* + 1. **Document VERSION 1.16**
       1. Define structures



* + - 1. Definitions:
         1. Completion Entry (CE)
         2. Completion Entry Size (CES)
      2. Define Q related structures
         1. struct metrics\_meta

u16 meta\_id

A device specific uniquely assigned ID to each new meta data buffer which is allocated. This is how we track meta data buffers, rather than supplying pointers to the buffer

u8 \*vir\_kern\_addr

The virtual kernel address to the dnvme allocated contiguous buffer

dma\_addr\_t meta\_dma\_addr

DMA address handle for meta data buffer within the kernel

* + - * 1. metrics\_meta\_data

metrics\_meta\_list[]

a linked list of structure metrics\_meta

dma\_pool \*pool\_ptr

Assigned by dma\_pool\_create as a result of calling IOCTL\_METABUF\_CREATE

* + - * 1. struct cq\_track

U16 cq\_id

The ID of the CQ which is associated to an interrupt vector.

U8 isr\_fired

== 0; (false) isr did NOT fire.

== !0; (true) isr fired.

* + - * 1. struct irq\_track

U16 int\_vec;

What interrupt vector is this structure referring to.

Should allow MSI-X support since it will hold 2048 vector numbers

u32 int\_count

count of the number of times a specific interrupt occurred.

strcut cq\_track cq[]

linked list; keeps track of all the CQ’s which this int\_vec is responsible for processing.

* + - * 1. struct cmd\_track

U16 unique\_id

This is the driver assigned unique ID for a particular cmd. This allows processing a CQ to “lookup” a completed cmd in a list of possible cmds that were submitted to hardware to find out certain parameters about the cmd when it was sent. Like if the cmd has a PRP list or not.

U8 opcode

cmd opcode

enum {CMD\_ADMIN, CMD\_NVME, CMD\_AON} cmdSet

What cmd set does the opcode belong?

U16 persist\_q\_id

Filled in by IOCTL\_SEND\_64B\_CMD when a persistent queue is being created. It is only assigned correct data for the following cmds:

admin create CQ

admin create SQ

admin delete CQ

admin delete SQ

nvme\_prps prp\_nonpersist;

Points to the PRP list for this cmd if and only if there is a PRP list.

If a buffer of memory can be described within a single physical page of memory then a PRP list won’t be necessary.

* + - * 1. struct metrics\_sq

struct public\_sq

U16 sq\_id;

Even admin Q’s are supported here since q\_id = {0…}

U16 cq\_id

The CQ ID to which this SQ is associated

U16 tail\_ptr

Driver makes available the actual value residing within the appropriate controller space SQxTDBL register.

NOTE: Spec states reading SQxTDBL registers are undefined, thus the driver keeps track of the actual value in this variable

U16 tail\_ptr\_vir

Driver make available the value it will write to the appropriate SQxTBDL register as a result of copying new cmds to the SQ, but have not yet been told to “ring the doorbell”, i.e. uncommitted cmds residing within an SQ. Multiple cmds can be submitted to a SQ with the intent that the doorbell will be rung later. This allows submission of many commands at once. The driver must keep track of the cmds copied, but not yet committed via this parameter.

This value = tail\_ptr + <num\_cmds\_submitted\_B4\_ring\_doorbell>

if (tail\_ptr\_vir == tail\_ptr) then 0 uncommitted cmds exist within an SQ. A cmd is committed when it is placed in the SQ and the doorbell has rung to notify the hdw it is there.

U16 head\_ptr

Driver make available the last completion entry (CE) read from an appropriately linked CQ. Each CE contains a SQ head ptr value, this value must be tracked by the driver. The only way the driver can learn of the new SQ head ptr is if it is told to reap an element from a particular CQ. The user app will request items to be reaped and at this time only, while reaping only those which have been requested to be reaped will the driver learn of the head\_ptr since this value is provided within each completion element (CE) within every CQ.

U16 elements

Driver make available the total number of elements within this Q

This is a 1 based value

struct private\_sq

U8 \*vir\_kern\_add

physical address pointer to the Q’s allocated kernel memory, this is not pointing to a PRP list.

nvme\_prps prp\_persist;

Contains the possible persistent PRP lists for the SQ, this will hold PRP list for Admin Create IO SQ Commands only. Nothing else is really considered persistent. All other cmds can relinquish the PRP list after the command completes, but not the creation of CQ and SQ elements.

U8 contig

0 is false and indicates nvme\_prps is used, vir\_kern\_add is not used, and that the memory encompassing the SQ is not contiguous. !0 indicates true and means vir\_kern\_add is used, nvme\_prps is not used, and that the memory encompassing the SQ is contiguous.

U32 size

length in bytes of the allocated Q in kernel memory, may not be physically contiguous, but will be virtually contiguous.

U16 unique\_cmd\_id;

Global and unique to each SQ on a per device level. It is an incrementing counter to use for every cmd issued in this particular SQ. For the driver must over write CMD.DW3.CID during an IOCTL\_SEND\_64B\_CMD with this value, as it increments this number for each cmd being submitted into this SQ to guarantee uniqueness.

U32 \*dbs; points to the doorbell register in this particular SQ.

cmd\_track\_list

linked list of struct cmd\_track, possibly one for each cmd issue to hdw to this device. Not every cmd submitted to this Q will end up in this list, only certain requirements forces the logic to track a particular cmd. See description of IOCTL\_SEND\_64B\_CMD for details.

dma\_addr\_t sq\_dma\_addr

DMA address handle for Contiguous SQ

* + - * 1. struct metrics\_cq

struct public\_cq

U16 q\_id

Even admin Q’s are supported here since q\_id = {0…}

U16 tail\_ptr

Driver make available the value it thinks the respective tail\_ptr is. This value must be calculated. It is calculated by traversing the CQ from the point located by head\_ptr until the P-bit in the next higher Q location does not correlate to a new unreaped item. Thus if the head\_Ptr == 5, and the next 8 CQ locations have - bit set correctly, then the tail\_ptr = (head\_ptr + 8) = 13. This math must include Q wrapping anomalies.

See IOCTL\_REAP\_INQUIRY for details upon how and when this gets updated.

U16 head\_ptr

Driver make available the actual value residing within the appropriate controller space CQxTDBL register.

NOTE: Spec states reading CQxTDBL registers are undefined, thus the driver keeps track of the actual value in this variable

U16 elements

Driver passes back the total number of elements within this Q

This is a 1 based value

U8 irq\_enable

Driver passes back whether IRQ’s are used to learn of elements from this Q, if not, then it must be polled. This does not indicate how items are actually reaped, rather only how items are learned they need to be reaped. All items are reaped via IOCTL\_REAP when the user app requests items to be returned.

If disabled == 0; polling does not mean there is a thread waiting upon this CQ, rather the CQ is checked for new items during an IOCTL\_REAP\_INQUIRY. Updates to the tail\_ptr occur at this point in time.

If enabled == !0; an ISR must fire in order to allow IOCTL\_REAP\_INQUIRY to return the number of elements waiting within the spec’d CQ. If an ISR does not fire then the hardware did not notify the kernel of new elements, and regardless of whether or not there are actually elements within the CQ IOCTL\_REAP\_INQUIRY will return 0. The ISR must fire, then and only then, are we allowed to “see” elements within a CQ. Updates to the tail\_ptr occur at the time of IOCTL\_REAP\_INQUIRY.

U16 int\_vec

This is the interrupt for this which this CQ was registered when created. This is only has meaning if and only if irq\_enable == !0 (enabled).

u8 pbit\_new\_entry

This is the value of the P bit in a CQ which indicates whether or not the next elemental entry is new or old.

dma\_addr\_t cq\_dma\_addr

DMA address handle for Contiguous CQ

struct private\_cq

U8 \*vir\_kern\_add

physical address pointer to the Q’s allocated kernel memory, this is not pointing to a PRP list.

nvme\_prps prp\_persist;

Contains the possible persistent PRP lists for the CQ, this will hold PRP list for Admin Create IO CQ Commands only. Nothing else is really considered persistent. All other cmds can relinquish the PRP list after the command completes, but not the creation of CQ and SQ elements.

U8 contig

0 is false and indicates nvme\_prps is used, vir\_kern\_add is not used, and that the memory encompassing the CQ is not contiguous. !0 indicates true and means vir\_kern\_add is used, nvme\_prps is not used, and that the memory encompassing the CQ is contiguous.

U32 size

length in bytes of the allocated Q in kernel memory, may not be physically contiguous, but will be virtually contiguous.

U32 \*dbs; points to the doorbell register in this particular CQ.

* + - 1. struct nvme\_prps
         1. U32 npages

No. of pages inside the PRP List only. Pages of PRP1 and PRP2 are not counted in here.

* + - * 1. U32 type

Describes the combination of PRP’s. Following combinations are possible:

|  |  |  |
| --- | --- | --- |
| PRP1 | PRP2 | type |
| NULL | NULL | NO\_PRP |
| Points to a page | NULL | PRP1 |
| Points to a page | Points to a page | PRP1|PRP2 |
| Points to a list of PRP’s | NULL | PRP1|PRP\_List |
| Points to a page | Points to a list of PRP’s | PRP2|PRP\_List |

Note: PRP1, PRP2 and PRP\_List are enums having values 1,2 and 4 respectively.

* + - * 1. \_\_LE64 \*\*vir\_prp\_list

Pointer to a list of virtual kernel pointers to the PRP pages inside PRP List

* + - * 1. \_\_LE64 prp1

Physical address in PRP1 of command

* + - * 1. \_\_LE64 prp2

Physical address in PRP2 of command

* + - * 1. dma\_addr\_t first\_dma

Starting physical address of a PRPList. It can be either present in PRP1 or PRP2 field of the command.

* + - * 1. U8 data\_dir

Direction of transfer of data to or from the device

* + - * 1. Unsigned long data\_buf\_ptr\_addr

Address of data buffer for specific command

* + - * 1. U32 data\_buf\_ptr\_size

Size of the data buffer passed in the command

* + - * 1. scatterlist \*sg

Pointer to the SG lists generated while building PRP’s

* + - * 1. U32 dma\_mapped\_pgs

Number of pages mapped to DMA area

* + - 1. struct metrics\_driver
         1. struct public\_driver

enum {INT\_NONE , INT\_MSI\_SINGLE, INT\_MSI\_MULTI, INT\_MSIX } irq\_type;

Driver detects the active interrupt request scheme during driver load time and stores this away under driver only metrics area, this can be change later but will always represent the active IRQ scheme in place. All schemes are mutually exclusive.

U32 driver\_version;

Updated when the source code changes, do not change if only the API/IOCTL changes

32 bits will represent the version as XX.Y.Z. So 0x10000 implies driver version is 1.0.0

U32 api\_version;

Resides in file nvme\_ioctl.h and describes the version of the API of the IOCTLs. It must be updated whenever the API changes, otherwise the user app could issue request which are not supported. This will not be updated when the code/logic changes, only when the IOCTL definition changes.

32 bits will represent the version as XX.Y.Z. So 0x10603 implies api version is 1.6.3

* + - * 1. struct private\_driver

TBD/unknown.

* + - 1. Define device related structures
         1. struct metrics\_device

metrics\_mtx

Mutex must be taken any time anything **metrics\_cq\_list[], metrics\_sq\_list[], metrics\_meta\_list[] or nvme\_device** needs to be written, modified, removed, freed, deleted and even when read because logic may remove a node while someone is trying to remove it.

metrics\_cq\_list

linked list of struct metrics\_cq

metrics\_sq\_list

linked list of struct metrics\_sq

metrics\_device

device specific data

metrics\_meta\_data

a structure to track all the meta data buffers which have been allocated

MUTEX irq\_track\_mtx

Mutex must be taken any time anything in **irq\_track\_list[]** needs to be written, modified, removed, freed, deleted and even when read because logic may remove a node while someone is trying to remove it.

irq\_track\_list

linked list of isr\_track; Note this could also be an array of size 2048 since that is the maximum MSI-X interrupt vector there could be. But using a linked list or an array decision will be left up to the implementer.

* + - * 1. metrics\_device\_list

linked list of struct metrics\_device, one for each device controlled by this driver

* + - 1. Define nvme\_device structures
         1. struct pci\_dev \*pdev; // pointer to the PCI device. This should point to the correct device opened in the driver.
         2. struct nvme\_ctrl\_reg \*nvme\_ctrl\_space; //pointer to the nvme register space.
         3. U8 \*bar0mapped // points to the BAR0 of the PCI device.
         4. device \*dmadev // points to the device inside the pci structure.
         5. U8 minor\_no; //Indicates the minor device number of this device
         6. Dma\_pool \*prp\_page\_pool //Indicates the dma pool for each device
         7. u32 meta\_unique\_cnt

Use to keep track and generate unique ID’s for meta data buffers assigned on a per device level.

* + 1. IOCTL\_GET\_DRIVER\_METRICS
       1. returns metrics\_driver.public\_driver
    2. IOCTL\_ERR\_CHK
       1. returns device status. See below notes when it returns SUCCESS or FAIL.
       2. Notes:
          1. Check PCI device status (STS) for the following errors,

Parity Error (DPE)

Received Master-Abort (RMA)

Master Data Parity Error (DPD)

If any of the above bits are set, then the device status returns FAIL.

* + - * 1. Checks if the device supports Capabilities List (CL) presence. If it is not set, then it will return FAIL.
        2. Next check is to determine if PCI power management capability is present as a minimum. If PCI power management is not supported, then this should return FAIL.
        3. Check NVME Controller Status bit. (CSTS)

Check if the controller fatal status bit is set (CFS). Return FAIL if it is set .

* + - * 1. If none of the above indicate fail then return SUCCESS.
    1. IOCTL\_GET\_Q\_METRICS
       1. U16 q\_id
          1. Pass the Q ID for which you require the metrics to be returned. This works equally well for admin, (q\_id == 0) as well as any other Q.
       2. enum {METRICS\_CQ, METRICS\_SQ} type
          1. returns metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].public\_cq or metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].public\_sq
       3. u32 nBytes
          1. Number of bytes to copy to the user buffer. The size must be able to hold the metrics data requested.
       4. u8 \*buffer
          1. Buffer to copy the metrics data, which is either public\_cq or public\_sq based on the type of q.
       5. Notes:
          1. If nBytes < sizeof(data\_we\_are\_returning) then alert failure to copy.
    2. IOCTL\_DEVICE\_STATE
       1. enum {ST\_ENABLE, ST\_DISABLE, ST\_DISABLE\_COMPLETELY} nvme\_state
          1. ST\_DISABLE and ST\_DISABLE\_COMPLETELY; is actually a controller reset; ST\_DISABLE performs a reset but leaves the ASQ, ACQ memory but does reset the pointers into ASQ and ACQ. ST\_DISABLE\_COMPLETELY completely free’s all kernel memory and destroys all queues.

The ordering of the logic which follows is important.

set CC.EN = 0;

We must try to wait for the controller to go idle before starting cleaning up memory under its feet. Thus wait 1s. for CSTS.RDY == 0, if it doesn’t within this time, then fail this operation, and do not cleanup, exit early.

for (i=1; i < metrics\_device\_list[this\_device].metrics\_sq\_list[last\_element]; i++) // all IO SQ’s wiped out

for (j=0; j < metrics\_device\_list[this\_device].metrics\_sq\_list[i].cmd\_track[j]; j++)

free the prp list described by metrics\_device\_list[this\_device].metrics\_sq\_list[i].cmd\_track[j].prp\_non\_persist, do NOT free the memory which the list is pointing to.

free the cmd\_track node at metrics\_device\_list[this\_device].metrics\_sq\_list[i].cmd\_track[j]

if (metrics\_device\_list[this\_device].metrics\_sq\_list[i].contig == true) then

free the memory pointed to by metrics\_device\_list[this\_device].metrics\_sq\_list[i].vir\_kern\_addr

else

free the prp list described by metrics\_device\_list[this\_device].metrics\_sq\_list[i].cmd\_track[j].prp\_non\_persist, do NOT free the memory which the list is pointing to.

free the metrics\_sq node at metrics\_device\_list[this\_device].metrics\_sq\_list[i]

Wipe out all cmds within the ASQ

for (j=0; j < metrics\_device\_list[this\_device].metrics\_sq\_list[0].cmd\_track[j]; j++)

free the prp list described by metrics\_device\_list[this\_device].metrics\_sq\_list[0].cmd\_track[j].prp\_non\_persist, do NOT free the memory which the list is pointing to.

free the cmd\_track node at metrics\_device\_list[this\_device].metrics\_sq\_list[0].cmd\_track[j]

if (new\_state == ST\_DISABLE\_COMPLETELY) then

if (metrics\_device\_list[this\_device].metrics\_sq\_list[0].contig == true) then

free the memory pointed to by metrics\_device\_list[this\_device].metrics\_sq\_list[0].vir\_kern\_addr

else

free the prp list described by metrics\_device\_list[this\_device].metrics\_sq\_list[0].cmd\_track[j].prp\_non\_persist, do NOT free the memory which the list is pointing to.

free the metrics\_sq node at metrics\_device\_list[this\_device].metrics\_sq\_list[0]

else // we are just ST\_DISABLE and will keep the actual ASQ memory, just reset pointers into that memory

Reset to default initial values all the contents of metrics\_device\_list[this\_device].metrics\_sq\_list[0].public\_sq

Do NOT touch the contents of metrics\_device\_list[this\_device].metrics\_sq\_list[0].private\_sq

for (i=1; i < metrics\_device\_list[this\_device].metrics\_cq\_list[last\_element]; i++) // all IO CQ’s wiped out

if (metrics\_device\_list[this\_device].metrics\_cq\_list[i].contig == true) then

free the memory pointed to by metrics\_device\_list[this\_device].metrics\_cq\_list[i].vir\_kern\_addr

else

free the prp list described by metrics\_device\_list[this\_device].metrics\_cq\_list[i].cmd\_track[j].prp\_non\_persist, do NOT free the memory which the list is pointing to.

free the metrics\_cq node at metrics\_device\_list[this\_device].metrics\_cq\_list[i]

Wipe out all cmds within the ACQ

if (new\_state == ST\_DISABLE\_COMPLETELY) then

if (metrics\_device\_list[this\_device].metrics\_cq\_list[0].contig == true) then

free the memory pointed to by metrics\_device\_list[this\_device].metrics\_cq\_list[0].vir\_kern\_addr

else

free the prp list described by metrics\_device\_list[this\_device].metrics\_cq\_list[0].cmd\_track[j].prp\_non\_persist, do NOT free the memory which the list is pointing to.

free the metrics\_cq node at metrics\_device\_list[this\_device].metrics\_cq\_list[0]

else // we are just ST\_DISABLE and will keep the actual ACQ memory, just reset pointers into that memory

Reset to default initial values all the contents of metrics\_device\_list[this\_device].metrics\_cq\_list[0].public\_cq

Do NOT touch the contents of metrics\_device\_list[this\_device].metrics\_cq\_list[0].private\_cq

if (new\_state == ST\_DISABLE\_COMPLETELY) then

write 0x00 to ACQ, ASQ, and AQA registers. admin queues are being destroyed in this case.

* + - * 1. ST\_ENABLE

Sets CC.EN = 1

Waits CAP.TO for the CSTS.RDY == 1, if it doesn’t come ready within TO then driver fails this operation. This is not a logical bug, but rather a hdw spec violation.

NOTE: Before enabling the user app should have already sent requests to satisfy system setup of proper admin Q’s and other duties described by section 7.6.1 in rev 1.0b spec.

* + 1. IOCTL\_CREATE\_ADMIN\_Q
       1. enum {ADMIN\_SQ, ADMIN\_CQ} type
       2. U16 elements
          1. Allocate contiguous kernel memory and it must be page aligned for the Q, maybe must use kmalloc() for this.
          2. if (type==ADMIN\_SQ) then allocate contiguous kernel memory in (bytes = elements \* 64B)

Set this in metrics\_device\_list[this\_device].metrics\_cq\_list[0].private\_cq.size

if (elements > 4096) then fail this request

* + - * 1. if (type==ADMIN\_CQ) then allocate contiguous kernel memory in (bytes = elements \* 16B)

Set this in metrics\_device\_list[this\_device].metrics\_cq\_list[0].private\_cq.size

if (elements > 4096) then fail this request

* + - * 1. this value is specified as a 1-based value
      1. NOTES:
         1. The driver must populate the appropriate ASQ or ACQ registers.
         2. The driver must update via read-modify-write sequence, the appropriate ACQS or ASQS field of AQA register
         3. Each Q ID must = 0
         4. if (CSTS.EN == 1) then fail this IOCLT for the spec only allows admin Q creation or modification while the controller is disabled. Doing otherwise will only cause a kernel panic and this would not be a valid test case any way.
         5. Memory

All the allocated memory should be reset

Create a node within at metrics\_device\_list[this\_device].metrics\_sq\_list[append] or at metrics\_device\_list[this\_device].metrics\_cq\_list[append] as appropriate to represent this new Q memory.

If a node already exists with ID=0, then cleanup and fail this request. The previous Q should have been destroyed before re-creating it. Basically we are not giving the ability to destroy any admin Q’s, rather you must do it by a controller reset, and then only if you specify ST\_DISABLE\_COMPLETELY.

if (type == ADMIN\_CQ) then

Set member metrics\_device\_list[this\_device].metrics\_cq\_list[append].public\_cq.pbit\_new\_entry

* + - * 1. Must use mmap to map the contiguous memory to user space.
    1. IOCTL\_PREPARE\_SQ\_CREATION
       1. U16 elements
          1. Total number of entries/elements that needs to be allocated in the kernel
          2. total memory allocated in ‘bytes’ = (elements \* 2^( CC.IOSQES))

Set this in metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].private\_cq.size

* + - * 1. This is a 1-based value
      1. U16 sq\_id;
         1. The user specified unique SQ ID; the driver should fail this request if this ID is already in use.
         2. We could have CQ’s with the same ID and that will be allowed.
      2. U16 cq\_id
         1. Pass the existing CQ ID to which this SQ needs to be associated.
         2. Do NOT fail this request if the cq\_id is not in existence already. The user app may actually want to test this boundary case and so we must allow this illegal state to progress.
      3. U8 contig
         1. This value must be saved in metrics\_device\_list[this\_device].metrics\_sq\_list[append].private\_sq.contig
         2. 0 indicates false that the SQ will not be contiguous. In the non-contiguous buffers are allocated by user space applications via the usual malloc() or new operator. The user mode virtual buffer address will then be passed to the kernel when a Create SQ Cmd is issued via IOCTL\_SEND\_64B\_CMD and the dataBuf parameter is in fact the pointer to the non-contiguous SQ. In this case the kernel will not use metrics\_device\_list[this\_device].metrics\_sq\_list[append].private\_sq.vir\_kern\_addr but rather create a PRP list in metrics\_device\_list[this\_device].metrics\_sq\_list[append].private\_sq.nvme\_prps when the IOCTL\_SEND\_64B\_CMD is issued. In this case the CMD.DW11.PC must == 0, and bitmask MASK\_PRP1 must be set to indicate to the kernel that it is OK to populate the cmd with the PRP list which is created.

The use space allocates the actual memory for the SQ.

* + - * 1. !0 indicates true that the SQ will be contiguous. metrics\_device\_list[this\_device].metrics\_sq\_list[append].private\_sq.vir\_kern\_addr will be used when the user app issues a Create SQ Cmd. The IOCTL\_SEND\_64B\_CMD will peek into the cmd and if a Create SQ Cmd is being issued, it must assume a previous IOCTL\_PREPARE\_SQ\_CREATION was issue to actually create the memory which is being references within that cmd. This assumption is made only when the cmd being issued has its CMD.DW11.PC bit set to true indicating a contiguous buffer is referenced. Thus the metrics\_device\_list[this\_device].metrics\_sq\_list[append].private\_sq.vir\_kern\_addr will be the pointer to that memory.

The driver allocates the actual memory for the SQ now.

* + - 1. NOTE
         1. This is the mechanism to create IO SQ memory in the kernel, it is not used for creating ASQ objects.
         2. Memory

Memory must be page aligned for all allocation, i.e. contig or non-contig.

All the allocated memory should be reset

Create a node within metrics\_device\_list[this\_device].metrics\_sq\_list[append] to represent this new Q memory, fill out both private and public portions of this node completely.

If a node already exists, then cleanup and fail this request. The previous Q should have been destroyed before re-creating it.

* + - * 1. Must use mmap to map the contiguous memory to user space.
    1. IOCTL\_PREPARE\_CQ\_CREATION
       1. U16 elements
          1. Total number of entries/elements that needs to be allocated in the kernel
          2. total memory allocated in ‘bytes’ = (elements \*2^( CC.IOCQES))

Set this in metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].private\_cq.size

* + - * 1. This is a 1-based value
      1. U16 cq\_id;
         1. The user specified unique CQ ID; the driver should fail this request if this ID is already in use.
         2. We could have SQ’s with the same ID and that will be allowed.
      2. U8 contig
         1. This value must be saved in metrics\_device\_list[this\_device].metrics\_sq\_list[append].private\_sq.contig
         2. 0 indicates false that the CQ will not be contiguous. In the non-contiguous buffers are allocated by user space applications via the usual malloc() or new operator. The user mode virtual buffer address will then be passed to the kernel when a Create CQ Cmd is issued via IOCTL\_SEND\_64B\_CMD and the dataBuf parameter is in fact the pointer to the non-contiguous CQ. In this case the kernel will not use metrics\_device\_list[this\_device].metrics\_cq\_list[append].private\_cq.vir\_kern\_addr but rather create a PRP list in metrics\_device\_list[this\_device].metrics\_cq\_list[append].private\_cq.nvme\_prps when the IOCTL\_SEND\_64B\_CMD is issued. In this case the CMD.DW11.PC must == 0, and bitmask MASK\_PRP1 must be set to indicate to the kernel that it is OK to populate the cmd with the PRP list which is created.

The use space allocates the actual memory for the CQ.

* + - * 1. !0 indicates true that the CQ will be contiguous. metrics\_device\_list[this\_device].metrics\_cq\_list[append].private\_cq.vir\_kern\_addr will be used when the user app issues a Create CQ Cmd. The IOCTL\_SEND\_64B\_CMD will peek into the cmd and if a Create CQ Cmd is being issued, it must assume a previous IOCTL\_PREPARE\_CQ\_CREATION was issue to actually create the memory which is being references within that cmd. This assumption is made only when the cmd being issued has its CMD.DW11.PC bit set to true indicating a contiguous buffer is referenced. Thus the metrics\_device\_list[this\_device].metrics\_cq\_list[append].private\_cq.vir\_kern\_addr will be the pointer to that memory.

The driver allocates the actual memory for the CQ now.

* + - 1. NOTE
         1. This is the mechanism to create IO CQ memory in the kernel, it is not used for creating ACQ objects.
         2. Memory

Memory must be page aligned for all allocation, i.e. contig or non-contig.

All the allocated memory should be reset

Create a node within metrics\_device\_list[this\_device].metrics\_cq\_list[append] to represent this new Q memory, fill out both private and public portions of this node completely.

If a node already exists, then cleanup and fail this request. The previous Q should have been destroyed before re-creating it.

Set member metrics\_device\_list[this\_device].metrics\_cq\_list[append].public\_cq.pbit\_new\_entry

* + - * 1. Duties

Create a node within metrics\_device\_list[this\_device].metrics\_cq\_list[append] to represent this new Q memory, fill out both private and public portions of this node completely.

If a node already exists, then cleanup and fail this request. The previous Q should have been destroyed before re-creating it.

* + - * 1. Must use mmap to map the contiguous memory to user space.
    1. IOCTL\_SEND\_64B\_CMD
       1. U16 q\_id
          1. the queue ID to which ‘cmd\_buf\_ptr\_ptr’ must be copied into the next available cmd location, q\_id=0 indicates this is an ASQ, all other values indicate this is an IO SQ.
          2. the next available location is known and updated by the kernel it is the metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].public\_sq.tail\_ptr\_vir value which points to the next free location to copy the current cmd. After the cmd is copied this pointer must point to the next available index with the SQ.
          3. if an attempt to send a cmd and there is only 1 empty place available then error out because that is the definition of a queue full condition.
       2. enum {CMD\_ADMIN, CMD\_NVME, CMD\_AON} cmd\_set;
          1. User app passes in what cmd set the current cmd belongs
       3. U16 bit\_mask
          1. MASK\_PRP1=0x01: indicates that PRP1 in this cmd can be overwritten by the driver.

if PRP1 is not needed then leave PRP1 alone.

* + - * 1. MASK\_PRP2=0x02: indicates that PRP2 in the cmd can be overwritten by the driver.

if PRP2 is not needed then leave PRP2 alone.

* + - * 1. MASK\_MPTR=0x04: indicates that MPTR in the cmd can be overwritten by the driver

if MPTR is not needed then leave MPTR alone

* + - * 1. Algorithm to populate PRP1/PRP2

The 'data\_buf\_ptr\_size’ indicates the total, exact number of bytes being xfer’d by this cmd and the MASK\_PRP1 and/or MASK\_PRP2 indicate whether or not we can update those fields to describe the ‘data\_buf\_ptr’ to the hdw.

If a ‘data\_buf\_ptr’ is passed and no MASK\_PRPx bits are set then return with an error.

if (data\_buf\_ptr\_size == 0) then skip this algorithm and do not touch PRP1 nor PRP2 of the cmd.

if (data\_buf\_ptr\_size ==0) and bit\_mask is missing both PRP1 and PRP2 values, then return with an error.

A prp list will only be necessary if the ‘bit\_mask’ allows updating the appropriate fields and the ‘data\_buf\_ptr’ cannot be completely described by only those prp fields. In other words, the last PRP field will be needed to point to a page which contains the prp pointers which describe the user’s data buffer.

The exact details of converting the ‘data\_buf\_ptr’ to a prp list will be left up the implementer but here are some details which cannot be missed in that algorithm

The allocation of memory to contain the prp list needs to be remembered because eventually it must be freed back to the system.

Memory

Persistent memory is memory describing the actual memory used to copy cmds to/from CQ or SQ’s.

This memory will be the memory described by the ‘data\_buf\_ptr’ in IOCTL\_SEND\_64B\_CMD when the command is a Create CQ or Create SQ cmd, or….

This memory will be the memory allocated during a IOCTL\_PREPARE\_?Q\_CREATION and the ‘contig’ parameter is true.

Non-persistent memory is all other memory which is not considered persistent memory.

Contiguous memory, which is supplied by the kernel, is remembered and described by metrics\_device\_list[this\_device].metrics\_cq\_list[].private\_cq.vir\_kern\_addr in all circumstances.

Memory supplied by user space is always treated and described as non-contiguous memory even if, just out of chance, it really is physically contiguous. A distinction will not be made for user supplied memory and is shown below by….

non-persistent **non-contiguous** memory is described and remembered by metrics\_device\_list[this\_device].metrics\_cq\_list[].private\_sq.cmd\_track[].prp\_nonpersist

non-persistent **contiguous** memory is described and remembered by metrics\_device\_list[this\_device].metrics\_cq\_list[].private\_sq.cmd\_track[].prp\_nonpersist

persistent non-contiguous memory is supplied by the user space and is described and remembered by

metrics\_device\_list[this\_device].metrics\_cq\_list[].private\_cq.prp\_persist, or…

metrics\_device\_list[this\_device].metrics\_sq\_list[].private\_sq.prp\_persist, where…

metrics\_device\_list[this\_device].metrics\_sq\_list[].private\_sq.contig **== false**

persistent contiguous memory is supplied by the kernel space and is described and remembered by

metrics\_device\_list[this\_device].metrics\_cq\_list[].private\_cq.vir\_kern\_addr, or…

metrics\_device\_list[this\_device].metrics\_sq\_list[].private\_sq.vir\_kern\_addr, where…

metrics\_device\_list[this\_device].metrics\_sq\_list[].private\_sq.contig **== true**

At any time if the system cannot provide enough dynamic kernel memory to make a prp list then error this IOCTL.

* + - 1. U8 \*cmd\_buf\_ptr
         1. This buffer must be DWORD aligned.
         2. Virtual address pointer to 64B cmd, this buffer is alloc’d in user space. These values will be copied into the appropriate SQ which is reference by ‘q\_id’ It should be copied to the next available location as indicated by metrics\_device\_list[this\_device].metrics\_sq\_list.public\_sq.tail\_ptr\_vir.

After the copy this pointer should be incremented to point to the next available location taking into account wrapping.

Do not make this copy and error out this IOCTL if there is only 1 empty cmd slot left in the specified SQ, because this state indicates a queue full condition.

* + - * 1. Any modifications/changes made to the cmd needs to be made in place. This will allow the user space to ‘see’ the changes that have occurring herein when this IOCTL has completed this request.
        2. After the driver modifies the CMD.DQ0.CID the CMD.DW0 will be copied back to user space so the tnvme can learn of the assigned CID by dnvme.
        3. if (cmd\_buf\_ptr == NULL) then fail this request
        4. if (q\_id == 0) then

cmd\_buf\_ptr\_size = 64;

* + - * 1. else

cmd\_buf\_ptr\_size = (2 ^ CC.IOSQES)

* + - 1. U32 data\_buf\_ptr\_size
         1. size in bytes of data\_buf\_ptr

if (data\_buf\_ptr\_size!=0) and (data\_buf\_ptr==NULL) then fail this request

if (data\_buf\_ptr\_size==0) and (data\_buf\_ptr!=NULL) then fail this request

if (data\_buf\_ptr\_size!= 0) and MASK\_PRP1 is missing then fail this request

* + - 1. U8 \*data\_buf\_ptr
         1. This buffer must be DWORD aligned.
         2. Virtual user address space pointer to the allocated data buffer to associate with the cmd.
         3. The user app has limited control of how many PRP entries will be generated by requesting the user buffer via normal user space allocation functions. Discontinuities are chunked up in pieces described by CC.MPS.
         4. Non-contiguous CQ/SQ’s are considered user data buffers and these pointers will be supplied here. This occurs when a IOCTL\_PREPARE\_?Q\_CREATION is issued with parameter ‘contig’==false.
      2. U16 meta\_buf\_ptr\_size
         1. Will not support right away, for now if (meta\_buf\_ptr\_size != 0) then fail this IOCTL
         2. For details on future support see section META\_DATA\_HANDLING
      3. U8 \*meta\_buf\_ptr
         1. Will not support right away, for now if (meta\_buf\_ptr != null) then fail this IOCTL
         2. For details on future support see section META\_DATA\_HANDLING
      4. U8 data\_dir
         1. Used for establishing DMA mapping directions
         2. When data\_dir = !0 the direction is DMA\_TO\_DEVICE and when data\_dir = 0 direction is DMA\_FROM\_DEVICE
         3. This parameter is ignored if and only if data\_buf\_ptr\_size == 0.
         4. When the cmd\_buf\_ptr describes Create IO Q’s then data\_buf\_ptr will only have non-null values when the Q’s are discontigous. Thus…

Discontinuous IOSQ’s should be created with data\_dir =!0

Discontinuous IOCQ’s should be created with data\_dir =0

* + - 1. NOTES:
         1. This IOCTL does NOT ring the doorbell
         2. Algorithm

NOTE: Perform all the error checking above before the following algorithm is coded

NOTE: The logical ordering of this algorithm is important

Check if the SQ is full before doing the following, remember to take SQ rollover into account

U16 num\_empty = (metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].public\_sq.tail\_ptr\_vir - metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].public\_sq.head\_ptr); //make note of roll over

if (num\_empty == 1) then

error out this IOCLT the SQ is completely full

else if (num\_empty == 0)

error our we have a kernel programming bug, better log this also

else // we can proceed because there is room for at least 1 more cmd.

if (cmd\_set==CMD\_ADMIN) && (CMD.DW0.opcode==0x01)) //create sq

Follow the algorithm outlined by ALGO\_HANDLE\_Q\_CREATION described below, but replace the ‘?’ with ‘s’ character.

else if (cmd\_set==CMD\_ADMIN) && (CMD.DW0.opcode==0x05)) //create cq

Follow the algorithm outlined by ALGO\_HANDLE\_Q\_CREATION described below, but replace the ‘?’ with ‘c’ character.

else if (cmd\_set==CMD\_ADMIN) && (CMD.DW0.opcode==0x00)) //delete sq

if (data\_buf\_ptr\_size != 0) then error this IOCTL, there should be no user buffer with this cmd

Assign metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].private\_sq.cmd\_track\_list[append].persist\_q\_id == CMD.DW10.QID;

Follow the algorithm outlined by ALGO\_TRACK\_CMD described below.

else if (cmd\_set==CMD\_ADMIN) && (CMD.DW0.opcode==0x04)) //delete cq

if (data\_buf\_ptr\_size != 0) then error this IOCTL, there should be no user buffer with this cmd

Assign metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].private\_sq.cmd\_track\_list[append].persist\_q\_id == CMD.DW10.QID;

Follow the algorithm outlined by ALGO\_TRACK\_CMD described below

else // we have to process the data\_buf\_ptr for all other cmds, i.e. this is the general case

if (data\_buf\_ptr\_size != 0) then

Follow the algorithm outlined by ALGO\_TRACK\_CMD described below

Create a prp list using data\_buf\_ptr and place the description of this list at metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].private\_sq.cmd\_track\_list[append].prp\_nonpersist

else // there is nothing to do, no buffer, no work to do

if (meta\_buf\_ptr\_size != 0) then

Create a PRP physical pointer to the meta\_data and place the value of that pointer in CMD.DW4.MPTR.low and CMD.DW5.MPTR.high

CMD.DW3.CID must always be overwritten by the driver.

The current value of metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].private\_sq.unique\_cmd\_id is written into CMD.DW3.CID

Increment metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].private\_sq.unique\_cmd\_id for the next cmd to arrive in this IOCTL

Copy the CMD to the appropriate SQ as follows

Synchronizing DMA segments:

if (metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].private\_sq.contig == false) then

Before adding command into the SQ dma\_sync\_sg\_for\_device() should be called with appropriate values from metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].private\_sq.prp\_persist structure

Remember to only copy cmd\_buf\_ptr\_size bytes, which is always calculated each time this IOCTL is called.

Copy the newly modified CMD, i.e. the one which this IOCTL has modified for the MPTR, PRP1, PRP2, and CID values at the element location pointed to by metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].public\_sq.tail\_ptr\_vir

Increment metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].public\_sq.tail\_ptr\_vir so that is points to the next available location within this SQ. We have already validated there was room for this cmd at the beginning of this algorithm. Make note of rollover of queue pointers

* + - * 1. ALGO\_TRACK\_CMD

Create a new cmd\_track node within metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].private\_sq.cmd\_track\_list[append]

Do not assign a value to metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].private\_sq.cmd\_track\_list[append].persist\_q\_id

Peek into CMD.DW0.opcode and set metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].private\_sq.cmd\_track\_list[append].cmd\_set appropriately.

Copy CMD.DW0.opcode to metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].private\_sq.cmd\_track\_list[append].opcode.

Copy CMD.DW3.CID to metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].private\_sq.cmd\_track\_list[append].unique\_id.

* + - * 1. ALGO\_HANDLE\_Q\_CREATION

if ((CMD.DW11.PC==1) && (metrics\_device\_list[this\_device].metrics\_?q\_list[CMD.DW10.QID].private\_?q.contig==false)) then error this IOCTL. We have a contiguous buffer, but the cmd states it is not, this will only help crash the kernel, not test the controller.

if ((metrics\_device\_list[this\_device].metrics\_?q\_list[CMD.DW10.QID].private\_?q.contig==false) && (data\_buf\_ptr==NULL)) then error this IOCTL. We have a non-contiguous Q being created and this is always supplied by user space which is not being done here.

if ((metrics\_device\_list[this\_device].metrics\_?q\_list[CMD.DW10.QID].private\_?q.contig==true) && (metrics\_device\_list[this\_device].metrics\_?q\_list[CMD.DW10.QID].private\_?q.vir\_kern\_addr==NULL)) then error this IOCTL. We have a contiguous buffer according to the cmd, but the user never called the appropriate IOCTL\_PREPARE\_?Q\_CREATION.

Create an entry in the cmd tracker as follows:

Assign metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].private\_sq.cmd\_track[append].persist\_q\_id == CMD.DW10.QID;

Follow the algorithm outlined by ALGO\_TRACK\_CMD described below

if (metrics\_device\_list[this\_device].metrics\_?q\_list[CMD.DW10.QID].private\_?q.contig==false) then

Create a prp list using data\_buf\_ptr and place the description of this list at metrics\_device\_list[this\_device].metrics\_?q\_list[CMD.DW10.QID].private\_?q.prp\_persist

else

Do nothing because the Q is already created, it’s contiguous, and already resides at metrics\_device\_list[this\_device].metrics\_?q\_list[CMD.DW10.QID].private\_?q.vir\_kern\_add

* + 1. IOCTL\_DUMP\_METRICS
       1. U16 flen;
          1. Length of the file name that is specified.
       2. U8 \*filename;
          1. Name of the file that need to be used for writing the metrics data.
       3. Notes:
          1. Dump all the metrics data structure in a structure way to the filename supplied.
    2. IOCTL\_RING\_SQ\_DOORBELL
       1. U16 q\_id;
          1. id of the SQ to ring its doorbell,
          2. seek for the SQ within metrics\_device\_list[this\_device].metrics\_sq\_list[]

Copy metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].public\_sq.tail\_ptr\_vir to:

metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].public\_sq.tail\_ptr

the appropriate SQxTDBL register

* + 1. IOCTL\_REAP\_INQUIRY
       1. U16 q\_id
          1. Pass the CQ ID for which queue needs to be reaped. q\_id=0 (ACQ) are supported, as well as all others.

if (q\_id == 0) then Completion Entry Size (CES) = 16B; it is OK to assume a constant

if (q\_id != 0) then Completion Entry Size (CES) = (metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].private\_cq.size / metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].public\_cq.elements)

* + - 1. U16 num\_remaining

Driver returns the number of Completion Entry (CE’s) there are waiting to reap from the specified CQ. This number is explained in the NOTES below.

This is a 1-based number

* + - 1. NOTES:
         1. Only this IOCTL is allowed to modify metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].public\_sq.tail\_ptr. There literally should be no other software which modifies this value, except during driver load time.
         2. If q\_id cannot be found in the existing list of nodes within metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id] then

return error code –EBADSLT

–EBADSLT should not be returned for any other error within this IOCTL

* + - * 1. Algorithm

if (metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].public\_cq.config == false) then

We need to make discontinuous Q memory coherent between hardware and the OS as follows:

If the command is reaped from discontig IOCQ then before accessing command from the discontig Queue dma\_sync\_sg\_for\_cpu should be called with appropriate values from prp\_persist structure

Traverse the CQ from metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].public\_cq.head\_ptr towards the tail of the queue and note the P-bit within the CE. The value of a new entry is indicated by metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].public\_cq.pbit\_new\_entry. Traverse until the P-bit indicates the element is NOT a new CE placed there by the hdw, this must therefore be the location of the metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].public\_cq.tail\_ptr, because the tail\_ptr indicates the next empty location to where an element will be placed by hdw.

Update the metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].public\_cq.tail\_ptr to this newly found CE

num\_remaining = metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].public\_cq.tail\_ptr - metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].public\_cq.head\_ptr

remember to take into account wrapping conditions for this calculation, note however we will never change the value of metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].public\_cq.pbit\_new\_entry in this IOCTL.

* + 1. IOCTL\_REAP
       1. U16 q\_id
          1. Pass the CQ ID for which queue needs to be reaped. Q ID = 0 (ACQ) are supported, as well as all others.

if (q\_id == 0) then Completion Entry Size (CES) = 16B; it is OK to assume a constant

if (q\_id != 0) then Completion Entry Size (CES) = (metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].private\_cq.size / metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].public\_cq.elements)

* + - 1. U16 element
         1. if (element == 0) then this means reap all CE’s within CQ specified by ‘q\_id’ until the buffer size permits and update head pointer accordingly.
         2. If the number of ‘element’ requested can’t be satisfied, because there aren’t enough CE’s to be reaped, then reap until the buffer permits.
         3. Allow partial reaping, where ‘element’ isn’t large enough to reap all CE’s in the CQ. In other words there are more CE’s to be reaped is allowed.
         4. num\_could\_reap = IOCTL\_REAP\_INQUIRY(q\_id);

We must effectively call IOCTL\_REAP\_INQUIRY internally to update the tail\_ptr and find out how many CE’s could be reaped at this time. Remember only IOCTL\_REAP\_INQUIRY is allowed to modify the tail\_ptr. This IOCTL\_REAP must use the values of tail\_ptr as set by IOCTL\_REAP\_INQUIRY’s internal calling scheme. Do this before reaping any elements.

* + - 1. U16 num\_remaining

Driver returns the number of Completion Entry (CE’s) there are waiting to reap from the specified CQ after reaping the specified ‘element’. Thus there are more waiting in the CQ and the caller may want to know this.

num\_remaining = (num\_could\_reap – element);

This is a 1-based number

* + - 1. U16 num\_reaped
         1. Driver Should return the number of CE that are actually reaped.
      2. U16 size
         1. Pass number of bytes of the ‘buffer’
         2. As long as there are enough ‘size’ bytes to copy all the number of ‘element’ entries this will be allowed, even if there is more buffer space that is needed.
      3. U8 \*buffer;
         1. Pass the user mode virtual buffer to which the completion elements will be copied starting from index = 0, byte = 0.
      4. NOTES:
         1. Only this IOCTL is allowed to modify metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].public\_cq.head\_ptr. There literally should be no other software which modifies this value, except during driver load time.
         2. Only this IOCTL is allowed to modify metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].public\_cq.pbit\_new\_entry.

This value is inverted only when reaping from a cq and the heat\_ptr as a result of reaping items from that Q wraps. As soon as the head\_ptr wraps then also this pbit\_new\_entry must be inverted.

* + - * 1. Purpose is to copy as many CE’s as indicated by the number ‘element’ into ‘buffer’ for the CQ with ID==‘q\_id’.
        2. ALGORITHM

For each CE which must be reaped do the following:

Allow metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].public\_cq.head\_ptr to point to the next items in the list because we are reaping this element.

Using CE.sq\_id and CE.cmdId perform a lookup to find the cmd\_track node within metrics\_device\_list[this\_device].metrics\_sq\_list[CE.sq\_id].private\_sq.cmd\_track\_list[x]. Call this node metrics\_device\_list[this\_device].metrics\_sq\_list[CE.sq\_id].private\_sq.cmd\_track\_list[cmd\_we\_tracked] for future reference.

if (we didn’t find any node during this lookup) then

Update metrics\_device\_list[this\_device].metrics\_sq\_list[CE.sq\_id].public\_sq.head\_ptr with value from CE.head\_ptr

Copy this CE into the next free element location of ‘data\_buf\_ptr’, there may be more CE’s to copy thus an indexer will have to be maintained for the next possible copy forthcoming.

Continue to the next for loop CE element we may have to process.

else continue to the next step on the next line below.

if (metrics\_device\_list[this\_device].metrics\_sq\_list[CE.sq\_id].private\_sq.cmd\_track\_list[cmd\_we\_tracked].cmdSet == CMD\_ADMIN) then

if (metrics\_device\_list[this\_device].metrics\_sq\_list[CE.sq\_id].private\_sq.cmd\_track\_list[cmd\_we\_tracked].opcode == 0x05) //creat CQ

if (CE.status == SUCCESS(0)) then follow ALGO\_CREATE\_CQ\_SUCCESS described below.

else follow ALGO\_CREATE\_CQ\_FAILED described below.

else if (metrics\_device\_list[this\_device].metrics\_sq\_list[CE.sq\_id].private\_sq.cmd\_track\_list[cmd\_we\_tracked].opcode == 0x01) //creat SQ

if (CE.status == SUCCESS(0)) then follow ALGO\_CREATE\_SQ\_SUCCESS described below.

else follow ALGO\_CREATE\_SQ\_FAILED described below.

else if (metrics\_device\_list[this\_device].metrics\_sq\_list[CE.sq\_id].private\_sq.cmd\_track\_list[cmd\_we\_tracked].opcode == 0x04) //del CQ

if (CE.status == SUCCESS(0)) then follow ALGO\_DELETE\_CQ\_ SUCCESS described below.

else follow ALGO\_DELETEE\_CQ\_ FAILED described below.

else if (metrics\_device\_list[this\_device].metrics\_sq\_list[CE.sq\_id].private\_sq.cmd\_track\_list[cmd\_we\_tracked].opcode == 0x00) // del SQ

if (CE.status == SUCCESS(0)) then follow ALGO\_DELETE \_SQ\_SUCCESS described below.

else follow ALGO\_DELETE \_SQ\_FAILED described below.

else the follow ALGO\_GEN\_PROCESSING described below.

else // this is not an admin cmd at all

Lookup CC.CSS and follow the appropriate algorithm based upon the cmd set chosen.

NVM cmd set

follow ALGO\_GEN\_PROCESSING described below.

AON cmd set

follow ALGO\_GEN\_PROCESSING described below.

The rest is TBD

Update the appropriate register CQxHDBL as indicated by ‘q\_id’ with the value residing within with metrics\_device\_list[this\_device].metrics\_sq\_list[CE.sq\_id].public\_sq.head\_ptr.

* + - * 1. ALGO\_CREATE\_CQ\_SUCCESS and ALGO\_DELETE\_CQ\_FAILED:

free cmd\_track node from the ASQ, i.e. the ASQ is metrics\_device\_list[this\_device].metrics\_sq\_list[CE.sq\_id].private\_sq.cmd\_track\_list[]. Do NOT free and child elements within this node like a prp list.

Update metrics\_device\_list[this\_device].metrics\_sq\_list[CE.sq\_id].public\_sq.head\_ptr with value from CE.head\_ptr

Copy this CE into the next free element location of ‘data\_buf\_ptr’, there may be more CE’s to copy thus an indexer will have to be maintained for the next possible copy forthcoming.

Continue to the next for loop CE element we may have to process

* + - * 1. ALGO\_CREATE\_CQ\_FAILED and ALGO\_DELETE\_CQ\_SUCCESS:

Let XXXX = metrics\_device\_list[this\_device].metrics\_sq\_list[CE.sq\_id].private\_sq.cmd\_track\_list[cmd\_we\_tracked].persist\_q\_id

Perform a lookup of within metrics\_device\_list[this\_device].metrics\_cq\_list[XXXX].private\_cq.contig to see if the Q’s memory is contiguous or not.

if (Q memory is contiguous) then

free the memory pointed to by metrics\_device\_list[this\_device].metrics\_cq\_list[XXXX].private\_cq.vir\_kern\_addr

else // memory is not contig

free the memory pointed to by metrics\_device\_list[this\_device].metrics\_cq\_list[XXXX].private\_cq.prp\_persist

free the prp list described by memory pointed to by metrics\_device\_list[this\_device].metrics\_cq\_list[XXXX].private\_cq.prp\_persist

free the cmd\_track node at metrics\_device\_list[this\_device].metrics\_sq\_list[CE.sq\_id].private\_sq.cmd\_track\_list[cmd\_we\_tracked]

free the metrics\_cq node at metrics\_device\_list[this\_device].metrics\_cq\_list[XXXX].

Update metrics\_device\_list[this\_device].metrics\_sq\_list[CE.sq\_id].public\_sq.head\_ptr with value from CE.head\_ptr

Copy this CE into the next free element location of ‘data\_buf\_ptr’, there may be more CE’s to copy thus an indexer will have to be maintained for the next possible copy forthcoming.

Continue to the next for loop CE element we may have to process

* + - * 1. ALGO\_CREATE\_SQ\_SUCCESS and ALGO\_DELETE\_SQ\_FAILED

free cmd\_track node from the ASQ, i.e. the ASQ is metrics\_device\_list[this\_device].metrics\_sq\_list[CE.sq\_id].private\_sq.cmd\_track\_list[]. Do NOT free and child elements within this node like a prp list.

Update metrics\_device\_list[this\_device].metrics\_sq\_list[CE.sq\_id].public\_sq.head\_ptr with value from CE.head\_ptr

Copy this CE into the next free element location of ‘data\_buf\_ptr’, there may be more CE’s to copy thus an indexer will have to be maintained for the next possible copy forthcoming.

Continue to the next for loop CE element we may have to process

* + - * 1. ALGO\_CREATE\_SQ\_FAILED and ALGO\_DELETE\_SQ\_SUCCESS:

Let XXXX = metrics\_device\_list[this\_device].metrics\_sq\_list[CE.sq\_id].private\_sq.cmd\_track\_list[cmd\_we\_tracked].persist\_q\_id

Perform a lookup of within metrics\_device\_list[this\_device].metrics\_sq\_list[XXXX].private\_sq.contig to see if the Q’s memory is contiguous or not.

if (Q memory is contiguous) then

free the memory pointed to by metrics\_device\_list[this\_device].metrics\_sq\_list[XXXX].private\_sq.vir\_kern\_addr

else // memory is not contig

free the memory pointed to by metrics\_device\_list[this\_device].metrics\_sq\_list[XXXX].private\_sq.prp\_persist

free the prp list described by memory pointed to by metrics\_device\_list[this\_device].metrics\_sq\_list[XXXX].private\_sq.prp\_persist

free the cmd\_track node at metrics\_device\_list[this\_device].metrics\_sq\_list[CE.sq\_id].private\_sq.cmd\_track\_list[cmd\_we\_tracked]

free the metrics\_sq node at metrics\_device\_list[this\_device].metrics\_sq\_list[XXXX].

Update metrics\_device\_list[this\_device].metrics\_sq\_list[CE.sq\_id].public\_sq.head\_ptr with value from CE.head\_ptr

Copy this CE into the next free element location of ‘data\_buf\_ptr’, there may be more CE’s to copy thus an indexer will have to be maintained for the next possible copy forthcoming.

Continue to the next for loop CE element we may have to process

* + - * 1. ALGO\_GEN\_PROCESSING

free the PRP list described by metrics\_device\_list[this\_device].metrics\_sq\_list[CE.sq\_id].private\_sq.cmd\_track\_list[cmd\_we\_tracked].prp\_persist

free the cmd\_track node from metrics\_device\_list[this\_device].metrics\_sq\_list[CE.sq\_id].private\_sq.cmd\_track\_list[cmd\_we\_tracked]

Update metrics\_device\_list[this\_device].metrics\_sq\_list[CE.sq\_id].public\_sq.head\_ptr with value from CE.head\_ptr

Copy this CE into the next free element location of ‘data\_buf\_ptr’, there may be more CE’s to copy thus an indexer will have to be maintained for the next possible copy forthcoming.

Continue to the next for loop CE element we may have to process

* + 1. ISR Design: ISR Routine
       1. Pin based IRQ’s are not supported in this manner. Pin based interrupts do not fit well within this design, because pin based IRQ’s remain active, thus creating an interrupt storm, for CQ’s that choose to only reap some, not all, of the elements within its CQ. The power of this design is to allow the intention of not processing all CE’s within a CQ to test things like seeing the hardware will stop inserting CE’s into a CQ by allowing the CQ to fill up, and then perhaps only remove 1 element to see if the hardware will not hang but insert into a free location now. The ability to allow the power of controlling the CQ at such fine grain forces us to invoke the hybrid approach of this design. Things like pin based IRQ’s will have to be tested in another manner, perhaps as a canned test embedded within dnvme which can surgically test components of pin based interrupts.
       2. There will be one global ISR routine connected to all possible interrupt vectors using request\_irq() called the TopHalf().
       3. The nvme spec states that admin Q’s only work with interrupts and it assume IRQ=0, but we can ignore this for polling support. Thus there is a mode to disable all interrupts.
       4. There will be top half and bottom half ISR processing. The top half is the global ISR routine and it should do the following:
          1. irq\_return\_t TopHalfIsr(int irq, void \*dev\_id, struct pt\_regs \*regs)

Do NOT take the metrics\_device\_list[this\_device].irq\_track\_mtx mutex. We never want the top\_half to have to gain access to the mutex, because gaining access to a mutex during an ISR is very bad, we will not take out this mutex during the top half.

Disable this ‘irq’ in the appropriate manner. Must consider all ways of disabling irq’s including INT\_MSI\_SINGLE, INT\_MSI\_MULTI, INT\_MSIX.

Schedule a work item in the system workqueue. Pass to this work item the values of ‘irq’ and ‘dev\_id’. This work item is the bottom half.

* + - * 1. BottomHalfIsr(int irq, void \*dev\_id)

Must grab the metrics\_device\_list[this\_device].irq\_track\_mtx mutex for the entire duration of this processing to guarantee coherency with the IOCTL\_REAP processing. Release the mutex upon exit of this routine.

Seek for the node within irq\_track\_list[x] where irq\_track\_list[x].int\_vec == irq.

Call this node irq\_track\_list[current\_irq] for future reference.

for each node in irq\_track\_list[current\_irq].cq\_track[x] do the following

irq\_track\_list[current\_irq].cq\_track[x].isr\_fired == TRUE

irq\_track\_list[current\_irq].cq\_track[x].isr\_count++;

if there are no elements within in irq\_track\_list[current\_irq].cq\_track[x] then log an error, but do nothing about it programmatically

Re-enable the ‘irq’ which was disabled in TopHalfIsr().

* + - 1. Notes:
         1. All an ISR is doing for the test environment is validating that an ISR was generated by the hardware and this action simply notifies/allows the kernel driver to process cmds within the associated CQ. The actually processing/reaping of those completion elements (CE) will be done by IOCTL\_REAP in a polling fashion regardless.
         2. The IOCTL\_REAP\_INQUIRY will return that no elements are in a CQ, unless the associated ISR has fired. This is enough to prove interrupts are working and it also allows ISR’s to work in a user space vs. kernel space polling API. IN other words the general interface into the kernel for reaping CE’s is polling regardless if interrupts are enabled for a particular CQ or not.
         3. The IOCTL\_REAP\_INQUIRY updates the metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].public\_cq.tail\_ptr if and only if the associated ISR has fired.
         4. The IOCTL\_REAP won’t be told there is anything to reap and thus won’t reap unless IOCTL\_REAP\_INQUIRY states there are items to reap. The IOCTL\_REAP\_INQUIRY is called from within IOCTL\_REAP to learn if there are any CE’s to reap. Thus IOCTL\_REAP won’t be able to reap unless an ISR has fired. This proves interrupts are working even though the interrupt didn’t do any processing of the CE’s in a CQ.
    1. ISR Design: IOCTL\_SET\_IRQ
       1. u8 enum {INT\_NONE, INT\_MSI\_SINGLE, INT\_MSI\_MULTI, INT\_MSIX} new\_irq;
          1. What is the new interrupt scheme which is desired, regardless of what it is now when this IOCTL has finished this new scheme will be active.
       2. U16 num\_irqs;
          1. Pass the number of interrupt vectors we wish to reserve and request the OS to grant to us at this time.
          2. Algorithm to verify validity of this number

if (new\_irq == INT\_MSI\_SINGLE) then

if (num\_irqs > 1) return failure

else if (new\_irq == INT\_MSI\_MULTI) then

if (num\_irqs > MSICAP.MC.MME) return failure

if (num\_irqs > 32) return failure

Notes

These interrupt vectors will/must range from (0 - (num\_irqs-1))

else if (new\_irq == INT\_MSIX) then

if(num\_irqs > MSIXCAP.MXC.TS) return failure

if (num\_irqs > 2048) return failure

Notes

These interrupt vectors will/must range from (0 - (num\_irqs-1))

* + - 1. Notes:
         1. At driver load time, or sometime really early before any devices are actually opened

The effectively call IOCTL\_SET\_IRQ with the value IRQ\_NONE to disable all IRQ’s.

Call the function to fill/find/discover the value of metrics\_driver.public\_driver.irq

Obviously this function should see IRQ’s disable and report IRQ\_NONE.

* + - * 1. IOCTL algorithm is as follows:

if (CSTS.EN == 1) then

fail this IOCTL request

The controller must be disabled for the interrupt scheme to be set/changed. Do not do this automatically in this handler, it should be placed upon the called so that they understand the repercussions of changing interrupts, this is extremely destructive and we should not hide that from the caller.

The caller can execute either ST\_DISABLE\_COMPLETELY or ST\_DISABLE to IOCTL\_DEVICE\_STATE.

Any access to metrics\_device\_list[this\_device].irq\_track\_list[] must grab the metrics\_device\_list[this\_device].irq\_track\_mtx mutex

Disable all ISR’s for all interrupt vectors as if INT\_NONE was issued.

if (new\_irq == INT\_NONE) then exit in success

The disabling ISR’s will be unique to each interrupt scheme, i.e. INT\_MSI\_SINGLE, INT\_MSI\_MULTI, INT\_MSIX. Each scheme has it own way to disable IRQ’s, we need to disable all IRQ’s.

Delete all nodes within metrics\_device\_list[current\_device].irq\_track\_list[] because we will be rebuilding a new one as follows.

if (new\_irq == INT\_MSI\_SINGLE) then

Create 1 node within metrics\_device\_list[current\_device].irq\_track\_list[]

call pci\_enable\_msi()

metrics\_device\_list[current\_device].irq\_track\_list[0].int\_vec = dev.irq\_number

this is a kernel structure and will be set by the kernel after calling pci\_enable\_msi()

else if (new\_irq == INT\_MSI\_MULTI) then

Create ‘num\_irqs’ nodes in metrics\_device\_list[current\_device].irq\_track\_list[]

call pci\_enable\_msi\_block()

if we don’t get at least what we ask for then return failure.

This call may allocate > we ask for because it has power of 2 requirements

metrics\_device\_list[current\_device].irq\_track\_list[0].int\_vec = dev\_irq\_number

metrics\_device\_list[current\_device].irq\_track\_list[1].int\_vec = dev\_irq\_number+1

metrics\_device\_list[current\_device].irq\_track\_list[2].int\_vec = dev\_irq\_number+2

do until num\_irqs is satisfied

else if (new\_irq == INT\_MSIX) then

Create ‘num\_irqs’ nodes in metrics\_device\_list[current\_device].irq\_track\_list[]

call pci\_enable\_msix()

if we don’t get at least what we ask for then return failure.

metrics\_device\_list[current\_device].irq\_track\_list[0].int\_vec = msix\_entry[0].vector;

metrics\_device\_list[current\_device].irq\_track\_list[1].int\_vec = msix\_entry[1].vector;

metrics\_device\_list[current\_device].irq\_track\_list[2].int\_vec = msix\_entry[2].vector;

do until num\_irqs is satisfied

Call request\_irq() for all elements of the newly created metrics\_device\_list[current\_device].irq\_track\_list[] and assign the top half callback defined earlier by TopHalfIsr(int irq, void \*dev\_id, struct pt\_regs \*regs)

The driver must fill/find/discover the value of metrics\_driver.public\_driver.irq because we just change the scheme.

* + 1. ISR Design : Additions to IOCTL\_CREATE\_ADMIN\_Q
       1. if (type == ADMIN\_Q) then
          1. Must grab the metrics\_device\_list[this\_device].irq\_track\_mtx mutex
          2. switch(metrics\_driver.public\_driver.irq)
          3. case INT\_MSI:

Create a new cq\_track node and add it to metrics\_device\_list[current\_device].irq\_track\_list[0].cq[append]

metrics\_device\_list[current\_device].irq\_track\_list[0].cq[append].cq\_id == 0 // this is a constant on purpose

metrics\_device\_list[current\_device].irq\_track\_list[0].cq[append].isr\_fired == FALSE

* + - * 1. case INT\_MSI\_MULTI and case INT\_MSIX

Search for node where this is true: metrics\_device\_list[current\_device].irq\_track\_list[x].int\_vec == 0; call this device\_list[current\_device].irq\_track\_list[one\_we\_found] for future reference

Create a new cq\_track node and add it to metrics\_device\_list[current\_device].irq\_track\_list[one\_we\_found].cq[append]

metrics\_device\_list[current\_device].irq\_track\_list[one\_we\_found].cq[append].cq\_id = 0 // this is a constant on purpose

metrics\_device\_list[current\_device].irq\_track\_list[one\_we\_found].cq[append].isr\_fired = FALSE

* + - 1. Notes:
         1. interrupt vector 0 is associated with admin CQ’s as stated in the spec.
    1. ISR Design : Additions to IOCTL\_SEND\_64B\_CMD
       1. if ((cmd\_set == CMD\_ADMIN) && (CMD.DW0.opcode == 0x05)) then // Create CQ cmd
          1. if (CMD.DW11.IEN == true) then

Must grab the metrics\_device\_list[this\_device].irq\_track\_mtx mutex

switch(metrics\_driver.public\_driver.irq)

case INT\_MSI:

Create a new cq\_track node and add it to metrics\_device\_list[current\_device].irq\_track\_list[0].cq[append]

metrics\_device\_list[current\_device].irq\_track\_list[0].cq[append].cq\_id == CMD.DW10.QID;

metrics\_device\_list[current\_device].irq\_track\_list[0].cq[append].isr\_fired == FALSE

case INT\_MSI\_MULTI and case INT\_MSIX

Search for node where this is true: metrics\_device\_list[current\_device].irq\_track\_list[x].int\_vec == CMD.DW11.IV; call this device\_list[current\_device].irq\_track\_list[one\_we\_found] for future reference

Create a new cq\_track node and add it to metrics\_device\_list[current\_device].irq\_track\_list[one\_we\_found].cq[append]

metrics\_device\_list[current\_device].irq\_track\_list[one\_we\_found].cq[append].cq\_id = CMD.DW10.QID;

metrics\_device\_list[current\_device].irq\_track\_list[one\_we\_found].cq[append].isr\_fired = FALSE

set metrics\_device\_list[current\_device].metrics\_cq\_list[CMD.DW10.QID].public\_cq.irq\_enabled = TRUE

this notifies that interrupts are now enabled for this CQ which is being created

set metrics\_device\_list[current\_device].metrics\_cq\_list[CMD.DW10.QID].public\_cq.int\_vec = CMD.DW11.IV

this notifies what interrupt vector is associated to this CQ.

* + 1. ISR Design: Additions to IOCTL\_REAP\_INQUIRY
       1. For each cmd we must process from CQ with id=q\_id do the following
          1. if (metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].public\_cq.irq\_enabled == true) then

Must grab the metrics\_device\_list[this\_device].irq\_track\_mtx mutex

Use metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].public\_cq.int\_vec to find the irq\_track within irq\_track\_list[]. Call this node irq\_track\_list[int\_we\_found] for future reference

seek for the cq\_track node within irq\_track.cq[] using ‘q\_id’. Call this node irq\_track\_list[int\_we\_found].cq[cq\_we\_found] for future reference.

if (irq\_track\_list[int\_we\_found].cq[cq\_we\_found].isr\_fired == true) then

We are allowed to perform the same algorithm we do for non-isr based reaping and return the number of elements within CQ with q\_id.

If we find there a no items to reap, then reset irq\_track\_list[int\_we\_found].cq[cq\_we\_found].isr\_fired = false; and return 0 to the caller.

else return 0

Nothing is in the request CQ even if there really are elements because the ISR has not fired as of yet. We may have to deal with aggregating ISR’s and thus we the host are not yet suppose to be notified of any complete elements. So we return 0.

* + 1. ISR Design: Additions to IOCTL\_REAP
       1. We must not have the bottom half processing making changes under our feet otherwise we will miss CE’s being placed into a CQ with this design. Thus do the following as the 1st thing which is done in this IOCTL\_REAP coding.
          1. if (metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].private\_cq.irq\_enabled == true) then

Must grab the metrics\_device\_list[this\_device].irq\_track\_mtx mutex for the entire duration of this IOCTL processing to guarantee coherency with the bottom half processing. Release the mutex upon exit of this IOCTL.

* + - * 1. **NOTE: WE must have recursive mutex otherwise this logic will not work. If we take the mutex here and then later when this logic internally generates a call to IOCTL\_REAP\_INQUIRY it will be grabbed again, thus recursive mutexs are needed in the current design. Verify this will be possible.**
      1. For each cmd we must process from CQ with id=q\_id do the following
         1. if (metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].private\_cq.irq\_enabled == true) then

Using metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].private\_cq.int\_vec find the irq\_track node within metrics\_device\_list[this\_device].irq\_track\_list[x] and call that node metrics\_device\_list[this\_device].irq\_track\_list[one\_we\_found] for future reference.

Lookup the cmd in the appropriate metrics\_device\_list[this\_device].metrics\_sq\_list[CE.sq\_id].private\_sq.cmd\_track\_list[x]; and if a cmd is found, and if that cmd is a Create CQ cmd, and the CE.status indicates that it FAILED, then do the following:

Remove the cq\_track node from metrics\_device\_list[this\_device].irq\_track\_list[one\_we\_found].cq[q\_id]; we did not create a CQ successfully, thus this CQ should no longer be associated with an interrupt vector.

Lookup the cmd in the appropriate metrics\_device\_list[this\_device].metrics\_sq\_list[CE.sq\_id].private\_sq.cmd\_track\_list[x]; and if a cmd is found, and if that cmd is a Delete CQ cmd, and the CE.status indicates that it SUCCESS, then do the following:

Remove the cq\_track node from metrics\_device\_list[this\_device].irq\_track\_list[one\_we\_found].cq[q\_id]; we did not create a CQ successfully, thus this CQ should no longer be associated with an interrupt vector.

Function exiting algorithm

As the last logic with this function, it must contain the following:

If there are 0 CE’s within the CQ specified by ‘q\_id’ then assign metrics\_device\_list[this\_device].irq\_track\_list[one\_we\_found].cq[q\_id].isr\_fired = false

It’s important to note that we will NOT reset this flag if >= 1 CE exists yet in this CQ.

Release the mutex which was taken at the beginning of this function

* + 1. Meta data handling
       1. Notes:
          1. This buffer must be DWORD aligned per nvme spec.
          2. The user space meta buffer can only be guaranteed to be virtually contiguous, the spec states rev 1.0b, p.117 that this meta data must be physically contiguous. This implies that if there is meta data then the dnvme must create contiguous buffer to an on-the-fly kernel allocated contiguous buffer to use in place of the virtually contiguous one. This approach may be necessary for a production driver, but for a test driver another approach is possible.
          3. When a meta data buffer has been created, it could remain for use by many IO ops. The memory can also be used to access meta data returned from the device after the lifetime of a cmd has ended.
          4. The meta data buffer will be track in its own dedicated linked list for each device dnvme controls.
          5. The MPTR pointer in the CMD.DW4, CMD.DW5 is not necessarily a PRP pointer because PRP ptrs can only point to a single page of memory defined by CC.MPS. Rather the MPTR can point to any length of buffer so as long as it is physically contiguous.
       2. Create IOCTL\_METABUF\_CREATE
          1. u16 alloc\_size

Maximum allowed value is 16KB, thus bit #15 must never be set.

his is an error out and fail this IOCTL situation

This value is sent as a parameter to the kernel API function dma\_pool\_create()

* + - * 1. Notes:

This IOCTL sets the maximum size of the meta data buffers size allocations. Each allocation attempt will consume this size block of DMA memory from the system,

This value can only be set 1 per IOCTL\_DEVICE\_STATE when the nvme\_state={DISABLE or DISABLE\_COMPLETELY}. A IOCTL\_SET\_STATE of this form will remove the DMA allocation pool and unlearns the previously set size, which allows a subsequent IOCTL\_METABUF\_CREATE to be sent again with new values.

* + - 1. IOCTL\_METABUF\_ALLOC
         1. u16 meta\_id

The dnvme uniquely assigned buffer ID. This must be remembered by user space so that when it comes time to delete a buffer the meta\_id will be the reference which indicates which buffer to delete/free

* + - * 1. NOTES:

This buffer must be DWORD aligned per nvme spec

The size of buffer allocated must already be setup by a previous call to IOCTL\_METABUF\_CREATE

This allocation, is successful, creates a new node in metrics\_device\_list[this\_device].metric\_meta\_data.metrics\_meta[append]

Assign value metrics\_device\_list[this\_device].metrics\_device.meta\_unique\_cnt to metrics\_device\_list[this\_device].metric\_meta\_data.metrics\_meta[append].meta\_id.

Increment metrics\_device\_list[this\_device].metrics\_device.meta\_unique\_cnt; for the next new buffer which may be needed.

A call to mmap must be done to map this allocated memory into user space so that tnvme may write and read to it.

* + - 1. IOCTL\_METABUF\_DELETE
         1. u16 meta\_id

The buffer ID of a previously allocated meta data buffer, which must have been allocated via IOCTL\_METABUF\_ALLOC.

* + - * 1. Notes:

**IMPORTANT:**

**We are NOT going to prevent tnvme from deleting a meta data buffer before a cmd is done using it. The prevention of this horrible scenario will be left up to the implementation of tnvme classes to prevent this. This could cause a kernel panic if not prevent. At some point we can’t prevent such catastrophes, but we can make it hard to do them. The power of this design is that we risk shooting ourselves in the foot.**

Algorithm:

Find the meta data node which describes the memory which is being requested to be freed. Traverse metrics\_device\_list[this\_device].metric\_meta\_data.metrics\_meta[x].meta\_id and find a match to the passed in value ‘meta\_id’

Call this node metrics\_device\_list[this\_device].metric\_meta\_data.metrics\_meta[one\_we\_found].meta\_id for future reference

tnvme must make certain to unmap this memory for it is being freed back to the system. Failure to do this could allow the user space app to write into kernel DMA memory causing unpredictable results.

Free the kernel meta data memory back to the system

delete the node metrics\_device\_list[this\_device].metric\_meta\_data.metrics\_meta[one\_we\_found] because it is now freed

* + - 1. Changes to IOCTL\_SEND\_64B\_CMD
         1. Remove parameter ‘meta\_buf\_size’ it is no longer needed.
         2. Remove parameter ‘meta\_buf\_ptr’ it is no longer needed
         3. Add a new parameter ‘meta\_buf\_id’

This is the buffer ID assigned by the kernel when the buffer was allocated via IOCTL\_METABUF\_ALLOC

It’s important to use a meta\_buf\_id to represent the meta data buffer, because if it was deleted before a cmd arrives in the kernel there won’t be any way to lookup the virtual kernel address to get the corresponding physical pointer to use in MPTR. Thus we will be safe from a catastrophe.

* + - * 1. If MASK\_MPTR is present in the bitmask then perform the following:

Search metrics\_device\_list[this\_device].metric\_meta\_data.metrics\_meta[x] and find the corresponding matching ‘meta\_buf\_id’.

If a node is absent then fail this IOCTL request.

else use the metrics metrics\_device\_list[this\_device].metric\_meta\_data.metrics\_meta[one\_we\_found].vir\_kern\_addr to create a physical pointer.

Use the physical pointer and write it to the CMD.DW4, CMD.DW5 within the cmd.

* + - 1. Changes to IOCTL\_REAP
         1. None.
      2. Changes to IOCTL\_DEVICE\_STATE
         1. The disabling and disabling\_completely must now free all previously allocated meta data buffers back to the system. Thus everything within metrics metrics\_device\_list[this\_device].metric\_meta\_data will be freed.
    1. Mmap device Operation: This call maps kernel virtual address to user space virtual address.
       1. Implement system call for mmap and it should return a virtual memory area into which the contiguous memory is mapped.
       2. vm->vm\_pgoff is used to determine SQ\_ID, CQ\_ID or Meta ID. The details on how to determine are as follows:
          1. In the User App:

User app passes off\_t offset parameter in its mmap call.

The bits 28 and 29 are used as below: (The offset is 32 bits in length and has 0 to 31 bit positions)

|  |  |  |  |
| --- | --- | --- | --- |
| **29th bit** | **28th bit** | **Type** | **ID (16 Bits)** |
| 0 | 0 | CQ Type | Bits 12 to 27 indicate CQ Id |
| 0 | 1 | SQ Type | Bits 12 to 27 indicate SQ Id |
| 1 | 0 | Meta Data | Bits 12 to 27 indicate meta id. |

To send sq\_id = 5, the offset value should be 0x10005000. (0x10005 \* PAGE\_SIZE)

To send cq\_id = 5, the offset value should be 0x5000. (0x5 \* PAGE\_SIZE)

To send meta\_id = 5, the offset value should be 0x20005000.

* + - * 1. In the kernel,

The offset value in the device call is already divided by PAGE\_SIZE. So we obtain the offset in kernel as (offset/4096).

Due to the division by PAGE\_SIZE, Check the 17th and 16th bits to determine the type of the Q or meta data. If 17th bit is 0 and 16th bit is 1 then we got SQ type with bits 0 to 15 indicating the SQ\_ID.

If the 17th and 16th bits are 0 then 0 to15 bits indicate CQ ID.

If the 17th bit is 1 and 16th bit is 0 then we have Meta data, and 0 to 15 bits indicate the meta\_id.

* + - * 1. Once the type and ID is determined, the next step is to map this memory into user space and return virtual address to user space which should be accessing the same memory that was allocated in Kernel.