NVMe over Falcon Transport Specification

Revision 0.9

Date Submitted: February 29, 2024
Date Approved: < TBD >

Authors: Prashant Chandra, Rakesh Gautam, Bart Van Assche Google

Table of Contents

1. License

Contributions to this Specification are made under the terms and conditions set forth in Open Web Foundation Contributor License Agreement (“OWF CLA 1.0”) (“Contribution License”) by:

Google

Usage of this Specification is governed by the terms and conditions set forth in the Open Web Foundation Final Specification Agreement (“OWFa 1.0”).

2. Compliance with OCP Tenets

2.1 Openness

The specification complies with the tenet of Openness by empowering the Community with Google’s production learnings to help modernize Ethernet. This includes leveraging production-proven technologies including Carousel, Snap, Swift, PLB, and CSIG that have been openly published previously.

2.2 Efficiency

The specification complies with the tenet of Efficiency through three key insights that achieve low latency in high-bandwidth, yet lossy, Ethernet data center networks. Fine-grained hardware-assisted round-trip time (RTT) measurements with flexible, per-flow hardware-enforced traffic shaping, and fast and accurate packet retransmissions, are combined with multipath-capable and PSP-encrypted Falcon connections. On top of this foundation, Falcon has been designed from the ground up as a multi-protocol transport capable of supporting Upper Layer Protocols (ULPs) with widely varying performance requirements and application semantics. The ULP mapping layer not only provides out-of-the-box compatibility with Infiniband Verbs RDMA and NVMe ULPs, but also includes additional innovations critical for warehouse-scale applications such as flexible ordering semantics and graceful error handling. Last but not least, the hardware and software are co-designed to work together to help achieve the desired attributes of high message rate, low latency, and high bandwidth, while maintaining flexibility for programmability and continued innovation.

2.3 Impact

The specification complies with the tenet of Impact by introducing a new technology that helps the industry modernize Ethernet. Falcon provides a helpful solution to address demanding workloads that have high burst bandwidth, high Operations per second, and low latency in massive scale AI/ML training, High Performance Computing, and real-time analytics.

2.4 Scale

The specification complies with the tenet of Scale by being designed from the ground up to deliver high bandwidth and low latency at scale to achieve low latency in high-bandwidth, yet lossy, Ethernet data center networks. Additionally, it is composed of production-proven technologies delivered at scale including Carousel, Snap, Swift, PLB, and CSIG.

2.5 Sustainability

The specification complies with the tenet of Sustainability by delivering an efficient Ethernet transport technology that minimizes retransmissions and other wasted effort and energy within an Ethernet network.

3. Change Log

Date	Version #	Author	Description
29 FEB 2024	0.9	Prashant Chandra Rakesh Gautam Bart Van Assche	Defines the NVMe over Falcon transport protocol

4. Scope

This specification describes the mapping of the NVMe ULP to the Falcon transport protocol including packet formats, supported operations and error handling modes.

Not in scope are:

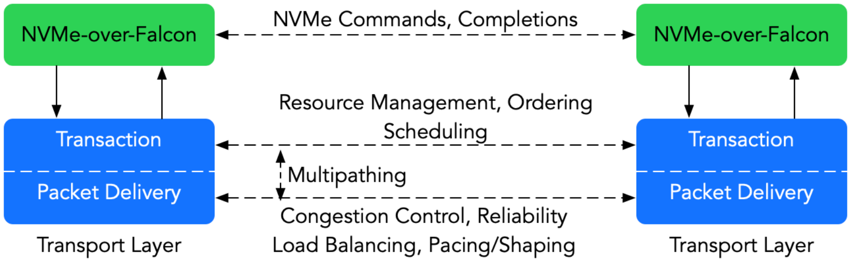
- Details of Falcon protocol.
- Details of applications' use of NVMe or Falcon.
- Details of implementing NVMe/Falcon in software stacks or hardware NICs.

5. Overview

This specification describes the mapping of the NVMe ULP to the Falcon transport protocol including packet formats, supported operations and error handling modes. The NVMe ULP is defined by the NVMe Specification and the mapping of NVMe over the Falcon transport defines how NVMe Read and Write commands are mapped to Falcon transactions to support disaggregated storage applications.

6. Protocol Architecture

6.1 Protocol Layers

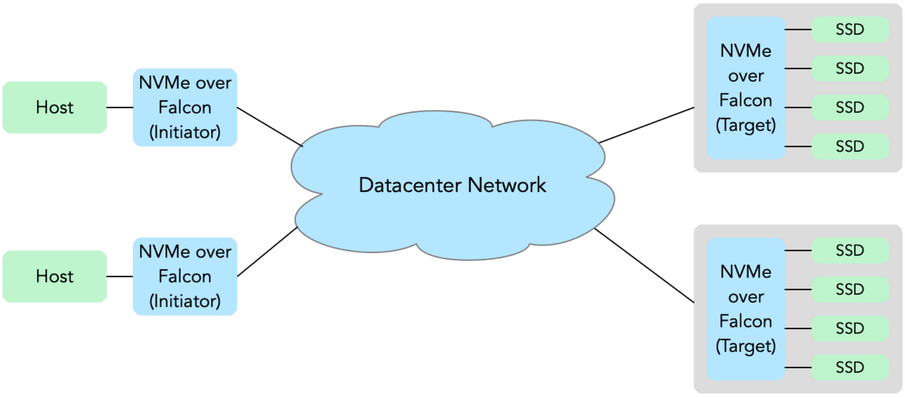


The figure above shows the protocol layering of NVMe as an upper layer protocol on top of Falcon. Falcon itself is composed of two sublayers: transaction sublayer and the packet delivery sublayer. The transaction sublayer primarily deals with ULP transactions and is responsible for resource allocation and transaction ordering. The packet delivery sublayer primarily deals with network packets and is responsible for reliable delivery and congestion control.

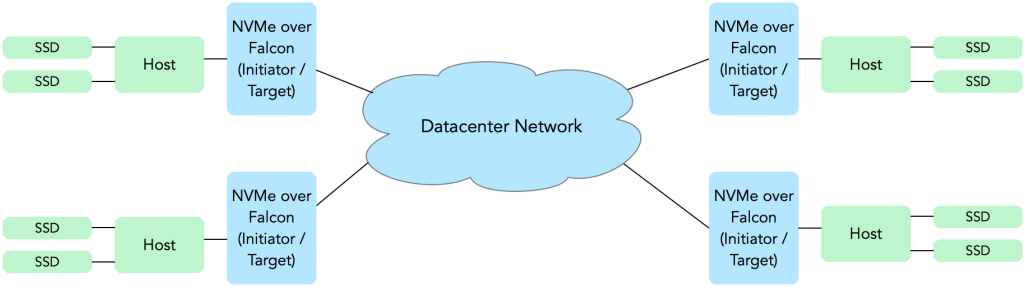
The NVMe-over-Falcon layer defines the mapping of NVMe commands and completions to Falcon packets and the wire-protocol used to communicate between NVMe-over-Falcon peers.

6.2 System Architecture

The NVMe over Falcon architecture defines two roles: initiator and target. The initiator and target roles may be implemented in foundational NICs, smart NICs or storage SoCs. A NVMe-over-Falcon initiator interacts with a software NVMe driver through the standard NVMe submission queue / completion queue interface. A NVMe-over-Falcon target interacts with a storage system such as SSDs, HDDs or software defined storage. An implementation may simultaneously take on the role of an initiator and a target.

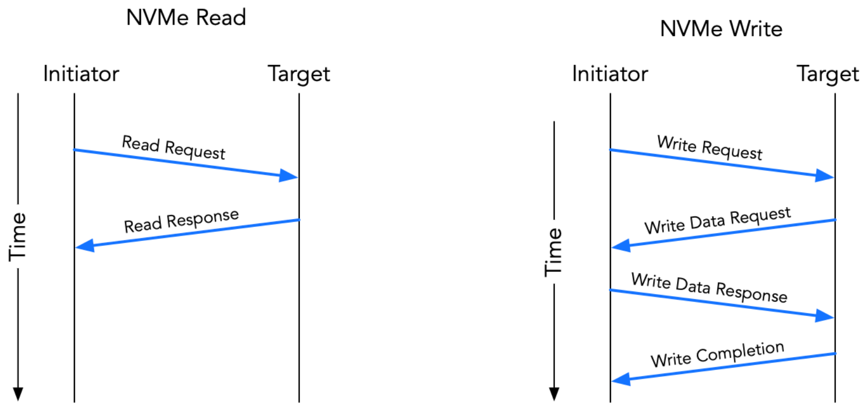

Host
NVMe over Falcon (Initiator)
Host
NVMe over Falcon (Initiator)
Datacenter Network
NVMe over Falcon (Target)
SSD
SSD
SSD
SSD
NVMe over Falcon (Target)
SSD
SSD
SSD
SSD

The figure above shows a system architecture where SSDs are disaggregated from hosts and packaged together in a flash tray that connects directly to the datacenter network. In this architecture the initiator is typically implemented in the NIC that connects to the host machine and the target is typically implemented in a storage SoC that connects to the flash tray.


SSD
SSD
Host
NVMe over Falcon (Initiator / Target)
SSD
SSD
Host
NVMe over Falcon (Initiator / Target)
Datacenter Network
NVMe over Falcon (Initiator / Target)
Host
SSD
SSD
NVMe over Falcon (Initiator / Target)
Host
SSD
SSD

The figure above shows a system architecture where SSDs are locally attached to hosts. In this architecture the NIC connected to the host plays the role of both an initiator and a target.

6.3 Communication Protocol



The high level end-to-end flow of NVMe Read and NVMe Write commands is shown in the figure above. For a Read command, the initiator sends one or more Read Request packets to the target and the target returns a Read Response packet containing the read data for each Read Request packet it receives from the initiator.

For a write command, the initiator sends one or more Write Request packets to the target. The target is responsible for pulling the write data from the initiator by sending a Write Data Request packet to the initiator for every Write Request packet it receives. The initiator then responds with a Write Data Response packet which contains the write data in the payload. When the target receives the Write Data Response packet(s) it submits a NVMe Write command to the storage device and upon receiving a write completion from the device, returns a Write Completion packet for every Write Data Response packet it receives from the initiator.

6.4 Virtual NVMe Devices

The NVMe-over-Falcon initiator presents Virtual NVMe devices to the host. These Virtual NVMe devices can be created on demand by a storage control plane. Just like physical NVMe SSDs, Virtual NVMe devices export the standard PCIe config space and MMIO register interface defined by the NVMe specification. The difference between virtual NVMe devices and physical NVMe SSDs is that the virtual devices do not contain flash memory and instead access flash memory on remote SSDs across the network in response to I/O commands posted to the submission queues.

Virtual NVMe devices are constructed by allocating flash memory from one or more physical SSDs attached to NVMe-over-Falcon targets in allocation units called chunks. A *chunk* is much larger than a logical block (for e.g. 32GB, 64GB, 128GB, etc.) and chunks from multiple targets can be arbitrarily interleaved to create the LBA space of a virtual NVMe device. The allocation of chunks and the mapping and translation between a LBA of the virtual NVMe device at the initiator, and the corresponding LBA on a physical SSD attached to a target is described in a later section.

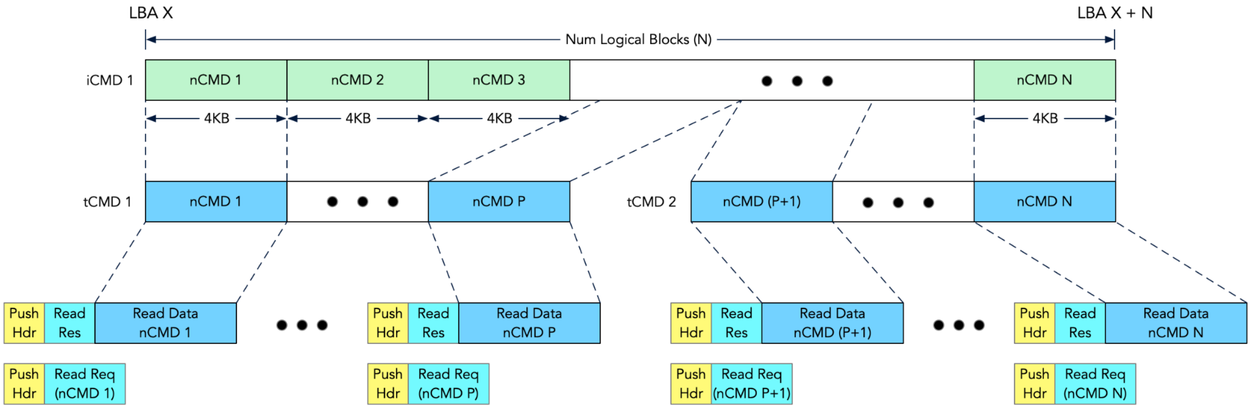
Virtual NVMe devices support multiple namespaces and I/O queue pairs (submission and completion queues) just like physical NVMe SSDs. A namespace on the virtual NVMe device maps to a namespace on the remote physical SSD. However, there is no direct mapping between a queue pair at the virtual NVMe device and a queue pair at the remote physical SSD. Virtual NVMe devices also export an admin queue pair. Commands posted on the admin queue pairs at the initiator are processed in software and typically will trigger control plane operations and may result in the creation of namespaces or I/O submission queues at the initiator and/or target sides.

6.5 Command Segmentation and Reassembly

Falcon transactions can be at most MTU-sized. Therefore, the NVMe-over-Falcon layer is responsible for segmenting large NVMe commands posted by software into MTU-sized Falcon transactions and aggregate individual completions received from Falcon into the NVMe command-level completion back to software.

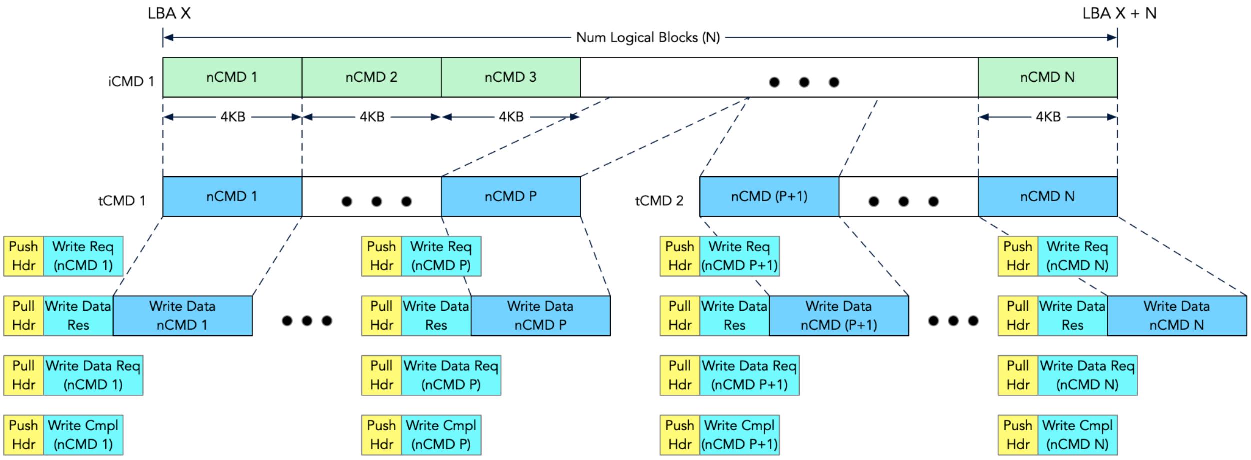
The term *iCMD* is used to denote the NVMe command submitted by the host (or guest VMs) on the initiator side.   NVMe Read and Write iCMDs can be of any size up to the Maximum Data Transfer Size (MDTS) supported by the NVMe controller device presented by the initiator. The term *tCMD* is used to denote the NVMe command submitted by the target to a directly connected SSD or software emulated block device. In the NVMe-over-Falcon architecture, an iCMD may be split across multiple targets and hence result in multiple tCMDs due to LBA allocation and striping described in the next section. A tCMD can also be of any size up to the MDTS supported by the attached storage device. It is assumed that control plane software ensures that the MDTS at the initiator side does not exceed the MDTS on the target storage device.

On the initiator side, NVMe Read and Write commands are segmented into 4KB sized units called *nCMDs* for transfer across the network. The segmentation ensures that each nCMD can map to a single MTU-sized Falcon transaction. NVMe-over-Falcon requires the network MTU to be large enough to support the 4KB data + metadata + packet headers. On the target side, nCMDs received from the initiator are reassembled into tCMDs before being presented to the storage device. The target generates completions per nCMD back to the initiator upon receiving a completion to the tCMD from the storage device. The initiator is responsible for tracking individual nCMD completions and returning a completion back to host software when the entire iCMD is completed.



The segmentation and reassembly of a NVMe Read iCMD is shown in the figure above. In the example shown above, the iCMD is split across two targets and results in two tCMDs. Each tCMD consists of multiple nCMDs. The NVMe-over-Falcon wire protocol sends a Read Request packet for every nCMD from the initiator to the target. The target returns a Read Data packet containing the read response payload for every Read Request received from the initiator.

The segmentation and reassembly of a NVMe Write iCMD is shown in the figure below. Similar to the Read example, the iCMD is split across two targets resulting in two tCMDs. The initiator sends a Write Request packet for every nCMD from the initiator to the target. The target pulls the write data for the nCMD by sending a Write Data Request packet back to the initiator. The initiator then returns the write data by including it in a Write Data Response packet back to the target. After the write operation is complete on the target side, the target sends a Write Completion packet for each Write Request packet back to the initiator.

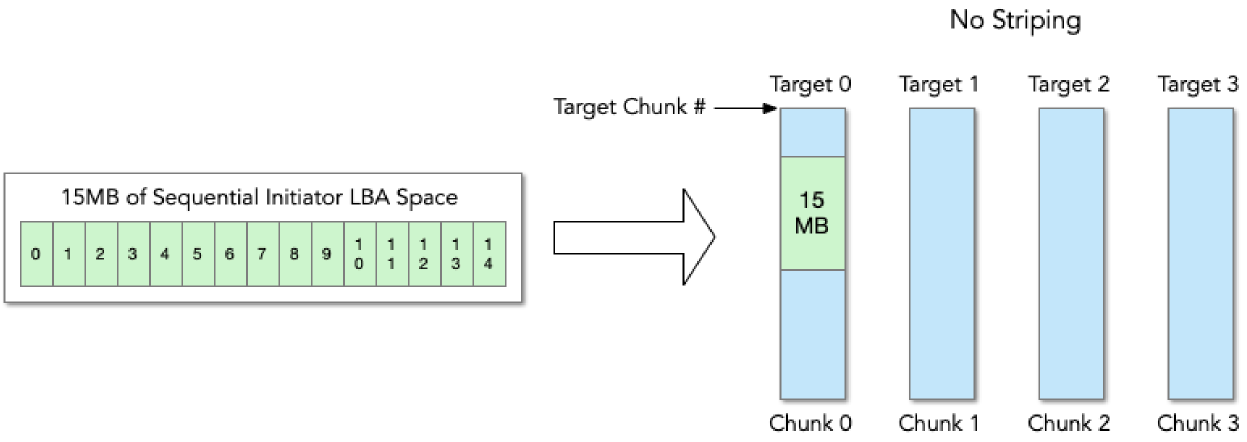


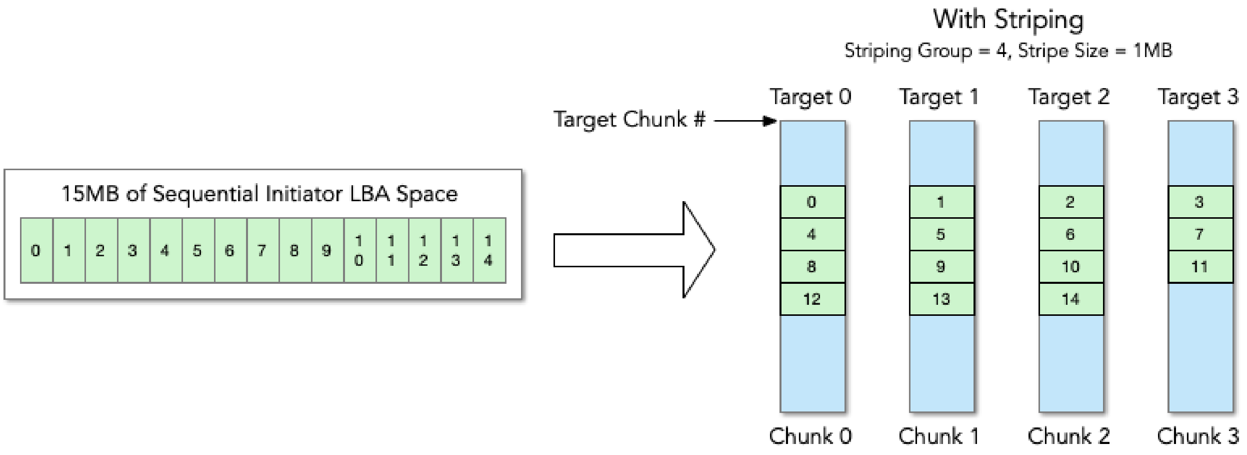
6.6 Mapping Initiator LBA to Target LBA

As stated earlier in the document, the unit of block allocation is called a chunk. A chunk is typically significantly larger than the block size. For e.g. the typical block size of a NVMe SSD is 4 KB whereas a typical chunk size could be on the order of 32 GB, 64 GB or 128 GB. In the NVMe-over-Falcon architecture, the chunk size must be a multiple of 1GB and can be configured per NVMe namespace.

A virtual NVMe device's flash storage can be composed of chunks from multiple target SSDs. Chunks from multiple target SSDs may be interleaved in an arbitrary fashion within the logical block address space of the virtual NVMe device. Therefore, logical block addresses (LBA) of the virtual NVMe device must be translated at the initiator to logical block addresses of the storage target. This translation is managed by an LBA translation table described in the next section.

6.6.1 Block Allocation and Striping





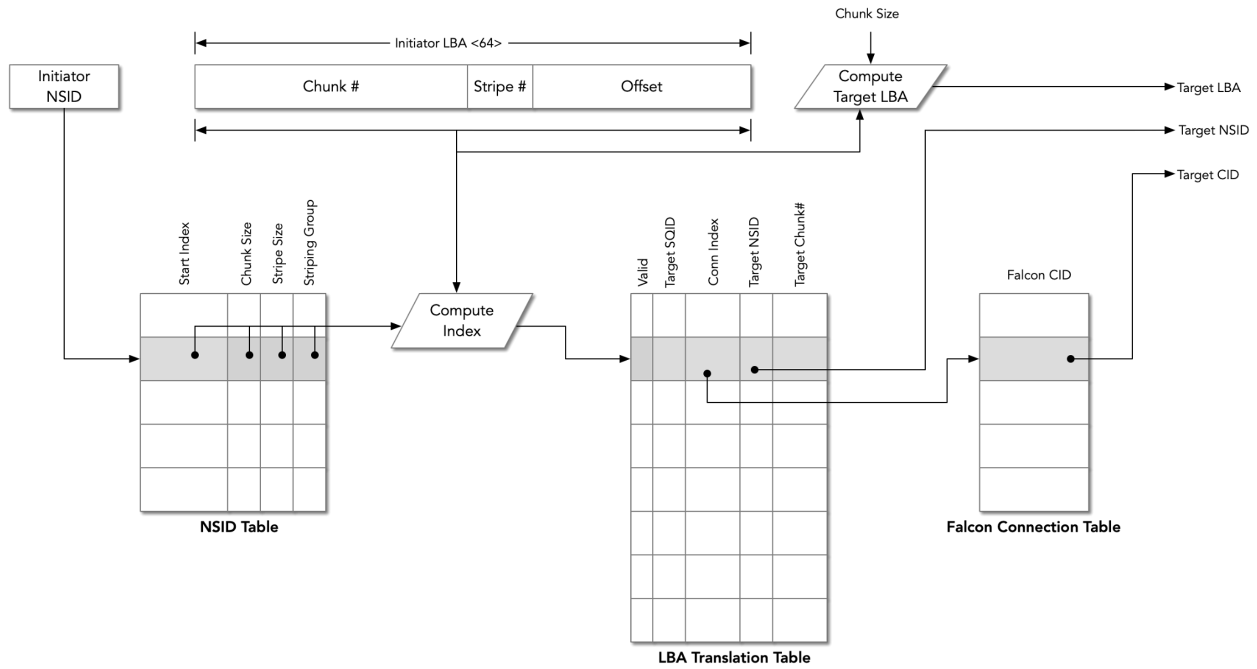
The LBA translation process supports RAID-0 striping of accesses across multiple targets. Striping improves the read and write bandwidth by parallelizing IOs across multiple storage targets. A *striping group* describes the number of segments striped together. In the NVMe-over-Falcon architecture, the striping group can be configured per NVMe namespace and must be a power of. The value 1 indicates no striping. The *stripe size* indicates the amount of LBA space that can be accessed sequentially within a given chunk.   The stripe size can be configured per NVMe namespace and can be any value in units of 256KB. The stripe size must be a multiple of the logical block size. The chunk size must be a multiple of the stripe size.

The RAID-0 striping is implemented such that for each sequential N chunks (where N is the striping group) stripe sized blocks data adjacent in LBA space go to different chunks (each chunk can be allocated on a different target). For example, if we assume that the stripe size is 1MB and the striping group is 4 (N=4), the first 1 MB block will go to the first chunk, the second 1 MB block to the second chunk, etc. Block N+1 will go again to the first chunk adjacent to the first block, N + 2 block to the second chunk and so on. This example is illustrated in the figure above.

6.6.2 LBA Translation

All NVMe commands received by an initiator must go through a LBA translation pipeline to map the initiator LBA to a specific target appliance / host and to compute the corresponding target LBA. The LBA translation pipeline is shown in the figure below and consists of 3 tables described below:

- The NSID table contains configuration parameters for a namespace.
- The LBA translation table contains parameters mapping an initiator chunk to a target chunk.
- The Falcon connection table contains the Falcon connection identifiers for connections from the initiator side to the target side.



The LBA translation process involves the following steps:

1. The initiator namespace ID is used as an index into the NSID table to retrieve various parameters associated with the namespace described below.
2. The index into the LBA translation table is computed based on the parameters obtained from the matching row in the NSID table. A lookup of the LBA translation table is performed to retrieve parameters specific to the initiator chunk.
3. The target LBA is computed based on the parameters obtained from the matching row in the LBA translation table.
4. The target Falcon CID is obtained by indexing into the Falcon connection table using the connection index obtained from the LBA translation table.

Each row of the NSID table must contain the information specified in the table below.

Field	Description
Start Index	This field encodes the first row of the LBA translation table that contains LBA translations for the global namespace specified by initiator NSID.
Chunk Size Shift (C)	This field encodes the chunk size as 2^C * 1GB.
Stripe Size Shift (S)	This field encodes the stripe size as 2^S * 256KB.
Striping Group Shift (N)	The striping group is a power of 2. This field encodes the striping group as 2^N.

The fields from the matching row of the NSID table are used to compute the index into the LBA translation table. The pseudocode for this calculation is given below.

```
ComputeLBATableIndex(initiator_lba, start_index, chunk_size_shift,
                     stripe_size_shift, striping_group_shift,
                     block_size_shift) {
  // Calculate the chunk_num and stripe_num.
  chunk_start = (chunk_size_shift + striping_group_shift + 30)
                - block_size_shift;
  stripe_start = (stripe_size_shift + 18 - block_size_shift);
  chunk_num = initiator_lba >> chunk_start;
  stripe_num = (initiator_lba >> stripe_start) &
               ((1 << (chunk_start - stripe_start)) - 1);

  // Compute index into LBA translation table.
  index = (chunk_num << striping_group_shift) |
          (stripe_num & ((1 << striping_group_shift) - 1));
  index += start_index;
}
```

Each row of the LBA translation table contains the information specified in the table below.

Field	Description
Valid	This flag must be set to 1 if the row contains a valid LBA translation entry.
Target SQID	This field encodes the submission queue id to be used at the target side.
Connection Index	This field encodes the index into the GRT connection table that specifies the GRT CID to be used to reach the target.
Target NSID	This field encodes the NSID of the target chunk. This field is only relevant for the NLF and local SSD use cases.
Target Chunk#	This field encodes the target chunk #. This field is used to compute the target LBA.

The fields from the matching row of the LBA translation table are used to compute the target LBA using the following equation:

```
ComputeTargetLBA(initiator_lba, chunk_size_shift, stripe_size_shift,
                 striping_group_shift, block_size_shift,
                 target_chunk_number) {
  // Calculate the chunk_num and stripe_num.
  chunk_start = (chunk_size_shift + striping_group_shift + 30)
                - block_size_shift;
  stripe_start = (stripe_size_shift + 18 - block_size_shift);
  chunk_num = initiator_lba >> chunk_start;
  stripe_num = (initiator_lba >> stripe_start) &
               ((1 << (chunk_start - stripe_start)) - 1);
  offset_in_stripe = initiator_lba & ((1 << stripe_start) - 1);

  // Compute the target LBA
  target_lba = (target_chunk_num << (chunk_start - striping_group_shift)) |
               ((stripe_num >> striping_group_shift) << stripe_start) |
                 offset_in_stripe;
}
```

Each namespace must be allocated a set of contiguous rows in the LBA translation table. Each row represents a chunk of LBA space and each chunk can be allocated on a different target appliance / host.

Online resizing of namespaces can be done by allocating a new set of contiguous rows in the LBA table, updating the Start Index in the NSID table and then setting the valid bit to 0 for all rows previously used by that namespace.

6.6.3 Command Splitting

The data transferred by the original NVMe command submitted by a host or guest VM may straddle a chunk or stripe boundary. When this occurs, the original NVMe command must be split into multiple target CMDs (tCMDs). It is assumed that the combination of the maximum data transfer size supported by the virtual NVMe device on the initiator side and the minimum stripe size (256KB) ensures that the original command can be split at most into two tCMDs. When a command is split, the initiator must correctly compute the starting LBA and the data transfer size in number of logical blocks for each of the two tCMDs. There are two cases to consider for command splitting and are described in the following sections.

The pseudocode to determine whether a command is split and whether it is aligned or unaligned is given below.

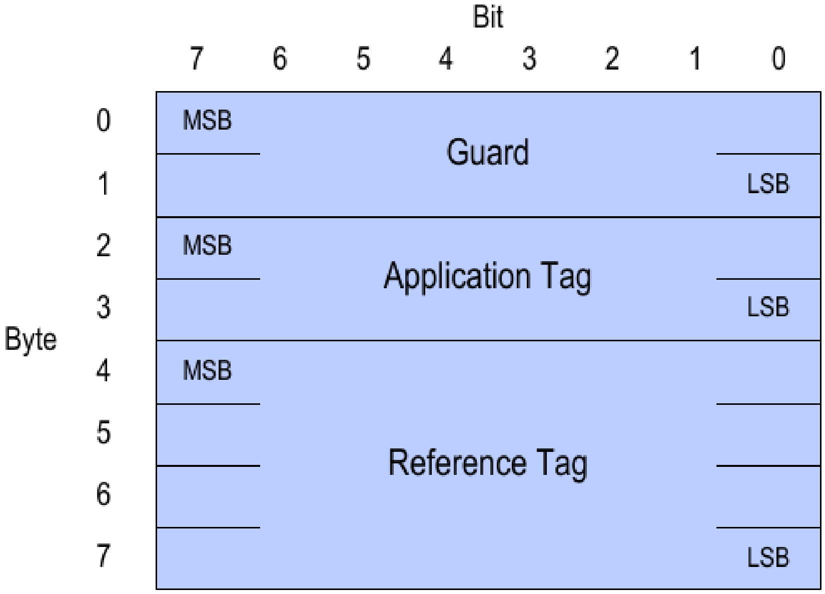
```
IsSplitCommand(cmd_first_lba, cmd_nlb, chunk_size_shift,
               stripe_size_shift, striping_group_shift, block_size_shift) {
  // Compute the last LBA of this CMD.
  cmd_last_lba = cmd_first_lba + cmd_nlb;

  // Compute the first and last chunk and stripe numbers
  chunk_start = ((chunk_size_shift + striping_group_shift + 30) -
                block_size_shift);
  stripe_start = (stripe_size_shift + 18 - block_size_shift);
  first_stripe_number = (cmd_first_lba >> stripe_start) &
                        ((1 << (chunk_start - stripe_start)) - 1);
  last_chunk_number = cmd_last_lba >> chunk_start;
  last_stripe_number = (cmd_last_lba >> stripe_start) &
                       ((1 << (chunk_start - stripe_start)) - 1);
```

```
   // Determine if the CMD is split across segment or stripe boundary
   is_split_cmd = (first_stripe_number != next_stripe_number);

   // Determine the NLB for first and second tCMD
   if (is_split_cmd) {
      // Compute the boundary of the stripe
      boundary_lba = (last_chunk_number << chunk_start) |
                     (last_stripe_num << stripe_start);

      nlb_1 = boundary_lba - cmd_first_lba;
      nlb_2 = cmd_last_lba - boundary_lba + 1;
   }
}
```

6.7 End-to-End Encryption and Data Protection

NVMe-over-Falcon supports the NVMe Protection Information (PI) feature which provides end-to-end data protection from the application to the NVM media and back.

Figure 256: Protection Information Format



The NVMe PI feature stores 8B of Protection Information per logical block. The Protection Information format is shown in the figure above and is contained in the metadata associated with each logical block. The Guard field contains a CRC-16 computed over the logical block data. The Application Tag is an opaque data field not interpreted by the NVMe controller and that may be used to disable checking of protection information. The Reference Tag associates logical block data with an address and protects against misdirected or out-of-order logical block transfer. Like the Application Tag, the Reference Tag may also be used to disable checking of protection information.

NVMe-over-Falcon supports end-to-end data encryption and protection. Data encryption is provided by 256-bit AES-XTS and can be enabled on the initiator side on a per-namespace basis. Protection Information support is based on the NVMe specification and can also be enabled on the initiator side on a per-namespace basis. The 8B of protection information on the initiator side can be provided by the host (if the host chose the namespace format that includes metadata) or inserted by the NVMe controller logic (if the host does not provide metadata) implemented in the initiator. If data encryption is enabled together with protection information, the 8B of protection information on the initiator side is also encrypted.

The initiator-side protection information is referred to as the *inner PI*. Since the inner PI may be encrypted (if the initiator namespace has encryption enabled), NVMe-over-Falcon initiator adds a second layer of protection called *outer PI*. The outer PI is also computed at the initiator side over the data block and inner PI and is not encrypted. This outer PI is stored along with the data block and inner PI on the target SSD and the target namespace is formatted in a way that allows the target SSD to verify the outer PI as described below. If inner PI is enabled, NVMe-over-Falcon requires outer PI to be present regardless of whether encryption is enabled. Hence the target namespaces must always be configured for 16B of metadata when PI is enabled.

The target side namespaces must be formatted as follows:

- The namespace must be configured for 16B of metadata.
- The *Protection Information Location (PIL)* bit must be set to 0. This means that protection information occurs at the end of the metadata.
- The *Protection Type (PI)* field must be set to 1. This means that the guard tag is used as a CRC, that the reference tag will contain the lowest 32 bits of the LBA and also that the application tag is verified upon reading.
- The *Metadata Settings (MSET)* bit must be cleared. This means that metadata and data are submitted as separate buffers to the NVMe controller.
- An *LBA Format (LBAF)* must be selected with a metadata size that is at least sixteen bytes.

The initiator computes the CRC-16 over the ciphertext data and the inner 8B of metadata and places that value in the guard field of the outer 8B of metadata. The initiator must do the following before it sends a Read or Write nCMD to the target:

1. The PRACT bit is cleared to avoid recomputation of the guard field by the SSD attached to the target. The PRCHK bit 2 is set to ‘1’ to enable protection information checking of the Guard field.

2. The 32 LSB bits of the target LBA are placed in the Reference Tag field. The PRCHK bit 0 is set to ‘1’ to enable protection information checking of the Reference Tag field.
3. The LSB 16 bits of the target LBA are placed in the Application Tag field. The PRCHK bit 1 is set to ‘1’ to enable protection information checking of the Application Tag field.
4. Set EILBRT to 32 LSB bits of the target LBA, ELBATM to 0xFFFFh and ELBAT to 16 LSB bits of the target LBA in Read commands.
5. Set ILBRT to 32 LSB bits of the target LBA, LBATM to 0xFFFFh and LBAT to 16 LSB bits of the target LBA in Write commands.

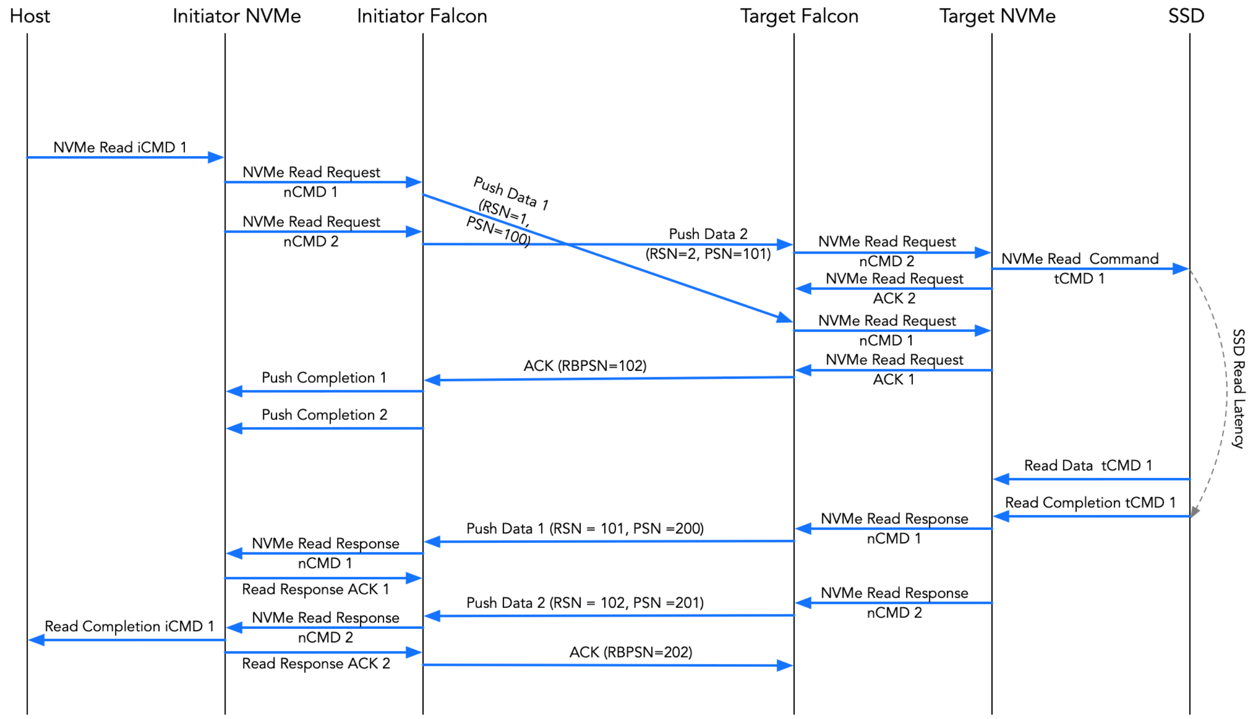
Implementations may allow for protection information checking to be disabled globally on the target side. When protection information is disabled, the initiator must do the following before it sends a Read or Write nCMD to the target:

1. Clear PRCHK bits 0, 1 and 2 to disable the checking of the Guard, Application Tag and Reference Tag fields.
2. Set EILBRT to 0xFFFFFFFFh and ELBATM and ELBAT to 0xFFFFh in Read commands.
3. Set ILBRT to 0xFFFFFFFFh and LBATM and LBAT to 0xFFFFh in Write commands.

7. NVMe Command Flows

The following sections describe the detailed NVMe command flow between the initiator and the target. The command flows described below show the life of a NVMe Read and a NVMe Write command. In the flows described below, it is assumed that the iCMD is split into two nCMDs at the initiator side and both nCMDs are delivered to the same target resulting in a single tCMD. In the example flows shown below, NVMe-over-Falcon is assumed to use an unordered Falcon connection. Falcon unordered mode provides the best performance for NVMe-over-Falcon since the NVMe protocol is itself unordered allowing commands to be completed in any order irrespective of the order in which they were submitted by software.

7.1 NVMe Read Flow

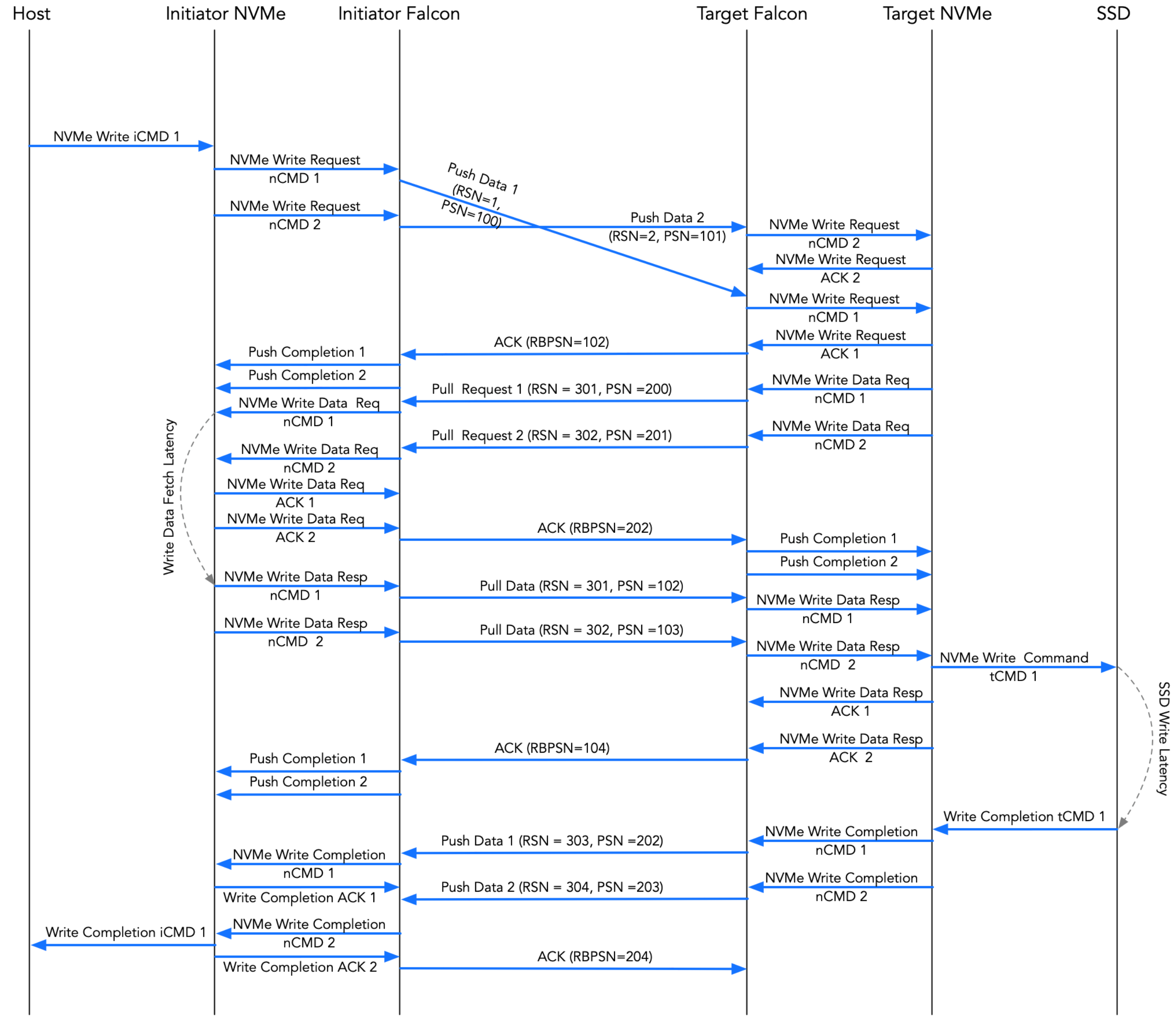


The life of a NVMe Read command is shown in the figure above. The connection is assumed to be an unordered connection. The following sequence of operations are performed at the initiator and target sides:

1. Software posts a NVM read command (iCMD1 for a 8KB read in this example) to a submission queue. This results in the initiator NVMe issuing two NVMe Read Requests (nCMD1 and nCMD2) to Falcon. The two requests are created because each NVMe Read Request is limited to 4KB in length and iCMD1 is a read for 8KB of data.
2. The initiator creates two push data packets (with RSN=1 and RSN=2) and transmits them to the target. The packet delivery sublayer assigns PSN=100 and PSN=101 to the two pull request packets.
3. The two push request packets are reordered by the network and arrive out of order at the target. Since the Falcon connection is unordered, the Falcon target delivers them to the NVMe-over-Falcon engine in the order in which they are received. The NVMe-over-Falcon target triggers the generation of the ACK packets acknowledging the receipt of the two push data packets. In the figure above, a single coalesced ACK is sent from the target to the initiator.

4. The NVMe-over-Falcon target submits the NVMe Read command (tCMD1) to the SSD when the first Read Request is received (for nCMD2).
5. When the SSD delivers the read completion, the target creates two NVMe Read Response packets and transmits them to the initiator. These packets map to push data packets at the Falcon transaction layer (with RSN=101 and RSN=102). The packet delivery sublayer assigns PSN=200 and PSN=201 to these push data packets.
6. The Falcon initiator receives the two push data packets and delivers them to the NVMe-over-Falcon initiator. The NVMe-over-Falcon initiator triggers the generation of ACK packets by the Falcon packet delivery sublayer acknowledging the receipt of the push data packets.
7. After the NVMe-over-Falcon initiator receives both NVMe Read Responses (for nCMD1 and nCMD2), it creates a read completion for iCMD1 and posts it to the completion queue.

7.2 NVMe Write Flow



The life of a NVMe Write command is shown in the figure above. The connection is assumed to be an unordered connection. The following sequence of operations are performed at the initiator and target sides:

1. Software posts a NVM write command (iCMD1 for a 8KB write in this example) to a submission queue. This results in the initiator NVMe issuing two NVMe Write Requests (nCMD1 and nCMD2) to Falcon. The two requests are created because each NVMe Write Request is limited to 4KB in length and iCMD1 is a write with 8KB of data.

2. The initiator creates two push data packets (with RSN=1 and RSN=2) and transmits them to the target. The packet delivery sublayer assigns PSN=100 and PSN=101 to the two pull request packets.
3. The two push request packets are reordered by the network and arrive out of order at the target. Since the Falcon connection is unordered, the Falcon target delivers them to the NVMe-over-Falcon engine in the order in which they are received. The NVMe-over-Falcon target triggers the generation of the ACK packets acknowledging the receipt of the two push data packets. In the figure above, a single coalesced ACK is sent from the target to the initiator.
4. The NVMe-over-Falcon target generates a Write Data Request packet for each nCMD and sends it to the initiator. These packets are mapped to pull request packets by Falcon and transmitted to the initiator with RSN=301 and RSN=302 (PSN=200 and PSN=201).
5. The Write Data Packets are received at the NVMe-over-Falcon initiator. The initiator triggers the generation of ACK packets acknowledging the receipt of the two pull request packets. The initiator fetches the write data associated with nCMD1 and nCMD2 and generates two Write Data Response packets and sends them to the target. The Write Data Response packets are mapped to pull data packets at the Falcon layer and are transmitted to the target with PSN=102 and PSN=103.
6. The NVMe-over-Falcon target receives the two Write Data Response packets and submits a NVMe Write command (tCMD1) to the SSD. It also acknowledges the receipt of the pull data packets which causes the Falcon target to send an ACK packet back to the initiator.
7. When the SSD delivers the write completion, the target creates two Write Completion packets and sends it to the initiator. These Write Completion packets are mapped to push data packets at the Falcon layer and are transmitted to the initiator with RSN=303 and RSN=304 (PSN=202 and PSN=203).
8. The Falcon initiator receives the two push data packets and delivers them to the NVMe-over-Falcon initiator. The NVMe-over-Falcon initiator triggers the generation of ACK packets by the Falcon packet delivery layer acknowledging the receipt of the push data packets.
9. After the NVMe-over-Falcon initiator receives both NVMe Write Completions (for nCMD1 and nCMD2), it creates a write completion for iCMD1 and posts it to the completion queue.

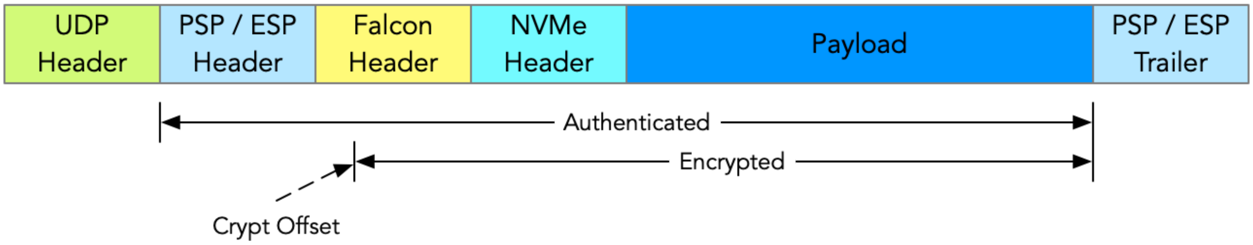
Implementations can choose to send Write Completion packets for all nCMDs except the last one without waiting for the write completion to come back from the SSD. The last Write Completion packet must be sent only after the write completion is received from the SSD. At the initiator side, NVMe-over-Falcon must wait for all nCMD Write Completion packets to be received before generating a write completion back to the host.

8. Wire Protocol

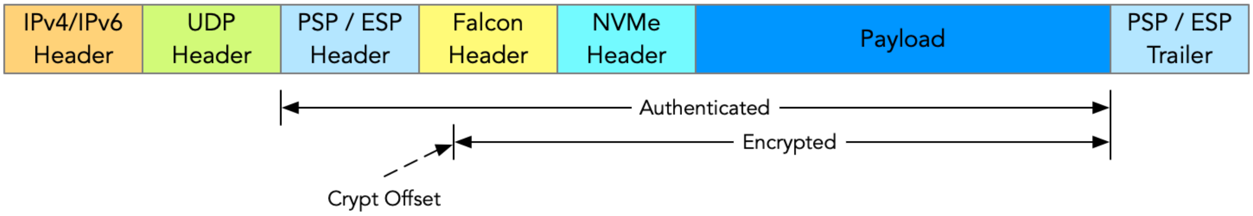
The NVMe-over-Falcon wire protocol defines the messages exchanged between the initiator and target systems in support of NVMe commands and completions. The wire protocol leverages Falcon for reliable transfer of messages across the network and uses the unordered reliable connection mode of Falcon.

8.1 Packet Format

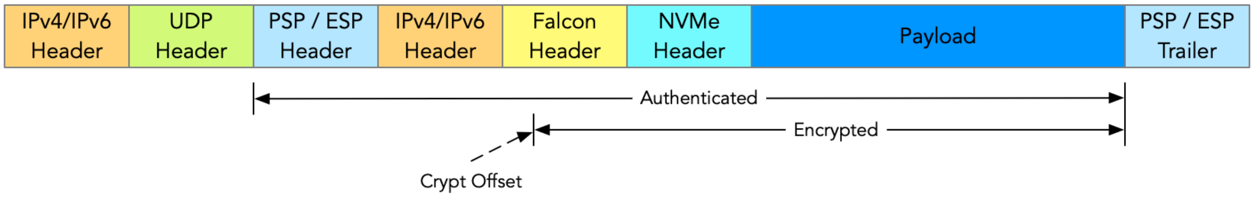
The packet format used by NVMe-over-Falcon is shown in the figure below. NVMe-over-Falcon packets are encapsulated within the payload of a Falcon packet. NVMe-over-Falcon defines multiple packet types for communication between the initiator and target that are listed in the table below and are described in detail in subsequent sections.



8.1.1 Transport Mode



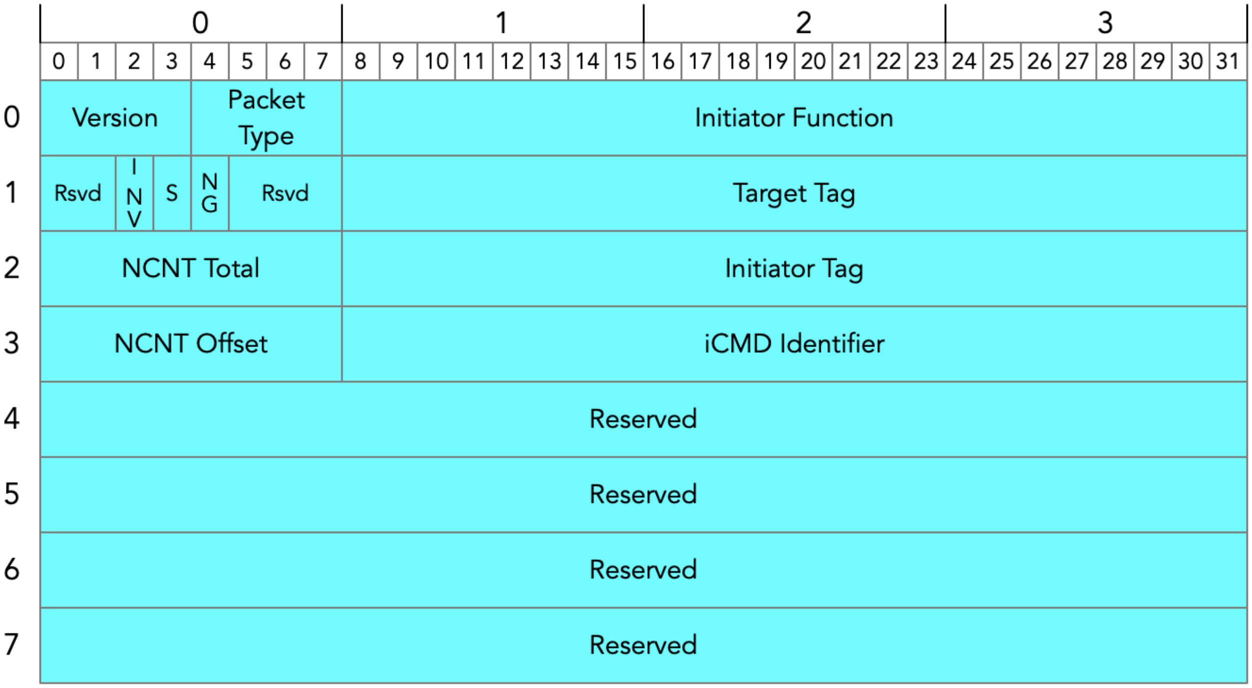
8.1.2 Tunnel Mode



8.1.3 NVMe Packet Types

NVMe Packet Type	Source	Falcon Packet Type	Description
0	Initiator	Push Request	Write Request
1	Target	Pull Request	Write Data Request
2	Initiator	Pull Response	Write Data Response
3	Target	Push Request	Write Completion
4	Initiator	Push Request	Read Request
5	Target	Push Request	Read Response
6-15	--	--	Reserved for future use

8.2 NVMe Base Transport Header (NBTH)



<table>
<tr><td></td><td colspan="8">0</td><td colspan="8">1</td><td colspan="8">2</td><td colspan="8">3</td></tr>
<tr><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td></tr>
<tr><td>0</td><td colspan="4">Version</td><td colspan="4">Packet Type</td><td colspan="24">Initiator Function</td></tr>
<tr><td>1</td><td colspan="2">Rsvd</td><td>INV</td><td>S</td><td>NG</td><td colspan="3">Rsvd</td><td colspan="24">Target Tag</td></tr>
<tr><td>2</td><td colspan="8">NCNT Total</td><td colspan="24">Initiator Tag</td></tr>
<tr><td>3</td><td colspan="8">NCNT Offset</td><td colspan="24">iCMD Identifier</td></tr>
<tr><td>4</td><td colspan="32">Reserved</td></tr>
<tr><td>5</td><td colspan="32">Reserved</td></tr>
<tr><td>6</td><td colspan="32">Reserved</td></tr>
<tr><td>7</td><td colspan="32">Reserved</td></tr>
</table>

The NVMe Base Transport Header (NBTH) must be present in every NVMe-over-Falcon packet. The NBTH format is as shown in the figure above. The following table documents the definition of various fields in the header.

Field	Width	Description
Version	4	This field encodes the version of the NVMe-over-Falcon wire protocol. It must be set to 1.
Packet Type	4	This field encodes the NVMe-over-Falcon packet type as defined in this table.
Initiator Function	24	This field encodes the <Host, PF, VF> tuple of the function that issued the NVMe command.
Data Invalid (INV)	1	This field is only valid for NVMe Write commands and is set to 1 to indicate an error in the NVMe Write data (such as protection information check failure on the initiator side).
Split Command (S)	1	This bit must be set to 1 when the original NVMe command is split across a stripe boundary.
Ncmd Group (NG)	1	This field is only valid when the Split Command (S) bit is set to 1. This bit must be set to 0 for all ncmds in the first stripe and to 1 for all ncmds in the second stripe.
Target Tag	24	This field uniquely identifies the tCMD state at the target side.
NCNT Total	8	When the original NVMe command is split into multiple nCMDs, this field indicates the total number of nCMDs that will be generated for the original command. This field uses 0 based numbering where a value of 0 indicates that there will be only 1 nCMD generated for the original command.
Initiator Tag	24	This field uniquely identifies the nCMD state at the initiator side.
NCNT Offset	8	When the original NVMe command is split into multiple nCMDs, this field indicates the relative number of the generated nCMD for the original command. For example, if the original command is split into 2 nCMDs, the first nCMD will have an NCNT_OFFSET value of 0, and the second will have an NCNT_OFFSET value of 1.
iCMD Identifier	24	This field identifies the parent iCMD of the nCMD associated with this packet.

8.3 Read Request Packet

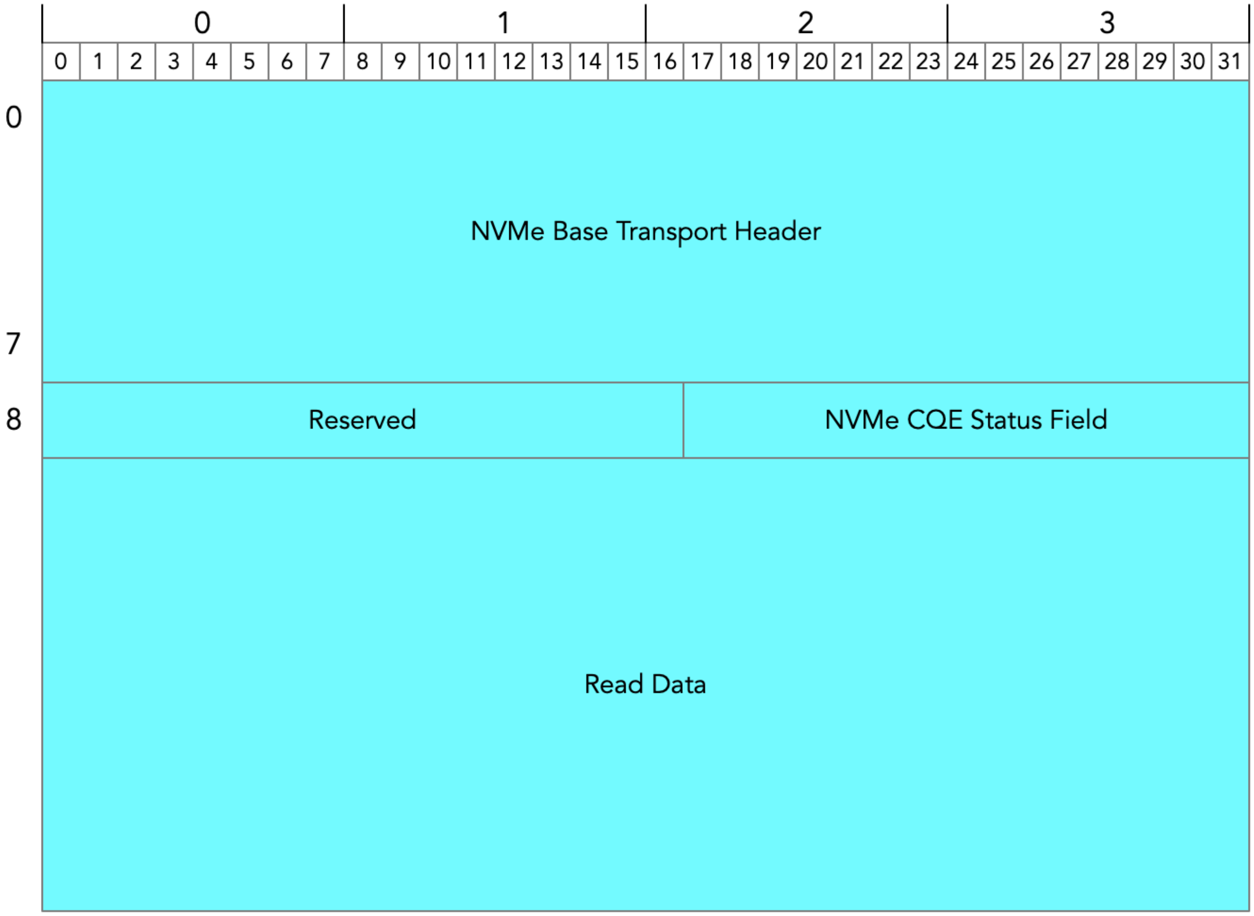
<table>
<tr><th></th><th colspan="8">0</th><th colspan="8">1</th><th colspan="8">2</th><th colspan="8">3</th></tr>
<tr><th></th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th><th>16</th><th>17</th><th>18</th><th>19</th><th>20</th><th>21</th><th>22</th><th>23</th><th>24</th><th>25</th><th>26</th><th>27</th><th>28</th><th>29</th><th>30</th><th>31</th></tr>
<tr><td></td><td colspan="32">NVMe Base Transport Header</td></tr>
<tr><td></td><td colspan="16">Initiator NSID</td><td colspan="16">Target NSID</td></tr>
<tr><td></td><td colspan="32">Target Starting LBA (SLBA) 31:0</td></tr>
<tr><td>10</td><td colspan="32">Target Starting LBA (SLBA) 63:32</td></tr>
<tr><td>11</td><td colspan="16">Number of Logical Blocks (NLB)</td><td colspan="10">Reserved</td><td colspan="4">PRINFO</td><td>F
U
A</td><td>LR</td></tr>
<tr><td>12</td><td colspan="8">Dataset Management (DSM)</td><td colspan="24">Reserved</td></tr>
<tr><td>13</td><td colspan="32">Expected Initial Logical Block Reference Tag (EILBRT)</td></tr>
<tr><td>14</td><td colspan="16">Expected Logical Block Application Tag Mask (ELBATM)</td><td colspan="16">Expected Logical Block Application Tag (ELBAT)</td></tr>
<tr><td>15</td><td colspan="24">Reserved</td><td colspan="8">Target SQID</td></tr>
</table>

The Read Request packet is sent by the initiator to the target to communicate a nCMD read request. The format of the Read Request packet is shown above. The following table documents the definition of various fields in the Read Request packet.

Field	Width	Description
Initiator NSID	16	This field encodes the 16 LSB bits of the NSID from the original NVMe command (iCMD).
Target NSID	16	This field encodes the 16 LSB bits of the NSID in the NVMe command presented to the SSDs at the target side.

Target SLBA	64	This field is obtained from the lookup of the LBA translation table. This field encodes the starting LBA of the iCMD and hence has the same value in all Read Request packets sent for a given iCMD. When the original command is split, this field encodes the starting LBA of the stripe group.
Number of Logical Blocks (NLB)	16	This field encodes the number of logical blocks in the original iCMD. When the original command is split, this field encodes the number of logical blocks in the stripe group.
Protection Information Field (PRINFO)	4	This field is encoded as specified earlier in this section.
Force Unit Access (FUA)	1	This field is encoded as per the original NVMe command (iCMD).
Limited Retry (LR)	1	This field is encoded as per the original NVMe command (iCMD).
Dataset Management (DSM)	8	This field is encoded as per the original NVMe command (iCMD).
Expected Initial Logical Block Reference Tag (EILBRT)	32	This field is encoded as specified earlier in this section.
Expected Logical Block Application Tag Mask (ELBATM)	16	This field is encoded as specified earlier in this section.
Expected Logical Block Application Tab (ELBAT)	16	This field is encoded as specified earlier in this section.
Target SQID	8	This field is obtained from the LBA translation table and encodes the submission queue id at the target side. The SQ numbering at the target side is 0 based and is global across all SSDs attached to the target. The mapping of target SQs to individual SSDs is implementation-defined.

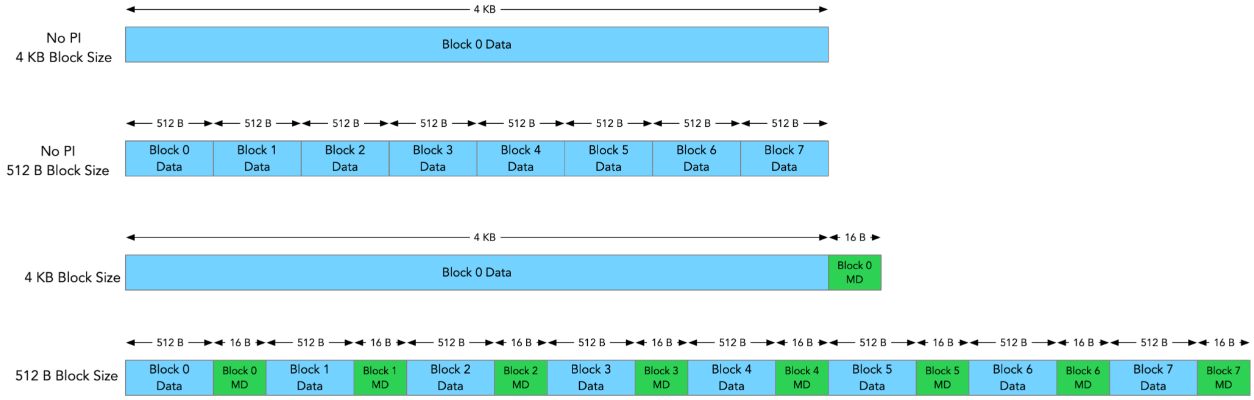
8.4 Read Response Packet



The Read Response packet is sent by the target to the initiator to communicate a nCMD read response. The format of the Read Response packet is shown above. The following table documents the definition of various fields in the Read Response packet.

Field	Width	Description
NVMe CQE Status Field	15	This field specifies the NVMe CQE status field that must be included in the completion back to software at the initiator side.
Read Data	<N>	This field contains the read data

The read data contains the data blocks and associated protection information metadata in DIF format as shown below for 4KB and 512B logical block sizes. The maximum size of the read data in the Read Response packet payload can be 4224B (4096B data + 128B metadata).



8.5 Write Request Packet

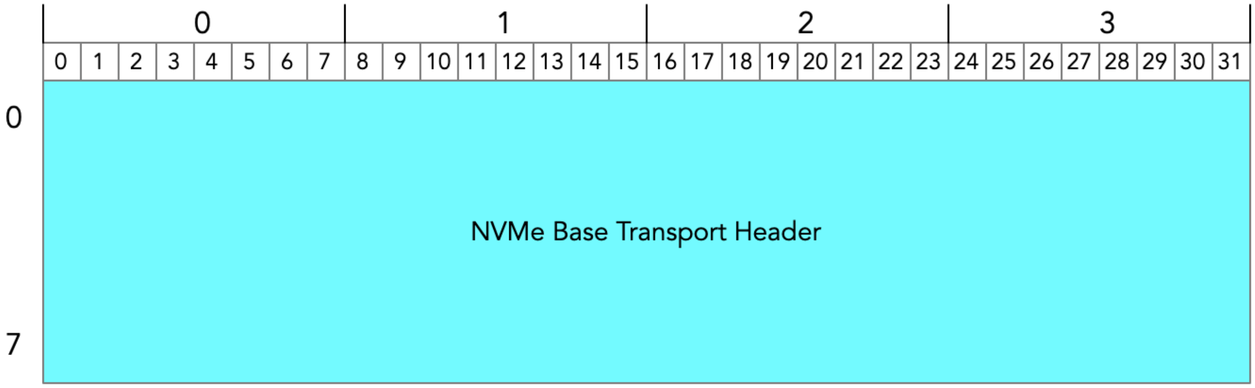
	0								1								2								3							
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	NVMe Base Transport Header																															
7																																
8	Initiator NSID																Target NSID															
9	Target Starting LBA (SLBA) 31:0																															
10	Target Starting LBA (SLBA) 63:32																															
11	Number of Logical Blocks (NLB)																Reserved				DTYPE				Rsvd		PRINFO				FUA	LR
12	Dataset Management (DSM)								Reserved								Directive Specific (DSPEC)															
13	Initial Logical Block Reference Tag (ILBRT)																															
14	Logical Block Application Tag Mask (LBATM)																Logical Block Application Tag (LBAT)															
15	Reserved																								Target SQID							

The Write Request packet is sent by the GNVMe initiator to the GNVMe target to communicate a nCMD write request. The format of the Write Request packet is shown above. The following table documents the definition of various fields in the Write Request packet.

Field	Width	Description
Initiator NSID	16	This field encodes the 16 LSB bits of the NSID from the original NVMe command (iCMD).
Target NSID	16	This field encodes the 16 LSB bits of the NSID in the NVMe command presented to the SSDs at the target side.
Target SLBA	64	This field is obtained from the lookup of the LBA translation table. This field encodes the starting LBA of the iCMD and hence has the same value in all Read Request packets sent for a given iCMD. When the original command is split, this field encodes the starting LBA of the stripe group.
Number of Logical Blocks (NLB)	16	This field encodes the number of logical blocks in the original iCMD. When the original command is split, this field encodes the number of logical blocks in the stripe group.
DTYPE	4	This field is encoded as per the original NVMe command (iCMD) and specifies the Directive Type associated with the Directive Specific field as described in Section 9.1 of the NVMe specification.
Protection Information Field (PRINFO)	4	This field is encoded as specified earlier in this section.
Force Unit Access (FUA)	1	This field is encoded as per the original NVMe command (iCMD).
Limited Retry (LR)	1	This field is encoded as per the original NVMe command (iCMD).
Dataset Management (DSM)	8	This field is encoded as per the original NVMe command (iCMD).
Directive Specific (DSPEC)	16	This field is encoded as per the original NVMe command and specifies the Directive Specific value associated with the Directive Type field as described in Section 9.1 of the NVMe specification.

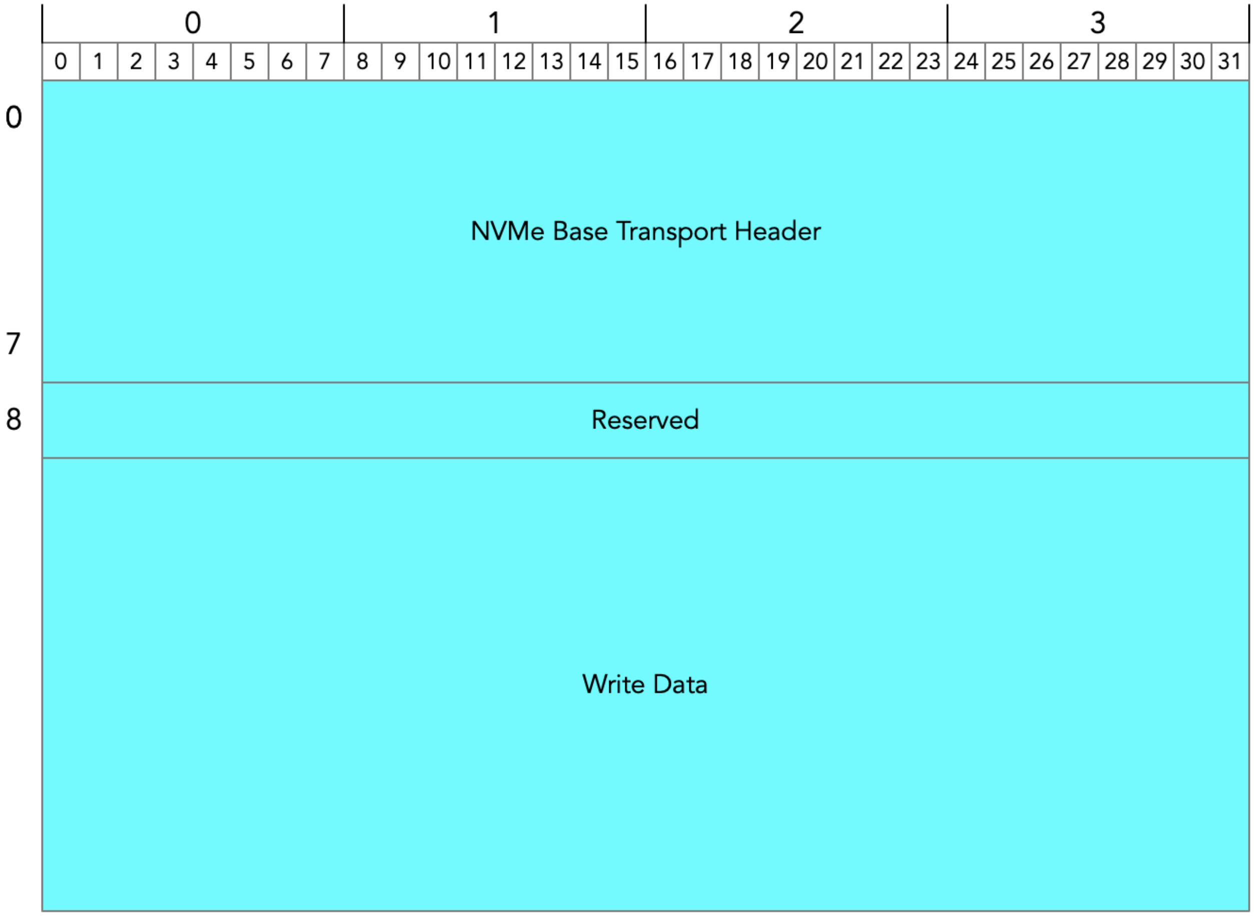
Initial Logical Block Reference Tag (ILBRT)	32	This field is encoded as specified earlier in this section.
Logical Block Application Tag Mask (LBATM)	16	This field is encoded as specified earlier in this section.
Logical Block Application Tab (LBAT)	16	This field is encoded as specified earlier in this section.
Target SQID	8	This field is obtained from the lookup of the LBA translation table and encodes the submission queue id at the target side. The SQ numbering at the target side is 0 based and is global across all SSDs attached to the target. The mapping of target SQs to SSDs is implementation-defined.

8.6 Write Data Request Packet



The Write Data Request packet is sent by the target to the initiator to pull the write data associated with a previously received nCMD. The format of the Write Data Request packet is shown above.

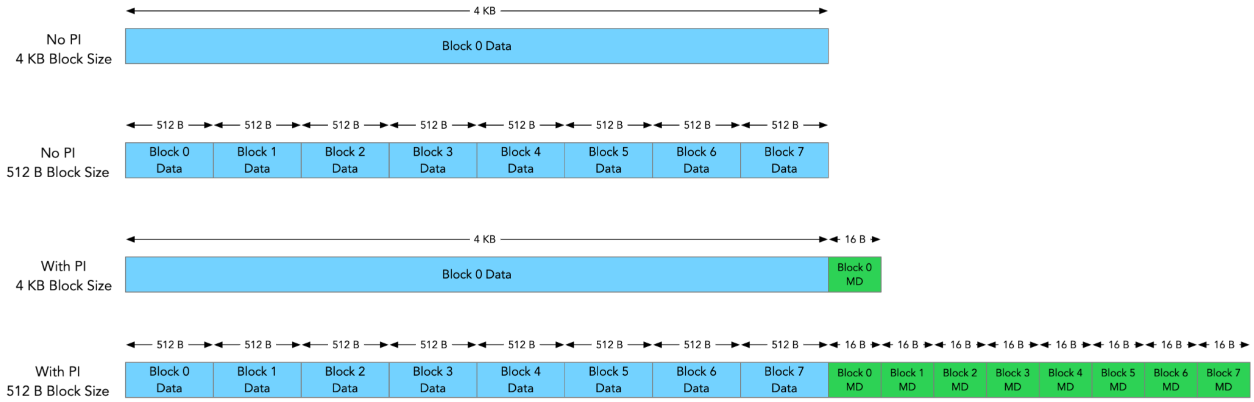
8.7 Write Data Response Packet



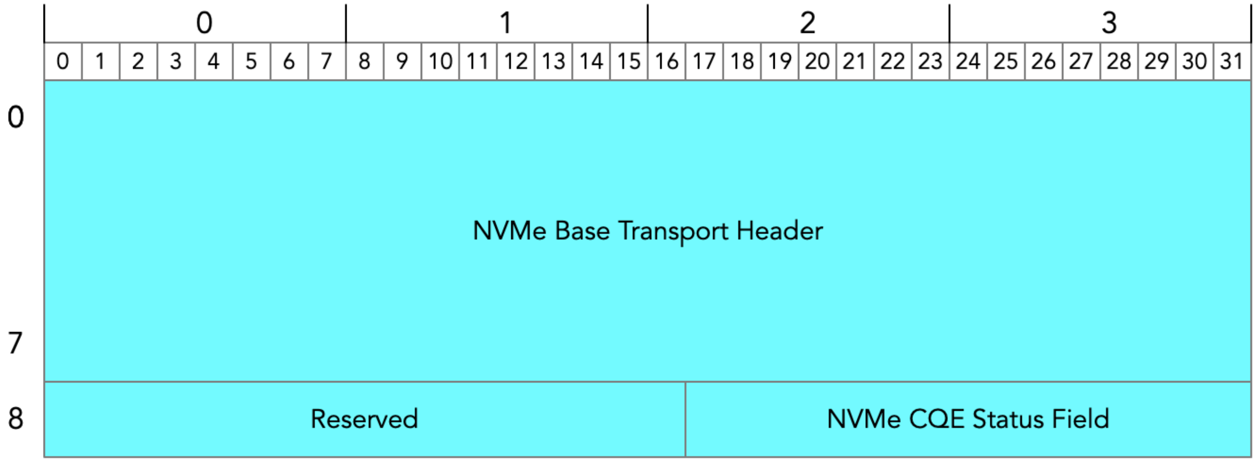
The Write Data Response packet is sent by the initiator to the target to communicate a nCMD write data. The format of the Write Data Response packet is shown above. The following table documents the definition of various fields in the Write Data Response packet.

Field	Width	Description
Write Data	<N>	This field contains the write data.

The write data contains the data blocks and associated protection information metadata in DIX format as shown below for 4KB and 512B logical block sizes. The maximum size of the write data in the Write Data Response packet payload can be 4224B (4096B data + 128B metadata).



8.8 Write Completion Packet



The Write Completion packet is sent by the target to the initiator to communicate a nCPL for a write nCMD. The format of the Write Completion packet is shown above. The following table documents the definition of various fields in the Write Completion packet.

Field	Width	Description
NVMe CQE Status Field	15	This field specifies the NVMe CQE status field that must be included in the completion back to software.