* + 1. **Document VERSION 1.0**
    2. Define structures
       1. 
       2. Define Q related structures
          1. struct isr\_track

U16 intVec;

What interrupt vector is associated with CQ’s.

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

U16 cq\_id[]

linked list of U16 values, where each value indicates the CQ ID for which the ISR for the ‘intVec’ must process

See section

* + - * 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.

U16 sq\_id

What is the SQ ID where this cmd was submitted.

Combined with the unique\_id we can exactly locate a unique cmd for this entire device. Driver could be controlling > 1 device thus we have metrics on a per device basis to avoid collisions.

U8 opcode

cmd opcode

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

What cmd set does the opcode belong?

prp\_element 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.

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.

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.

* + - * 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 irqEnable

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; 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 request not when the ISR fires. Updates to the tail\_ptr occur at this point in time.

If enabled; then the ISR updates the tail\_ptr at this point in time and later when the IOCTL\_REAP\_INQUIRY is received the user app can learn of items to be reaped because the ISR did the math already. No traversing the CQ during this call is needed since that is the job of the ISR. Simply return the number of elements contained within the CQ as calculated by the ISR. This validates the ISR is working.

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.

U32 size

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

* + - 1. struct prp\_element
         1. U8 \*prp\_list; // points to a PRP list, NULL indicates there is no PRP list
         2. U8 num\_pages; // how many physical pages consist of this PRP list? Last ptr in each page will point to next PRP list.
         3. U16 sq\_id; // ID of the SQ for which this PRP list is being used
         4. U16 unique\_id; // unique ID of the cmd within SQ for which this PRP list was created.
      2. struct metrics\_driver
         1. struct public\_driver

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

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.

Note that the ACQ must use IRQ 0 to reap SCQ items and that polling will only be supported for IO CQ’s .

Thus metrics\_device\_list[].metrics\_cq\_list[].public\_cq.irqEnabled == true for ACQ always without exception.

U16 driver\_version;

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

U16 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.

* + - * 1. struct private\_driver

TBD/unknown.

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

prp\_persist\_list

linked list of prp\_element

contains all persistent PRP lists for the device, this will hold PRP list for Admin Create IO Q 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.

NOTE:

If a buffer of memory can be described within a single physical page of memory then a PRP list won’t be necessary. This could be a reason something may not be found within this list. Sometimes we are allowed to describe memory in 2 PRP elements and also a PRP list wouldn’t be necessary here.

metrics\_cq\_list

linked list of struct metrics\_cq

metrics\_sq\_list

linked list of struct metrics\_sq

isr\_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. IOCTL\_GET\_DRIVER\_METRICS
       1. returns metrics\_driver.public\_driver
    2. 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. IOCTL\_DEVICE\_STATE
       1. enum {ST\_ENABLE, ST\_DISABLE, ST\_DISABLE\_COMPLETELY} newState
          1. ST\_DISABLE\_COMPLETELY; is actually a controller reset; completely destroys even the ASQ and ACQ’s; consider this a clean up the kernel call which everything is dropped on the floor and freed.

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, but still perform the cleanup as follows anyway because cleanup is a main function of this call.

Drop all outstanding cmds from all SQ’s on the floor.

For all SQ’s within metrics\_device\_list[this\_device].metrics\_sq\_list[i] do the following

For each node contained within metrics\_device\_list[this\_device].metrics\_sq\_list[i].private\_sq.cmd\_track\_list[j] do the following:

free PRP list pointed to by metrics\_device\_list[this\_device].metrics\_sq\_list[i].private\_sq.cmd\_track\_list[j].prp\_non\_persist.prp\_list if this is not a NULL pointer.

Remove the element at metrics\_device\_list[this\_device].metrics\_sq\_list[i].private\_sq.cmd\_track\_list[j]

This cmd is dropped on the floor.

Cleanup all SQ, CQ’s; q\_id!=0 are these

Traverse metrics\_device\_list[this\_device].metrics\_sq\_list and metrics\_device\_list[this\_device].metrics\_cq\_list and do the following:

free memory metrics\_device\_list[this\_device].metrics\_sq\_list[i].private\_sq.vir\_kern\_add and metrics\_device\_list[this\_device].metrics\_cq\_list[i].private\_cq.vir\_kern\_add as appropriate

remove this element from the linked list.

Traverse metrics\_device\_list[this\_device].prp\_persist\_list and do the following:

free each element of the metrics\_device\_list[this\_device].prp\_persist\_list because this points to a memory page containing PRP elements, i.e. this is a PRP list and the last element could be pointing to another list of pointers. Thus use prp\_list.persist.num\_pages to decide how many pages to free. Remember we are deleting the PRP list of pages, not what those pages are pointing to.

remove this element from the linked list.

Cleanup ASQ; q\_id==0 is the ASQ

Traverse the metrics\_device\_list[this\_device].metrics\_sq\_list

free memory metrics\_device\_list[this\_device].metrics\_sq\_list[i].private\_sq.vir\_kern\_add

remove this element from the linked list.

Verify no items remain in the metrics\_device\_list[this\_device].metrics\_sq\_list, if not then we have a logical bug. log some type of exception in this case.

Cleanup ACQ; q\_id==0 is the ACQ

Traverse the metrics\_device\_list[this\_device].metrics\_cq\_list

free memory metrics\_device\_list[this\_device].metrics\_cq\_list[i].private\_cq.vir\_kern\_add

remove this element from the linked list.

Verify no items remain in the metrics\_device\_list[this\_device].metrics\_cq\_list, if not then we have a logical bug. log some type of exception in this case.

Write 0x00 to ACQ, ASQ and AQA registers

Can’t have the hardware thinking there are admin Q’s if we accidentally re-enable it.

All outstanding SQ elements were dropped on the floor, for we destroyed those Q’s and stopped tracking them

All outstanding CQ elements were dropped on the floor, for we destroyed those Q’s and stopped tracking them

All outstanding ASQ elements were dropped on the floor, for we destroyed those Q’s and stopped tracking them

All outstanding ACQ elements were dropped on the floor, for we destroyed those Q’s and stopped tracking them

* + - * 1. ST\_DISABLE; is actually a controller reset; leaves the ASQ and ACQ’s if there were constructed but resets those Q pointers and drop s cmds on the floor; consider this a clean up the kernel call which everything is dropped on the floor and freed.

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, but still perform the cleanup as follows anyway because cleanup is a main function of this call.

Drop all outstanding cmds from all SQ’s on the floor.

For all SQ’s within metrics\_device\_list[this\_device].metrics\_sq\_list[i] do the following

For each node contained within metrics\_device\_list[this\_device].metrics\_sq\_list[i].private\_sq.cmd\_track\_list[j] do the following:

free PRP list pointed to by metrics\_device\_list[this\_device].metrics\_sq\_list[i].private\_sq.cmd\_track\_list[j].prp\_non\_persist.prp\_list if this is not a NULL pointer.

Remove the element at metrics\_device\_list[this\_device].metrics\_sq\_list[i].private\_sq.cmd\_track\_list[j]

This cmd is dropped on the floor.

Cleanup all SQ, CQ’s; q\_id!=0 are these

Traverse metrics\_device\_list[this\_device].metrics\_sq\_list and metrics\_device\_list[this\_device].metrics\_cq\_list and do the following:

free memory metrics\_device\_list[this\_device].metrics\_sq\_list[i].private\_sq.vir\_kern\_add and metrics\_device\_list[this\_device].metrics\_cq\_list[i].private\_cq.vir\_kern\_add as appropriate

remove this element from the linked list.

Traverse metrics\_device\_list[this\_device].prp\_persist\_list and do the following:

free each element of the metrics\_device\_list[this\_device].prp\_persist\_list because this points to a memory page containing PRP elements, i.e. this is a PRP list and the last element could be pointing to another list of pointers. Thus use prp\_list.persist.num\_pages to decide how many pages to free. Remember we are deleting the PRP list of pages, not what those pages are pointing to.

remove this element from the linked list.

Cleanup ASQ; q\_id==0 is the ASQ

Traverse the metrics\_device\_list[this\_device].metrics\_sq\_list

reset all the pointers to default values, this Q must be init’d like it was just created.

Verify no 1 remains in the metrics\_device\_list[this\_device].metrics\_sq\_list and that it must be q\_id=0

Cleanup ACQ; q\_id==0 is the ACQ

Traverse the metrics\_device\_list[this\_device].metrics\_cq\_list

reset all the pointers to default values, this Q must be init’d like it was just created.

Verify no 1 remains in the metrics\_device\_list[this\_device].metrics\_sq\_list and that it must be q\_id=0

All outstanding SQ elements were dropped on the floor, for we destroyed those Q’s and stopped tracking them

All outstanding CQ elements were dropped on the floor, for we destroyed those Q’s and stopped tracking them

All outstanding ASQ elements were dropped on the floor, for we reset the pointers within metrics\_sq\_list. and stopped tracking them

All outstanding ACQ elements were dropped on the floor, for we reset the pointers within metrics\_cq\_list. and stopped tracking them

* + - * 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 for the Q, maybe must use kmalloc() for this.
          2. if (type==ADMIN\_SQ) then allocate contiguous kernel memory in (bytes = elements \* 64B)

if (bytes > 4096) then fail this request

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

if (bytes > 4096) then fail this request

* + - * 1. this value is specified as a 1-based value
        2. 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.

* + - 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. The spec only allows admin Q creation or modification while the controller is disabled, See IOCTL\_DEVICE\_STATE, p.37 of spec rev/ 1.0b. This IOCTL does not check this requirement; rather it will be left up the user app to guarantee this is the sitation.
    1. IOCTL\_ALLOCATE\_SQ
       1. U16 elements
          1. Total number of entries/elements that needs to be allocated in the kernel
          2. total memory allocated in ‘bytes’ = (elements \* CC.IOSQES)
          3. This is a 1-based value
       2. U8 discontinuity;
          1. ==0 indicates false, where discontinuities are invoked by using vmalloc() Kernel call. How the buffer is discontinuous cannot be specified, accept what the driver creates.
          2. ==!0 indicated true, where it is continuous are invoked using kmalloc() Kernel call.

If continuous was requested but the kernel cannot satisfy one as large as requested, then this should fail.

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

allocate discontinuous or continuous as requested and allocated ‘bytes’ number of memory

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.

Do NOT create a node within the metrics\_device\_list[this\_device].prp\_persist\_list, this will be done when the user passes the virtual address to IOCTL\_SEND\_64B\_CMD.

* + - * 1. The 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 will assume a previous IOCTL\_ALLOOCATE\_SQ was issue to actually create the memory which is being references within that cmd. Thus the \_device\_list[this\_device].metrics\_sq\_list[append].private\_sq.vir\_kern\_addr will be the pointer to that memory.
    1. IOCTL\_ALLOCATE\_CQ
       1. U16 elements
          1. Total number of entries/elements that needs to be allocated in the kernel
          2. total memory allocated in ‘bytes’ = (elements \* CC.IOCQES)
          3. This is a 1-based value
       2. U8 discontinuity;
          1. ==0 indicates false, where discontinuities are invoked by using vmalloc() Kernel call. How the buffer is discontinuous cannot be specified, accept what the driver creates.
          2. ==!0 indicated true, where it is continuous are invoked using kmalloc() Kernel call.

If continuous was requested but the kernel cannot satisfy one as large as requested, then this should fail.

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

allocate discontinuous or continuous as requested and allocated ‘bytes’ number of memory

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.

Do NOT create a node within the metrics\_device\_list[this\_device].prp\_persist\_list, this will be done when the user passes the virtual address to IOCTL\_SEND\_64B\_CMD.

* + - * 1. The 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 will assume a previous IOCTL\_ALLOOCATE\_CQ was issue to actually create the memory which is being references within that cmd. Thus the \_device\_list[this\_device].metrics\_cq\_list[append].private\_csq.vir\_kern\_addr will be the pointer to that memory.
    1. IOCTL\_SEND\_64B\_CMD
       1. U16 q\_id
          1. the queue ID to which ‘cmdBuf’ must be copied into the next available location, queue 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 and is the metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].public\_sq.tail\_ptr\_vir
       2. enum {CMD\_ADMIN, CMD\_NVME, CMD\_AON} cmdSet;
          1. User app passes in what cmd set this cmd belongs
       3. U16 bitMask
          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.

see “Algorithm to populate PRP1/PRP2”

* + - * 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.

see “Algorithm to populate PRP1/PRP2”

* + - * 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 'dataBufSize’ indicates the total, exact number of bytes being xferr’d by this cmd and the MASK\_PRP1 and/or MASK\_PRP2 indicate whether or not we can use those fields to describe the ‘dataBuf’ to the hdw. If a ‘dataBuf’ is passed and no MASK\_PRPx bits are set then obviously return with an error.

if (dataBufSize == 0) then skip this algorithm and do not touch PRP1 nor PRP2 of the cmd.

if (dataBufSize > CC.MPS) then we may have to create a PRP list for this cmd because PRP1 isn’t enough to describe this entire buffer

if (dataBufSize > (2\* CC.MPS)) then we definitely will have to create a PRP list for this cmd because PRP1 and PRP2 arn’t enough to describe this entire buffer

The exact details of converting the ‘userBuf’ to 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.

Create a new node in metrics\_device\_list[this\_device].prp\_persist\_list[append] if we are creating an IO SQ or an IO CQ.

Create a new node in metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].private\_sq.cmd\_track[append].prp\_nonpersist[append] for all other cmds.

Determine where to place this new node by investigating if ((cmdSet==CMD\_ADMIN) && (CMD.opcode==0x01)) then we are creating IO SQ. if ((cmdSet==CMD\_ADMIN) && (CMD.opcode==0x05)) then we are creating IO CQ. Persistent PRP lists must remain until a controller reset occurs or a Delete IO Q Cmd is issued.

Obviously if the system cannot provide enough dynamic kernel memory to make this PRP list this should be considered a failure

* + - 1. U8 \*cmdBuf
         1. Virtual address pointer to 64B cmd, buffer 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. Then this pointer should be incremented to point to the next available location taking into account wrapping.
         2. if (cmdBuf == NULL) then fail this request
      2. U32 dataBufSize
         1. size in bytes of dataBuf

if (dataBufSize !=0) && (cmdBuf == NULL) then fail this request

if (dataBufSize ==0) && (cmdBuf != NULL) then fail this request

if (dataBufSize!= 0) and MASK\_PRP1 is missing then fail this request

* + - 1. U8 \*dataBuf
         1. Virtual user address space pointer to the allocated data buffer to associate with the cmd. Refer to section
         2. 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.
      2. U16 metaBufSize
         1. size in bytes of metaBuf

if (metaBufSize!=0) && (metaBuf == NULL) then fail this request

if (metaBufSize ==0) && (metaBuf!= NULL) then fail this request

if (metaBufSize != 0) and MASK\_MPTR is missing then fail this request

if (metaBufSize > CC.MPS) then fail this request.

* + - 1. U8 \*metaBuf
         1. Virtual user address space pointer to the user allocated meta data buffer to associate with the cmd.
         2. if MASK\_MPTR is set then convert this address to a physical address and write it into CMD.MTPR field of this cmd.

otherwise do not write into CMD.MPTR

* + - 1. NOTES:
         1. Does NOT ring the doorbell
         2. CMD.DW3.CID must be overwritten by the driver.

The driver increments the variable in metrics metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].private\_sq.unique\_cmd\_id and then write that new value to CMD.DW3.CID for every cmd submitted to guarantee uniqueness for the entire device per each SQ.

* + - * 1. Creating IO Q’s

if ((cmdSet == CMD\_ADMIN) & (CMD.opcode == 0x01 or 0x05))

if (metrics\_device\_list[this\_device].metrics\_?q\_list[append].private\_?q.vir\_kern\_addr == NULL)

return error because the user never called the appropriate IOCTL\_ALLOCATE\_?Q to create the Q memory.

if (dataBuf != NULL)

return error because for the creation of an IO Q there will never be a buffer, rather the address stored at metrics\_device\_list[this\_device].metrics\_?q\_list[append].private\_?q.vir\_kern\_addr is being used as the start of this Q’s memory.

The address metrics\_device\_list[this\_device].metrics\_?q\_list[append].private\_?q.vir\_kern\_addr must be run through the algorithm to create a PRP list and if a list is necessary if must be placed within metrics\_device\_list[this\_device].prp\_persist\_list[append] because it is considered persistent.

* + - * 1. Cmd tracking

Create a new node within metrics\_device\_list[this\_device].metrics\_sq\_list[i].private\_sq.cmd\_track\_list[append] only when…

if ((cmdSet == CMD\_ADMIN) && (CMD.opcode == 0x00 or 0x01 or 0x04 or 0x05)) or…

The ‘dataBuf’ requires a PRP list.

How to create a new node

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

Look into the ‘cmdBuf’ and remove the opcode and copy that to metrics\_device\_list[this\_device].metrics\_sq\_list[i].private\_sq.cmd\_track\_list[append].opcode

copy ‘cmdSet’ to metrics\_device\_list[this\_device].metrics\_sq\_list[i].private\_sq.cmd\_track\_list[append].cmdSet

copy metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].private\_sq.unique\_cmd\_id to metrics\_device\_list[this\_device].metrics\_sq\_list[i].private\_sq.cmd\_track\_list[append].unique\_id

* + 1. 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

if (q\_id != 0) then Completion Entry Size (CES) = CC.IOCQES.

* + - 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. Algorithm

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

num\_to\_reap = 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

The ISR is responsible for updating the metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].public\_cq.tail\_ptr so this routine should **NOT** also do it.

else // no ISR to update the tail ptr, we must do that now, here

Traverse the CQ from metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].public\_cq.head\_ptr forward and note the P-bit within the CE. Traverse until the P-bit indicates the element is not a new CE placed there by the hdw, this is the location of the metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].public\_cq.tail\_ptr.

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

num\_to\_reap = 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

* + 1. ISR for CQ’s (This is not an IOCTL)
       1. An ISR will be notified by a hardware interrupt when items are placed into various CQ’s. All ISR’s must figure out which CQ’s they are responsible for performing the following algorithm upon.
          1. Traverse metrics\_device\_list[this\_device].isr\_track\_list[i].intVec until you find the interrupt vector triggered for this ISR.
          2. This ISR is thus responsible for all CQ’s listed in metrics\_device\_list[this\_device].isr\_track\_list[this\_ISR\_INT\_VEC].cq\_list[]
       2. Algorithm
          1. The ISR must perform this calculation for each CQ it is responsible for monitoring

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

log this as an exception, how can a ISR be active but the configuration states that this CQ does not have an associated ISR?

else

Traverse the CQ from metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id\_monitoring].public\_cq.head\_ptr forward and note the P-bit within the CE. Traverse until the P-bit indicates the element is not a new CE placed there by the hdw, this is the location of the metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id\_monitoring].public\_cq.tail\_ptr.

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

Next CQ to process, continue or exit if done

* + 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 CES = 16B

if (q\_id != 0) then CES = CC.IOCQES.

* + - 1. U16 element
         1. Pass the number of completion elements to reap. if (size < (element \* <CES>)) then fail this request
         2. If the number of ‘element’ requested can’t be satisfied, because there aren’t enough CE’s to be reaped, then fail this request.
         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 to be reaped.
         4. num\_could\_reap = IOCTL\_REAP\_INQUIRY(‘q\_id’);

Effectively call IOCTL\_REAP\_INQUIRY internally to update the tail\_ptr and find out how many CE’s could be reaped at this time.

* + - 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’.

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

This is a 1-based number

* + - 1. 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.
      2. U8 \*buffer;
         1. Pass the user mode virtual buffer to which the completion elements will be copied starting from index = 0.
      3. NOTES:
         1. Definitions:

Completion Entry (CE)

Completion Entry Size (CES)

* + - * 1. Purpose is to copy ‘element’ completion items into ‘buffer’ for the CQ with ID == ‘q\_id’.
        2. ALGORITHM: reaping on CQ with ID=0, i.e algorithm specific to ACQ

Start reaping at CE pointed to by = metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].public\_cq.head\_ptr and stop reaping at when ‘element’ CE’s have been processed.

This IOCTL is responsible for updating metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].public\_cq.head\_ptr at the end of this algorithm and writing this new value to the appropriate CQxHDBL register.

For each ‘element’ we are told to reap from CQ with ‘q\_id’ do the following:

Perform a lookup of the CE in our list of issued trackable cmds.

The lookup of a cmd may not be successful, not all cmds will be tracked, only specific ones need special attention.

Use CE.sq\_id and CE.cmd\_id and located the cmd within metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].private\_sq.cmd\_track[i]

Call this index into cmd\_track\_list[] “current\_cmd”, such that metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].private\_sq.cmd\_track[current\_cmd] references the cmd data for this cmd.

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

If a lookup actuall found an item in the trackable cmd list then do the following:

if (metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].private\_sq.cmd\_track[current\_cmd].opcode == 0x01 or 0x05) and (metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].private\_sq.cmd\_track[current\_cmd].cmdSet == CMD\_ADMIN) // Create IO Q’s

if (CE.status == failure)

Search metrics\_device\_list[this\_device].prp\_persist\_list[i] for the node which describes the PRP list for this cmd. It may not exist if the Q size was small enough where it didn’t need a PRP list. But if one is found, then free the PRP list memory pointed to by metrics\_device\_list[this\_device].prp\_persist\_list[i].prp\_list

free either metrics\_device\_list[this\_device].metrics\_cq\_list[i].private\_cq.vir\_kern\_add or metrics\_device\_list[this\_device].metrics\_sq\_list[i].private\_sq.vir\_kern\_add depending upon the opcode.

remove the node from either metrics\_device\_list[this\_device].metrics\_cq\_list[i] or metrics\_device\_list[this\_device].metrics\_sq\_list[i] depending upon the opcode

if (metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].private\_sq.cmd\_track[current\_cmd].opcode == 0x00 or 0x04) and (metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].private\_sq.cmd\_track[current\_cmd].cmdSet == CMD\_ADMIN) // Delete IO Q’s

if (CE.status == success)

Search metrics\_device\_list[this\_device].prp\_persist\_list[i] for the node which describes the PRP list for this cmd. It may not exist if the Q size was small enough where it didn’t need a PRP list. But if one is found, then free the PRP list memory pointed to by metrics\_device\_list[this\_device].prp\_persist\_list[i].prp\_list

free either metrics\_device\_list[this\_device].metrics\_cq\_list[i].private\_cq.vir\_kern\_add or metrics\_device\_list[this\_device].metrics\_sq\_list[i].private\_sq.vir\_kern\_add depending upon the opcode.

remove the node from either metrics\_device\_list[this\_device].metrics\_cq\_list[i] or metrics\_device\_list[this\_device].metrics\_sq\_list[i] depending upon the opcode

else // else clause from 2) above

if ((CE.status == success) or (CE.status == failure))

if (metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].private\_sq.cmd\_track[current\_cmd].prp\_list != NULL) then this PRP list needs to be freed, otherwise there is no PRP list assoc with this cmd.

Finalize processing this CE; always do this processing.

Copy this CE into ‘dataBuf’

We may have to return > 1 CE so keep this in mind. Indicated by ‘element’ parameter.

Remove the node from metrics\_device\_list[this\_device].cmd\_track\_list[current\_cmd] for we have completely processed this cmd now. Realize there may not be this cmd in this list, only certain cmds actually make it to this list.

Update the appropriate register CQxHDBL with the new value since we have just processed this CE.

Actually you should do 1 write at the end of all processing in case we have to return more ‘element’ to the user. This will reduce the number of writes to CQxHDBL.

* + - * 1. ALGORITHM: reaping on CQ with ID!=0, i.e algorithm specific to all IO CQ’s**. Lookup CC.CSS and follow the appropriate algorithm based upon the cmd set chosen.**

NVM Command Set

Start reaping at CE pointed to by = metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].public\_cq.head\_ptr and stop reaping at when ‘element’ CE’s have been processed.

This IOCTL is responsible for updating metrics\_device\_list[this\_device].metrics\_cq\_list[q\_id].public\_cq.head\_ptr at the end of this algorithm and writing this new value to the appropriate CQxHDBL register.

For each ‘element’ we are told to reap from CQ with ‘q\_id’ do the following:

Perform a lookup of the CE in our list of issued trackable cmds.

The lookup of a cmd may not be successful, not all cmds will be tracked, only specific ones need special attention.

Use CE.sq\_id and CE.cmd\_id and located the cmd within metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].private\_sq.cmd\_track[i]

Call this index into cmd\_track\_list[] “current\_cmd”, such that metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].private\_sq.cmd\_track[current\_cmd] references the cmd data for this cmd.

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

if ((CE.status == success) or (CE.status == failure))

if (metrics\_device\_list[this\_device].metrics\_sq\_list[q\_id].private\_sq.cmd\_track[current\_cmd].prp\_list != NULL) then this PRP list needs to be freed, otherwise there is no PRP list assoc with this cmd.

Finalize processing this CE; always do this processing.

Copy this CE into ‘dataBuf’

We may have to return > 1 CE so keep this in mind. Indicated by ‘element’ parameter.

Remove the node from metrics\_device\_list[this\_device].cmd\_track\_list[current\_cmd] for we have completely processed this cmd now. Realize there may not be this cmd in this list, only certain cmds actually make it to this list.

Update the appropriate register CQxHDBL with the new value since we have just processed this CE.

Actually you should do 1 write at the end of all processing in case we have to return more ‘element’ to the user. This will reduce the number of writes to CQxHDBL.

AON Command Set

TBD/unknown