64 register into a 32-bit memory location
S_R64U_32	Memory store of the upper 32-bits of an R64 register into a 32-bit memory location
S_ZERO_32	Memory store of all zeros into a 32-bit memory location
S_SEAR_32	Memory store of replicated AR register bit 31 value into a 32-bit memory location
Register Moves	
M_R64_R64	Move contents of one R64 register into another R64 register
M_R64_UAR	Move unsigned contents of a 32-bit AR register into a R64 register
M_R64_SAR	Move signed contents of a 32-bit AR register into an R64 register
Flash Page Pointer Conversions	
MlpToR64	Convert memory line pointer in an AR register to a controller address in an R64 register
R64ToMlp	Convert a controller address in an R64 register to a memory line pointer address in an AR register
Arithmetic Operations	
AddR64R64	Add two R64 register values and store result in an R64 register
AddR64Ar	Add R64 register value to a signed AR register value and store result in an R64 register
AddR64UAr	Add R64 register value to an unsigned AR register value and store result in an R64 register
SubR64R64	Subtract two R64 register values and store result in an R64 register

Table 12.5 64-bit Register File Instructions

- Instructions that operate on the 64-bit register file are summarized in Table 12.5.

Instruction	Description
reg64_loadi	reg64 memory load immediate instruction
reg64_storei	reg64 memory store immediate instruction
reg64_move	reg64 register to register move
int32_rtom_reg64	Convert int32 in AR[1] to reg64 in memory
int32_mtor_reg64	Convert int32 in memory to reg64 in R64[2]
int32_rtor_reg64	Convert unit32 in AR to reg64 in R64
unit32_rtom_reg64	Convert unit32 in AR to reg64 in memory

Table 12.6 64-bit Register File Prototypes

Instruction	Description
unit32_mtor_reg64	Convert unit32 in memory to reg64 in R64
uint32_rtor_reg64	Convert unit32 in AR to reg64 in R64
uint64_mtor_reg64	Convert uint64 in memory to reg64 in R64
reg64_rtom_uint64	Convert reg64 in R64 to uint64 in memory
int64_mtor_reg64	Convert int64 in memory to reg64 in R64
reg64_rtom_int64	Convert reg64 in R64 to int64 in memory

Table 12.6 64-bit Register File Prototypes

1. AR is an abbreviation for Architectural Register
2. R64 corresponds to the 64-bit register file

- The function prototypes described in Table 12.6 are provided to allow the compiler to operate directly on a Reg64 type. The implementation of these prototypes using the instructions shown in Table 12.5 is described in Section 12.6.2, "Reg64 Type Prototypes" on page 12-13.

12.6.1 64-bit Instructions

12.6.1.1 L_R64_U32

- Memory load of an unsigned 32-bit value into an R64 register.
- L_R64_U32(out R64 Val, in AR Addr, imm Offset)
  - Val - Value loaded from memory.
  - Addr - 32-bit pointer address.
  - Offset - immediate address offset value
    - The 8-bit immediate value has a range of 0 to 255
  - Operation
    - Vaddr ← AR[Addr] + Offset
    - R64[Val] ← (0^{32} | | Load32(Vaddr))

12.6.1.2 L_R64_S32

- Memory load of a signed 32-bit value into an R64 register.
- L_R64_U32(out R64 Val, in AR Addr, imm Offset)
  - Val - Value loaded from memory.
  - Addr - 32-bit pointer address.
  - Offset - immediate address offset value
    - The 8-bit immediate value has a range of 0 to 255
  - Operation
    - Vaddr ← AR[Addr] + Offset
    - R64[Val] ← ($Load32(Vaddr)_{31}^{32}$ | | Load32(Vaddr))

12.6.1.3 L_R64U_32

- Memory load of a 32-bit value in memory into the upper 32-bits of an R64 register. The lower 32-bits are preserved.
- L_R64_U32(inout R64 Val, in AR Addr, imm Offset)
  - Val - Value loaded from memory.
  - Addr - 32-bit pointer address.
  - Offset - immediate address offset value
    - The 8-bit immediate value has a range of 0 to 255
  - Operation
    - Vaddr ← AR[Addr] + Offset
    - R64[Val] ← (Load32(Vaddr) || $R64[Val]_{31..0}$)

12.6.1.4 S_R64L_32

- Store lower 32-bits of an R64 register into a 32-bit memory location.
- S_R64L_32(in R64 Val, in AR Addr, imm Offset)
  - Val - Value containing data to store to memory.
  - Addr - 32-bit pointer address.
  - Offset - immediate address offset value
    - The 8-bit immediate value has a range of 0 to 255
  - Operation
    - Vaddr ← AR[Addr] + Offset
    - Store32(Vaddr) ← $R64[Val]_{31..0}$

12.6.1.5 S_R64U_32

- Store upper 32-bits of an R64 register into a 32-bit memory location.
- S_R64U_32(in R64 Val, in AR Addr, imm Offset)
  - Val - Value containing data to store to memory.
  - Addr - 32-bit pointer address.
  - Offset - immediate address offset value
    - The 8-bit immediate value has a range of 0 to 255
  - Operation
    - Vaddr ← AR[Addr] + Offset
    - Store32(Vaddr) ← $R64[Val]_{63..32}$

12.6.1.6 S_ZERO_32

- Store zero into a 32-bit memory location.
- S_R64U_32(in AR Addr, imm Offset)
  - Addr - 32-bit pointer address.
  - Offset - immediate address offset value
    - The 8-bit immediate value has a range of 0 to 255

- Operation
  - Vaddr ← AR[Addr] + Offset
  - Store32(Vaddr) ← 0^{32}

12.6.1.7 S_SEAR_32

- Replicate AR register sign bit 32 times and store this value into a 32-bit memory location.
- S_SEAR_32(in AR Val, in AR Addr, imm Offset)
  - Val - Value containing data to store to memory.
  - Addr - 32-bit pointer address.
  - Offset - immediate address offset value
    - The 8-bit immediate value has a range of 0 to 255
  - Operation
    - Vaddr ← AR[Addr] + Offset
    - Store32(Vaddr) ← $AR[Val]_{31}{}^{32}$

12.6.1.8 M_R64_R64

- R64 register to register move.
- M_R64_R64(out R64 RegOut, in R64 RegIn)
  - RegOut - Destination R64 register
  - RegIn - Source R64 register
  - Operation
    - R64[RegOut] ← R64[RegIn]

12.6.1.9 M_R64_UAR

- Move unsigned 32-bit value in an AR register to an R64 register.
- M_R64_UAR(out R64 RegOut, in AR RegIn)
  - RegOut - Destination R64 register
  - RegIn - Source R64 register
  - Operation
    - R64[RegOut] ← (0^{32} || AR[RegIn])

12.6.1.10 M_R64_SAR

- Move signed 32-bit value in an AR register to an R64 register.
- M_R64_UAR(out R64 RegOut, in AR RegIn)
  - RegOut - Destination R64 register
  - RegIn - Source R64 register
  - Operation
    - R64[RegOut] ← ($AR[RegIn]_{31}{}^{32}$ || AR[RegIn])

12.6.1.11 MlpToR64

- Convert memory line pointer in an AR register to a controller address in an R64 register.
- MlpToR64(out R64 Addr, in AR Mlp)
  - Addr - Controller address (byte address)
  - Mlp - Memory line pointer (pointer to a 64B aligned memory line)
  - Operation:
    - Shift the Flash page pointer left by 5 bits and place it the result in the R64 register.
    - R64[Addr] $\leftarrow$ (0^{27} || AR[Mlp] || 0^{5})

12.6.1.12 R64ToMlp

- Convert controller address in an R64 register to a memory line pointer in an AR register.
- R64ToMlp(out AR Mlp, in R64 Addr)
  - Mlp - Memory line pointer (pointer to a 64B aligned memory line)
  - Addr - Controller address (byte address)
  - Operation:
    - Shift the R64 register value right five bits and place the bottom 32-bits of the resulting value in an AR register.
    - AR[Mlp] $\leftarrow$ $AR[Addr]_{36..5}$

12.6.1.13 AddR64R64

- Add two R64 registers and store result in an R64 register.
- AddR64R64(out R64 Result, in R64 Op1, in R64 Op2)
  - Result - Addition result value
  - Op1 - Operand 1
  - Op2 - Operand 2
  - Operation:
    - Add two R64 register values
    - R64[Result] $\leftarrow$ R64[Op1] + R64[Op2]

12.6.1.14 AddR64Ar

- Add value in an R64 register with a signed value in an AR register and store result in an R64 register.
- AddR64Ar(out R64 Result, in R64 Op1, in AR Op2)
  - Result - Addition result value
  - Op1 - Operand 1
  - Op2 - Operand 2 (signed AR)
  - Operation:
    - Sign extend Op2 AR register value to create a 64-bit value, add it to the R64 register and place the result in a an R64 register.
    - R64[Result] $\leftarrow$ R64[Op1] + ($AR[Op2]_{31}^{32}$ || AR[Op2])

12.6.1.15 AddR64UAr

- Add value in an R64 register with an unsigned value in an AR register and store result in an R64 register.
- AddR64UAR(out R64 Result, in R64 Op1, in AR Op2)
  - Result - Addition result value
  - Op1 - Operand 1
  - Op2 - Operand 2 (unsigned AR)
  - Operation:
    - R64[Result] ← R64[Op1] + (0^{32} | | AR[Op2])

12.6.1.16 SubR64R64

- Subtract two R64 registers and place the result in an R64 register.
- R64SubR64R64(out R64 Result, in R64 Op1, in R64 Op2)
  - Result - Subtraction result value
  - Op1 - Operand 1
  - Op2 - Operand 2
  - Operation:
    - Add two R64 register values
    - R64[Result] ← R64[Op1] - R64[Op2]

12.6.2 Reg64 Type Prototypes

12.6.2.1 Reg64_loadi

```
proto reg64_loadi(out reg64 v, in reg64 *p, in immediate o} {} {
    L_R64_U32 v, p, o;
    L_R64U_32 v, p, o + 4;
}
```

12.6.2.2 Reg64_storei

```
proto reg64_storei(in reg64 v, in reg64 *p, in immediate o} {} {
    S_R64L_32 v, p, o;
    S_R62U_32 v, p, o + 4;
}
```

12.6.2.3 Reg64_move

```
proto reg64_storei(out reg64 v, in reg64 vs} {} {
    M_R64_R64 v, vs;
}
```

12.6.2.4 Int32_rtom_reg64

```
proto int32_rtom_reg64(in reg32 v, in reg64 *p, in immediate o} {} {
    S32I v, p, o;
    S_SEAR_32 p, o + 4;
}
```

12.6.2.5 Int32_mtor_reg64

```
proto int32_mtor_reg64(out reg64 v, in reg32 *p, in immediate o} {} {
    L_R64_S32 v, p, o;
}
```

12.6.2.6 Int32_rtor_reg64

```
proto int32_rtor_reg64(out reg64 v, in reg32 vs} {} {
    M_R64_SAR v, vs;
}
```

12.6.2.7 Uint32_rtom_reg64

```
proto uint32_rtom_reg64(in reg32 v, in reg64 *p, in immediate o} {} {
    S32I v, p, o;
    S_ZERO_32 v, p, o + 4;
}
```

12.6.2.8 Uint32_mtor_reg64

```
proto uint32_mtor_reg64(out reg64 v, in reg32 *p, in immediate o} {} {
    L_R64_U32 v, p, o;
}
```

12.6.2.9 Uint32_rtor_reg64

```
proto uint32_rtor_reg64(out reg64 v, in reg32 vs} {} {
    M_R64_UAR v, vs;
}
```

12.6.2.10 Uint64_mtor_reg64

```
proto uint64_mtor_reg64(out reg64 v, in uint64 *p, in immediate o} {} {
    L_R64_U32 v, p, o;
    L_R64U_32 v, p, o + 4;
}
```

12.6.2.11 Reg64_rtom_uint64

```
proto reg64_rtom_uint64(in reg64 v, in uint64 *p, in immediate o} {} {
    S_R64L_32 v, p, o;
    S_R62U_32 v, p, o + 4;
}
```

12.6.2.12 Int64_mtor_reg64

```
proto int64_mtor_reg64(out reg64 v, in int64 *p, in immediate o} {} {
    L_R64_U32 v, p, o;
    L_R64U_32 v, p, o + 4;
}
```

12.6.2.13 Reg64_rtom_int64

```
proto reg64_rtom_int64(in reg64 v, in int64 *p, in immediate o} {} {
    S_R64L_32 v, p, o;
    S_R62U_32 v, p, o + 4;
}
```

12.7 Message Network Interface

- A processor node implements 15 inbound and 15 outbound message queues. These queues are labeled InMsgQ[15:1] and OutMsgQ[15:1].

- Inbound message queue 0 (InMsgQ[0]) is used to return memory network interface command status.
- Outbound message queue 0 does not exist.

- The processor's data RAM (D-RAM) is used as message queue memory for all message queues.

12.7.1 Outbound Message Queues

- Each of the 15 outbound message queues has an associated free queue and the state shown in Table 12.7.

State Field	Size (Bits)	Description
OutMsgQNode	8	**Outbond Message Queue Node ID** This state field contains the Node ID containing the inbound message queue interface (i.e., the destination node) with which this outbound message queue is associated.
OutMsgQRcvQ	5	**Outbond Message Queue Receive Queue ID** This state field contains the Receive Queue ID within the inbound message queue interface with which this outbound message queue is associated.
OutMsgQMsk	1	**Outbond Message Queue Interrupt Mask** An interrupt may be generated when the free queue associated with the outbond message queue is not empty. This state bit is an interrupt mask bit associated with the outbound message queue. Interrupts from this outbound message queue are masked when this bit is set.

Table 12.7 Outbound Message Queue State

- Outbound message queue operations are performed using the custom instructions listed in Table 12.8. These instructions are described in detail in sections 12.7.1.1 through 12.7.1.8.

Instruction	Description
SendMsg	Send message data
PostMsg	Post message
GetFreeMsgBuf	Get a free message buffer pointer
OutMsgQSts	Get free queue status of all outbound message queues
WrOutMsgQNode	Write OutMsgQNode (Outbond Message Queue Node ID) state
RdOutMsgQNode	Read OutMsgQNode (Outbond Message Queue Node ID) state
WrOutMsgQRcvQ	Write OutMsgQRcvQ state (Outbond Message Queue Receive Queue ID) state
RdOutMsgQRcvQ	Read OutMsgQRcvQ state (Outbond Message Queue Receive Queue ID) state
WrOutMsgQMsk	Write OutMsgQMsk (Outbond Message Queue Interrupt Mask) state
RdOutMsgQMsk	Read OutMsgQMsk (Outbond Message Queue Interrupt Mask) state

Table 12.8 Outbound Message Queue Instruction Summary

12.7.1.1 Send Message (SendMsg)

- Send a message network data packet. If the message network is congested (i.e., the message network output buffer is full), then the instruction blocks until the packet is sent.
- SendMsgn(in AR OutMsgQ, inout AR NodeAddr, in AR Data)
  - ° OutMsgQ - ID of the outbound message queue on which this instruction operates.
    - Selecting outbound message queue 0 produces undefined results
    - Selecting an umimplemented outbound message queue ID produces undefined results
  - ° NodeAddr - Destination node byte address where the data should be written. After the instruction completions NodeAddr is incremented by four (i.e., one DWord).
    - Bits zero through 18 contain the destination node byte address and are incremented by four. Addition carries are not propagated outside of this field.
    - Bits 19 through 31 are ignored by the instruction and are not modified by the instruction.
  - ° Data - 32-bit message data value
- Table 12.9 shows the message network packet that is generated when this instruction executed.

Packet Field	Packet Field Value
Node ID	OutMsgQNode state corresponding to OutMsgQ
Node Address	NodeAddr[18:2] (instruction parameter)
Data	Data (instruction parameter)

Table 12.9 Message Network Packet Generated by a SendMsg Instruction

12.7.1.2 Post Message (PostMsg)

- Post a message network buffer pointer. If the message network is congested (i.e., the message network output buffer is full), then the instruction blocks until the packet is sent.
- PostMsg(in AR OutMsgQ, in AR MsgBufAddr)
  - ° OutMsgQ - ID of the outbound message queue on which this instruction operates.
    - Selecting outbound message queue 0 produces undefined results
    - Selecting an umimplemented outbound message queue ID produces undefined results
  - ° MsgBufAddr - Node byte address pointing to the message buffer containing the posted message.
    - Bits zero through 18 contain the destination node byte address
    - Bits 19 through 31 are ignored by the instruction and are not modified by the instruction.

Packet Field	Packet Field Value
Node ID	OutMsgQNode state corresponding to OutMsgQ

Table 12.10 Message Network Packet Generated by a PostMsg Instruction

Packet Field	Packet Field Value
Node Address	(0x7 FF80 >> 2) OutMsgQRcvQ state corresponding to OutMsgQ
Data[18:0] (Node Address)	MsgBufAddr[18:0] (instruction parameter)
Data[23:19] (Queue ID)	OutMsgQ[4:0] (instruction parameter)
Data[31:24] (Source Node ID)	Node ID of this node

Table 12.10 Message Network Packet Generated by a PostMsg Instruction

12.7.1.3 Get Free Message Buffer (GetFreeMsgBuf)

- Get a message buffer pointer from the free queue associated with this outbound message queue if one is available. This instruction is non-blocking.
- GetFreeMsgBuf(out AR MsgBufAddr, In OutMsgQ)
  - NodeAddr - Message buffer byte address dequeued from the free queue if the free queue is not empty. If the free queue is empty, then zero is returned.
  - OutMsgQ - ID of the outbound message queue on which this instruction operates.

12.7.1.4 Outbound Message Queue Status (OutMsgQSts)

- Return the status of all of the free queues.
- OutMsgQSts(out AR Status)
  - Status - Each of the bottom 16-bits in this register represent a free queue associated with an outbound message queue. A bit is set if the corresponding free queue is not empty. The top 16-bits are always zero.
    - 0 - Free queue is empty
    - 1 - Free queue is not empty

12.7.1.5 Write Outbound Message Queue Node (WrOutMsgQNode)

- Make it array/table NAMExx
- Write OutMsgQNode state
- WrOutMsgQNode(in AR NodeID, out ST OutMsgQNode)
  - Node ID - New value of OutMsgQNode state

12.7.1.6 Read Outbound Message Queue Node (RdOutMsgQNode)

- Read OutMsgQNode state
- RdOutMsgQNode(out AR NodeID, out ST OutMsgQNode)
  - Node ID - Value of OutMsgQNode state

12.7.1.7 Write Outbound Message Queue Receive Queue (WrOutMsgQRcvQ)

- Write OutMsgQRcvQ state
- WrOutMsgQNode(in AR RcvQ, out ST OutMsgQRcvQ)
  - RcvQ - New value of OutMsgQRcvQ state

12.7.1.8 Read Outbound Message Queue Receive Queue (RdOutMsgQRcvQ)

- Read OutMsgQRcvQ state

- RdOutMsgQRcvQ(out AR RcvQ, out ST OutMsgQRcvQ)
  - RcvQ - Value of OutMsgQNode state

12.7.1.9 Write Outbound Message Queue Interrupt Mask (WrOutMsgQMsk)

- Write OutMsgQMsk state
- WrOutMsgQMsk(in AR Msk, out ST OutMsgQMsk)
  - Msk - New value of OutMsgQMsk state

12.7.1.10 Read Outbound Message Queue Receive Queue (RdOutMsgQMsk)

- Read OutMsgQMsk state
- RdOutMsgQMsk(out AR Msk, out ST OutMsgQMsk)
  - Msk - Value of OutMsgQMsk state

12.7.2 Inbound Message Network Interface

- Each of the 15 inbound message queues has an associated receive queue and the state shown in Table 12.7.

State Field	Size (Bits)	Description
InMsgQMsk	1	**Inbound Message Queue Interrupt Mask** An interrupt may be generated when the receive queue associated when the inbound message queue is not empty. This state bit is an interrupt mask bit associated with the inbound message queue. Interrupts from this inbound message queue are masked when this bit is set.

Table 12.11 Outbound Message Queue State

- Inbound message queue operations are performed using the custom instructions listed in Table 12.8. These instructions are described in detail in sections 12.7.2.1 through 12.7.2.6.

Instruction	Description
GetMsg	Get a posted message from a specific inbound message queue
GetNextMsgQ	Get the message queue ID of a message queue that has receive data
FreeMsgBufPtr	Free a message buffer pointer
InMsgQSts	Get receive queue status of all inbound message queues
WrInMsgQMsk	Write InMsgQMsk (Inbound Message Queue Interrupt Mask) state
RdInMsgQMsk	Read InMsgQMsk (Inbound Message Queue Interrupt Mask) state

Table 12.12 Outbound Message Queue Instruction Summary

12.7.2.1 Get Message (GetMsg)

- Get a message buffer pointer from the receive queue associated with the inbound message queue if the receive queue is not empty. This instruction is non-blocking.
- GetRcvMsg(out AR MsgBufPtr, in AR InMsgQ)

- MsgBufPtr - Message buffer pointer dequeued from the receive queue if the receive queue is not empty. If the receive queue is empty, then a value of zero is returned.
- InMsgQ - ID of the inbound message queue on which this instruction operates.
  - Inbound message queue 0 returns memory network command status
  - Selecting an umimplemented inbound message queue ID produces undefined results

12.7.2.2 Get Next Message Queue (GetNextMsgQ)

- Get the inbound message queue ID corresponding to an unmasked non-empty receive queue. Hardware ensures that non-empty receive queues are serviced in a fair manner. This instruction is non-blocking.
  - This instruction also returns status for inbound message queue 0 (i.e., memory network interface instruction status)
  - Fairness is only guaranteed if the mask (InMsgQMask) remains constant. Hardware does not ensure fairness if this instruction is executed with different mask (InMsgQ-Mask) values.
- GetAnyMsg(in AR InMsgQMask, out AR Status, out AR InMsgQ)
  - Status - Return status indicating if an unmasked non-empty receive queue exists.
    - 0 - All unmasked receive queues are empty. The value in InMsgQ return value is undefined.
    - 1 - An unmasked non-empty receive queue exists and the corresponding message queue ID is returned in InMsgqQ.
  - InMsgQ - Inbound message queue ID that corresponds to an unmasked non-empty receive queue. If all unmasked receive queues are empty, then the value is unchanged.

12.7.2.3 Free Message Buffer Pointer (FreeMsgBufPtr)

- Free a message buffer pointer by writing it to the free queue of the associated outbound message queue. If the message network is congested (i.e., the message network output buffer is full), then the instruction blocks until the packet is sent.
- FreeMsgBufPtr(in MsgBufPtr)
  - MsgBufPtr - Message buffer pointer to be freed.
    - MsgBufPtr[18:0] contain the node address of the message buffer
    - MsgBufPtr[23:19] Contain the outbound message queue ID
    - MsgBufPtr[31:24] Contain the source node ID

Packet Field	Packet Field Value
Node ID	MsgBufPtr[31:24]
Node Address	(0x7_FF00 >> 2) | (MsgBufPtr[23:19] << 2)
Data[18:0] (Node Address)	MsgBufPtr[31:24]
Data[23:19] (Queue ID)	0x0
Data[31:24] (Node ID)	0x0

Table 12.13 Message Network Packet Generated by a PostMsg Instruction

12.7.2.4 Inbound Message Queue Status (InMsgQSts)

- Return the status of all of the receive queues.
- InMsgQSts(out AR Status)
  - Status - Each of the bottom 16-bits in this register represent a receive queue associated with an inbound message queue. A bit is set if the corresponding receive queue is not empty. The top 16-bits of the register are always zero.
    - 0 - Receive queue is empty
    - 1 - Receive queue is not empty

12.7.2.5 Write Inbound Message Queue Interrupt Mask (WrInMsgQMsk)

- Write InMsgQMsk state
- WrInMsgQMsk(in AR Msk, out ST InMsgQMsk)
  - Msk - New value of InMsgQMsk state

12.7.2.6 Read Inbound Message Queue Receive Queue (RdInMsgQMsk)

- Read InMsgQMsk state
- RdInMsgQMsk(out AR Msk, out ST InMsgQMsk)
  - Msk - Value of InMsgQMsk state

12.7.3 Message Queue Helper Instructions

12.7.3.1 Get Message Pointer (GetMsgPtr)

- This instruction takes a message buffer pointer as a parameter and returns a local node processor address pointer to the message buffer. This is a pointer to the processor's local D-MEM where the message is located.
- GetMsgPtr(out AR MsgPtr, in AR MsgBufPtr)
  - MsgBufPtr - Message buffer pointer.
  - MsgPtr - Local node processor address pointer to the message buffer.
    - MsgPtr = 0x7FF8_0000 | MsgBufPtr[18:0]

12.8 Memory Network Interface

- The processor interfaces to two distinct memory regions: local node processor memory (I-RAM and D-RAM) and controller memory.
  - I-RAM and D-RAM in local node processor memory is directly addressable by the processor.
  - Controller memory is not in the processor's memory map and is not directly accessible by the processer. Controller memory is accessed using the memory network interface instructions summarized in Table 12.14.

Instruction	Description
MoveIn	Move data from controller memory to processor D-RAM

Table 12.14 Memory Network Interface Instruction Summary

Instruction	Description
MoveOut	Move data from processor D-RAM to controller memory
MoveInAndLock	Lock data structure in controller memory and move data from controller memory to processor D-RAM.
Xfer	Transfer data in controller memory (i.e., between buffer RAM and DRAM)
MemAdd	Atomic add of a 32-bit value to a DWord in controller memory.
MemXor	Atomic XOR of a 32-bit value with a DWord in controller memory.
FreeMemTag	Free a Memory Tag.

Table 12.14 Memory Network Interface Instruction Summary

- Since local node processor memory operates in the traditional manner, the remainder of this section focuses on memory network interface instructions used to access controller memory.
  - To read a value from controller memory the processor must move a copy of the data into D-RAM and read the value from local memory (i.e., D-DRAM). This is performed using a MoveIn instruction.
  - To write a value to controller memory the processor must create a local copy of the data in D-RAM and copy the value to controller memory. This is performed using a MoveOut instruction.
  - Since the Flash controller contains multiple processor, a mechanism must exist for mutual exclusion. This is accomplished using the MoveInAndLock instruction. This instruction reads a data quantity from controller memory, sets a lock bit in controller memory, and returns the data value as well as the initial lock bit state to the processor. This is is performed as an atomic operation and may be used as an extended atomic test and set.
  - The Xfer command allows data to be transferred between buffer RAM and DRAM. This may be viewed as a short DMA operation.
  - The MemAdd instruction provides an atomic add instruction that operates directly on controller memory. The instruction reads a DWord from controller memory, adds a 32-bit value, and writes the new value back to controller memory. This is performed as an atomic operation.
  - The MemXor instruction provides an atomic XOR instruction that operates directly on controller memory. The instruction reads a DWord from controller memory, XORs a 32-bit value, and writes the new value back to controller memory. This is performed as an atomic operation.
- Memory network interface instructions may be executed in a posted or non-posted manner.
  - A posted memory network interface instruction is one in which there is no indication to software that the operation has completed execution in Buffer RAM or DRAM. Posted memory network interface instructions are blocking.
    - Posted memory network interface instructions block if the memory network is congested and the instruction cannot be accepted by the memory network interface.
    - Posted memory network interface instructions block when it is unsafe to access D-RAM.

- A MoveOut instruction reads values out of D-RAM and writes it to controller memory. It is unsafe to modify these D-RAM values while the instruction is reading them. Therefore, executed a MoveOut instruction in posted manner causes the instruction to block until the data has been read from D-RAM.

° A non-posted memory network instruction is one that provides an indication to software when it completes.

- When a non-posted memory network instruction is issued by the processor, an 8-bit tag is returned.

° A value of 0xFF indicates that the memory network interface command FIFO is full or that there are no remaining tags. The instruction must be reissued. All other tag values indicate that the instruction has been accepted by the memory network interface.

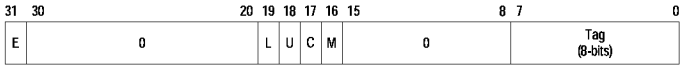


Figure 12.3 Memory Network Interface Message Buffer Pointer Format

Field	Description
TAG	Tag. This field contains the tag value returned for the corresponding non-posted memory network interface instruction.
M	Malformed Request. This bit is set to indicate that a malformed request error was detected while processing the memory network interface instruction.
C	Corrected Error. This bit is set to indicate that a corrected error was detected while processing the memory network interface instruction.
U	Uncorrected Error. This bit is set to indicate that an uncorrected error was detected while processing the memory network interface instruction.
L	Lock Not Granted. This bit is set if the corresponding memory network interface instruction requested a lock and the lock was not granted.
E	Exception. This bit is set if the L, U, C or M bit is set in the message buffer pointer.

Table 12.15 Memory Network Interface Message Buffer Pointer Field Description

- The memory network interface supports up to 32 outstanding non-posted memory network interface instructions. There is no limit on the number of outstanding posted memory network interface instructions.

° When a non-posted memory network interface instruction is accepted by the memory network interface, a tag value in the range of 1 through 32 is returned. A tag value of 0xFF indicates that the memory network interface instruction was not accepted and must be reissued.

° When a non-posted memory network interface instruction completes execution, a message buffer pointer is written to inbound message queue 0. The message buffer pointer may be accessed using the mechanisms described in Section 12.7, "Message Network Interface" on page 12-14.

- The format of a message buffer pointer for a completed memory network interface instruction is shown in Figure 12.3 and described in Table 12.15. This format is different from the memory buffer pointer format shown in Figure 6.7 and used by message queues.

- After software has completed processing of a message buffer pointer for a completed memory network interface instruction, the message buffer pointer must be freed using the FreeMemTag instruction. The format of the message buffer pointer is that shown in Figure 12.3, but all fields except the tag are ignored by hardware.
- Hardware does not reuse a tag for a subsequent memory network interface instruction until it has been freed by software.
  - During initialization the free queue must be initialized with 32 unique tags in the range 1 through 32. Initializing the free queue with values outside this range produce undefined results.

12.8.1 Memory Network Instructions

12.8.1.1 Move In (MoveIn)

- Move data from controller memory to local node processor D-RAM.
- This instruction is always non-posted.
- MoveIn(in AR64 CAddr, in AR LAdddr, imm DWords, out AR Tag)
  - CAddr - Starting controller address of the data that is to be moved in.
    - CAddr must be DWord aligned.
  - LAddr - Starting local D-RAM address where the data that is moved in should be placed.
    - LAddr must be DWord aligned.
  - DWords - Immediate value containing the number of DWords that should be transferred. The immediate value has a range of 0 to 63 and corresponds to the number of DWords minus one (i.e., zero represents one DWord, one represents two DWords and so on).
    - The address range CAddr to (CAddr + DWords) must not cross a quad memory line boundary (i.e., a 256B boundary).
  - Tag - Value assigned by hardware and used to track completion.
    - A value of 0xFF indicates that the instruction was not accepted and must be reissued.
    - A value of 1 to 32 corresponds to a tag for an accepted memory network interface instruction.

12.8.1.2 Move Out (MoveOut)

- Move data from local node processor D-RAM to controller memory.
- This instruction may be posted or non-posted.
- MoveOut(in AR LAdddr, in AR64 CAddr, imm DWords, imm Posted, out AR Tag)
  - LAddr - Starting local D-RAM address of the data that is to be moved out.
    - LAddr must be DWord aligned.
  - CAddr - Starting controller address where the data that is moved out should be placed.
    - CAddr must be DWord aligned.

- DWords - Immediate value containing the number of DWords that should be transferred. The immediate value has a range of 0 to 63 and corresponds to the number of DWords minus one (i.e., zero represents one DWord, one represents two DWords and so on).
  - The address range CAddr to (CAddr + DWords) must not cross a quad memory line boundary (i.e., a 256B boundary).
- Posted - Immediate value indicating if the operation should be performed in a posted or non-posted manner.
  - 0 - Non-Posted
  - 1 - Posted
- Tag - Value assigned by hardware and used to track completion if the command was non-posted.
  - A value of 0xFF indicates that the instruction was not accepted and must be reissued.
  - A value of 1 to 32 corresponds to a tag for an accepted memory network interface instruction.
  - The tag is not used by posted MoveOut instructions and its value is undefined.

12.8.1.3 Move In and Lock (MoveInAndLock)

- Move data from controller memory to local node processor D-RAM and atomically lock the data structure in controller memory.
  - The original value of the lock bit in controller memory is returned in the Lock Not Granted (L) field of the memory network interface message buffer pointer.
    - If the original value of the lock bit was zero, then the L field is zero.
    - If the original value of the lock bit was one, then the L field is one.
  - The data value returned in local memory is the value before the lock bit was set by this instruction.
- This instruction is always non-posted.
- MoveIn(in AR64 CAddr, in AR LAdddr, imm DWords, imm LockBit, out AR Tag)
  - CAddr - Starting controller address of the data that is to be moved in.
    - CAddr must be DWord aligned.
  - LAddr - Starting local D-RAM address where the data that is moved in should be placed.
    - LAddr must be DWord aligned.
  - DWords - Immediate value containing the number of DWords that should be transferred. The immediate value has a range of 0 to 63 and corresponds to the number of DWords minus one (i.e., zero represents one DWord, one represents two DWords and so on).
    - The address range CAddr to (CAddr + DWords) must not cross a quad memory line boundary (i.e., a 256B boundary).
  - LockBit - Immediate value containing the position of the lock bit relative to the starting address of the data that is to be moved in (i.e., CAddr). The immediate value has a range of 0 to 511.

- Regardless of the number of DWords being transferred, the lock bit must be contained in the memory line associated with CAddr. Thus, the lock bit must be less than 512 or DWords*32, whichever is smaller. Values outside this range produce undefined results.
  - A value of zero corresponds to bit zero of the DWord at CAddr.
  - A value of 31 corresponds to bit 31 of the DWord at CAddr.
  - A value of 32 corresponds to bit zero of the DWord at (CAddr + 4).
  - Since the lock bit must be in the first memory line of the completion, the LockBit value must be less than 512 (i.e., 32 * 16).
  - Since the lock bit must be contained in the first memory line of the associated completion, the LockBit value must be less than DWords*32.

° Tag - Value assigned by hardware and used to track completion.

- A value of 0xFF indicates that the instruction was not accepted and must be reissued.
- A value of 1 to 32 corresponds to a tag for an accepted memory network interface instruction.

12.8.1.4 Transfer (Transfer)

- Move data between buffer RAM and DRAM.
- This instruction may be posted or non-posted.
- Transfer(in AR64 SrcAdddr, in AR64 DstAddr, imm DWords, imm Posted, out AR Tag)

° SrcAddr - Starting source address in controller memory

- SrcAddr must be DWord aligned.
- SrcAddr and DstAddr must be in different functional units (i.e., buffer RAM or DRAM controller). The behavior when SrcAddr and DstAddr are in the same functional unit is undefined.
- SrcAddr and DstAddr must have the same offset within a memory line (i.e., address bits 0 through 5 must be identical).

° DstAddr - Starting destination address in controller memory.

- DstAddr must be DWord aligned.
- SrcAddr and DstAddr must be in different functional units (i.e., buffer RAM or DRAM controller). The behavior when SrcAddr and DstAddr are in the same functional unit is undefined.
- SrcAddr and DstAddr must have the same offset within a memory line (i.e., address bits 0 through 5 must be identical).

° DWords - Immediate value containing the number of DWords that should be transferred. The immediate value has a range of 0 to 63 and corresponds to the number of DWords minus one (i.e., zero represents one DWord, one represents two DWords and so on).

- The address range SrcAddr to (SrcAddr + DWords) must not cross a quad memory line boundary (i.e., a 256B boundary).
- The address range DstAddr to (DstAddr + DWords) must not cross a quad memory line boundary (i.e., a 256B boundary).

° Posted - Immediate value indicating if the operation should be performed in a posted or non-posted manner.

- 0 - Non-Posted
- 1 - Posted

° Tag - Value assigned by hardware and used to track completion if the command was non-posted.

- A value of 0xFF indicates that the instruction was not accepted and must be reissued.
- A value of 1 to 32 corresponds to a tag for an accepted memory network interface instruction.
- The tag is not used by posted MoveOut instructions and its value is undefined.

12.8.1.5 Memory Add (MemAdd)

- Add a 32-bit value to a DWord in Controller Memory and write the new value to the same location in controller memory. This is performed in an atomic manner.
- This instruction may be posted or non-posted. In either case the value in controller memory (either before or after the add) is not returned in the completion.
- MemAdd(in AR64 CAdddr, in AR Value, imm Posted, out AR Tag)
  - ° CAddr - Address of DWord to operate on in controller memory.
    - CAddr must be DWord aligned.
  - ° Value - 32-bit value to add to DWord in controller memory.
  - ° Posted - Immediate value indicating if the operation should be performed in a posted or non-posted manner.
    - 0 - Non-Posted
    - 1 - Posted
  - ° Tag - Value assigned by hardware and used to track completion if the command was non-posted.
    - A value of 0xFF indicates that the instruction was not accepted and must be reissued.
    - A value of 1 to 32 corresponds to a tag for an accepted memory network interface instruction.
    - The tag is not used by posted MemAdd instructions and its value is undefined.

12.8.1.6 Memory XOR (MemXor)

- XOR a 32-bit value with a DWord in Controller Memory and write the new value to the same location in controller memory. This is performed in an atomic manner.
- This instruction may be posted or non-posted. In either case the value in controller memory is not retuned in the completion.
- MemXor(in AR64 CAdddr, in AR Value, imm Posted, out AR Tag)
  - ° CAddr - Address of DWord to operate on in controller memory.
    - CAddr must be DWord aligned.
  - ° Value - 32-bit value to XOR with DWord in controller memory.
  - ° Posted - Immediate value indicating if the operation should be performed in a posted or non-posted manner.
    - 0 - Non-Posted

- 1 - Posted

° Tag - Value assigned by hardware and used to track completion if the command was non-posted.

- A value of 0xFF indicates that the instruction was not accepted and must be reissued.
- A value of 1 to 32 corresponds to a tag for an accepted memory network interface instruction.
- The tag is not used by posted MemXor instructions and its value is undefined.

12.8.1.7 Free Memory Tag (FreeMemTag)

- This instruction frees a memory tag and allows it to be used for subsequent memory network interface instructions.
  - ° This instruction is also used to initialize memory tags. During initialization this instruction should be executed 32 times with tag values of one to 32.
- FreeMemTag(in AR Tag)
  - ° Tag - Value returned to hardware for use in subsequent memory network interface instructions.

12.9 CAR Registers

- Table 12.16 shows the processor node address map.
  - ° The CAR address is formed using the node address as described in Section 4.2.3. "CAR Address Space" on page 4-4.
  - ° The local node processor address is formed by adding the node address in Table 12.16 to the local node base address 0x7FF8_0000

Node Address	Register Mnemonic	Register Description
0x6_7FFC	SIMDBG	"SIMDBG - Simulation Debug" on page 27

Table 12.16 Flash Controller CSR Address Map

12.9.1 Debug Registers

SIMDBG - Simulation Debug

Bit Field	Field Name	Type	Default Value	Description
31:0	SIMDBG	RO	0x0	**Simulation Debug** This 32-bit read-only register is for simulation debug. The format and use of this register are defined by the simulation environment and firmware running on the processor node.



DRAM Interface

13.1 Introduction



Functional Unit Messages

14.1 Introduction

- This chapter defines the format and encoding of functional unit messages. These are messages between hardwired functional units and processors. In addition to these messages, processors may exchange messages among themselves. The format and encoding of these messages are under firmware control and are outside the scope of this specification.

14.2 CmdReady

DWord	Bit Field	Field Name	Description
15:0	31:0	Cmd	**Command** This field contains the corresponding DWord of the 64-byte NVMHCI command.
16	7:0	SQE	**Submission Queue Engine** This field contains the ID of the submission queue engine with which this command is associated.
16	31:8	Reserved	**Reserved**

Table 14.1 CmdReady Message Format

14.3 DmaCpl

DWord	Bit Field	Field Name	Description
0	31:0	Tag0	**Tag 0** This field contains opaque data that is passed from the request to the corresponding completion.
1	31:0	Tag1	**Tag 1** This field contains opaque data that is passed from the request to the corresponding completion.
2	1:0	Status	**Status** This field encodes the status of the operation. *0 (OK) Success -- no error* *1 (Malformed) Malformed request* *2 (Data) Data integrity error* *others (Reserved) Reserved*
2	31:2	Reserved	**Reserved**

Table 14.2 DmaCpl Message Format

14.4 EgressDma

DWord	Bit Field	Field Name	Description
0	19:0	ByteCnt	**Byte Count** This field contains the number of bytes to be transferred.
0	31:20	Reserved	**Reserved**
1	31:0	LHostAddr	**Lower Host Memory Address** This field contains the lower 32-bits of the 64-bit host memory address. The ingress and egress DMAs only support 32-bit aligned DMA transfers; therefore the bottom two bits of the LHostAddr and LAddr fields must be zero.
2	31:0	UHostAddr	**Upper Host Address** This field contains the upper 32-bits of the 64-bit host memory address.
3	31:0	LAddr	**Lower Memory Address** This field contains the lower 32-bits of the 37-bit controller memory address. The ingress and egress DMAs only support 32-bit aligned DMA transfers; therefore the bottom two bits of the LHostAddr and LAddr fields must be zero.
4	4:0	UAddr	**Upper Memory Address** This field contains the upper 5-bits of the 37-bit controller memory address.
4	23:5	Reserved	**Reserved**
5	31:0	Tag0	**Tag 0** This field contains opaque data that is passed from the request to the corresponding completion.
6	31:0	Tag1	**Tag 1** This field contains opaque data that is passed from the request to the corresponding completion.

Table 14.3 EgressDMA Message Format

14.5 EgressTlp

DWord	Bit Field	Field Name	Description
[7:0]	31:0	TLP	**TLP Contents** This field contains the contents of the TLP to be transmitted and must contain a well formed and complete PCIe header along with any required data. The host interface automatically adds the LCRC. Since the host interface never routes completions to the message network via IngressTlp messages, a transmitted TLP may only be of type posted or completion (i.e., not non-posted since the corresponding completion from the PCIe interface will not be returned). The TLP contents may be up to eight DWords to accommodate the maximum sized supported transmit TLP (i.e., 4 DWord header and 4 data DWords), but may be shorter for smaller TLPs.

Table 14.4 EgressTlp Message Format

14.6 FlashErase

DWord	Bit Field	Field Name	Description
0	4:0	FlashOp	**Flash Operation** This field encodes the Flash operation to be performed. This field must be set to two in FlashErase requests. *0 (Read) Flash page read* *1 (Write) Flash page write* *2 (Erase) Flash block erase* *others (Reserved) Reserved*
0	9:5	OutMsgQ	**Outbound Message Queue Index** This field contains the index of the Flash completion outbound message queue to be used for the completion (FlashOpCpl) corresponding to this request.
0	31:10	Reserved	**Reserved**
1	31:0	Reserved	**Reserved**
2	31:0	FlashAddr	**Flash Address** This field encodes the physical Flash address of the block to be erased.
3	31:0	Reserved	**Reserved**
4	31:0	Tag0	**Tag 0** This field contains opaque data that is passed from the request to the corresponding completion.
5	31:0	Tag1	**Tag 1** This field contains opaque data that is passed from the request to the corresponding completion.

Table 14.5 FlashErase Message Format

14.7 FlashOpCpl

DWord	Bit Field	Field Name	Description
0	31:0	Tag0	**Tag 0** This field contains opaque data that is passed from the request to the corresponding completion.
1	31:0	Tag1	**Tag 1** This field contains opaque data that is passed from the request to the corresponding completion.
2	1:0	Status	**Status** This field encodes the status of the operation. *0 (OK) Success -- no error* *1 (Malformed) Malformed request* *2 (Error) Operation completed with an error* *others (Reserved) Reserved*
2	15:2	EccErrors	**Total Number of ECC Bit Errors** If the completion is associated with a Flash read request and the Status field indicates success, then this field contains the total number of ECC bit errors that were detected in the Flash page. In all other cases, the contents of this field are undefined. The state of the EccOff bit in Flash read requests has no affect on this field (i.e., the ECC reports the errors but does not correct them).
2	25:16	ChunkEccErrors	**Maximum Number of Chunk ECC Bit Errors** If the completion is associated with a Flash read request and the Status field indicates success, then this field contains the maximum number of ECC bit errors that were detected in any ECC chunk (1080B) associated with the Flash page. In all other cases, the contents of this field are undefined. The state of the EccOff bit in Flash read requests has no affect on this field (i.e., the ECC reports the errors but does not correct them)
2	29:26	Retries	**Number Retries** If the Status field indicates success, then this field contains the total number poll retries required before it was determined that the Flash operation completed. In all other cases the contents of this field are undefined. A value of zero in this field indicates that no retries were required and that the Flash device indicated completion the first time it was polled.
2	31:30	Reserved	**Reserved**
3	31:0	FlashAddr	**Flash Address** This field contains the the physical Flash address contained in the corresponding request.
4	31:0	Addr	**Memory Address** For corresponding FlashRead and FlashWrite requests, this field contains the upper 32-bits of the 37-bit controller memory address contained in the corresponding request. For corresponding FlashErase requests this field is zero.

Table 14.6 FlashOpCpl Message Format

14.8 FlashRead

DWord	Bit Field	Field Name	Description
0	4:0	FlashOp	**Flash Operation** This field encodes the Flash operation to be performed. This field must be set to zero in FlasRead requests. *0 (Read) Flash page read* *1 (Write) Flash page write* *2 (Erase) Flash block erase* *others (Reserved) Reserved*
0	9:5	OutMsgQ	**Outbound Message Queue Index** This field contains the index of the Flash completion outbound message queue to be used for the completion (FlashOpCpl) corresponding to this request.
0	10	EccOff	**ECC Correction Off** When this bit is set, ECC correction is turned off and the data written to the Flash page frame in controller memory is the unmodified value read from Flash. This bit has no effect on the EccErrors and ChunkEccErrors fields in the corresponding FlashOpCpl message (i.e., both field continue to report the number of detected errors). It is an error to set both the EccOff and EccDisable bits at the same time. Doing so produces undefined results.
0	15:11	Permute	**Logical Page Permutation** This field encodes the manner in which 4KB logical pages read from a Flash page are permuted before they are written to controller memory. A value of zero indicates that no permutation should be performed.
0	30:16	Reserved	**Reserved**
0	31	EccDisable	**ECC Disable** This bit turns off the both the ECC error detection and ECC error detection logic. Data written to the Flash page frame in controller memory is the unmodified value read from Flash and since error detection is disabled, no errors are reported. When this bit is set, the EccErrors and ChunkEccErrors fields in the corresponding FlashOpCpl message always contain a value of zero. It is an error to set both the EccOff and EccDisable bits at the same time. Doing so produces undefined results.
1	31:0	Mask	**Read Mask** Each bit if this field represents 512B where bit zero represents the first 512 bytes of page data, bit one represents the second 512B of page data and so on. When a bit in this field is set, the corresponding bytes of the Flash page data stored in controller memory are preserved and are not overwritten by the contents read from Flash (if the data were actually read from Flash).
2	31:0	FlashAddr	**Flash Address** This field encodes the physical Flash address of the page to be read.

Table 14.7 FlashRead Message Format

DWord	Bit Field	Field Name	Description
3	31:0	Address	**Memory Address** This field contains the upper 32-bits of the 37-bit controller memory address that points to the start of the page frame data in controller memory. This address identifies a Flash page frame in controller memory. Since Flash page data is always 64-byte aligned bits zero through five are implicitly zero.
4	31:0	Tag0	**Tag 0** This field contains opaque data that is passed from the request to the corresponding completion.
5	31:0	Tag1	**Tag 1** This field contains opaque data that is passed from the request to the corresponding completion.

Table 14.7 FlashRead Message Format

14.9 FlashWrite

DWord	Bit Field	Field Name	Description
0	4:0	FlashOp	**Flash Operation** This field encodes the Flash operation to be performed. This field must be set to one in FlasWrite requests. *0 (Read) Flash page read* *1 (Write) Flash page write* *2 (Erase) Flash block erase* *others (Reserved) Reserved*
0	9:5	OutMsgQ	**Outbound Message Queue Index** This field contains the index of the Flash completion outbound message queue to be used for the completion (FlashOpCpl) corresponding to this request.
0	31:10	Reserved	**Reserved**
1	31:0	Reserved	**Reserved**
2	31:0	FlashAddr	**Flash Address** This field encodes the physical Flash address of the page to be written.
3	31:0	Address	**Memory Address** This field contains the upper 32-bits of the 37-bit controller memory address that points to the start of the page frame data in controller memory. This address identifies a Flash page frame in controller memory. Since Flash page data is always 64-byte aligned bits zero through five are implicitly zero.
4	31:0	Tag0	**Tag 0** This field contains opaque data that is passed from the request to the corresponding completion.

Table 14.8 FlashWrite Message Format

DWord	Bit Field	Field Name	Description
5	31:0	Tag1	**Tag 1** This field contains opaque data that is passed from the request to the corresponding completion.

Table 14.8 FlashWrite Message Format

14.10 IngressDma

DWord	Bit Field	Field Name	Description
0	19:0	ByteCnt	**Byte Count** This field contains the number of bytes to be transferred.
0	31:20	Reserved	**Reserved**
1	31:0	LHostAddr	**Lower Host Memory Address** This field contains the lower 32-bits of the 64-bit host memory address. The ingress and egress DMAs only support 32-bit aligned DMA transfers; therefore the bottom two bits of the LHostAddr and LAddr fields must be zero.
2	31:0	UHostAddr	**Upper Host Address** This field contains the upper 32-bits of the 64-bit host memory address.
3	31:0	LAddr	**Lower Memory Address** This field contains the lower 32-bits of the 37-bit controller memory address. The ingress and egress DMAs only support 32-bit aligned DMA transfers; therefore the bottom two bits of the LHostAddr and LAddr fields must be zero.
4	4:0	UAddr	**Upper Memory Address** This field contains the upper 5-bits of the 37-bit controller memory address.
5	31:0	Tag0	**Tag 0** This field contains opaque data that is passed from the request to the corresponding completion.
6	31:0	Tag1	**Tag 1** This field contains opaque data that is passed from the request to the corresponding completion.

Table 14.9 IngressDMA Message Format

14.11 IngressTlp

DWord	Bit Field	Field Name	Description
[7:0]	31:0	TLP	**TLP Contents** This field contains the contents of the received TLP and includes header and data. The LCRC is removed by the host interface. The TLP contents may be up to eight DWords to accommodate the maximum sized supported transmit TLP (i.e., 4 DWord header and 4 data DWords), but may be shorter for smaller TLPs.

Table 14.10 IngressTlp Message Format

14.12 CplQueueUpdate

DWord	Bit Field	Field Name	Description
0	0	InsertHead	**Insert Submission Queue Head Value** When this bit is set, the head value associated with the submission queue engine specified by the SQE field is inserted into bits 0 through 15 of the Completion DWord 0 (CmpDWord0) field prior to being written to the completion queue entry.
0	1	InsertPhaseTag	**Insert Phase Tag** When this bit is set, the phase tag bit value associated with the completion queue engine specified by the CQE field is inserted into bit 16 of the Completion DWord 1 (CmpDWord1) field prior to being written to the completion queue entry.
0	15:2	Reserved	**Reserved**
0	23:16	CQE	**Completion Queue Engine ID** This field specifies the completion queue engine with which this message request is associated.
0	31:24	SQE	**Submission Queue Engine ID** This field specifies the submission queue engine used when the InsertHead bit is set in this register. Valid values for this field are zero through 32. Values outside this range produce undefined results.
1	31:0	CmpDWord0	**Completion DWord 0** This field contains the first DWord of the NVMHCI completion queue entry.
2	31:0	CmpDWord1	**Completion DWord 1** This field contains the second DWord of the NVMHCI completion queue entry.
3	31:0	CmpDWord2	**Completion DWord 2** This field contains the third DWord of the NVMHCI completion queue entry.

Table 14.11 CplQueueUpdate Message Format

DWord	Bit Field	Field Name	Description
4	31:0	CmpDWord3	**Completion DWord 3** This field contains the fourth DWord of the NVMHCI completion queue entry.
5	31:0	Tag0	**Tag 0** This field contains opaque data that is passed from the request to the corresponding completion.
6	31:0	Tag1	**Tag 1** This field contains opaque data that is passed from the request to the corresponding completion.

Table 14.11 CplQueueUpdate Message Format

14.13 CplQueueUpdateCpl

DWord	Bit Field	Field Name	Description
0	31:0	Tag0	**Tag 0** This field contains opaque data that is passed from the request to the corresponding completion.
1	31:0	Tag1	**Tag 1** This field contains opaque data that is passed from the request to the corresponding completion.
2	1:0	Status	**Status** This field encodes the status of the corresponding NVMHCI completion queue update request (i.e., CplQueueUpdate) message. *0 (OK) Success -- no error* *1 (Malformed) Malformed request* *2 (Full) Completion queue was full* *3 (Disabled) Selected completion queue engine was disabled* *others (Reserved) Reserved*

Table 14.12 CplQueueUpdateCpl Message Format

Revision History

- ***February 18, 2010***
  - *Changed the format of functional unit completion messages. The tag fields now come first. The intent of this change is that tag 0 could be used to demux completions that target the same queue in a processor.*

- ***April 19, 2010***
  - *Added text in the Flash Controller chapter showing how page frames are addressed.*
  - *Changed number of submission and completion queues from 33 to 17.*
  - *Changed doorbell register pointers to 15-bits (SQxTDBL, CQxHDBL, SQT field in the SQE[16:0]TDBL register, CHQ field in the CQE[16:0]HDBL register).*
  - *Added MPS field to the PCIECTL register. Delaware will have the MPS field hardwired to 0x0 since it will only support and MPS of 128B.*
  - *Added REGPROC to the MEMLBAR register in the host interface. This register field controls how access to hardware registers are treated.*
  - *Added disabled completion status to the CplQueueUpdateCpl message. Added corresponding text to Section 8.7 NVMHCI Completion Queue Manager.*
  - *Updated size of an NVMHCI completion entry from 2 DWords to 4 DWords. Updated text in host interface chapter and updated CplQueueUpdate message.*
  - *Replaced signals chapter with a system configuration chapter.*

- ***April 27,2010***
  - *Added restriction that page frame data must start on a 4KB boundary and that the page frame meta data must start on a 256B boundary.*
  - *Changed the FTSYNC2 register to a per Flash channel control (i.e., CH[7:0]FTSYNC2) in the Flash controller.*
  - *Removed MEN field from the MEMLBAR register in the host interface. Fixed typo by making the REGPROC field two bits.*

- ***May 12, 2010***
  - *Swapped the order of address fields in the IngressDMA message to match what was implemented in the RTL.*
  - *Made the fields in the SQExTDBL and CQExHDBL registers read-write to match RTL implementation.*
  - *Fixed typos in host interface chapter ([33:0] changed to [16:0]).*
  - *Removed byte count restriction in IngressDMA and EgressDMA messages. The byte count no longer needs to be a multiple of four.*
  - *Updated system configuration chapter to match what we are really building now.*
  - *Added source and destination address restriction clarification to XFER instruction in the processor complex chapter.*
  - *Added Flash timing register configuration to flash chapter.*

- ***May 14, 2010***
  - *Add Calibration Mode (CM) bit to the FCCTL register and added flash calibration chapter to the flash controller chapter.*

- ***May 22, 2010***
  - *Fixed inconsistency between processor node assignments in Table 3.5 and Figure 3.2. Figure 3.2 was updated with the assignments in Table 3.5.*
  - *Added Simulation Debug (SIMDBG) register to processor node. The intent of this register is to allow a console printf mes-*

sage during RTL simulations.



DRAM Interface

10.1 Introduction

The DRAM Interface (DINTF) is a memory controller that architecturally acts as a cache engine for the Delaware's flash data. External DDR2 DRAM is utilized to hold data structures required for Flash controller operations (e.g., logical to physical mapping table) and to act as read cache / write buffer for storage data.

10.2 Feature List

- Uses the external DDR2 DRAM as a Cache for the flash memory.
- Utilizes the Xilinx DDR2 Controller IP.
- Allows accesses and operations on data in DRAM via the following interfaces:
  - DRAM Data Network
  - Buffer RAM Data Network (only from DRAM to BR)
- Data Network read/write commands payload vary from 64 bytes to 256 bytes with 64-byte (one memory line) increment.
- For read requests that spans more than one memory line, all read data is accumulated until the last memory line is read before a read completion is generated.
- Supported commands are Rd, RdLock, RdXorWr, Wr, WrLine and Xfer.
- Commands are processed in order of arrival, there is no command re-ordering.
- No ECC of DRAM.
- Supports 1 rank/1GB DDR2 DRAM DIMM compatible with the Xilinx DDR2 Controller.
- Runs the core at 125MHz and the DDR2 Controller/interface at 250/500MHz.

10.3 Overview

The DINTF is the DRAM Cache controller for the Delaware's flash data. Besides the flash data the DRAM memory is extended to hold the Flash Page Mapping Table and Block Table, the PCIe Configuration space and the NVMHCI commmand structure and table. The DRAM Cache is managed by the DRAM Buffer Processor (DBP) from the Processor Complex.

Multiple processors and other blocks can access the DRAM via the DRAM Data Network Interface. Data Network commands are DRAM Reads/Writes commands with some extra actions. Each command has a header and a variable payload from one to four memory lines (64-byte to 256-byte in 64-byte increment).

The DINTF is composed of the DRAM Controller (DRC) and the DDR2 Controller. The DRC decodes the Data Network commands into memory read/write requests. The Xilinx DDR2 Controller IP converts the DRC memory read/write commands into DRAM commands. It also takes care of all DRAM related function such as refresh, power up initialization and DQ/DQS synchronization.

The DRC and the rest of the Delaware core logic runs at 125MHz and the DRR2 Controller IP runs at 250MHz with a 64-bit data bus.

Figure 10.1 DRC Block Diagram

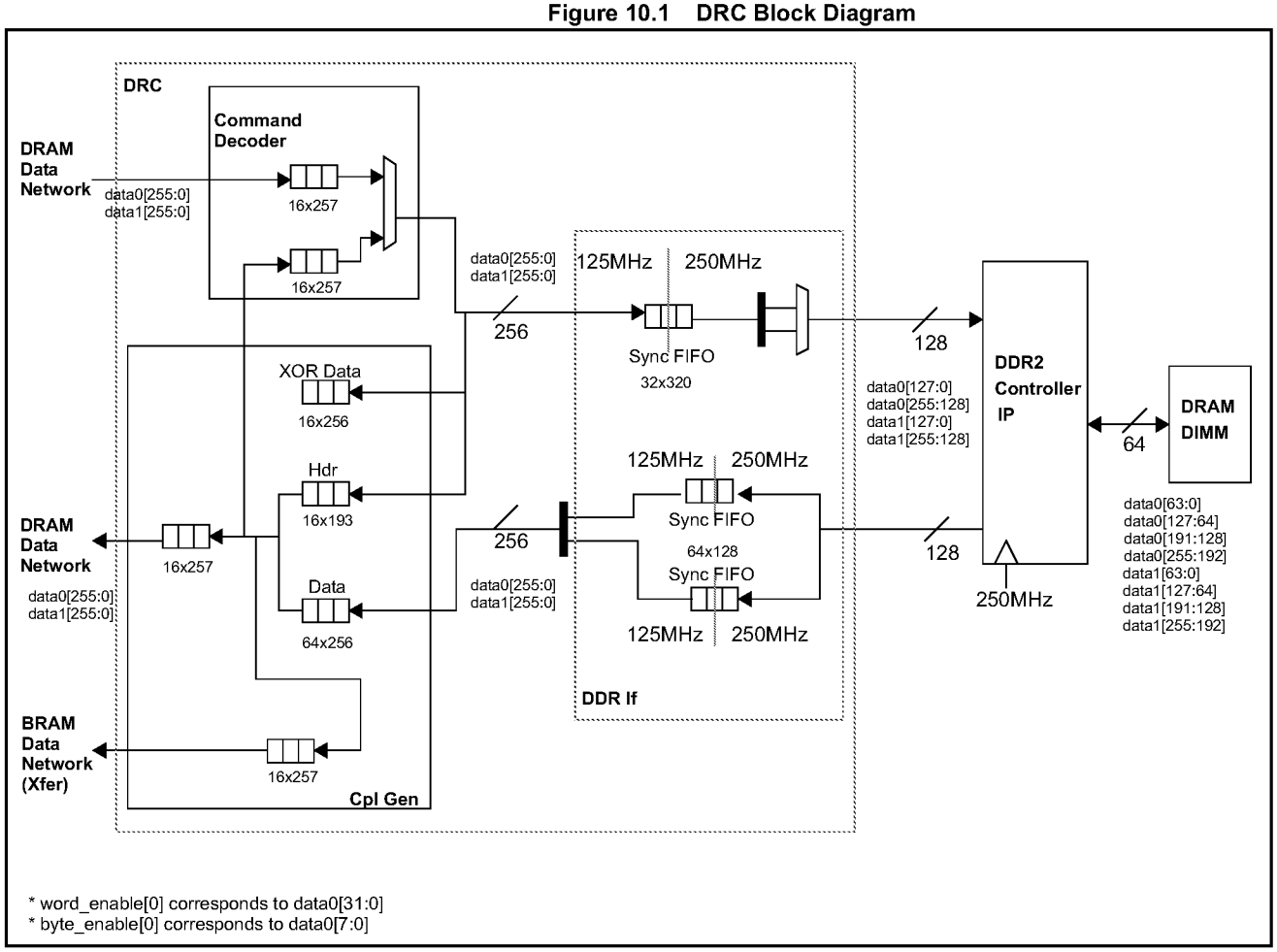
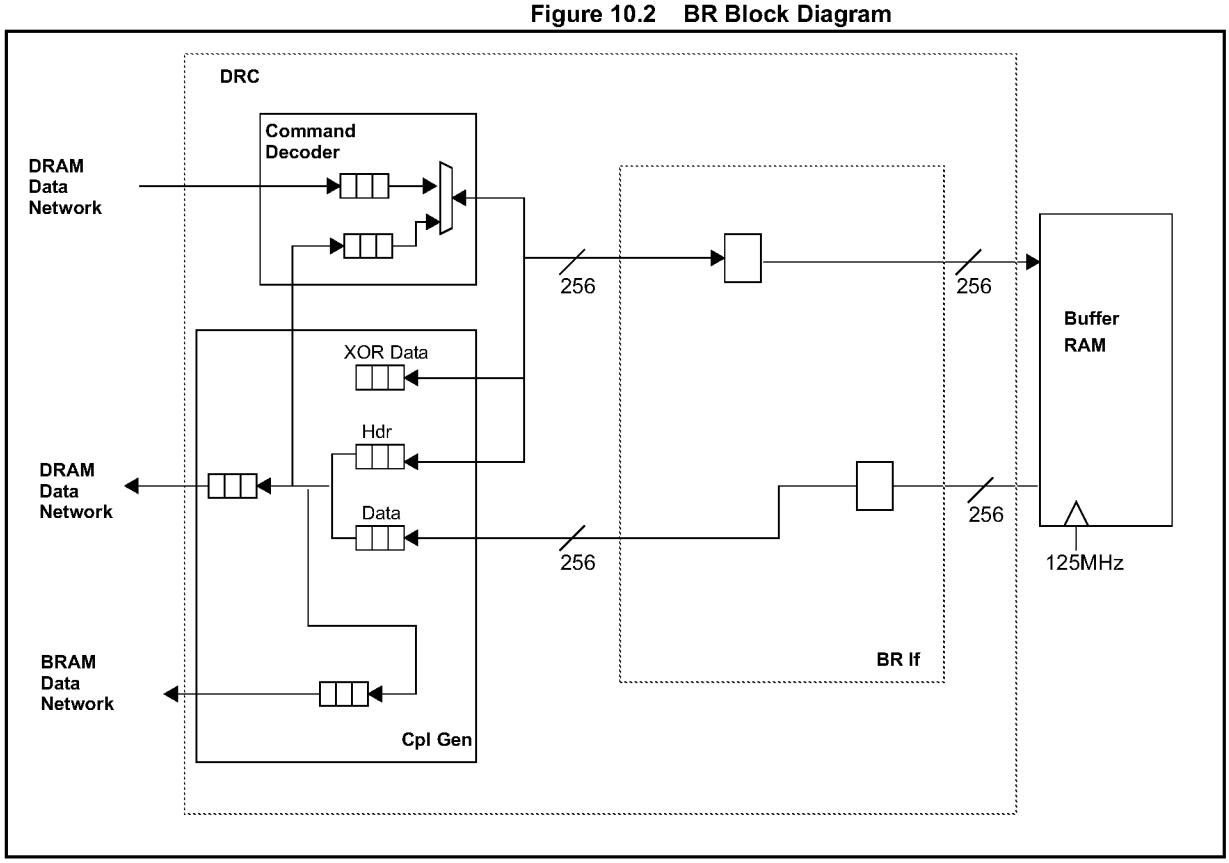


Figure 10.2 BR Block Diagram



10.4 Functional Operation

10.4.1 DRAM Controller (DRC)

10.4.1.1 Command Decoder (CDEC)

When it is ready to accept new commands, the Command Decoder (CDEC) block pulls a command from an upstream Data Network node and decode it. The Data Network commands come in a packet format that the CDEC converts into control/data signals format.

Command	Command Size	DRAM Action	Completion	Extra Action
Read	Header	Read	Header + 1-4 lines	
Read Lock	Header	Read-Write	Header + 1-4 lines	Write Lock bit
Read/XOR/ Write	Header + 1-4 lines	Read-Write	Header	Write XOR
Transfer	Header	Read	Header	Write to BR
Write	Header + 1-4 lines	Write DW Enbl	Header	
Write Line	Header	Write Byte Enbl	Header	

Table 10.1 Commands Processed by the Command Decoder

* Each (memory) line is 64 bytes

Although the commands are in terms of memory line unit (64B) the Delaware internal data path is only 256 bits (32B). Therefore the CDEC will generate read/write requests in 256-bit term, this also facilitate the re-use of this block for the Buffer RAM design. For each read/write command, the output command/address is decoded as followed:

Command Size	Output Cmd (one per cycle)
1 line	{Cmd_addr, 0}, {Cmd_addr, 1}
2 lines	{Cmd_addr, 0}, {Cmd_addr, 1}, {Cmd_addr+1, 0}, {Cmd_addr+1, 1}
3 lines	{Cmd_addr, 0}, {Cmd_addr, 1}, {Cmd_addr+1, 0}, {Cmd_addr+1, 1}, {Cmd_addr+2, 0}, {Cmd_addr+2, 1}

Table 10.2 Output Commands/Address generated by the Command Decoder

Command Size	Output Cmd (one per cycle)
4 lines	{Cmd_addr, 0}, {Cmd_addr, 1}, {Cmd_addr+1, 0}, {Cmd_addr+1, 1}, {Cmd_addr+2, 0}, {Cmd_addr+2, 1}, {Cmd_addr+3, 0}, {Cmd_addr+3, 1}

Table 10.2 Output Commands/Address generated by the Command Decoder

For the ReadLock and Read/XOR/Write commands, a write is done after a read. The CDEC issues a DRAM read to the DDR interface block first. When the read data is returned the CPGEN block generates a write command to CDEC. This means that the CDEC has to arbitrate between the Data Network command and the return Write command. The return Write command always has higher priority, Since the the CDEC can be busy issuing up to 8 256-bit read/write request per Data Network command, the return Write command must be queued in an 8-deep fifo.

```
output cdec_memif_valid;
output cdec_memif_wr;
output [30:0] cdec_memif_addr;  // add 1 more bit for 256b resolution
output [255:0] cdec_memif_data;
output [31:0] cdec_memif_mask;
```

The command sent to CPGEN is generated on the last request of a command.

```
output cdec_cpgen_valid;
output [2:0] cdec_cpgen_req_type;
output [2:0] cdec_cpgen_cmd_len;  //(in 32B or 256-bit)
output [30:0] cdec_cpgen_addr;
output [9:0] cdec_cpgen_req_id;
output [7:0] cdec_cpgen_req_tag;
output [255:0] cdec_cpgen_xor_mask;
```

10.4.1.2 DDR Interface (DDRIF)

The DDR Sync block takes the Read/Write request from CDEC and converts it into a format accepted by the DDR2 Controller. It also does a rate conversion from 256-bit/125MHz to 128-bit/250MHz and vice versa.

Since the requests coming from CDEC are in 256-bit format, the DDRIF will generate a request to the DDR Controller every other CDEC command. For example, a read command comes in a low and high address, only the high address is sent to the DDR Controller. For a write request, a low address with write data is sent to the data FIFO of the DDR Controller first, the high address would send a write command.

NOTE: The DRC design can be used for the Buffer RAM design if this block is replaced to interface to an internal SRAM.

10.4.1.3 Completion Generator (CPGEN)

The Completion Generator (CPGEN) receives completion requests from the CDEC block. It has a data fifo that accumulates up to 4 memory lines of a read request before genersting a completion. This is done to ensure the Rd Cpl will be sent without gaps.

```
case (req_type)
  Rd: wait until data fifo has enough data that matches the cmd_len before
sending Cpl.
  Wr/WrLn: generate completion.
  RdLck: generate a wr command to CDEC when read data is returned. The comple-
tion will be generated when the wr is done.
  RdXorWr: XOR the read data with mask and generate a wr to CDEC.
  Xfer: wait until data fifo has enough data that matches the cmd_len before
sending Cpl, also send a BR wr command to the XPROC.

output cpgen_cdec_valid;
output [9:0] cpgen_cdec_req_id;
output [7:0] cpgen_cdec_req_tag;
output [30:0] cpgen_cdec_addr;
output [255:0] cpgen_cdec_data;

output cdec_xproc_valid;
output [2:0] cpgen_xproc_cmd_len;  //(in 32B or 256-bit)
output [30:0] cdec_xproc_addr;
output [255:0] cdec_xproc_data;
```

10.4.1.4 Transfer Processor (XPROC)

When receive a request from the CPGEN block, the XPROC block converts the request into a BR write command and send it over the BR Data Network. The XPROC will issues a Write request to the BR with the same payload size as the original XFER request from the Data Network.

10.4.2 Latency Estimate

All units are in 125MHz clk (8ns).

Command	Note	Refresh	CDEC Delay	DDR I/f Delay	DDR CTL + DRAM Delay	CPGEN Delay	Total (125MHz clks)
Read (64B)	Without Precharge and Active	0	2	6	14	11	33
	With Precharge and Active	0	2	6	18	11	37
	With Precharge and Active	25	2	6	18	11	62
Write (64B)	Without Refresh	0	2	0	0	5	7
	With Refresh	25	2	0	0	5	32

Table 10.3 DINTF Latency

Notes

1. Introduction

IDT provides a low-level Windows device driver for its enterprise programmable NAND Flash Drive Controller. This driver adopts Microsoft Storport driver interface and is developed in the form of Storport miniport driver. IDT's Storport miniport takes advantage of the advanced features provided by Storport driver to better suit the target enterprise applications that require higher I/O throughput performance.

This document describes the overall architecture and in-depth implementation details regarding:

- *Where the driver is positioned in Windows storage driver stack*
- *How the driver supports and processes various requests*
- *How the driver interacts with its upper layer drivers that Windows Operating System provides*
- *How the driver works with the eNVMHCI Controller to complete the requests*

At the end (Section 5), an example is provided to demonstrate how the miniport driver processes a data transfer from system memory onto the flash memory via an eNVMHCI defined Write command.

2. Storport Driver Stack Overview

Windows applications send storage read/write (I/O) requests through the Storport driver stack (see Figure 1).

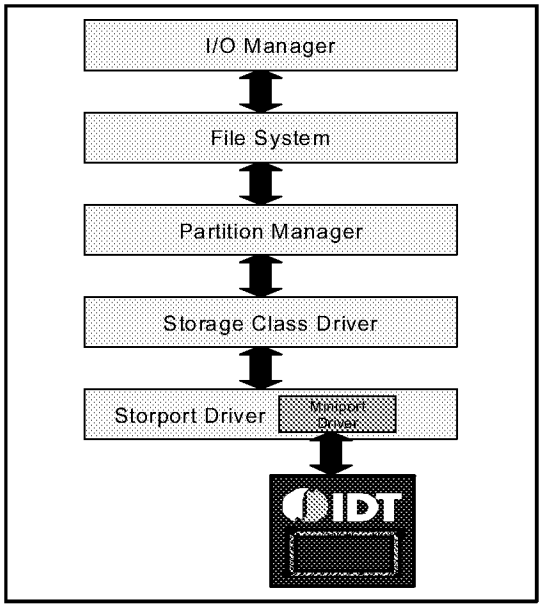


Figure 1 The I/O Request Path Through the Storport Driver Stack

The I/O Manager processes and passes I/O request packets (IRP) onto the Storage Class driver. The Storage Class driver constructs a new SCSI request block (SRB) for each IRP and pass down to each lower driver layer, which successively routes the request to the driver that's assoicated with the target LU, capable of processing the request and returning with appropriate status in SRB structure.

When an SRB comes to Storport driver, it needs to do one of the followings:

- *Complete the SRB without passing it on to lower layers if no further processing is necessary.*
- *Queue the SRB on behalf of IDT's Flash Drive Controller if notified by IDT's miniport driver that the*

Notes

controller is busy.

- *Pass the SRB down to IDT's miniport driver for further processing.*

When IDT's miniport driver receives the SRB, it handles the request based on the SRB Function and the encoded 16-byte Command Descriptor Block (CDB):

- *Complete the SRB with existing information the driver has and return appropriate status.*
- *Complete the SRB by converting the request into eNVMHCI defined command if there is Command Header/Table available and return with appropriate status when completed.*
- *If the number of Command Headers/Tables in the driver's free queue reaches to the designated threshold, notify Storport to hold up requests until certain amount of requests had been completed.*
- *Should there is error(s) while processing the request and reported in Command Status Entry Description bits, return with the error status and note the details in event logging.*

More details of the implementation of IDT's miniport driver is elaborated in Section 3 and 4.

3. Miniport Driver Overview

IDT's miniport driver is designed to optimize the I/O throughput while accessing the associated flash memory complex via IDT's eNVMHCI Controller. The driver is composed of two portions: Storport callback routines and functional modules (see Figure 2).

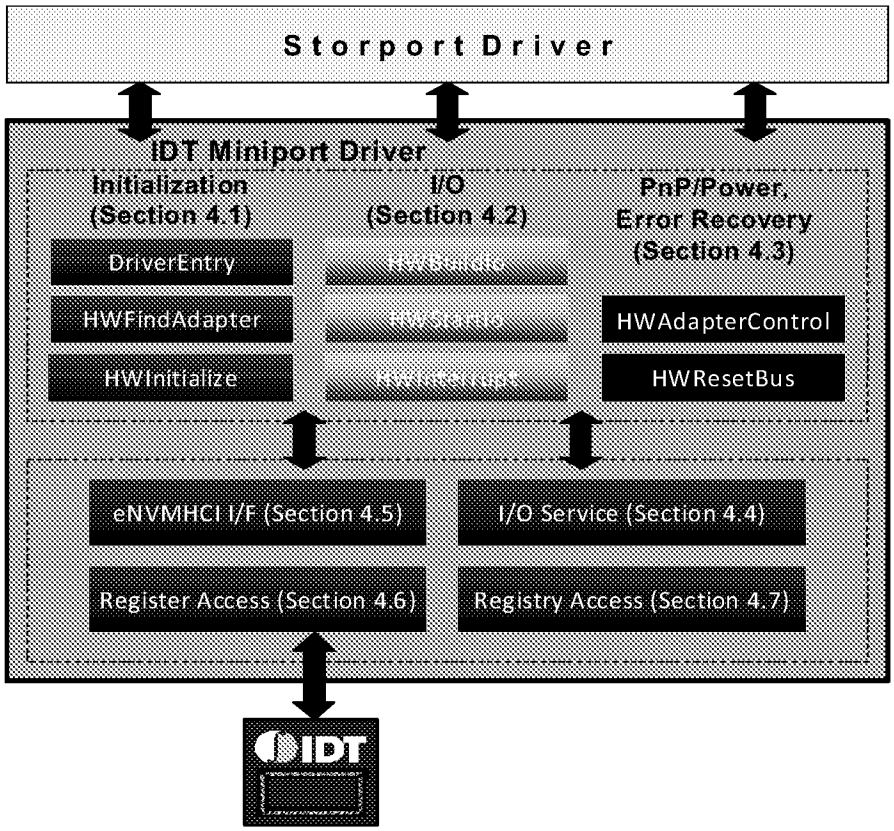


Figure 2 IDT Miniport Driver Block Diagram

The interface to Storport driver includes three categories: Initialization, I/O and PnP(Power Management and Error Recovery). The functional modules (in blue) are designed to support I/O and control requests, handle error siuations, access Registry and interface to IDT's eNVMHCI controller. More details on routine basis will be described in next section.

Notes

4. Miniport Driver Design

In this section, the implementation details of the each callback function category and functional module are disclosed in a dedicated sub-section. The details of each main routine include partial or all the following information. And Section 4.8 is dedicated to the shared data structures.

- *The routine prototype*
- *The purposes and tasks of the routine*
- *The associated data structure(s) for easier references.*
- *The flow chart of the events within the routine*

4.1 Initalizatoin Routines

4.1.1 DriverEntry

ULONG DriverEntry (PVOID DriverObject, PVOID RegistryPath);

In DriverEntry, an instance of data structure HW_INITIALIZATION_DATA needs to be allocated and populated with the information listed in Table 1:

Members	TYPES	Descriptions
HwInitialzationDatasize	ULONG	Size of the HW_INITIALIZATION_DATA structure.
AdapterInterfaceType	INTERFACE TYPE	PCIBus
HwInitialize	PHW_INITIALIZE	Driver initialization callback function, IdtInitialize (Section 4.1.3)
HwStartIo	PHW_STARTIO	I/O function entry point, IdtStartIo (Section 4.2.2)
HwInterrupt	PHW_INTERRUPT	Interrupt service routine, IdtISR (Section 4.2.7)
HwFindAdapter	PHW_FIND_ADAPTER	Device realization callback function, IdtFindAdapter (Section 4.1.2)
HwResetbus	PHW_RESET_BUS_BUS	Bus reset function to clear error condition, IdtReset (Section 4.3.2)
HwAdapterControl	PHW_STOP_ADAPTER	Power management and PnP function entry point, IdtAdapterCtrl (Section 4.3.1)
HwBuildIo	PHW_BUILDIO	Function to support asynchronous I/O prior to calling HwStartIo, IdtBuildIo (Section 4.2.1)
DeviceExtensionSize	ULONG	Size of the DEVICE_EXTENSION structure (Section 4.8.5)
SrbExtensionSize	ULONG	Size of the SRB_EXTENSION structure (Section 4.8.1)
NumberOfAccess-Ranges	ULONG	Only the range of memory addresses, set as 1
NeedPhysicalAddresses	BOOLEAN	TRUE to indicate the driver needs to use physical addresses
AutoRequestSense	BOOLEAN	TRUE to indicate the capability to perform a request-sense operation without requiring an explicit request to do so.
TaggedQueuing	BOOLEAN	TRUE to indicate supporting tagged queuing
MultipleRequestPerLu	BOOLEAN	TRUE to indicate queuing supported per logical unit
MapBuffers	BOOLEAN	Set as STOR_MAP_NON_READ_WRITE_BUFFERS, which means mapping the SRB buffers to system virtual address only for non read/write requests
VendorIdLength	USHORT	Set as 4 for {'1', '1', '1', 'd'}
VendorId	PVOID	Assigned to {'1', '1', '1', 'd'}
DeviceIdLength	USHORT	Set as 4.
DeviceId	PVOID	TBD
Others		Set as NULL or 0

Table 1 Data Structure - HW_INITIALIZATION_DATA

Notes

Before it returns, call StorPortInitialize with DriverObject, RegistryPath and the pointer to the instance of HW_INITIALIZATION_DATA to register a port driver instance with Storport driver for IDT's Controller. The initial hand-shaking between Storport driver and the miniport driver is depicted in Figure 3. Step 1 and 2 is completed with DriverEntry. Based on the returned IDs, Storport driver works with the PnP Manager to see if the target device is present or not. If yes, it calls IdtFindAdapter in Step 3 to proceed the initialization process. Otherwise, Storport driver notes the associated device object state as absent for now. If a target device is plugged in later, it will work with the PnP Manager again in similar process.

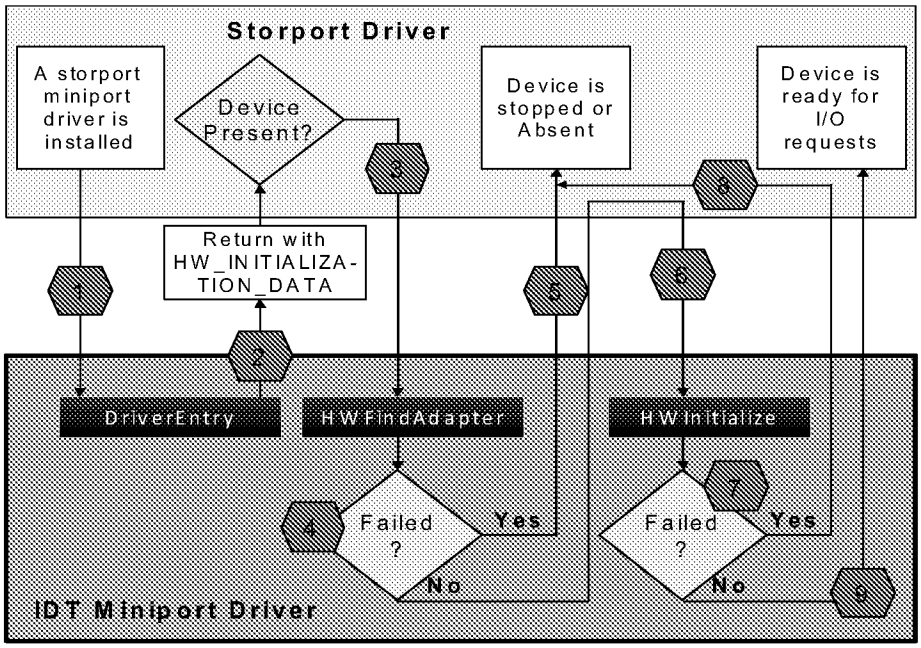


Figure 3 Driver Initialization Flow Chart

4.1.2 IdtFindAdapter

```
ULONG IdtFindAdapter (
    PVOID pDE, // Pointer to an allocated DEVICE_EXTENSION structure
    PVOID HwContext, // Pointer to the instance of HW_INITIALIZATION_DATA structure
    PVOID BusInformation, // PCI bus specific information
    PCSTR ArgumentString, // Environment context in which we are being called
    PPORT_CONFIGURATION_INFORMATION ConfigInfo, // Port configuration and resource data
    PBOOLEAN Again // not used
);
```

The IdtFindAdapter is then called by StorPortInitialize after it finds the vendor/device IDs matching the IDs passed in DriverEntry. If the IDs of HwContext belong to the device that is supported by the miniport driver, IdtFindAdapter completes the followings without any failures and return SP_RETURN_FOUND. If the IDs are not matched, it returns SP_RETURN_NOT_FOUND. If the IDs are matched, and yet any of the followings fails, it returns SP_RETURN_ERROR. Step 3 to 5 in Figure 3 are completed with this routine.

- *Populate DEVICE_EXTENSION members*
- *Access Registry to retrieve the following pre-determined values:*
  - Maximum number of NVM pages in a single transfer operation (MaxPages), each page is 4K or 8K bytes in length.
  - Maximum number of PRDT entries (MaxPRDTEntries), which is used to set the maximum number of Scatter/Gather elements in a single transfer operation.
  - The size of Metadata (MaxMetaDataSize)
- *Call MmMapIoSpace to remap BAR0 to an non-paged and non-cached I/O space for Controller/ Port related register access and save it in RegBase0 of IDT_PCI_CONFIG*

Notes

- *Ensure the adapter itself is ready for I/O requests by checking the following fields:*
  - PI in Generic Host Control is non-zero.
  - Find out the number of Command FIFO slots available via NCS field of CAP.
  - NCE of PxCMD of the port register is not set.
  - PRDY of PxCMD of the port register is set.
  - NVMP of PxCMD of the port register is set.
- *Access Command FIFO register to find out how many FIFO slots and Command Status registers are supported. Assuming we have 16 CPU processors in current system and 4096 FIFO slots supported, there are 256 FIFO slots supported for each processor and the maximum outstanding I/O requests is 256. IDT's miniport driver allocates 256 Command Headers/Tables for each processor.*
- *If BAR5 is supported, call MmMapIoSpace to remap BAR5 to an non-paged and non-cached I/O space for PCI Configuration register access and save it in RegBase5 of IDT_PCI_CONFIG. If BAR5 is not supported, call StorPortGetBusData instead to retrieve it.*
- *If succeeds, remember all the virtual address and values of the members of AccessRanges structure in DEVICE_EXTENSION. Otherwise, return with SP_RETURN_ERROR.*
- *Call StorPortGetHighestNodeNumber and StorPortGetNodeAffinity for the current processor topology and find out the number of processors in the system.*
- *Call StorPortGetMSIInfo to retrieve MSI/MSI-X related information that controller supports.*
- *Call IdtGetCPUTopology to get the current CPU topology in the system.*
- *Fill in fields of* PORT_CONFIGURATION_INFORMATION*:*
  - Assign IdtMSIISR to HwMSInterruptRoutine due to MSI-X support
  - Set MaximumTransferLength as (MaxPages * NVMPageSize).By default, it's 4 MB (1024 * 4096 or 512 * 8192)
  - Set NumberofPhysicalBreaks as MaxPRDTEntries. By default, it's 32
  - Assign StorSynchronizeFullDuplex to SynchronizationModel
  - Assign InterruptSynchronizePerMessage to InterruptSynchronizationMode
  - Set CachesData as TRUE
- *Return SP_RETURN_FOUND if all the above succeeds.*

4.1.3 IdtInitialize

```
ULONG IdtInitialize (
  PVOID pDE, // Pointer to an allocated DEVICE_EXTENSION structure
);
```

After IdtFindAdapter returns successfully, Storport driver calls IdtInitialize to finalize the entire initialization process (Step 6 and 7 in Figure 3). IdtInitialize completes the following tasks:

- *When Identify command is supported, allocate an non-paged memory buffer (4K bytes) for one Command Header/Table and data buffer that receives 512-byte data from the Controller*
- *Set the Interrupt bit of Command Header so that hardware sets PxIS.CCS to 1 when completed*
- *Driver polls the PxIS.CCS to see if the command is completed.*
- *If succeeds, issue an eNVMHIC Identify command to retrieve the critical information below and save in the IDT_DEVICE_INFO structure. The information is used to determine the sizes of upcoming Command Header/Table allocations.*
  - Sector size in bytes (512 bytes by default)
  - NVM page size in bytes (4096 bytes by default)
  - Metadata size in bytes per NVM page, when 0, means not supported (14 bytes by default)
  - Capacity, total number of NVM pages
- *If the Identify command fails, it returns FALSE and logs SP_ADAPTER_INTERNAL_ERROR with IDT_DRV_ERROR_INIT. If identify command is not support or fails, the driver uses the default values.*
- *Free the allocated buffer for Identify command.*
- *For each processor, call StorPortAllocateContiguousMemorySpecifyCacheNode to allocate a contiguous, non-cached and non-paged memory buffer for the entire Command Header/Table and Status Event structure based on the number of FIFO slots per processor and the pre-defined numbers*

Notes

below, which are read back from Registry. The starting address of this buffer needs to be 1-Kbyte aligned.

- The size of Metadata Block (MetaDataSize)
- The pre-defined MaxPages per Command Table, each page is represented by one PRD Index entry (4K or 8K bytes in length)
- The pre-defined MaxPRDTEntries (each is 16 bytes long) per Command Table and each entry can transfer a data block up to 256KB.
- The complete layout of the Command Header/Table/Status structure in memory is depicted in Figure 4. Due to the maximum outstanding I/O requests in Storport driver is 254 per device, the maximum number of Command Header entries to allocate is 256, which is divided by the number of CPU cores in current system. If there are 16 cores, N is equal to 16 in Figure 4
- The MSI-X vector 0 is used for interrupting for error situations or management purposes. MSI-X vector 1 to 31are used to associated different CPU cores. In other words, the MSI-X# field in all allocated Command Headers for CPU core X is initialized as X+1
- Each Command Header entry is 64 bytes in length. The lower 32 bytes are used to stored proprietary information and the upper 32 bytes for eNVMHCI defined Command Header information. More details of how the 64-byte information are defined can be found in structure _CMD_HEADER in Section 4.5.3. The important fields of the proprietary half of the Command Header are:
  - The Queue Pointers of LIST_ENTRY, Flink and Blink. These pointer are used in double linked - list queue and initialized when the Command Header entry is added into FreeQ.
  - The Tag number is used as index number to locate the exact physical or virtual addresses of the Command Header when driver need to know which Command Header entry is being used for the just completed I/O when interrupted due to an I/O completion. In other words, the first allocated Command Header entry owns Tag number as 0, the second one has 1, the final entry N has Tag number as N-1. The Command Tag field in the associated Command Header contains same number as the assigned Tag number.
  - The reason the driver needs to locate the Command Header entry with Tag number is to recover the original SRB that is required to complete the assoicated I/O request in IdtDpc. The pSrb is used for saving the original SRB
  - Used field is used to indicate whether the Command Header entry is free or not. When the entry is in FreeQ, it is zeroed out. When it's being used for a request, it is one.
  - The CPU# field indicates which CPU core the current Command Header is associated with and it's initialized when the Command Header is allocated successfully

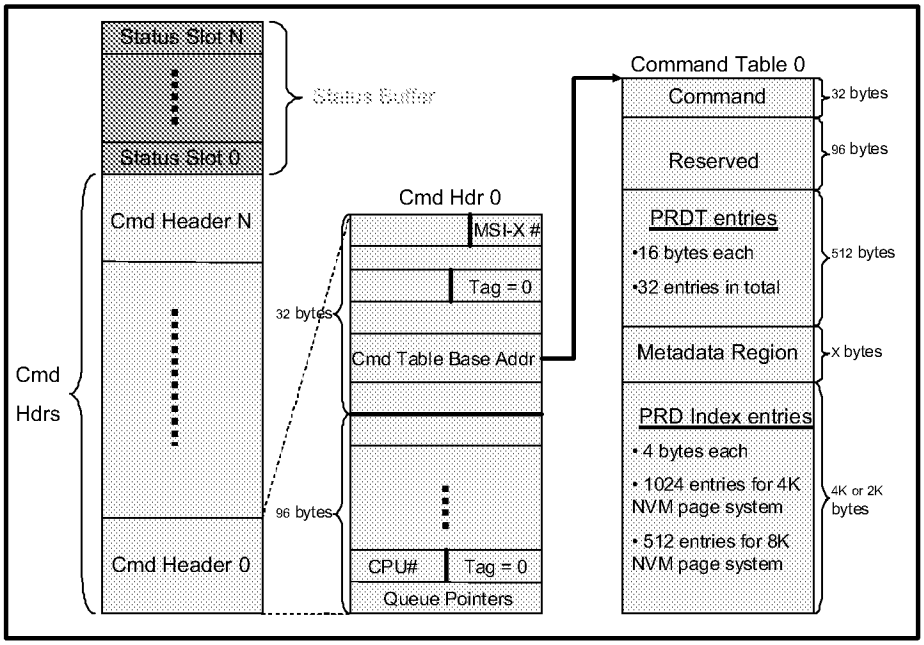


Figure 4 Command Header/Table and Status Event Buffer Layout

Notes

– If Metadata is supported and the sizes are 14 and 32 bytes for each 4K and 8K pages, respectively, the size of each allocated Command Table calculated in Table 2. The size of Metadata Region (X) is sum of Metadata Size and Padding Size.

NVMPage Size	Metadata Size	Padding Size	Command Table size
4096 bytes	14436 bytes	1308 bytes	32+96+512+14436+1308+4096 = 20480 bytes
8912 bytes	16384 bytes	1408 bytes	32+96+512+16384+1408+2048 = 20480 bytes

Table 2 Command Table Size When Metadata Supported

– When Metadata is not supported, the size of each allocated Command Table is 8K and 4K bytes for NVMPageSize is 4K and 8K bytes, respectively. The calculation is done in Table 3 below and the size of Metadata Region (X) is Metadata (Padding) Size.

NVMPage Size	Metadata (Padding) Size	Command Table size
4096 bytes	3456 bytes	32+96+512+3456+4096 = 8192 bytes
8912 bytes	1408 bytes	32+96+512+1408+2048 = 4096 bytes

Table 3 Command Table Size When Metadata Not Supported

- *Program all PxSEBxxBase and PxSEBxxINFO registers with the starting Status Event buffer address, associated MSI-X vector and number of entries in the buffer.*
- *Call InitIdtQ to initialize Command Header/Table structures, FreeQ and the associated variables. More queue management routines are described in Section 4.4.6*
- *In _CORE_CMD_INFO structure, initialize the following variables :*
  – FIFOEntries as the number of FIFO slots supported by Controller per processor
  – MaxEntries as the number of allocated Command Header/Table/Status entries.
  – CmdHdrStart and CmdHdrSts as 64-bit physical starting address.
  – Requests, Completions and OutStandingIO as 0.
  – FreeQEntries as FIFOEntries
  – CoreNumber and MSIXVector as the assoicated numbers.
  – Call StorPortInitializeDpc to initialize same number of DPC objects as MaxEntries, each object is associated with a Command Header by programming the object in pDPC of _CMD_HEADER.
- *Call StorPortInitializePerfOpts to set up NUMA I/O optimizations :*
  – Enable concurrnet channel I/O and set up the number of concurrent channel as the number of CPU processors.
  – Enable dynamic interrupt redirection and range of the MSI-X vectors supported.
  – Enable DPC redirection.
- *Enable interrupt for all desired sources via PxIE register if supported.*
- *If MSI-X is used, do the followings to enable interrupt:*
  – Set MXE bit of MXC as 1
  – Clear FM bit of MXC to 0
  – Clear all the associated bits of the supported MSI-X vectors in MSI-X Table to 0
- *If any of above steps fails, call StorPortLogSystemEvent with SP_UNEXPECTED_DISCONNECT and IDT_DRV_INIT_CMD, and return FALSE (Step 8). Otherwise, return TRUE to indicate that the controller is ready for processing requests (Step 9).*

4.1.4 IdtExit

```
VOID IdtExit (
  PDEVICE_EXTENSION pDE, // Pointer to an allocated DEVICE_EXTENSION structure
```

Notes

```
);
```

In the driver exit function, IdtExit, the followings need to be done before driver gets unloaded:

- *Disable interrupt.*
- *Force a Flush command with P bit set to flush the controller cache.*
- *Abort all the queued requests and return SRB_STATUS_SELECTION_TIMEOUT.*
- *Free memory allocated for Command Headers/Tables and Status Event buffer.*
- *Free other allocated resources.*

4.2 I/O Routines

The main I/O routines required in the miniport driver are IdtBuildIo and IdtStartIo, which are exposed to Storport driver as callback functions in HW_INITIALIZATION_DATA structure. With the new asynchronous I/O process, Storport driver pass down SRBs to IdtBuildIo first for pre-processing the SRBs. It returns TRUE when there is no error situations to notify Storport that it can call IdtStartIo for the SRB immediately. Otherwise, Storport will abort the request.

In IdtStartIo, IDT's miniport driver enables greater I/O concurrency and performance by taking advantage of MSI-X and NUMA features. The I/O processing diagram is illustrated in Figure 3 and more implementation details in the I/O routines are elaborated in this section.

4.2.1 IdtBuildIo

```
BOOLEAN IdtBuildIo (
    PVOID Context, // Pointer to an allocated DEVICE_EXTENSION structure
    PSCSI_REQUEST_BLOCK pSrb // Request block
);
```

IdtBuildIo is the entry point when Storport driver passes down an I/O request to IDT's miniport driver (Step 1). In IdtBuildIo, the first thing to do is ensuring the current state of both the controller and the driver is ready for processing requests. If not, return FALSE in Step 2. Otherwise, proceeds to find out if data transfer between host and device is necessary by SRB Function and SrbFlags. If not, simply return TRUE. Otherwise, do the followings (Step 3):

- *Refer to the Cdb to find out starting sector and transfer size for the request*
- *Call StorPortGetScatterGatherList to retrieve the pointer to the Scatter/Gather list*
- *Convert the Scatter/Gather elements into PRDT and PRD Index entries pointered by PtrPRDT and PtrPRDI of SrbExtension, respectively*
- *It finishes by returning TRUE to tell Storport driver that miniport driver is ready to process the request (Step 4).*

Notes

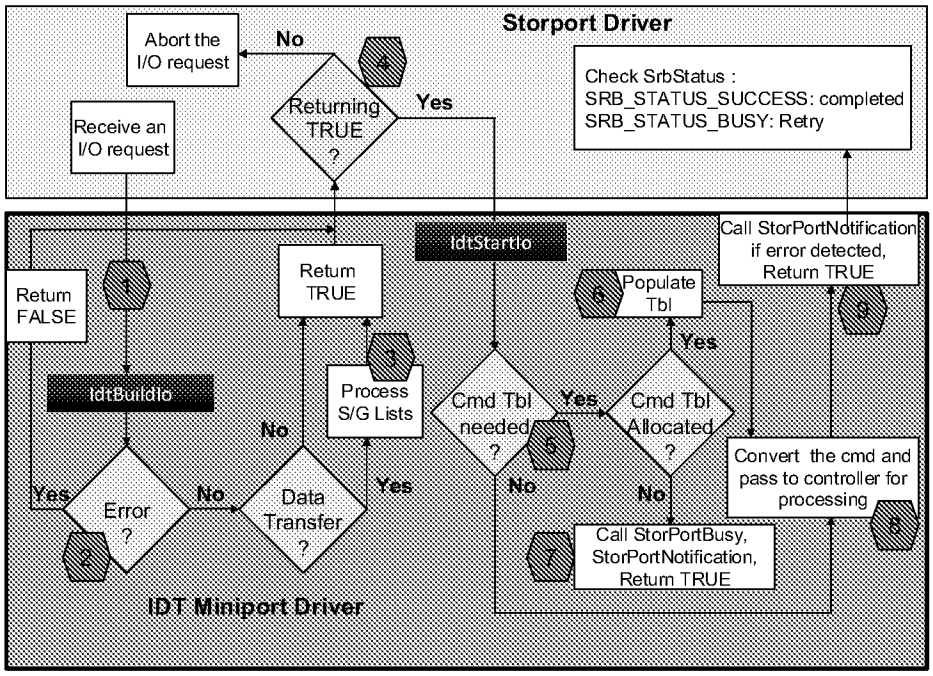


Figure 5 I/O Request Processing Diagram

4.2.2 IdtStartIo

```
VOID IdtStartIo (
    PVOID Context, // Pointer to an allocated DEVICE_EXTENSION structure
    PSCSI_REQUEST_BLOCK pSrb // Request block
);
```

IdtStartIo is the main function processing the I/O request received from Storport driver after pre-processing had been done in IdtBuildIo. IdtStartIo dispatches the requests based on the SRB Function in Step 5. The following steps are engaged to complete a given request:

- *Check DeviceState and DeviceFlags to see if the device is in normal state to process the request. If not, do the followings:*
  - Set SrbStatus as SRB_STATUS_BUSY to force Storport driver to re-send the request.
  - Call StorPortNotification with RequestComplete and return TRUE.
- *Dispatch the request based on its SRB->FUNCTION.*
- *For SRB_FUNCTION_EXECUTE_SCSI, call IdtProcessRW to complete the request.*
- *For SRB_FUNCTION_RESET_BUS:*
  - Call StorPortPause to block any new I/O.
  - Set IDT_DEV_STATE_RESET bit in DeviceState.
  - Call StorPortSynchronizeAccess with function parameter, IdtSyncReset.
- *For SRB_FUNCTION_POWER, if the DevicePowerState is StorPowerDeviceD3:*
  - If the device is still accessible, disable interrupt, call IdtAbortIo to abort any pending requests and clear all pending interrupts.
  - Set IDT_DEV_STATE_RESET bit in DeviceState.
  - Call StorPortLogSystemEvent with SP_UNEXPECTED_DISCONNECT.
  - *Call StorPortNotification with RequestComplete and return TRUE.*
- *For SRB_FUNCTION_IO_CONTROL:*
  - Make sure the IOCTL code is valid and the provided buffer is big enough. If not:
    - Call StorPortLogSystemEvent with SP_PROTOCOL_ERROR/IDT_DRV_ERROR_IO_BUF.
    - Set SrbStatus as SRB_STATUS_INVALID_REQUEST.
    - Call StorPortNotification with RequestComplete and return TRUE.

Notes

- Call IdtProcessIOCTL to engage further process for the proprietary defined IOCTL codes.

◆ *For other unsupported SRB function codes:*

- Call StorPortLogSystemEvent with SP_PROTOCOL_ERROR.
- Set SrbStatus as SRB_STATUS_BAD_FUNCTION.
- Call StorPortNotification with RequestComplete and return TRUE.

To summarize, refer to the following table for SRB function codes:

SRB Function Codes	Supported ?	Descriptions
SRB_FUNCTION_EXECUTE_SCSI	Yes	Most I/O related requests
SRB_FUNCTION_ABORT_COMMAND	Yes	Abort the SRB pointed by NextSrb
SRB_FUNCTION_RESET_DEVICE	Yes	Handled by IdtReset
SRB_FUNCTION_RESET_LOGICAL_UNIT	Yes	Handled by IdtReset
SRB_FUNCTION_RESET_BUS	Yes	Handled by IdtReset
SRB_FUNCTION_RELEASE_RECOVERY	No	
SRB_FUNCTION_RECEIVE_EVENT	No	
SRB_FUNCTION_SHUTDOWN	Yes	Called to unload the driver
SRB_FUNCTION_FLUSH	Yes	Called to flush cache, no hardware caching and simply return SUCCESS
SRB_FUNCTION_IO_CONTROL	Yes	Called to process proprietary defined IO control codes defined in Table 5
SRB_FUNCTION_LOCK_QUEUE	No	
SRB_FUNCTION_UNLOCK_QUEUE	No	
SRB_FUNCTION_DUMP_POINTERS	No	

Table 4 Summary of SRB Function Codes

4.2.3 IdtProcessRW

```
VOID IdtProcessRW (
  PDEVICE_EXTENSION pDE, // Pointer to an allocated DEVICE_EXTENSION structure
  PSCSI_REQUEST_BLOCK pSrb // Request block
);
```

This routine is meant to process SRB_FUNCTION_EXECUTE_SCSI requests, which require data transfer between system memory and the NVM subsystem. It follows the steps below :

1. CallCall StorPortGetCurrentProcessorNumber to decide which Command Header/Table structure and MSI-X vector to use for the transfer.
2. Call AcqQentry to allocate a Command Header from the FreeQ associated with the processor. If the FreeQEntries is equal to the threshold value saved in FIFOThreshold (Step 7),
   - Or DeviceStatus with IDT_DEV_STS_STORPORTQ bit if the bit is not set.
   - Call StorPortBusy to indicate the number of requests that the adapter must complete before resuming I/O requests to IDT's miniport driver. Storport driver queues the current request and retry it later.
   - Call StorPortNotification with RequestComplete to the end of processing the request.
   - Set SrbStatus as SRB_STATUS_BUSY
   - Return TRUE
3. Set up Command Header fields properly (Step 6), including:
   - PRDTL (the number of PRDT entries)
   - DT (indicate if meta data or not)
   - Write bit (1 means write, 0 means read)
   - Interrupt bit as 1

Notes

- MSI-X vector associated with the current processor

4. Copy the prepared PRDT and PRD Index entries, pointed by PtrPRDT and PtrPRDI respectively, to the Command Table.
5. Convert the SCSI command encoded in Cdb of SRB into eNVMHCI command, populate the Command field in Command Table, issue the command to Controller for processing by programming Doorbell_XBAregisters(Step 8).
6. If there is any errors or failures detected, set up the proper SRB status in SrbStatus and Call StorportNotification, then return TRUE. Otherwise, simply return TRUE (Step 9).

4.2.4 IdtProcessIOCTL

```
BOOLEAN IdtProcessIOCTL (
  PDeviceExtension pDE, // Pointer to an allocated DEVICE_EXTENSION structure
  PSCSI_REQUEST_BLOCK pSrb // Request block
);
```

This routine is implemented to process SRB_FUNCTION_IO_CONTROL requests, which indicates the request is sent with one of the proprietary defined IOCTL codes (See Table 2). The following steps are required:

- *Cast Srb->DataBuffer as PSRB_IO_CONTROL pointer. The IOCTL code is encoded in PSRB_IO_CONTROL->ControlCode. For the unrecognized codes, set Srb->SrbStatus as SRB_STATUS_INVALID_REQUEST, otherwise, set it as SRB_STATUS_SUCCESS when completed.*
- *Process the encoded control codes with an allocated command table, for example, IDT_IOCTL_QUERY_OUTSTANDINGIO only need to copy pDE->OutstandingIOCnt to ReturnCode of PSRB_IO_CONTROL.*
- *For the control codes that require a command table, call IdtBuildCmd with the specified command defined in NVMHCI_CMD.*
- *After all necessary processes are completed, call StorPortNotification with RequestComplete and return TRUE.*

Proprietary IOCTL Codes	Descriptions
IOCTLOP_GET_SECTOR_SIZE	Get sector size of the NVM device
IOCTLOP_GET_PAGE_SIZE	Get page size in sectors of the NVM device
IOCTLOP_GET_MIN_RW	Get the recommended minimum Read(bit0-15)/ write(bit16-31) size in NVM pages.
IOCTLOP_GET_SECTOR_ALIGN	Get the sector offset within the NVM page where the first sector is placed.
IOCTLOP_GET_LINEAR_CMD	Get the allowance if capable of processing command out of order
IOCTLOP_SET_LINEAR_CMD	Enable/Disable the capability of processing command out of order via Set Features command.
IOCTLOP_IO_COUNT_QUERY	Vendor specific/defined to query current out-standing IOs in NVM subsystem.

Table 5 Proprietary defined IOCTL Codes

4.2.5 IdtAbortIo

```
BOOLEAN IdtAbortIo (
  PDeviceExtension pDE, // Pointer to an allocated DEVICE_EXTENSION structure
);
```

Notes

This function is called to abort all outstanding requests with the following steps:

- *Firstly, clear PxCMD.ST as 0 to prevent Controller processing any commands*
- *Loop thru each Command Header, if Used is not zero (which means the entry is being used by a outstanding request), set SrbStatus as SRB_STATUS_SUCCESS and call StorPortNotification for the request*
- *Re-initializes Command Header/Table/Status structure, FreeQ and its associated variables to contain all the header entries by calling InitIdtQ.*

4.2.6 Interrupt Sources

The miniport driver hooks up the interrupt sources described in the table below. MSI-X vector 0 is used for the error or other management interrupts while other vectors are used to associate CPU cores for I/O Command Complettion interrupts.

Interrupt sources	Handlings
Host Bus Fatal Error	Abort all the outstanding requests in the miniport driver's request queue, notify Storport driver no more requests allowed and return SRB_STATUS_ERROR.
Host Bus Data Error	Return SRB_STATUS_SUCCESS for the current request.
Fatal Error	Abort all the outstanding requests in the miniport driver's request queue, notify Storport driver no more requests allowed and return SRB_STATUS_ERROR.
Command Completion	Check other PxIS bits to see if there is any errors associated with the completion. If yes, return appropriate error code, otherwise, return success.

Table 6 Interrupt sources and handlings

4.2.7 IdtISR

```
BOOLEAN
IdtISR (
  PVOID pDE, // Pointer to an allocated DEVICE_EXTENSION structure
);
```

This function is the callback function registered in HW_INITIALIZATION_DATA as legacy INTx interrupt service routine. Since the Controller supports MSI-X capability, this function simply return TRUE for now.

4.2.8 IdtMSIISR

```
BOOLEAN
IdtMSIISR (
  PVOID pDE, // Pointer to an allocated DEVICE_EXTENSION structure
  ULONG MsgVector, // MSI-X vector
);
```

When any one of the interrupt sources gets triggered, IdtIMSISR is invoked to handle the interrupt. However, the driver does the followings to ensure it needs to service the interrupt:

- *Ensure the passed-in parameters are valid, if pDE is NULL or MsgVector is out of supported range:*
  - Log error with SP_PROTOCOL_ERROR/IDT_DRV_ERROR_INVALID_PARAM
  - Return FALSE
- *Disable interrupt by masking out the MsgVector in MSI-X Table.*
- *Confirm the Controller is in normal state by checking bits in DeviceState. If any bit is set:*
  - Log error with SP_INTERNAL_ADAPTER_ERROR/IDT_DRV_ERROR_DEV_DOWN
  - Return FALSE.
- *Then read back the PxIS register (if supported) to make sure it's our interrupt. If no bit is set, return FALSE*
  - Log error with SP_PROTOCOL_ERROR/IDT_DRV_ERROR_BOGUS_INT

Notes

- Return FALSE

- *Now, the further processing requires using the CurStatusPtr associated with the current CPU core to locate the correct entry in the associated Command Status Event buffer.*
- *Fetch the Command Tag number and the I/O completion status saved in Entry Description field*
- *If there is any errors, Log error with SP_INTERNAL_ADAPTER_ERROR/ IDT_CMD_ERROR_ABORT and set SrbStatus accordingly*
- *Set SrbStatus as SRB_STATUS_SUCCESS and move the CurStatusPtr to next entry*
- *Based on the Command Tag number, locate the correct entry in the Command Header list and its physical address. Convert it into virtual address with StorPortGetVirtualAddress and find LIST_ENTRY associated with the Command Header entry.*
- *Optionally, call KeSetTargetProcessorDpc to specify the current processor that runs the DPC routine.*
- *Call StorPortIssueDpc with the DPC object associated with Command Header entry, SrbExtension and DEVICE_EXTENSION to queue the DPC object for later execution.*

4.2.9 IdtDpc

```
VOID
IdtDpc (
   PSTOR_DPC Dpc, // Pointer to the associated DPC object
   PVOID pDE, // Pointer to an allocated DEVICE_EXTENSION structure
   PVOID SystemArgument1 // SrbExtension
   PVOID SystemArgument2, // Command Header entry
);
```

If there is no error found, for Flush, Read/Write and Set Features commands, *add the header entry back to FreeQ by calling AppQentry after clearing PRDTL, DT, Used and W fields in Command Header, and setting, Command, PRDT and PRD Index Table regions in Command Table as 0s.*

For Get Features, Identify and Get Status commands, the read back information needs to be compiled into appropriate data structure or variables in the DataBuffer provided in SRB. The Command Header entry is added back to FreeQ in same manner as Flush, Write and Set Features commands.

After free the Command Header, add 1 to Completions. Clear the Valid bit of the Command Status Event entry to indicate the entry is ready for another request. To complete the request, the miniport driver needs to notify Storport driver by calling StorPortNotification with RequestComplete, pDE and the original SRB.

After moving the CurStatusPtr to next Command Status Event entry, check its Valid bit to see if it's set or not. If yes, process the entry as above. If no, re-enable the interrupt by clearing the associated bit of the MSI-X vector and return. Whenever moving the CurStatusPtr, checking to see if it reaches the end of the entries. If yes, point it to the first entry.

4.3 PnP, Power Management and Error Recovery

This section is dedicated to the implementation assoicated with Windows PnP, Power Management and error recovery processing. There are two callback functions that need to be registered via HW_INITIALIZATION_DATA structure to serve as entry points when handling PnP and Power Management related requests and error recovering situations.

4.3.1 IdtAdapterControl

```
SCSI_ADAPTER_CONTROL_STATUS
IdtAdapterControl (
   PVOID pDE, // Pointer to an allocated DEVICE_EXTENSION structure
   SCSI_ADAPTER_CONTROL_TYPE ControlTypeSrb // Windows defined SCSI control type
   PVOID pParameters, // Pointer to parameters needed for this type (optional)
);
```

This routine is implemented to process Windows defined SCSI related control types and return either ScsiAdapterControlSuccess or ScsiAdapterControlUnsuccessful. The control types IDT's miniport driver supports are:

Notes

- *ScsiQuerySupportedControlTypes.*

This control code is used to determine which control types are supported by IDT's miniport driver such that driver can cast pParameter to PSCSI_SUPPORTED_CONTROL_TYPE_LIST and specify the supported types, which include ScsiQuerySupportedControlTypes, ScsiStopAdapter and ScsiRestart-Adapter.

- *ScsiStopAdapter.*

When receiving this control type, the miniport driver checks IDT_DEV_STATE_STOP bit in DeviceState first, if the adapter already stop, simply return ScsiAdapterControlSuccess. Otherwise, disable interrupt, call IdtAbortIo to call off all the pending IOs, and set IDT_DEV_STATE_STOP bit in DeviceState.

- *ScsiRestartAdapter.*

When receiving this control type, retrieve pre-defined values from Registry, re-initialize Command Header/Table/Status structures, FreeQ and the related variables, ensure the adapter is in normal state by verify registers' bits are programmed properly, and clear IDT_DEV_STATE_STOP bit in DeviceState. Return ScsiAdapterControlSuccess if none of the re-initialization steps fails.

4.3.2 IdtReset

```
BOOLEAN IdtReset (
  PVOID pDE, // Pointer to an allocated DEVICE_EXTENSION structure
  ULONG PathID, // Indicates the controller to reset
);
```

This function gets called to reset the controller and the miniport driver itself when a bus reset happens. It does the followings to handle the situation:

- *Call StorPortPause to block any new I/O requests.*
- *Set IDT_DEV_STATE_RESET bit in DeviceState*
- *Call StorPortSynchronizeAccess with function parameter IdtSyncReset to start re-initialization*
- *Return TRUE*

4.3.3 IdtSyncReset

```
BOOLEAN IdtSyncReset (
  PVOID pDE, // Pointer to an allocated DEVICE_EXTENSION structure
  PVOID NotUsed, // Dummy pointer
);
```

This function gets called to really reset the controller and the miniport driver itself by doing the followings:

- *Loop through all Command Header entries to see if there is any pending requests.*
- *If yes, abort all outstanding I/O requests.*
- *Reset the controller*
- *Re-initialize Command Headers/Tables and Status entries with InitIdtQ*
- *Re-initialize FreeQ and the associated variables with InitIdtQ*
- *Re-enable MSI-X interrupt*
- *Call StorPortResume to notify Storport driver resuming I/O requests.*
- *Clear IDT_DEV_STATE_RESET bit in DeviceState*
- *Return TRUE*

4.4 I/O Service

The I/O Service module details how miniport driver is implemented to takes advantage of MSI-X and NUMA to boost I/O performance. Associated with each active processors are a dedicated MSI-X vector and a Command Headers/Table/Status structure. This section describes how these resources are acquired, configurated during driver initialization phase, how they are managed per processor basis during run time, how the queues are managed in miniport driver to work with MSI-X, NUMA I/O and StorPort driver scheme.

Notes

4.4.1 Registry Initialization for MSI-X

By default, IDT's miniport driver supports MSI-X, which requires its INF file to set up necessary layout in Registry under HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\PCI\VEN_111d&DEV_hhhh entry as below:

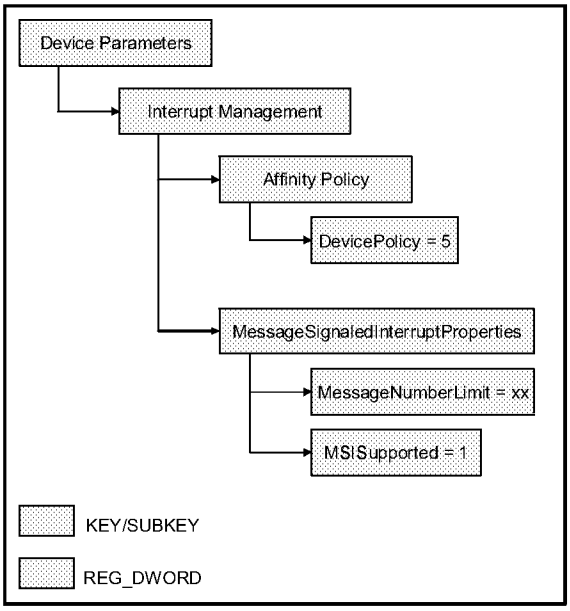


Figure 6 Registry Key Setup for MSI-X Support

The DevicePolicy is set as 5 (IrqPolicySpreadMessagesAcrossAllProcessors) to specify the system assigns different message-based interrupts to different processors. MessageNumberLimit is used to indicate maximum message interrupts supported. Lastly, MSISupported is set as 1 to indicate MSI-X is supported.

4.4.2 MSI-X Related Configuration

While IDT's Miniport driver is initializing, the following MSI-X related configuration is required:

- *In IdtInitalize, IDT's miniport driver calls StorPortGetMSIInfo to collect all the information associated with each message, including its vector ID, physical address and data.*
- *In addition to that, same number of DPC (Deferred Procedure Call) objects as MaxEntries are created with StorPortInitializeDpc to associated each MSI-X vector and specify its associated routine to handle the post-interrupt issues, such as releasing the Command Header entry and update current number of requests, to shorten the processing in interrupt service routine.*
- *In IdtFindAdapter, the MSI interrupt service routine is specified as IdtMSIISR and interrupt synchronization mode is also specified as InterruptSynchronizePerMessage in the PORT_CONFIGURATION_INFORMATION.*

4.4.3 NUMA I/O Related Configurations

The following initializations are required to configure NUMA I/O properly in IdtInitialize function:

- *The number of NUMA nodes in current system needs to be retrieve by calling StorPortGetHighestNodeNumber. If it returns zero, which means it's not an NUMA supported system. Set CanNUMA of DeviceExtension as FALSE and skip all the following NUMA related configurations.*
- *If NUMA I/O is supported, call StorPortAllocateContiguousMemorySpecifyCacheNode to allocate Command Header/Table and Status Event buffers per CPU core and initialize the queues when successfully allocated. If any allocations fail, reduce the size and start 2nd round of allocation for*

Notes

them.

- *Call StorPortInitializePerfOpts with a populated PERF_CONFIGURATION_DATA structure.* The ConcurrentChannels is used to specified the number of concurrent calls to IdtStartIo that IDT's miniport driver supports. This number is determined by the number of active CPU cores. *Passing Query as FALSE to force Storport initialize the optimizations by the flags specified in the structure. If it returns other than STOR_STATUS_SUCCESS, call again with QUERY as TRUE to allow Storport driver initialize the optimization it supports. The specified optimization flags listed in Table 4 are bitwise-ORed.*

Flags	Meanings
STOR_PERF_DPC_REDIRECTION	DPC processing should be spread out over multiple CPUs
STOR_PERF_CONCURRENT_CHANNELS	Concurrent calls to the IdtStartIo routine are supported
STOR_PERF_INTERRUPT_MESSAGE_RNAGES	Interrupt redirection is supported
STOR_PERF_OPTIMIZE_FOR_COMPLETE_DURING_STARTIO	IDT's miniport driver will complete I/Os concurrently with the submission of new I/Os

Table 7 DPC/Interrupt Redirection Flags

4.4.4 Runtime Processing

When IdtStartIo gets called to process an I/O request that requires data transfer with Command Header/Table and Status Event buffer, the following MSI-X and NUMA I/O related processing are mandatory to enhance the performance:

1. Call StorPortGetScatterGatherList and based on the Scatter/Gather list and Cdb, miniport driver decides it's a read or write command and how many PRTD entries and PRD Index entries are required.
2. Call StorPortGetCurrentProcessorNumber to decide which Command Header/Table structure to use for the transfer.
3. For a given request, process it as the followings:
   - *Allocate a Command Header entry for the request from FreeQ.*
   - *Call StorPortGetStartIoPerfParams with STARTIO_PERFORMANCE_PARAMETERS structure to retrieve the MSI-X vector and link the current CPU with which to interrupt when I/O is completed.*
   - *Populate the fields in the header, including the associated MSI-X vector and set Used as 1.*
   - *Populate the Command, the PRDT entries and PRD Index entries in the Command Table.*
   - *Update Requests, FreeQEntries, etc*
   - *Save the Command Header entry, MSI-X vector and processor number in SrbExtension.*
   - *Issue the command via Doorbell_XBA regsiters*
   - *Set Srb->SrbStatus = SRB_STATUS_SUCCESS*
   - *Call StorPortNotification with RequestComplete flag and return TRUE*

Notes

When IdtMSIISR gets called due to I/O completions (which can be decided by non-zero MSI-X vector), the following steps are followed to completed the I/O:

1. With the MSI-X vector, IDT's miniport driver figures out the CPU core on which it's running and which CurStatusPtr it should use to locate the correct Command Status Event entry. From the entry, it looks into the associated I/O completion status in Entry Decription field and Command Tag number.
2. Move the CurStatusPtr to next entry
3. If the value of Entry Description is zero, which means I/O is completed successfully.
4. Based on the Command Tag number, locate the correct entry in the Command Header list and its physical address. Convert it into virtual address with StorPortGetVirtualAddress and find LIST_ENTRY associated with the Command Header entry.
5. Optionally call KeSetTargetProcessorDpc to specify the current processor that runs the DPC routine.
6. Call StorPortIssueDpc with the DPC object associated with Command Header entry, SrbExtension and DEVICE_EXTENSION to queue the DPC object for later execution.

When IdtDpc routine gets called, the following issues are completed:

1. Releasing the Command Header entry back to FreeQ
2. Update variables like Completions and FreeQEntries
3. Set SrbStatus accordingly
4. Call StorPortNotification with RequestComplete, pDE and the associated SRB.

4.4.5 Command Header Queue Initialization

Whenever the Command Header/Table strcuture and FreeQ need to be initialized, for each CPU core, InitIdtQ initializes the Command Header/Table structures depending on the the following variables:

- *In DeviceExtension:*
  - MaxPRDTEntries
  - MaxMetadataSize
  - MaxPages
- *In _CORE_CMD_INFO*
  - MaxEntries
  - CmdHdrStart

Here are the tasks need to be done:

- *Populate static fields of each _CMD_HEADER structure*
- *Initialize FreeQ its associated variables*
- *All the Command Headers are added into FreeQ initially*

4.4.6 Queue Management Routines

- *BOOLEAN InitIdtQ (PDEVICE_EXTENSION pDE);*
  - Initialize FreeQ with InitializeListHead
  - Add all Command Headers into FreeQ with InsertTailList
  - Return TRUE if succeeds, otherwise FALSE
- *PLIST_ENTRY AcqQEntry(PLIST_ENTRY pList, BOOLEAN FromTail);*
  - Acauire an entry from FreeQ
  - The entry can be removed from the tail or head of the queue, determined by FromTail
  - Call RemoveTailList or RemoveHeadList
  - Return the entry if succeeds, otherwise return NULL pointer
- *BOOLEAN AppQEntry(PLIST_ENTRY pList, PLIST_ENTRY pListEntry, BOOLEAN FromTail)*
  - Insert an entry into FreeQ
  - The entry can be inserted at the tail or head of the queue, detemined by FromTail
  - Call InsertTailList or InsertHeadList
  - Return TRUE if succeeds, otherwise FALSE

Notes

- *VOID EmptyQ(PLIST_ENTRY pList)*
  - Empty a given queue

4.4.7 Queue Manipulation

In normal situation, whenever an I/O request requires a Command Header/Table entry to complete, the entry can be acquired from FreeQ when available. The pre-processing for an I/O request in IdtBuildIo has looped through the scatter/gather list to constructed PRDT and PRD Index entries.

In IdtStart, it calls AcqQEntry to have a Command Header ready for the request and copies the PRDT and PRD Index entries into the Command Table in addition to setting up other fields in the Command Header/Table, and then issues the command to the Controller.

When IdtMSIISR gets called due to an I/O completion, with the passed-in MSI-X vector, it figures out which CPU Core the routine is running on and locates the correct Command Status entry with CurStatusPtr. From the Command Tag field in the 32-bit Status Event entry, retrieve the Tag number that can be used to locate the physical address of the Command Header entry. When IdtDpc gets called, the entry will be added back into FreeQ when completing the I/O processing.

4.5 eNVMHCI I/F

This module handles the conversions between the accepted SRB and eNVMHCI Commands. When IdtBuildIo pre-processes the SRB, after some first level error screening without finding any errors, such as:

- *Is the Function supported ?*
- *Is the data buffer big enough ?*
- *Is the CdbLength correct ?*
- *Are the PathId, TargetId and Lun correct ?*
- *Does it require Command Table to complete the SRB ?*

If the answer to the last question is yes, then it calls IdtBuildCmd for the SRB.

4.5.1 IdtBuildCmd

```
BOOLEAN IdtBuildCmd (
   PDeviceExtension pDE, // Pointer to an allocated DEVICE_EXTENSION structure
   PSCSI_REQUEST_BLOCK pSrb, // Request block
   enum NVMHCI_CMD cmd // the decided command to build
);
```

This function is called by IdtBuildIo to pre-process the requests that require command header/table. Based on the SRB Function and CDB in SRB structure, Table 5 includes the supported mapping to eNVMHCI defined commands.

SRB Function	Command Descriptor Block	eNVMHCI Command
SRB_FUNCTION_EXECUTE_SCSI	SCSIOP_READ_CAPACITY16	Identify
SRB_FUNCTION_EXECUTE_SCSI	SCSIOP_READ16 or SCSIOP_READ12	Read
SRB_FUNCTION_EXECUTE_SCSI	SCSIOP_WRITE16 or SCSIOP_WRITE12	Write
SRB_FUNCTION_IO_CONTROL	IOCTLOP_GET_LINEAR_CMD	Get Features - Get the allowance if capable of processing command out of order
SRB_FUNCTION_IO_CONTROL	IOCTLOP_SET_LINEAR_CMD	Set Features - Enable/Disable the capability of processing command out of order

Table 8 SRB to eNVMHCI command mapping

Notes

For NVMHCI_READ and NVMHCI_WRITE commands, get a pointer to Storport scatter/gather list by calling StorPortGetScatterGatherList with pDE and pSrb. The pointer, pSSGL, is pointing to STOR_SCATTER_GATHER_LIST. The pSSGL->NumberOfElements indicates how many list entries included in this request and it is saved into pSrbExt->ElementCount. The pSSGL->List is the first element and it's assigned to pSSGE for looping through the list.

When it loops through the entries, the pSSGE->length is used to determine the size of a PRDT entry that is used to transfer the data. The length is put down in DBC field. The sum of all the DBCs indicates the total transfer size using the Command Header/Table entry acquired by calling AcqQEntry function.

After some pre-processing, it calls IdtPRDProcess with the following parameters to construct the entries:

- *pSrbExt: The SRB extension associated with the request.*
- *pCHE: Pointer to an allocated Command Header entry*
- *pSSGL: Pointer to the SG List*
- *Offset: Sector offset to where start to transfer with the command table. If not zero, the Elements must be 1 to indicate the multiple command tables are used for a single element.*
- *SectorLo: The starting sector address, lower 32-bits. It's constructed by ((PCDB)pSrb->Cdb)->CDB16.LogicalBlock[0] (MSB) to LogicalBlock[3](LSB).*
- *SectorHi: The starting sector address, upper 32-bits. It's constructed by ((PCDB)pSrb->Cdb)->CDB16.LogicalBlock[4] (MSB) to LogicalBlock[7](LSB).*
- *Sectors: The number of sectors to transfer. It's constructed by ((PCDB)pSrb->Cdb)->CDB16.TransferLength[0] (MSB) to TransferLength[3] (LSB).*

For all the supported commands, the 32-byte Command Header pointed by pCmdHdr, requires to be programmed as below:

```
pCmdHdr->Write = 1; //for data transfer from host to device, 0 otherwise.
pCmdHdr->DataTransfer = 2; //data transfer only
pCmdHdr->PRDTL = pCmdTbl->PRDTEntries;
```

Besides, the starting 32 bytes of Command Table, pointed by pCmdTbl, is programmed based on the command type, which has its own structure defined in _CMD_PARAMETERS in Section 4.5.3.

4.5.2 IdtPRDProcess

```
BOOLEAN IdtPRDProcess (
   PSRB_EXTENSION pSrbExt, // Pointer to the SRB extension
   PCMD_HEADER_ENTRY pCHE, // Pointer to an allocated Command Header entry
   PSCSI_SCATTER_GATHER_LIST pSSGL, // Pointer to the SG List
   ULONG Offset, // Sector offset to where start to transfer with the command table
   ULONG32 SectorLo, // The starting sector address, lower 32-bits
   ULONG32 SectorHi, // The starting sector address, upper 32-bits
   ULONG32 Sectors, // The number of sectors to transfer
);
```

This function is mainly for constructing PRDT and PRD Index entries within command table. Firstly, we assume the sector size is 512 bytes and NVM page size is 4096 bytes. For each scatter/gather element, the pSSGE->Length is examined to decide whether the element requires multiple PRDT entries or not so that we can call different function to complete the PRDT and PRD Index entries construction.

The pseudo codes below demonstrating how to construct the PRDT and PRT Index entries for READ and WRITE commands on each scatter/gather base:

```
#define SECTORSIZE 512
#define NVMPAGESIZE 4096
#define SECTORS1PAGE (NVMPAGESIZE / SECTORSIZE)
#define MAXSIZE     (256 * 1024)
#define MAXSECTOR   (MAXSIZE / SECTORSIZE)

ULONG BuildSinglePRDEntry (PCMD_TBL pCmdTbl,
                           PSTOR_SCATTER_GATHER_ELEMENT pSSGE,
```

Notes

```
                                        ULONG SectorLo) // Starting sector of transfer
{
   ULONG32 NextOffset = 0;
   // Number of sectors to transfer
   ULONG32 TotalSectors = (pSSGE->Length / SECTORSIZE) + (pSSGE->Length % SECTORSIZE) ? 1 : 0;
   ULONG32 StartSector = SectorLo & 0x7; // 0 based
   ULONG32 Sectors1stPage = SECTORS1PAGE - StartSector;
   ULONG32 Sectors = TotalSectors;
   PPRD_INDEX_ENTRY pPRDIdxEntryTemp = pCmdTbl->pPRDIdxEntry;

   //Set up PRD Entry here
   pCmdTbl->pPRDEntry->DBAL = pSSGE->PhysicalAddress.LowPart;
   pCmdTbl->pPRDEntry->DBAU = pSSGE->PhysicalAddress.HighPart;
   pCmdTbl->pPRDEntry->DataByteCnt = (pSSGE->Length - 1) | 3;
   pCmdTbl->pPRDEntry->Reserved0 = TotalSectors;;

   // First PRD index entry
   pPRDIdxEntryTemp->Offset = NextOffset;
   pPRDIdxEntryTemp->PRDEntry = 0;
   // If the number of available sectors in first page is enough,  return the number of sectors transferred
   if (Sectors <= Sectors1stPage)
      return TotalSectors;
   pPRDIdxEntryTemp++;
   Sectors -= Sector1stPage;
   NextOffset = (Sector1stPage * SECTORSIZE) >> 2;
   while (1) {
      pPRDIdxEntryTemp->Offset = NextOffset;
      pPRDIdxEntryTemp->PRDEntry = 0;
      if (Sectors <= SECTORS1PAGE)
         return TotalSectors;
      Sectror -= SECTORS1PAGE;
      // Calculate next offset
      NextOffset += (NVMPAGESIZE >> 2);
      pPRDIdxEntryTemp++;
   }
}

ULONG BuildMultiplePRDEntry (PCMD_TBL pCmdTbl,
                             PSTOR_SCATTER_GATHER_ELEMENT pSSGE,
                             ULONG SectorLo,
                             ULONG StartingOS)
{
   // Multiple PRD entry is required due to pSSGE->Length > MAXSIZE
   ULONG32 NextOffset = StartingOS;
   // Number of sectors to transfer
   ULONG32 TotalSectors = (pSSGE->Length / SECTORSIZE) +  (pSSGE->Length % SECTORSIZE) ? 1 : 0;
   ULONG32 Sectors = TotalSectors;
   // Figure out how many entries needed and build them now
   ULONG Length = pSSGE->Length;
   PPRD_ENTRY pPRDEntryTemp = pCmdTbl->pPRDEntry;
   PPRD_INDEX_ENTRY pPRDIdxEntryTemp = pCmdTbl->pPRDIdxEntry;

   for (int PRDentry = 1; Length > MAXSIZE;) {
      pPRDEntryTemp->DBAL = pSSGE->PhysicalAddress.LowPart;
      pPRDEntryTemp->DBAU = pSSGE->PhysicalAddress.HighPart;
      pPRDEntryTemp->DataByteCnt = (MAXSIZE - 1) | 3;
      pPRDEntryTemp->Reserved0 = (MAXSECTOR);
      PSSGE->PhysicalAddress.LowPart += MAXSIZE;
      Length -= MAXSIZE;
      pPRDEntryTemp++;
```

Notes

```
        PRDentry++;
    }
    pCmdTbl->PRDEntries  = PRDentry;
    // Last entry constructed here
    pPRDEntryTemp->DBAL = pSSGE->PhysicalAddress.LowPart;
    pPRDEntryTemp->DBAU = pSSGE->PhysicalAddress.HighPart;
    pPRDEntryTemp->DataByteCnt = (Length - 1) | 3;
   pPRDEntryTemp->Reserved0 = (Length / SECTORSIZE);


    // Set up PRD Index Table entries now
    pPRDEntryTemp = pCmdTbl->pPRDEntry;
    ULONG32 StartSector = SectorLo & 0x7;
    ULONG32 Sectors1stPage = SECTORS1PAGE - StartSector;
    for (int i = 0; i < PRDentry - 1; i++) {
        ULONG32 MaxSectors = ((PPRD_ENTRY) pPRDEntryTemp + 1)->Reserved0;
        pPRDIdxEntryTemp->Offset = NextOffset;
        pPRDIdxEntryTemp->PRDEntry = i;
        pPRDIdxEntryTemp++;
        MaxSectors -= Sector1stPage;
        NextOffset = (Sector1stPage * SECTORSIZE) >> 2;
        while(MaxSectors) {
            pPRDIdxEntryTemp->Offset = NextOffset;
            pPRDIdxEntryTemp->PRDEntry = i;
            if (MaxSectors <= SECTORS1PAGE) {
                // Last index entry for current PRD Entry
                // Prepare NextOffset also
                NextOffset += ((MaxSectors * 512) >> 2);
                break;
            }
            MaxSectror -= SECTORS1PAGE;
            NextOffset += ((NVMPAGESIZE) >> 2);
            pPRDIdxEntryTemp++;
        }
        pPRDEntryTemp++;
    }
    // Set up PRD Index Table associated with last PRD Entry
    // If no more sector, return total sectors
    Sectors = pPRD(MAXSECTOR * (PRDentry - 1));
    if(!Sectors)
        return TotalSectors; // should not happen
    pPRDIdxEntryTemp->Offset = NextOffset;
    pPRDIdxEntryTemp->PRDEntry = PRDentry;
    if (Sectors <= Sectors1stPage)
        return TotalSectors;
    pPRDIdxEntryTemp++;
    Sectors -= Sector1stPage;
    nextOffset = (Sector1stPage * 512) >> 2;
    while (1) {
        pPRDIdxEntryTemp->Offset = NextOffset;
        pPRDIdxEntryTemp->PRDEntry = PRDentry;
        if (Sectors <= SECTORS1PAGE)
            return TotalSectors;
        Sectror -= SECTORS1PAGE;
        // Calculate next offset
        NextOffset += (NVMPAGESIZE >> 2);
        pPRDIdxEntryTemp++;
    }
}
```

Notes

4.5.3 eNVMCHI Command Data Structures

```
typedef enum NVMHCI_CMD {
   NVMHCI_FLUSH = 0x1,
   NVMHCI_READ = 0x10,
   NVMHCI_GET_FEATURES = 0x11,
   NVMHCI_IDENTIFY = 0x12,
   NVMHCI_GET_STATUS = 0x13,
   NVMHCI_WRITE = 0x20,
   NVMHCI_SET_FEATURES = 0x21,
   NVMHCI_DATASET_MANAGEMENT = 0x22,
   VENDOR_DEFINED = 0x80
};


//
// Commmand Status Event fields
//
typedef struct _CMD_SE {
   ULONG16 Valid: 1; // Valid bit, 1 indicates pending completed entry, driver needs to clear it
                     // after processing it and make it free for future use
   ULONG16 ED: 2; // Entry Description, 0 means successfully completed, otherwise, error
   ULONG16 Reserved: 12; // Reserved
   ULONG16 PageFree: 1; // Page Free for future use
   ULONG16 CmdTag; // Command Tag programmed by controller
} CMD_SE, *PCMD_SE;


typedef struct _CMD_HEADER { // aligned in 128 byte
   // Proprietary info in the first 96 bytes for queuing, etc
   ULONGLONG List; // The List entry for double linked-list (DW0 & 1).
   CMD_SE StatusEvent; // Assigned Status Event entry (DW2)
   ULONG16 CoreNumber: 15; // the processor number processing the I/O (DW3)
   ULONG16 Used: 1; // The entry is being used if not zero (DW3)
   ULONG16 PRDEntries; // Indicates the number of PRD index entries included in the table.
   ULONGLONG pSrb; // The original SRB associated with the request (DW4 & 5)
   ULONGLONG pDpc; // Dedicated DPC object associated with the entry (DW6 & 7)
   ULONGLONG CmdTblPhysAddr; // The physical addr of the associated Command Table (DW8 & 9)
   ULONGLONG CmdTblVirtAddr; // The virtual addr of the associated Command Table (DW10 & 11)
   CMD_PARAMETERS CmdParameter; // Which command parameter to use. (32 bytes, DW12 - 19)
   ULONGLONG pPRDTEntry; // first PRDT entry, casted as PPRDT_ENTRY* for access (DW20 & 21)
   ULONGLONG pPRDIEntry; // first PRD Index entry, casted as PPRDI_ENTRY* for access (DW22 & 23)
   // Command Header fields in eNVMHCI specification
   UCHAR CmdLen:4; // The length of the command to be transfer in DWORDs.
                   // Valid values are  from 1 to 8
   UCHAR PRDIPresent:1; // PRDI entry present, 1 means yes, 0 means no
   UCHAR Interrupt:1; // Set to '1', controller sets PxIS.CCS to '1' when completed.
                      // Clear to '0', controller shall not set PxIS.CCS to '1'.
   UCHAR Write:1; // Set to '1', indicates the transfer is from system memory to NVM device,
          // otherwise, from NVM device to system memory or no data transferring
   UCHAR DataTranser:2; // Indicates what data types involved in the transfer.
                        // 00b=no data or metadata, 01b=metadata only,
                        // 10b=data transfer only, 11b=both data and metadata transfer.
   UCHAR Reserved1:7; // Reserved field
   USHORT PRDTL; //  Length of the scatter/gather descriptor table in entries (0 to 65535)
   ULONG32 PITO; //The offset within Command Table of the PRD Index Table
   ULONG32 CTBAL; //The lower 32-bit of the physical address of the command table [06:00] = 0
   ULONG32 CTBAU; // The upper 32-bit of the physical address of the command table
   ULONG32 MO; // The offset within the Command Table of the metadata if DataTransfer[0] set to 1
```

Notes

```
    ULONG16 CmdTag; // Command Tag number associated with the Header entry
    ULONG16 Reserved2; // Reserved field
    UCHAR OverallCmdStatus; // Command status that is common for all commands
    UCHAR CmdStatus; // Command specific status for the associated command
    UCHAR Reserved3; // Reserved field
    UCHAR VendorStatus; // Vendor specific  status
    UCHAR MSIXVector; // MSI-X vector associated with the CPU Core, valid from 0 to 31
    UCHAR Reserved4; // Reserved field
    USHORT Reserved5; // Reserved field
} CMD_HEADER, *PCMD_HEADER;


typedef struct _PRD_ENTRY {
    ULONG32 DBAL; // The lower 32-bit of the physical address of the data block
    ULONG32 DBAU; // The upper 32-bit of the physical address of the data block
    ULONG32 Reserved0; // Reserved field, but used to indicate the number of sectors transferred by the entry
    ULONG32 DataByteCnt; //Zero-based value that indicates the length, in bytes, of the data block.
                         // Valid values are from 0 to 256KB. Bits[1:0] shall be 11b to indicate DWORD granularity
} PRD_ENTRY, *PPRD_ENTRY;


typedef struct _PRD_INDEX_ENTRY {
    USHORT PRDEntry; // the entry index pointing to PRDT
    USHORT Offset; // The offset in DWORDs to the DBAL of the PRD Entry
} PRD_INDEX_ENTRY, *PPRD_INDEX_ENTRY;


typedef union _CMD_PARAMETERS {
    // READ Command
    struct _CMD_READ {
        UCHAR Opcode; // 10h
        UCHAR Retry:1; // If set, this command is a retry.
        UCHAR AttrSet:1; //If set, indicates the Attributes are present and valid.
        UCHAR Workload:1; // If set, indicates the workload information is valid.
        UCHAR Reserved0:5; // Reserved field.
        USHORT Timeout; // The relative time from the command issue until the command should be completed by device.
        ULONG32 SSAL; // Lower 32-bit starting sector address.
        ULONG32 SSAU; // upper 32-bit starting sector address.
        USHORT Sectors; // The number of sectors for the transfer.
        USHORT Reserved1; // Reserved field.
        UCHAR Priority:2; // Indicates the priority of the request, from 0 (low) to 3 (critical).
        UCHAR UpcomingRead:2; // Indicates the number of queued read requests that are yet to be issued by driver.
        UCHAR UpcomingWrite:4; // Indicates the number of queued write requests that are yet to be issued by driver.
        UCHAR AvgWriteSize; // Average size in NVM pages of queued write requests that are yet to be issued by driver.
        USHORT Reserved2; // Reserved field.
        ULONG32 Attributes; // Command attributes specified, valid only when AttrSet is 1.
    } CMD_READ;

    // WRITE Command
    struct _CMD_WRITE {
        UCHAR Opcode; // 20h
        UCHAR Retry:1; // If set, this command is a retry.
        UCHAR AttrSet:1; //If set, indicates the Attributes are present and valid.
        UCHAR Workload:1; // If set, indicates the workload information is valid.
        UCHAR Reserved0:5; // Reserved field.
        USHORT Timeout; // The relative time from the command issue until the command should be completed by device.
        ULONG32 SSAL; // Lower 32-bit starting sector address.
        ULONG32 SSAU; // upper 32-bit starting sector address.
        USHORT Sectors; // The number of sectors for the transfer.
        USHORT Reserved1; // Reserved field.
        UCHAR Priority:2; // Indicates the priority of the request, from 0 (low) to 3 (critical).
        UCHAR UpcomingRead:2; // Indicates the number of queued read requests that are yet to be issued by driver.
```

Notes

```
    UCHAR UpcomingWrite:4; // Indicates the number of queued write requests that are yet to be issued by driver.
    UCHAR AvgWriteSize; // Average size in NVM pages of queued write requests that are yet to be issued by driver.
    USHORT Reserved2; // Reserved field.
    ULONG32 Attributes; // Command attributes specified, valid only when AttrSet is 1.
} CMD_WRITE;

// FLUSH Command
struct _CMD_FLUSH {
    UCHAR Opcode; // 01h
    UCHAR PowerDownFlush:1; // If set, the flush is being issued immediately prior to a power down condition.
    UCHAR Reserved0:7; // Reserved field.
    USHORT Timeout; // The relative time from the command issue until the command should be completed by device.
    ULONG32 Reserved1; // Reserved field.
    ULONG32 Reserved2; // Reserved field.
    ULONG32 Reserved3; // Reserved field.
    ULONG32 Reserved4; // Reserved field.
    ULONG32 Reserved5; // Reserved field.
} CMD_FLUSH;

// GET FEATURES Command
struct _CMD_GET_FEATURES {
    UCHAR Opcode; // 11h
    UCHAR Reserved0; // Reserved field.
    USHORT Timeout; // The relative time from the command issue until the command should be completed by device.
    USHORT Feature; // What feature to retrieve.
    UCHAR Reserved1; // Reserved field.
    USHORT Reserved2; // Reserved field.
    ULONG32 Reserved3; // Reserved field.
    USHORT TransferCount; // the transfer count in DWORDs
    UCHAR Reserved4; // Reserved field.
    USHORT Reserved5; // Reserved field.
    ULONG32 Reserved6; // Reserved field.
    ULONG32 Reserved7; // Reserved field.
} CMD_GET_FEATURES;

// SET FEATURES Command
struct _CMD_SET_FEATURES {
    UCHAR Opcode; // 21h
    UCHAR Reserved0; // Reserved field.
    USHORT Timeout; // The relative time from the command issue until the command should be completed by device.
    USHORT Feature; // What feature to retrieve.
    UCHAR Reserved1; // Reserved field.
    USHORT Reserved2; // Reserved field.
    ULONG32 Reserved3; // Reserved field.
    USHORT TransferCount; // the transfer count in DWORDs
    UCHAR Reserved4; // Reserved field.
    USHORT Reserved5; // Reserved field.
    ULONG32 Reserved6; // Reserved field.
    ULONG32 Reserved7; // Reserved field.
} CMD_SET_FEATURES;

//GET STATUS Command
struct _CMD_GET_STATUS {
    UCHAR Opcode; // 13h
    UCHAR Reserved0; // Reserved field.
    USHORT Timeout; // The relative time from the command issue until the command should be completed by device.
    ULONG32 Reserved1; // Reserved field.
    ULONG32 Reserved2; // Reserved field.
    ULONG32 Reserved3; // Reserved field.
    UCHAR PageType; // the page type to return, only valid when NextStatus is set
```

Notes

```
        UCHAR NextStatus;1; // If set, the host requests the next status page that the NVM subsystem has to return.
        UCHAR Reserved4:7; // Reserved field.
        USHORT Reserved5; // Reserved field
        ULONG32 Reserved6; // Reserved field.
    } CMD_GET_STATUS;


    // IDENTIFY Command
    struct _CMD_IDENTIFY {
        UCHAR Opcode; // 12h
        UCHAR Reserved0; // Reserved field.
        USHORT Timeout; // The relative time from the command issue until the command should be completed by device.
        ULONG32 Reserved1; // Reserved field.
        ULONG32 Reserved2; // Reserved field.
        ULONG32 Reserved3; // Reserved field.
        ULONG32 Reserved4; // Reserved field.
        ULONG32 Reserved5; // Reserved field.
    } CMD_IDENTIFY;
} CMD_PARAMETERS;


typedef struct _HEALTH_RANGE {
        UCHAR VitalsRO:1; //  If set, indicates this range is read only.
        UCHAR VitalsBU:1; // If set, indicates this range should be backed up.
        UCHAR VitalsLOST:1; // If set, indicates this range's data has been lost.
        UCHAR VitalsNOUSE:1; // If set, indicates this range's data should no longer be used.
        UCHAR Reserved0:4; // Reserved field
        UCHAR Reserved1; //Reserved field
        USHORT Reserved2; //Reserved field
        ULONG32 Length; // The length of the range in NVM pages
        ULONG64 StaringSectorAddr; // Starting sector address
} HEALTH_RANGE, *PHEALTH_RANGE;



typedef struct _INFO_GET_STATUS {
    UCHAR PageType; // Which page type information is returned. For example Health Status (01h)
    UCHAR PageParameter; // Page parameter
    union {
        struct _HEALTH {
            UCHAR ErrorStatusComm:1; //  If set, indicates device is present, but communication could not be established.
            UCHAR ErrorStatusInit:1; // If set, indicates the device is present, but is unable to be initialized.
            UCHAR ErrorStatusNoUsed:1; // If set, indicates the device should no longer be used.
            UCHAR Reserved0:5; // Reserved field
            UCHAR DevStatusRO:1; //If set, the device is now read only
            UCHAR DevStatusNoBU:1; // If set, the device should not be backed up
            UCHAR Reserved1:6; // Reserved field
            UCHAR Reserved2[12]; //Reserved field
            HEALTH_RANGE HealthRange[3];  //  Health range reports
        } HEALTH;
    };
} INFO_GET_STATUS, *PINFO_GET_STATUS;


typedef struct _INFO_IDENTIFY {
    UCHAR[20] SerialNum; // Serial number of the device
    UCHAR[8] FirmwareRev; // Firmware revision
    UCHAR[40] ModelNum; // Model number of the device
    ULONG32 MftCode; // Manufacture code Bit 0-7: Module revision ID, such as 01h
                     //Bit 8-15: Day of Module manufacture, such as 25h (25th day of the month)
                     //Bit 16-23: Month of  Module manufacture, such as 07h (July)
                     //Bit 24-31: Year of  Module manufacture, such as 10h (2010)
    USHORT SectorSize; // Sector size in bytes
```

Notes

```
    USHORT NVMPageSize; // NVM page size in sectors
    USHORT OpWtSize; // Optimal write transfer size in NVM pages
    USHORT MetadataSize; // Metadata block size in bytes
    USHORT AllocUnit; // Allocation unit in NVM pages
    ULONG32 Capacity; // Total number of NVM pages in device
    USHORT Cap; // Bit wise capabilities, Bit 0-13 are reserved.
                    //Bit 14: If set, partial page metadata operations supported
                    //Bit 15: If set, asynchronous status notification supported.
    USHORT SectorAlign; // The physical offset in sectors within the NVM page where the first sector is placed.
    USHORT MaxBurstSpd; // The maximum burst speed in 10 MB/s units.
    USHORT RecMinWTs; // The recommended minimum number of writes for better performance.
    USHORT RecMinRDs; // The recommended minimum number of reads for better performance.
    USHORT DatasetMagSupport; // Bit wise configurations: Bit 0: If set, supports Dataset Management command
                                  // the Attributes field in Read and Write commands is supported.
                                  //Bit 1: If set, supports Read/Write Frequency and Write Latency context attributes.
                                  //Bit 2: If set, supports Deallocate and Write Prepare context attributes.
                                  //Bit 3: If set, supports Atomic R/W and Command Access Size context attributes.
                                  //Bit 4-15: Reserved.
    USHORT PRDITable_Pref; // The prefences bit of PRD Index entries
    USHORT WriteAtomicity; // The configuration of write atomicity.

} INFO_IDENTIFY, *PINFO_IDENTIFY;
```

4.6 Register Access

In IdtFindAdapter, a memory mapped base address for accessing the registers of IDT's adapter is determined by calling MmMapIoSpace. The address is saved in IDT_PCI_CONFIG.RegBasex and its associated physical address is saved in IDT_PCI_CONFIG.RegBasexPhys for later reference. IDT's miniport driver provides a set of routines for accessing the registers. They share same return type as BOOLEAN, which indicates the access is successful (TRUE) or not (FALSE).

```
BOOLEAN IdtRegRead8 (PDEVICE_EXTENSION pDE, ULONG32 Offset, UCHAR* Value);
BOOLEAN IdtRegRead16 (PDEVICE_EXTENSION pDE, ULONG32 Offset, USHORT* Value);
BOOLEAN IdtRegRead32 (PDEVICE_EXTENSION pDE, ULONG32 Offset, ULONG32* Value);
BOOLEAN IdtRegRead64 (PDEVICE_EXTENSION pDE, ULONGLONG Offset, ULONGLONG* Value);


BOOLEAN IdtRegWrite8 (PDEVICE_EXTENSION pDE, ULONG32 Offset, UCHAR Value);
BOOLEAN IdtRegWrite16 (PDEVICE_EXTENSION pDE, ULONG32 Offset, USHORT Value);
BOOLEAN IdtRegWrite32 (PDEVICE_EXTENSION pDE, ULONG32 Offset, ULONG32 Value);
BOOLEAN IdtRegWrite64 (PDEVICE_EXTENSION pDE, ULONGLONG Offset, ULONGLONG Value);
```

The above routines in turn call the read/write routines listed below, which are provided by Storport driver to complete the access:

```
UCHAR StorPortReadRegisterUchar(pDE, PUCHAR pOffset);
USHORT StorPortReadRegisterUshort(pDE, PUSHORT pOffset);
ULONG StorPortReadRegisterUlong(pDE, PULONG pOffset);
ULONG64 READ_REGISTER_ULONG64(PULONG64 pOffset);


VOID StorPortWriteRegisterUchar(pDE, PUCHAR pOffset, UCHAR Value);
VOID StorPortWriteRegisterUshort(pDE, PUSHORT pOffset, UCHAR Value);
VOID StorPortWriteRegisterUlong(pDE, PULONG pOffset, ULONG Value);
VOID WRITE_REGISTER_ULONG64(PULONG64 pOffset, ULONG64 Value);
```

4.7 Registry Access

IDT's miniport driver creates some keys and their corresponding values in Registry after the driver is installed via INF file. These values are referenced while the driver is initialized and running. With the values listed below, the miniport driver can allocate appropriate resources during initialization:

- *MaxPRDTEntries: Max PRDT entries in one Command Table*
- *MaxMetadataSize: Max Metadata block size in bytes*

Notes

- *MaxPages: Max NVM pages transferred with one Command Table*
- *MaxTranSize: Max transfer size in bytes with one Command Table*
- *FIFOThresholdRatio: The ratio number of outstanding I/Os referred as a triggering point for Storprt driver queuing, for example, 20 means 20% of MaxEntries of CORE_CMD_INFO.*
- *CmdTO: Command Timeout in 10 millisecond units, etc*

During the run time, the following keys are referenced to optimize the overall throughput:

- *ReadCmdAborted: The number of Read Command aborted*
- *WriteCmdAborted: The number of Write Command aborted*
- *NFEs: The number of non-fatal error occurred.*
- *FEs: The number of fatal error occurred*
- *CmdUncompletedTO: The number of command not completed due to timeout., etc.*

To access the above keys and values in Registry, Storport driver provides two functions for this purpose: StorPortRegistryWrite and StorPortRegistryRead. The miniport driver needs to allocate the big enough buffer to accommodate the read/write data.

4.8 Common Data Structures

This section includes all the common data structure defined in IDT's miniport driver. These data structures are not yet disclosed in previous sections.

4.8.1 SRB_EXTENSION

```
typedef struct _SRB_EXTENSION {
    ULONG16 CoreNumber; // Processor number, which is zero-based
    ULONG16 MSIXVector; // MSI-X vector id, which is zero-based
    ULONG64 StartSector; // Starting sector number for the transfer
    ULONG32 TransferSize; // Transfer size in bytes
    ULONG32 SrbExtFlags;;  // Retry, pending or other flags
    LIST_ENTRY CmdHdrEntry;  // Pointer to the command header entry used for the request
    PRD_ENTRY PtrPRDT[pDE->MaxPRDTEntries]; // Prepared PRDT entries, max SG elements 32 by default
    ULONG32 PtrPRRI[pDE->MaxPages]; // Prepared PRD Index entries, max transfer length 4 MB by default
} SRB_EXTENSION, *PSRB_EXTENSION;
```

4.8.2 IDT_DEVICE_INFO

```
typedef struct _IDT_DEVICE_INFO {
    USHORT VendorID; //IDT vendor ID 111dh.
    USHORT DeviceID; //Device ID associated with the device.
    ULONG32 BDF; //The bus, device, function numbers associated with the device
    UCHAR DeviceCount;  // The number of devices managed by the controller
    UCHAR DeviceName[16];  // device name string
    INFO_IDENTIFY InfoIdentify; // device information retrieved via Identify command
} IDT_DEVICE_INFO, *PIDT_DEVICE_INFO;
```

4.8.3 _CORE_CMD_INFO per CPU core

```
typedef struct _CORE_CMD_INFO {
    ULONG32 MaxEntries; // Allocated number of Command Header entries, no greater than FIFOEntries
    ULONG32 FIFOEntries; // A hardware-supported max number of Command Header entries
    ULONG32 FIFOThreshold; // The threshold value used to trigger Storport driver queuing,
                           // FIFOThreshold = (MaxEntries * FIFOThresholdRatio) / 100
    STOR_PHYSICAL_ADDRESS CmdHdrStart; // Starting physical address of Command List
    STOR_PHYSICAL_ADDRESS CmdStsStart; // Starting physical address of Command Status Event Buffer
    PSTOR_DPC Dpc[32]; // DPC objects associated with each Command Header
    ULONG32 Requests; // Current accumulated I/O requests
    ULONG32 Completions; // Current accumulated, completed I/O requests
    ULONG16 OutStandingIO; // Number of current outstanding I/O requests
```

Notes

```
    ULONG16 CoreNumber; // Processor number, which is zero-based
    ULONG16 MSIXVector; // MSI-X vector id, which is zero-based
    ULONG16 CurStatusPtr; // Current pointer to the Command Status Event buffer
    ULONG32 FreeQEntries; // Number of currently available entries in Free Q
    LIST_ENTRY FreeQ; // Command Header Free Q
} CORE_CMD_INFO, *PCORE_CMD_INFO;
```

4.8.4 IDT_PCI_CONFIG

```
typedef struct _GEN_HOST_CTRL {
    ULONG32 CAP;  // Controller Capabilities
    ULONG32 Reserved; // Reserved registers
    ULONG32 IS; // Interrupt Status, bit significant.
    ULONG32 PI; // Port Implementation, bit significant.
    ULONG32 VS; // eNVMHCI Version numbers
} GEN_HOST_CTRL, *PGEN_HOST_CTRL;

typedef struct _PORT_CTRL {
    ULONG32 PxCLB;  // Port x Command List Base Address, 00h
    ULONG32 PxCLBU;  // Port x Command List Base Address Upper 32-bits, 04h
    ULONG32 PxIS;  // Port x Interrupt Status, 10h
    ULONG32 PxIE;  // Port x Interrupt Enable, 14h
    ULONG32 PxCMD;  // Port x Command and Status, 18h
    ULONG32 PxSIG;  // Port x Signature, 24h
    ULONG32 PxCI; // Port x Command Issue, 38h.
    ULONG32 PxDoorbell_LBA;  // Port x lower 32-bit doorbell register, 3ch
    ULONG32 PxDoorbell_UBA; // Port x upper 32-bit dorrbell register, 40h.
    // more register offsets here...such as MSI-X, etc
} PORT_CTRL, *PPORT_CTRL;

typedef struct _IDT_PCI_CONFIG {
    PCI_COMMON_CONFIG PciCommonConfig;  // Common header of PCI Config Space
    GEN_HOST_CTRL GenHostCtrl; // Generic Host Control registers
    PORT_CTRL PortCtrl[MAX_PORT]; // Port specific control registers.
    PVOID RegBase0; // The base address mapped for accessing Controll/Port related registers.
    ULONG RegBase0Phys; // The physical address for accessing Controll/Port related registers.
    PVOID RegBase5; // The base address mapped for accessing  PCI Config Space registers.
    ULONG RegBase5Phys; // The physical address for accessing PCI Config Space registers.
} IDT_PCI_CONFIG, *PIDT_PCI_CONFIG;


typedef struct _IDT_MSIX {
    BOOLEAN supported; // MSI-X supportted ?
    USHORT MaxVector;  // Maximum vector number supported
    USHORT MinVector;  // Minimum vector number supported
    USHORT MSIXData;  // data signature of MSI-X message
    PHYSICALADDRESS MSIXAddr // physical address where the messge will send to.
    ULONG32 TableOffset;  // MSI-X Table offset to the selected BAR physical address
    UCHAR BARSelect4Table;  // Indicates which BAR is select to base for MSI-X Table
    ULONG32 PBAOffset;  // PBA Table offset to the selected BAR physical address
    UCHAR BARSelect4PBA;  // Indicates which BAR is select to base for PBA Table
    ULONGLONG MSIXTable;  // Starting virtual address of MSI-X Table
    ULONGLONG PBATable;  // Starting virtual address of MSI-X PBA Table
} IDT_MSIX, *PIDT_MSIX;
```

4.8.5 Device_Extension

```
// Device States
#define IDT_DEV_STATE_NORMAL     0x0
#define IDT_DEV_STATE_RESET      0x1
#define IDT_DEV_STATE_ERROR      0x2
```

Notes

```
#define IDT_DEV_STATE_READONLY  0x4
#define IDT_DEV_STATE_STOP      0x8


// Device Status bits
#define IDT_DEV_STS_STORPORTQ   0x1
#define IDT_DEV_STS_NUMA        0x2
#define IDT_DEV_STS_MSIX        0x4
#define IDT_DEV_STS_INTX        0x8
#define IDT_DEV_STS_METADATA    0x10
#define IDT_DEV_STS_PACKCMD     0x20


#define IDT_DEV_FLAG_RD_ERROR       0x1
#define IDT_DEV_FLAG_WT_ERROR       0x2
#define IDT_DEV_FLAG_INIT_FAILURE   0x4
#define IDT_DEV_FLAG_RESET_FAILURE  0x8
#define IDT_DEV_FLAG_QUEUE_ERROR    0x10
#define IDT_DEV_FLAG_OUT_OF_QUEUE   0x20
#define IDT_DEV_FLAG_CMD_TIMEOUT    0x40
#define IDT_DEV_FLAG_CMD_ERR_HS     0x80
#define IDT_DEV_FLAG_CMD_ERR_ABRT   0x100
#define IDT_DEV_FLAG_CMD_ERR_NFE    0x200
#define IDT_DEV_FLAG_CMD_ERR_FE     0x400
#define IDT_DEV_FLAG_CMD_ERR_FLUSH  0x800
#define IDT_DEV_FLAG_CMD_ERR_GETF   0x1000
#define IDT_DEV_FLAG_CMD_ERR_SETF   0x2000
#define IDT_DEV_FLAG_CMD_ERR_IDEN   0x4000
#define IDT_DEV_FLAG_CMD_ERR_GETS   0x8000

typedef struct _DEVICE_EXTENSION {
    IDT_PCI_CONFIG IdtPciConfig;  // PCI config space information
    IDT_DEVICE_INFO IdtDevInfo; // device information retrieved via Identify command
    ULONG32 DeviceFlags; // Device flags.
    ULONG32 DeviceState; // Device current state, 0 means in normal state.
    ULONG32 DeviceStatus; // Device status bits.
    ULONG32 DeviceCap; // Device Capability flags.
    USHORT NUMANodes; // The number of NUMA nodes in current system.
    GROUP_AFFINITY GroupAffinity[MAX_CPU_GROUP]; // The CPU topologies
    USHORT CPUCores; // The number of processor cores in current system.
    IDT_MSIX IdtMsix; // The IDT MSI-X structure
    CORE_CMD_INFO CoreCmdInfo[32]; // Command hdr/tbl info per CPU core, up to 32 supported
    DWORD_PTR pCmdStsBuf; //The virtual address of the starting Command Status Event Buffer.
    DWORD_PTR pCmdHdr; // The virtual address of the starting Command Header.
    ULONG32 TotalCmdTablesAlloc; // The total number of Command Tables allocated
    ULONG32 TotalCmdTablesHW; // The total number of Command Tables the Controller supports.
    ULONG32 MaxPRDTEntries; // Max PRDT entries in one Command Table, 32 by default
    ULONG32 MaxMetadataSize; // Max Metadata block size in bytes
    ULONG32 MaxPages; // Max NVM pages transferred with one Command Table, ie PRD Index entries
    ULONG32 MaxTranSize; // Max transfer size in bytes with one Command Table, 4MB by default
    ULONG32 CmdTO; // Command Timeout in 10-millisecond units, minimum 10 units
    ULONG32 FIFOThresholdRatio; // The threshold ratio from Registry
    ULONG32 PaddingSize; // The padding size in bytes when allocating Command Tables
    ULONG32 PortsEnabled; // The number of Ports supported
    ULONG32 Ports; // The bit-wise port implemented from PI register
    ULONG32 SrbExtSize; // The size of SrbExtension, including PRDT and PRDI entries
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

Notes

4.8.6 Proprietary Error Logging Codes

The error codes defined in Table 6 are used with SP_xxx error codes defined by Storport driver in system event logging. These proprietary codes are meant to be more specific and easier to understand what goes wrong when reading the log.

Categories	Error Codes	Descriptions
Miniport Driver	IDT_DRV_ERROR_INIT_CMD	When driver is initialized, memory allocated for command list fails. Log with SP_UNEXPECTED_DISCONNECT.
Miniport Driver	IDT_DRV_ERROR_INIT_AR	When the access range is not found for register base. Log it with SP_UNEXPECTED_DISCONNECT
Miniport Driver	IDT_DRV_ERROR_INIT_REG	When the register access is no available. Log with With SP_UNEXPECTED_DISCONNECT.
Miniport Driver	IDT_DRV_ERROR_IO_BUF	When building the Command Table for transferring data from device to host, the provided buffer length, DataTransferlength, is too small to accommodate the read back data. Log with SP_REQUEST_TIMEOUT.
Miniport Driver	IDT_DRV_ERROR_IO_TBL	When receiving a request that requires a Command Table, we don't have any available. Log with SP_INTERNAL_ADAPTER_ERROR.
Miniport Driver	IDT_DRV_ERROR_BOGUS_INT	When driver gets interrupts, but none of the IS bits get set to indicate the interrupt is ours. Log with SP_PROTOCOL_ERROR.
Miniport Driver	IDT_DRV_ERROR_BAD_PARAM	When receiving invalid parameters in callback functions. Log with SP_PROTOCOL_ERROR.
eNVMHCI Commands	IDT_DRV_ERROR_DEV_DOWN	Whenprocessing I/O, device is not in normal state. Log with SP_INTERNAL_ADAPTER_ERROR
eNVMHCI Commands	IDT_CMD_ERROR_ABORT	When the ABRT bit is set in Overall Command Status to indicate the command is aborted due to a fatal error condition. Log with SP_INTERNAL_ADAPTER_ERROR
eNVMHCI Commands	IDT_CMD_ERROR_NME	When the NME bit is set in Command Specific Status after a command is completed. It indicates the failure of the command is due to media error. Log with SP_INTERNAL_ADAPTER_ERROR
eNVMHCI Commands	IDT_CMD_ERROR_RETR	When the RETR bit is set in Command Specific Status after a command is completed. It indicates the NVM subsystem requests the host software to retry the request. Log with SP_INTERNAL_ADAPTER_ERROR
eNVMHCI Commands	IDT_CMD_ERROR_CONF	When the CONF bit is set in Command Specific Status after a Dataset Management command is completed. It indicates conflicting attribute settings were provided. Log with SP_INTERNAL_ADAPTER_ERROR
eNVMHCI Commands	IDT_CMD_ERROR_UNAL	When the UNAL bit is set in Command Specific Status after a command is completed. It indicates a provided range did not start on an NVM page aligned boundary. Log with SP_INTERNAL_ADAPTER_ERROR
eNVMHCI Commands	IDT_CMD_ERROR_ADDR	When the ADDR bit is set in Command Specific Status after a Get/Set Features command is completed. It indicates the specified feature address is not supported in the NVM subsystem. Log with SP_INTERNAL_ADAPTER_ERROR
eNVMHCI Commands	IDT_CMD_ERROR_PARM	When the PARM bit is set in Command Specific Status after a Get/Set Features command is completed. It indicates the specified parameters are malformed. Log with SP_INTERNAL_ADAPTER_ERROR

Table 9 IDT Error Codes for Logging

Notes

Categories	Error Codes	Descriptions
eNVMHCI Commands	IDT_CMD_ERROR_UNC	When the UNC bit is set in Command Specific Status after a command is completed. It indicates the page could not be returned due to media error, bus transfer error or other error conditions. Log with SP_INTERNAL_ADAPTER_ERROR
eNVMHCI Commands	IDT_CMD_ERROR_PT	When the PT bit is set in Command Specific Status after a Get Status command is completed. It indicates the page is invalid or not supported. Log with SP_INTERNAL_ADAPTER_ERROR
eNVMHCI Commands	IDT_CMD_ERROR_NXT	When the NXT bit is set in Command Specific Status after a Get Status command is completed. It indicates the host requests the next status page and there is no status page to return. Log with SP_INTERNAL_ADAPTER_ERROR
eNVMHCI Commands	IDT_CMD_ERROR_HEALTH_X	When any bit is set in Critical Error Status after a Get Status command for Health Status Page. It indicates when X=0, the NVM device is present, but communication could not be established. wHen X=1, the NVM device is present, but is unable to be initialized. when X=2, the NVM device is no longer being used. Log with SP_INTERNAL_ADAPTER_ERROR
eNVMHCI Commands	IDT_CMD_ERROR_OOR	When OOR bit is set in Command Specific Status after a Read command, it indicates the sectors requested are not in the user accessible address range provided in the Identify command. Log with SP_INTERNAL_ADAPTER_ERROR

Table 9 IDT Error Codes for Logging

5. Example

This section provides an example that demonstrates how driver processes a request writing data into the flash memory associated with IDT's controller. When the request arrives in IDT's miniport driver in the form of SRB, the IdtBuildIo is the entry point to do some pre-processing before the real data transfer can happen between host memory and the NVM subsystem.

After IdtBuildIo confirms the pSrb->Function = SRB_FUNCTION_EXECUTE_SCSI and pSrb->SrbFlags has SRB_FLAGS_DATA_OUT bit set, which indicates the followings:

- *The data needs to be transferred from host system to the target LU.*
- *One Command Header/Table is required for the transfer.*
- *The pSrb->Cdb[0] needs to be SCSIOP_WRITE, pSrb->Cdb[2 - 9] indicates the starting sector to write on and pSrb->Cdb[10 - 13] contains the value for total transfer length in bytes.*
- *The scatter/gather list associate with the request requires pre-processing and can be retrieved with StorPortGetScatterGatherList.*

5.1 Assumptions

- *The critical values encrypted in Cdb are in standard CDB16 format and listed in Table 7 below:*

Starting Sector	Values	Transfer length	Values
pSrb->Cdb[2]	0x00	pSrb->Cdb[10]	0x00
pSrb->Cdb[3]	0x00	pSrb->Cdb[11]	0x13
pSrb->Cdb[4]	0x00	pSrb->Cdb[12]	0x06

Table 10 Cdb Values

Notes

Starting Sector	Values	Transfer length	Values
pSrb->Cdb[5]	0x00	pSrb->Cdb[13]	0x00
pSrb->Cdb[6]	0x03		
pSrb->Cdb[7]	0x34		
pSrb->Cdb[8]	0x45		
pSrb->Cdb[9]	0x56		

Table 10 Cdb Values

- *The associated scatter/gather lists below and pStorSGList->NumberOfElements = 3.*

Scatter/Gather Lists	Values
pStorSGList->List[0].PhysicalAddress.LowPart	0xCDCD0000
pStorSGList->List[0].PhysicalAddressHighPart	0x00
pStorSGList->List[0].Length	0x120000 (2304 Sectors)
pStorSGList->List[1].PhysicalAddress.LowPart	0x56EF2000
pStorSGList->List[1].PhysicalAddress.HighPart	0x20
pStorSGList->List[1].Length	0x10200 (129 Sectors)
pStorSGList->List[2].PhysicalAddress.LowPart	0x8888EE00
pStorSGList->List[2].PhysicalAddress.HighPatr	0x00
pStorSGList->List[2].\Length	0x400 (2 Sectors)

Table 11 Scatter/Gather List Information

- *The NVM page size is 4096 bytes and the sector size is 512 bytes.*
- *No Metadata transfer required.*
- *The maximum transfer size is 0x2000000 bytes (8192 NVM pages = 32 MB) for each command table.*

5.2 PRD Entries

Based the pre-defined maximum transfer size, all three elements in the scatter/gather list (total transfer size is 0x130600 = 1246720 bytes) can be transfer with a single command table.

5.2.1 PRDT Entries

Because the maximum byte count of each PRDT entry is 256 Kilobytes, there are 5 PRDT entries used for the first element of the scatter/gather list. The PRDT entries are constructed as in Table 9.

S/G Lists	PRDT Entry	DBAU	DBAL	Byte Cnt
List 0	0	0x00000000	0xCDCD0000	0x3FFFF
List 0	1	0x00000000	0xCDD10000	0x3FFFF
List 0	2	0x00000000	0xCDD50000	0x3FFFF
List 0	3	0x00000000	0xCDD90000	0x3FFFF
List 0	4	0x00000000	0xCDDD0000	0x1FFFF
List 1	5	0x00000020	0x56EF2000	0x101FF
List 2	6	0x00000000	0x8888EE00	0x3FF

Table 12 Constructed PRDT Entries

Notes

5.2.2 PRD Index Entries

According to the Cdb[2 - 9], the starting sector is 0x3344556, which means the starting sector offset is 6 of NVM Page n (zero-based). The PRD Index entries is constructed as the tables below. There are totally 257 entries constructed.

Lists	NVM Pages	DW Offset	PRDT Entry
List 0	n	0x00000000	0
List 0	n+1	0x00000100	0
List 0	n+2	0x00000500	0
...	...	...	...
List 0	n+64	0x0000FD00	0
List 0	n+65	0x00000100	1
List 0	n+66	0x00000500	1
...	...	...	...
List 0	n+128	0x0000FD00	1
List 0	n+129	0x00000100	2
List 0	n+130	0x00000500	2
...	...	...	...
List 0	n+192	0x0000FD00	2
List 0	n+193	0x00000100	3
List 0	n+194	0x00000500	3
...	...	...	...
List 0	n+256	0x0000FD00	3
List 0	n+257	0x00000100	4
List 0	n+258	0x00000500	4
...	...	...	...
List 0	n+288	0x00007D00	4
List 1	n+289	0x00000100	5
List 1	n+290	0x00000500	5
...	...	...	...
List 1	n+304	0x00003C00	5
List 2	n+305	0x00000080	6

Table 13 Constructed PRD Index Entries

5.3 Command Parameters

The following fields need to be programmed in the Command Parameter range:

Notes

Fields	Values
Opcode	0x20
T	0. If set as 1, meaning a retry.
A	1
W	1
Timeout	10 (100 ms)
Starting Sector Low	0x3344556
Starting Sector High	0
Number of Sectors	0x983
PR	1
UR	0
UW	0
Attributes	0

Table 14 Command Parameters

PTO/SB/16 (11-08)
Approved for use through 09/30/2010 OMB 0651-0032
U.S. Patent and Trademark Office: U.S. DEPARTMENT OF COMMERCE
Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number

Provisional Application for Patent Cover Sheet

This is a request for filing a PROVISIONAL APPLICATION FOR PATENT under 37 CFR 1.53(c)

Inventor(s)

Inventor 1 — Remove

Given Name	Middle Name	Family Name	City	State	Country
Peter	Z.	Onufryk	Flanders	NJ	US

Inventor 2 — Remove

Given Name	Middle Name	Family Name	City	State	Country
Cesar	A.	Talledo	San Jose	CA	US

Inventor 3 — Remove

Given Name	Middle Name	Family Name	City	State	Country
Kwok		Kong	Cupertino	CA	US

Inventor 4 — Remove

Given Name	Middle Name	Family Name	City	State	Country
Alex		Chang	Cupertino	CA	US

Inventor 5 — Remove

Given Name	Middle Name	Family Name	City	State	Country
Craig		Hackney	Sunnyvale	CA	US

Inventor 6 — Remove

Given Name	Middle Name	Family Name	City	State	Country
Jack		Wynne	Los Altos	CA	US

Inventor 7 — Remove

Given Name	Middle Name	Family Name	City	State	Country
Ganesh		Seshan	Fremont	CA	US

Inventor 8 — Remove

Given Name	Middle Name	Family Name	City	State	Country
Ihab		Jaser	San Jose	CA	US

PTO/SB/16 (11-08)
Approved for use through 09/30/2010 OMB 0651-0032
U.S. Patent and Trademark Office: U.S. DEPARTMENT OF COMMERCE
Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number

Inventor 9					Remove
Given Name	Middle Name	Family Name	City	State	Country ¡
David		Dooley	San Jose	CA	US
Inventor 10					Remove
Given Name	Middle Name	Family Name	City	State	Country ¡
Lambert		Fong	San Ramon	CA	US
Inventor 11					Remove
Given Name	Middle Name	Family Name	City	State	Country ¡
Inna		Levit	Sunnyvale	CA	US
Inventor 12					Remove
Given Name	Middle Name	Family Name	City	State	Country ¡
Toan		Tran	Sunnyvale	CA	US
Inventor 13					Remove
Given Name	Middle Name	Family Name	City	State	Country ¡
Christopher	I. W.	Norrie	San Jose	CA	US
Inventor 14					Remove
Given Name	Middle Name	Family Name	City	State	Country ¡
Jayesh		Patel	San Jose	CA	US
Inventor 15					Remove
Given Name	Middle Name	Family Name	City	State	Country ¡
Alessia		Marelli	Milan		IT

All Inventors Must Be Listed – Additional Inventor Information blocks may be generated within this form by selecting the **Add** button. Add

Title of Invention	Non Volatile Memory Host Controller Interface Device
Attorney Docket Number (if applicable)	IDT-2730-PR

Correspondence Address

PTO/SB/16 (11-08)
Approved for use through 09/30/2010 OMB 0651-0032
U.S. Patent and Trademark Office: U.S. DEPARTMENT OF COMMERCE
Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number

Direct all correspondence to (select one):	
◉ The address corresponding to Customer Number	○ Firm or Individual Name
Customer Number	33087

The invention was made by an agency of the United States Government or under a contract with an agency of the United States Government.
◉ No.
○ Yes, the name of the U.S. Government agency and the Government contract number are:

Doc Code: **TR.PROV**
Document Description: Provisional Cover Sheet (SB16)

PTO/SB/16 (11-08)
Approved for use through 09/30/2010 OMB 0651-0032
U.S. Patent and Trademark Office: U.S. DEPARTMENT OF COMMERCE
Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number

Entity Status

Applicant claims small entity status under 37 CFR 1.27

○ Yes, applicant qualifies for small entity status under 37 CFR 1.27

◉ No

Warning

Petitioner/applicant is cautioned to avoid submitting personal information in documents filed in a patent application that may contribute to identity theft. Personal information such as social security numbers, bank account numbers, or credit card numbers (other than a check or credit card authorization form PTO-2038 submitted for payment purposes) is never required by the USPTO to support a petition or an application. If this type of personal information is included in documents submitted to the USPTO, petitioners/applicants should consider redacting such personal information from the documents before submitting them to USPTO. Petitioner/applicant is advised that the record of a patent application is available to the public after publication of the application (unless a non-publication request in compliance with 37 CFR 1.213(a) is made in the application) or issuance of a patent. Furthermore, the record from an abandoned application may also be available to the public if the application is referenced in a published application or an issued patent (see 37 CFR1.14). Checks and credit card authorization forms PTO-2038 submitted for payment purposes are not retained in the application file and therefore are not publicly available.

Signature

Please see 37 CFR 1.4(d) for the form of the signature.

Signature	/Kenneth Glass/			Date (YYYY-MM-DD)	2010-08-16
First Name	Kenneth	Last Name	Glass	Registration Number (If appropriate)	42587

This collection of information is required by 37 CFR 1.51. The information is required to obtain or retain a benefit by the public which is to file (and by the USPTO to process) an application. Confidentiality is governed by 35 U.S.C. 122 and 37 CFR 1.11 and 1.14. This collection is estimated to take 8 hours to complete, including gathering, preparing, and submitting the completed application form to the USPTO. Time will vary depending upon the individual case. Any comments on the amount of time you require to complete this form and/or suggestions for reducing this burden, should be sent to the Chief Information Officer, U.S. Patent and Trademark Office, U.S. Department of Commerce, P.O. Box 1450, Alexandria, VA 22313-1450. DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. **This form can only be used when in conjunction with EFS-Web. If this form is mailed to the USPTO, it may cause delays in handling the provisional application.**

Privacy Act Statement

The Privacy Act of 1974 (P.L. 93-579) requires that you be given certain information in connection with your submission of the attached form related to a patent application or paten. Accordingly, pursuant to the requirements of the Act, please be advised that : (1) the general authority for the collection of this information is 35 U.S.C. 2(b)(2); (2) furnishing of the information solicited is voluntary; and (3) the principal purpose for which the information is used by the U.S. Patent and Trademark Office is to process and/or examine your submission related to a patent application or patent. If you do not furnish the requested information, the U.S. Patent and Trademark Office may not be able to process and/or examine your submission, which may result in termination of proceedings or abandonment of the application or expiration of the patent.

The information provided by you in this form will be subject to the following routine uses:

1. The information on this form will be treated confidentially to the extent allowed under the Freedom of Information Act (5 U.S.C. 552) and the Privacy Act (5 U.S.C 552a). Records from this system of records may be disclosed to the Department of Justice to determine whether disclosure of these records is required by the Freedom of Information Act.
2. A record from this system of records may be disclosed, as a routine use, in the course of presenting evidence to a court, magistrate, or administrative tribunal, including disclosures to opposing counsel in the course of settlement negotiations.
3. A record in this system of records may be disclosed, as a routine use, to a Member of Congress submitting a request involving an individual, to whom the record pertains, when the individual has requested assistance from the Member with respect to the subject matter of the record.
4. A record in this system of records may be disclosed, as a routine use, to a contractor of the Agency having need for the information in order to perform a contract. Recipients of information shall be required to comply with the requirements of the Privacy Act of 1974, as amended, pursuant to 5 U.S.C. 552a(m).
5. A record related to an International Application filed under the Patent Cooperation Treaty in this system of records may be disclosed, as a routine use, to the International Bureau of the World Intellectual Property Organization, pursuant to the Patent Cooperation Treaty.
6. A record in this system of records may be disclosed, as a routine use, t o a n other federal agency for purposes of National Security review (35 U.S.C. 181) and for review pursuant to the Atomic Energy Act (42 U.S.C. 218(c)).
7. A record from this system of records may be disclosed, as a routine use, to the Administrator, General Services, or his/her designee, during an inspection of records conducted by GSA as part of that agency's responsibility to recommend improvements in records management practices and programs, under authority of 44 U.S.C. 2904 and 2906. Such disclosure shall be made in accordance with the GSA regulations governing inspection of records for this purpose, and any other relevant (i.e., GSA or Commerce) directive. Such disclosure shall not be used to make determinations about individuals.
8. A record from this system of records may be disclosed, as a routine use, to the public after either publication of the application pursuant to 35 U.S.C. 122(b) or issuance of a patent pursuant to 35 U.S.C. 151. Further, a record may be disclosed, subject to the limitations of 37 CFR 1.14, as a routine use, to the public if the record was filed in an application which became abandoned or in which the proceedings were terminated and which application is referenced by either a published application, an application open to public inspection or an issued patent.
9. A record from this system of records may be disclosed, as a routine use, to a Federal, State, or local law enforcement agency, if the USPTO becomes aware of a violation or potential violation of law or regulation.

Doc Code: **TR.PROV**
Document Description: Provisional Cover Sheet (SB16)

PTO/SB/16 (11-08)
Approved for use through 09/30/2010 OMB 0651-0032
U.S. Patent and Trademark Office: U.S. DEPARTMENT OF COMMERCE
Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number

Provisional Application for Patent Cover Sheet

This is a request for filing a PROVISIONAL APPLICATION FOR PATENT under 37 CFR 1.53(c)

Inventor(s)

Inventor 1 — Remove

Given Name	Middle Name	Family Name	City	State	Country
Rino		Micheloni	Milan		IT

Inventor 2 — Remove

Given Name	Middle Name	Family Name	City	State	Country
Igor		Ziper	Sunnyvale	CA	US

All Inventors Must Be Listed – Additional Inventor Information blocks may be generated within this form by selecting the **Add** button. — Add

Title of Invention	Non Volatile Memory Host Controller Interface Device
Attorney Docket Number (if applicable)	IDT-2730-PR

Correspondence Address

Direct all correspondence to (select one):

◉ The address corresponding to Customer Number	○ Firm or Individual Name
Customer Number	33087

The invention was made by an agency of the United States Government or under a contract with an agency of the United States Government.

◉ No.

○ Yes, the name of the U.S. Government agency and the Government contract number are:

Doc Code: **TR.PROV**
Document Description: Provisional Cover Sheet (SB16)

PTO/SB/16 (11-08)
Approved for use through 09/30/2010 OMB 0651-0032
U.S. Patent and Trademark Office: U.S. DEPARTMENT OF COMMERCE
Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number

Entity Status

Applicant claims small entity status under 37 CFR 1.27

○ Yes, applicant qualifies for small entity status under 37 CFR 1.27

◉ No

Warning

Petitioner/applicant is cautioned to avoid submitting personal information in documents filed in a patent application that may contribute to identity theft. Personal information such as social security numbers, bank account numbers, or credit card numbers (other than a check or credit card authorization form PTO-2038 submitted for payment purposes) is never required by the USPTO to support a petition or an application. If this type of personal information is included in documents submitted to the USPTO, petitioners/applicants should consider redacting such personal information from the documents before submitting them to USPTO. Petitioner/applicant is advised that the record of a patent application is available to the public after publication of the application (unless a non-publication request in compliance with 37 CFR 1.213(a) is made in the application) or issuance of a patent. Furthermore, the record from an abandoned application may also be available to the public if the application is referenced in a published application or an issued patent (see 37 CFR1.14). Checks and credit card authorization forms PTO-2038 submitted for payment purposes are not retained in the application file and therefore are not publicly available.

Signature

Please see 37 CFR 1.4(d) for the form of the signature.

Signature	/Kenneth Glass/			Date (YYYY-MM-DD)	2010-08-16
First Name	Kenneth	Last Name	Glass	Registration Number (If appropriate)	42587

This collection of information is required by 37 CFR 1.51. The information is required to obtain or retain a benefit by the public which is to file (and by the USPTO to process) an application. Confidentiality is governed by 35 U.S.C. 122 and 37 CFR 1.11 and 1.14. This collection is estimated to take 8 hours to complete, including gathering, preparing, and submitting the completed application form to the USPTO. Time will vary depending upon the individual case. Any comments on the amount of time you require to complete this form and/or suggestions for reducing this burden, should be sent to the Chief Information Officer, U.S. Patent and Trademark Office, U.S. Department of Commerce, P.O. Box 1450, Alexandria, VA 22313-1450. DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. **This form can only be used when in conjunction with EFS-Web. If this form is mailed to the USPTO, it may cause delays in handling the provisional application.**

Privacy Act Statement

The Privacy Act of 1974 (P.L. 93-579) requires that you be given certain information in connection with your submission of the attached form related to a patent application or paten. Accordingly, pursuant to the requirements of the Act, please be advised that : (1) the general authority for the collection of this information is 35 U.S.C. 2(b)(2); (2) furnishing of the information solicited is voluntary; and (3) the principal purpose for which the information is used by the U.S. Patent and Trademark Office is to process and/or examine your submission related to a patent application or patent. If you do not furnish the requested information, the U.S. Patent and Trademark Office may not be able to process and/or examine your submission, which may result in termination of proceedings or abandonment of the application or expiration of the patent.

The information provided by you in this form will be subject to the following routine uses:

1. The information on this form will be treated confidentially to the extent allowed under the Freedom of Information Act (5 U.S.C. 552) and the Privacy Act (5 U.S.C 552a). Records from this system of records may be disclosed to the Department of Justice to determine whether disclosure of these records is required by the Freedom of Information Act.
2. A record from this system of records may be disclosed, as a routine use, in the course of presenting evidence to a court, magistrate, or administrative tribunal, including disclosures to opposing counsel in the course of settlement negotiations.
3. A record in this system of records may be disclosed, as a routine use, to a Member of Congress submitting a request involving an individual, to whom the record pertains, when the individual has requested assistance from the Member with respect to the subject matter of the record.
4. A record in this system of records may be disclosed, as a routine use, to a contractor of the Agency having need for the information in order to perform a contract. Recipients of information shall be required to comply with the requirements of the Privacy Act of 1974, as amended, pursuant to 5 U.S.C. 552a(m).
5. A record related to an International Application filed under the Patent Cooperation Treaty in this system of records may be disclosed, as a routine use, to the International Bureau of the World Intellectual Property Organization, pursuant to the Patent Cooperation Treaty.
6. A record in this system of records may be disclosed, as a routine use, t o a n other federal agency for purposes of National Security review (35 U.S.C. 181) and for review pursuant to the Atomic Energy Act (42 U.S.C. 218(c)).
7. A record from this system of records may be disclosed, as a routine use, to the Administrator, General Services, or his/her designee, during an inspection of records conducted by GSA as part of that agency's responsibility to recommend improvements in records management practices and programs, under authority of 44 U.S.C. 2904 and 2906. Such disclosure shall be made in accordance with the GSA regulations governing inspection of records for this purpose, and any other relevant (i.e., GSA or Commerce) directive. Such disclosure shall not be used to make determinations about individuals.
8. A record from this system of records may be disclosed, as a routine use, to the public after either publication of the application pursuant to 35 U.S.C. 122(b) or issuance of a patent pursuant to 35 U.S.C. 151. Further, a record may be disclosed, subject to the limitations of 37 CFR 1.14, as a routine use, to the public if the record was filed in an application which became abandoned or in which the proceedings were terminated and which application is referenced by either a published application, an application open to public inspection or an issued patent.
9. A record from this system of records may be disclosed, as a routine use, to a Federal, State, or local law enforcement agency, if the USPTO becomes aware of a violation or potential violation of law or regulation.

Electronic Patent Application Fee Transmittal

Application Number:	
Filing Date:	
Title of Invention:	Non Volatile Memory Host Controller Interface Device
First Named Inventor/Applicant Name:	Peter Z. Onufryk
Filer:	Kenneth Glass
Attorney Docket Number:	IDT-2730-PR

Filed as Large Entity

Provisional Filing Fees

Description	Fee Code	Quantity	Amount	Sub-Total in USD($)
Basic Filing:				
Provisional application filing	1005	1	220	220
Pages:				
Claims:				
Miscellaneous-Filing:				
Petition:				
Patent-Appeals-and-Interference:				
Post-Allowance-and-Post-Issuance:				
Extension-of-Time:				

Description	Fee Code	Quantity	Amount	Sub-Total in USD($)
Miscellaneous:				
			Total in USD ($)	**220**

Electronic Acknowledgement Receipt

EFS ID:	8230754
Application Number:	61374242
International Application Number:	
Confirmation Number:	4168
Title of Invention:	Non Volatile Memory Host Controller Interface Device
First Named Inventor/Applicant Name:	Peter Z. Onufryk
Customer Number:	33087
Filer:	Kenneth Glass
Filer Authorized By:	
Attorney Docket Number:	IDT-2730-PR
Receipt Date:	16-AUG-2010
Filing Date:	
Time Stamp:	22:15:16
Application Type:	Provisional

Payment information:

Submitted with Payment	yes
Payment Type	Credit Card
Payment was successfully received in RAM	$220
RAM confirmation Number	8472
Deposit Account	501006
Authorized User	GLASS,KENNETH
The Director of the USPTO is hereby authorized to charge indicated fees and credit any overpayment as follows: Charge any Additional Fees required under 37 C.F.R. Section 1.16 (National application filing, search, and examination fees)	

File Listing:

Document Number	Document Description	File Name	File Size(Bytes)/ Message Digest	Multi Part /.zip	Pages (if appl.)
1	Specification	Data_Network.pdf	122742 2dc30badbcde08359a0d021fea26abdc0f6fa717	no	4
Warnings:					
Information:					
2	Specification	Delaware_NVMHCI_DMA.pdf	188743 5e5889cf9631798e4fe9367b0278c6121b0a4601	no	6
Warnings:					
Information:					
3	Specification	Message_Network_Node.pdf	106839 35507b93fcc9036cf13ec53b2c74b68bfff9ce6c	no	3
Warnings:					
Information:					
4	Specification	PwrPT-Disclosurep.pdf	488162 c5c3ce2e834736b92bd81603ac2ac113519aa858	no	7
Warnings:					
Information:					
5	Specification	delaware_Firmware-Architecture-p.pdf	1040418 be4ca3f26ee2187be8fd3097c85fdb89ee00b808	no	132
Warnings:					
Information:					
6	Specification	delaware_hw_spec-p.pdf	1265214 69d42d19a39a7c3a232ea5c2eb81029aa93b81f3	no	156
Warnings:					
Information:					
7	Specification	delaware_memory_interface-p.pdf	66097 d287dfca9a5c3dcb9ec90cf26f3f4af55e2e84f1	no	8
Warnings:					
Information:					
8	Specification	delaware_Windows_Driver_Architecture_Spec-p.pdf	412972 7dd01ecd76039d09baeb0f1ce70126efb8d8f82a	no	34
Warnings:					
Information:					

9	Application Data Sheet	ProvisionalSB-p.pdf	47286 9bcb6d22053f71634dce458dbfff48c19662461d	no	5
Warnings:					
Information:					
This is not an USPTO supplied ADS fillable form					
10	Application Data Sheet	ProvisionalSB2-p.pdf	34362 7e919b327214d423ee1f0cc74fd949825daf78be	no	3
Warnings:					
Information:					
This is not an USPTO supplied ADS fillable form					
11	Fee Worksheet (PTO-875)	fee-info.pdf	29288 55dea78635eec11beee5580cc383ebad9830e3db	no	2
Warnings:					
Information:					
		Total Files Size (in bytes):	3802123		

This Acknowledgement Receipt evidences receipt on the noted date by the USPTO of the indicated documents, characterized by the applicant, and including page counts, where applicable. It serves as evidence of receipt similar to a Post Card, as described in MPEP 503.

<u>New Applications Under 35 U.S.C. 111</u>
If a new application is being filed and the application includes the necessary components for a filing date (see 37 CFR 1.53(b)-(d) and MPEP 506), a Filing Receipt (37 CFR 1.54) will be issued in due course and the date shown on this Acknowledgement Receipt will establish the filing date of the application.

<u>National Stage of an International Application under 35 U.S.C. 371</u>
If a timely submission to enter the national stage of an international application is compliant with the conditions of 35 U.S.C. 371 and other applicable requirements a Form PCT/DO/EO/903 indicating acceptance of the application as a national stage submission under 35 U.S.C. 371 will be issued in addition to the Filing Receipt, in due course.

<u>New International Application Filed with the USPTO as a Receiving Office</u>
If a new international application is being filed and the international application includes the necessary components for an international filing date (see PCT Article 11 and MPEP 1810), a Notification of the International Application Number and of the International Filing Date (Form PCT/RO/105) will be issued in due course, subject to prescriptions concerning national security, and the date shown on this Acknowledgement Receipt will establish the international filing date of the application.

Electronic Acknowledgement Receipt

EFS ID:	8230754
Application Number:	61374242
International Application Number:	
Confirmation Number:	4168
08/25/2010 UEDUVIJE 00000013 501006 61374242 01 FC:1085 1080.00 DA	
Title of Invention:	Non Volatile Memory Host Controller Interface Device
First Named Inventor/Applicant Name:	Peter Z. Onufryk
Customer Number:	33087
Filer:	Kenneth Glass
Filer Authorized By:	
Attorney Docket Number:	IDT-2730-PR
Receipt Date:	16-AUG-2010
Filing Date:	
Time Stamp:	22:15:16
Application Type:	Provisional

Payment information:

Submitted with Payment	yes
Payment Type	Credit Card
Payment was successfully received in RAM	$220
RAM confirmation Number	8472
Deposit Account	501006
Authorized User	GLASS,KENNETH
The Director of the USPTO is hereby authorized to charge indicated fees and credit any overpayment as follows: Charge any Additional Fees required under 37 C.F.R. Section 1.16 (National application filing, search, and examination fees)	

UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE
United States Patent and Trademark Office
Address: COMMISSIONER FOR PATENTS
P.O. Box 1450
Alexandria, Virginia 22313-1450
www.uspto.gov

APPLICATION NUMBER	FILING or 371(c) DATE	GRP ART UNIT	FIL FEE REC'D	ATTY.DOCKET.NO	TOT CLAIMS	IND CLAIMS
61/374,242	08/16/2010		1300	IDT-2730-PR		

CONFIRMATION NO. 4168

33087
GLASS & ASSOCIATES
P.O. BOX 1220
LOS GATOS, CA 95031-1220

FILING RECEIPT

OC000000043212498

Date Mailed: 08/27/2010

Receipt is acknowledged of this provisional patent application. It will not be examined for patentability and will become abandoned not later than twelve months after its filing date. Any correspondence concerning the application must include the following identification information: the U.S. APPLICATION NUMBER, FILING DATE, NAME OF APPLICANT, and TITLE OF INVENTION. Fees transmitted by check or draft are subject to collection. Please verify the accuracy of the data presented on this receipt. **If an error is noted on this Filing Receipt, please submit a written request for a Filing Receipt Correction. Please provide a copy of this Filing Receipt with the changes noted thereon. If you received a "Notice to File Missing Parts" for this application, please submit any corrections to this Filing Receipt with your reply to the Notice. When the USPTO processes the reply to the Notice, the USPTO will generate another Filing Receipt incorporating the requested corrections**

Applicant(s)

Peter Z. Onufryk, Flanders, NJ;
Cesar A. Talledo, San Jose, CA;
Kwok Kong, Cupertino, CA;
Alex Chang, Cupertino, CA;
Craig Hackney, Sunnyvale, CA;
Jack Wynne, Los Altos, CA;
Ganesh Seshan, Fremont, CA;
Ihab Jaser, San Jose, CA;
David Dooley, San Jose, CA;
Lambert Fong, San Ramon, CA;
Inna Levit, Sunnyvale, CA;
Toan Tran, Sunnyvale, CA;
Christopher I.W. Norrie, San Jose, CA;
Jayesh Patel, San Jose, CA;
Alessia Marelli, Milan, ITALY;
Rino Micheloni, Milan, ITALY;
Igor Ziper, Sunnyvale, CA;

Power of Attorney:
Kenneth Glass--42587

If Required, Foreign Filing License Granted: 08/25/2010

The country code and number of your priority application, to be used for filing abroad under the Paris Convention, is **US 61/374,242**

Projected Publication Date: None, application is not eligible for pre-grant publication

Non-Publication Request: No

Early Publication Request: No

Title

Non Volatile Memory Host Controller Interface Device

PROTECTING YOUR INVENTION OUTSIDE THE UNITED STATES

Since the rights granted by a U.S. patent extend only throughout the territory of the United States and have no effect in a foreign country, an inventor who wishes patent protection in another country must apply for a patent in a specific country or in regional patent offices. Applicants may wish to consider the filing of an international application under the Patent Cooperation Treaty (PCT). An international (PCT) application generally has the same effect as a regular national patent application in each PCT-member country. The PCT process **simplifies** the filing of patent applications on the same invention in member countries, but **does not result** in a grant of "an international patent" and does not eliminate the need of applicants to file additional documents and fees in countries where patent protection is desired.

Almost every country has its own patent law, and a person desiring a patent in a particular country must make an application for patent in that country in accordance with its particular laws. Since the laws of many countries differ in various respects from the patent law of the United States, applicants are advised to seek guidance from specific foreign countries to ensure that patent rights are not lost prematurely.

Applicants also are advised that in the case of inventions made in the United States, the Director of the USPTO must issue a license before applicants can apply for a patent in a foreign country. The filing of a U.S. patent application serves as a request for a foreign filing license. The application's filing receipt contains further information and guidance as to the status of applicant's license for foreign filing.

Applicants may wish to consult the USPTO booklet, "General Information Concerning Patents" (specifically, the section entitled "Treaties and Foreign Patents") for more information on timeframes and deadlines for filing foreign patent applications. The guide is available either by contacting the USPTO Contact Center at 800-786-9199, or it can be viewed on the USPTO website at http://www.uspto.gov/web/offices/pac/doc/general/index.html.

For information on preventing theft of your intellectual property (patents, trademarks and copyrights), you may wish to consult the U.S. Government website, http://www.stopfakes.gov. Part of a Department of Commerce initiative, this website includes self-help "toolkits" giving innovators guidance on how to protect intellectual property in specific countries such as China, Korea and Mexico. For questions regarding patent enforcement issues, applicants may call the U.S. Government hotline at 1-866-999-HALT (1-866-999-4158).

LICENSE FOR FOREIGN FILING UNDER

Title 35, United States Code, Section 184

Title 37, Code of Federal Regulations, 5.11 & 5.15

GRANTED

The applicant has been granted a license under 35 U.S.C. 184, if the phrase "IF REQUIRED, FOREIGN FILING LICENSE GRANTED" followed by a date appears on this form. Such licenses are issued in all applications where

the conditions for issuance of a license have been met, regardless of whether or not a license may be required as set forth in 37 CFR 5.15. The scope and limitations of this license are set forth in 37 CFR 5.15(a) unless an earlier license has been issued under 37 CFR 5.15(b). The license is subject to revocation upon written notification. The date indicated is the effective date of the license, unless an earlier license of similar scope has been granted under 37 CFR 5.13 or 5.14.

This license is to be retained by the licensee and may be used at any time on or after the effective date thereof unless it is revoked. This license is automatically transferred to any related applications(s) filed under 37 CFR 1.53(d). This license is not retroactive.

The grant of a license does not in any way lessen the responsibility of a licensee for the security of the subject matter as imposed by any Government contract or the provisions of existing laws relating to espionage and the national security or the export of technical data. Licensees should apprise themselves of current regulations especially with respect to certain countries, of other agencies, particularly the Office of Defense Trade Controls, Department of State (with respect to Arms, Munitions and Implements of War (22 CFR 121-128)); the Bureau of Industry and Security, Department of Commerce (15 CFR parts 730-774); the Office of Foreign AssetsControl, Department of Treasury (31 CFR Parts 500+) and the Department of Energy.

NOT GRANTED

No license under 35 U.S.C. 184 has been granted at this time, if the phrase "IF REQUIRED, FOREIGN FILING LICENSE GRANTED" DOES NOT appear on this form. Applicant may still petition for a license under 37 CFR 5.12, if a license is desired before the expiration of 6 months from the filing date of the application. If 6 months has lapsed from the filing date of this application and the licensee has not received any indication of a secrecy order under 35 U.S.C. 181, the licensee may foreign file the application pursuant to 37 CFR 5.15(b).