	Controlled version is located in the project repository. Printed versions are uncontrolled unless stamped “Controlled Copy” in RED.

Software Design Document for NXP iSSD

(NVMe based intelligent SSD Card)

Product requirement id	N/A
Version	A1-06
Date	30/09/1026
iCAP Classification	NXP internal

REVIEW / OFFICIAL SIGN-OFF

Role	Name	Date

Table of Contents

List of Figures

List of Tables

Version Tracking

Date (DDMMYYYY)	Version	Comments	Author
04May2015	A1-01	Initial Draft	VVDN Technologies
05May2015	A1-02	• Updated Use Cases Section with more details and added section for performance targets • Added table of figures	VVDN Technologies

23Aug2015	A1-03	• Added NVMe Register Update Handling Sequence (Figure 20) • Updated Handling Submission Queue Doorbell (Figure 24) • Added 3.2.7.3 Software Implementation Structure • Added Queue scheduler Flowchart (Figure 29) • Added Flowchart for Selecting highest priority IO SQ (Figure 30) • Added 3.2.8 Thread Architecture • Added Sequence diagram for Thread Architecture (Figure 31) • Added 3.2.9 Init Sequence with flowchart • Added 3.4General SQ and CQ Entry Structure with Table 5: General SQ entry format and Table 6: General CQ entry format • Updated 3.6.1.4 Processing the Commands flowchart • Updated Figure 47: Handling the Read / Write IO Command Flow Chart • Added Figure 48: DMA Setup • Added Figure 49: IO Completion Handling in IO processor thread	VVDN Technologies

25Aug2015	A1-04	Updated FPGA design details with control registers and DMA operations and necessary structures.	VVDN Technologies
11Nov2015	A1-05	• Updated 3.8.1 -DDR based FPGA-DMAdesign details with control registers, operations and necessary structures. • Added Figure 57 - High Level FTL flowchart. • Added 3.7.6.1 - nvme_biodetails: An Interface between NVMe and Memory Device Manager Modules (DDR/NAND).	VVDN Technologies

30Sep2016	A1-06	• Modified thread Architecture • Modified DDR DMA handling flowcharts as per latest descriptor structure • Added Unaligned pages Handling • Added NAND DMA manager flowchart • Modified NAND DMA Descriptor Structure • Modified the FPGA Interrupts from MSI to GPIO • Added the GPIO Interrupt registers • Added the NAND design • Added the DMA table and Control registers offsets of the Supported FPGA Images • Removed the data Interleaving handler diagram in DDR DMA • Removed Scatter- gather DMA	VVDN Technlogies

1 Overview

This document describes the requirements of an LS2085A based NVMe intelligent SSD Card which "VVDN" is developing for "NXP". These requirements have been derived from the requirement specifications provided by NXP and subsequent discussions and email exchanges with NXP.

This SDD is made for the reference of:

- Product Managers at VVDN to confirm the design before development
- Engineering Teams at VVDN for Architecture, Design and Development of the Hardware board.
- System Integration and Verification teams at VVDN.

1.1 Acronyms and Definitions

Table 1: Acronyms and Definitions

ACRONYMS	DEFINITIONS
NVMe	Non-Volatile Memory Express
LS	Layer Scape
SG	Scatter-Gather
DMA	Direct Memory Access
FPGA	Field Programmable Gate Array
PRP	Physical Region Page
RR	Round Robin
WRR	Weighted Round Robin
SQ	Submission Queue
CQ	Completion Queue
DB	Doorbell
OCM	On Chip Memory (FPGA - used for DMA descriptors)
LBA	Logical Block Address

1.2 Scope of the Project

The scope of this project is to design, develop and deliver the NVMe intelligent SSD Card software meeting the requirements specified in the PRD.

1.3 Product Overview

LS2085A based Intelligent SSD Card is an LS2 based NVMe compatible Intelligent SSD Card in a PCI Express form factor meeting the specifications defined by NXP. This is a 2 board solution where LS2085A is in the main board which is described as NIC Card whereas FPGA and the NVDIMMs are sitting on the Storage Card. The device will be enumerated as a PCIe Gen3 device.

The proposed modular architecture contains these (2) PCBs identified as:

- **"NIC Card"** - Contains Processor, Memory 10G interface etc.
- **"Storage Card"**- Contains FPGA and DDR3/NAND DIMM Connector

The NIC Card have 4xPCIe interface to Root Complex devices (PC) where this card is act as an End Point. Similarly for the Storage card, NIC will be the Root complex device with two 4xPCIe interface.

The LS2085A is an eight core ARM® Cortex®-A57 64-bit processor, clocked at 2 GHz frequency.

NVMe is a specification for accessing solid-state drives (SSDs) attached through the PCI Express (PCIe) bus. "NVM" stands as an acronym for non-volatile memory, which is used in SSDs.

The flash storage used in SSDs can offer greater bandwidth for read-write operations when compared to conventional hard-disks, due to the absence of any moving parts. Yet, there exists a bottleneck in the SATA interface used to connect the SSDs to the host processor; this bottleneck is overcome by replacing this SATA interface with a much faster PCIe interface.

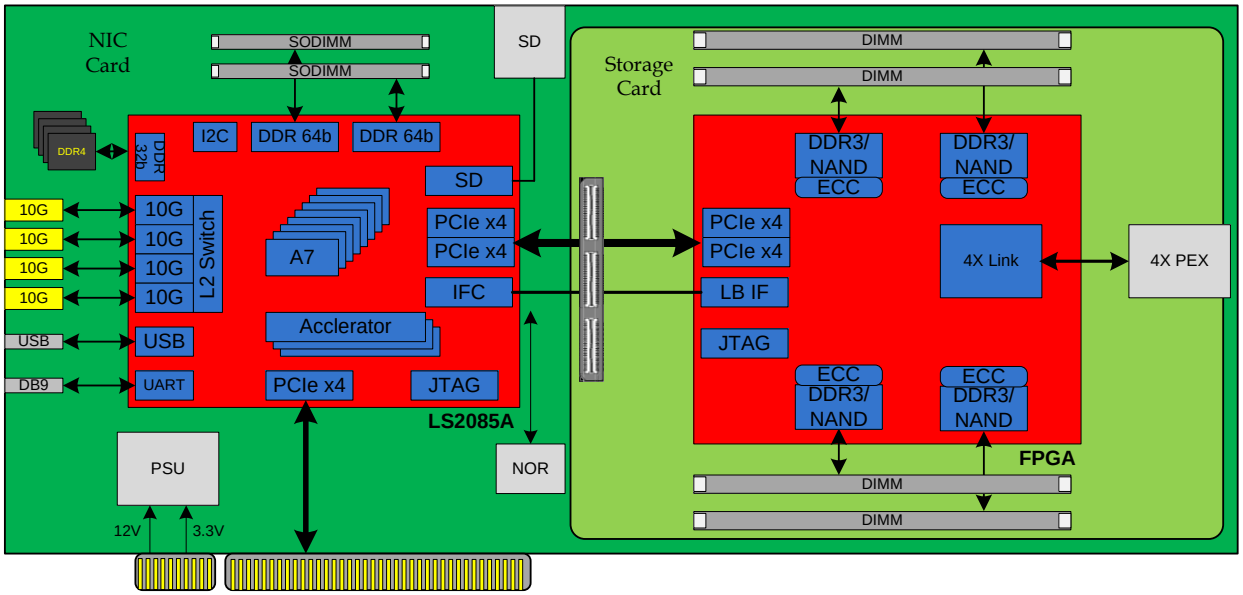


Figure 1: Block Diagram

LS2 device interfaces utilized in the design:

- 4 x 10G XFI
- 1 x USB IF
- 1 x 4-lane PCIe 3.0 end point for host interface (to communicate with host PC)
- 2 x 4-lane PCIe 3.0/2.0 Root complex interface (to interface NVMe Controller in FPGA)
- 1 x UART Interface for system console / debug
- 1 x SD card Interface
- 1 x IFC master interface to control various blocks in the FPGA
- 2 x SODIMM for DDR4 interface

FPGA interfaces required in FP utilized in the design:

- 2 x 4-lane PCIe3.0 endpoint (for LS2 interface)
- 1 x 16 bit local bus interface for back door communication
- 2 x 4-lane SERDES interface for card to card communication
- 4 x DIMM support for DDR3/NAND controller interface

Provide a brief description of the feature. Include background/historical information that will give the reader context that will help him understand the feature. Things to consider:

2 Use Cases

The NVMe iSSD card can be used in two modes,

- iNIC
- iSSD

Mode shall be selected during the card boot at the U-Boot prompt. Following U-Boot command is provided for Mode Selection

setenv bootimg <mode> {mode= inic, issd}

2.1 iNIC Mode

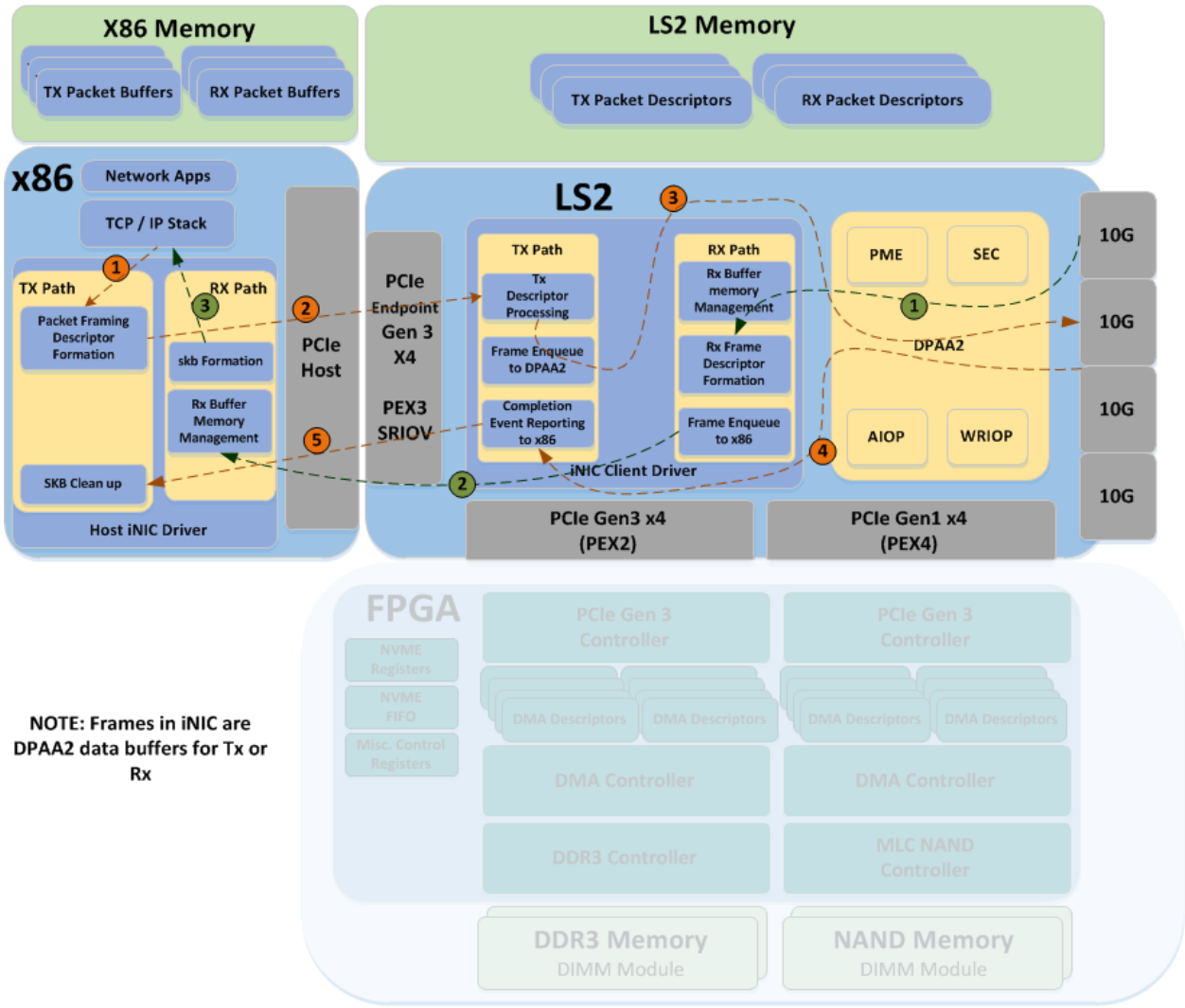


Figure 2: iNIC Datapath

Current Performance Throughput @ 4KMTU packets	Transmit path	Receive Path
iNIC driver with 1 x 10G ports	1 x 8.9Gbps	1 x 9.5Gbps
LS2 native 10G Network Driver with 1 x 10G ports	1 x 9.6Gbps	1 x 9.8Gbps

2.2 iSSD mode

2.2.1 Description

The NVMe iSSD can act as a storage card connected to a host system over a PCIe Gen3x4 Interface. It would act as a high capacity and high performance storage system using NVMe as an access method for data and control.

2.2.2 Use case setup

Below diagram shows the hardware setup and the software data path of NVMe storage card mode of this product. In issd mode the iSSD Card is plugged into the Gen3 x4 capable PCIe slot in the host machine. The host machine enumerates this card during boot time, allocates required IO and memory resources and loads the appropriate NVMe host driver that supports it.

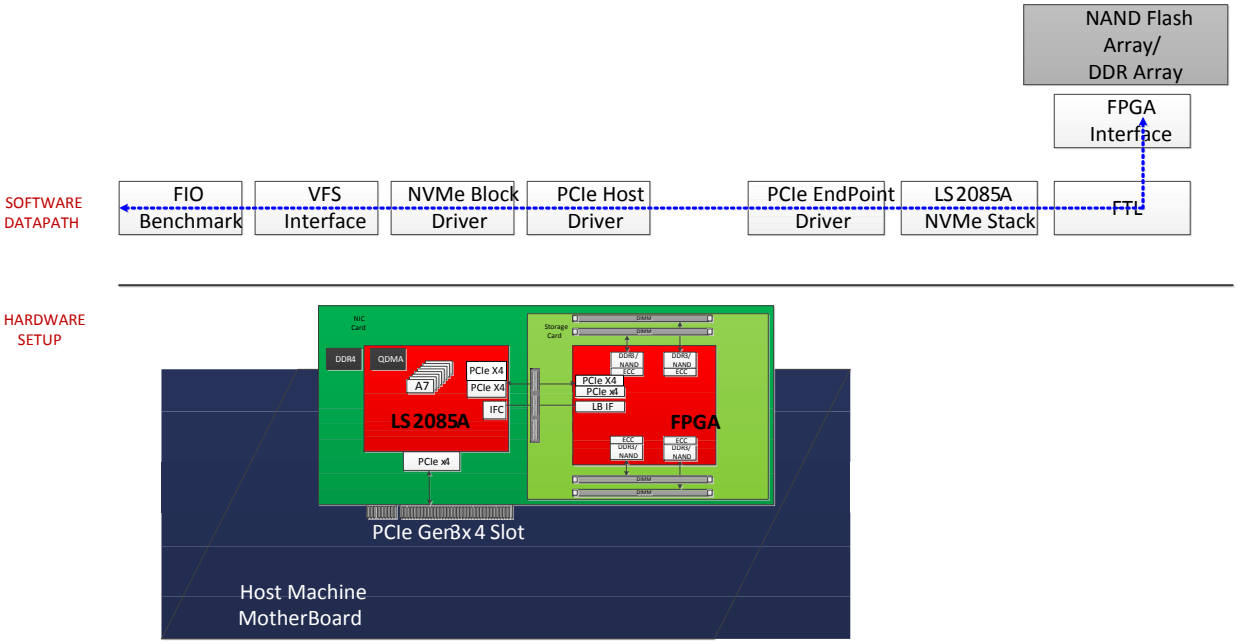


Figure 3: NVMe storage card

2.2.2.1 iSSD Mode - NVMe

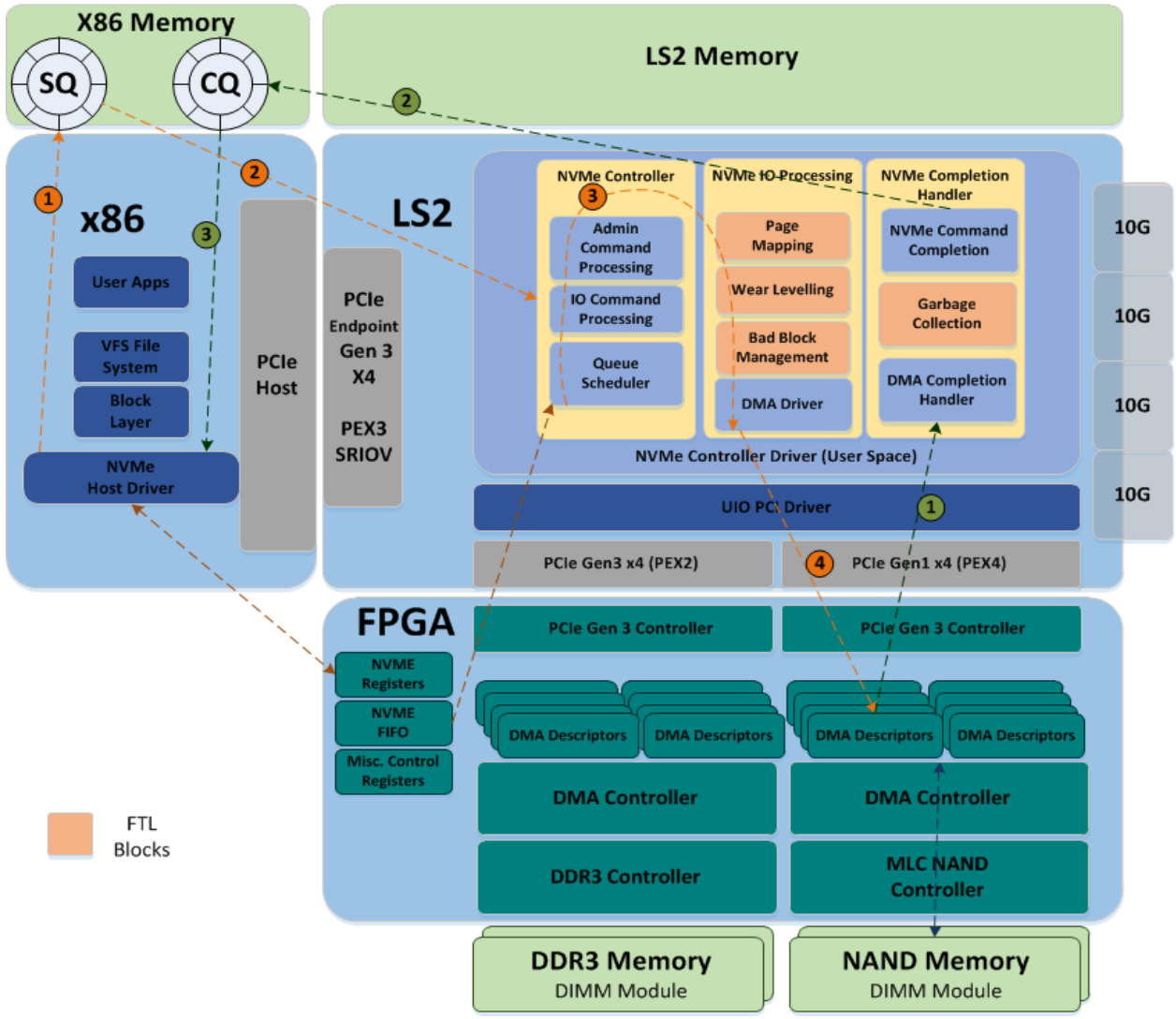


Figure 4: iSSD Datapath

2.2.2.1.1 Targeted Performance

	READ (Gbps)	WRITE (Gbps)
FPGA DDR	2.8	3.8
FPGA NAND	2.4	3.2

3 High Level Design

3.1 Software Architecture

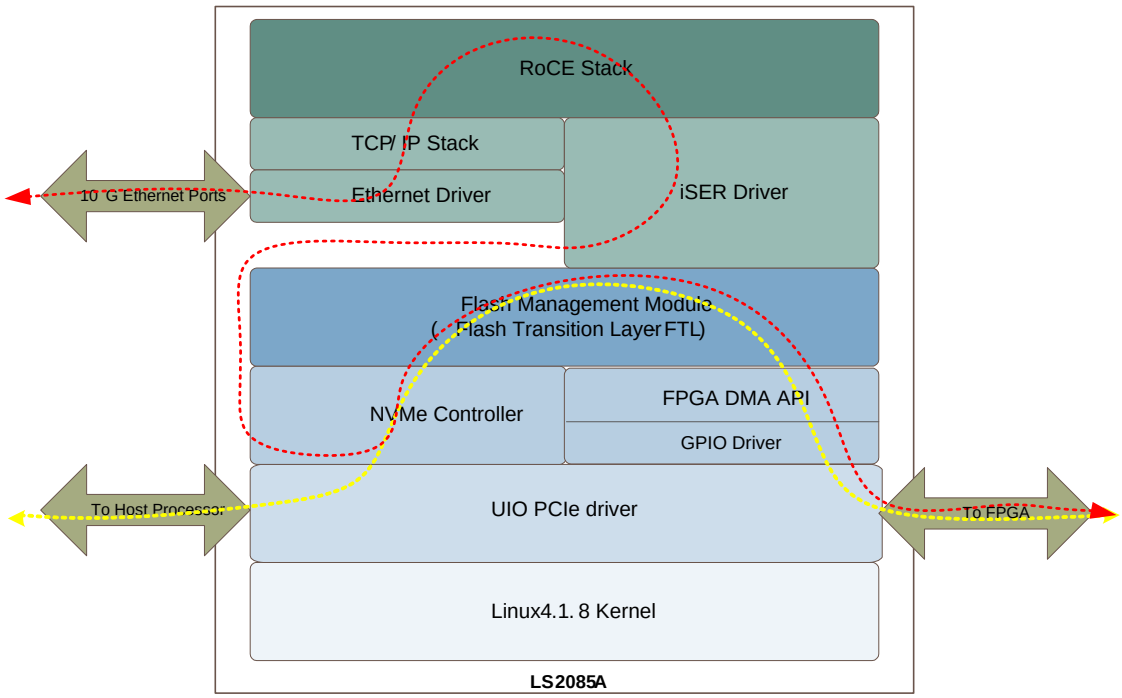


Figure 5: Software Architecture Diagram

3.1.1 Typical Data Flow

A typical IO traction between the Host PC to the memory connected to the NVMe card, involves the following steps:

- The host processor executes a block IO request to the NVMe controller via the PCIe interface to the LS2 processor
- The NVMe controller processes the request and routes it to the FTL layer
- The FTL layer translates the requested LBA (logical) addresses to the corresponding physical flash memory addresses
- The physical memory address are fed to the DMA controller, which initiates the DMA transfer between the host processor and the memory connected to the NVMe FPGA
- The NVMe FPGA DMA API interfaces the NVMe controller and the FPGA

3.1.2 Use-case Diagrams

3.1.2.1 NVMe Controller Driver

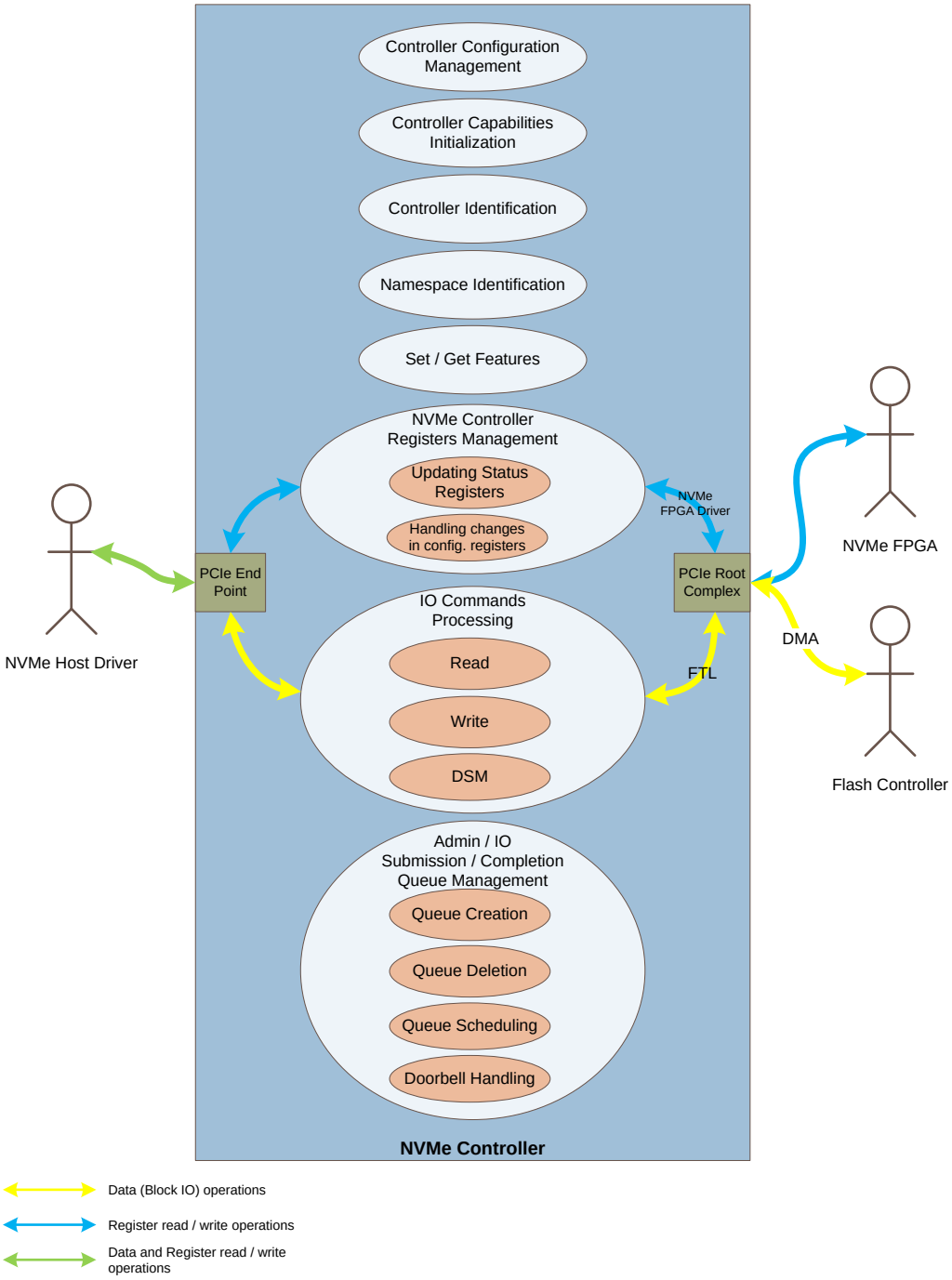


Figure 6: NVMe Controller Use-Case Diagram

The NVMe Controller driver is the core module for the LS2 based NVMe device. This module would implement the NVMe stack as per the NVMe 1.2 standards. It supports the following:

1. NVMe Admin Commands
2. NVMe IO Commands
3. NVMe Register sets
4. NVMe Admin and IO Queues

3.1.2.2 PCIe End Point Driver - uio_pci_generic Driver

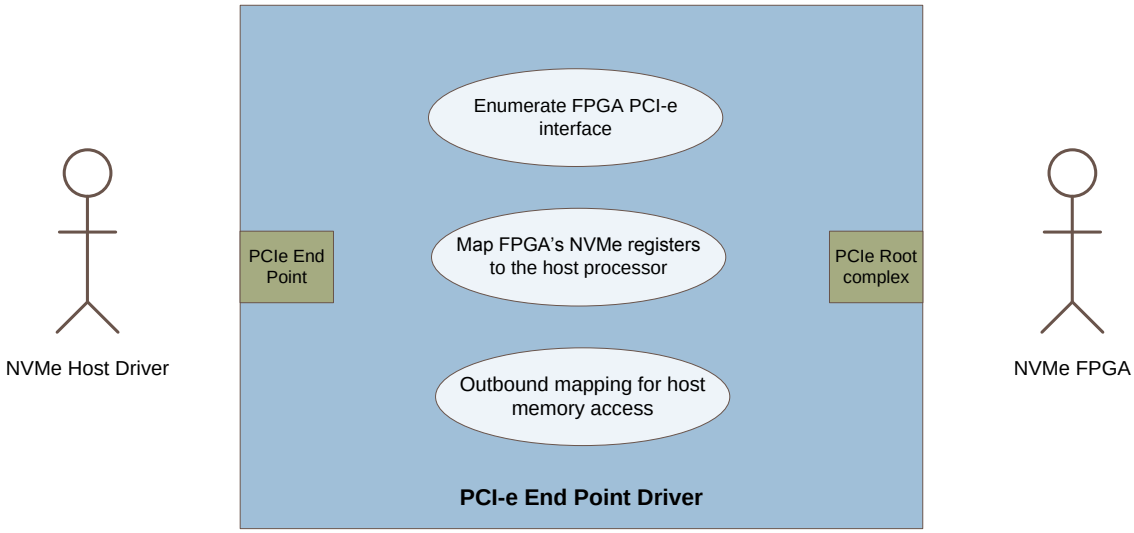


Figure 7: PCI-e End Point Driver Use-Case Diagram

The PCI-e end-point - uio_pci_generic driver configures the PCI-e x4 interface connecting to the Host processor and enable the FPGA 2 PCIe x4 interface. The End-Point driver exposes the FPGA's PEX2 BAR2 NVMe registers to the Host Processor on its own BAR0 registers through inbound iATU operation, via which the host processor could be able to access the PEX2 BAR2 registers of the NVMe FPGA.

The PEX3 PCIe (host interface) BAR2 is meant for MSI-x interrupts to the Host processor's NVMe driver. This is used to intimate the completion of a command issued by the host processor.

3.1.2.3 Flash Management Layer

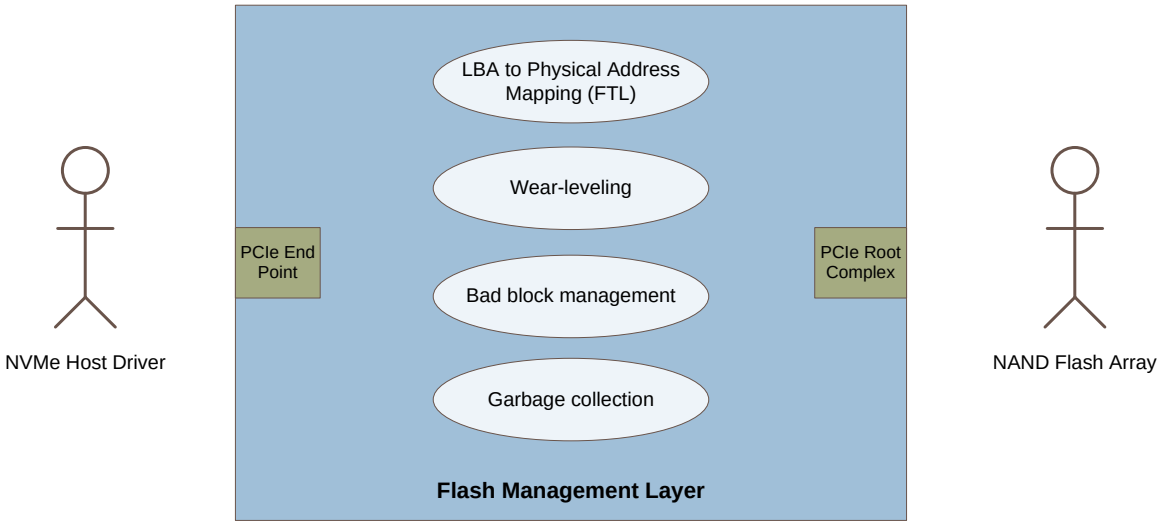


Figure 8: Flash Management Layer Use-Case Diagram

The Flash Management Layer is one of the crucial modules in this NVMe-Controller. The following are the responsibilities of the Flash Management Layer:

1. **Flash Translation Layer (FTL)**
   - Converts the LBA (Logical Block Array) address provided by the Host processor into its corresponding Physical address
2. **Wear-leveling**
   - Erasing the same block on a NAND flash can result in wear-and-tear of the block and eventually the block unusable
   - To make sure the whole flash erase-write is distributed across all the blocks of the flash, the wear-leveling algorithm is used
3. **Bad block management**
   - When a block is marked as bad, the block is to be skipped during write. The bad block management algorithm ensures this
4. **Garbage collection**
   - Due to the employment of wear-leveling algorithm, the modified sectors are not re-written on the same location, else it is written back on the flash on a different location
   - When all the sectors of a specific block is moved to a new location, the block would be erased and made ready for reuse

3.1.2.4 LS2 FPGA Communications

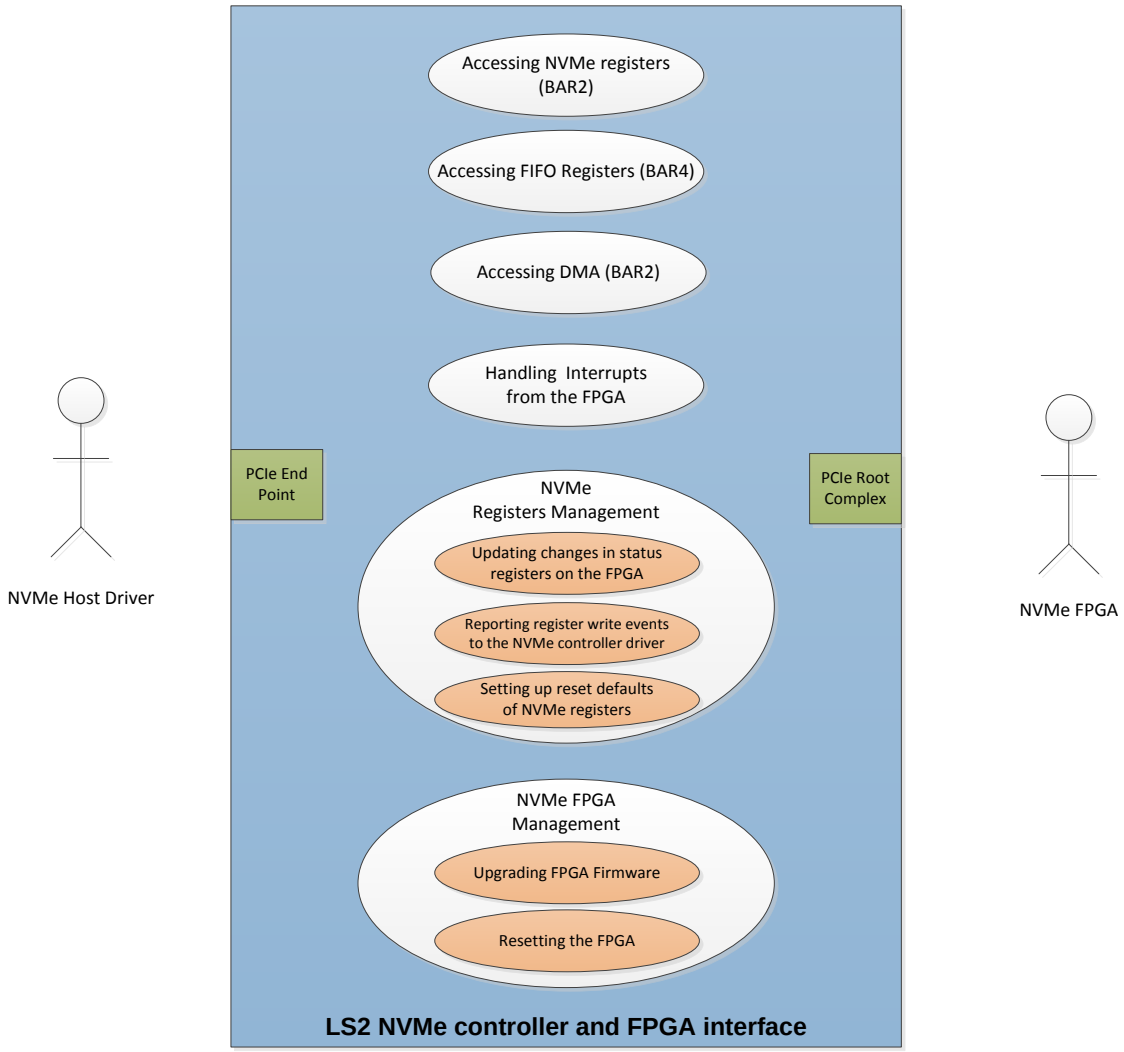


Figure 9: LS2 NVMe controller and FPGA Interactions

The NVMe FPGA is a PCI-e device, which sits on the PIC-e host interface of the LS2 processor. The NVMe registers are hosted on the FPGA in its BAR2 register of its PCI configuration space.

The LS2 FPGA communication happens during the NVMe registers write, FIFO Registers read . The IOs from the host are converted into pages and then the descriptor is formed with the IO information. The formed descriptor will be written into the OCM memory of FPGA. The NVMe IO completer thread handles this interrupt and reports to the NVMe controller, it is also responsible for updating the status registers on the FPGA.

3.2 Theory of Operations

3.2.1 NVMe PCIe Layer Software Flow

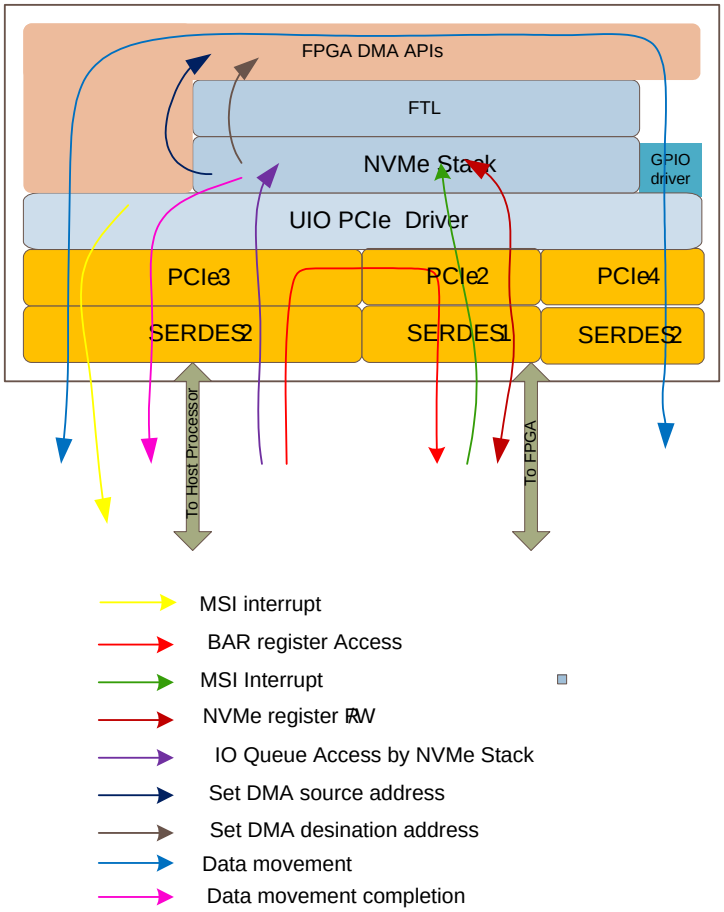


Figure 10: NVMe Data Flow Diagram

.

This diagram explains the flow of various NVMe transactions across the two PCIe interfaces on the LS2 processor.

3.2.2 PCIe Endpoint driver

3.2.2.1 Host - PCIe Endpoint Interaction

PCIe Endpoint layer interaction with the host system is shown in the figure.

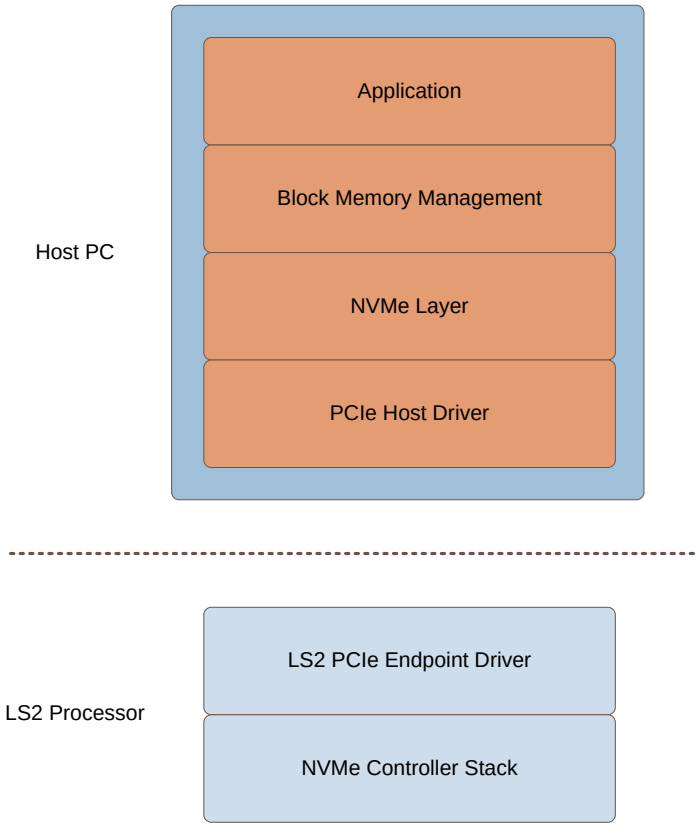


Figure 11: NVMe Host Driver Architecture

In the host processor, NVMe devices are recognized as block storage devices. The NVMe layer together with the block layer emulates a disk and is accessed in software by reading or writing ranges of blocks. The NVMe may be exposed as a single or as multiple NVMe volumes, depending on the namespace configuration. Each NVMe volume provides a range of logically-contiguous blocks. Hence, any read and write operations from the user space get an abstraction from the block access.

The LS2 is enumerated as a PCI endpoint. A PCI endpoint driver configures the LS2 to a device mode. The driver also sets the required PCI configuration parameters such that the LS2 PCI device is detected by the host as an NVMe controller.

3.2.2.2 PCIe Endpoint Configuration for PCI Controller

The PCI Express controller can be configured to operate as either a PCI Express root complex (RC) or endpoint (EP) device. The endpoint mode is selected at power on reset (POR) by reset configuration field, RCW[HOST_AGT_PEX]. SERDES protocol select may be used for setting the link width as 4x. Also the PCI controller is configured for 64 bit access.

3.2.2.3 PCIe Configuration Header

The PCIe3 host interface is configured as two physical functions(PF0,PF1)

PF0 - Non-Volatile Memory Controller

PF1 - Ethernet Controller

In EP mode, a PCI Express controller specific configuration details are specified by the configuration type 0 register set.

Some of the basic configuration header details are,

Vendor Identification: A unique number describing the originator of the PCI device.

PCI_VENDOR_ID_FREESCALE	0x1957

Device Identification: A unique number describing the device itself.

PCI_DEVICE_ID_NVME_CONTROLLER	0x0953

PF0 Class Code: This identifies the type of device that this is.

CLASS_CODE_NVME_CONTROLLER	0x010802

PF1 Class Code: This identifies the type of device that this is.

CLASS_CODE_ETHERNET_CONTROLLER	0x020000

3.2.2.4 PCIe Address Range Translation

Outbound and Inbound platform transactions to PCI Express controller are first mapped in the CPU's iATU (Internal Address Translation Unit) to a translation window to determine which PCI Express transactions are to be translated. The translation window parameters can be defined for the input address range and the target address range. So for any address issued, if the address range falls in the 'start base address' and 'End base address' , the address translation happens and the corresponding TLP's are routed to the target address as shown in the figure.

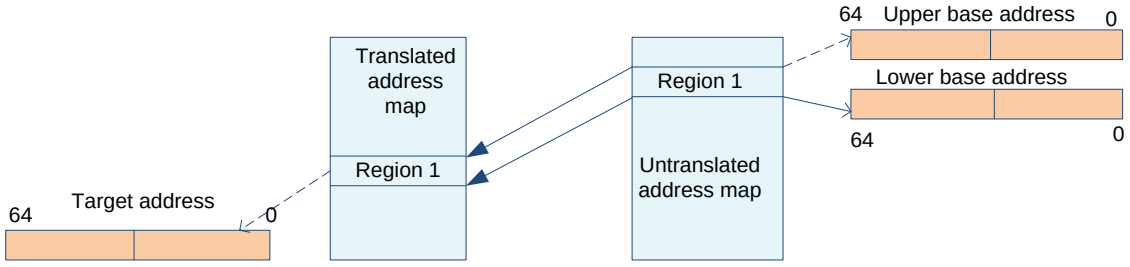


Figure 12: Address Translation

So for any inbound transaction, the data can be forwarded to the FPGA module, and for any outbound transaction the memory on the host would be accessible by the FPGA.

The iATU registers can be accessed through PCIe configuration access. The iATU can remap six outbound and six inbound address regions. The incoming address range and target address range can be specified for each translation window through PEX module internal configuration.

3.2.2.5 PCIe BAR Registers mapping

The PCI endpoint BAR 0 register is mapped to the address space corresponding to the FPGA PEX2 BAR2 register. This is possible through an address translation of the two address spaces. So for any inbound transaction for NVMe register access, the address corresponding to bar 0 is set as the inbound address in the translation window. The LS2 address space corresponding to FPGA device bar2 is set as target location.

3.2.2.6 PCIe Host Queue Access

For a memory access from LS2 to host, the LS2 acts as the initiator. The request from LS2 can be considered as an outbound transaction from the PCI endpoint controller. In the LS2 memory space, a region can be defined (corresponding to a bar register or reserved memory) for address mapping to the host. So, once an address is issued by LS2 in this address range, it is actually forwarded to the host target address space. Also, if any data needs to be copied from the host target address, it can be done by a memcpy or DMA read from the host to the reserved window memory in the LS2. LS2 can do outbound configuration to access 8GB memory of host. LS2 PEX3 maps the host memory in 0x14_0000_0000 to 15_FFFF_FFFF 8GB address range.

3.2.2.7 PCIe Endpoint Driver Sequence

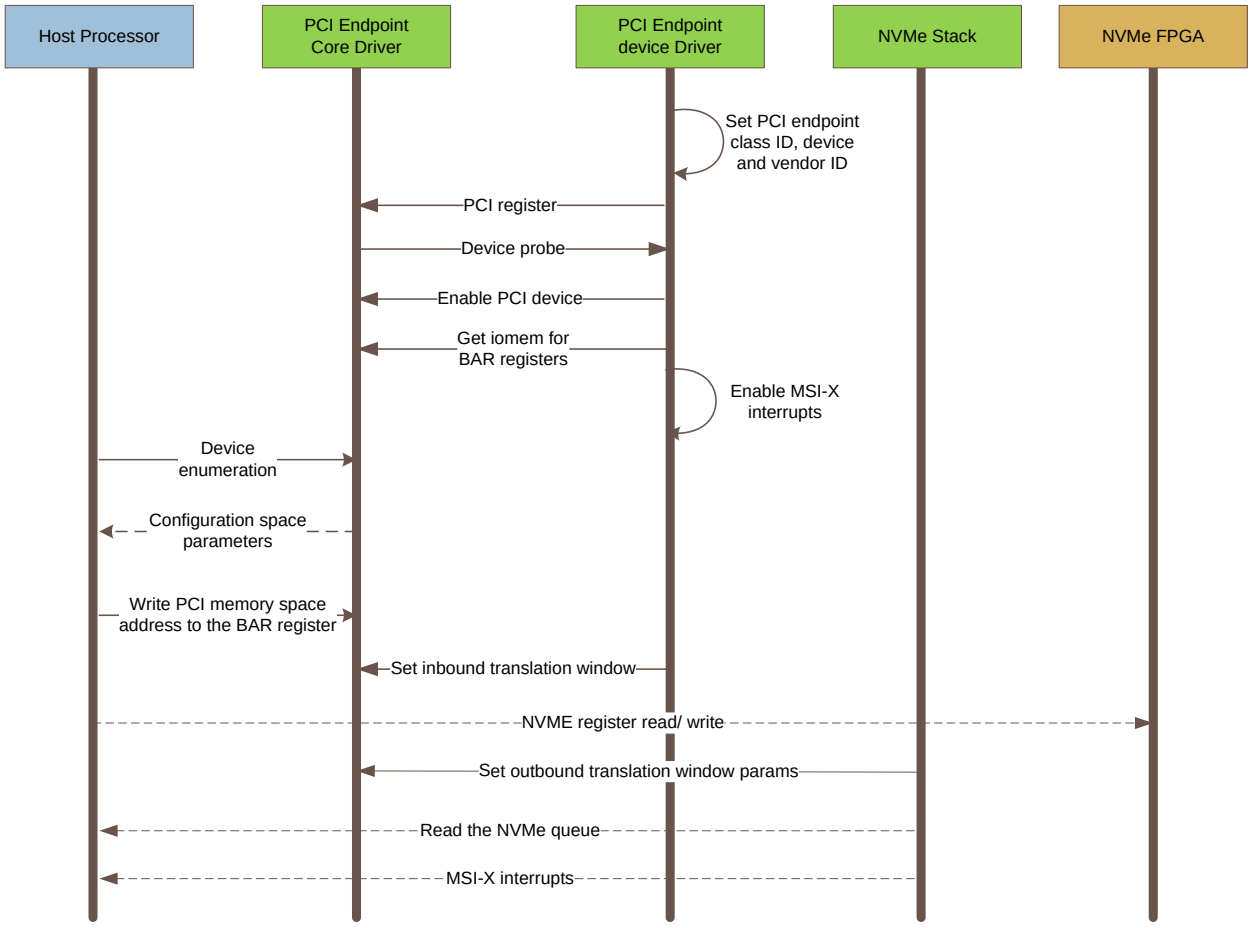


Figure 13: PCI-e End-point Driver Sequence Diagram

3.2.3 NVME PCIe Host Driver

This driver offers a PCI host interface for the FPGA device. The NVMe PCI driver together with the PCI core driver enumerates the FPGA device. The FPGA will be detected as PCI endpoint device in the bus number 3 device number 0 and function number 0. After enumeration, the FPGA device is allocated with a physical address space in the LS2. So the FPGA memory space will be accessible from the root complex using the FPGA bar registers.

The NVME controller registers are implemented in FPGA memory space corresponding to BAR2. Since BAR registers is memory addressable from the host, all the NVME registers can be accessed directly from the NVMe stack in the LS2 core.

The MSI interrupts may be configured, such that FPGA status may be send back to the upper block layers. An MSI interrupt is triggered whenever a register write happens in the NVMe controller registers. This interrupt notifies the NVMe stack about the register update.

FPGA card maintains FIFO registers in BAR 4, to maintain the current and previous values of NVMe controller. So memory corresponding to BAR 4 may also is mapped to the root complex interface.

3.2.3.1 PCIe Root Complex Configuration for PCI Controller

The PCI controller may be configured as a root complex device. The Root complex is selected at power on reset (POR) by reset configuration field, RCW[HOST_AGT_PEX]. SERDES protocol select may be used for setting the link width as 8x.

3.2.3.2 FPGA PCIe Configuration Header

The vendor id, device id and class code corresponding to the FPGA may be specified in the configuration header.

PCI_VENDOR_ID_FPGA	0x1957
PCI_DEVICE_ID_FPGA	0x0953
CLASS_CODE_FPGA	0x010802

3.2.4 Command Submission and Completion

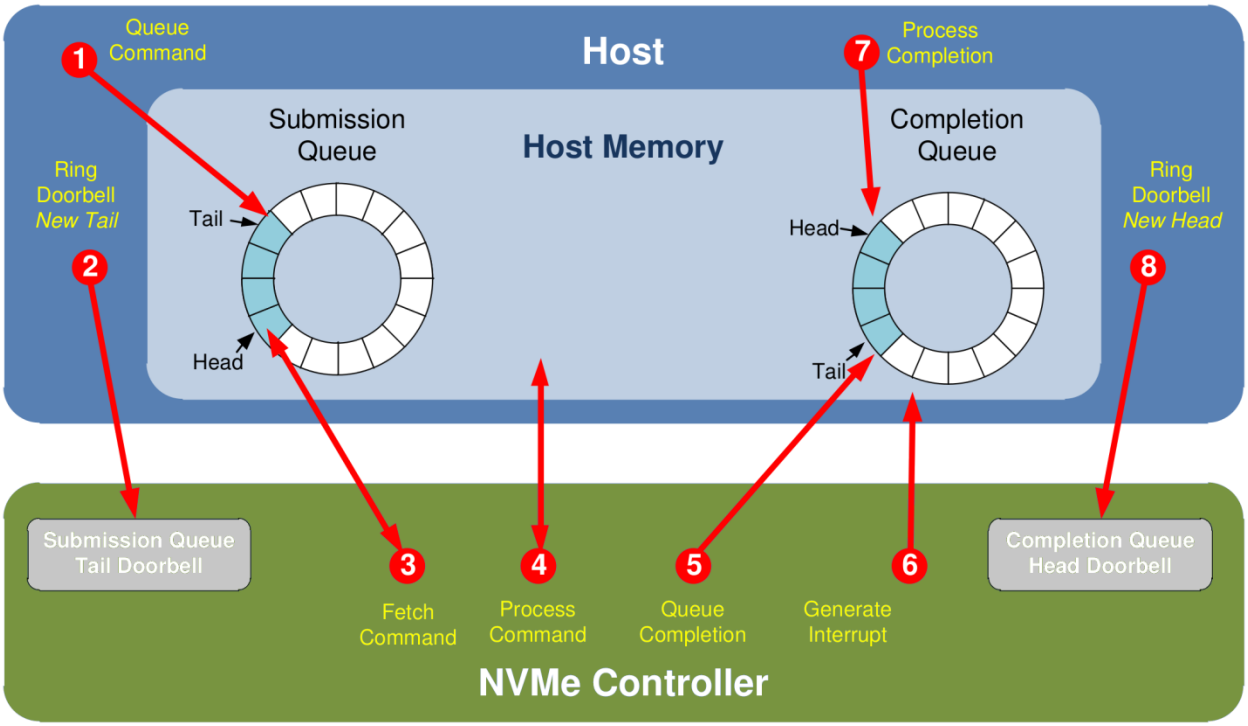


Figure 14: Command Submission and Completion Sequence Diagram

1. The host creates a command for execution within the appropriate Submission Queue in memory.

2. The host updates the Submission Queue Tail Doorbell register with the new value of the Submission Queue Tail entry pointer. This indicates to the controller that a new command(s) is submitted for processing.
3. The controller fetches the command(s) in the Submission Queue from memory for future execution. Arbitration is the method used to determine the Submission Queue from which the controller starts processing the next candidate command.
4. The controller then proceeds with execution of the next command. Commands may complete out of order (the order submitted or started execution).
5. After the command has completed execution, the controller writes a completion queue entry to the associated Completion Queue. As part of the completion queue entry, the controller indicates the most recent SQ entry that has been fetched.
6. The controller optionally generates an interrupt to the host to indicate that there is a completion queue entry to process. It is an MSI interrupt. Based on interrupt coalescing settings, an interrupt may or may not be indicated for the command.
7. The host processes the completion queue entry in the Completion Queue. This includes taking any actions based on error conditions indicated.
8. The host writes the Completion Queue Head Doorbell register to indicate that the completion queue entry has been processed. The host may process many entries before updating the associated CQHDBL register.

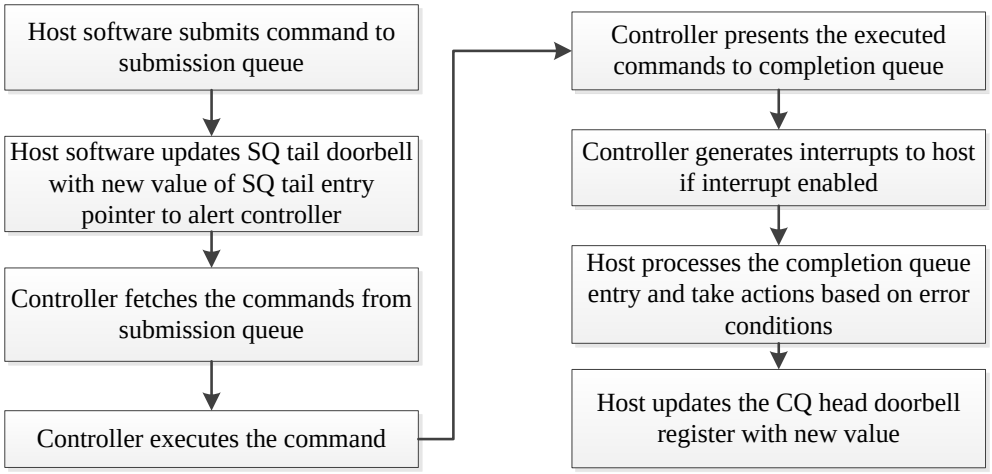


Figure 15: Command Submission and Completion Flow Chart

3.2.5 NVMe Register Access from the Host Processor

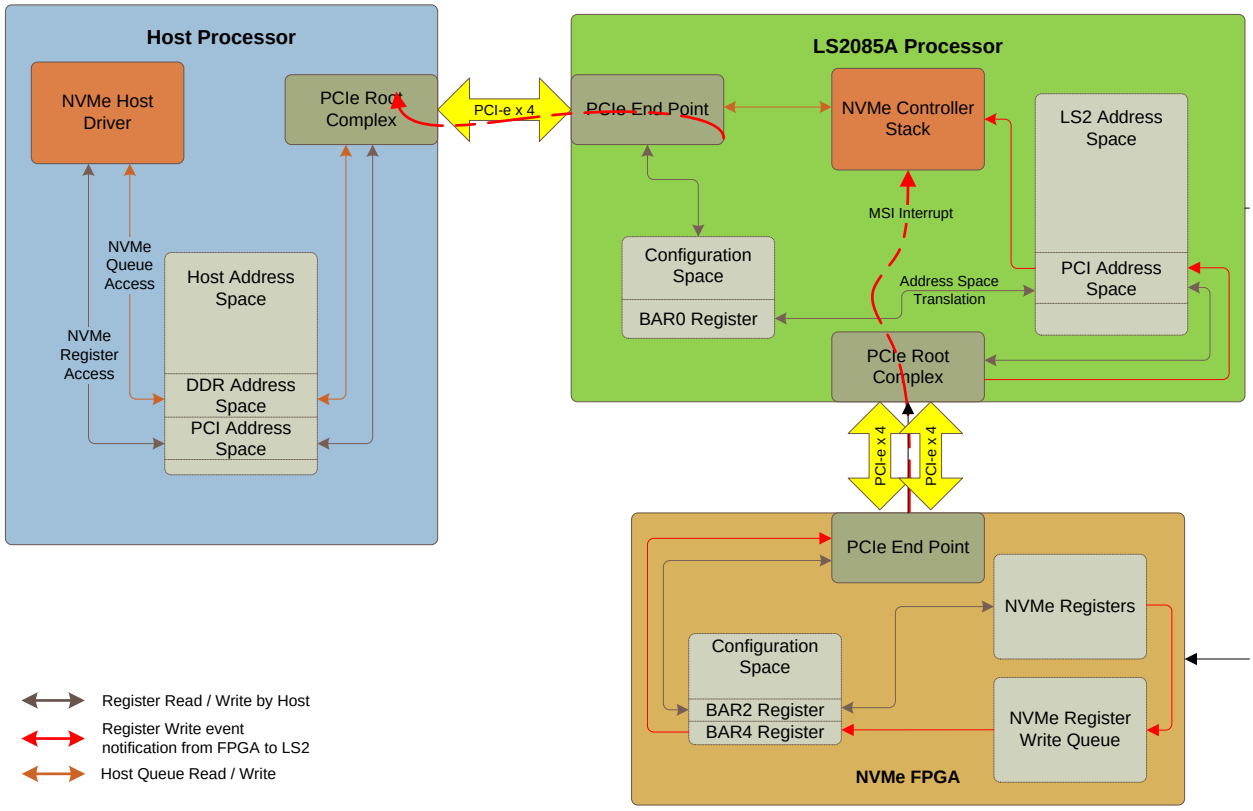


Figure 16: NVMe Host Driver Register Access

The above figure details the end to end data flow from the host processor to the FPGA during a register read / write operation by the host processor.

3.2.5.1 Register Read Sequence

The following is the sequence of events takes place during a register read by the host processor

1. The NVMe registers residing on the FPGA are exposed on the BAR0 of the PCI configuration space of the LS2 through UIO driver
2. The host processor IO remaps these registers to its address space
3. When a read operation is performed on this address space, the PCI read commands are sent to the LS2 processor's end-point driver
4. As the NVMe registers on the FPGA are mapped to the memory space of the LS2 processor, the address translation takes place and the request is routed by the UIO driver.
5. Read commands are sent on the PCI-e bus to the NVMe FPGA's BAR0 registers
6. The value read is transmitted back to the host processor

3.2.5.2 Register Write Sequence

The following is the sequence of events takes place during a register write by the host processor

1. The host processor writes the register changes to the BAR2 of the LS2 processor's end-point PCI configuration space
2. As the NVMe registers on the FPGA are mapped to the memory space of the LS2 processor, the address translation takes place and the write request is routed by UIO driver and eventually to the FPGA's register space
3. The FPGA queues this register write event in its BAR4 register space, in the form as a FIFO
4. It raises and GPIO interrupt to the LS2 processor to intimate a new entry has been added to the register write queue and updates the FIFO count register
5. The LS2 processor de-queues the write event from the BAR4 FIFO and processes the change in the corresponding register
6. The new and offset values of the register write event are masked with the read-only register mask, to mask out changes to the read-only and reserved bits
7. The resulting masked new register values are compared for changes in the bits and the corresponding offset of the changed register bits are computed
8. The new value and offset is used to process the new register value and the corresponding changes are applied onto the NVMe stack
9. If the offset corresponds to a Doorbell register, then the doorbell processing takes place

The structure of the FIFO element in the FPGA is as follows:

```
typedef struct reg_write_event {
        uint64_t new_value;
        uint64_t offset;
} reg_write_event_t;
```

Each register write event is de-queued from the FIFO on the FPGA upon the reading it, by the LS2 processor.

The old and new value of the register is compared to find the modified bit on the register and necessary action is performed.

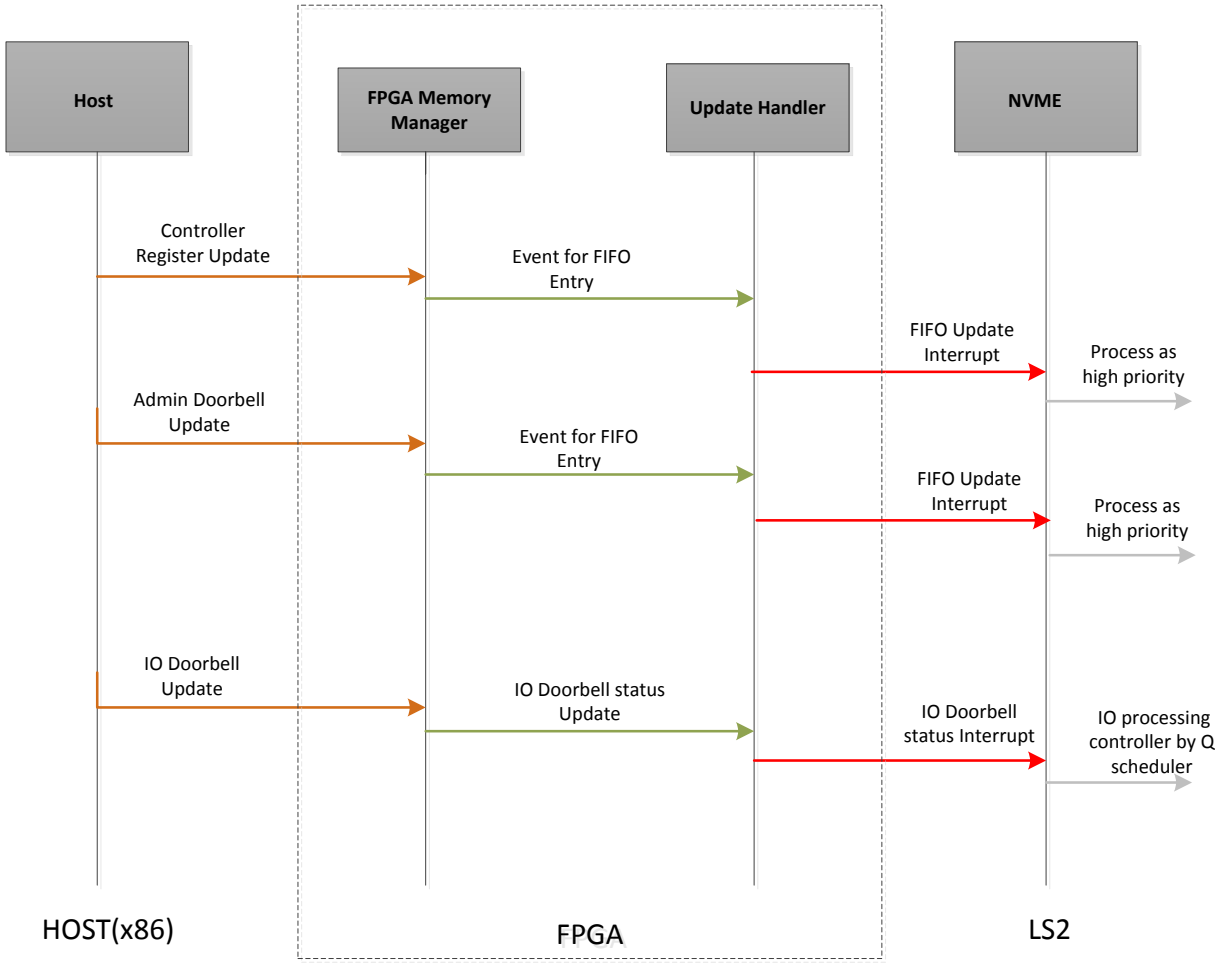


Figure 17: NVMe Register Update Handling Sequence

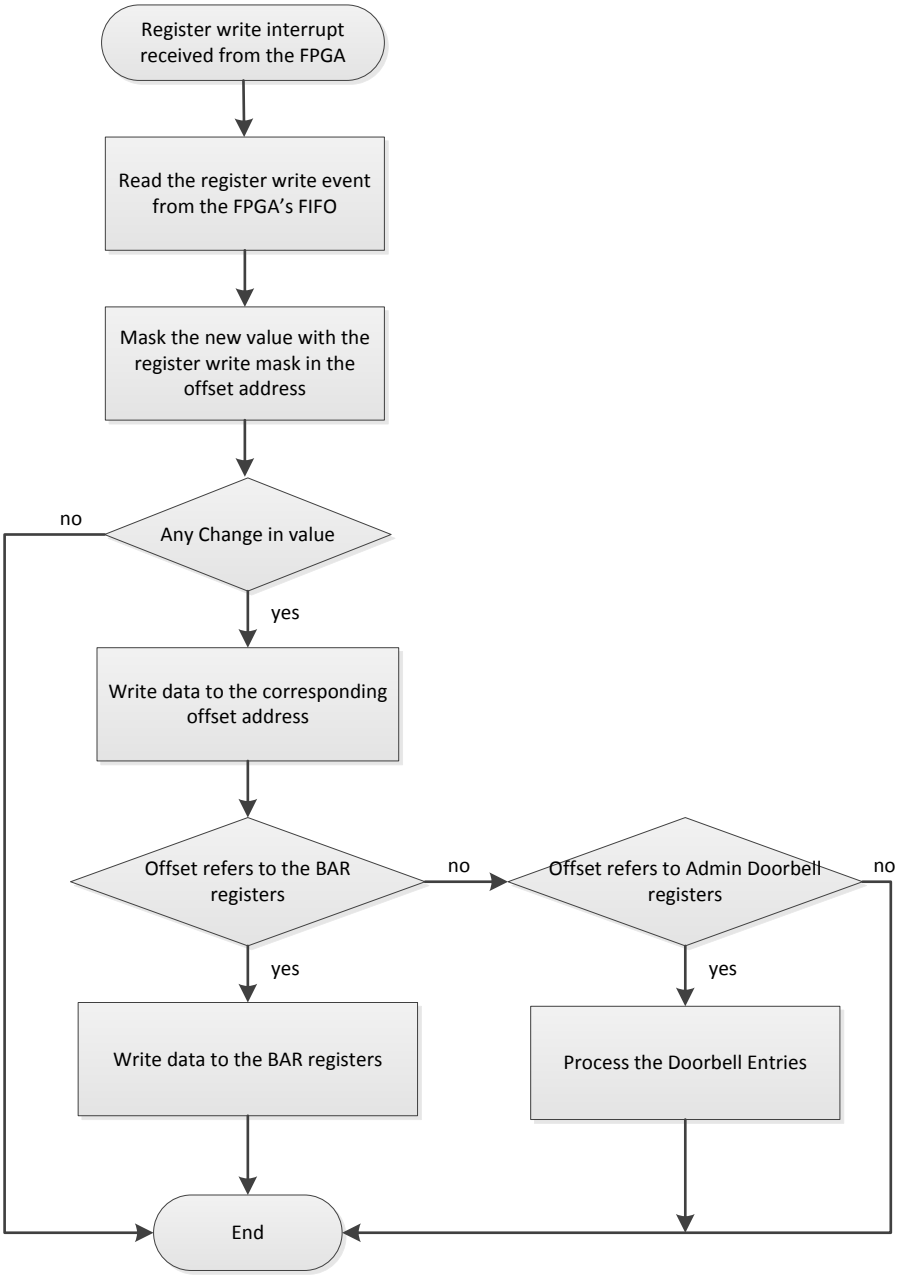


Figure 18: Handling Register Writes Flow Chart

3.2.5.3 Door Bell Register Write Processing

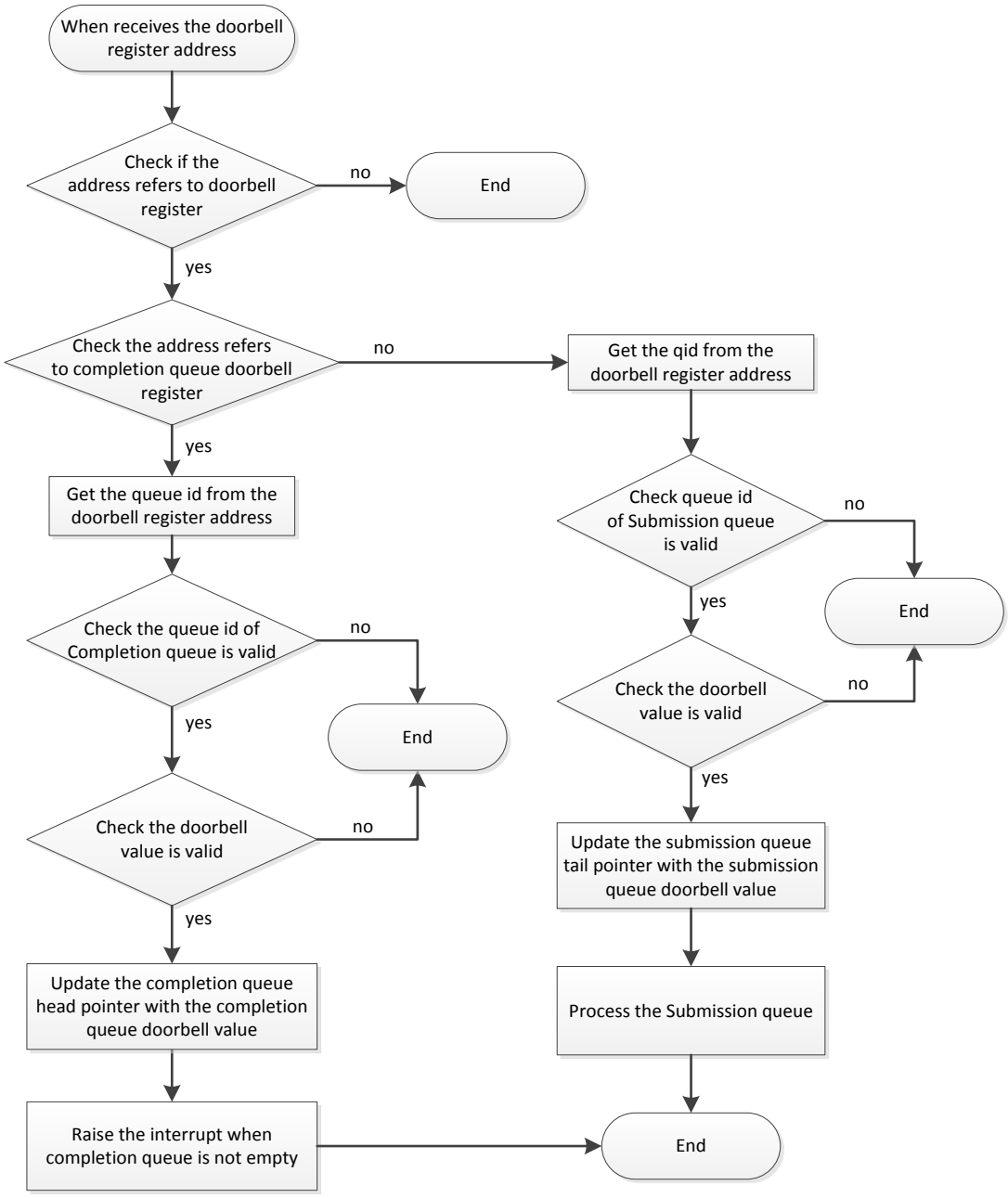


Figure 19: Handling Doorbell Register Writes Flow Chart

3.2.5.4 Processing Changes in NVMe Controller Registers

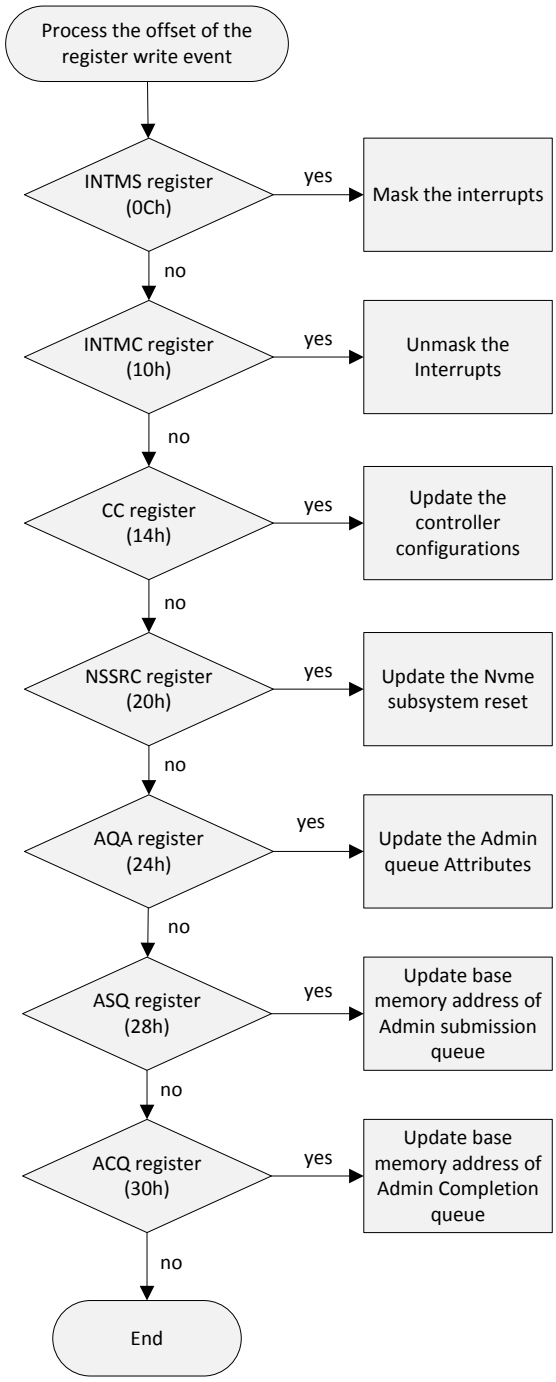


Figure 20: Processing Changes in NVMe Controller Registers Flow Chart

3.2.6 Submission Queue Processing

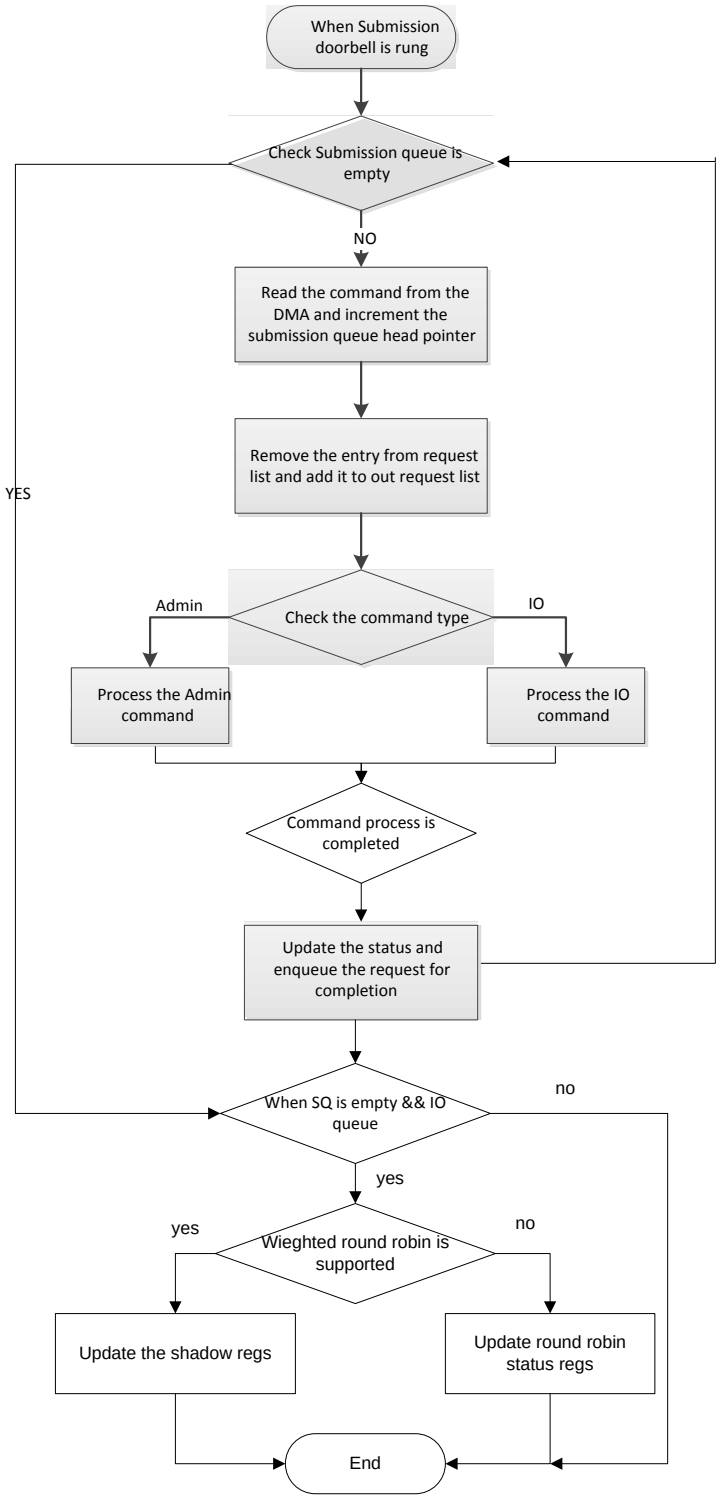


Figure 21: Handling Submission Queue Doorbell Flow Chart

3.2.6.1 Admin Command Processing

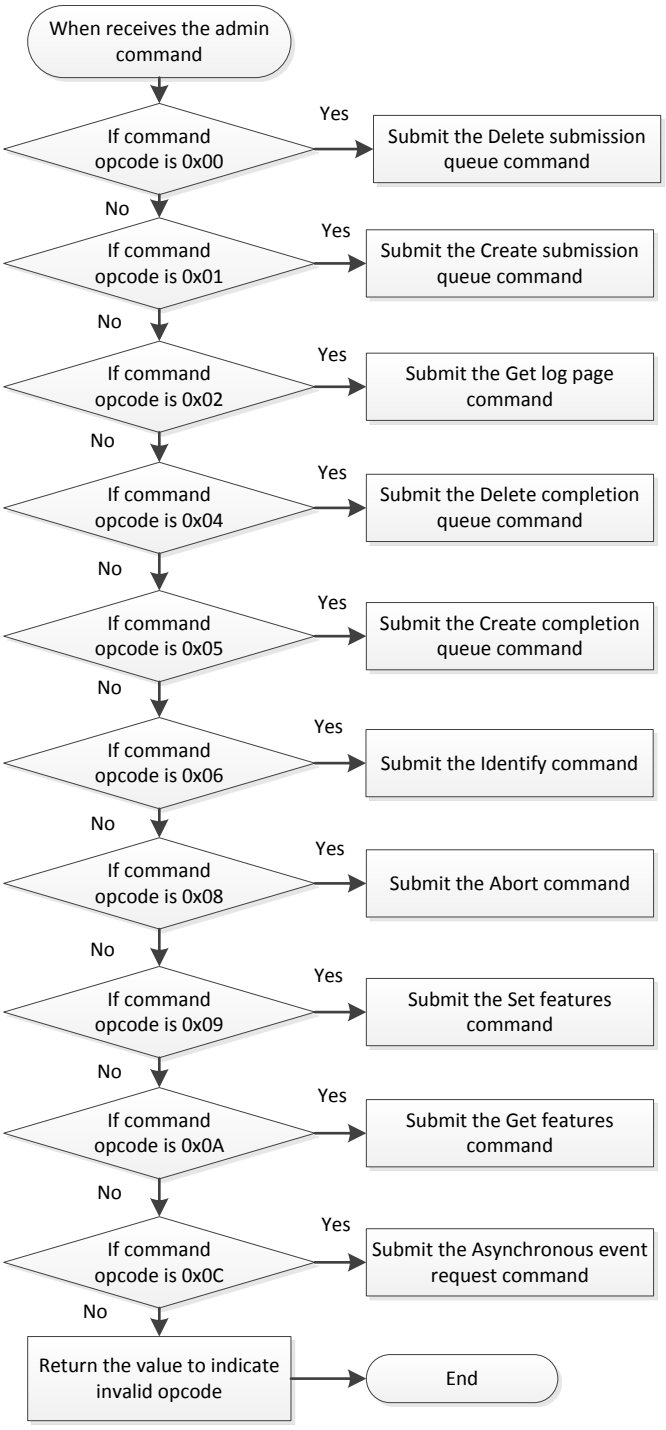


Figure 22: Processing Admin Commands Flow Chart

3.2.6.2 IO Command Processing

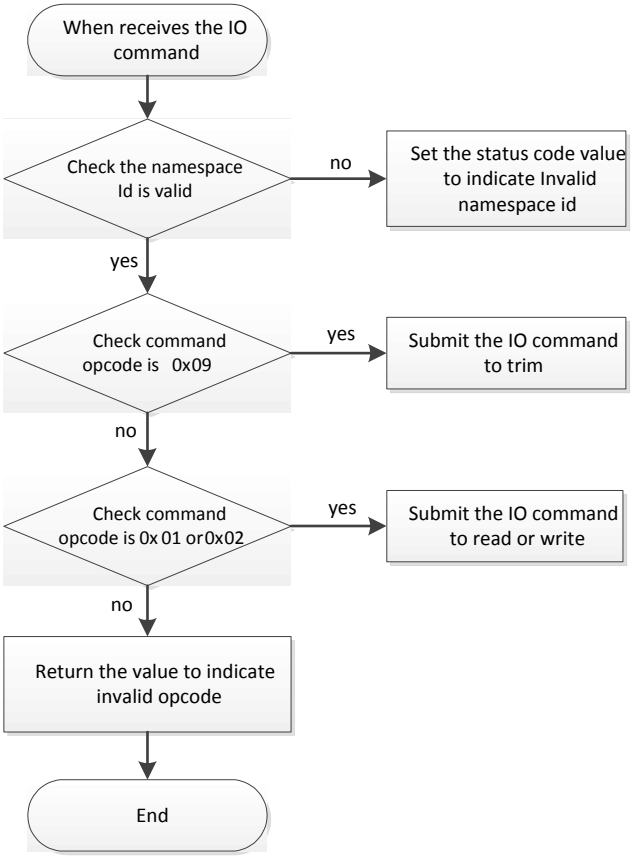


Figure 23: Handling IO Commands Flow Chart

3.2.7 Command Arbitration

After a command is submitted to the submission queue, the host processor updates the Submission Queue Tail Pointer to the Submission Queue Doorbell and moves the Submission Queue Tail Pointer past the corresponding. The NVMe controller transfers submitted commands to its local memory for subsequent processing.

Arbitration is the method used to determine the Submission Queue from which the controller will start processing the next candidate command(s). Once a Submission Queue is selected using arbitration, the Arbitration Burst setting determines the maximum number of commands that the controller may start processing from that Submission Queue before arbitration shall again take place.

The NVMe controller can advertise its supported Arbitration Mechanism through its Controller Capabilities Register (CAP.AMS). The Host can select the desired Arbitration Mechanism through the Controller Configuration register (CC.AMS).

Three methods of arbitration can be supported by the NVMe controller.

CAP.AMS – Arbitration Mechanism Selection

Table 2: Arbitration Mechanism Selection Capabilities Register

Bit	Value	Arbitration Mechanism Supported
Default Supported		Round Robin
17	1	Weighted Round Robin with Urgent Priority Class
18	1	Vendor Specific

CC.AMS - Arbitration Mechanism Selection

Table 3: Arbitration Mechanism Selection Configuration Register

Bits	Value	Arbitration Mechanism Supported
13:11	000	Round Robin
	001	Weighted Round Robin with Urgent Priority Class
	010-110	Reserved
	111	Vendor Specific

3.2.7.1 Round-Robin Arbitration

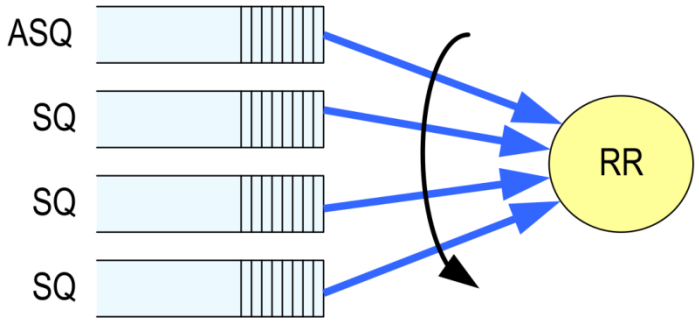


Figure 24: Round-Robin Arbitration

With round-robin based command arbitration, all Submission Queues (including the Admin Submission Queue) are treated with the same priority. Multiple commands are processed from each queue per round,

before processing the next queue. This number of commands being processed on each submission queue per round is based on the Arbitration Burst setting.

3.2.7.2 Weighted Round-Robin with Urgent Priority Class Arbitration

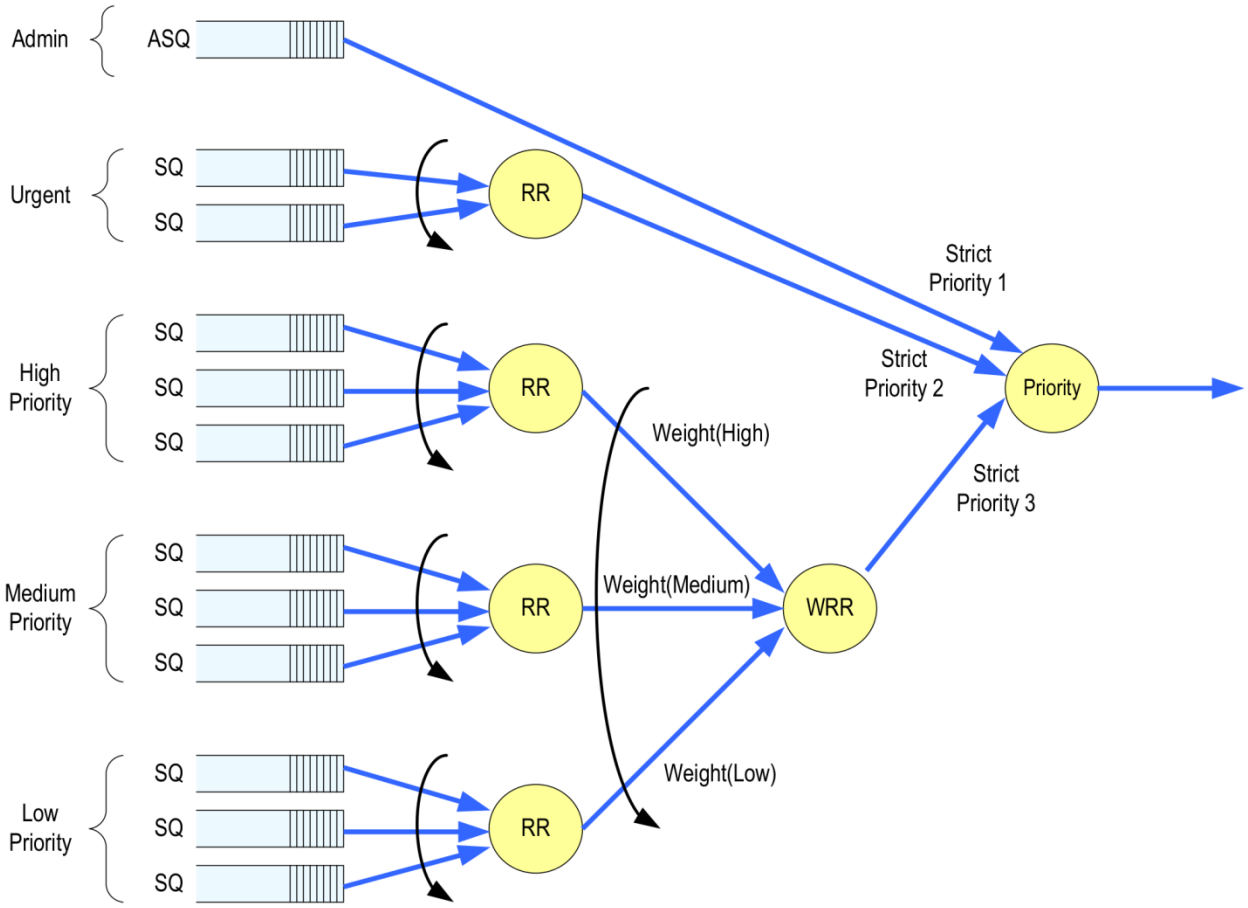


Figure 25: Weighted Round-Robin with Urgent Priority Class Arbitration

In this arbitration mechanism, there are three strict priority classes and three weighted round robin priority levels. If Submission Queue A is of higher strict priority than Submission Queue B, then all candidate commands in Submission Queue A shall start processing before candidate commands from Submission Queue B start processing.

The highest strict priority class is the Admin class that includes any command submitted to the Admin Submission Queue. This class has the highest strict priority above commands submitted to any other Submission Queue.

The next highest strict priority class is the Urgent class. Any I/O Submission Queue assigned to the Urgent priority class is serviced next after commands submitted to the Admin Submission Queue, and before any commands submitted to a weighted round robin priority level.

The lowest strict priority class is the Weighted Round Robin class. This class consists of the three weighted round robin priority levels (High, Medium, and Low) that share the remaining bandwidth using weighted round robin arbitration.

Host software controls the weights for the High, Medium, and Low service classes via the “Set Features” Admin command. Round robin is used to arbitrate within multiple Submission Queues assigned to the same weighted round robin level. The number of candidate commands that may start processing from each Submission Queue per round is either the Arbitration Burst setting or the remaining weighted round robin credits, whichever is smaller.

3.2.7.3 Software Implementation Structure

The software implementation of Q Scheduler mainly depends on a 128 bit register inside FPGA memory. This 128 bit register is used as a status register for all the 128 IO SQ Doorbells.

If bit n is set to 1, IO SQ with ID ‘n+1’ has unprocessed commands; else there are no unprocessed commands yet. Based on this register and with help of some mask and shadow (local copy) registers, the Q scheduler flow gets to process all the commands from all the SQ’s as specified in the previous section.

```
structNvmeQsched {
        uint64_t        *iodb_status_regs;
        uint64_t        prio_mask_regs[4][2];
        uint64_t        prio_shadow_regs[4][2];
        uint16_t        NumOfActiveSQs;
        int             msqid;
};
```

- **prio_shadow_regs:**128 bit registers for 128 IO SQs with various NVMe priority levels – URGENT, HIGH, MEDIUM, And LOW. Accessed as 2 64-bit values. Shadow registers are always obtained by doing the following

  prio_shadow_regs[prio_level] = iodb_status_regs&prio_mask_regs[prio_level]
  whereprio_level is one of URGENT, HIGH, MEDIUM, And LOW.

- **prio_mask_regs:**128 bit register for each of the 4 priority levels. Bit n-1 in prio_mask_regs[p] is set to 1 if an IO SQ of ID ‘n’ is created with priority ‘p’. If the IO SQ is deleted, the corresponding bit in the corresponding priority mask register is reset to 0. Always modified only when IO SQ is created or deleted.
- **iodb_status_regs:** Refers to address of the 128 bit I/O SQ doorbell status register in the FPGA. Used to fetch the current status of all IO DB registers.
- **NumOfActiveSQs:** Number of active I/O Submission Queues. Used for iteratively checking all bits in all active Queues and fetch Q’s with commands to be processed.

- **msqid:**Identifier used by the Q scheduler to send IOSQ IDs to IO processor thread via message queue.

3.2.7.4 Queue Scheduler Flowchart

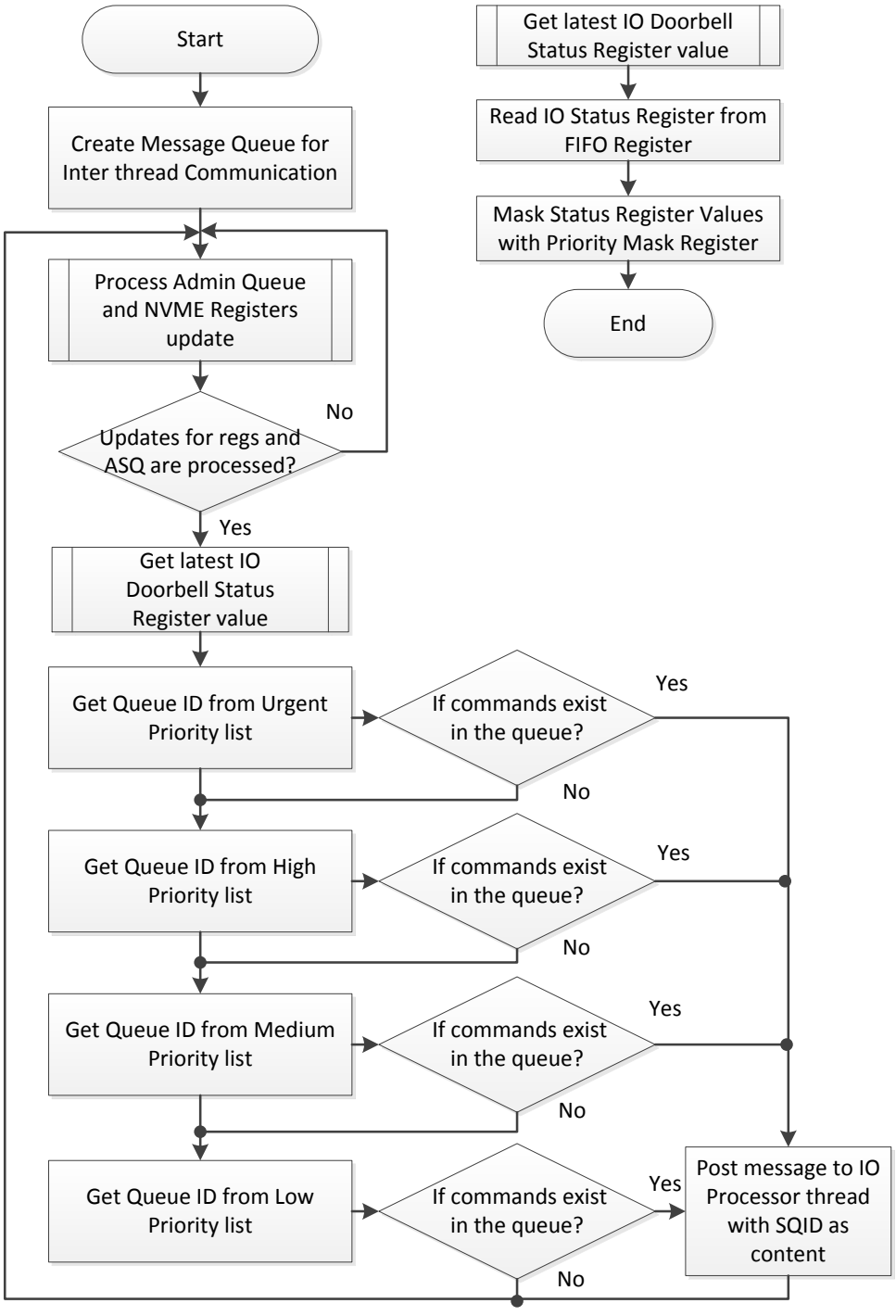


Figure 26: Queue scheduler Flowchart

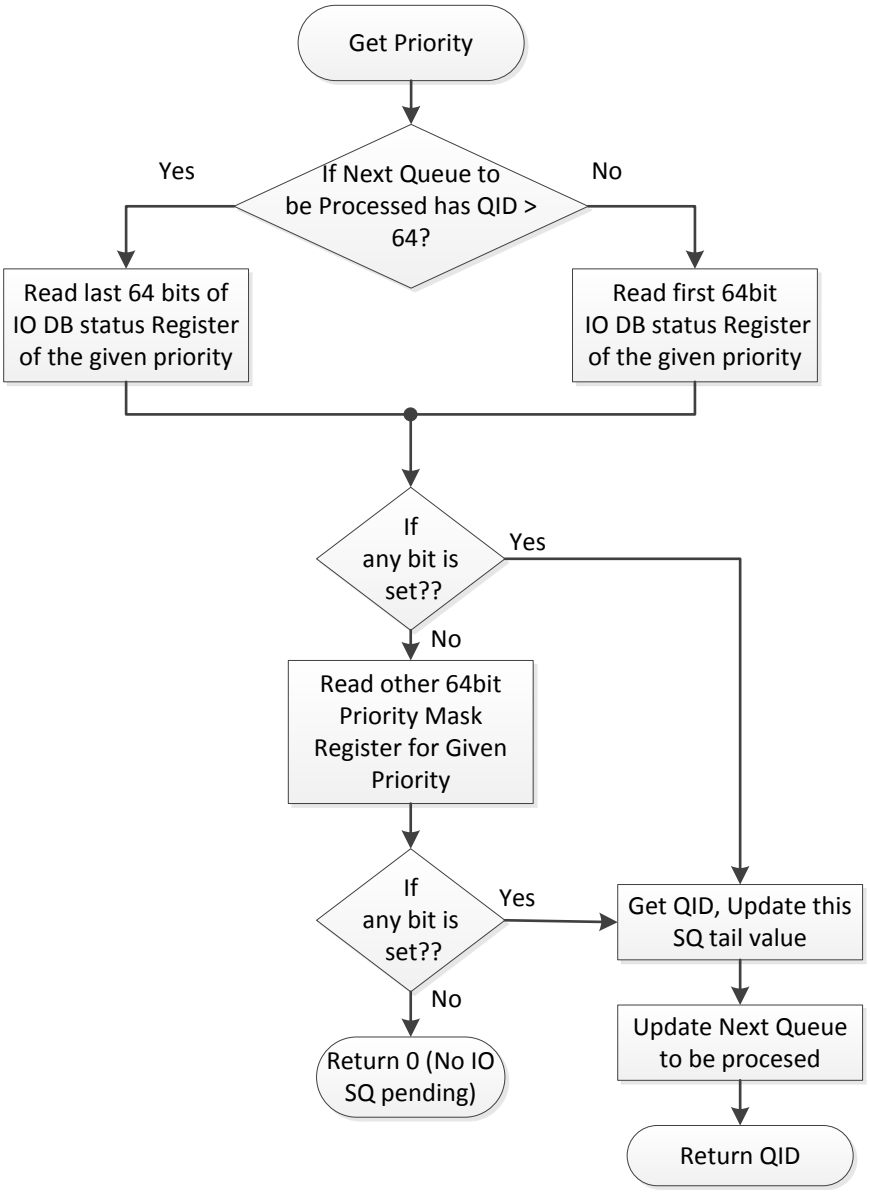


Figure 27: Weighted Round Robin Queue Scheduling Flowchart

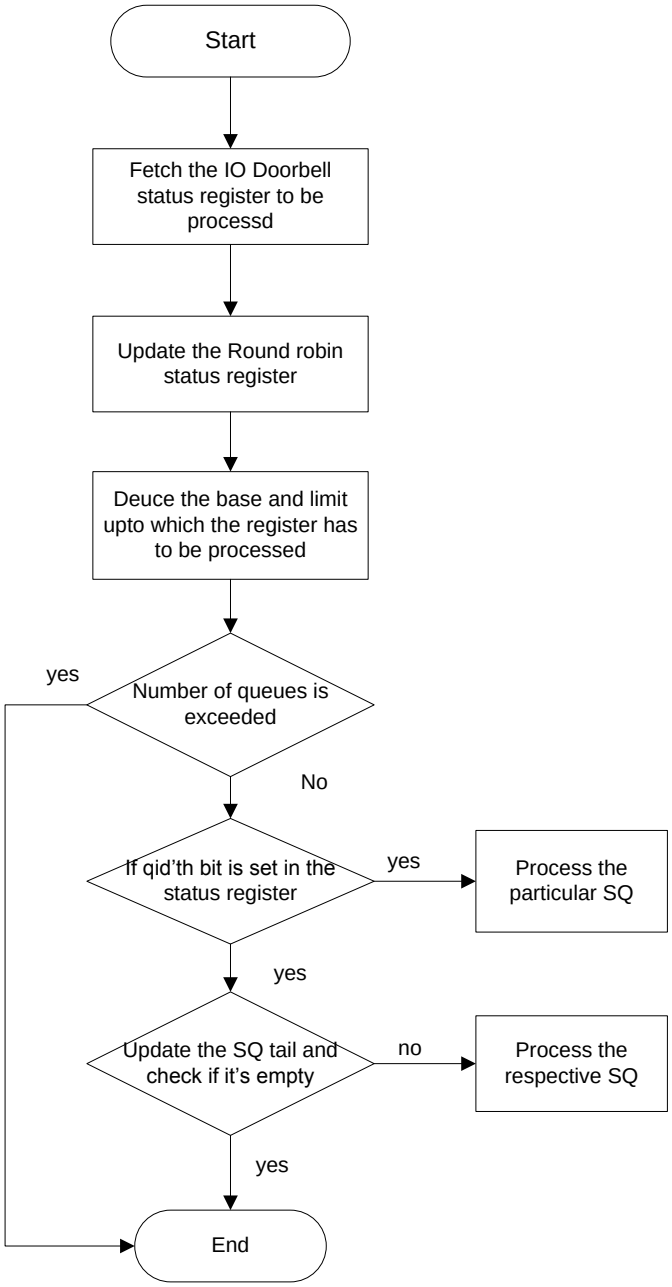


Figure 28: Round Robin Queue Scheduling

3.2.7.5 Vendor Specific Arbitration

This option can be used for developing an arbitration in the future which is able to provide the desired performance.

3.2.8 Thread Architecture

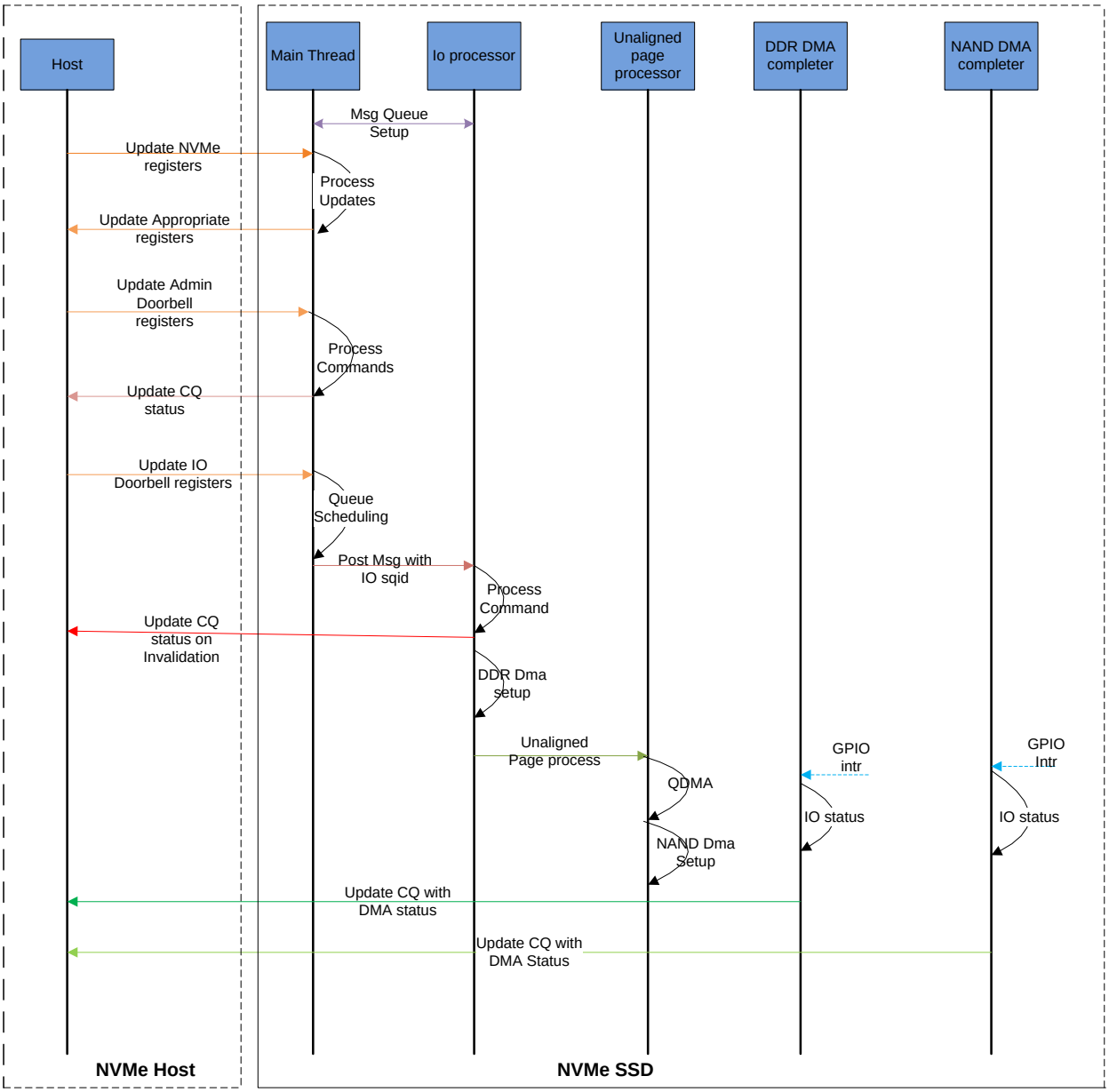


Figure 29: Sequence diagram for Thread Architecture

3.2.9 Initialization Sequence

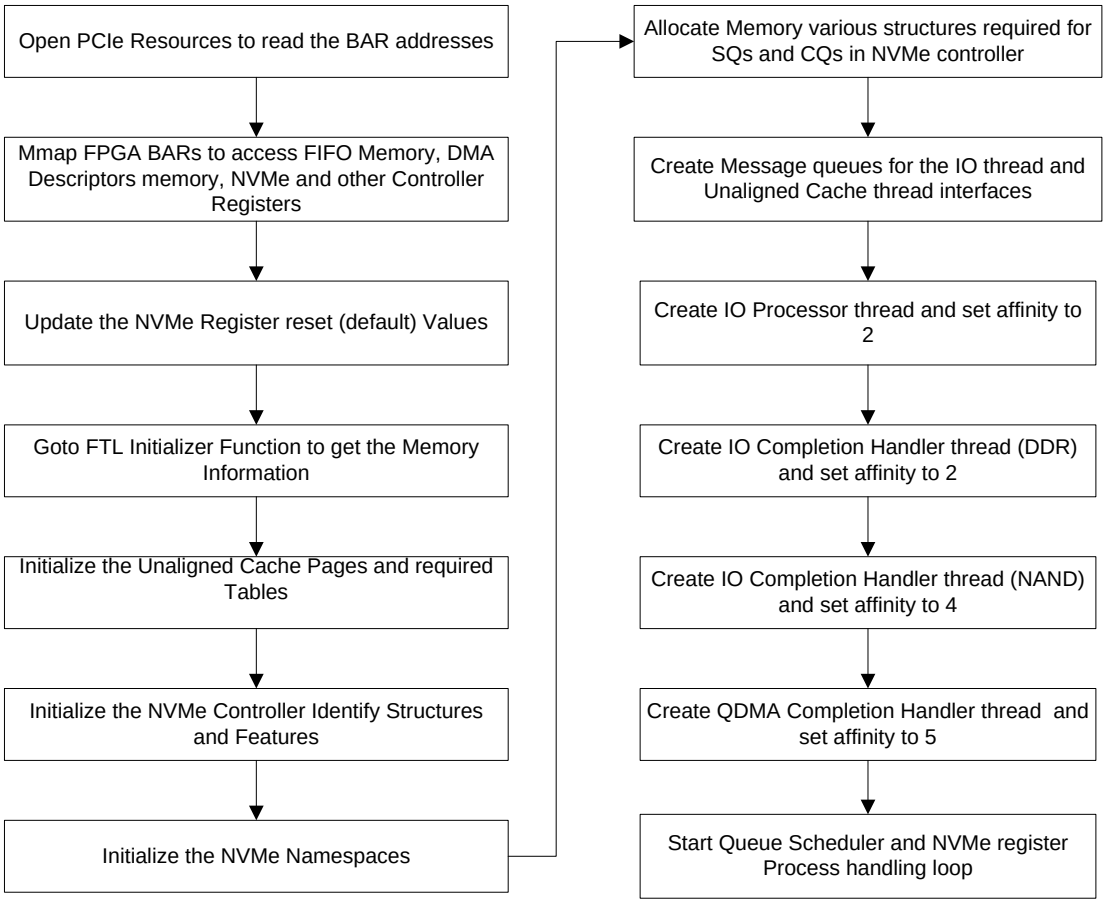


Figure 30: Initialization sequence Flowchart

3.3 NVMe Controller Registers

Table 4: NVMe Controller Registers

Start	End	Symbol	Description
00h	07h	CAP	Controller Capabilities
08h	0Bh	VS	Version
0Ch	0Fh	INTMS	Interrupt Mask Set
10h	13h	INTMC	Interrupt Mask Clear
14h	17h	CC	Controller Configuration
18h	1Bh	Reserved	Reserved
1Ch	1Fh	CSTS	Controller Status
20h	23h	NSSR	NVM Subsystem Reset
24h	27h	AQA	Admin Queue Attributes
28h	2Fh	ASQ	Admin Submission Queue Base Address
30h	37h	ACQ	Admin Completion Queue Base Address
38h	3Bh	CMBLOC	Controller Memory Buffer Location
3Ch	3Fh	CMBSZ	Controller Memory Buffer Size
40h	EFFh	Reserved	Reserved
F00h	FFFh	Reserved	Command Set Specific
1000h	1003h	SQ0TDBL	Submission Queue 0 Tail Doorbell (Admin)
1000h + (1 * (4 << CAP.DSTRD))	1003h + (1 * (4 << CAP.DSTRD))	CQ0HDBL	Completion Queue 0 Head Doorbell (Admin)
1000h + (2 * (4 << CAP.DSTRD))	1003h + (2 * (4 << CAP.DSTRD))	SQ1TDBL	Submission Queue 1 Tail Doorbell
1000h + (3 * (4 << CAP.DSTRD))	1003h + (3 * (4 << CAP.DSTRD))	CQ1HDBL	Completion Queue 1 Head Doorbell
1000h + (4 * (4 << CAP.DSTRD))	1003h + (4 * (4 << CAP.DSTRD))	SQ2TDBL	Submission Queue 2 Tail Doorbell
1000h + (5 * (4 <<CAP.DSTRD))	1003h + (5 * (4 <<CAP.DSTRD))	CQ2HDBL	Completion Queue 2 Head Doorbell
…	…	…	…
1000h+ (2y * (4 <<CAP.DSTRD))	1003h + (2y * (4 <<CAP.DSTRD))	SQyTDBL	Submission Queue y Tail Doorbell

1000h + ((2y + 1) * (4 <<CAP.DSTRD))	1003h +((2y + 1) * (4 <<CAP.DSTRD))	CQyHDBL	Completion Queue y Head Doorbell

3.3.1 Software implementation of NVMe BAR0 registers

```
typedefstructNvmeBar{
        cap_t           cap
        vs_t            vs;
        intms_t         intms;
        intmc_t         intmc;
        cc_t            cc;
        uint32_t        rsvd1;
        csts_t          csts;
        nssr_t          nssr;
        aqa_t           aqa;
        asq_t           asq;
        acq_t           acq;
        cmbloc_t        cmbloc;
        cmbsz_t         cmbsz;
        uint64_t        rsvd2[63];
        doorbell_t      db[MAX_NUM_IO_QUEUES+1];
}__attribute__((packed)) NvmeBar_t;

typedefstruct doorbell {
        uint32_t        sqtdbl;
        uint32_t        rsvd1;
        uint32_t        cqhdbl;
        uint32_t        rsvd2;
} __attribute__((packed)) doorbell_t;

NvmeBar_t volatile *pNvmeBar0;
StaticconstNvmeBar_tNvmeWriteMask;
StaticconstNvmeBar_tNvmeBarReset;
```

3.3.2 Controller Capabilities Register (CAP)

3.3.2.1 Software implementation

```
Typedefstructcap_reg {
        uint16_t        mqes;
        uint8_t         cqr:1;
        uint8_t         ams:2;
        uint8_t         rsvd1:5;
        uint8_t         to;
        uint8_t         dstrd:4;
        uint8_t         nssrs:1;
        uint8_t         css;
        uint8_t         rsvd2:3;
        uint8_t         mpsmin:4;
        uint8_t         mpsmax:4;
        uint8_t         rsvd3;
} __attribute__((packed)) cap_t ;
```

3.3.2.2 Register Write Mask

```
staticconstNvmeBar_tNvmeWriteMask = {
        .cap = {
                .mqes           =       0xFFFF,
                .cqr            =       0x1,
                .ams            =       0x3,
                .rsvd1          =       0x1F,
                .to             =       0xFF,
                .dstrd          =       0xF,
                .nssrs          =       0x1,
                .css            =       0xFF,
                .rsvd2          =       0x7,
                .mpsmin         =       0xF,
                .mpsmax         =       0xF,
                .rsvd3          =       0xFF,
```

```
        },
};
```

3.3.2.3 Register Reset Value

```
staticconstNvmeBar_tNvmeBarReset = {
        .cap = {
                .mqes           =       0xFFFF,
                .cqr            =       0x1,
                .ams            =       0x0,
                .rsvd1          =       0x0,
                .to             =       0x1,
                .dstrd          =       0x0,
                .nssrs          =       0x1,
                .css            =       0x1,
                .rsvd2          =       0x0,
                .mpsmax         =       0xF,
                .mpsmin         =       0x0,
                .rsvd3          =       0x0,
        },
};
```

3.3.3 Version (VS)

3.3.3.1 Software implementation

```
typedefstructvs_reg {
        uint8_t         rsvd;
        uint8_t         mnr;
        uint16_ t       mjr;
} __attribute__((packed)) vs_t;
```

3.3.3.2 Register Write Mask

```
staticconstNvmeBar_tNvmeWriteMask = {
        .vs = {
                .rsvd           =       0xFF,
```

```
                .mnr            =       0xFF,
                .mjr            =       0xFFF,
        },
};
```

3.3.3.3 Register Reset Value

```
staticconstNvmeBar_tNvmeBarReset = {
        .vs = {
                .rsvd           =       0,
                .mnr            =       0x2,
                .mjr            =       0x1,
        },
};
```

3.3.4 Interrupt Mask Set (INTMS)

3.3.4.1 Software implementation

```
typedefstructintms_reg {
        uint32_tivms;
}__attribute__((packed)) intms_t;
```

3.3.4.2 Register Write Mask

```
staticconstNvmeBar_tNvmeWriteMask = {
        .intms = {
                .ivms           =       0x0,
        },
};
```

3.3.4.3 Register Reset Value

```
staticconstNvmeBar_tNvmeBarReset = {
        .intms = {
                .ivms           =       0x0,
        },
};
```

3.3.5 Interrupt Mask Clear (INTMC)

3.3.5.1 Software implementation

```
typedefstructintmc_reg {
        uint32_t        ivmc;
}__attribute__((packed)) intmc_t;
```

3.3.5.2 Register Write Mask

```
staticconstNvmeBar_tNvmeWriteMask = {
        .intmc = {
                .ivmc           =       0x0
        },
};
```

3.3.5.3 Register Reset Value

```
staticconstNvmeBar_tNvmeBarReset = {
        .intmc = {
                .ivmc           =       0x0,
        },
};
```

3.3.6 Controller Configuration (CC)

3.3.6.1 Software implementation

```
typedefstructcc_reg {
        uint8_t         en:1;
        uint8_t         rsvd1:3;
        uint8_t         iocss:3;
        uint8_t         mps:4;
        uint8_t         ams:3;
        uint8_t         shn:2;
```

```
        uint8_t         iosqes:4;
        uint8_t         iocqes:4;
        uint8_t         rsvd2;
} __attribute__((packed)) cc_t;
```

3.3.6.2 Register Write Mask

```
staticconstNvmeBar_tNvmeWriteMask = {
        .cc = {
                .rsvd1  =       0x7,
                .rsvd2  =       0xFF,
        },
};
```

3.3.6.3 Register Reset Value

```
staticconstNvmeBar_tNvmeBarReset = {
        .cc = {
                .en             =       0x0,
                .rsvd1          =       0x0,
                .iocss          =       0x0,
                .mps            =       0x0,
                .ams            =       0x0,
                .shn            =       0x0,
                .iosqes         =       0x0,
                .iocqes         =       0x0,
                .rsvd2          =       0x0,
        },
};
```

3.3.6.4 Processing Changes in the Register

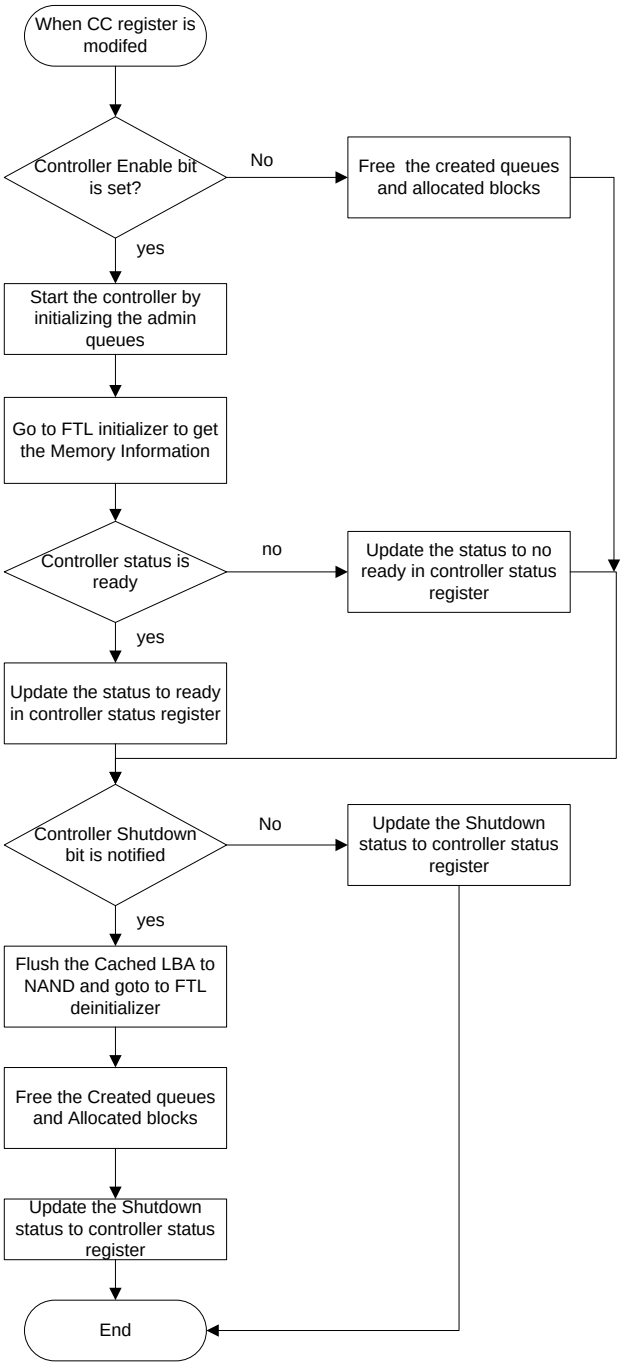


Figure 31: Processing Change in Controller Configuration Flow Chart

3.3.7 Controller Status (CSTS)

3.3.7.1 Software implementation

```
typedefstructcsts_reg {
        uint8_t  rdy:1;
        uint8_t  cfs:1;
        uint8_t  shst:2;
        uint8_t  nssro:1;
        uint8_t  pp:1;
        uint32_trsvd:26;
}__attribute__((packed)) csts_t;
```

3.3.7.2 Register Write Mask

```
staticconstNvmeBar_tNvmeWriteMask = {
        .csts = {
                .rdy            =       0x1,
                .cfs            =       0x1,
                .shst           =       0x3,
                .pp             =       0x1,
                .rsvd           =       0x3FFFFFF,
        },
};
```

3.3.7.3 Register Reset Value

```
staticconstNvmeBar_tNvmeBarReset = {
        .csts = {
                .rdy            =       0x0,
                .cfs            =       0x0,
                .shst           =       0x0,
                .nssro          =       0x0,
```

```
                .pp             =       0x0,
                .rsvd           =       0x0,
        },
};
```

3.3.8 NVM Subsystem Reset (NSSR)

3.3.8.1 Software implementation

```
typedefstructnssr_reg {
        uint32_t        nssrc;
}__attribute__((packed)) nssr_t;
```

3.3.8.2 Register Write Mask

```
staticconstNvmeBar_tNvmeWriteMask = {
        .nssr = {
                .nssrc  =       0x0
        },
};
```

3.3.8.3 Register Reset Value

```
staticconstNvmeBar_tNvmeBarReset = {
        .nssr = {
                .nssrc  =       0x0,
        },
};
```

3.3.9 Admin Queue Attributes (AQA)

3.3.9.1 Software implementation

```
typedefstructaqa_t {
        uint16_t        asqs:12;
```

```
        uint8_t         rsvd1:4;
        uint16_t        acqs:12;
        uint8_t         rsvd2:4;
} __attribute__((packed)) aqa_t;
```

3.3.9.2 Register Write Mask

```
staticconstNvmeBar_tNvmeWriteMask = {
        .aqa = {
                .rsvd1          =       0xF,
                .rsvd2          =       0xF,
        },
};
```

3.3.9.3 Register Reset Value

```
staticconstNvmeBar_tNvmeBarReset = {
        .aqa = {
                .asqs           =       0x0,
                .rsvd1          =       0x0,
                .acqs           =       0x0,
                .rsvd2          =       0x0,
        },
};
```

3.3.10 Admin Submission Queue Base Address (ASQ)

3.3.10.1 Software implementation

```
typedefstructasq_reg {
        uint16_t        rsvd:12;
        uint64_t        asqb:52;
} __attribute__((packed)) asq_t;
```

3.3.10.2 Register Write Mask

```
staticconstNvmeBar_tNvmeWriteMask = {
```

```
        .asq = {
                .rsvd           =       0xFFF,
        },
};
```

3.3.10.3 Register Reset Value

```
staticconstNvmeBar_tNvmeBarReset = {
        .asq = {
                .rsvd           =       0x0,
                .asqb           =       0x0,
        },
};
```

3.3.11 Admin Completion Queue Base Address (ACQ)

3.3.11.1 Software implementation

```
typedefstructacq_reg {
        uint16_t        rsvd:12;
        uint64_t        acqb:52;
}__attribute__((packed)) acq_t;
```

3.3.11.2 Register Write Mask

```
staticconstNvmeBar_tNvmeWriteMask = {
        .acq = {
                .rsvd           =       0xFFF,
        },
};
```

3.3.11.3 Register Reset Value

```
staticconstNvmeBar_tNvmeBarReset = {
        .acq = {
                .rsvd           =       0x0,
```

```
                .acqb           =       0x0,
        },
};
```

3.3.12 Controller Memory Buffer Location (CMBLOC)

3.3.12.1 Software implementation

```
typedefstructcmbloc_reg {
        uint8_t         bir:3;
        uint16_t        rsvd:9;
        uint32_t        ofst:20;
}__attribute__((packed)) cmbloc_t;
```

3.3.12.2 Register Write Mask

```
staticconstNvmeBar_tNvmeWriteMask = {
        .cmbloc = {
                .rsvd           =       0x1FF,
                .bir            =       0x7,
                .ofst           =       0xFFFFF,
        },
};
```

3.3.12.3 Register Reset Value

```
staticconstNvmeBar_tNvmeBarReset = {
        .cmbloc = {
                .bir            =       0x0,
                .rsvd           =       0x0,
                .ofst           =       0x0,
        },
};
```

3.3.13 Controller Memory Buffer Size (CMBSZ)

3.3.13.1 Software implementation

```
typedefstructcmbsz {
        uint8_t         sqs:1;
        uint8_t         cqs:1;
        uint8_t         lists:1;
        uint8_t         rds:1;
        uint8_t         wds:1;
        uint8_t         rsvd:3;
        uint8_t         szu:4;
        uint32_t        sz:20;
}__attribute__((packed)) cmbsz_t;
```

3.3.13.2 Register Write Mask

```
staticconstNvmeBar_tNvmeWriteMask = {
        .cmbsz = {
                .rsvd           =       0x7,
                .sqs            =       0x1,
                .cqs            =       0x1,
                .lists          =       0x1,
                .rds            =       0x1,
                .wds            =       0x1,
                .szu            =       0xF,
                .sz             =       0xFFFFF,
        },
};
```

3.3.13.3 Register Reset Value

```
staticconstNvmeBar_tNvmeBarReset = {
        .cmbsz = {
                .sqs            =       0x0,
                .cqs            =       0x0,
```

```
                .lists          =       0x0,
                .rds            =       0x0,
                .wds            =       0x0,
                .rsvd           =       0x0,
                .szu            =       0x0,
                .sz             =       0x0,
        },
};
```

3.4 General SQ and CQ Entry Structure

The general structure of SQs and CQs as specified with the common fields are explained below. There are some commands/responses where some specific fields are added or modified. Such customized fields are explained in the individual command and response sections.

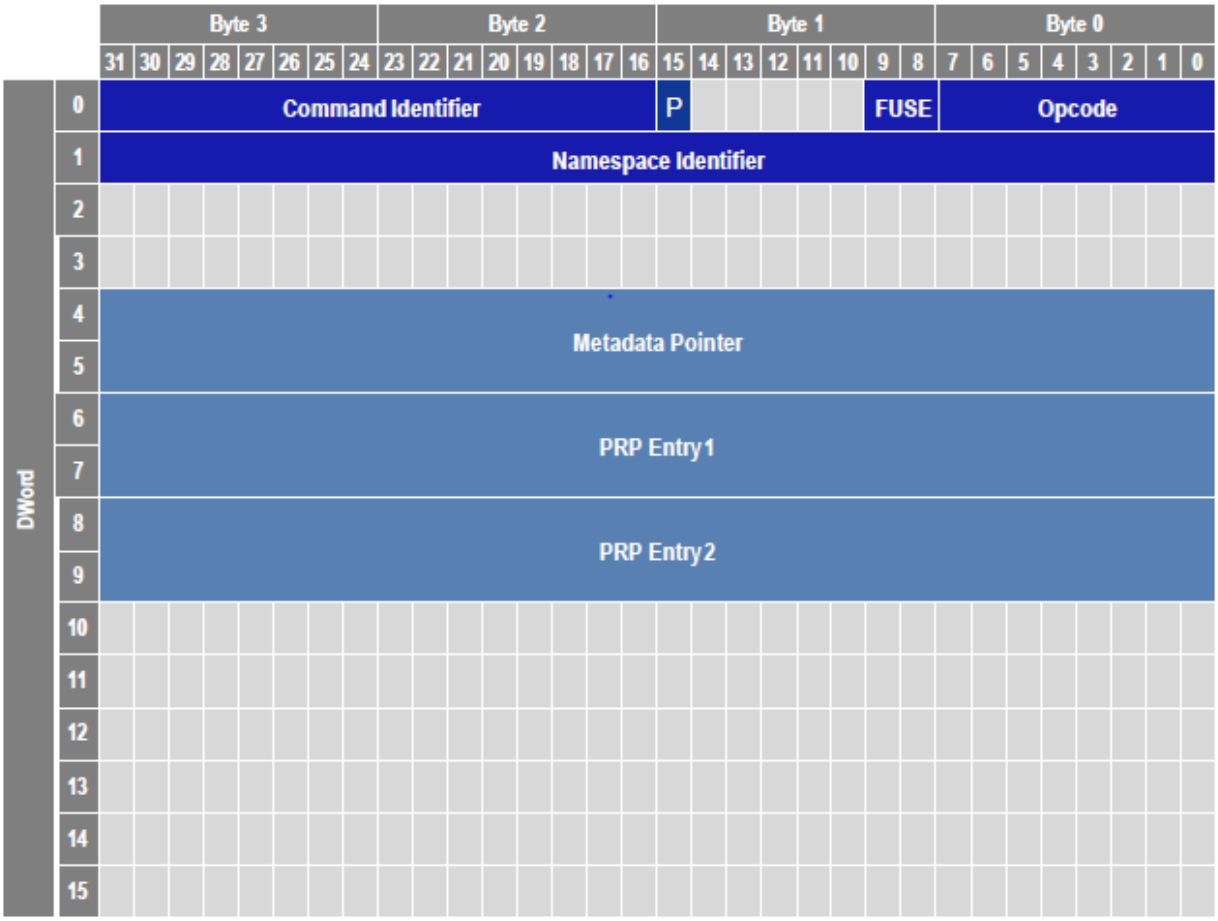


Table 5: General SQ entry format

- **Opcode** – Command operation code
- **Fused Operation (FUSE)** – specifies if two commands should be executed as atomic unit (optional)
- **Command Identifier** – Command ID within submission queue
- **Namespace** – Namespace on which command operates
- **PRP Entry 1** – First PRP entry for the command or PRP list pointer depending on the command.
- **PRP Entry 2**– Second PRP entry for the command or PRP list pointer depending on the command.
- **Metadata pointer**– Pointer to contiguous buffer containing metadata.

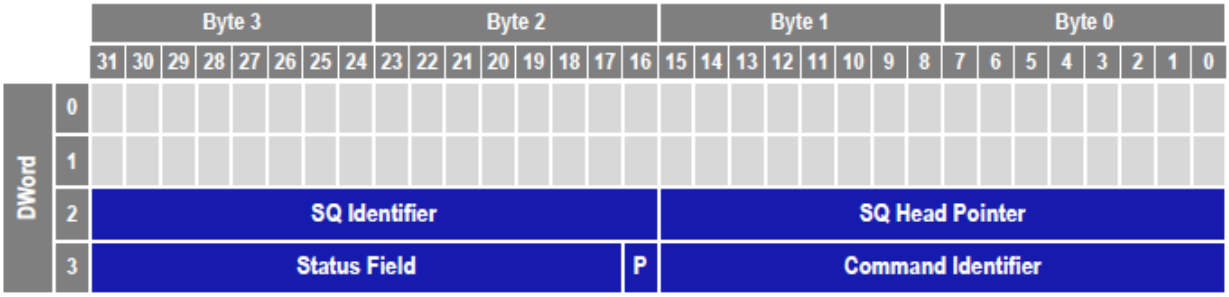


Table 6: General CQ entry format

- **SQ Identifier** – Submission queue associated with completed command
- **Command Identifier** – Command ID within submission queue
- **Phase Tag (P)** – Toggled whenever complete CQ list is traversed and new CQ entry starts from beginning of the list.
- **Status Field** – Status associated with completed command
  A value of zero indicates successful command completion

3.5 NVMe Admin Commands

Admin Command	Opcode	Category
Delete I/O Submission Queue	00h	Queue Management
Create I/O Submission Queue	01h	
Delete I/O Completion Queue	04h	
Create I/O Completion Queue	05h	
Identify	06h	Configuration
Set Features	09h	
Get Features	0Ah	
Get Log Page	02h	Status Reporting
Asynchronous Event Request	0Ch	
Abort	08h	Abort Command

Table 7: Admin Command Set

3.5.1 Software Implementation

```
struct nvme_command {
```

```
        union {
                structnvme_common_command  common;
                structnvme_rw_command      rw;
                structnvme_identify        identify;
                structnvme_features        features;
                structnvme_create_cq       create_cq;
                structnvme_create_sq       create_sq;
                structnvme_delete_queue    delete_queue;
                structnvme_abort_cmd       abort;
        };
};

enumnvme_admin_opcode {
        nvme_admin_delete_sq        =    0x00,
        nvme_admin_create_sq        =    0x01,
        nvme_admin_get_log_page     =    0x02,
        nvme_admin_delete_cq        =    0x04,
        nvme_admin_create_cq        =    0x05,
        nvme_admin_identify         =    0x06,
        nvme_admin_abort_cmd        =    0x08,
        nvme_admin_set_features     =    0x09,
        nvme_admin_get_features     =    0x0a,
        nvme_admin_async_event      =    0x0c,
};
```

3.5.2 Create I/O Submission Queue Command

The Create I/O Submission Queue command is used to create I/O Submission Queues.

3.5.2.1 Software Implementation Structure

```
structnvme_create_sq {
        uint8_t         opcode;
        uint8_t         flags;
        uint16_t        command_id;
```

```
        uint32_t        rsvd1[5];
        uint64_t        prp1;
        uint64_t        rsvd8;
        uint16_t        sqid;
        uint16_t        qsize;
        uint16_t        sq_flags;
        uint16_t        cqid;
        uint32_t        rsvd12[4];
};
```

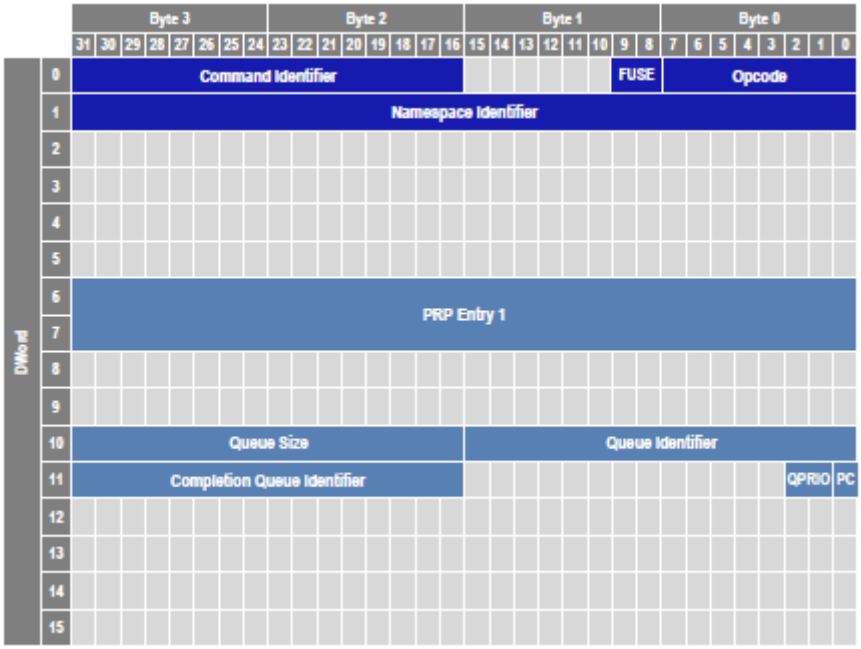


Table 8 : Create IO Submission Queue Command Format

- **Queue Identifier** – Submission queue ID number
- **Queue Size** – Number of entries in submission queue (zero based value)
- **Completion Queue Identifier** – Completion queue ID number associated with submission queue
- **Queue Priority (QPRIO)** – Queue Priority when WRR with urgent priority service class priority is selected
- **Physically Contiguous (PC)**
  - 1–Submission queue is physically contiguous in host memory
  - 0 – Submission queue is not physically contiguous
- **PRP Entry 1** – When not physically contiguous, this is a pointer to a PRP list that contains host pages

- **Command Specific Error Values**
  - Completion Queue Invalid
  - Invalid Queue Identifier
  - Maximum Queue Size Exceeded

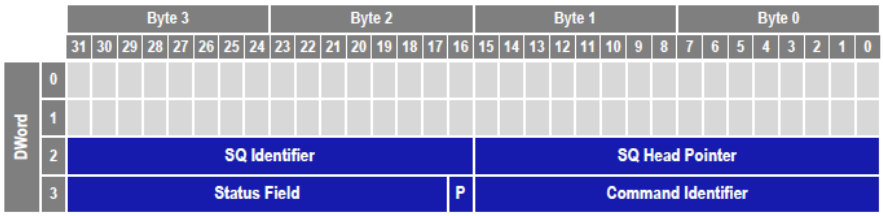


Table 9: Create IO Submission Queue Response Format

3.5.2.2 Processing the Command

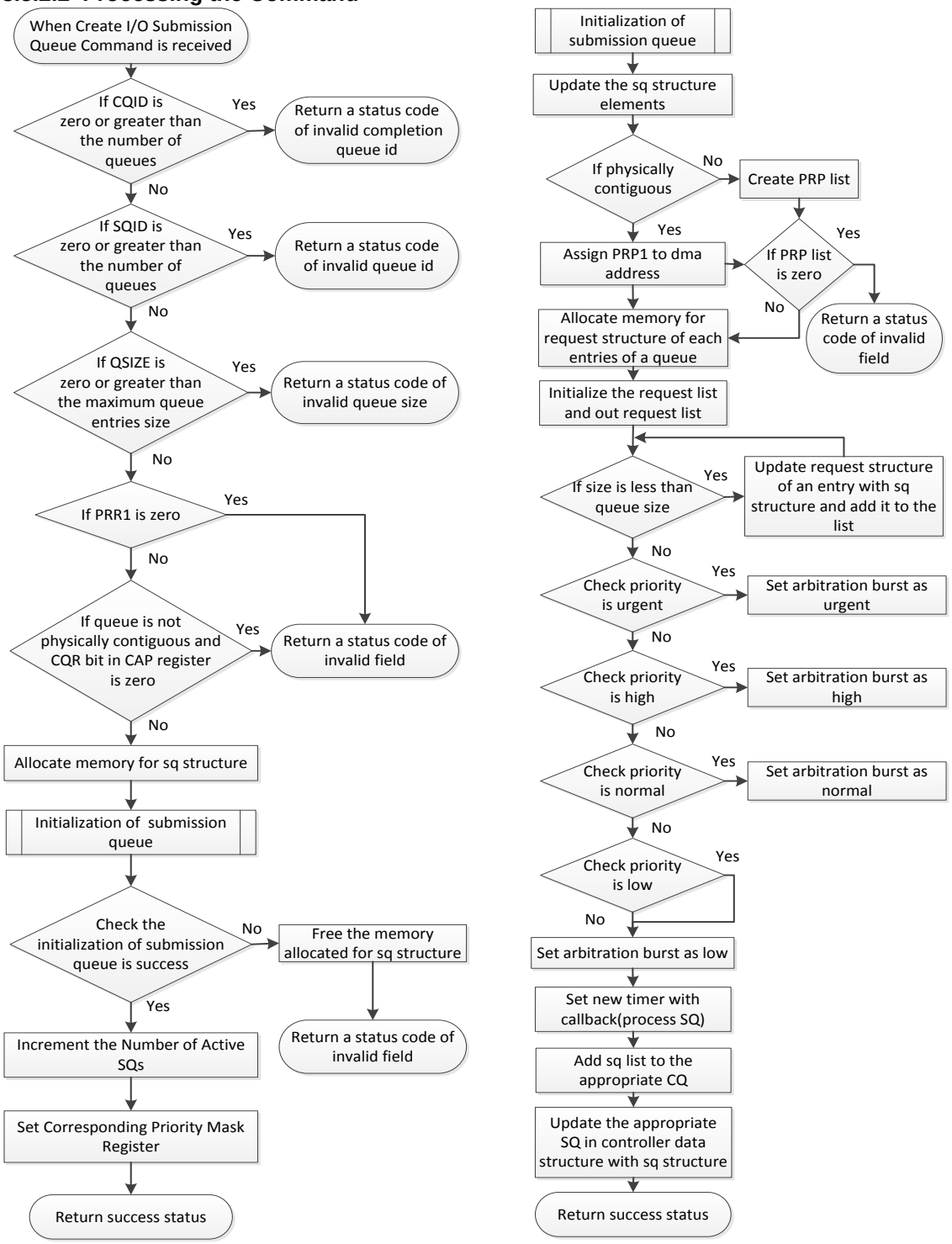


Figure 32: Handling Create IO Submission Queue Flow Chart

3.5.3 Create I/O Completion Queue Command

The Create I/O Completion Queue command is used to create all I/O Completion Queues with the exception of the Admin Completion Queue

3.5.3.1 Software Implementation Structure

```
structnvme_create_cq {
        uint8_t         opcode;
        uint8_t         flags;
        uint16_t        command_id;
        uint32_t        rsvd1[5];
        uint64_t        prp1;
        uint64_t        rsvd8;
        uint16_t        cqid;
        uint16_t        qsize;
        uint16_t        cq_flags;
        uint16_t        irq_vector;
        uint32_t        rsvd12[4];
};
```

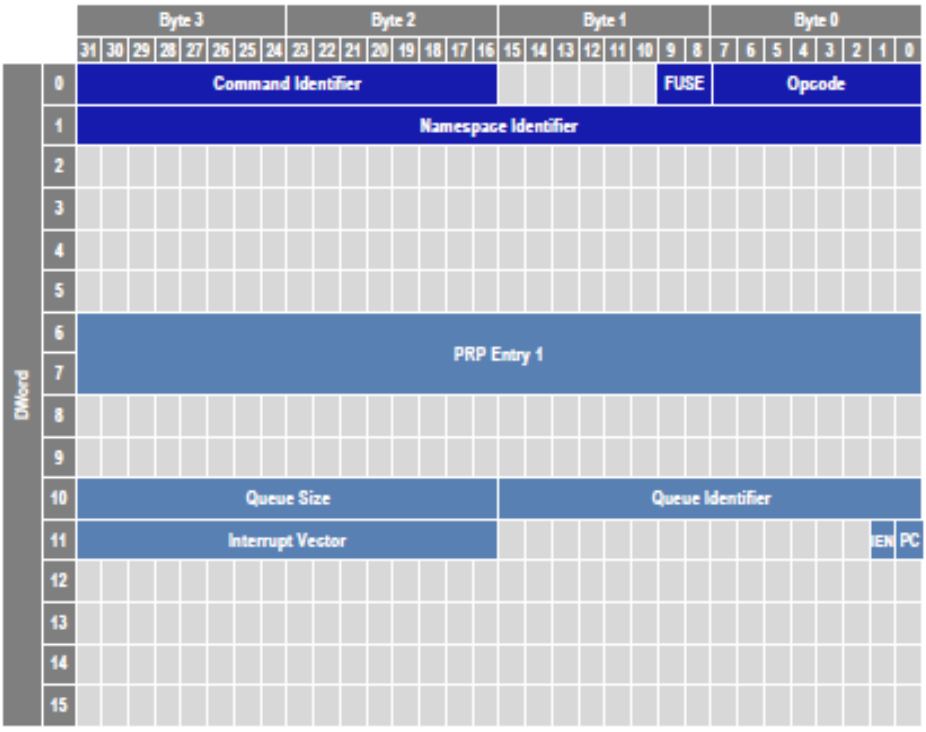


Table 10: Create IO Completion Queue Command Format

- **Queue Identifier** – completion queue ID number
- **Queue Size** – Number of entries in completion queue (zero based value)
- **Interrupt Vector** – MSI-X or MSI vector number
- **Interrupt Enable (IEN)**
    - 0 – Interrupts disabled
    - 1 – Interrupts enabled
- **Physically Contiguous (PC)**
    - 1 – Submission queue is physically contiguous in host memory
    - 0 – Submission queue is not physically contiguous
- **PRP Entry 1** – When not physically contiguous, this is a pointer to a PRP list that contains host pages

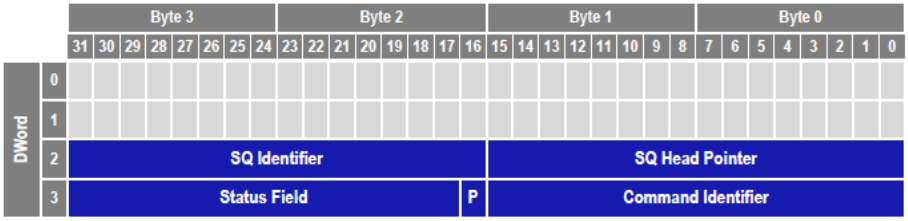


Table 11: Create IO Completion Queue Response Format

3.5.3.2 Processing the Command

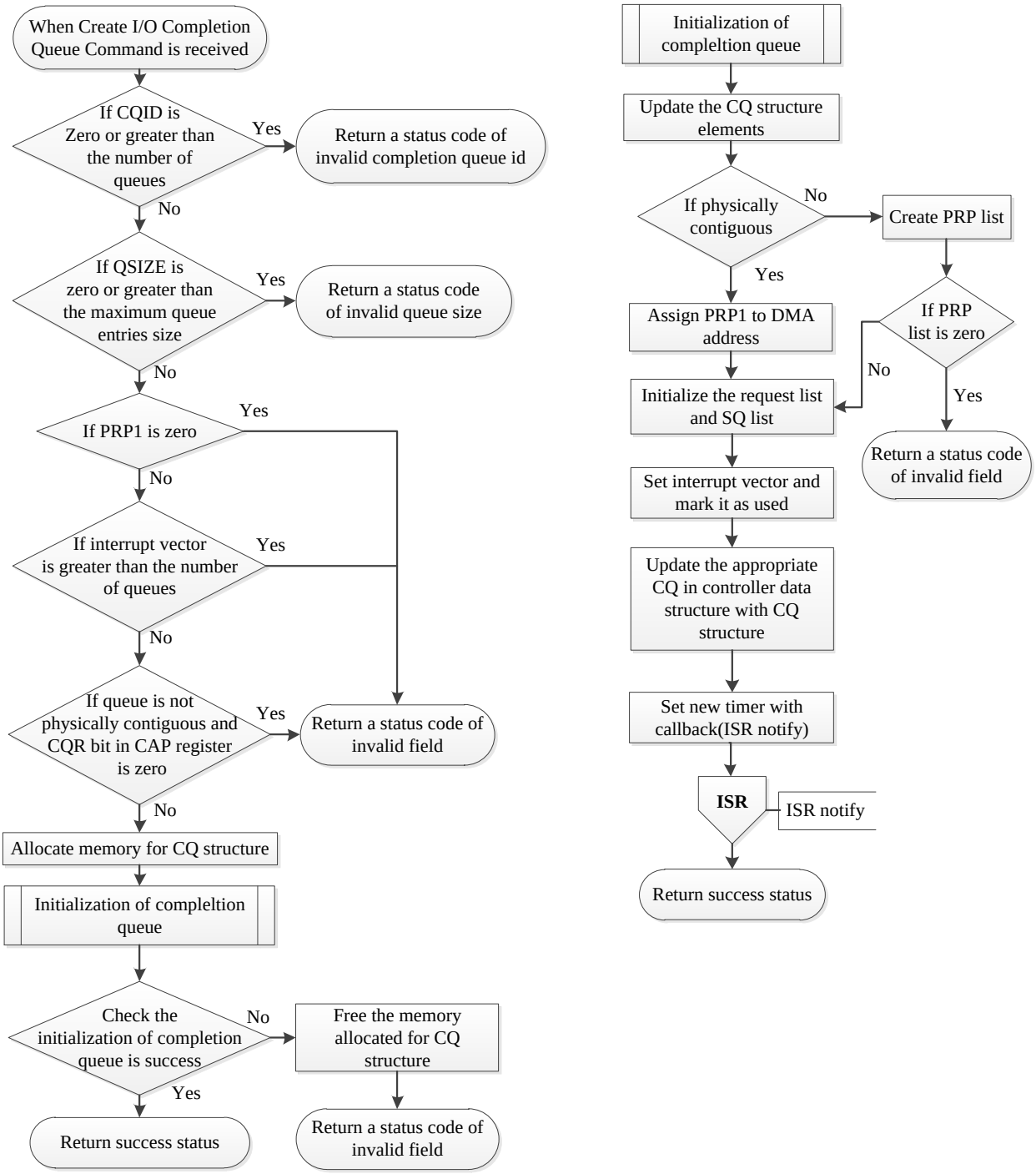


Figure 33: Handling Create IO Completion Queue Command Flow Chart

3.5.4 Asynchronous Event Request Command

Asynchronous events are used to notify host software of status, error, and health information as these events occur.

3.5.4.1 Software Implementation Structure:

```
structNvmeAerResult {
        uint8_t         event_type;
        uint8_t         event_info;
        uint8_t         log_page;
        uint8_t         resv;
};
```

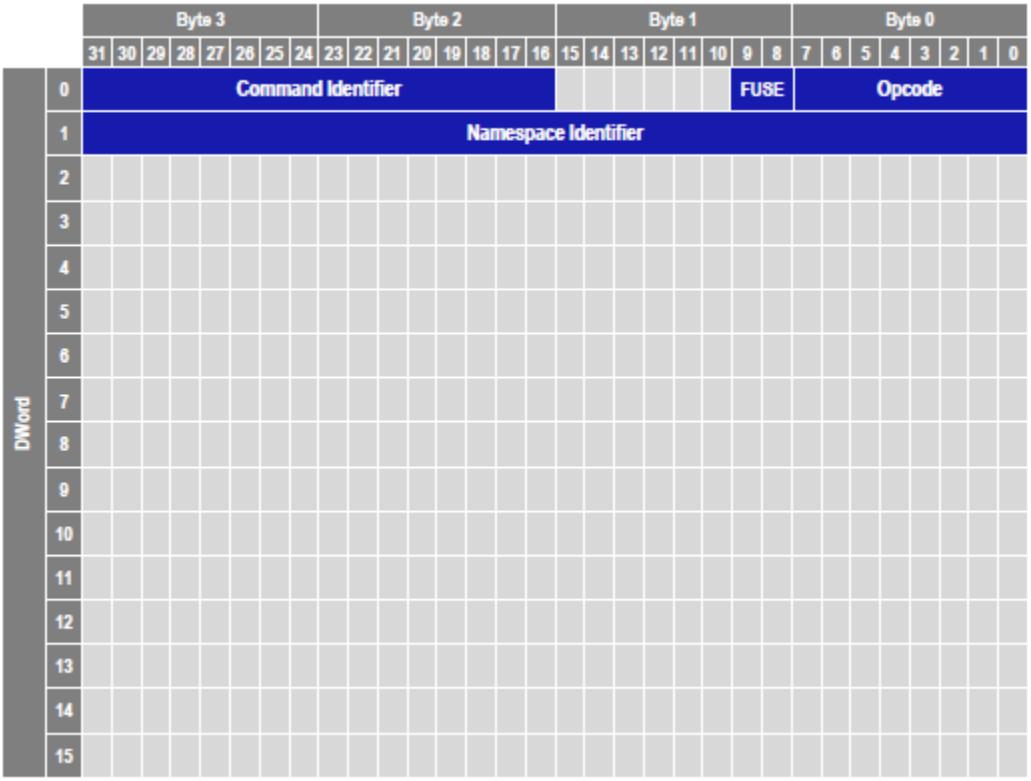


Table 12: Asynchronous Event Request Command Format

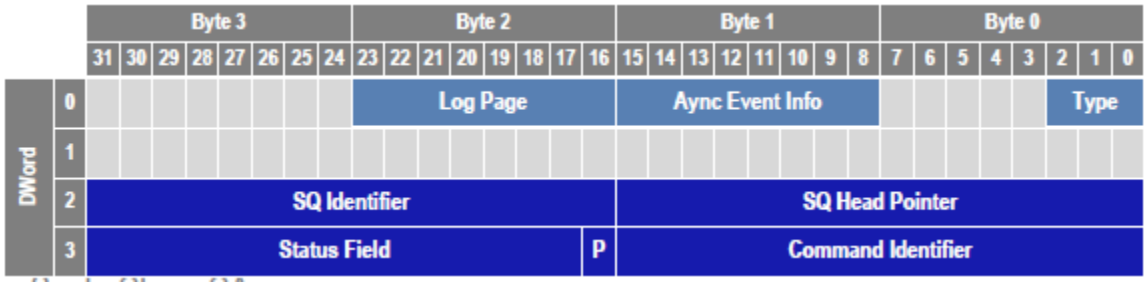


Table 13: Asynchronous Event Request Response format

- **Type** – Type of asynchronous event

  Error Status

  SMART / Health status

  Vendor Specific
- **Async Event Info** – Provides error type specific details

  Examples:

  Temperature above threshold

  Invalid doorbell value write

  Spare space below threshold
- **Log Page** – ID of log page to retrieve more information and clear mask (using Get Log Page)

3.5.4.2 Processing the Command

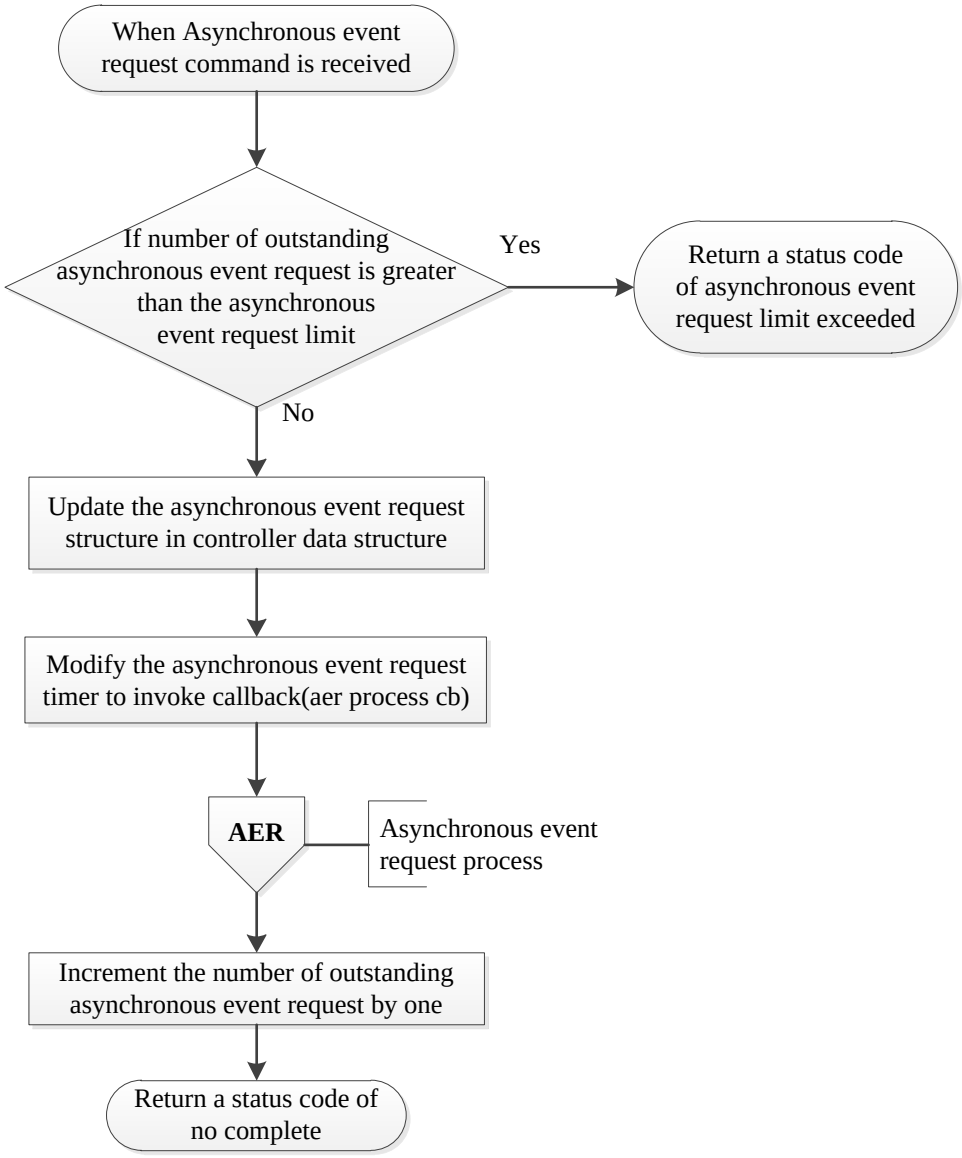


Figure 34: Handling Asynchronous Event Request Command Flow Chart

3.5.5 Delete I/O Completion Queue Command

The Delete I/O Completion Queue command is used to delete an I/O Completion Queue.

3.5.5.1 Software Implementation Structure

```
structnvme_delete_queue {
        uint8_t         opcode;
        uint8_t         flags;
        uint16_t        command_id;
        uint32_t        rsvd1[9];
        uint16_t        qid;
        uint16_t        rsvd10;
        uint32_t        rsvd11[5];
};
```

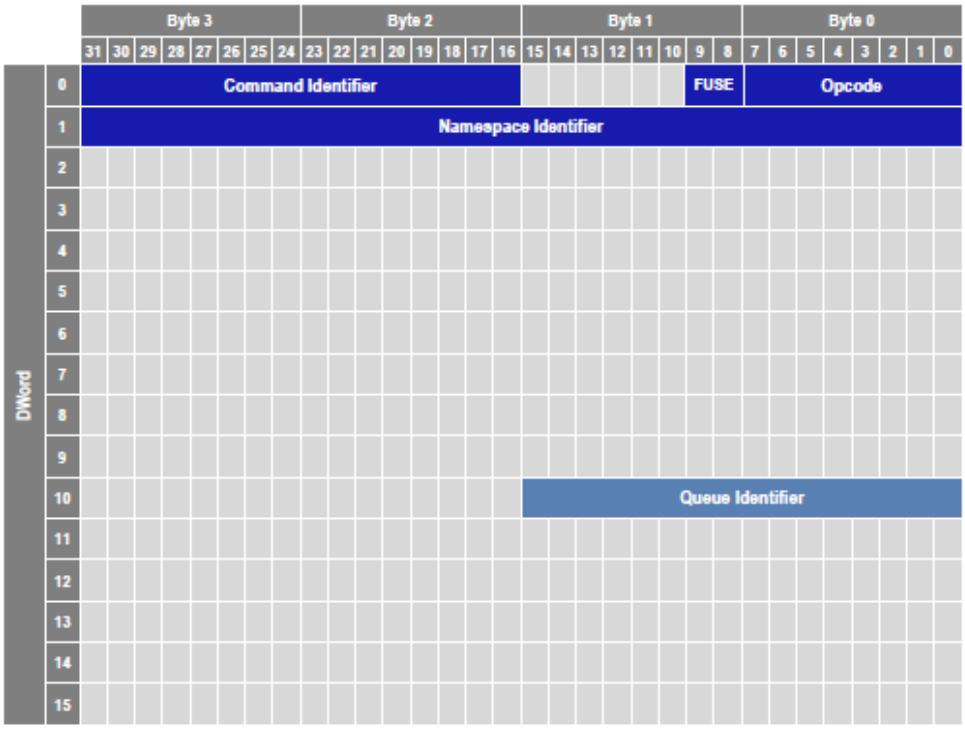


Table 14: Delete IO Completion Queue Command Format

- **Queue Identifier** – Completion queue ID number

	Byte 3								Byte 2								Byte 1								Byte 0							
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DWord 0																																
DWord 1																																
DWord 2	SQ Identifier																SQ Head Pointer															
DWord 3	Status Field															P	Command Identifier															

Table 15: Delete IO Completion Queue Response Format

3.5.5.2 Processing the Command

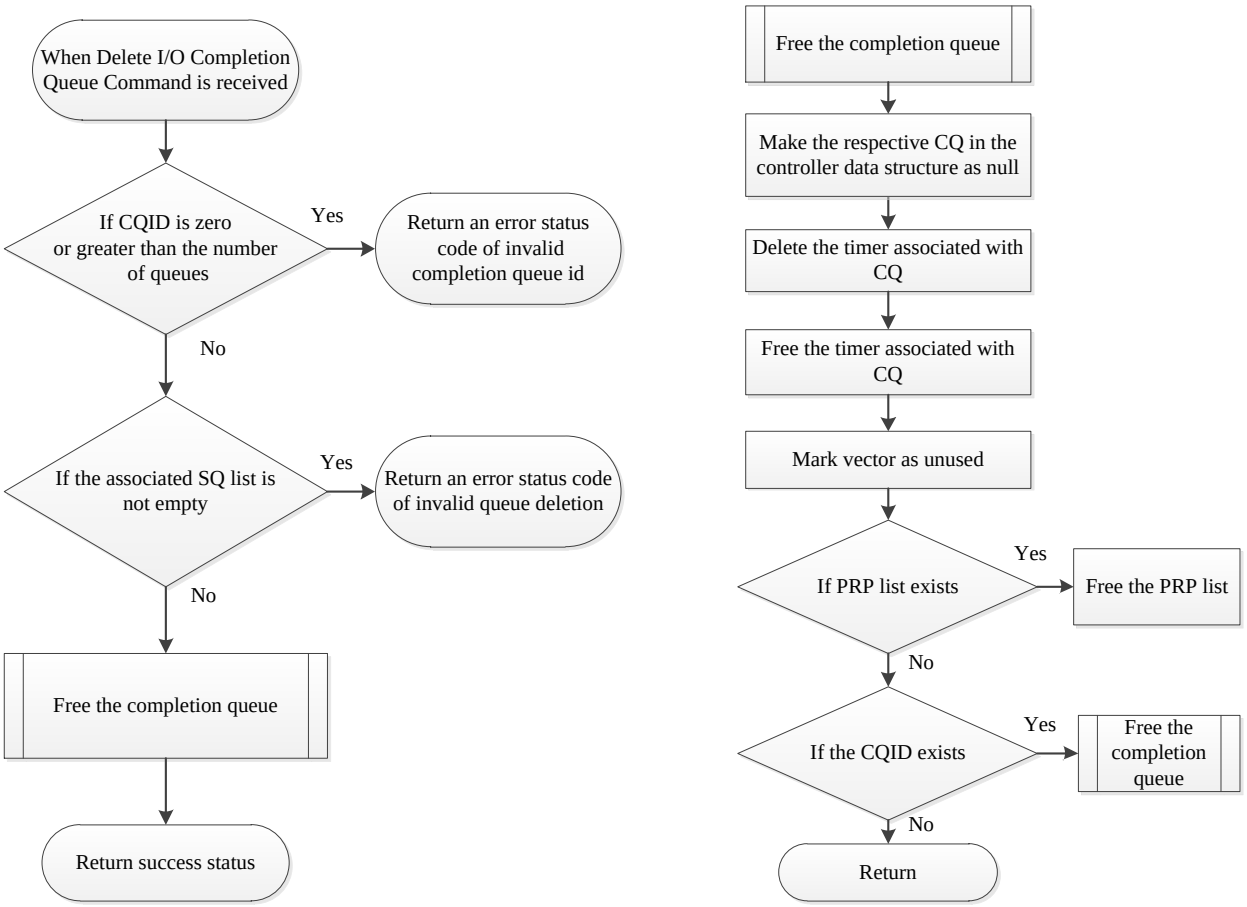


Figure 35:Handling Delete IO Completion Queue Command Flow Chart

3.5.6 Delete I/O Submission Queue Command

The Delete I/O Submission Queue command is used to delete an I/O submission Queue

3.5.6.1 Software Implementation Structure

```
structnvme_delete_queue {
        uint8_t         opcode;
        uint8_t         flags;
        uint16_t        command_id;
        uint32_t        rsvd1[9];
        uint16_t        qid;
        uint16_t        rsvd10;
        uint32_t        rsvd11[5];
};
```

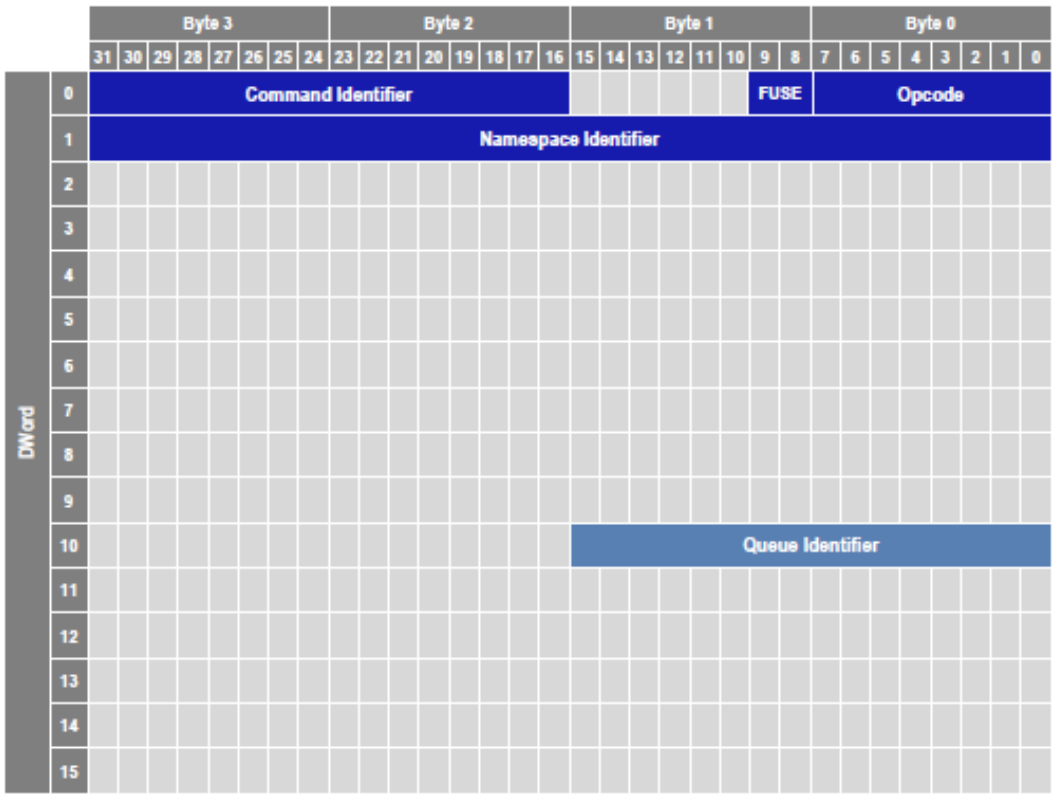


Table 16: Delete IO Submission Queue Command Format

- **Queue Identifier** – Submission queue ID number
- **Command Specific Error Values**
    - Invalid Queue Identifier

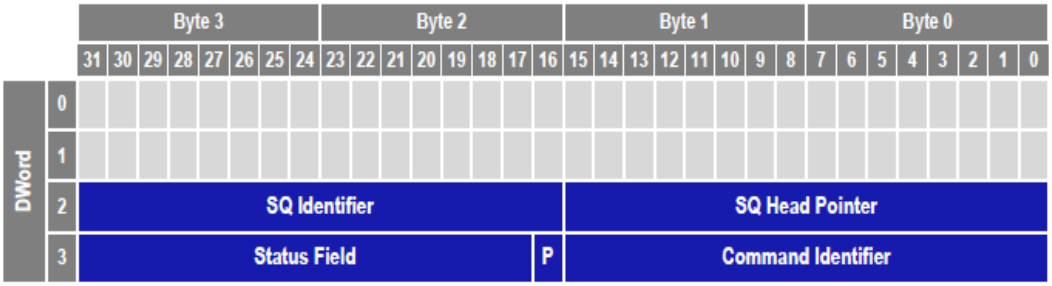


Table 17: Delete IO Submission Queue Response Format

3.5.6.2 Processing the Command

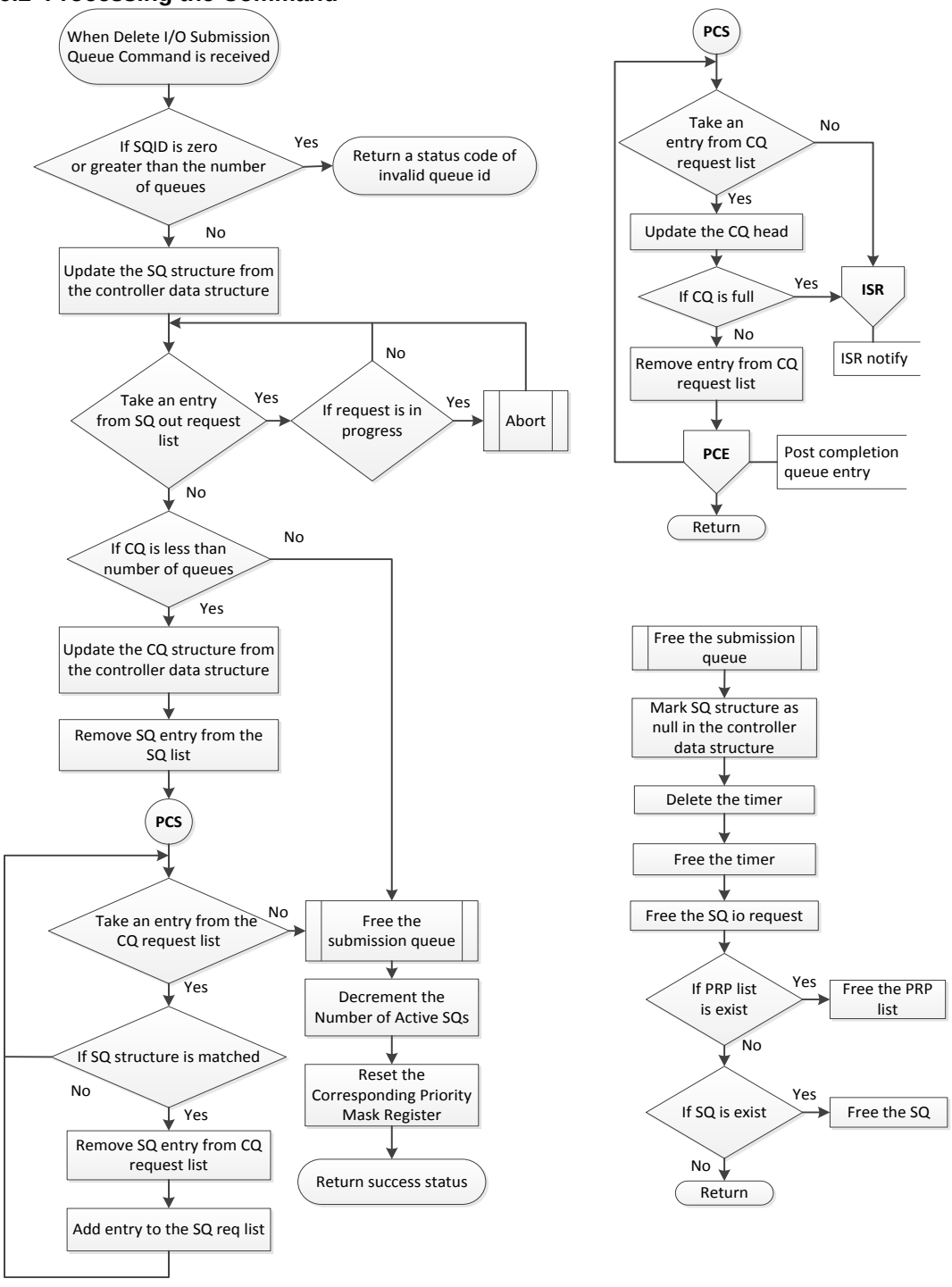


Figure 36:Handling Delete IO Submission Queue Command Flow Chart

3.5.7 Get Log Page Command

The Get Log Page command returns a data buffer containing the log page requested.

3.5.7.1 Software Implementation Structure

```
struct nvme_smart_log {
        uint8_t         critical_warning;
        uint8_t         temperature[2];
        uint8_t         avail_spare;
        uint8_t         spare_thresh;
        uint8_t         percent_used;
        uint8_t         rsvd6[26];
        uint8_t         data_units_read[16];
        uint8_t         data_units_written[16];
        uint8_t         host_reads[16];
        uint8_t         host_writes[16];
        uint8_t         ctrl_busy_time[16];
        uint8_t         power_cycles[16];
        uint8_t         power_on_hours[16];
        uint8_t         unsafe_shutdowns[16];
        uint8_t         media_errors[16];
        uint8_t         num_err_log_entries[16];
        uint32_t        warning_temp_time;
        uint32_t        critical_comp_time;
        uint16_t        temp_sensor[8];
        uint8_t         rsvd216[296];
};
```

```
struct Nvme ErrorLog {
        uint64_t        error_count;
        uint16_t        sqid;
        uint16_t        cid;
        uint16_t        status_field;
```

```
        uint16_t        param_error_location;
        uint64_t        lba;
        uint32_t        nsid;
        uint8_t          vs;
        uint8_t         resv[35];
};
```

```
struct Nvme FwSlotInfoLog {
        uint8_t         afi;
        uint8_t         reserved1[7];
        uint8_t         frs1[8];
        uint8_t         frs2[8];
        uint8_t         frs3[8];
        uint8_t         frs4[8];
        uint8_t         frs5[8];
        uint8_t         frs6[8];
        uint8_t         frs7[8];
        uint8_t         reserved2[448];
};
```

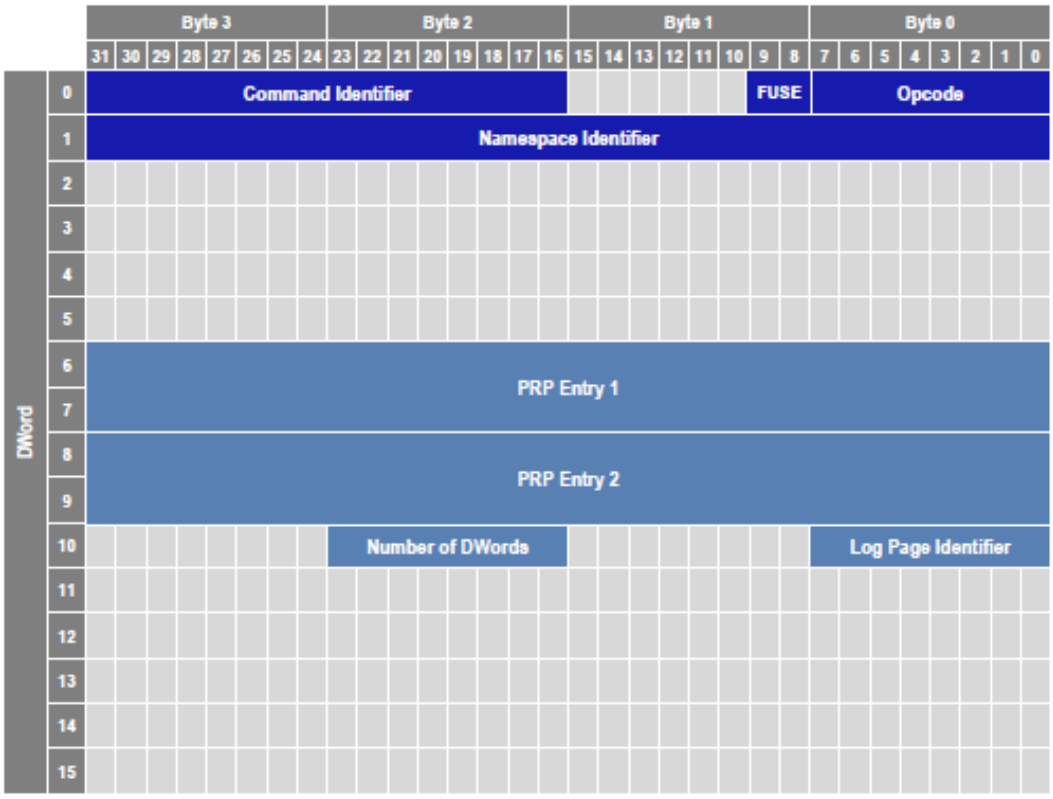


Table 18: Get Log Page Command Format

- **PRP Entry 1** – Starting address of where log page should be written
- **PRP Entry 2** – Starting address of where remainder of remainder of log page should be written
- **Number of DWords –** Number of DWords to transfer
- **Log Page Identifier –** ID of log page to retrieve

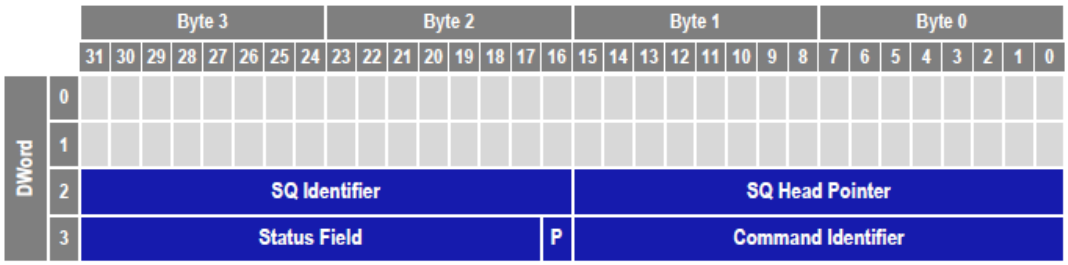


Table 19: Get log page Response Format

3.5.7.2 Processing the Command

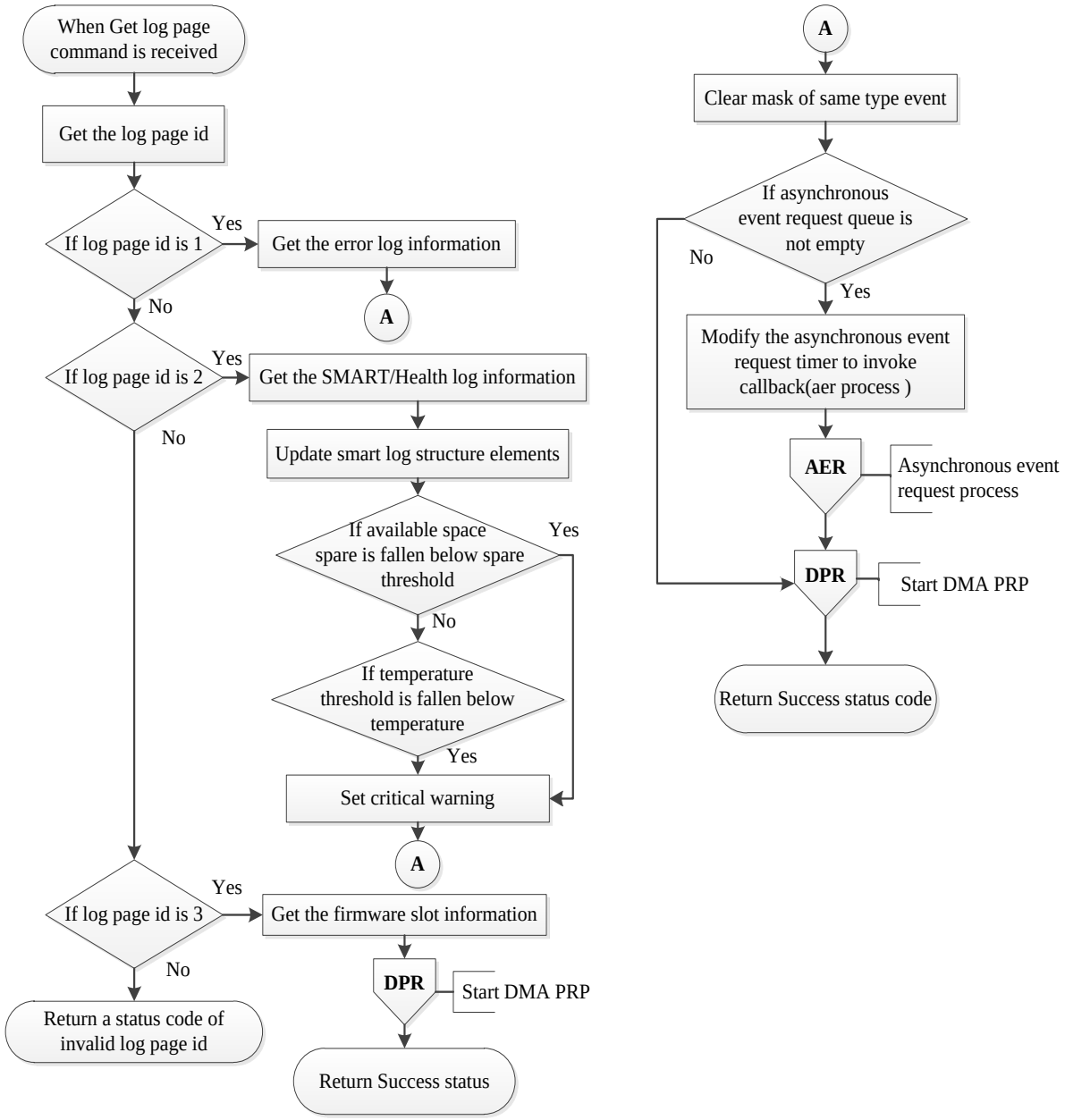


Figure 37: Handling Get Log Page Command Flow Chart

3.5.8 Get Features Command

The Get Features command retrieves the attributes of the Feature specified.

3.5.8.1 Software Implementation Structure

```
structnvme_features {
        uint8_t         opcode;
        uint8_t         flags;
        uint16_t        command_id;
        uint32_t        nsid;
        uint64_t        rsvd2[2];
        uint64_t        prp1;
        uint64_t        prp2;
        uint32_t        fid;
        uint32_t        dword11;
        uint32_t        rsvd12[4];
};
```

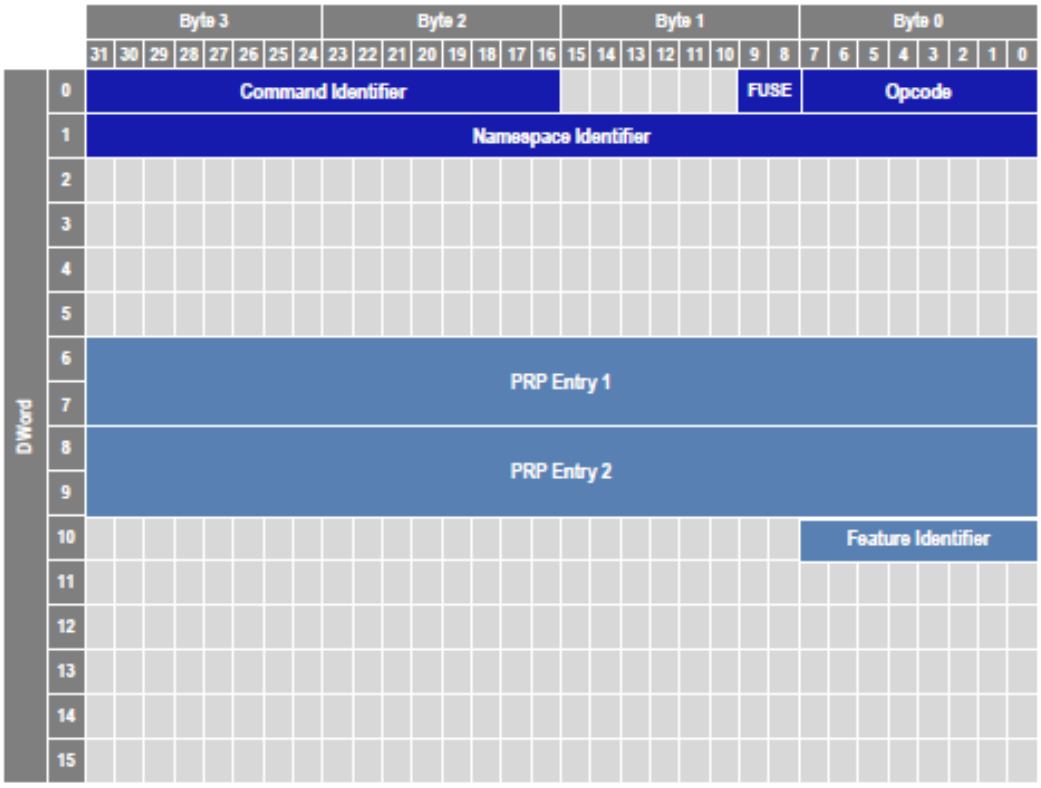


Table 20: Get Features Command Format

- **PRP Entry 1** – Starting address of where feature data should be written (used by some features)
- **PRP Entry 2** – Starting address of where remainder of where feature data should be written (used by some features)
- **Feature Identifier –** ID of feature
  Feature value returned in memory (PRPs) or in DWord 0 (Result) of completion entry

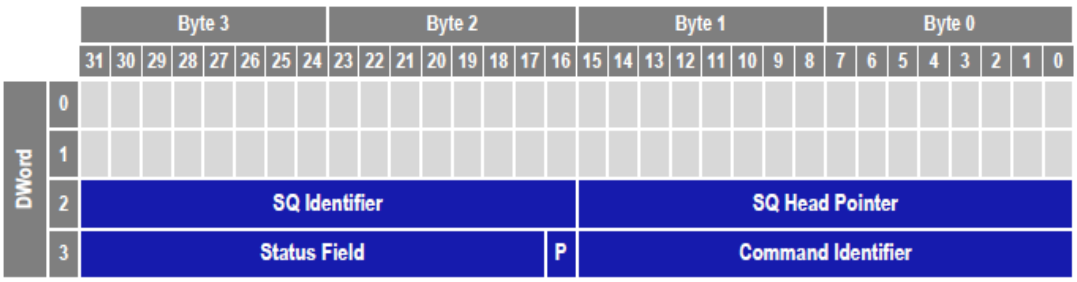


Table 21: Get Features Response Format

3.5.8.2 Processing the Command

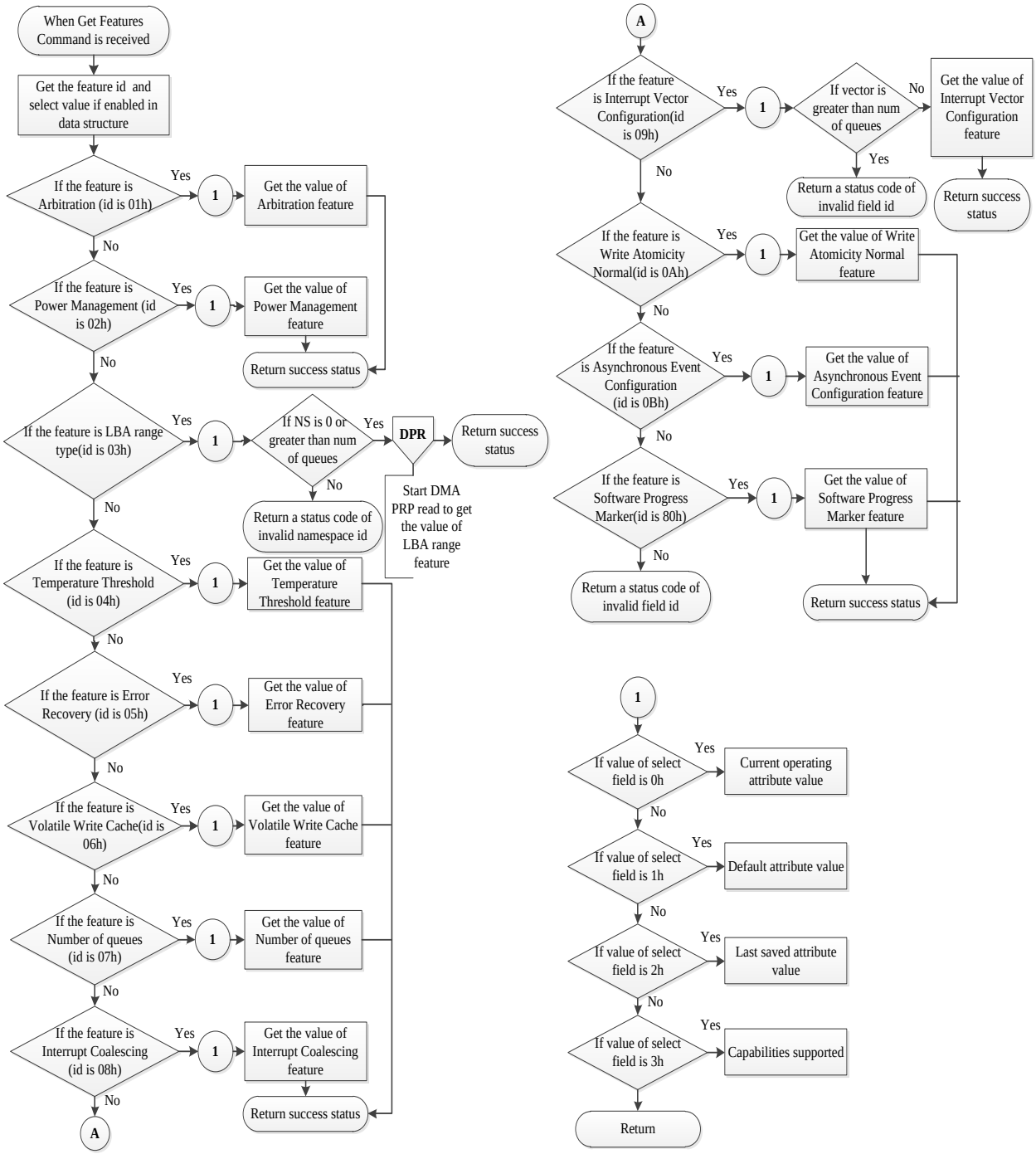


Figure 38: Handling Get Features Command Flow Chart

3.5.9 Set Feature Command

The Set Features command specifies the attributes of the Feature indicated

3.5.9.1 Software Implementation Structure

```
struct nvme_features {
        uint8_t         opcode;
        uint8_t         flags;
        uint16_t        command_id;
        uint32_t        nsid;
        uint64_t        rsvd2[2];
        uint64_t        prp1;
        uint64_t        prp2;
        uint32_t        fid;
        uint32_t        dword11;
        uint32_t        rsvd12[4];
};
```

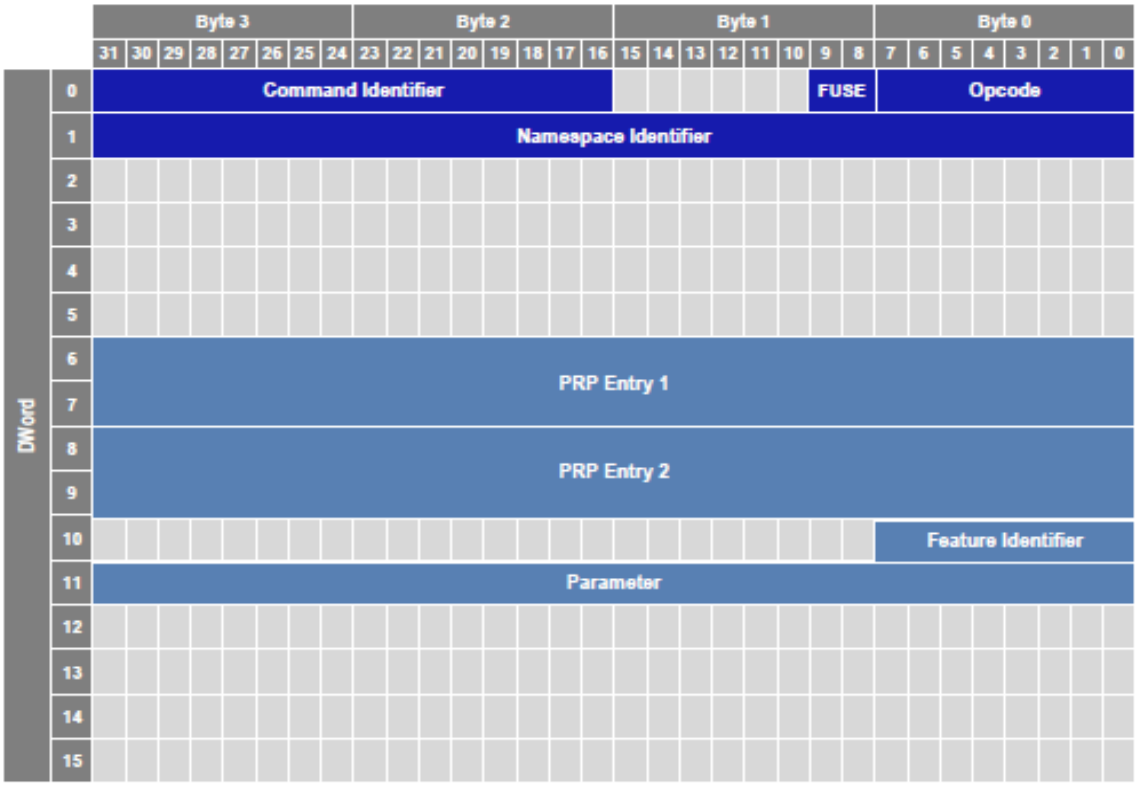


Table 22: Set Features Command Format

- **PRP Entry 1** – Starting address of where Feature data is located (used by some features)
- **PRP Entry 2** – Starting address of where remainder of where feature data is located (used by some features)
- **Parameter – Feature parameter** (used by some features**)**
- **Feature Identifier – ID of feature**

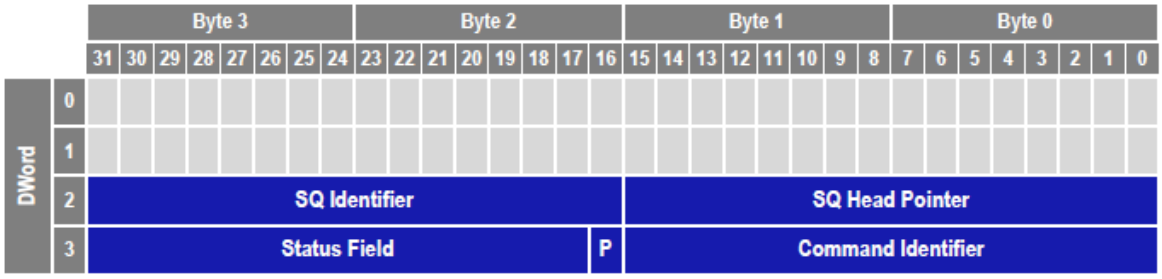


Table 23: Set features Response Format

3.5.9.2 Processing the Command

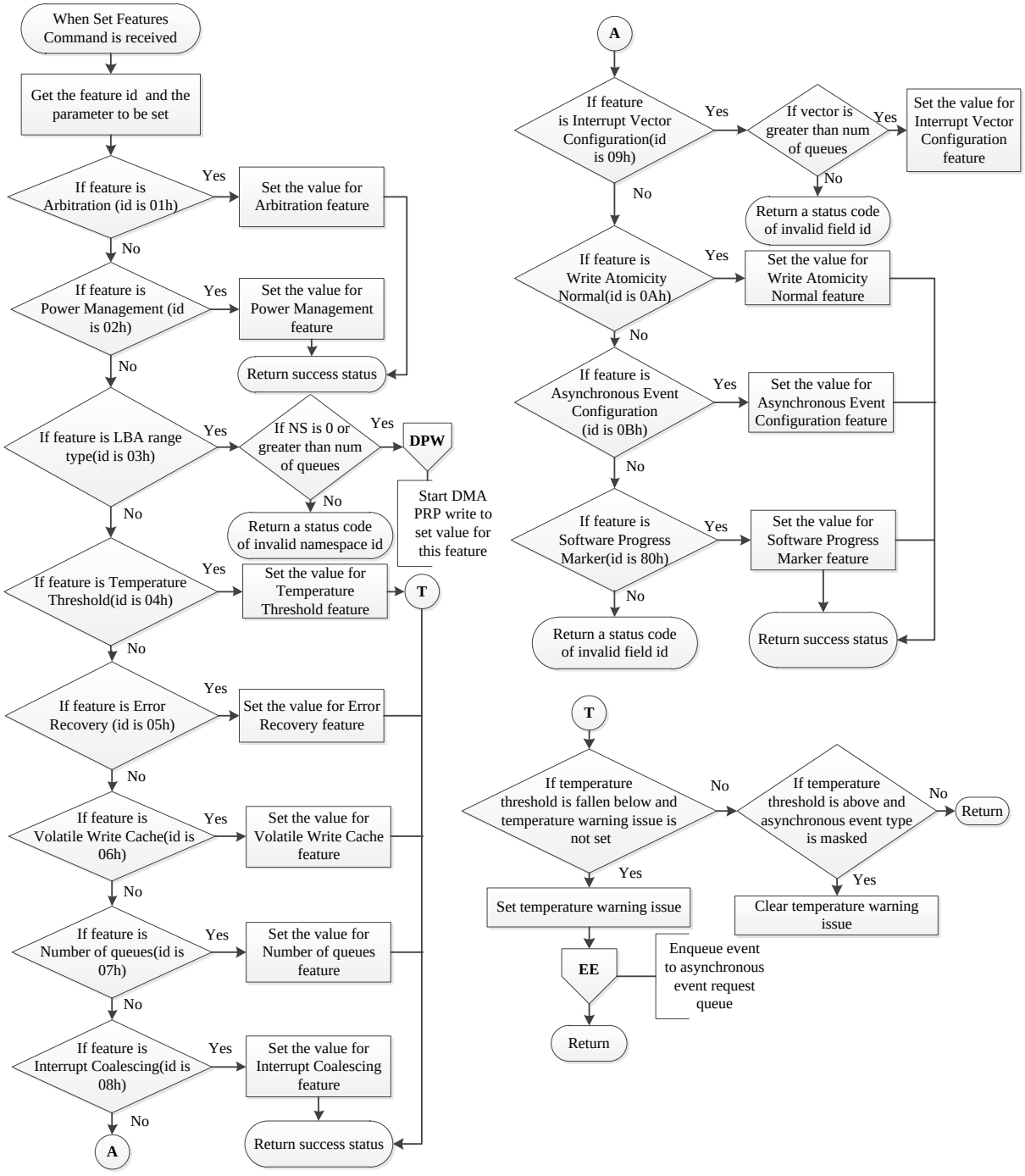


Figure 39: Handling Set Features Command Flow Chart

3.5.10 Identify Command

The Identify command returns a data buffer that describes information about the NVM subsystem, the controller or the namespace(s).

3.5.10.1 Software Implementation Structure

```
struct nvme_identify {
        uint8_t         opcode;
        uint8_t         flags;
        uint16_t        command_id;
        uint32_t        nsid;
        uint64_t        rsvd2[2];
        uint64_t        prp1;
        uint64_t        prp2;
        uint32_t        cns;
        uint32_t        rsvd11[5];
};
```

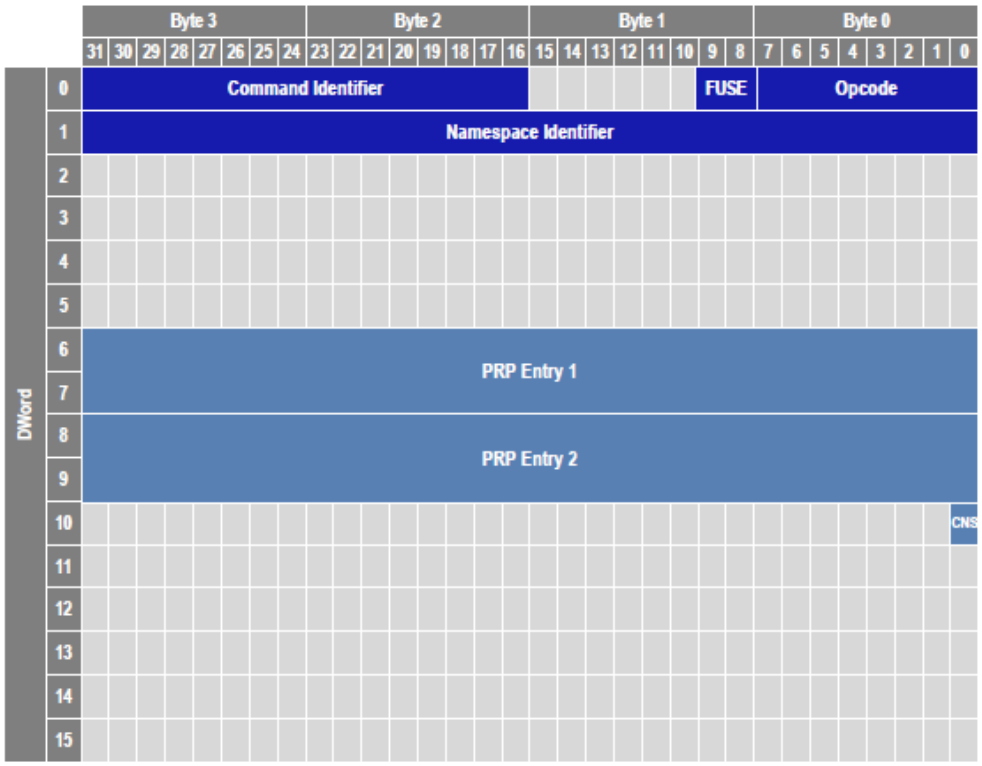


Table 24: Identify Command Format

- **PRP Entry 1** – Starting address of where 4KB data structure is to be written.
  - Offset may be non-zero
- **PRP Entry 2** – Starting address of where remainder of 4KB data structure is to be written
- **Controller or Namespace Structure (CNS)**
  - 00b – Return corresponding namespace data structure
  - 01b – Return corresponding controller data structure
  - 10b – Return list of 1024 active namespace IDs starting at the Namespace Identifer

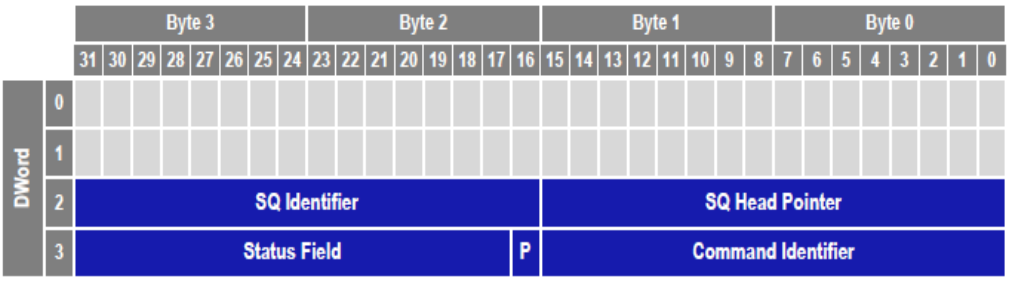


Table 25: Identify Response Format

3.5.10.2 Processing the Command

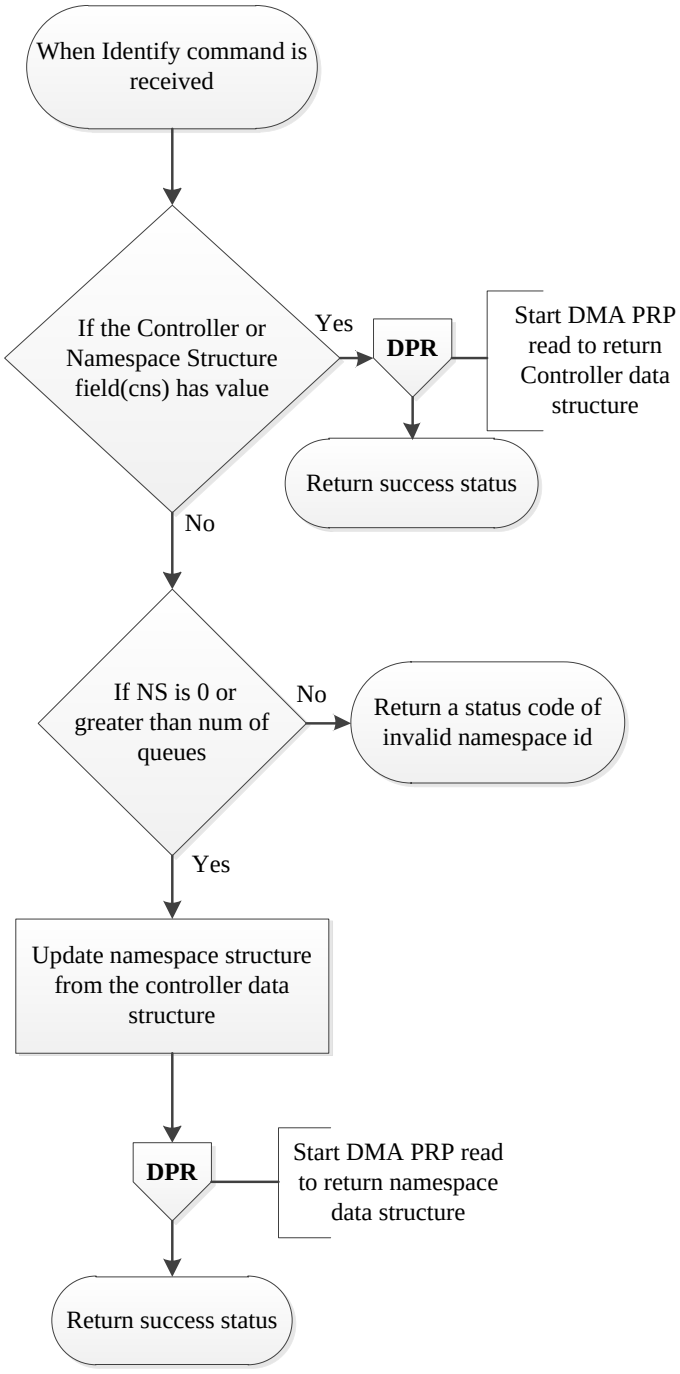


Figure 40: Handling Identify Command Flow Chart

3.5.11 Abort command

The Abort command is used to abort a specific command previously submitted to the Admin Submission Queue or an I/O Submission Queue

		Byte 3								Byte 2								Byte 1								Byte 0							
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DWord	0	Command Identifier																						FUSE		Opcode							
	1	Namespace Identifier																															
	2																																
	3																																
	4																																
	5																																
	6																																
	7																																
	8																																
	9																																
	10	Command Identifier																Submission Queue ID															
	11																																
	12																																
	13																																
	14																																
	15																																

Table 26: Abort Command Format

- **Submission Queue ID**– ID of submission queue on which command was issued
- **Command Identifier**– ID of the command to abort

		Byte 3								Byte 2								Byte 1								Byte 0							
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DWord	0																																A
	1																																
	2	SQ Identifier																SQ Head Pointer															
	3	Status Field															P	Command Identifier															

Table 27: Abort Response Format

- **A**– Abort status
  - **0** – Command was aborted
  - **1** – Command was not aborted

3.5.11.1 Processing the Command

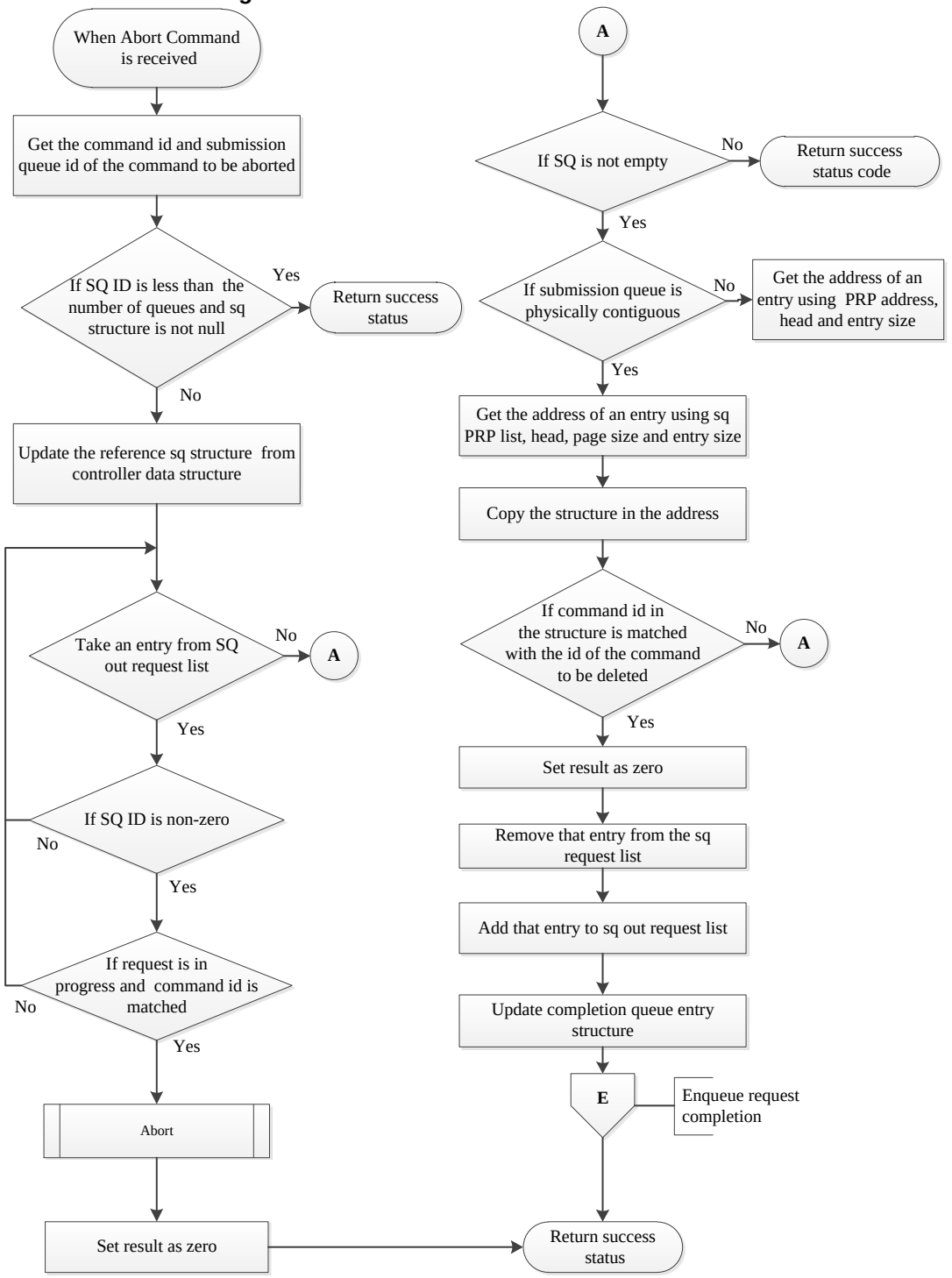


Figure 41: Handling Abort Command Flow Chart

3.5.12 Sub Functions for processing Commands

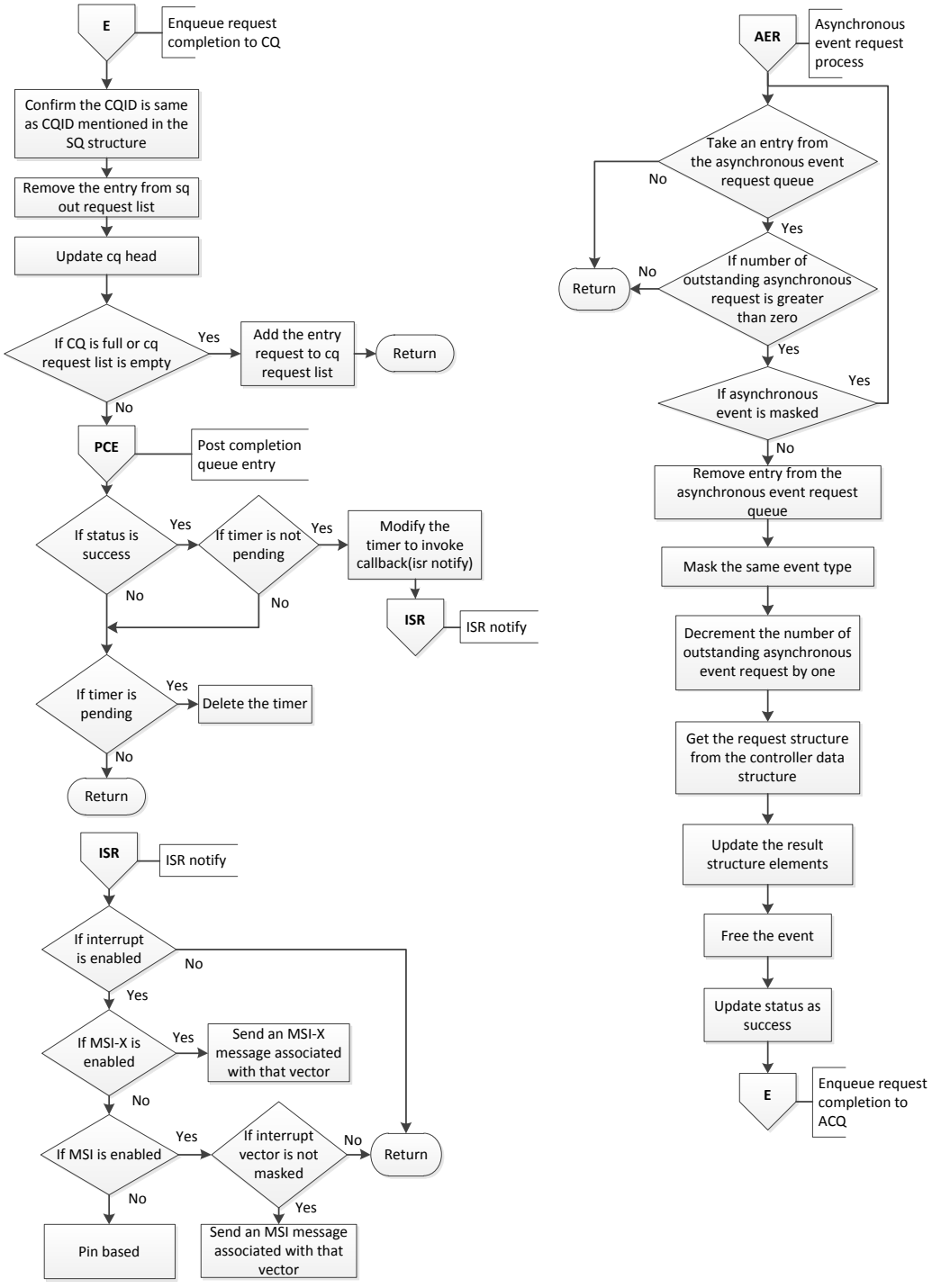


Figure 42: Enqueue request completion to CQ and Asynchronous event request process

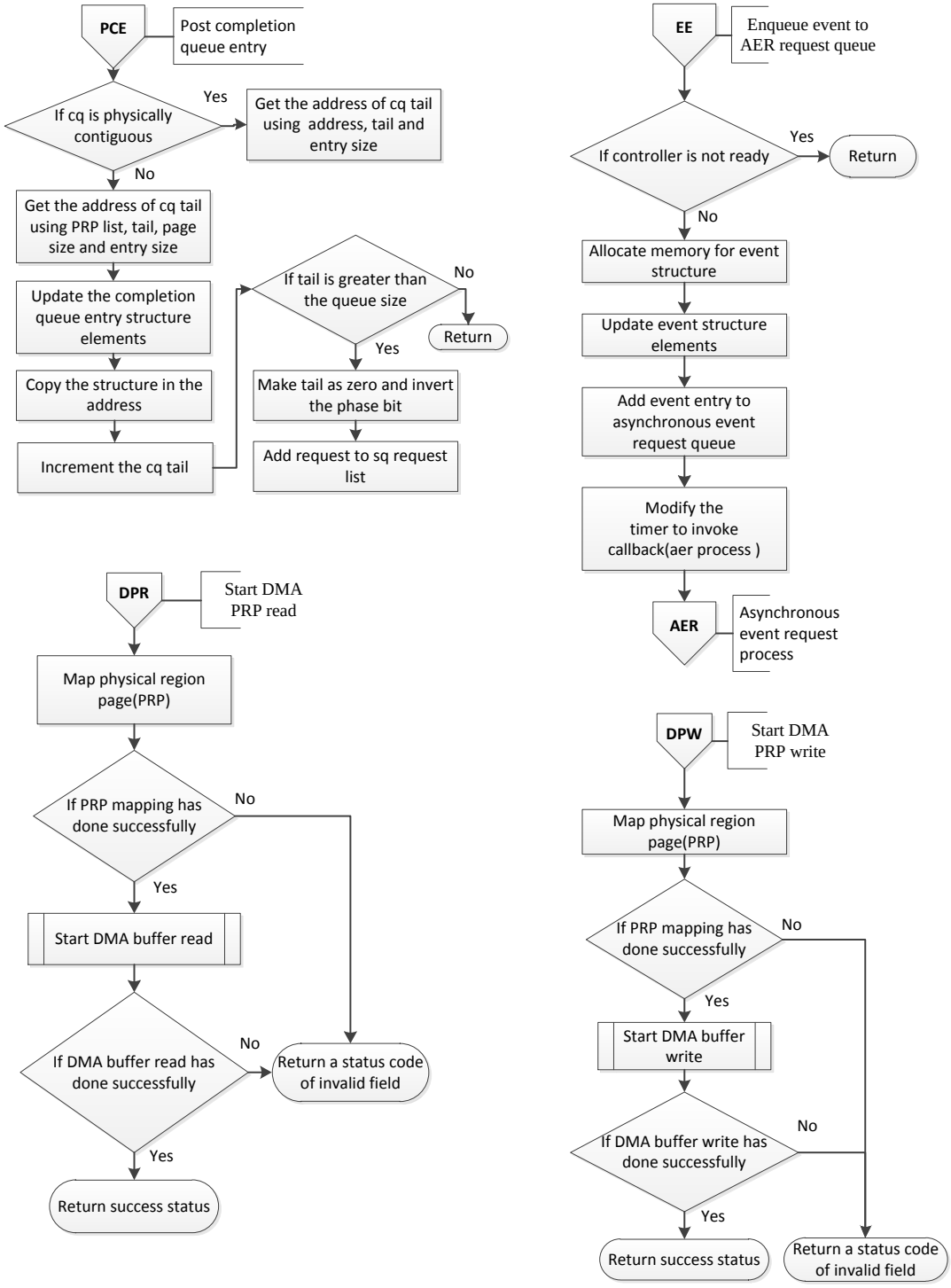


Figure 43: DMA PRP read/write, Post completion queue entry and enqueue event to AER queue

3.6 NVMe IO Commands

I/O Command	Opcode
Write	01h
Read	02h
DSM	09h

Table 28: NVMe IO Command set

3.6.1 Read / Write Commands

3.6.1.1 Read Command

The Read command reads data and metadata, if applicable, from the NVM controller for the LBAs indicated. The command may specify protection information to be checked as part of the read operation.

3.6.1.2 Write Command

The Write command writes data and metadata, if applicable, to the NVM controller for the logical blocks indicated. The host may also specify protection information to include as part of the operation.

3.6.1.3 Software Implementation Structure

```
struct nvme_rw_command {
        uint8_t         opcode;
        uint8_t         flags;
        uint16_t        command_id;
        uint32_t        nsid;
        uint64_t        rsvd2;
        uint64_t        metadata;
        uint64_t        prp1;
        uint64_t        prp2;
        uint64_t        slba;
        uint16_t        length;
        uint16_t        control;
        uint32_t        dsmgmt;
        uint32_t        reftag;
        uint16_t        apptag;
        uint16_t        appmask;
};
```

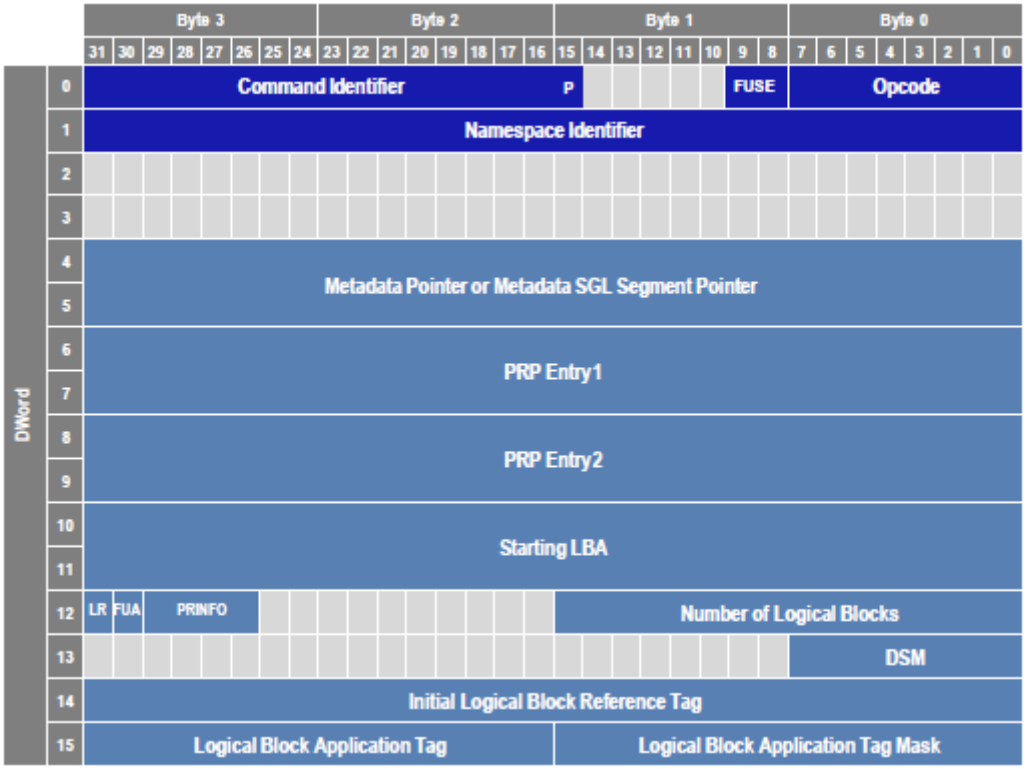


Table 29: Read / Write IO Command Format

- **PRP Entry 1, PRP Entry 2, Metadata Pointer**– Host buffers to write data read from NVM / Host buffers for read data to be written to NVM
- **Starting LBA** – Address of first logical block to read / written
- **Number of Logical Blocks** – Number of logical blocks to read / write
- **Protection Information Field (PRINFO)**
  - **Protection Information Action**
    - Pass protection information or read / write and strip / insert
  - **Protection Information Check**
    - Guard field check or no check
    - Application tag field check or no check
    - Reference tag field check or no check
- **Force Unit Access (FUA)**

  Return data from NVM
- **Limited Retry (LR)**

  Apply limited retry or apply all available error recovery means to return data / write data to NVM
- **Data Set Management (DSM)**
- **Protection Information Related Fields**
  - Initial Logical Block Reference Tag
  - Logical Block Application Tag Mask
  - Logical Block Application Tag

3.6.1.4 Processing the Commands

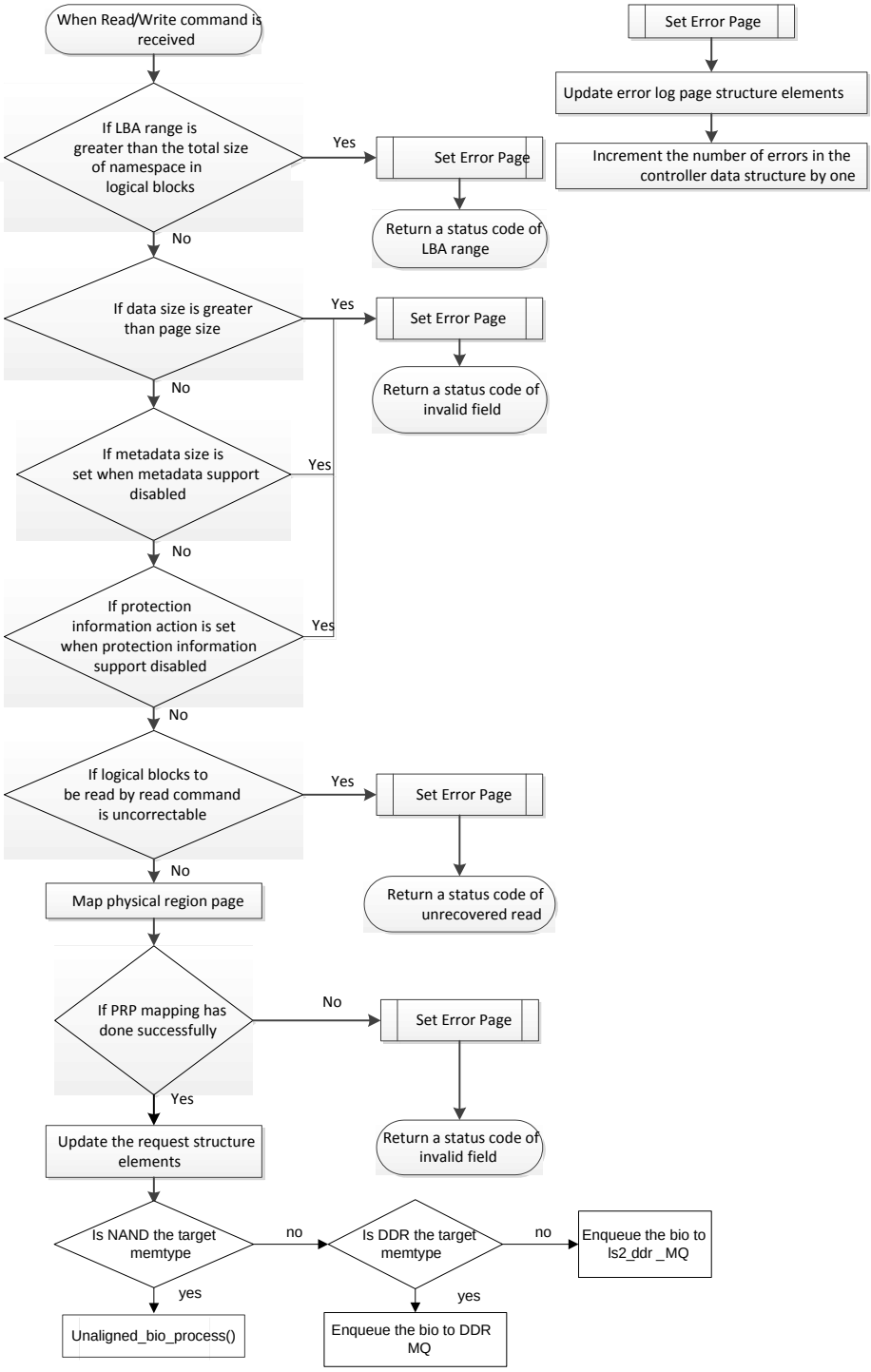


Figure 44: Handling the Read / Write IO Command Flow Chart

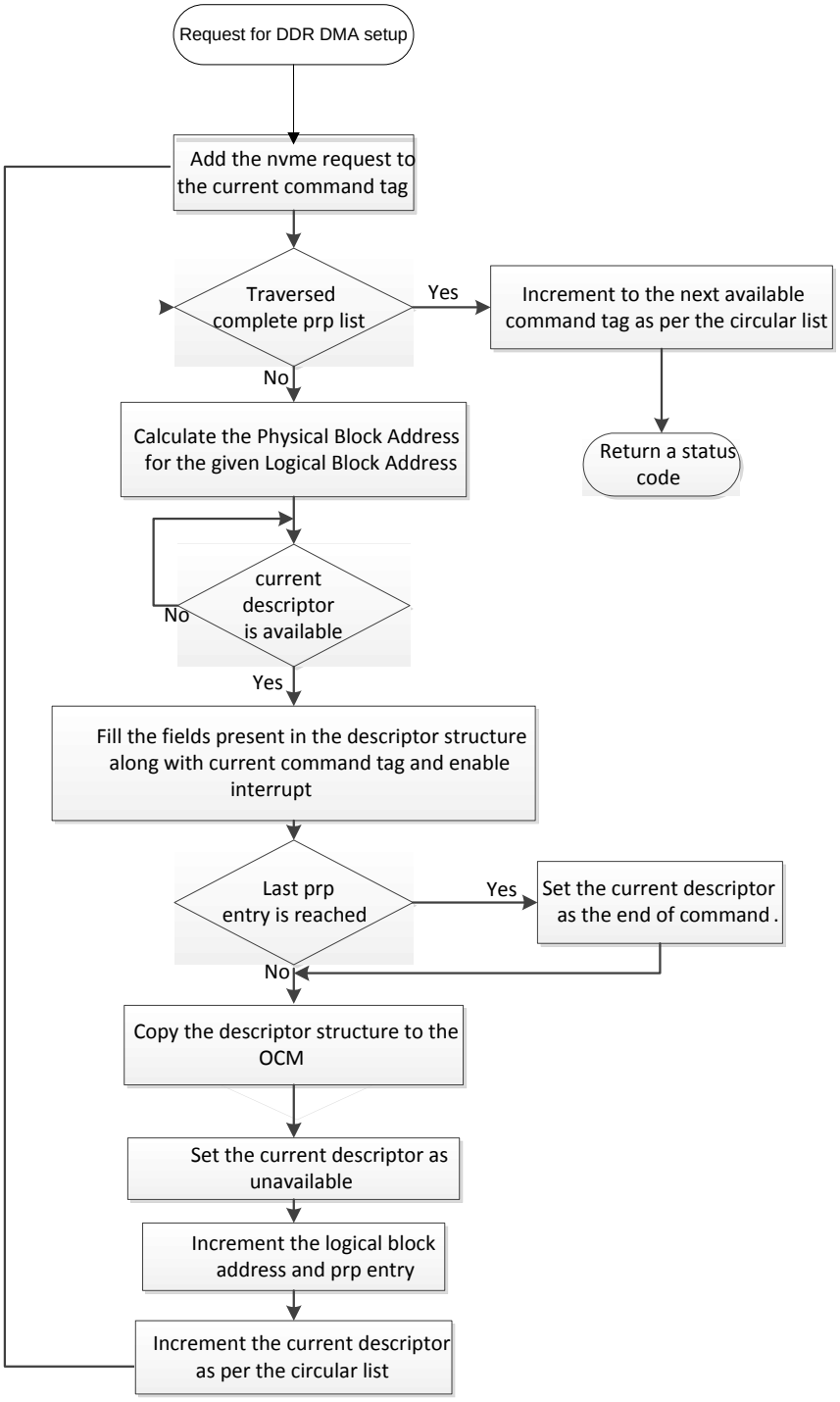


Figure 45: DDR DMA Setup Flowchart

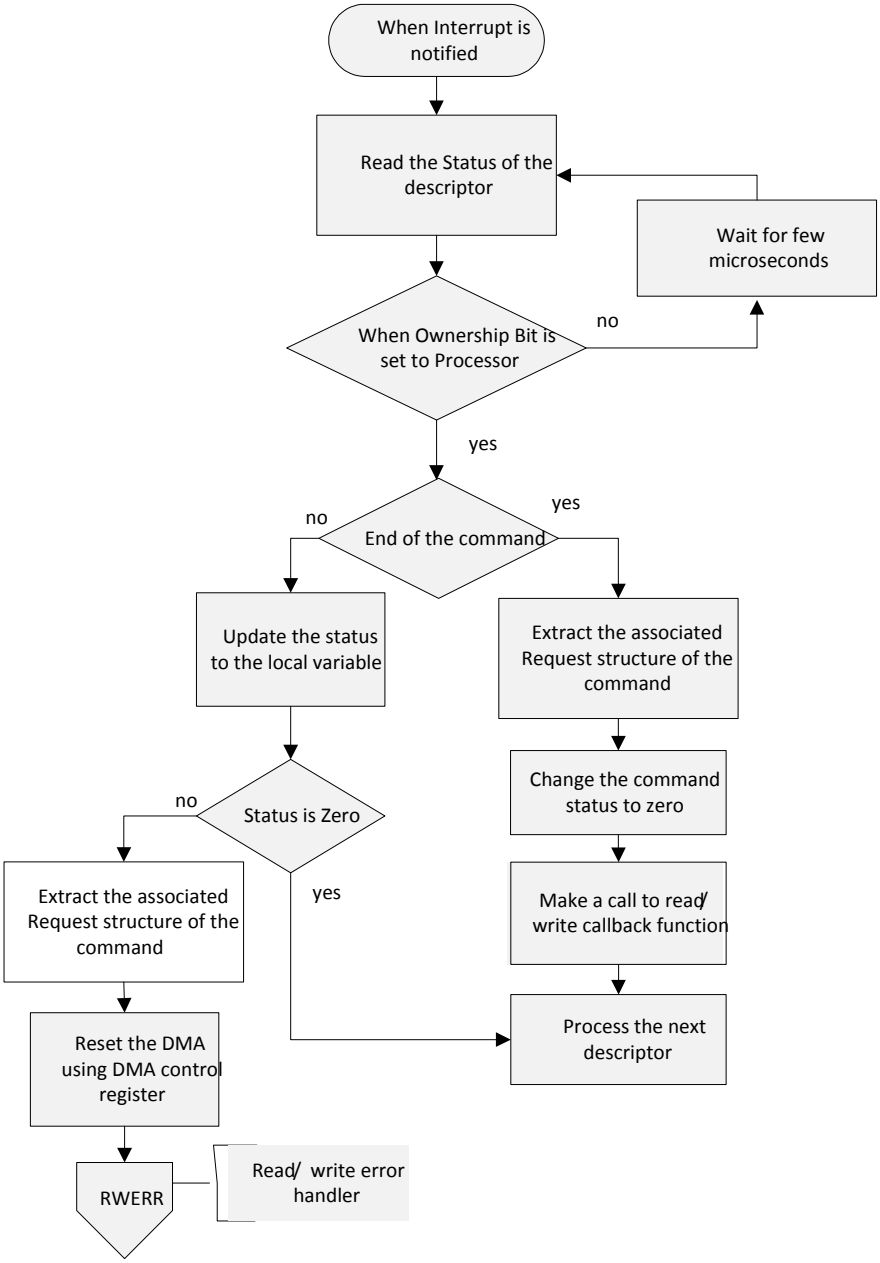


Figure 45: DDR IO Completion Handler Flowchart

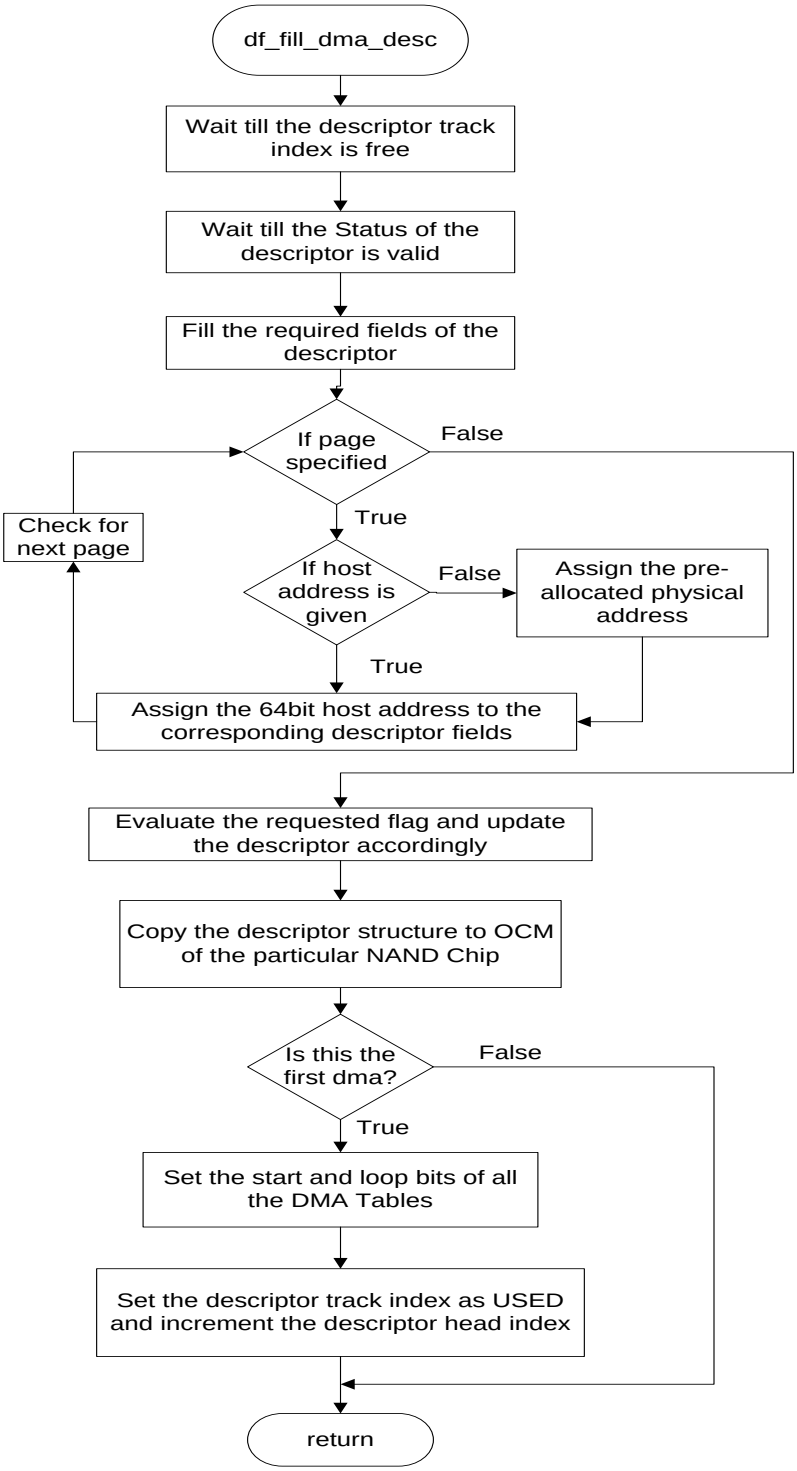


Figure 46: NAND DMA Setup Flowchart

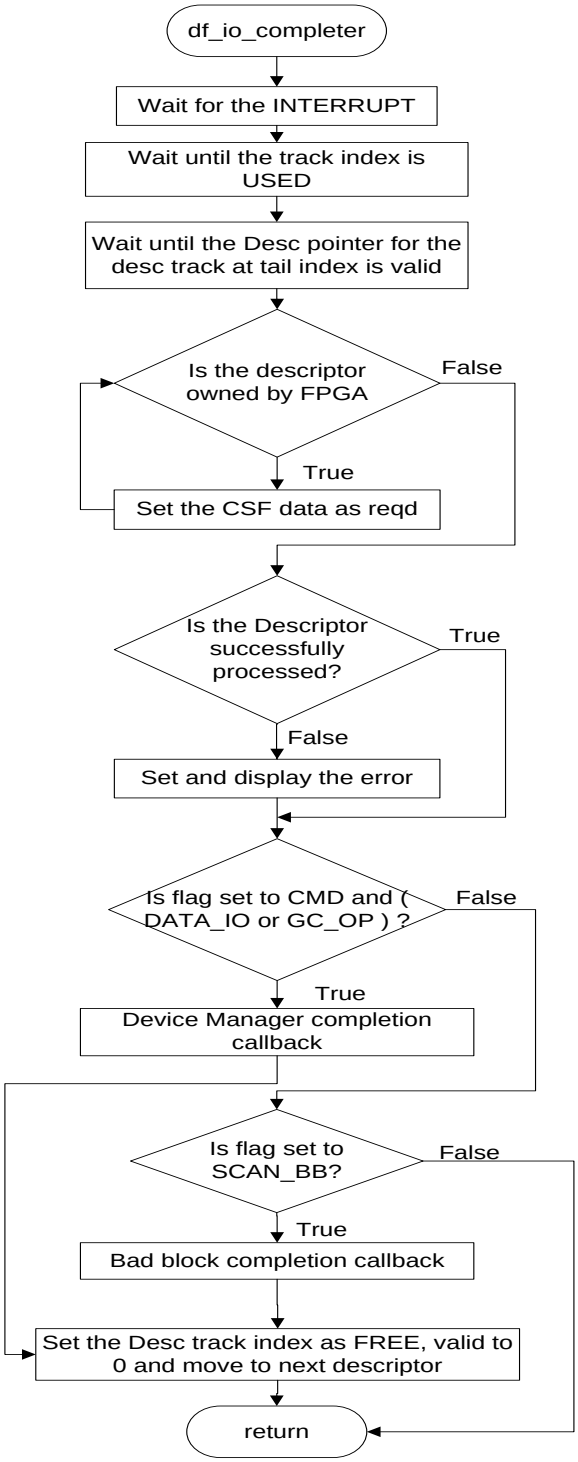


Figure 47: NAND IO Completer

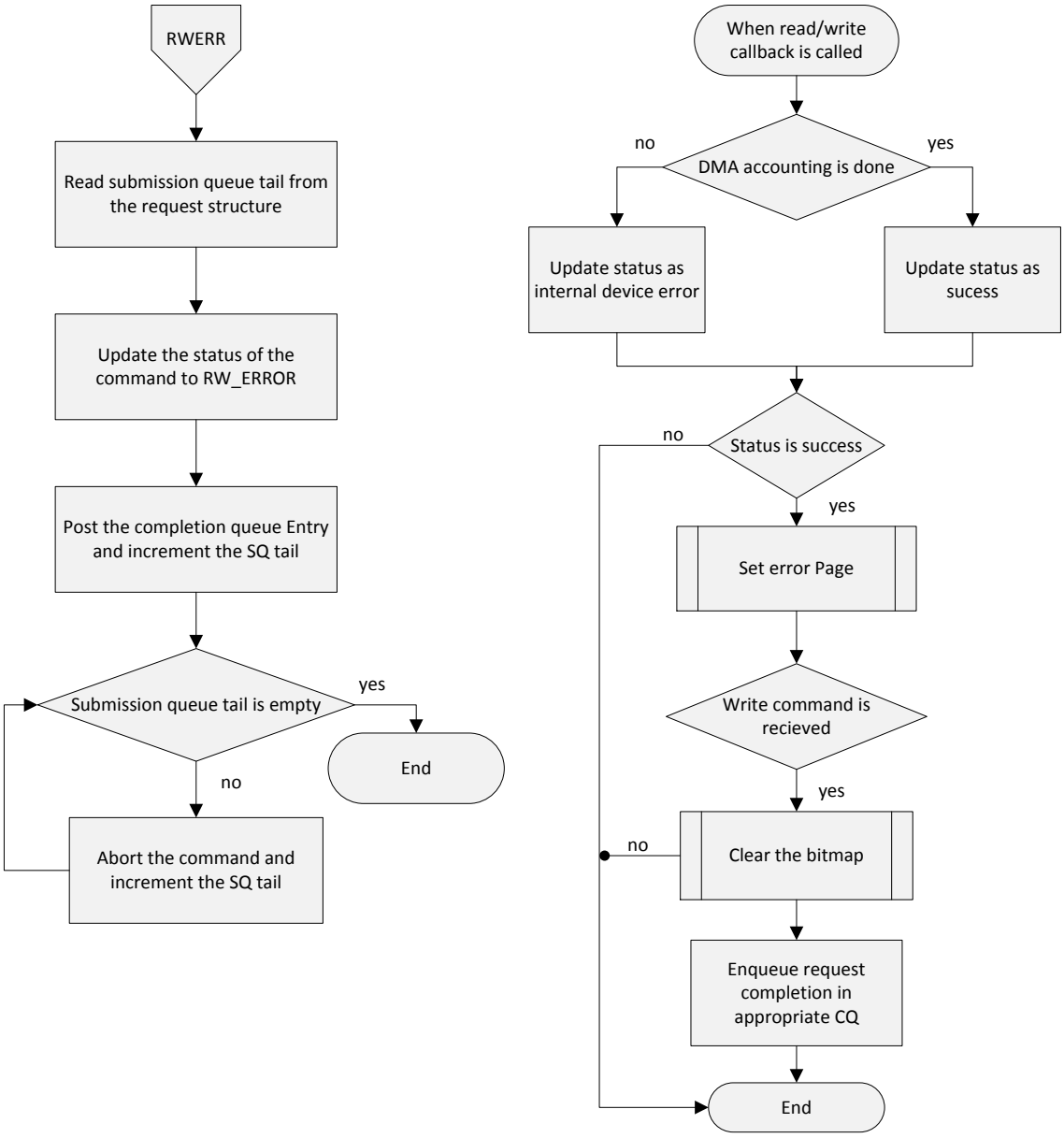


Figure 48: Associate Functions

3.6.2 Flush Command

The Flush command shall commit data and metadata associated with the specified namespace(s) to non-volatile media. The flush applies to all commands completed prior to the submission of the Flush command.

DWord	Byte 3				Byte 2				Byte 1				Byte 0			
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	Command Identifier															
1	Namespace Identifier															
2																
3																
4																
5																
6																
7																
8																
9																
10																
11																
12																
13																
14																
15																

DWord	Byte 1								Byte 0							
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0							FUSE		Opcode							
1	Namespace Identifier															
2																
3																
4																
5																
6																
7																
8																
9																
10																
11																
12																
13																
14																
15																

Table 30: Flush IO Command Format

3.6.2.1 Processing the Command

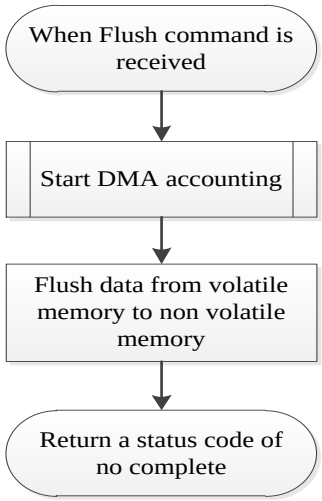


Figure 49: Handling Flush IO Command Flow Chart

3.6.3 Dataset management command

The Dataset Management command is used by the host to indicate attributes for ranges of logical blocks. This includes attributes like frequency that data is read or written, access size and information that may be used to optimize the performance and reliability.

The Attribute-Deallocate field of the command data is set to 1 then the NVM subsystem may deallocate all provided ranges. If a read occurs to a deallocated ranges of blocks, the controller shall return all zeroes, all ones, or the last written data to the associated LBAs. If the deallocated or unwritten logical block error is enabled and a read occurs to a deallocated range, then the read shall fail with the Unwritten or Deallocated Logical Block status code.

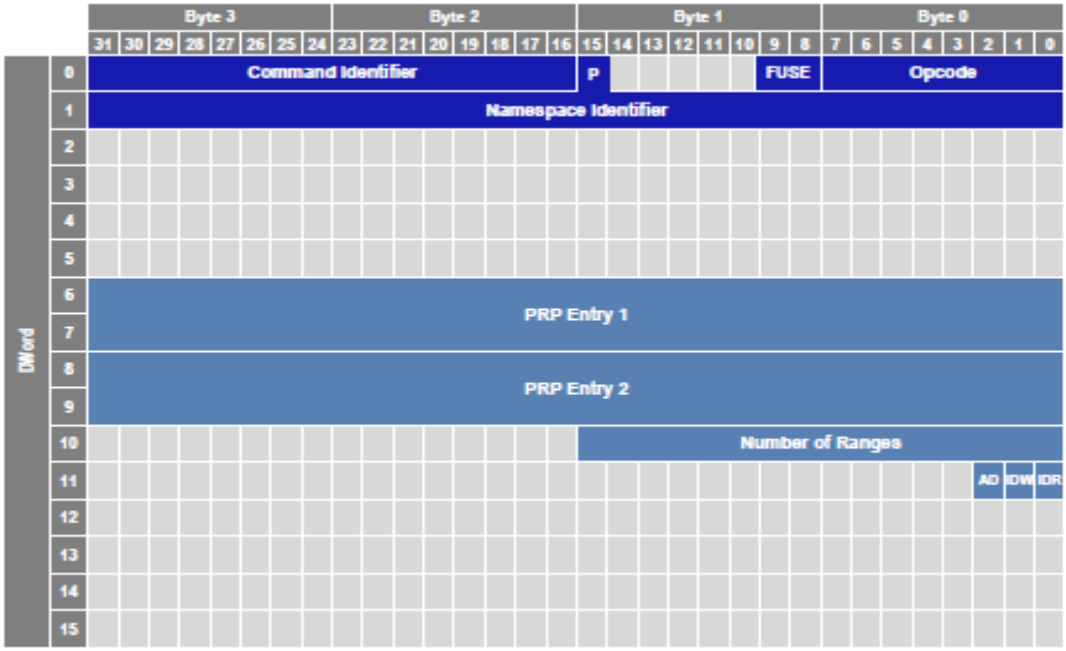


Table 31 Dataset Management command

3.6.3.1 Processing the Command

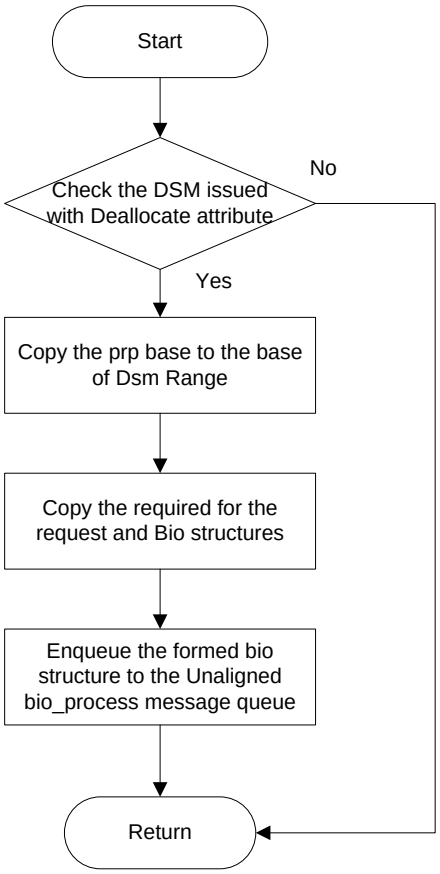


Table 32: Handling DSM command Flowchart

3.6.4 Unaligned IO Processing

When the IOs are about to written in the NAND then the number of kernel pages given to the NAND shall be multiple of 8. Each NAND page could accommodate 16K data. Since the multi-plane operation is being done two pages of the block of consecutive plane are programmed at the same instant. The IO issued with the size not aligned to 8 pages are termed as Unaligned IOs. When an Unaligned IO is issued by the host then the unaligned pages shall be cached in the LS2 DDR. The cached pages are flushed to NAND when the unaligned cache page count exceeds 8.

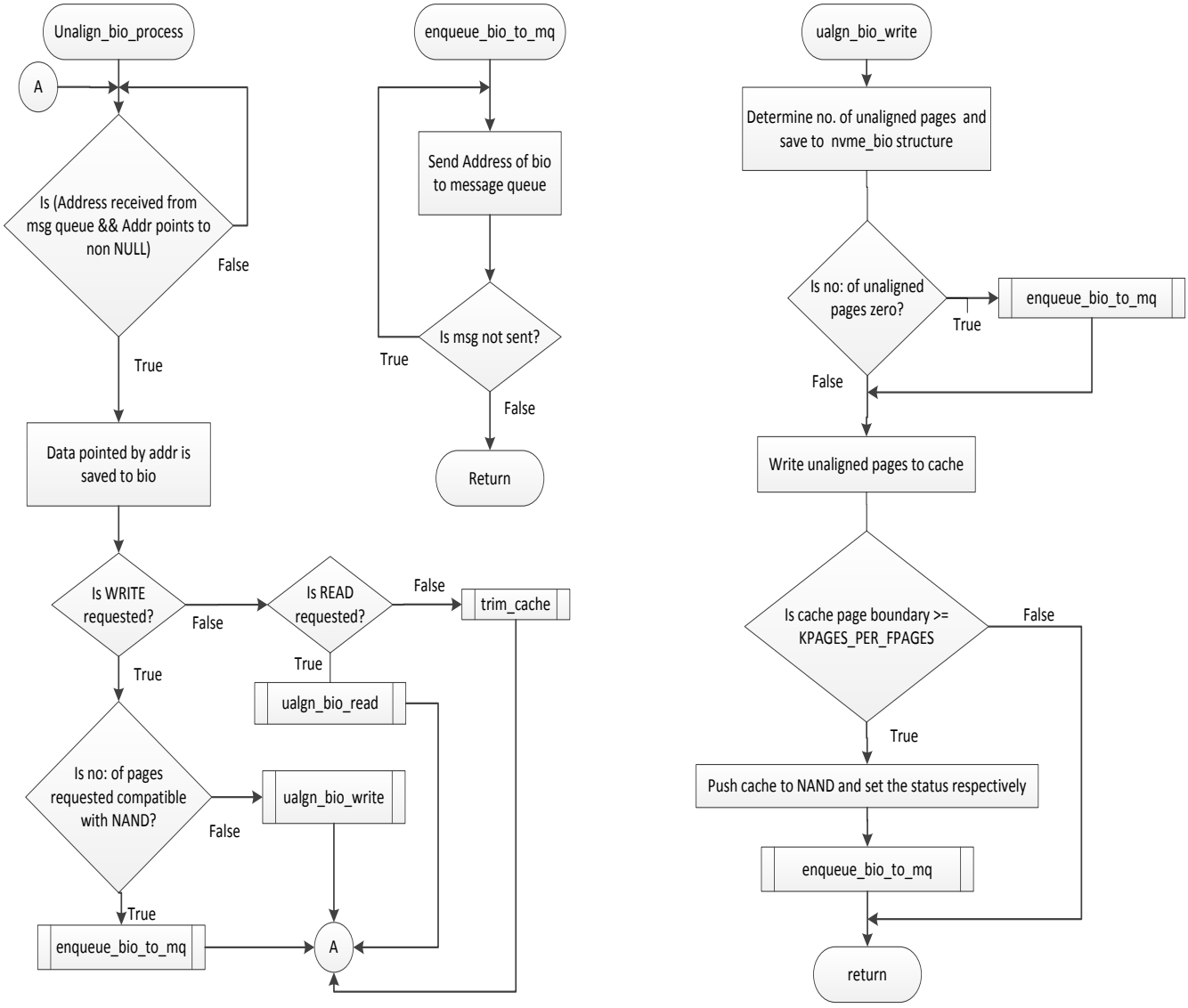


Figure 50: Unaligned Page write

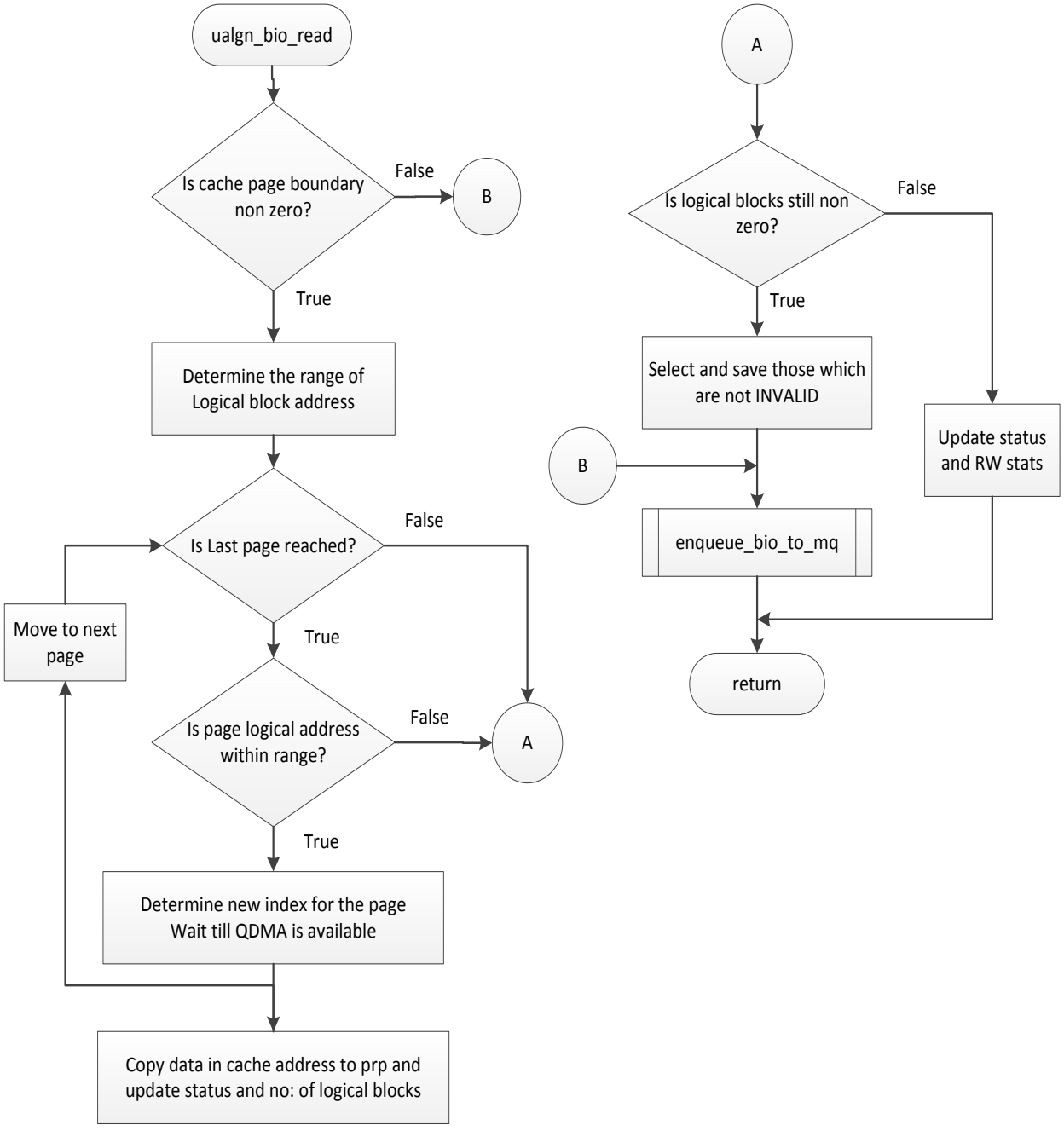


Figure 51: Unaligned Page Read

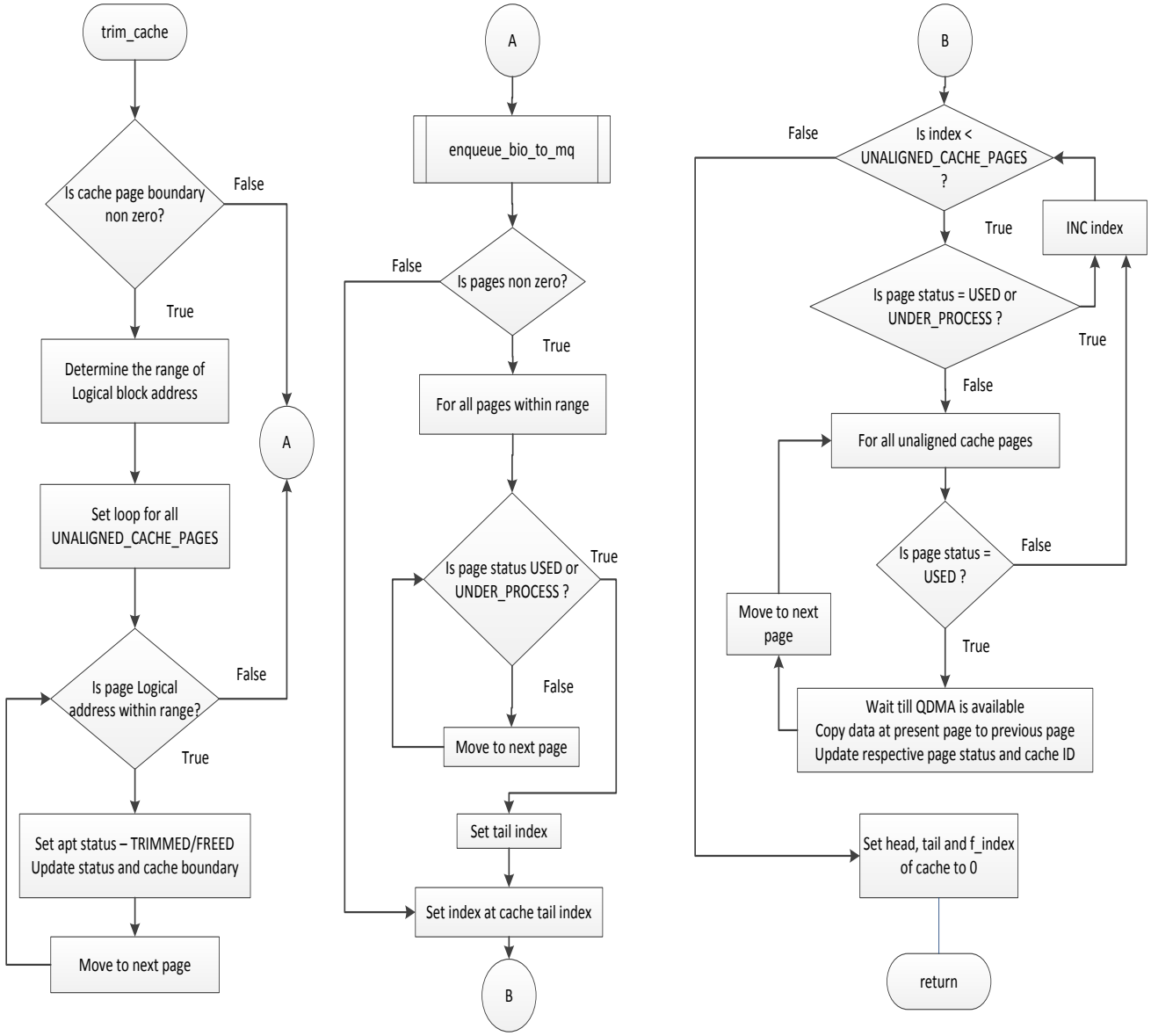


Figure 52: Unaligned Page Trim

3.7 FTL – Flash Translation Layer

The Flash Translation Layer is responsible for converting the LBA to the corresponding flash address.

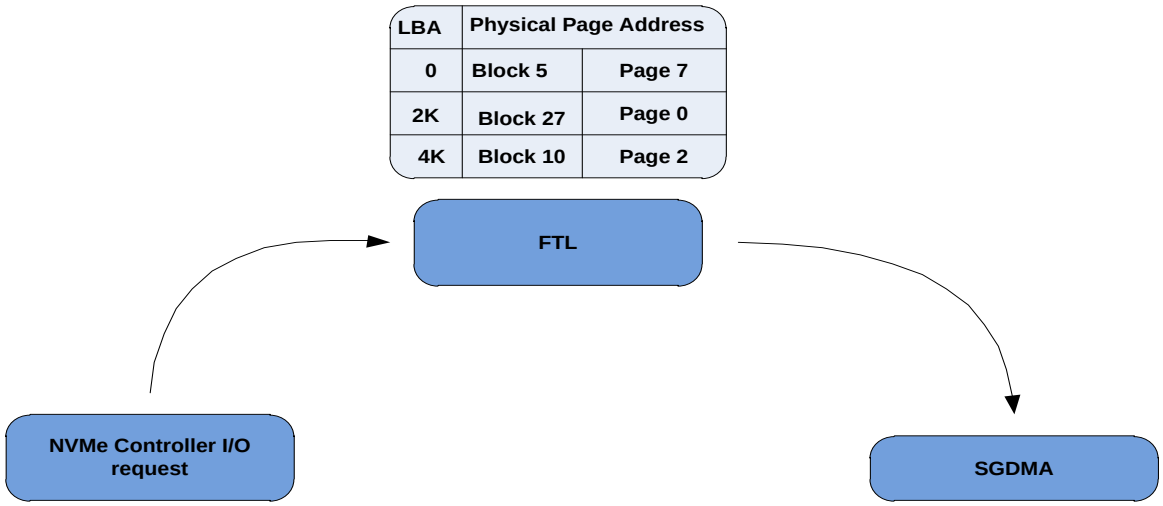


Figure 53: Handling I/O request

3.7.1 Pages, erase blocks and segments

All NAND flash chips are physically organized into "pages" and "erase blocks." A page is the smallest unit that can be addressed in a single read or write operation by the embedded microcontroller on a managed flash device, and it has an effective size between 2KB and 32KB in current consumer flash drives. This means that while a single 512-byte access is possible on the host interface (USB, ATA, MMC ...) it takes almost the same time as a full page access inside of the drive.

Although it is usually possible to write single pages, the data cannot be overwritten without being erased first and erasing is only possible in much larger units, typically between 128KB and 2MB. The controllers groups these erase blocks into even larger segments, called "erase block groups," "allocation units," or simply "segments." The most common size for these segments is 4MB for drives in the multi-gigabyte class, and all operations on the drive happen in these units; in particular, the drive will never erase any unit smaller than a segment.

The drives have a single lookup table which contains a mapping between logical segments and physical segments. On a typical 8GB SD card using 4MB segments, this table contains a little under 2000 entries, which is small enough to be kept in the RAM of the card's microcontroller at all times. A small number of physical segments are set aside in a pool to handle wear leveling, bad blocks and garbage collection.

Ideally, the drive expects

- All data to be written in full segments.
- write updated information to a new empty page and then divert all subsequent read requests to its new address
- ensure that newly-programmed pages are evenly distributed across all of the available flash so that it wears evenly

- keep a list of all the old invalid pages so that at some point later on they can all be recycled ready for reuse

3.7.2 Wear leveling

When a segment that already contains data is written to, a new segment is allocated from the free pool and the drive writes the new data into that segment. Once the segment has been written to from start to finish, the lookup table will be updated to point to the new segment, while the old segment is put into the free pool and erased in the background.

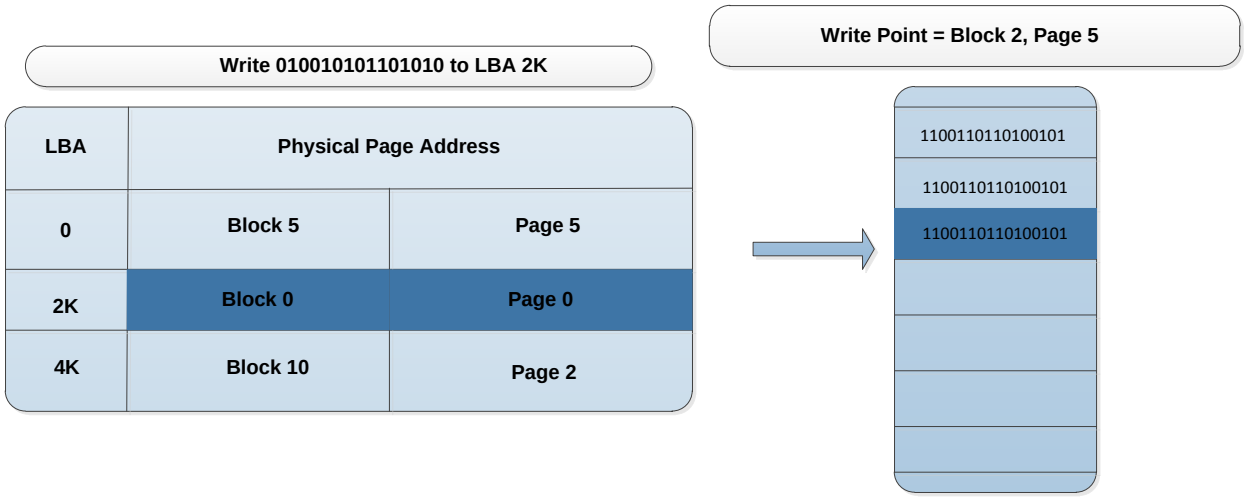


Figure 54: Page Write

When data is changed, the content of the page is copied into an internal register, the data is updated, and the new version is stored in a free page

Block Info Table

Block	Erased	Erase count	Valid Page Count	Sequence Number	Bad Block Indicator
0	False	3	15	5	False
1	True	7	0	-	False
2	False	0	4	9	False

Table 33: Page Write Table

By always allocating a new segment, the drive can avoid wearing out a single physical segment in cases where the host always writes to the same block addresses. Instead, all writes are statistically distributed to all the segments that get written to from time to time.

3.7.3 Garbage Collection

Writing 4 MB segments at once is more the exception than the rule, so drives need to cope with partial updates of segments. While data gets written to a logical segment, the controller normally has an old and a new physical segment associated with it. In order to free up the extra segment, it has to combine all the logical blocks in that segment into physical blocks on only one segment and discard all the previously used physical blocks, a process called garbage collection. A number of garbage collection techniques can be observed in current drives, including special optimizations using caching in RAM or NOR flash and dynamically adapting to the access patterns.

Block Info Table

Block	Erased	Erase count	Valid Page Count	Sequence Number	Bad Block Indicator
0	False	3	13	5	False
1	False	7	1	12	False
2	False	0	3	9	False

Table 34: Page Erase Table

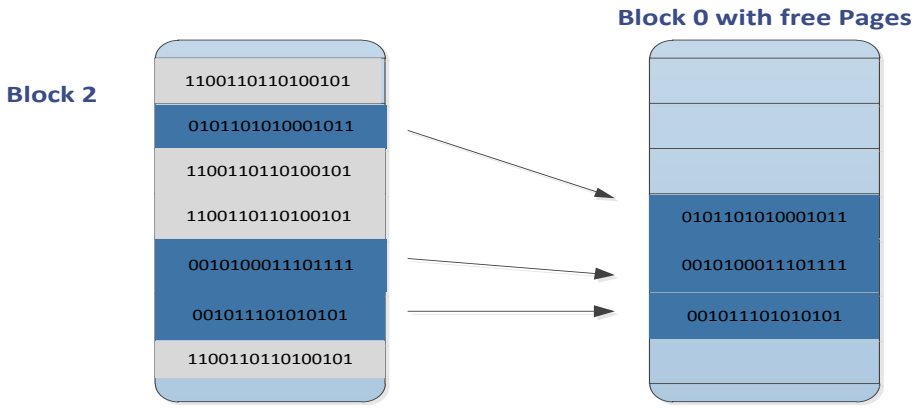
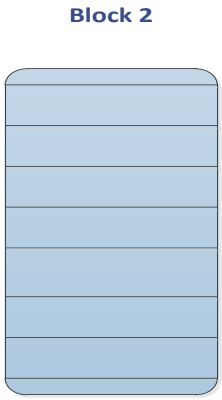


Figure 55 : Valid Page Collection

The erase command is triggered automatically by the garbage collection process in the SSD controller when it needs to reclaim stale pages to make free space.



Block	Erased	Erase count	Valid Page Count	Sequence Number	Bad Block Indicator
0	False	3	13	5	False
1	False	7	1	12	False
2	True	0	3 0	9	False

Table 35: Page Erase Update

3.7.3.1 Block remapping

Fortunately, a significant number of flash drives support random access within a logical segment, by remapping logical blocks to free physical blocks as they get written. Since this requires maintaining another lookup mechanism, both read and write accesses are slightly slower than the ideal linear-access behavior, and a small amount of out-of-band data needs to be reserved to store the lookup table.

This method also does not allow efficient writing in any small units when the manufacturers optimize for larger blocks in order to keep the size of the lookup table small. Writing the same block repeatedly still requires a full garbage-collection, which makes this method unsuitable for storing an ext3 journal or any other data that frequently gets written to the same area on the drive.

3.7.4 Data logging

The best random-access behavior is provided by using the same approach that log-structured filesystems like jffs2, logfs or nilfs2 and block-remappers like UBI in Linux use. Data that is written anywhere in the logical segment always goes to the next free block in the new physical segment, and the drive keeps a log of all the writes cached. Once the last free block is used up, a garbage collection is performed using a third physical segment.

In the end, writing this way is slower than the other two approaches in the best case, because every block is written at least twice, but the worst case is much better.

This approach is normally used only in the first few segments on the drive, which contain the file allocation table in FAT32 preformatted drives. Some drives are also able to use this mode when they detect access patterns that match writes to a FAT32 style directory entry.

Obviously, any such optimizations don't normally do the right thing when a different filesystem is used on the drive than it was intended for, but there is some potential for optimization, e.g. by ensuring that the ext3 journal uses the blocks that are designed to hold the FAT.

3.7.5 Data Structures for Mapping

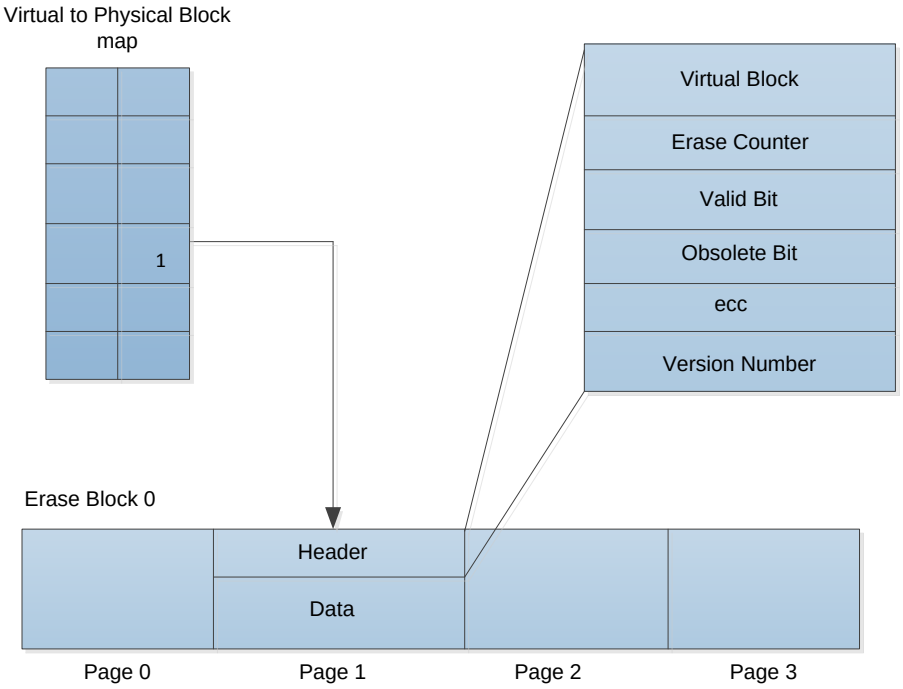


Figure 56: Block Mapping

Direct maps allow efficient mapping of blocks to sectors, and inverse maps allow efficient mapping of sectors to blocks.Inverse maps as shown above are stored on the flash device itself. When a block is written to a sector, the identity of the block is also written.

3.7.6 NVMe-FTL Interface

The interface layer between NVMe and FTL is called "**nvme_bio**" which converts all NVMe Logical Block based IO requests to FTL-format.

The original FTL interface expects a Block IO request KERNEL structure as input. Since the FTL is ported to user-space, a new IO structure is introduced to have proper interface with NVMe IO requests to FTL, but with the some similarities with Kernel BIO structure – It all gives the target sector based request info as FTL expects.

It has only the necessary parameters NVMe must give to FTL for proper handling of all sector-based requests and thus calculating the pages or blocks to be accessed accordingly on the NAND targets.

3.7.6.1 NVMe Bio Structure

The nvme_bio structure definition is as below:

```
struct nvme_bio {
      void        *req_info;
```

```
        uint64_t    slba;
        uint16_t    offset;
        uint64_t    nlb;
        uint64_t    prp[128];
        uint64_t    size;
        uint32_t    req_type;
        uint16_t    nalloc;
        uint16_t    nprps;
}
```

Description:

req_info : Pointer to the nvme IO request structure with all SQ and corresponding CQ info. Used by completion handler once a BIO is completed to post Complete Queue entry to Host.

slba : Starting logical block address.

offset : offset within the first sector.

nlb : Number of required logical blocks starting from first sector.

prp : Pointer to host memory which holds data for IO operation (Physical Region Page).

size : Total size of data for IO operation in bytes.

req_type : Request type(read/write).

nalloc : used for checking dynamic allocation for prp array. (Not used by FTL, but for NVMe BIO request management)

nprps : number of prps.

NOTE: The same nvme_bio structure could be used by DDR based setup; no change/manipulation is needed in any of the structure member.

3.7.7 FTL High Level Flow Diagram

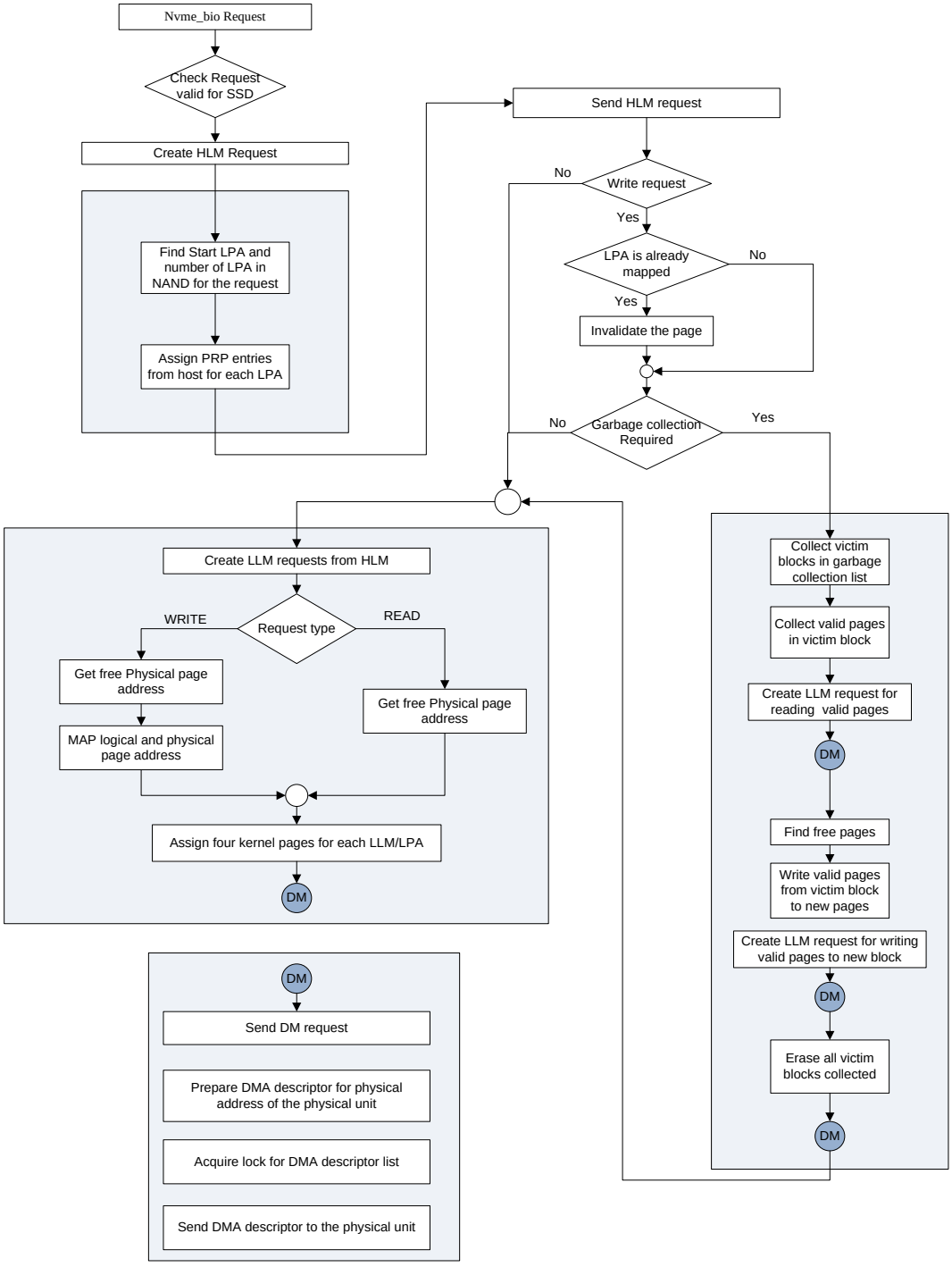


Figure 57: FTL High Level Flow chart

3.8 FPGA Software Design

The FPGA software design consists of two major branches – DDR-based and NAND-based, mainly because of the differences in the memory access at software and hardware levels in FPGA.

3.8.1 DDR Based Design

This section explains the DDR based implementation in FPGA software, along with the DMA based IO operations requested by NVMe.

3.8.1.1 Block Diagram

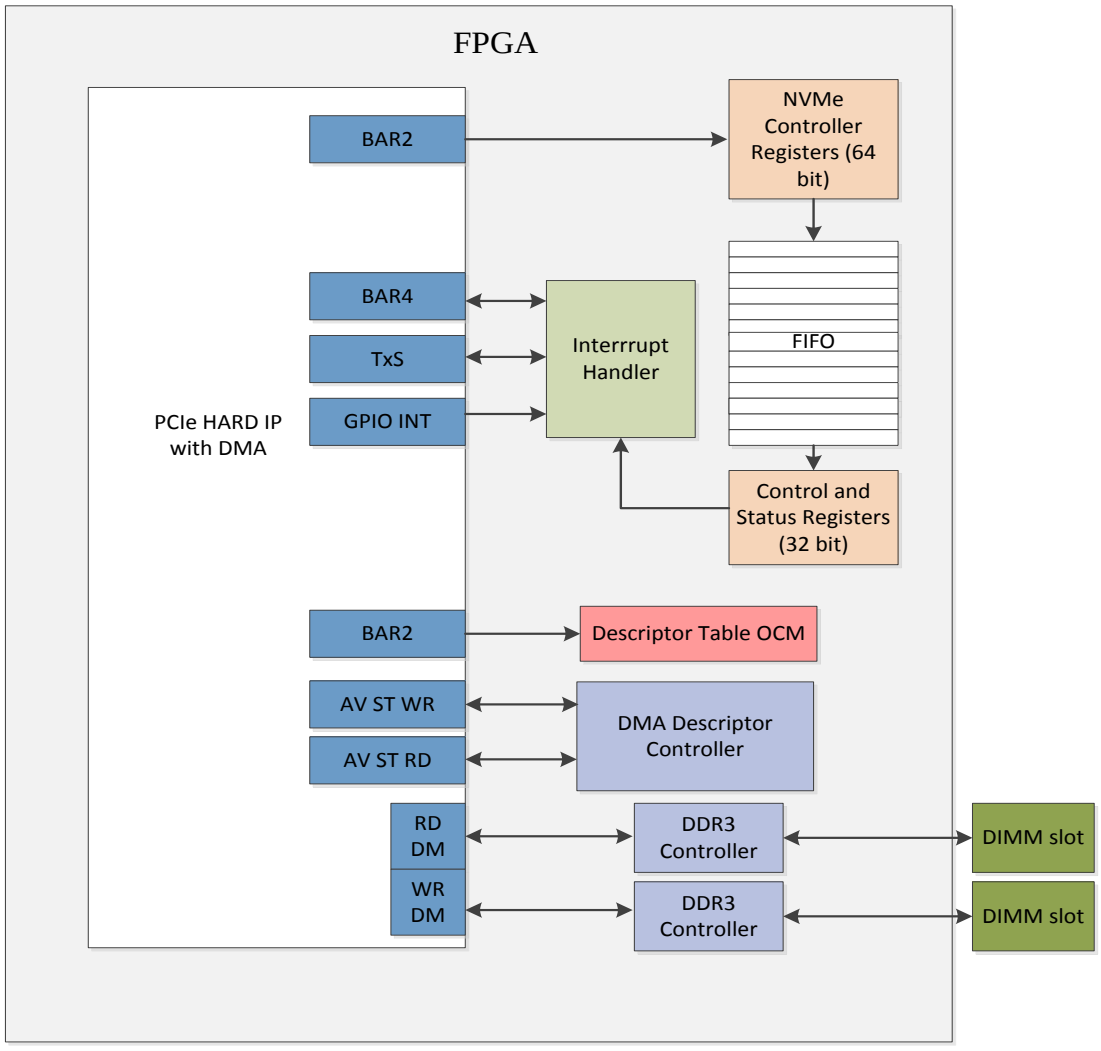


Figure 58: FPGA Block Diagram

3.8.1.2 Descriptor Structure

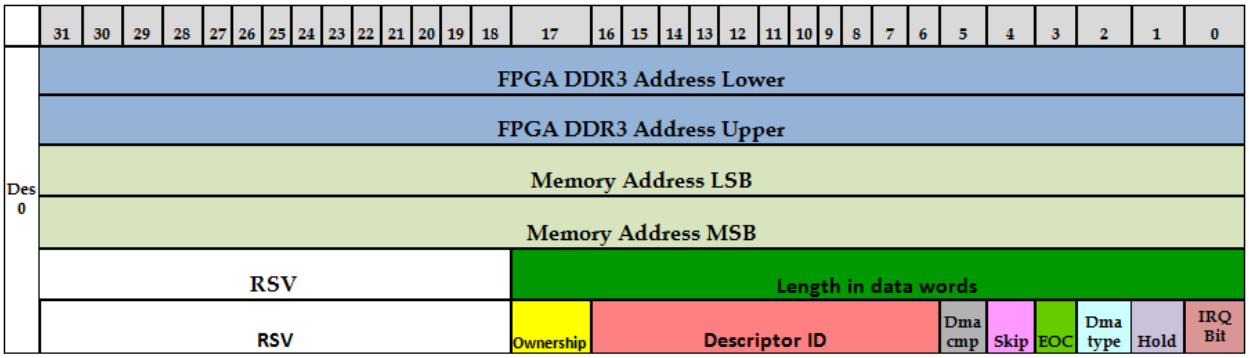


Table 36: Descriptor Structure

- **FPGA DDR3 Address Lower :** 32 bit lower address for 4 GB addressing
- **FPGA DDR3 Address Upper :** We areusing**2**bits fromthese32 bits along with Lower address to address upto 16 GB
- **Memory address LSB :** Lower 32 bits address of Host Memory
- **Memory address MSB :** Upper 32 bits address of Host Memory
- **Interrupt bit:** Host can set this bit to 1 to raise the interrupt after that particular descriptor operation is completed.
- **Hold bit:** Host can set this bit to one to hold the next descriptor fetch till the start signal from host.
- **DMA Type:**To select commands for the DDR(Only Read and Write in case of DDR3)
- **Length in Data Words :** Length of data transmission in Data Words
- **EOC bit:** Software can write a '1' to identify the end descriptor in a command
- **Skip bit:** Software can write a '1' to skip the specific descriptor
- **DMA Complete:** When a DMA is successfully completed this bit is set to '1'.
- **Command ID :** To select commands for the DDR(Only Read and Write in case of DDR3, other commands are ignored)
- **Descriptor ID :** ID of the descriptor
- **Ownership bit:** To set the ownership of the descriptor between FPGA and processor. If this bit is set to 1 by host, then the ownership is transferred to FPGA DMA Descriptor Controller, Descriptor controller can fetch and execute that descriptor. After completing a descriptor operation FPGA will update the status of operation in to the status field and set this bit to 0 to transfer the ownership back to processor.

3.8.1.2.1 Software Implementation Structure

```
typedef struct desc {
      uint64_t    ppa;
      uint64_t    prp;
      uint32_t    data_len:18;
      uint32_t    rsv3:14;
      uint32_t    irq_en:1;
      uint32_t    hold: 1;
```

```
      uint32_t    opcode:1;
      uint32_t    eoc:1;
      uint32_t    skip:1;
      uint32_t    dma_cmp:1;
      uint32_t    cmd_tag:11;
      uint32_t    ownedByFpga:1;
      uint32_t    rsv4:14;
} desc_t;
```

- **ppa:** Address of the DDR3.
- **prp:** 64 bit host address.
- **data_len:** Length of data transfer in bytes.
- **rsv3:** Reserved bits.
- **irq_en:** Interrupt bit.If this bit is set, then interrupt will be raised when this descriptor is processed.
- **hold:** Hold bit is set to hold the next descriptor fetch till the start signal from host.
- **Opcode:** Gives about type of request(read/write).
- **eoc:** End of the command indication bit.
- **skip:** If the descriptor is skipped without filling any fields, then skip bit will be set.
- **dma_cmp:** This bit will be set if the descriptor has been successfully processed by DMA controller.
- **cmd_tag**: For each command, ID will be provided to identify that particular command.
- **ownedByFpga:** To set the ownership of the descriptor between FPGA and processor. If this bit is set to 1 by host, FPGA has to handle the descriptor otherwise FPGA will skip to next descriptor. After a particular operation, FPGA will set this bit to 0.
- **rsv4:** Reserved bits.

3.8.1.3 DMA Control and Status Register (CSR)

31		17	16	15	14	13	12	11	10		0
Reserved			Reset	Loop	Start	Error Code			Descriptor ID		

Table 37: Control and Status Register for descriptor Controller

- **Descriptor ID:** This 8 bit field specifies which descriptor in the Descriptor table is the cause for Interrupt.
- **Start:** Start bit to start DMA transaction for the specified Descriptor controller. Software can write a '1' to start DMA
- **Loop:** If this bit is set to '0' by the software, Descriptor controller stops after reading and processing the 128 descriptors. Else the Descriptor controller starts again to process from descriptor '0'.
- **Reset:** This bit is set to reset the entire DMA logic as well as descriptor table once an error occurs in the data transfer. Should be set separately for both the PCIe's.
- **Error Code:** This three bit field specifies the type of error.

Error Code			Description
0	0	0	No IRQ
0	0	1	Write Time Out
0	1	0	Read Time Out
0	1	1	No Error, User Requested Interrupt
0	0	0	No Error, User Requested Interrupt with hold

Table 38 Error Code for CSR

3.8.1.4 Interrupt Control Register

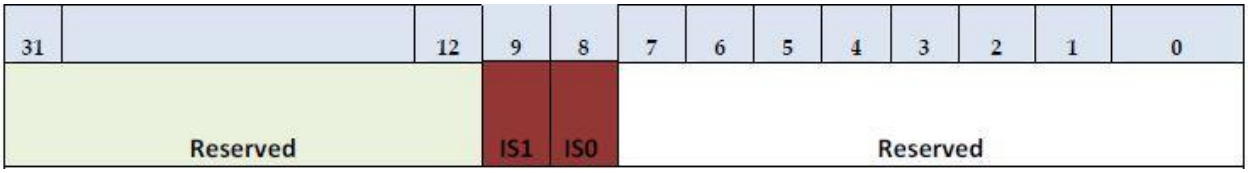


Table 39: Interrupt Control Register

- **ISn:** Individual Interrupt Status bits. Sets when an interrupt occurs. Need to be cleared by software to resume DMA operations in case of IS0 and IS1

3.8.1.5 Descriptor Table Size Register

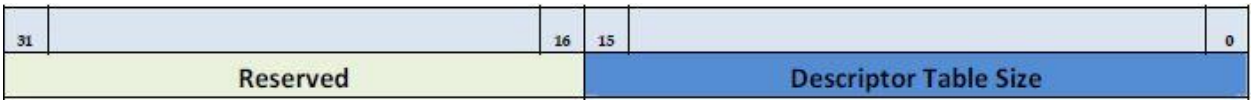


Table 40: Descriptor Table Size Register

- By default descriptor table size will be 1024, which is the maximum value. Software can update this size using the Descriptor table size register, there by the maximum number of descriptors is configurable**.**

3.8.1.6 Interrupt Generation

GPIO Interrupt	Function
PIN AE19	Descriptor Controller 0(PCIe2)
PIN AG18	NVMe FIFO Interrupt
PIN AG19	NVMe Doorbell Interrupt

Table 41: MSI IRQ functions

3.8.1.6.1 DMA Interrupts (PIN AE19)

Once an Interrupt condition is occurred, the corresponding Descriptor controller will generate the specific interrupt. The Interrupts should to be enabled globally as well as individually for this to happen. Once the interrupt is raised software can read the corresponding CSR to check the status. Software need to manually clear this bit after interrupt service routine to resume the descriptor controller operation.

3.8.1.6.2 NVMe Interrupts (PIN AG18 & AG19)

Once an Interrupt condition is occurred, single interrupt will be generated and no need to clear the interrupts manually from software side.

3.8.1.7 FPGA Control/Status Register

This is a 32-bit register bank for different Control/Status register. Following registers are included in this register bank. See register descriptions for details.

- FIFO read data
- FIFO read count
- RESET register
- Interrupt register
- Submission queue write status register0
- Submission queue write status register1
- Submission queue write status register2
- Submission queue write status register3
- GPIO Interrupt register

Address	Registers description	Type	Width	Reset Value
0x0000	RESERVED	NA	32 bit	0xFFFFFFFF
0x0001	FIFO_READ_DATA	Read Only	32 bit	0x00000000
0x0002	FIFO_RD_CNT	Read Only	32 bit	0x00000000
0x0003	RST_REG	Write Only	32 bit	0xFFFFFFFF
0x0004	FIFO_REG_INTR	Write Only	32 bit	0x00000000
0x0005	QUEUE_STATUS_REG0	Read/Write	32 bit	0x00000000
0x0006	QUEUE_STATUS_REG1	Read/Write	32 bit	0x00000000
0x0007	QUEUE_STATUS_REG2	Read/Write	32 bit	0x00000000
0x0008	QUEUE_STATUS_REG3	Read/Write	32 bit	0x00000000
0x0009	GPIO_CSR	Read/Write	32 bit	0x00000000
0x4001	GPIO_INTR	Read/Write	32 bit	0x00000000

Table 42: FPGA Control/Status Registers

- **FIFO_READ_DATA:** Register used to have various entries from a FIFO where each entry is generated due to an NVMe register update – the registers include Controller registers and Admin SQ/CQ Doorbell registers. 1 FIFO entry has fields as below:
    - 64 bit New Value: The new value of updated register in 64-bit format.
    - 64 bit Offset: Offset of the NVMe register updated.

64 bit New Value	64 bit Offset

NOTE: All IO SQ Doorbell register updates will go to the IO-SQDB status register where each bit in the 128 bit register as explained in the Q Scheduler section.

- **FIFO_RD_CNT:** A count register which is used to indicate the current number of FIFO entries. For each new FIFO entry we'll have an increment of 1 in the FIFO_RD_COUNT register. The maximum value as of now being 256, and it is guaranteed the fifo entries cannot go beyond that count because the entries are generated only due to Normal registers and Admin commands. These are always processed as highest priority. Moreover, these updates come up during NVMe driver initialization and controlling IO Queues which is not frequent.
- **RST_REG:** Writing a particular bit sequence shall reset the FPGA modules.
    - **Bit 0: FIFO reset**
    - **Bits 1-31: Reserved.** Will be updated for other module reset functions.
- **FIFO_REG_INTR:** MSI interrupt control register in FPGA.
- **QUEUE_STATUS_REGx:** 128 bit status register given by four 32 bit registers to hold IO SQDB status.
- **GPIO:** Allows LS2 to configure and control/mask interrupts via GPIOs connected between LS2 and FPGA.

3.8.1.7.1 Software Implementation Structure

```
#define DESC_TBL_COUNT          1
#define DESC_TBL_ENTRIES        128

#define MAX_LBA_COUNT           2097152
#define PAGE_SIZE               4096

#define DESC_TBL_OFFSET         0x0000000000200000
#define CSR_OFFSET              0x0000000000008000
#define INTR_OFFSET             0x0000000000008004
#define DESC_SIZE_OFFSET        0x0000000000008008
#define DESC_SIZE               0x20
```

DESC_TBL_COUNT: Total number of Tables(each table represent one ddr)
DESC_TBL_ENTRIES: Maximum descriptor count per table
MAX_LBA_COUNT: Maximum number of Logical block address
PAGE_SIZE: Page size
DESC_TBL_OFFSET: PCIe register base address of descriptor table
CSR_OFFSET: Control and status register base address
INTR_OFFSET: Interrupt control register base address

DESC_SIZE_OFFSET: Descriptor Table Size Register base address
DESC_SIZE: Size of each descriptor in bytes

- **desc_ptr:** Pointer to the fpga memory where the particular descriptor is stored.
- **available:** Flag to indicate whether the particular descriptor can be used for dma operation.
  nvme bio thread sees if set to 1, it resets to 0 and uses the available descriptor;
  completion thread sees if set to 0,uses descriptor status for completion info and set status as 1 to make it available

```
struct desc_info{
        uint64_t                *desc_ptr;
        volatile uint8_t        available;
} g_desc_info[DESC_TBL_COUNT * DESC_TBL_ENTRIES];
```

3.8.2 NAND Based Design

This section explains the NAND based implementation in FPGA software, along with the DMA based IO operations requested by NVMe.

3.8.2.1 Block Diagram

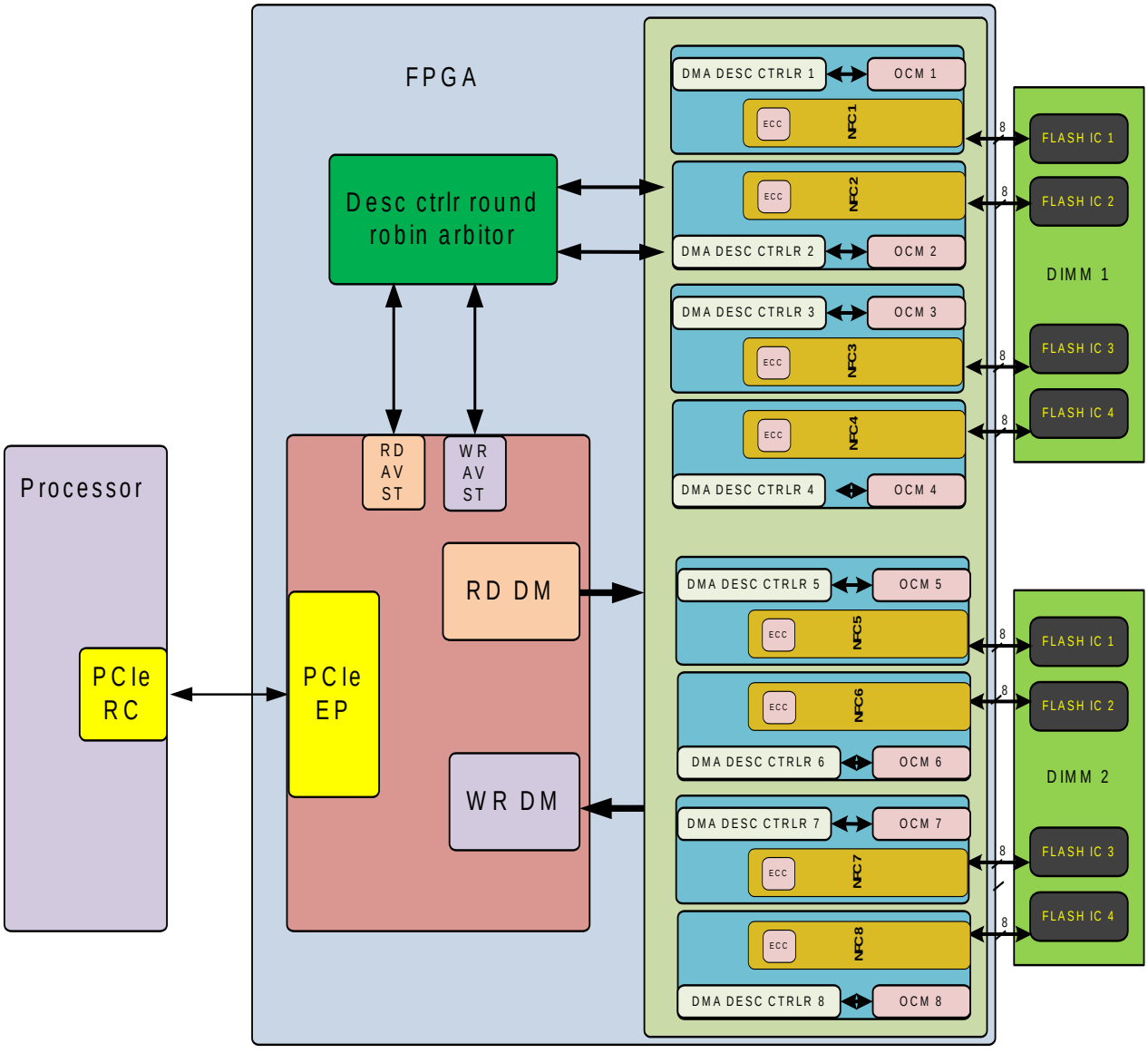


Figure 59: NAND Block Diagram

3.8.2.2 Descriptor Structure

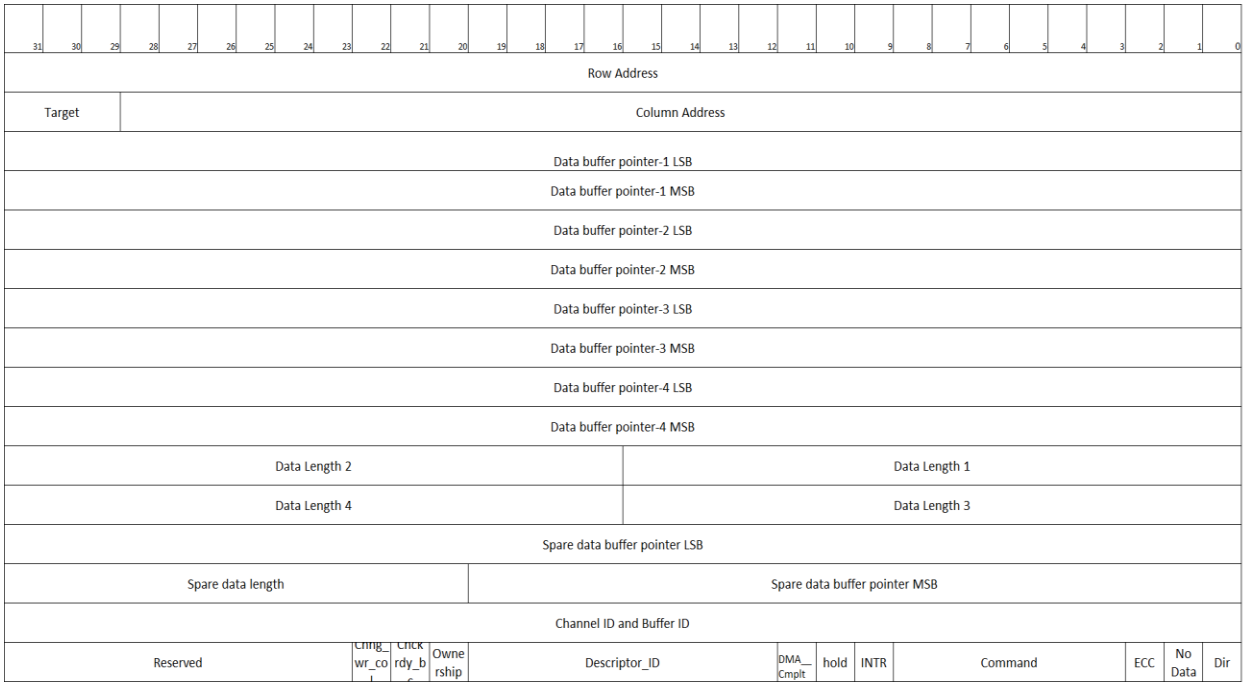


Table 43: NAND DMA Descriptor

Row Address : This register defines the address row selected in the NAND Flash page. The values of Row3, Row2 and Row 1 depend on the NAND Flash component. Address is used for Set feature, Get feature and Volume Select commands.
Column Address: This register defines the address column selected in the NAND Flash page
Target selection : Gives the device in a NAND Bank channel targeted by the command.
Data Buffer Pointer LSB 1: Lower 32 bits address of host memory Page 1.
Data Buffer Pointer MSB 1 : Upper 32 bits address of host memory Page 1.
Data Buffer Pointer LSB 2: Lower 32 bits address of host memory Page 2.
Data Buffer Pointer MSB 2 : Upper 32 bits address of host memory Page 2.
Data Buffer Pointer LSB 3: Lower 32 bits address of host memory Page 3.
Data Buffer Pointer MSB 3 : Upper 32 bits address of host memory Page 3.
Data Buffer Pointer LSB 4: Lower 32 bits address of host memory Page 4.
Data Buffer Pointer MSB 4 : Upper 32 bits address of host memory Page 4.
Data Length 1: Length of the data to transferred from data buffer pointer 1.
Data Length 2: Length of the data to transferred from data buffer pointer 2.
Data Length 3: Length of the data to transferred from data buffer pointer 3.
Data Length 4: Length of the data to transferred from data buffer pointer 4.
Spare Data Buffer Pointer LSB: Lower 32 bits address of spare data memory.
Spare Data Buffer Pointer MSB: Upper 20 bits address of spare data memory.

Spare Data length : Length of the data to transferred from spare data buffer pointer.
Direction Control : Direction of the data transfer in Data phase. DIR = 0 means read DMA operation. DIR = 1 means write DMA operation.
No data : This parameter indicates that “no data” is used with this command if the bit is set to 1.
ECC : ECC disable . Minimum 1 KB data to be transferred for ECC generation and correction.
Command : This Field indicates the command to be executed by the NAND.
Interrupt : Host can set this bit to 1 to raise the interrupt after that particular descriptor operation is completed.
Hold : Host can set this bit to one to hold the next descriptor fetch till the start signal from host.
Dma_Cmp : This bit will be set once the particular descriptor is processed successfully.
Desc_id : ID of the descriptor
Ownership : To set the ownership of the descriptor between FPGA and processor. If this bit is set to 1 by host, then the ownership is transferred to FPGA DMA descriptor controller, Descriptor controller can fetch and execute that descriptor. After completing a descriptor operation FPGA will update the status of operation in to the status field and set this bit to 0 to transfer the ownership back to processor.
Chk_rdy_bsy :This bit is set to use multi LUN feature.
Multiplane_bit : This bit is use for change write column multi-plane or page program multi-plane operation. If this bit is set multi-plane operation will be done.

3.8.2.2.1 Software Implementation Structure

```
typedef struct desc {
      uint32_t   row_addr;
      uint32_t   column_addr:29;
      uint32_t   target:3
      uint64_t   data_buffer_ptr1;
      uint64_t   data_buffer_ptr2;
      uint64_t   data_buffer_ptr3;
      uint64_t   data_buffer_ptr4;
      uint16_t   data_len1;
      uint16_t   data_len2;
      uint16_t   data_len3;
      uint16_t   data_len4;
      uint64_t   OOB_data_buffer_ptr:52;
      uint16 t   OOB len:12;
      uint16_t   channel_buffer_id;
      uint16_t   dir:1;
      uint16_t   no_data:1;
      uint16_t   ecc:1;
      uint16 t   cmd:6;
      uint32_t   irq_en:1;
      uint32_t   hold: 1;
      uint32_t   dma_cmp: 1;
      uint32_t   desc_id: 8;
      uint32_t   ownedByFpga: 1;
      uint32_t   check_rdy_bdy:1;
      uint32_t   change_pln   :1;
} desc_t;
```

3.8.2.3 DMA Control and Status Register (CSR)

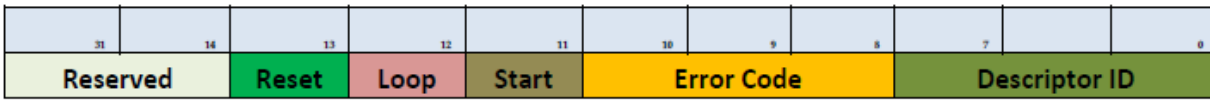


Table 44 Control and Status Register

Descriptor ID: This 8 bit field specifies which descriptor in the Descriptor table is the cause for Interrupt.

Error Code: This three bit field specifies the type of error.

Error Code			Description
0	0	0	No IRQ
0	0	1	Write Time Out
0	1	0	Read Time Out
0	1	1	No Error, User Requested Interrupt
1	0	0	No Error, User Requested Interrupt with hold

Table 45: Error code and status

Start: Start bit to start DMA transaction for the specified Descriptor controller. Software can write a '1' to start DMA.
Loop: If this bit is set to '0' by the software, Descriptor controller stops after reading and processing the 256 descriptors. Else the Descriptor controller starts again to process from descriptor '0'.
Reset: Logic reset, which will reset all the state machines to IDLE and all pointers to initial value.

CSR register is updated from FPGA logic on each Descriptor processing completion.

3.8.2.4 Descriptor Table Size Register

By default descriptor table size will be 16, which is the maximum value. Software can update this size using the Descriptor table size register, there by the maximum number of descriptors is configurable.

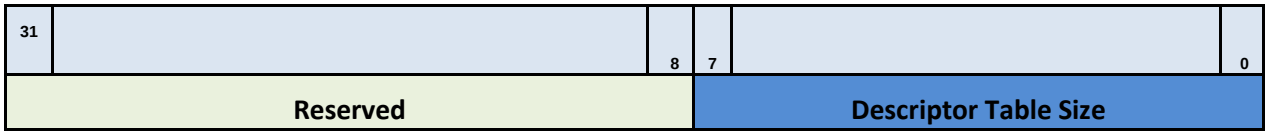


Table 46: Descriptor Table Size Register

3.8.2.5 GPIO Interrupt Registers

NAND Descriptor Interrupt : AF19

IO Doorbell Interrupt : AG19

FIFO Interrupt : AG18

Interrupt flags should be cleared from the software with offset 0x10004 (byte addressing) of PEX2's BAR4 in case of FPGA Image 03.01.00.

3.8.2.6 FPGA Image Versions

Two versions of FPGA images are supported by the NVMe Software such as 02.05.02 and 03.01.00. NAND DMA registers offsets are different for the two mentioned versions.

3.8.2.6.1 Image Version 02.05.02

The Image 02.05.02 Supports Asynchronous mode-5.There are 8 descriptor tables corresponding to 8 NAND chips. The offset shall be added to the base address of PEX4's BAR-2.

	Descriptor structure OCM
Descriptor Table 0	0x0000 - 0x03FF
Descriptor Table 1	0x0800 - 0x0BFF
Descriptor Table 2	0x1000 - 0x13FF
Descriptor Table 3	0x1800 - 0x1BFF
Descriptor Table 4	0x2000 - 0x23FF
Descriptor Table 5	0x2800 - 0x2BFF
Descriptor Table 6	0x3000 - 0x33FF
Descriptor Table 7	0x3800 - 0x3BFF

Table 47: Descriptor Table Offsets(02.05.02)

The DMA control registers offsets are as follows. These offsets shall be added to the base address of BAR4.

	Control Registers
Desc_ctrl_reg 0	0x0000 - 0x003F
Desc_ctrl_reg 1	0x0040 - 0x007F
Desc_ctrl_reg 2	0x0080 - 0x00BF
Desc_ctrl_reg 3	0x00C0 - 0x00FF
Desc_ctrl_reg 4	0x0100 - 0x013F
Desc_ctrl_reg 5	0x0140 - 0x017F
Desc_ctrl_reg 6	0x0180 - 0x01BF
Desc_ctrl_reg 7	0x01C0 - 0x01FF

Table 48: Descriptor Control Register Offsets(02.05.02)

3.8.2.6.2 Image Version 03.01.00

The Image 03.01.00 supports Synchronous mode-1 of NAND. There are 8 descriptor tables corresponding to 8 NAND chips. The offset shall be added to the base address of PEX2's BAR-2.

	Descriptor structure OCM
Descriptor Table 0	0x10000 - 0x103FF
Descriptor Table 1	0x10800 - 0x10BFF
Descriptor Table 2	0x11000 - 0x113FF
Descriptor Table 3	0x11800 - 0x11BFF
Descriptor Table 4	0x12000 - 0x123FF
Descriptor Table 5	0x12800 - 0x12BFF
Descriptor Table 6	0x13000 - 0x133FF
Descriptor Table 7	0x13800 - 0x13BFF

Table 49: Descriptor Table offsets(03.01.00)

The DMA control registers offsets are as follows. These offsets shall be added to the base address of BAR4.

	Control Registers
Desc_ctrl_reg 0	0x4000 - 0x403F
Desc_ctrl_reg 1	0x4040 - 0x407F
Desc_ctrl_reg 2	0x4080 - 0x40BF
Desc_ctrl_reg 3	0x40C0 - 0x40FF
Desc_ctrl_reg 4	0x4100 - 0x413F
Desc_ctrl_reg 5	0x4140 - 0x417F
Desc_ctrl_reg 6	0x4180 - 0x41BF
Desc_ctrl_reg 7	0x41C0 - 0x41FF

Table 50: Descriptor Control Register Offsets(03.01.00)

3.9 Physical Region Page (PRP) Entry

A physical region page (PRP Entry) is a pointer to a physical memory page. PRPs are used as a scatter/gather mechanism for data transfers between the controller and memory. To enable efficient out of order data transfers between the controller and the host, PRP entries should be in a fixed size. The size of the physical memory page is configured by host software in CC.MPS (Controller Configuration

Memory Page Size). The layout of a PRP entry consists of a Page Base Address and an Offset. The size of the Offset field is determined by the physical memory page size configured in CC.MPS.

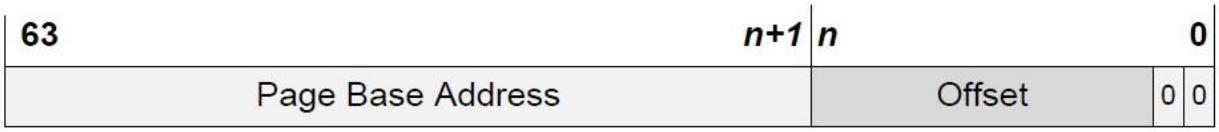


Figure 60: Physical Region Page

3.9.1.1 Physical Region Page (PRP) List

A physical region page list (PRP List) is a set of PRP entries in a single page of contiguous memory. A PRP List describes additional PRP entries that could not be described within the command itself. Any PRP entries described within the command are not duplicated in a PRP List. If the amount of data to transfer requires multiple PRP List memory pages, then the last PRP entry before the end of the memory page shall be a pointer to the next PRP List, indicating the next segment of the PRP List.

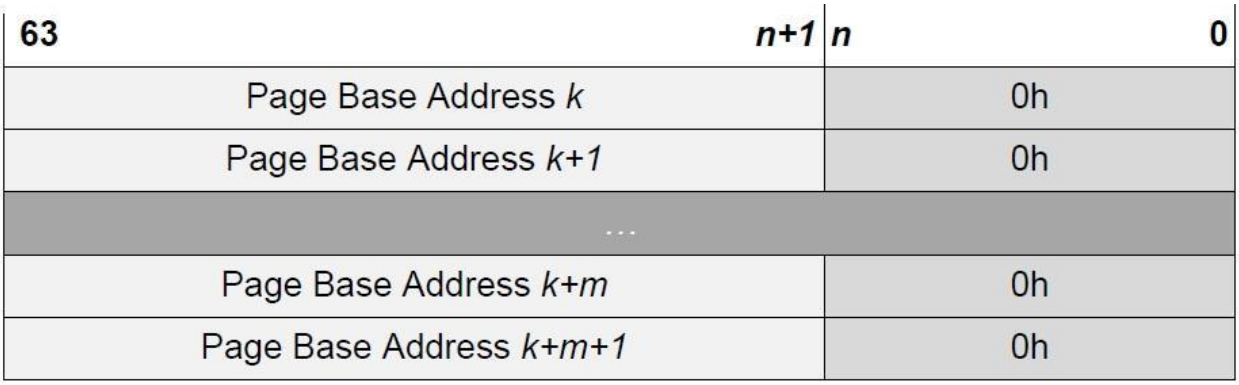


Figure 61: PRP List

3.10 RoCE

3.10.1 Introduction

RoCE (**R**DMA **o**ver **C**onverged **E**thernet) is a network protocol (RoCEv1 – *Network L2,* RoCEv2 – *Network L3*) that allows RDMA operations over a converged Ethernet network. It is basically Infiniband RDMA Verbs supported over Ethernet i.e. IBoE (Infiniband over Ethernet). Here the Infiniband transport layer and network layer are encapsulated in an Ethernet link layer. Network-intensive applications like networked storage or cluster computing need a network infrastructure with a high bandwidth and low latency. The advantages of RDMA over other network application programming interfaces such as Berkeley sockets are lower latency, lower CPU load and higher bandwidth.

3.10.2 iSSD- RoCE

The NVMe iSSD can act as a storage over Network card. Here the NVMe memory subsystem attached to the LS2085A SoC is exposed to any client connected over the network as a SCSI target. RDMA over Ethernet (RoCE) is the protocol over which the SCSI over RDMA (SRP) software stack runs. The 4x10G network ports available in the iSSD card are used for transferring the SCSI data to/from the client. RoCE works only in iSSD mode.

3.10.3 Use case setup

The application for which Soft RoCE would be used initially in this product is to support Storage Area Network over RoCE network protocol. Below diagram shows the hardware and software setup for the same.

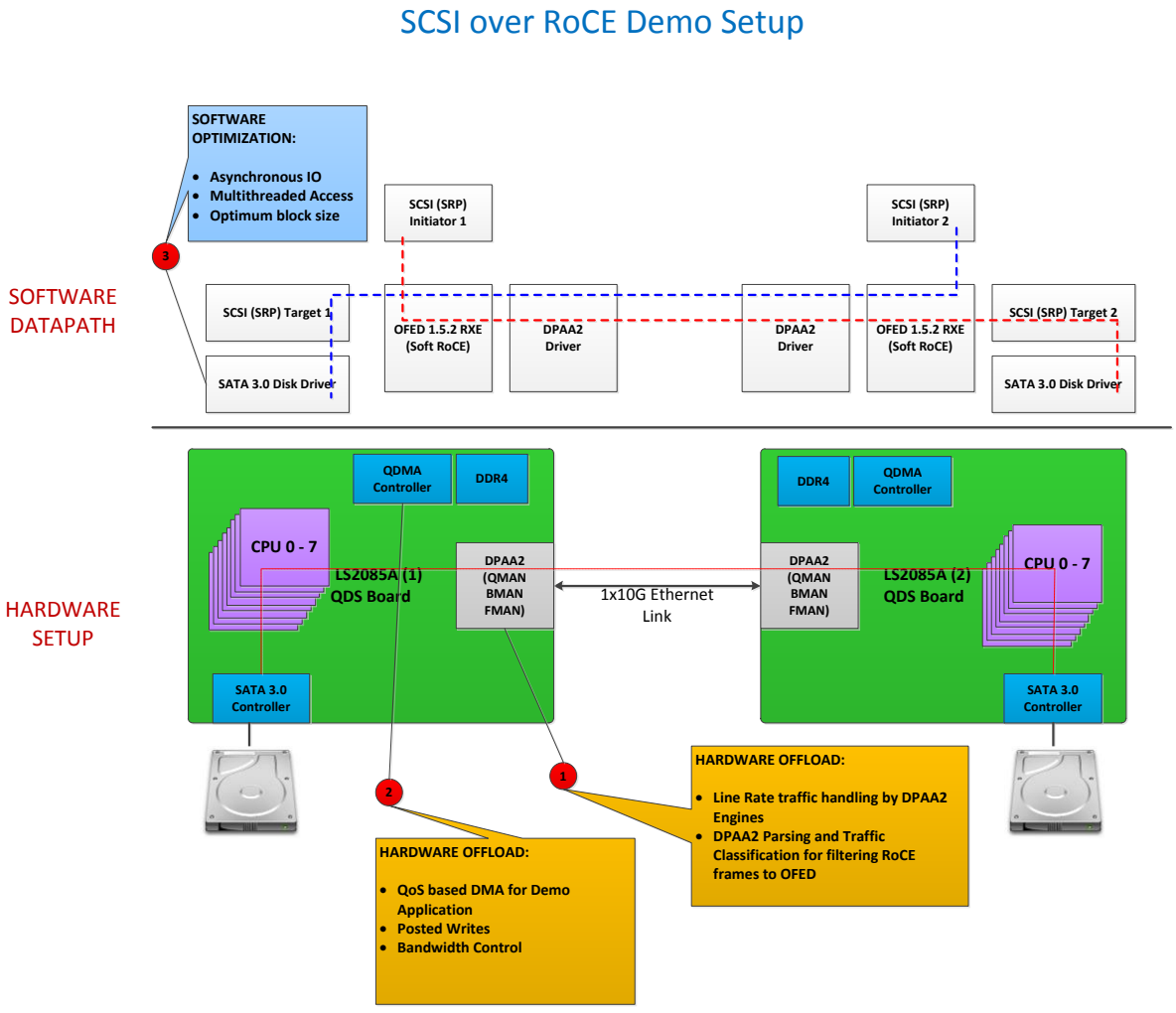


Figure 62 SCSI over RoCE demo setup

3.10.4 Soft RoCE

To support RoCE generally an RNIC is required (RDMA supported NIC) Which does the offloading of RDMA processing. But since the target platform does not have an RNIC infrastructure, we are going to use an emulated version of RNIC, which is supported by a software stack known as **"Soft RoCE"** and is a part of **OFED 1.5.2** (Open fabrics application stack from System Fabrics).

The diagram in the following page shows the general software stack involved in a Soft RoCE.

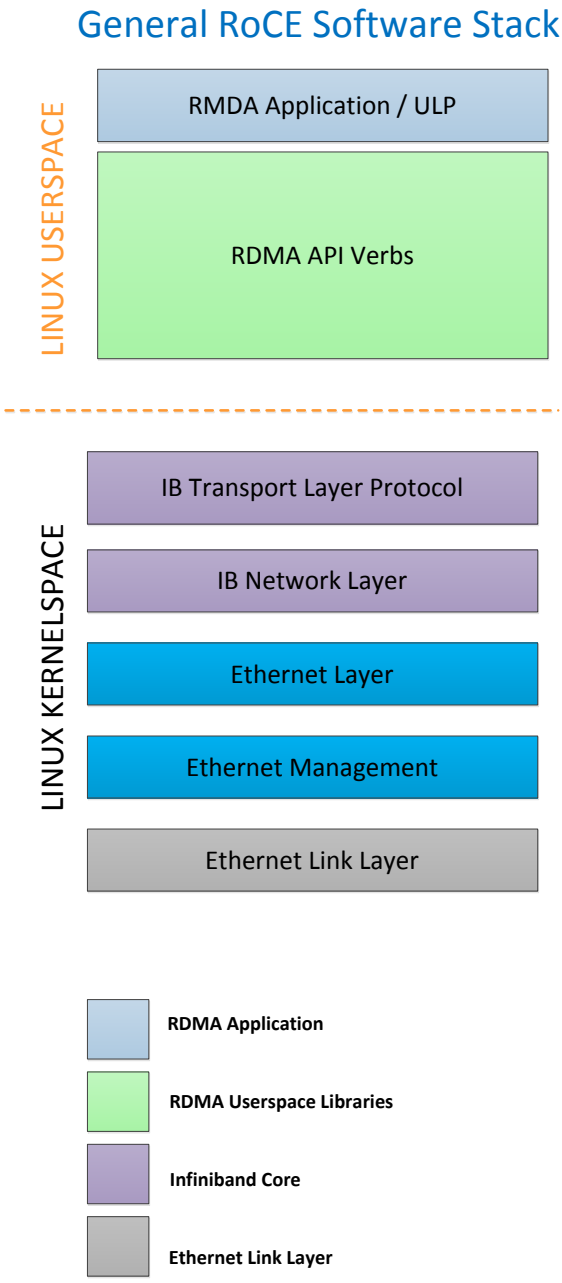


Figure 63: General RoCE Software Stack

3.10.5 Porting Information

The following software components would be ported to LS2085A SDK to support RoCE and its initial application

1. OFED 1.5.2 (With Soft RoCE support)
2. SRP Target & Initiator 3.0.1

NOTE: *Porting details would be added once the porting process is complete.*

3.11 Latency & Performance

The goal of latency and performance measurement is to benchmark various parts of the iSSD software stack (both end to end and subsets). Below diagram shows an overview of the two major data paths that exists in the iSSD system.

- Local Storage DataPath
- Remote Storage DataPath

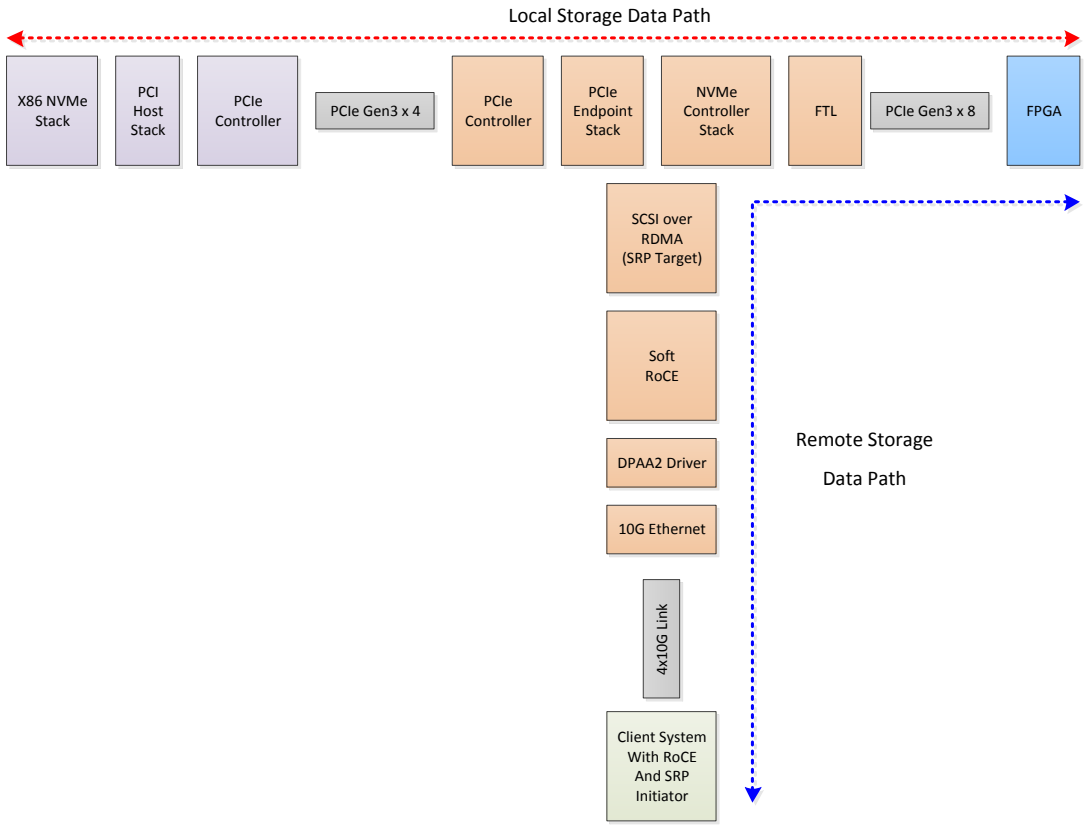


Figure 64: Storage Data path Overview

Initial latency and performance tests would cover the following paths

Over PCIe Bus (Local Storage Data path):

- FPGA Read / Write Latency from LS2
- FPGA Read / Write Latency from x86
- Memory Read / Write Latency from FPGA to LS2 (For reading DMA Descriptors)
- Memory Read / Write Latency from FPGA to x86 (Using DMA, End to End data transfer)
- Memory Read / Write Latency from LS2 to x86 (NVMe – Submission Queue Read and Completion Queue Write)

Over 10G Link (Remote Storage Data path):

- 4x10G Link Performance over RoCE
- Memory Read / Write Latency from Client to FPGA through the SRP Target Stack (End to End data transfer)

APPENDIX A - REFERENCES

Title	Location

APPENDIX B - ACRONYMS AND DEFINITIONS

For Acronyms, Definitions and Terms which may not be listed refer to the NXP Glossary located at http://glossary.NXP.net.

Term	Definition

APPENDIX C –TEMPLATE REVISION SHEET

Template location:
http://compass.NXP.net/livelink/livelink/211576410/Engineering_Design_Document_Template.doc?func=doc.Fetch&nodeid=211576410

Rev Date	Author	Reviewer/Approver	Description
09/06/2011	John Cortell, Chris Freehill		0.1. Added template revision sheet. Initial updates.
09/06/2011	Simon Lang		0.2. Added appendices. Changed header/footer. Formatting changes.
09/15/2011	Simon Lang	Tim Freehill	1.0 formal release