# Peanut NVMe Extension Module 1.1.0

User Manual v1

Mar. 2024

## Disclaimer and Copyrights

All information herein is the property of and owned solely by Cell Computers Ltd. who shall have and keep the sole right to file patent applications or any other kind of intellectual property protection in connection with such information.

Changes are periodically added to the information herein. Furthermore, Cell Computers Ltd. reserves the right to make any change or improvement in the specifications data, information, and the like described herein, at any time without notice.

# Contents

# Introduction

This extension module is to provide Peanut extension functions for accessing NVMe SSDs in command level. Besides, this extension module also defines constants and structures for a higher-level programming; this module also supports run-time command trace and statistics. All of them are provided to ease the programming tasks to qualify NVMe SSD products.

## Namespace

The namespace[1] `nvme` is created under Peanut global namespace by this extension.

All the extension constants and functions registered to Peanut can only be used/called with namespace resolution `nvme::` preceded. Two namespaces, `admc` and `ioc`, are created under the `nvme` namespace; namespaces created by this module are as the following.

| | |
|---|---|
| `nvme::admc::` | For extension functions to issue commands through Admin Submission Queue |
| `nvme::ioc::` | For extension functions to issue commands through I/O Submission Queues |
| `nvme::` | For extension functions and constants used to initialize this extension module, get information of devices, allocate buffers, get command status, list command history, access configuration space and controller registers …, etc. |

## Programming Model

A NVMe command is issued to the device by calling a corresponding extension function with required arguments; a submission queue handle or a controller handle is one of the arguments. A IO submission queue handle must be specified for sending an IO command and the controller handle must be specified for sending an Admin command.

If the function is not able to issue the command, a zero-value command handle is returned to the caller function and an error code is updated through a call-by-reference argument.

If the function successfully sends a command to the device, a command handle is returned from the callee function. The command handle is used to poll the completion status of the command and the `nvme::pspp()` function is called with the command handle to retrieve the following information:

(1) the completion flag
(2) time-out flag
(3) command Status Field value; that is bit17:31 of DW3 of the completion queue entry
(4) the Command Specific value; that is DW0 of the completion queyeentry

[1] The namespace here is the namespace defined by the Peanut language. The namespace here does not mean a collection space of LBA data in a NVMe controller.

Issuing multiple commands can be achieved by calling corresponding functions iteratively. Except fused commands that the programmer needs to pay attention to their issuing order, there exists no dependencies between any two commands. At any moment, it's the programmer's freedom to send a new commmand to the device or to poll the command status for an sent command.

It is safe for multiple tasks to send commands into same Submission Queues. Sending one command in a task and polling the command status in another task is also allowed.

## Programming Note

We design the function interfaces in the way that a reader interprets the parameters intuitively. Say there is a function requiring to pass size information, then the function would not be designed to adapt 0's based approach for passing size information but 1's based approach would be used. This kind of interfaces would be seen in functions implemented for sending NVMe commands for which NVMe spec. adapts 0's based approach to interpret some CDW value(s). To enhance the difference between the function's definition and the specification's definition, the function parameter fields would be displayed in the bold font in the following paragraph. But if the function parameter would be "passed through" to the CDWs and the function would not interpret the parameter, the parameter value would follow what the spec. defines and 0's based scheme would be used for that parameter.

## Example Note

In the following paragraph, each supported function is depicted by a table and an example code; the following illustrate flReset() function as the example. We want the readers note that the example code would focus on the described function's input(s), output(s), and related functions to which are greatly related to. For example init() function is highly related to flReset() and is seen in the example code of flReset() function; but the example code may not include the statements declaring the variables; like below example code, no statement declaring variable EC and cntlHandle; the example code may either loss some detailed flow(s) how to get e.g. a controller handle or a submission queue handle, or a buffer handle; some lines with dot-dot-dot (. . .) may be seen.

Also note that functions referred with c:: prefix in the examples are implemented in Peanut CLib extension module.

| Prototype | `flReset(&EC, cntlHdle)` | |
|---|---|---|
| Description | Do a function level reset on the specified controller. | |
| Return Value | 0 : failed<br>1 : successful | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |

| | | |
|---|---|---|
| | cntlHdle | A controller handle. |
| Note | After a function level reset, the controller should be initialized again for further accesses. | |
| See Also | initA(), init() | |

```
    c::printf("Function Level Reset:\n");
    if (nvme::flReset(EC, cntlHandle))
        c::printf("Info: Function Level Reset success\n");
    else
    {
        c::printf("Error: Function Level Reset fail; error = %d\n", EC);
        return (0);
    }
    nvme::init(EC, cntlHandle);
```

# Terminologies

| | |
|---|---|
| Device | A physical SSD card. A device is represented a controller handle and the controller is a physical function controller. |
| Device Number | An integer to represent a scanned device. Say there are N devices scanned, these devices would be numbered from 0 to N-1. |
| Function | A PCIe function. A function can be physical or virtual. |
| Function Number | An integer to represent a PCIe function within a device.<br>If the device has only one function, the valid function number within the device is 0.<br>If the device has multiple functions and the number of active functions is N, valid function numbers are integers range from 0 to N-1. Function 0 is a physical function. |
| Event | Events are defined as important things happened on a controller, including power off events, power on events, and resets. These events would be recorded for tracing. |
| DDB | DMA data buffer. Both the host and the device can access. A DMA data buffer is used for data transfer. |
| SQId | Submission queue Id. The Admin submission queue Id of a function is 0. IO submission queue Id ranges from 1 to N if N IO submission queues are created. |
| CQId | Completion queue Id. The Admin completion queue Id of a function is 0. IO completion queue Id ranges from 1 to N if N IO completion queues are created. |
| Controller Handle | An integer value representing a controller, a primary or secondary controller. |
| Namespace Handle | An integer value representing a namespace; a collection of logic blocks. |
| SQ Handle | An integer value to represent a Submission Queue. |
| Command Handle | An integer value to represent a sent command. |
| Event Handle | An integer value to represent one event information. |
| Buffer Handle | An integer value to represent a buffer. |
| Command Status | A 64-bit integer indicating the status of a sent command. The 64-bit value convey a command completion flag, a command time-out flag, the Status Field of the completion queue entry, and the Command Specific value of the completion queue entry. |

## Extension constants in namespace `nvme`

| Name | Description |
|---|---|
| `majorVer` | Major version number of this extension module. |
| `minorVer` | Minor version number of this extension module. |
| `patchVer` | Patch version number of this extension module. |
| `version` | Version string of this extension module. |
| `cmdCplFlag` | A 1-bit constant value; indicating the completion of a sent command. |
| `cmdToFlag` | A 1-bit constant value; indicating time-out of a sent command. |
| `cmdStsMask` | The mask for retrieving the Status value (DW3 bit17~bit31 in a CQ entry) from the 64-bit CmdStatus value returned by `nvme::pspp`() function. |
| `cmdStsLowBit` | The low bit position of the Status value within the 64-bit CmdStatus value returned by `nvme::pspp`() function. |
| `cmdSpcStsMask` | The mask for retrieving the Command Specific value (DW0 in a CQ entry) from the 64-bit CmdStatus value returned by `nvme::pspp`() function. |
| `cmdSpcStsLowBit` | The low bit position of the Command Specific value within the 64-bit CmdStatus value returned by `nvme::pspp`() function. |
| `PRP` | A constant value to represent the PRP buffer type. |
| `SGL` | A constant value to represent the SGL buffer type. |
| `HMB` | A constant value to represent the HMB (host memory buffer) buffer type. |

```
showStatus(stsVal)
{
    var Status, SCTDescp, SCDescp;
    var nCmdSpec;

    Status = (stsVal & nvme::cmdStsMask) >> nvme::cmdStsLowBit;
    if (Status)
    {
        c::printf("  stsVal = %Xh, Status = %04Xh, ", stsVal, Status);
        GetNVMeCmdCplStatusDescription(Status, SCTDescp, SCDescp);
        c::printf("  SCT: %s, SC: %s\n", SCTDescp, SCDescp);
    }

    nCmdSpec = (stsVal & nvme::cmdSpcStsMask) >> nvme::cmdSpcStsLowBit;
    if (nCmdSpec)
        c::printf("  Command specific status %08Xh\n", nCmdSpec);

    if (stsVal & nvme::cmdToFlag)
        c::printf("  Comand timeout\n");
}

main()
{
    var EC, cntlHandle;
    var n, buf, cmdHandle, stsVal;
    c::printf("NVMe Extension Module version: %s", nvme::version);
```

```
    ...
    cntlHandle = nvme::getCntlHandle(EC, n, 0);
    buf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 512);
    cmdHandle = nvme::getLogPage(EC, cntlHandle, buf, 0xFFFFFFFF);

    do
        stsVal = nvme::pspp(cmdHandle);
    while (stsVal == 0);
    if (stsVal != nvme::cmdCplFlag)
    {
        showStatus(stsVal);
        return;
    }
    ...
}
```

# Extension functions in namespace `nvme`

scan()

| Prototype | `scan(&EC)` | |
|---|---|---|
| Description | Scan nvme SSDs. | |
| Return Value | 0 : failed<br>1 : successful | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| Note | Before calling to other NVMe extension functions, this function should be called. | |
| See Also | `put()` | |

```
main()
{
    var EC;
    var nDevs;

    if (!nvme::scan(EC))
    {
        c::printf("Error: fail to scan devices; error = %d.\n", EC);
        return (0);
    }

    nDevs = nvme::getDevNum();
    c::printf("Number of devices: %d", nDevs);
}
```

put()

| Prototype | `put()` |
|---|---|
| Description | Release allocated resources for scanned devices. |
| Return Value | 1 |
| Parameter | N/A |
| Note | This function is called before the script terminates. |
| See Also | |

```
main()
{
    var EC;
    var nDevs;

    if (!nvme::scan(EC))
    {
        c::printf("Error: fail to scan devices; error = %d.\n", EC);
        return (0);
    }
    nDevs = nvme::getDevNum();
    c::printf("Number of devices: %d", nDevs);

    ...

    nvme::put();
}
```

initA()

| | | |
|---|---|---|
| Prototype | `initA(&EC, cntlHdle)`<br>`initA(&EC, cntlHdle, CBStr)`<br>`initA(&EC, cntlHdle, CBStr, &ASQSz)`<br>`initA(&EC, cntlHdle, CBStr, &ASQSz, &ACQSz)`<br>`initA(&EC, cntlHdle, CBStr, &ASQSz, &ACQSz, IOCQNumPerPCCT)` | |
| Description | Initialize the controller with Admin SQ creation and Admin CQ creation. | |
| Return Value | 0 : failed<br>1 : successful | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | CBStr<br>(optional) | A string of an user function name. The function will be called while an illegal completion queue entry is received. An empty string denotes this initialization does not register a callback function for illegal completion queue entries.<br>The named function should be implemented with 2 parameters. The first one is used to received the number of elements of the 1-dimension array of the second parameter. |
| | ASQSz<br>(optional) | This parameter is a call-by-reference one. The argument should be an elementary variable. The size of the Admin Submission Queue is specified as an input. After the function returns, the real size of the created ASQ is updated to this variable. |
| | ACQSz<br>(optional) | This parameter is a call-by-reference one. The argument should be an elementary variable. The size of the Admin Completion Queue is specified as an input. After the function returns, the real size of the created ACQ is updated to this variable. |
| | IOCQNumPerPCCT<br>(optional) | This argument specifying the number of IO Completion Queues polled by a polling command completion thread (PCCT), which is started by the NVMe extension module. The finalized value is updated to the variable after this function returns. |
| Note | Neither an IO SQ nor an IO CQ is created by this function. While this function is called, the programmer wants to manually create IO Submission Queues and IO Completion Queues. | |
| See Also | init() | |

```
main()
{
    var EC, n, nDevs, nASQSize = 32;
    var cntlHandle;

    if (!nvme::scan(EC))
    {
        c::printf("Error: fail to scan devices; error = %d.\n", EC);
        return (0);
    }
```

```
    nDevs = nvme::getDevNum();
    c::printf("Number of devices: %d", nDevs);

    for (n = 0; n < nDevs; ++n)
    {
        cntlHandle = nvme::getCntlHandle(EC, n, 0);
        nvme::initA(EC, cntlHandle, "", nASQSize);
    }
}
```

```
ICECallback(argc, argv[])
{
    c::printf("Illegal CQ Entry on CQ %d of controller %p\n"
            argv[1], argv[0]);
    c::printf("    CDW1 = %08X\n", argv[2]);
    c::printf("    CDW2 = %08X\n", argv[3]);
    c::printf("    CDW3 = %08X\n", argv[4]);
    c::printf("    CDW4 = %08X\n", argv[5]);
}

main()
{
    var EC, n, nDevs, nASQSize = 32;
    var cntlHandle;

    if (!nvme::scan(EC))
    {
        c::printf("Error: fail to scan devices; error = %d.\n", EC);
        return (0);
    }
    nDevs = nvme::getDevNum();
    c::printf("Number of devices: %d", nDevs);

    for (n = 0; n < nDevs; ++n)
    {
        cntlHandle = nvme::getCntlHandle(EC, n, 0);
        nvme::initA(EC, cntlHandle, "ICECallback", nASQSize);
    }
}
```

init()

<table>
<tr><td>Prototype</td><td colspan="2">init(&EC, cntlHdle)<br>init(&EC, cntlHdle, CBStr)<br>init(&EC, cntlHdle, CBStr, struct nvme::queueInitParams_t &QInitParams)</td></tr>
<tr><td>Description</td><td colspan="2">Initialize the controller with creations of the Admin SQ, Admin CQ, IO CQs, and IO SQs.</td></tr>
<tr><td>Return Value</td><td colspan="2">0 : failed<br>1 : successful</td></tr>
<tr><td rowspan="4">Parameter</td><td>EC</td><td>This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0.</td></tr>
<tr><td>cntlHdle</td><td>A controller handle.</td></tr>
<tr><td>CBStr<br>(optional)</td><td>A string of an user function name. The function will be called while an illegal completion queue entry is received. An empty string denotes this initialization does not register a callback function for illegal completion queue entries.<br>The named function should be implemented with 2 parameters. The first one is used to received the number of elements of the 1-dimension array of the second parameter.</td></tr>
<tr><td>QInitParams<br>(optional)</td><td>This parameter is a call-by-reference one. The argument should be a nvme::queueInitParams_t structure. This structure is defined by the extension module as the following:<br><pre>struct queueInitParams_t
{
  var nASQSize;
  var nACQSize;
  var nIOSQNum;
  var nIOSQSize;
  var nIOSQNumPerIOCQ;
  var nIOCQSize;
  var nIOCQNumPerPCCT;
  setDef() {
      nASQSize = 0;
      nACQSize = 0;
      nIOSQNum = 0;
      nIOSQSize = 0;
      nIOSQNumPerIOCQ = 0;
      nIOCQSize = 0;
      nIOCQNumPerPCCT = 0;
  }
};</pre>The nIOSQNumPerIOCQ field specifies the queue pairing relationship between IOSQs and IOCQs. If this number is N, then the extension module pairs IOSQ 1 ~ IOSQ N with IOCQ 1; pairs IOSQ N+1 ~ IOSQ 2N with IOCQ 2.<br>The nIOCQNumPerPCCT field specifies the number of IOCQs polled by 1 polling command completion thread (PCCT).</td></tr>
</table>

| | | |
|---|---|---|
| | | This function will apply default value for fields whose value is 0 in this structure for the queu creations.<br>The real values that this function used to create queues are updated to this structure after this function returns. |
| Note | This function uses integers 1 ~ N as the SQIds for the creations of the IOSQs if N IOSQs are created.<br>This function uses integers 1 ~ M as the CQIds for the creations of the IOCQs if M IOCQs are created. | |
| See Also | initA() | |

```
main()
{
    var EC, n, nDevs;
    var cntlHandle;
    struct nvme::queueInitParams_t initQParams;

    if (!nvme::scan(EC))
    {
        c::printf("Error: fail to scan devices; error = %d.\n", EC);
        return (0);
    }
    nDevs = nvme::getDevNum();
    c::printf("Number of devices: %d", nDevs);


    for (n = 0; n < nDevs; ++n)
    {
        cntlHandle = nvme::getCntlHandle(EC, n, 0);

        // Note: initQParams might be updated by init()
        //       so initQParams should be refreshed
        initQParams.setDef();
        initQParams.nASQSize = 32;
        initQParams.nACQSize = 32;
        initQParams.nIOSQNum = 4;
        initQParams.nIOSQSize = 64;
        initQParams.nIOSQNumPerIOCQ = 2;

        nvme::init(EC, cntlHandle, "", initQParams);
    }

    ...

    nvme:put();
}
```

flReset()

| | | |
|---|---|---|
| Prototype | `flReset(&EC, cntlHdle)` | |
| Description | Do a function level reset on the specified controller. | |
| Return Value | 0 : failed<br>1 : successful | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| Note | After a function level reset, the controller should be initialized again for further accesses.<br>If virtual functions are enabled, this function will not do the reset. | |
| See Also | initA(), init() | |

```
    c::printf("Function Level Reset:\n");
    if (nvme::flReset(EC, cntlHandle))
        c::printf("Info: Function Level Reset success\n");
    else
    {
        c::printf("Error: Function Level Reset fail; error = %d\n", EC);
        return (0);
    }
    nvme::init(EC, cntlHandle);
```

cntlReset()

| | | |
|---|---|---|
| Prototype | `cntlReset(&EC, cntlHdle)` | |
| Description | Do a controller reset on the specified controller. | |
| Return Value | 0 : failed<br>1 : successful | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| Note | After a controller reset, the controller should be initialized again for further accesses. If virtual functions are enabled, this function will not do the reset. | |
| See Also | initA(), init() | |

```
    c::printf("Controller Reset:\n");
    if (nvme::cntlReset(EC, cntlHandle))
        c::printf("Info: Controller Reset success\n");
    else
    {
        c::printf("Error: Controller Reset fail; error = %d\n", EC);
        return (0);
    }
    nvme::init(EC, cntlHandle);
```

NVMSSReset()

| | | |
|---|---|---|
| Prototype | `NVMSSReset(&EC, cntlHdle)` | |
| Description | Do an NVM Subsystem reset on the specified controller. | |
| Return Value | 0 : failed<br>An event handle : successful | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| Note | After an NVM Subsystem reset, the controller should be initialized again for further accesses. Since an NVM Subsystem reset causes a reestablishment of the PCIe PHY link, the rescan() function should be called to confirm the PCIe PHY link is reestablished before calling the controller initialization function.<br>If virtual functions are enabled, this function will not do the reset. | |
| See Also | rescan(), initA(), init() | |

```
    c::printf("NVM Subsystem Reset:\n");
    if ((evtHandle = nvme::NVMSSReset(EC, cntlHandle)))
        c::printf("Info: NVM Subsystem Reset success\n");
    else
    {
        c::printf("Error: NVM Subsystem Reset fail; error = %d\n", EC);
        return (0);
    }
    if (rescan(EC, cntlHandle, evtHandle))
        nvme::init(EC, cntlHandle);
    else
    {
        c::printf("Error: fail to rescan the device; error = %d\n", EC);
        return (0);
    }
```

PCIeHotReset()

| | | |
|---|---|---|
| Prototype | `PCIeHotReset(&EC, cntlHdle)` | |
| Description | Do a PCIe hot reset on the specified controller. | |
| Return Value | 0 : failed<br>An event handle : successful | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| Note | After a PCIe hot reset, the controller should be initialized again for further accesses. Since a PCIe hot reset causes a reestablishment of the PCIe PHY link, the rescan() function should be called to confirm the PCIe PHY link is reestablished before calling the controller initialization function.<br>If virtual functions are enabled, this function will not do the reset. | |
| See Also | rescan(), initA(), init() | |

```
    c::printf("PCIe Hot Reset:\n");
    if ((evtHandle = nvme::PCIeHotReset(EC, cntlHandle)))
        c::printf("Info: PCIe Hot Reset success\n");
    else
    {
        c::printf("Error: PCIe Hot Reset fail; error = %d\n", EC);
        return (0);
    }
    if (rescan(EC, cntlHandle, evtHandle))
        nvme::init(EC, cntlHandle);
    else
    {
        c::printf("Error: fail to rescan the device; error = %d\n", EC);
        return (0);
    }
```

powerOffEvent()

| Prototype | `powerOffEvent(&EC, cntlHdle)` | |
|---|---|---|
| Description | Informs the extension module that a power-off event happened on the specified controller. This function de-initializes the specified controller and makes a power-off event in the event trace. | |
| Return Value | 0 : failed<br>1 : successful | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| Note | The method turning off the power supply to the device (represented as a controller handle) is done by other functions but not by this function. | |
| See Also | powerOnEvent() | |

```
scom_powerOff(hSCOM)
{
    scom::putchar(hSCOM, 0xA0);
    scom::putchar(hSCOM, 0x01);
    scom::putchar(hSCOM, 0x01);
    scom::putchar(hSCOM, 0xA2);
}

main()
{
    ...
    nvme::shutdownNotify(EC, cntlHandle, 1 /* normal shutdown */);
    scom_powerOff(hSCOM);
    nvme::powerOffEvent(EC, cntlHandle);
    sleep(3000);
    ...
}
```

powerOnEvent()

| Prototype | `powerOnEvent(&EC, cntlHdle)` | |
|---|---|---|
| Description | Informs the extension module that a power-on event happened on the specified controller. | |
| Return Value | 0 : failed<br>An event handle : successful | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| Note | The method turning on the power supply to the device (represented as a controller handle) is done by other functions but not by this function. | |
| See Also | powerOffEvent(), rescan(), init(), initA() | |

```
scom_powerOn(hSCOM)
{
    scom::putchar(hSCOM, 0xA0);
    scom::putchar(hSCOM, 0x01);
    scom::putchar(hSCOM, 0x00);
    scom::putchar(hSCOM, 0xA1);
}
main()
{
    ...
    shutdownNotify(cntlHandle);
    scom_powerOff(hSCOM);
    nvme::powerOffEvent(EC, cntlHandle);
    sleep(3000);
    scom_powerOn(hSCOM);
    evtHandle = scom_powerOn(EC, cntlHandle);
    if (nvme::rescan(EC, cntlHandle, evtHandle))
        nvme::init(EC, cntlHandle);
    ...
}
```

rescan()

| Prototype | `rescan(&EC, cntlHdle, evtHdle)` | |
|---|---|---|
| Description | Rescan the specified controller after some event which makes to reestablish the device's PHY link. | |
| Return Value | 0 : failed<br>1 : successful | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | evtHdle | The event handle. This handle records the event making the PHY link be reestablished. |
| Note | The function may spend as long as the worst case time that CSTS.RDY register denotes. | |
| See Also | NVMSSReset(), PCIeHotReset(), powerOnEvent(), init(), initA() | |

```
scom_powerOn(hSCOM)
{
    scom::putchar(hSCOM, 0xA0);
    scom::putchar(hSCOM, 0x01);
    scom::putchar(hSCOM, 0x00);
    scom::putchar(hSCOM, 0xA1);
}
main()
{
    ...
    shutdownNotify(cntlHandle);
    scom_powerOff(hSCOM);
    nvme::powerOffEvent(EC, cntlHandle);
    sleep(3000);
    scom_powerOn(hSCOM);
    evtHandle = scom_powerOn(EC, cntlHandle);
    if (nvme::rescan(EC, cntlHandle, evtHandle))
        nvme::init(EC, cntlHandle);
    ...
}
```

getDevNum()

| Prototype | `getDevNum()` |
|---|---|
| Description | Get the number of detected devices. |
| Return Value | Number of devices detected. |
| Parameter | N/A |
| Note | Say there are N devices detected, then each device is numbered from 0 to N-1.<br>This function is called after the scan() function has been called. |
| See Also | scan() |

```
main()
{
    var EC;
    var nDevs, n;
    var cntlHandle;

    if (!nvme::scan(EC))
    {
        c::printf("Error: fail to scan devices; error = %d.\n", EC);
        return (0);
    }

    nDevs = nvme::getDevNum();
    c::printf("Number of devices: %d", nDevs);
    ...
}
```

getFuncNum()

| Prototype | `getFuncNum(&EC, devNo)` | |
|---|---|---|
| Description | Get the number of active PCIe functions of a device. | |
| Return Value | -1 : failed if devNo is invalid<br>Otherwise : number of active PCI functions, including physical and virtual functions, of the specified device. | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | devNo | An integer number used to represent a device by the extension module. Valid values range from 0 to N-1 if there are N devices detected. |
| Note | | |
| See Also | getDevNum(), getCntlHandle() | |

```
main()
{
    var EC;
    var nDevs, n;
    var nFuncs;

    if (!nvme::scan(EC))
    {
        c::printf("Error: fail to scan devices; error = %d.\n", EC);
        return (0);
    }

    nDevs = nvme::getDevNum();
    c::printf("Number of devices: %d", nDevs);
    for (n = 0; n < nDevs; ++n)
    {
        nFuncs = nvme::getFuncNum(EC, n);
        ...
    }
}
```

getCntlHandle()

| | | |
|---|---|---|
| Prototype | `getCntlHandle(&EC, devNo, funcNo)` | |
| Description | Get the controller handle by device number and function number. | |
| Return Value | 0 : failed<br>Otherwise: a controller handle is returned. | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | devNo | A device number. If N devices are detected, valid device numbers are 0, 1, 2, …, N-1. |
| | funcNo | A function number. If the device has M active functions, valid function numbers are 0, 1, 2, …, M-1. |
| Note | | |
| See Also | getDevNum(), getFuncNum() | |

```
main()
{
    var EC;
    var nDevs, n;
    var nFuncs, f;
    var cntlHandle;

    if (!nvme::scan(EC))
    {
        c::printf("Error: fail to scan devices; error = %d.\n", EC);
        return (0);
    }

    nDevs = nvme::getDevNum();
    c::printf("Number of devices: %d", nDevs);
    for (n = 0; n < nDevs; ++n)
    {
        nFuncs = nvme::getFuncNum(EC, n);
        for (f = 0; f < nFuncs; ++f)
        {
            cntlHandle = nvme::getCntlHandle(EC, n, f);
            ...
        }
    }
}
```

getCntlHandleBySQHandle()

| Prototype | `getCntlHandleBySQHandle(&EC, sqHdle)` | |
|---|---|---|
| Description | Get the controller handle to which the specified sq handle belongs. | |
| Return Value | 0 : failed<br>Otherwise: a controller handle is returned. | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | sqHdle | A submission queue handle. |
| Note | | |
| See Also | getDevNum(), getFuncNum() | |

```
aTaskFunc(sqHandle)
{
    var EC;
    var cntlHandle;

    // look up the controller handle
    cntlHandle = nvme::getCntlHandleBySQHandle(EC, sqHandle);
    ...
}

const MAX_TASKS = 128;

main()
{
    var tids[MAX_TASKS];

    ...
    nIOSQNum = nvme::getIOSQNum(EC, cntlHandle);
    if (nIOSQNum > MAX_TASKS)
        nIOSQNum = MAX_TASKS;
    for (qi = 1; qi <= nIOSQNum; ++qi)
    {
        sqh = getSQHandle(EC, cntlHandle, qi);
        tids[sq – 1] = task_create(aTaskFunc, sqh);
        ...
    }
    ...
}
```

getCNTLID()

| | | |
|---|---|---|
| Prototype | `getCNTLID(&EC, cntlHdle)` | |
| Description | Get the controller's CNTLID (controller ID) reported in the Identify Controller Data Structure. | |
| Return Value | -1 : failed<br>Otherwise: the CNTLID is returned. | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | cntlHdle | A controller handle. |
| Note | | |
| See Also | getCntlHandle() | |

```
// Detach all controllers from namespaceID = nsid

bufHandle = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 4096);
nvme::ddbWriteInt16(EC, bufHandle, 0, 1, nFuncs);

for (f = 0; f < nFuncs && f < 4096/2 - 1; ++f)
{
    cntlHandle = nvme::getCntlHandle(EC, n, f);
    CNTLID = nvme::getCNTLID(EC, cntlHandle);
    nvme::ddbWrite(EC, bufHandle, (f + 1) * 2, 1, CNTLID);
}

nvme::admc::nsAtchmt(EC, cntlHandle, buf, SEL=1, nsid);
```

getDevNo()

| Prototype | `getDevNo(&EC, cntlHdle)` | |
|---|---|---|
| Description | Get the device number numbered by the extension module. The specified controller belongs to the device. | |
| Return Value | -1 : failed<br>Otherwise: the device number is returned. | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | cntlHdle | A controller handle. |
| Note | | |
| See Also | getCntlHandle() | |

```
    // Detach all controllers from namespaceID = nsid

    bufHandle = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 4096);
    nvme::ddbWriteInt16(EC, bufHandle, 0, 1, nFuncs);

    for (f = 0; f < nFuncs && f < 4096/2 - 1; ++f)
    {
        cntlHandle = nvme::getCntlHandle(EC, n, f);
        CNTLID = nvme::getCNTLID(EC, cntlHandle);
        nvme::ddbWrite(EC, bufHandle, (f + 1) * 2, 1, CNTLID);
    }

    nvme::admc::nsAtchmt(EC, cntlHandle, buf, SEL=1, nsid);
```

getPageSize()

| | | |
|---|---|---|
| Prototype | `getPageSize(&EC, cntlHdle)` | |
| Description | Get the page size set into the controller. | |
| Return Value | -1 : failed<br>Otherwise: the page size in bytes. | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | cntlHdle | A controller handle. |
| Note | The page size information is set into Controller Configuration (CC) register. | |
| See Also | getCntlHandle() | |

```
// set up Host Memory Buffer

struct nvme::HMBInfo_t HMBInfo;
struct nvme::setFeaturesOptParams_t sfOptParams;

nPageSize = nvme::getPageSize(EC, cntlHandle);
buf = nvme::ddbAlloc(EC, cntlHandle, nvme::HMB, 128 * 0x100000, HMBInfo);

sfOptParams.clean();
sfOptParams.CDW12 =  HMBInfo.nTotalSize / nPageSize;
sfOptParams.CDW13 =  HMBInfo.nDescpListPhyAddr        & 0xFFFFFFFF;
sfOptParams.CDW14 = (HMBInfo.nDescpListPhyAddr >> 32) & 0xFFFFFFFF;
sfOptParams.CDW15 =  HMBInfo.nDescpCount;

cmdHandle = nvme::admc::setFeatures(EC,
                                    cntlHandle,
                                    nFeatureId=0x0D,
                                    bSave=0,
                                    nCDW11Value=1,
                                    nilBuf,
                                    sfOptParams);
```

supportVF()

| Prototype | `supportVF(&EC, cntlHdle)` | |
|---|---|---|
| Description | Check whether or not the controller support SR-IOV virtual functions. | |
| Return Value | -1 : failed<br>0 : virtual function is not supported<br>1 : virtual  is supported | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | cntlHdle | A controller handle. |
| Note | | |
| See Also | getFuncNum(), getMaxVFNum() | |

```
// Disable all SR-IOV virtual functions

if ((r = nvme::supportVF(EC, cntlHandle)) && r != -1)
    nvme::enableVF(EC, cntlHandle, 0);
```

getMaxVFNum()

| Prototype | `getMaxVFNum(&EC, cntlHdle)` | |
|---|---|---|
| Description | Get the maximum number of virtual functions that the controller supports. | |
| Return Value | -1 : failed<br>Otherwise : the max. number of virtual functions the controller supports | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | cntlHdle | A controller handle. |
| Note | | |
| See Also | supportVF() | |

```
// Enable all SR-IOV virtual functions

if ((r = nvme::supportVF(EC, cntlHandle)) && r != -1)
{
    nMaxVFNum = getMaxVFNum(EC, cntlHandle);
    nVFNum = nvme::enableVF(EC, cntlHandle, nMaxVFNum);
    ...
}
```

enableVF()

| Prototype | `enableVF(&EC, cntlHdle, num_vfs)` | |
|---|---|---|
| Description | Enable or disable SR-IOV functions. | |
| Return Value | -1 : failed<br>Otherwise : number of virtual functions enabled | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | cntlHdle | A controller handle. |
| | num_vfs | Number of virtual functions to be enabled if num_vfs > 0.<br>Disable virtual functions if num_vfs = 0. |
| Note | | |
| See Also | supportVF() | |

```
// Enable all SR-IOV virtual functions

if ((r = nvme::supportVF(EC, cntlHandle)) && r != -1)
{
    nMaxVFNum = getMaxVFNum(EC, cntlHandle);
    nvme::enableVF(EC, cntlHandle, 0);
    nVFNum = nvme::enableVF(EC, cntlHandle, nMaxVFNum);
    ...
}
```

getNsNum()

| Prototype | `getNsNum(&EC, cntlHdle)` | |
|---|---|---|
| Description | Get the number of NVMe namespaces attached to the specified controller. | |
| Return Value | -1 : failed<br>Otherwise : number of namespaces | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | cntlHdle | A controller handle. |
| Note | | |
| See Also | getCntlHandle(), getNsIds(), getNsHandle() | |

```
    cntlHandle = nvme::getCntlHandle(EC, 0, 0);
    nvme::init(EC, cntlHandle);

    nNsNum = nvme::getNsNum(EC, cntlHandle);
    if (nNsNum <= 0) {
        c::printf("Error: no namespace exists in device 0\n");
        return;
    }
```

getNextNsId()

| Prototype | `getNextNsId(&EC, cntlHdle, baseId)` | |
|---|---|---|
| Description | Get next NVMe namespace Id which is attached to the controller and is greater than the specified baseId. | |
| Return Value | -1 : failed<br>0 : No more NVMe namespace Id<br>Otherwise : a NVMe namespace Id | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | cntlHdle | A controller handle. |
| | baseId | An NVMe namespace Id from which to get the next NVMe namespace Id. |
| Note | | |
| See Also | getNsIds(), getNsNum(), getNsHandle() | |

```
    var nsids[MAX_NAMESPACE_NUM], nNsNum, nsid;

    cntlHandle = nvme::getCntlHandle(EC, 0, 0);
    nvme::init(EC, cntlHandle);

    nNsNum = 0;
    nsid = 0;
    while (nNsNum < MAX_NAMESPACE_NUM &&
            (nsid = nvme::getNextNsId(EC, cntlHandle, nsid)) > 0)
    {
        nsids[nNsNum++] = nsid;
    }

    if (nNsNum <= 0)
    {
        c::printf("Error: no namespace exists in device 0\n");
        return;
    }
```

getNsIds()

| | | |
|---|---|---|
| Prototype | `getNsIds(&EC, cntlHdle, nArrSz, &arr[])` | |
| Description | Get Ids of namespaces attached to the specified controller. | |
| Return Value | -1 : failed<br>Otherwise : number of namespace Ids stored in `arr` array by this function. | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | cntlHdle | A controller handle. |
| | nArrSz | Size of the `arr` array. |
| | arr | An 1-dimension array for this function to store namespace Ids. |
| Note | | |
| See Also | getNextNsId(), getNsNum(), getNsHandle() | |

```
    var nsids[64], nNsNum;

    cntlHandle = nvme::getCntlHandle(EC, 0, 0);
    nvme::init(EC, cntlHandle);

    nNsNum = nvme::getNsIds(EC, cntlHandle, 64, nsids);
    if (nNsNum <= 0) {
        c::printf("Error: no namespace exists in device 0\n");
        return;
    }
```

getNsHandle()

| Prototype | `getNsHandle(&EC, cntlHdle, nsid)` | |
|---|---|---|
| Description | Get the namespace handle of the namespace with Id as nsid in the specified controller. | |
| Return Value | 0 : failed<br>Otherwise : the namespace handle | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | nsid | A namespace Id. |
| Note | | |
| See Also | getNsNum(), getNextNsId(), getNsIds() | |

```
    // Set up the 'first' namespace in the controller be the
    // default target accessed by all the commands sent into
    // any Submission Queue without explicitly specifying
    // the accessed namespace in commands.

    var nsids[4], nNsNum;
    var nsHandle;
    var sqHandle, nIOSQNum, qi;

    cntlHandle = nvme::getCntlHandle(EC, 0, 0);
    nvme::init(EC, cntlHandle);

    nNsNum = nvme::getNsIds(EC, cntlHandle, 4, nsids);
    if (nNsNum <= 0) {
        c::printf("Error: no namespace exists in device 0\n");
        return;
    }

    nsHandle = nvme::getNsHandle(EC, cntlHandle, nsids[0]);

    nIOSQNum = nvme::getIOSQNum(EC, cntlHandle);
    for (qi = 0; qi <= nIOSQNum; ++qi)
    {
        sqHandle = nvme::getSQHandle(EC, cntlHandle, qi);
        nvme::setSQAttr(EC, sqHandle, "DefaultNamespace", nsHandle);
    }
```

setNsAttr()

| | | |
|---|---|---|
| Prototype | `setNsAttr(&EC, nsHdle, attr_name, attr_value)` | |
| Description | Set attribute to the namespace. | |
| Return Value | 0 : failed<br>1 : successful | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | nsHdle | A namespace handle. |
| | attr_name | The attribute name, which is a string. Valid attributes: "AutoGenChecksumOnLBAData". |
| | attr_value | The attribute value.<br><br>For "AutoGenChecksumOnLBAData" attribute, valid values are 0 and 1.<br>0 : not to generate checksum on LBA data.<br>1 : generate checksum on LBA data.<br>If the "AutoGenChecksumOnLBAData" attribute is set as 1, a 8-byte checksum will be automatically generated and be put at the first 8 bytes for each LBA data. |
| Note | | |
| See Also | getNsHandle(), setSQAttr() | |

```
nvme::setSQAttr(EC, sq_handle, "DefaultNamespace", ns_handle);
nvme::setNsAttr(EC, ns_handle, "AutoGenChecksumOnLBAData", 1);

cmdHandle = nvme::ioc::write(EC, sq_handle, dbuf1, 0, 1);
...
cmdHandle = nvme::ioc::read(EC, sq_handle, dbuf2, 0, 1);
...

// calculate the checksum based on the read data
cksum_cal = nvme::ddbCalculateChecksum64(EC, buf2, 8, 512 - 8);

// get the written checksum
cksum_read = nvme::ddbReadInt64(EC, buf2, 0, 1);

// compare checksum
if (cksum_cal != cksum_read)
    c::printf("Error: IO fail\n");
```

getMaxIOSQNum()

| Prototype | `getMaxIOSQNum(&EC, cntlHdle)` | |
|---|---|---|
| Description | Get the maximum number of IO SQs supported by the controller. | |
| Return Value | -1 : failed<br>Otherwise : the maximum number of IO SQs supported | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | cntlHdle | A controller handle. |
| Note | The return value denotes the capability that the maximum number of IO SQs supported by the device but not the current number of IO SQs created by the device. | |
| See Also | getIOSQNum(), getMaxIOCQNum() | |

```
main()
{
    var EC, n, nDevs;
    var cntlHandle, nIOSQNum;
    struct nvme::queueInitParams_t initQParams;

    if (!nvme::scan(EC))
    {
        c::printf("Error: fail to scan devices; error = %d.\n", EC);
        return (0);
    }
    nDevs = nvme::getDevNum();
    c::printf("Number of devices: %d", nDevs);

    for (n = 0; n < nDevs; ++n)
    {
        cntlHandle = nvme::getCntlHandle(EC, n, 0);
        nIOSQNum = nvme::getMaxIOSQNum(EC, cntlHandle);
        if (nIOSQNum > 16)
            nIOSQNum = 16;

        initQParams.setDef();
        initQParams.nASQSize = 32;
        initQParams.nACQSize = 32;
        initQParams.nIOSQNum = nIOSQNum;
        initQParams.nIOSQSize = 64;
        initQParams.nIOSQNumPerIOCQ = 8;

        nvme::init(EC, cntlHandle, "", initQParams);

        if (initQParams.nIOSQNum == nIOSQNum)
            c::printf("OK");
        else
            c::printf("NG");
    }
```

getMaxIOCQNum()

| Prototype | `getMaxIOCQNum(&EC, cntlHdle)` | |
|---|---|---|
| Description | Get the maximum number of IO CQs supported by the controller. | |
| Return Value | -1 : failed<br>Otherwise : the maximum number of IO CQs supported | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | cntlHdle | A controller handle. |
| Note | The return value denotes the capability that the maximum number of IO CQs supported by the device but not the current number of IO CQs created by the device. | |
| See Also | getIOCQNum(), getMaxIOSQNum() | |

```
main()
{
    var EC, n, nDevs;
    var cntlHandle, nIOSQNum, nIOCQNum;
    struct nvme::queueInitParams_t initQParams;

    if (!nvme::scan(EC))
    {
        c::printf("Error: fail to scan devices; error = %d.\n", EC);
        return (0);
    }
    nDevs = nvme::getDevNum();
    c::printf("Number of devices: %d", nDevs);

    for (n = 0; n < nDevs; ++n)
    {
        cntlHandle = nvme::getCntlHandle(EC, n, 0);

        nIOSQNum = nvme::getMaxIOSQNum(EC, cntlHandle);
        if (nIOSQNum > 16)
            nIOSQNum = 16;

        nIOCQNum = nvme::getMaxIOCQNum(EC, cntlHandle);
        if (nIOCQNum > nIOSQNum)
            nIOCQNum = nIOSQNum;

        initQParams.setDef();
        initQParams.nASQSize = 32;
        initQParams.nACQSize = 32;
        initQParams.nIOSQNum = nIOSQNum;
        initQParams.nIOSQSize = 64;
        initQParams.nIOSQNumPerIOCQ = nIOSQNum / nIOCQNum +
                                      (nIOSQNum % nIOCQNum) ? 1 : 0;

        nvme::init(EC, cntlHandle, "", initQParams);
        ...
    }
```

getMaxIOQSize()

| Prototype | `getMaxIOQSize(&EC, cntlHdle)` | |
|---|---|---|
| Description | Get the maximum size of an IO queue, either an IO SQ or an IO CQ, of the specified controller. | |
| Return Value | -1 : failed<br>Otherwise : the maximum size of an IO queue | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | cntlHdle | A controller handle. |
| Note | The return value denotes the capability that can be supported by the controller but not the maximum size of IO queues created currently. | |
| See Also | | |

```
main()
{
}
```

getIOSQNum()

| | | |
|---|---|---|
| Prototype | `getIOSQNum(&EC, cntlHdle)` | |
| Description | Get the number of IO SQs created on the specified controller. | |
| Return Value | -1 : failed<br>Otherwise : the number of IO SQs created on the specified controller | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | cntlHdle | A controller handle. |
| Note | | |
| See Also | | |

```
struct nvme::queueInitParams_t initQParams;

initQParams.setDef();
initQParams.nIOSQNum = 1024;
initQParams.nIOSQNumPerIOCQ = 4;

// get real numbers of IOSQs/IOCQs created by init() function
if (nvme::init(EC, cntlHandle, "", initQParams))
{
    nIOSQNum = nvme::getIOSQNum(EC, cntlHandle);
    nIOCQNum = nvme::getIOCQNum(EC, cntlHandle);
    c::printf("Number of IOSQs = %d\n", nIOSQNum);
    c::printf("Number of IOCQs = %d\n", nIOCQNum);
}
```

getIOCQNum()

| Prototype | `getIOCQNum(&EC, cntlHdle)` | |
|---|---|---|
| Description | Get the number of IO CQs created on the specified controller. | |
| Return Value | -1 : failed<br>Otherwise : the number of IO CQs created on the specified controller | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | cntlHdle | A controller handle. |
| Note | | |
| See Also | | |

```
// get the default number of IOSQs/IOCQs created by init() function
if (nvme::init(EC, cntlHandle))
{
    nIOSQNum = nvme::getIOSQNum(EC, cntlHandle);
    nIOCQNum = nvme::getIOCQNum(EC, cntlHandle);
    c::printf("Number of IOSQs = %d\n", nIOSQNum);
    c::printf("Number of IOCQs = %d\n", nIOCQNum);
}
```

getNextIOSQId()

| | | |
|---|---|---|
| Prototype | `getNextIOSQId(&EC, cntlHdle, baseId)` | |
| Description | Get next IOSQ Id which belongs to the controller and is greater than the specified baseId. | |
| Return Value | -1 : failed<br>0 : No more IOSQ Id<br>Otherwise : an IOSQ Id | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | cntlHdle | A controller handle. |
| | baseId | An IOSQ Id from which to get the next IOSQ Id. |
| Note | | |
| See Also | | |

```
// list all IOSQ Ids
sqid = 0;
while (1)
{
    sqid = getNextIOSQId(EC, cntlHandle, sqid);
    if (sqid <= 0)
        break;

    c::printf(" %d", sqid);
}
c::printf("\n");
```

getNextIOCQId()

| | | |
|---|---|---|
| Prototype | `getNextIOCQId(&EC, cntlHdle, baseId)` | |
| Description | Get next IOCQ Id which belongs to the controller and is greater than the specified baseId. | |
| Return Value | -1 : failed<br>0 : No more IOCQ Id<br>Otherwise : an IOCQ Id | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | cntlHdle | A controller handle. |
| | baseId | An IOCQ Id from which to get the next IOCQ Id. |
| Note | | |
| See Also | | |

```
// list all IOCQ Ids and corresponding IOSQs
cqid = 0;
while (1)
{
    cqid = nvme::getNextIOCQId(EC, cntlHandle, cqid);
    if (cqid <= 0)
        break;

    c::printf("CQ = %d, SQ =", cqid);
    sqid = 0;
    while (1)
    {
        sqid = nvme::getNextPairedIOSQId(EC, cntlHandle, cqid, sqid);
        if (sqid <= 0)
            break;
        c::printf(" %d", sqid);
    }
    c::printf("\n");
}
```

getPairedIOCQId()

| | | |
|---|---|---|
| Prototype | `getPairedIOCQId(&EC, cntlHdle, iosqid)` | |
| Description | Get the paired IO CQ Id of the specified IO SQ Id. | |
| Return Value | 0 : failed<br>Otherwise : an IOCQ Id | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | iosqid | An IOSQ Id |
| Note | | |
| See Also | | |

```
// list the paired IOCQ of each IOSQ created by init() function
if (nvme::init(EC, cntlHandle))
{
    sqid = 0;
    while ((sqid = nvme::getNextIOSQId(EC, cntlHandle, sqid)) > 0)
    {
        cqid = nvme::getPairedIOCQId(EC, cntlHandle, sqid);
        c::printf("SQId:%04X CQId:%04X\n", sqid, cqid);
    }
}
```

getNextPairedIOSQId()

| Prototype | `getNextPairedIOSQId(&EC, cntlHdle, iocqid, baseid)` | |
|---|---|---|
| Description | Get next paired IOSQ Id of an IOCQ. | |
| Return Value | -1 : failed<br>0 : no more paired IOSQ Id<br>Otherwise : an IO SQ Id | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | cntlHdle | A controller handle. |
| | iocqid | An IOCQ Id to which the next IOSQ Id paired. |
| | baseid | An IOSQ Id from which to get the next IOSQ Id. |
| Note | | |
| See Also | | |

```
// delete iocq's paired IOSQs
sqid = 0;
while (1)
{
    sqid = nvme::getNextPairedIOSQId(EC, cntlHandle, cqid, sqid);
    if (sqid == 0)
        break;

    cmd = nvme::admc::deleteIOSQ(EC, cntlHandle, sqid);
    ...
}
```

getSQHandle()

| | | |
|---|---|---|
| Prototype | `getSQHandle(&EC, cntlHdle, sqid)` | |
| Description | Get the sq handle of the specified SQ Id. | |
| Return Value | 0 : failed<br>Otherwise : an SQ handle | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | sqid | An SQ Id. Say there are N IO SQs. The valid SQ Ids of a controller range from 0 to N, where SQ Id 0 is the Admin SQ Id. |
| Note | | |
| See Also | | |

```
// get the handle of the Admin SQ
asqHandle = nvme::getSQHandle(EC, cntlHandle, 0);

// get the handle of the first IO SQ, and issue a flush command
sqid = nvme::getNextIOSQId(EC, cntlHandle, 0);
sqh = nvme::getSQHandle(EC, cntlHandle, sqid);
cmdHandle = nvme::ioc::flush(EC, sqh, 0xFFFFFFFF);
```

getSQId()

| Prototype | `getSQId(&EC, sqcmdHdle)` | |
|---|---|---|
| Description | Get the SQ Id of the specified SQ handle. | |
| Return Value | -1 : failed<br>Otherwise : an SQ Id | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | sqcmdHdle | An SQ handle or a command handle. |
| Note | | |
| See Also | | |

```
stsVal = nvme::pspp(cmdHandle);
if (stsVal && stsVal != nvme::cmdCplFlag)
{
    qid = nvme::getSQId(EC, cmdHandle);
    cid = nvme::getCId(cmdHandle);

    c::printf("QID=%04X CID=%04X : command fail\n", qid, cid);
}
```

getSQSize()

| Prototype | `getSQSize(&EC, sqHdle)` | |
|---|---|---|
| Description | Get the SQ size. | |
| Return Value | -1 : failed<br>Otherwise : size of the SQ | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | sqHdle | An SQ handle. |
| Note | | |
| See Also | | |

```
// no metadata; LBA data size = 512

sqSize = nvme::getSQSize(EC, sqHandle);
for (idx = 0; idx < sqSize - 1; ++idx)
    bufs[idx] = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 4096);
...

if (nvme::isSQEmpty(EC, sqh))
{
    // launch mass reads
    for (idx = 0; idx < sqSize - 1; ++idx)
    {
        lba = c::rand() % (nCapacity - 7);
        do
            cmds[idx] = nvme::ioc::read(EC, sqh, bufs[idx], lba, 8);
        while (cmds[idx] == 0);
    }
}
```

setSQAttr()

| | | |
|---|---|---|
| Prototype | `setSQAttr(&EC, sqHdle, attr_name, attr_value)` | |
| Description | Set SQ attribute. | |
| Return Value | 0 : failed<br>1 : successful | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | sqHdle | An SQ handle. |
| | attr_name | The attribute name. The attribute name is a string. Valid attribute names: “PollingMode” and “DefaultNamespace” |
| | attr_value | The attribute value.<br>If attr_name is “PollingMode”, only 0 means false and all the other values mean true.<br>If attr_name is “DefaultNamespace”, the valid value is a namespace handle. |
| Note | If the “PollingMode” attribute is set as false, value 1 will be returned from the function which successfully pass a command into this SQ and the script does not need to poll the completions of the issued commands. This behavior is used for generating a high-throughput workload without caring data content.<br><br>If the “DefaultNamespace” attribute is set with a namespace handle, the namespace would be the default accessed namespace. Any field of a command entry related to size of LBA, number of total LBAs, and nsid, etc. will refer to the default namespace. But note that any function call still can overwrite the default namespace by explicitly specifying a namespace Id as its argument. | |
| See Also | write(), ddbCalculateChecksum64() | |

```
// no metadata; LBA data size = 512

bufs[0] = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 4096);
bufs[1] = nvme::ddbAlloc(EC, cntlHandle, nvme::SGL, 4096);

sqid = nvme::getNextIOSQId(EC, cntlHandle, 0);
sqHandle = nvme::getSQHandle(EC, cntlHandle, sqid);
nvme::setSQAttr(EC, sqHandle, "PollingMode", 0);  // set polling mode off

// the script can iteratively issue commands to the device without
// caring the completions of issued commands
nIssueNum1 = nvme::getSQCmdIssueNum(EC, sqHandle);
stime = c::time();
counter = 0;
while (c::time() - stime < 60)
{
    lba = c::rand() % (nCapacity - 7);
    if (nvme::ioc::read(EC, sqHandle, bufs[c::rand()%2], lba, 8)
        ++counter;
}
```

```
nIssueNum2 = nvme::getSQCmdIssueNum(EC, sqHandle);
if (nIssueNum2 – nIssueNum1 == counter)
    c::printf(“OK”);
else
    c::printf(“NG”);
```

isSQFull()

| Prototype | `isSQFull(&EC, sqHdle)` | |
|---|---|---|
| Description | Query whether the specified SQ is full or not. | |
| Return Value | -1 : failed<br>0 : not full<br>1 : full | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | sqHdle | An SQ handle. |
| Note | | |
| See Also | | |

```
// no metadata; LBA data size = 512

buf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 4096);

if (!nvme::isSQFull(EC, sqh))
{
    // launch 1 read command
    lba = c::rand() % (nCapacity - 7);
    do
        cmd = nvme::ioc::read(EC, sqh, buf, lba, 8);
    while (cmd == 0);
}
```

isSQEmpty()

| | | |
|---|---|---|
| Prototype | `isSQEmpty(&EC, sqHdle)` | |
| Description | Query whether the specified SQ is empty or not. | |
| Return Value | -1 : failed<br>0 : not empty<br>1 : empty | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | sqHdle | An SQ handle. |
| Note | | |
| See Also | | |

```
// no metadata; LBA data size = 512

sqSize = nvme::getSQSize(EC, sqHandle);
for (idx = 0; idx < sqSize - 1; ++idx)
    bufs[idx] = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 4096);
...

if (nvme::isSQEmpty(EC, sqh))
{
    // launch mass reads
    for (idx = 0; idx < sqSize - 1; ++idx)
    {
        lba = c::rand() % (nCapacity - 7);
        do
            cmds[idx] = nvme::ioc::read(EC, sqh, bufs[idx], lba, 8);
        while (cmds[idx] == 0);
    }
}
```

readPCIConfig8()
readPCIConfig16()
readPCIConfig32()

| | | |
|---|---|---|
| Prototype | `readPCIConfig8(&EC, cntlHdle, address)`<br>`readPCIConfig16(&EC, cntlHdle, address)`<br>`readPCIConfig32(&EC, cntlHdle, address)` | |
| Description | Read the configuration space value. | |
| Return Value | 0 : failed if EC is not 0<br>0 : the register value if EC is 0<br>Otherwise : the register value | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code **MAY** be stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | address | The register address. |
| Note | | |
| See Also | | |

```
ID = readPCIConfig32(EC, cntlHandle, 0);
c::printf("VID = %04X\n", ID & 0xFFFF);
c::printf("DID = %04X\n", (ID >> 16) & 0xFFFF);
```

readCntlReg32()

| | | |
|---|---|---|
| Prototype | `readCntlReg32(&EC, cntlHdle, address)` | |
| Description | Read the controller register value. | |
| Return Value | 0 : failed if EC is not 0<br>0 : successful if EC is 0<br>Otherwise : the register value | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code **MAY** be stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | address | The register address. |
| Note | | |
| See Also | | |

```
VS = readCntlReg32(EC, cntlHandle, 8);
c::printf("Major Version Number: %d\n", (VS >> 16) & 0xFFFF);
c::printf("Minor Version Number: %d\n", (VS >> 8) & 0xFF);
```

shutdownNotify()

| | | |
|---|---|---|
| Prototype | `shutdownNotify(&EC, cntlHdle, value)`<br>`shutdownNotify(&EC, cntlHdle, value, waittime)` | |
| Description | Initiate a shutdown notification to the controller by read-modify-write the Controller Configuration register. | |
| Return Value | 0 : failed<br>1 : successful | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | value | A 2-bit notification value to write into the Controller Configuration register.<br>1 : Normal shutdown notification<br>2 : Abrupt shutdown notification |
| | waittime (optional) | A maximum time length to wait for the shutdown process done by the device. If there is no argument passed to this parameter, this function apply the CMDTO value to wait for the shutdown process done. This value is in seconds. Maximum value is 255. |
| Note | | |
| See Also | | |

```
    nvme::shutdownNotify(EC, cntlHandle, 1 /* normal shutdown */);
    scom_powerOff(hSCOM);
    nvme::powerOffEvent(EC, cntlHandle);
```

ddbAlloc()

<table>
<tr><td>Prototype</td><td colspan="2">ddbAlloc(&EC, cntlHdle, type, size)<br>ddbAlloc(&EC, cntlHdle, type, size, struct nvme::HMBInfo_t &hmbInfo)</td></tr>
<tr><td>Description</td><td colspan="2">Allocate a DMA data buffer.</td></tr>
<tr><td>Return Value</td><td colspan="2">0 : failed<br>Otherwise : a DMA data buffer (DDB) handle</td></tr>
<tr><td rowspan="5">Parameter</td><td>EC</td><td>This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0.</td></tr>
<tr><td>cntlHdle</td><td>A controller handle.</td></tr>
<tr><td>type</td><td>Buffer type; valid values are nvme::PRP, nvme::SGL, and nvme::HMB</td></tr>
<tr><td>size</td><td>Buffer size in bytes requested.</td></tr>
<tr><td>hmbInfo (optional)</td><td>This parameter is a call-by-reference one. The argument should be a nvme::HMBInfo_t structure. This structure is defined by the extension module as the following:<br><pre>struct HMBInfo_t
{
  var nPageSize; /*in bytes*/
  var nTotalSize; /*in bytes*/
  var nDescpListPhyAddr;
  var nDescpCount;
  clean()
  {
    nPageSize = 0;
    nTotalSize = 0;
    nDescpListPhyAddr = 0;
    nDescpCount = 0;
  }
};</pre>This function updates this parameter after a successful HMB buffer allocation.</td></tr>
<tr><td>Note</td><td colspan="2">The hmbInfo parameter can be specified only if the type value is nvme::HMB.</td></tr>
<tr><td>See Also</td><td colspan="2"></td></tr>
</table>

```
buf1 = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 512);
buf2 = nvme::ddbAlloc(EC, cntlHandle, nvme::SGL, 512);
cmd1 = nvme::ioc::read(EC, sqhandle, buf1, 0, 1);
cmd2 = nvme::ioc::read(EC, sqhandle, buf2, 1, 1);
```

ddbFree()

| | | |
|---|---|---|
| Prototype | `ddbFree(&EC, bufHdle)` | |
| Description | To free an allocated DDB buffer. | |
| Return Value | 0 : failed<br>1 : successful | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | bufHdle | A buffer handle. |
| Note | | |
| See Also | | |

```
buf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 512);
cmdHandle = nvme::ioc::read(EC, sqhandle, buf, 0, 1);
...
nvme::ddbFree(EC, buf);

// buf now cannot be used; error code is returned if it is used any more
cmdHandle = nvme::ioc::read(EC, sqhandle, buf, 0, 1);
if (cmdHandle || EC == 0)
    c::printf("Incorrect!\n");
```

ddbSize()

| | | |
|---|---|---|
| Prototype | `ddbSize(&EC, bufHdle)` | |
| Description | Get the allocated buffer size in bytes. | |
| Return Value | 0 : failed<br>Otherwise : the buffer size | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | bufHdle | A buffer handle. |
| Note | | |
| See Also | | |

```
buf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 129);
c::printf("buffer size = %d\n", nvme::ddbSize(EC, buf));
```

ddbCalculateChecksum64()

| | | |
|---|---|---|
| Prototype | `ddbCalculateChecksum64(&EC, bufHdle, offset, size)` | |
| Description | Calculate a 64-bit checksum. | |
| Return Value | 0 : failed if EC is not 0<br>0 : the 64-bit checksum value if EC is 0<br>Otherwise : the 64-bit checksum value | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code **MAY** be stored on this variable if the return value is 0. |
| | bufHdle | A buffer handle. |
| | offset | The location from which the checksum will be calculated. |
| | size | The size in bytes for calculating the 64-bit checksum. |
| Note | Note that both offset and size should be a multiple of 8.<br>This function is used to generate a 64-bit checksum by doing XOR operations on the specified data.<br>The same approach is used by the nvme::ioc::write() function to generate checksum values for all LBAs. The nvme::ioc::write() function automatically generates a 64-bit checksum value by the LBA data from byte offset 8 to the end of LBA data if the "AutoGenChecksumOnLBAData" attribute of the namespace is set as 1. The 64-bit checksum value will be put at the beginning of each LBA data. The LBA data used to calculate the checksum includes metadata if the metadata exists and the metadata is transferred immediately after the LBA data. | |
| See Also | | |

```
buf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 512);
cmdHandle = nvme::ioc::read(EC, sqhandle, buf, 0, 1);
...
cksum64_0 = nvme::ddbCalculateChecksum64(EC, buf, 8, 512 – 8);
cksum64_1 = nvme::ddbReadInt64(EC, buf, 0, 1);
if (cksum64_0 != cksum64_1)
    c::printf("miscompare\n");
```

ddbReadASCIIString()

| | | |
|---|---|---|
| Prototype | `ddbReadASCIIString(&EC, bufHdle, &str, offset, max_len)` | |
| Description | Read a zero-terminated ASCII string, up to max_len characters, from a DDB. | |
| Return Value | -1 : failed<br>Otherwise : the string length in bytes | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | bufHdle | A buffer handle. |
| | str | This parameter is a call-by-reference one. The argument should be an elementary variable. The read string is stored in this variable. |
| | offset | The location from which to read the zero-terminated string. |
| | max_len | The maximum length to read the string. |
| Note | This function stops reading the string content if a value-0 character is encountered or the read length is max_len. | |
| See Also | | |

```
var EC, buf, cmdHandle, cns, ns, CNTLID;
var stsVal, SN, MN, FR;

buf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 4096);

// identify controller
cmdHandle = nvme::ioc::identify(EC, cntlHandle, buf, cns=1, ns=0, CNTLID=0);
do
    stsVal = nvme::pspp(cmdHandle);
while (stsVal == 0);

if (stsVal != nvme::cmdCplFlag)
    goto errorHandling;

nvme::ddbReadASCIIString(EC, buf, SN,  4, 20);
nvme::ddbReadASCIIString(EC, buf, MN, 24, 40);
nvme::ddbReadASCIIString(EC, buf, FR, 64, 8);
```

ddbWriteASCIIString()

| | | |
|---|---|---|
| Prototype | `ddbWriteASCIIString(&EC, bufHdle, offset, str)` | |
| Description | Write a string into a DDB. | |
| Return Value | -1 : failed<br>Otherwise : the written string length | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | bufHdle | A buffer handle. |
| | offset | The location from which to write the string. |
| | str | An ASCII string. |
| Note | If the string length is N, only N characters are written into the DDB. No value-0 character is written. To form a zero-terminated string in the DDB, the ddbWriteInt8() function can be called to put a value-0 character after the string. | |
| See Also | | |

```
buf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 4096);
...

offset += nvme::ddbWriteASCIIString(EC, buf, offset, "MaxMethods");
```

ddbReadInt8()
ddbReadUInt8()

| | | |
|---|---|---|
| Prototype | `ddbReadInt8(&EC, bufHdle, offset)`<br>`ddbReadUInt8(&EC, bufHdle, offset)` | |
| Description | Read an 8-bit signed/unsigned integer from DDB. | |
| Return Value | 0 : failed if EC is not 0<br>0 : the read value if EC is 0<br>Otherwise : the read value | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code **MAY** be stored on this variable if the return value is 0. |
| | bufHdle | A buffer handle. |
| | offset | The location from which to read the value. |
| Note | | |
| See Also | | |

```
var EC, buf, cmdHandle, cns, ns, CNTLID;
var stsVal, nMDTS;

buf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 4096);

// identify controller
cmdHandle = nvme::ioc::identify(EC, cntlHandle, buf, cns=1, ns=0, CNTLID=0);
do
    stsVal = nvme::pspp(cmdHandle);
while (stsVal == 0);

if (stsVal != nvme::cmdCplFlag)
    goto errorHandling;

nMDTS = nvme::ddbReadUInt8(EC, buf, 77);
```

ddbWriteInt8()
ddbWriteUInt8()

| | | |
|---|---|---|
| Prototype | `ddbWriteInt8(&EC, bufHdle, offset, value)`<br>`ddbWriteUInt8(&EC, bufHdle, offset, value)` | |
| Description | Write an 8-bit signed/unsigned integer to a DDB. | |
| Return Value | 0 : failed<br>1 : successful | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | bufHdle | A buffer handle. |
| | offset | The location from which to write the value. |
| | value | The value to be written. |
| Note | Only the least 8-bit value is written. | |
| See Also | | |

```
buf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 512);

// write 0x55 at offset 0x13 of the buffer
nvme::ddbWriteUInt8(EC, buf, 0x13, 0x55);
```

ddbReadInt16()
ddbReadInt32()
ddbReadInt64()
ddbReadUInt16()
ddbReadUInt32()
ddbReadUInt48()

| | | |
|---|---|---|
| Prototype | `ddbReadInt16(&EC, bufHdle, offset, bLE)`<br>`ddbReadInt32(&EC, bufHdle, offset, bLE)`<br>`ddbReadInt64(&EC, bufHdle, offset, bLE)`<br>`ddbReadUInt16(&EC, bufHdle, offset, bLE)`<br>`ddbReadUInt32(&EC, bufHdle, offset, bLE)`<br>`ddbReadUInt48(&EC, bufHdle, offset, bLE)` | |
| Description | Read a 16/32/64-bit signed/unsigned integer from a DDB. | |
| Return Value | 0 : failed if EC is not 0<br>0 : the read value if EC is 0<br>Otherwise : the read value | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code **MAY** be stored on this variable if the return value is 0. |
| | bufHdle | A buffer handle. |
| | offset | The location where to read the value. |
| | bLE | Whether or not the integer format is the little endian.<br>1 : little endian<br>0 : big endian |
| Note | | |
| See Also | | |

```
var EC, buf, cmdHandle, cns, ns, CNTLID;
var stsVal, nCapacity, LBAF0;

buf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 4096);

// identify namespace
cmdHandle = nvme::ioc::identify(EC, cntlHandle, buf, cns=0, ns=1, CNTLID=0);
do
    stsVal = nvme::pspp(cmdHandle);
while (stsVal == 0);

if (stsVal != nvme::cmdCplFlag)
    goto errorHandling;

// read values from the identify namespace data structure
nCapacity = nvme::ddbReadInt64(EC, buf, 8, 1);
LBAF0 = nvme::ddbReadUInt32(EC, buf, 128, 1);
```

ddbWriteInt16()
ddbWriteInt32()
ddbWriteInt64()
ddbWriteUInt16()
ddbWriteUInt32()

| | | |
|---|---|---|
| Prototype | `ddbWriteInt16 (&EC, bufHdle, offset, bLE, value)`<br>`ddbWriteInt32 (&EC, bufHdle, offset, bLE, value)`<br>`ddbWriteInt64 (&EC, bufHdle, offset, bLE, value)`<br>`ddbWriteUInt16(&EC, bufHdle, offset, bLE, value)`<br>`ddbWriteUInt32(&EC, bufHdle, offset, bLE, value)` | |
| Description | Write a 16/32/64-bit signed/unsigned integer to a DDB. | |
| Return Value | 0 : failed<br>1 : successful | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | bufHdle | A buffer handle. |
| | offset | The location where to read the value. |
| | bLE | Whether or not the integer format is the little endian.<br>1 : little endian<br>0 : big endian |
| | value | The value to be written. |
| Note | Only the least N-bit value is written into the DDB, where N is 16, 32, or 64. | |
| See Also | | |

```
buf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 512);
nvme::ddbFillPat8(EC, buf, 0);

// write values into the buffer
nvme::ddbWriteInt16(EC, buf, 0, 1, 0x1234);
nvme::ddbWriteInt16(EC, buf, 2, 0, 0x1234);
nvme::ddbWriteInt32(EC, buf, 9, 0, 0x12345678);

nvme::ddbDump(EC, buf, 0, 0, 512);
```

ddbFillPat8()

| | | |
|---|---|---|
| Prototype | `ddbFillPat8(&EC, bufHdle, value)`<br>`ddbFillPat8(&EC, bufHdle, value, offset)`<br>`ddbFillPat8(&EC, bufHdle, value, offset, len)` | |
| Description | Fill 8-bit values into a segment of the DDB. | |
| Return Value | 0 : failed<br>Otherwise : the length of data filled into the DDB. | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | bufHdle | A buffer handle. |
| | value | The filled value. |
| | offset (optional) | The beginning location from which the data value would be filled. |
| | len (optional) | The total length, in bytes, of the DDB to be filled. |
| Note | If len is not specified, the data value is filled into the buffer from the offset to the end of the buffer.<br>If neither offset nor len is specified, the data value is filled into the full buffer. | |
| See Also | | |

```
buf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 512);
nvme::ddbFillPat8(EC, buf, 0xFF);

// fill random values into byte 128 ~ 383 of buf
nvme::ddbFillPatRand(EC, buf, 128, 256);

nvme::ddbDump(EC, buf, 0, 0, 512);
```

ddbFillPat32()

| | | |
|---|---|---|
| Prototype | `ddbFillPat32(&EC, bufHdle, value, bLE)`<br>`ddbFillPat32(&EC, bufHdle, value, bLE, offset)`<br>`ddbFillPat32(&EC, bufHdle, value, bLE, offset, len)` | |
| Description | Fill 32-bit values into a segment of the DDB. | |
| Return Value | 0 : failed<br>Otherwise : the length of data filled into the DDB. | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | bufHdle | A buffer handle. |
| | value | The filled value. |
| | bLE | Whether or not the integer format is the little endian.<br>1 : little endian<br>0 : big endian |
| | offset (optional) | The beginning location from which the data value would be filled. |
| | len (optional) | The total length, in bytes, of the DDB to be filled. |
| Note | If len is not specified, the data value is filled into the buffer from the offset to the end of the buffer.<br>If neither offset nor len is specified, the data value is filled into the full buffer. | |
| See Also | | |

```
buf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 512);
nvme::ddbFillPat32(EC, buf, 0, 1);

// fill random values into byte 128 ~ 383 of buf
nvme::ddbFillPatRand(EC, buf, 128, 256);

nvme::ddbDump(EC, buf, 0, 0, 512);
```

ddbFillPatRand()

| | | |
|---|---|---|
| Prototype | `ddbFillPatRand(&EC, bufHdle)`<br>`ddbFillPatRand(&EC, bufHdle, offset)`<br>`ddbFillPatRand(&EC, bufHdle, offset, len)` | |
| Description | Fill random values into a segment of the DDB. | |
| Return Value | 0 : failed<br>Otherwise : the length of data filled into the DDB. | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | bufHdle | A buffer handle. |
| | offset (optional) | The beginning location from which the data value would be filled. |
| | len (optional) | The total length, in bytes, of the DDB to be filled. |
| Note | If len is not specified, the data value is filled into the buffer from the offset to the end of the buffer.<br>If neither offset nor len is specified, the data value is filled into the full buffer. | |
| See Also | | |

```
buf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 512);
nvme::ddbFillPat8(EC, buf, 0);

// fill random values into byte 128 ~ 383 of buf
nvme::ddbFillPatRand(EC, buf, 128, 256);

nvme::ddbDump(EC, buf, 0, 0, 512);
```

ddbCopyFromMemAddr()

| Prototype | ddbCopyFromMemAddr(&EC, bufHdle, offset, maddr, len) | |
|---|---|---|
| Description | Copy data from memory address to the DDB. | |
| Return Value | 0 : failed<br>Otherwise : the length of data copied | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | bufHdle | A buffer handle. |
| | offset | The beginning location of the DDB into which data will be copied. |
| | maddr | The memory address. |
| | len | Total length of data, in bytes, to be copied. |
| Note | | |
| See Also | | |

```
main(argc, argv[])
{
    ...

    fw_filesize = c::util::getFileSize(argv[1]);
    nTxSize = 32768;

    ddbBuf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, nTxSize);
    memHandle = c::malloc(nTxSize);
    nSrcMemAddr = c::util::getMemAddr(memHandle);

    c::fseek(fwImgFileHandle, 0, c::SEEK_SET);
    for (nAccReadSize = 0;
         nAccReadSize < fw_filesize;
         nAccReadSize += nThisReadSize)
    {
        nvme::ddbFillPat8(EC, ddbBuf, 0);
        if (fw_filesize - nAccReadSize > nTxSize)
            nThisReadSize = nTxSize;
        else
            nThisReadSize = fw_filesize - nAccReadSize;

        c::fread(memHandle, 1, nThisReadSize, fwImgFileHandle);
        nvme::ddbCopyFromMemAddr(EC,
                                 ddbBuf,
                                 0,
                                 nSrcMemAddr,
                                 nThisReadSize);

        r = fw_download(logfd,
                        cntlHandle,
                        ddbBuf,
                        nTxSize/4,
                        nAccReadSize/4);
        if (!r)
        {
```

```
            nvme::ddbFree(EC, ddbBuf);
            goto errorHandle;
        }
    }
...
}
// return value:
//  1 : success
//  0 : fail
fw_download(logfd, cntlHandle, bufHandle, DWordsNum, offset)
{
    var EC;
    var cmdHandle = 0;
    var tries = 0;
    var stsVal;

    // send the command
    while (cmdHandle == 0 && tries < 5)
    {
        if (tries)
            sleep(50);
        cmdHandle = nvme::admc::fwImgDwnld(EC,
                                           cntlHandle,
                                           bufHandle,
                                           DWordsNum,
                                           offset);
        ++tries;
    }

    if (cmdHandle == 0)
        return (0);

    // poll the command status
    while (1)
    {
        stsVal = nvme::pspp(cmdHandle);
        if (stsVal == nvme::cmdCplFlag)
            break;
        else if (stsVal)
        {
            c::util::logPrintf(logfd, "Error: fwImgDwnld command fail\n");
            showStatusErrors(logfd, stsVal);
            return (0);
        }
        sleep(2);
    }

    return (1);
}
```

ddbDump()

| | | |
|---|---|---|
| Prototype | `ddbDump(&EC, bufHdle, fileHdle, offset)`<br>`ddbDump(&EC, bufHdle, fileHdle, offset, len)` | |
| Description | Dump DDB contents in the console and optionally save the contents into a file. | |
| Return Value | 0 : failed<br>Otherwise : the length of data dumped | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | bufHdle | A buffer handle. |
| | fileHdle | A file handle for saving the DDB contents.<br>If 0 is given, contents will be displayed in the console only. |
| | offset | The beginning location from which data will be dumped. |
| | len (optional) | Total length of data, in bytes, to be dumped. |
| Note | If len is not specified, the data value is dumped from the offset to the end of the buffer. | |
| See Also | | |

```
buf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 512);
readCmd = nvme::ioc::read (EC, sq1, buf2, 0, 1);
do
    stsVal = nvme::pspp(readCmd);
while (stsVal == 0);
nvme::ddbDump(EC, buf, 0, 0);
```

ddbCompareN()

| | | |
|---|---|---|
| Prototype | `ddbCompareN(&EC, bufHdle1, bufHdle2, len)` | |
| Description | Compare two DDB buffers. | |
| Return Value | -1 : failed<br>0 : two DDBs have the same content in the first len bytes.<br>1 : two DDBs have different content in the first len bytes. | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | bufHdle1 | A buffer handle. |
| | bufHdle2 | A buffer handle. |
| | len | Total length, in bytes, to compare. |
| Note | | |
| See Also | | |

```
buf1 = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 512);
buf2 = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 512);

nvme::ddbFillPatRand(EC, buf1);
nvme::ddbFillPat8(EC, buf2, 0);

writeCmd = nvme::ioc::write(EC, sq1, buf1, 0, 1);
flushCmd = nvme::ioc::flush(EC, sq1, 0xFFFFFFFF);
readCmd  = nvme::ioc::read (EC, sq1, buf2, 0, 1);

...
// wait command completions
if (nvme::ddbCompareN(EC, buf1, buf2, 512) != 0)
    c::printf("miscompare\n");
else
    c::printf("identical\n");
```

dumpTrace()

| | | |
|---|---|---|
| Prototype | `dumpTrace(&EC, sqctHdle, fileHdle, limit)` | |
| Description | Dump trace of commands and events in the console, and optionally save them into a file. | |
| Return Value | 0 : failed<br>1 : successful | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | sqctHdle | An SQ handle or a controller handle. |
| | fileHdle | A file handle for saving the trace.<br>If 0 is given, contents will be displayed in the console only. |
| | limit | The maximum number of commands or events to be dumped.<br>If 0 is given, all commands and events are dumped. |
| Note | Each SQ keeps the latest 9999 command records.<br>Each controller keeps the latest 127 event records.<br>If sqctHdle is an SQ handle, only the latest records of the commands issued to that SQ will be dumped.<br>If sqctHdle is a controller handle, the latest commans belonging to any SQ of this controller or events will be dumped. | |
| See Also | | |

```
// dump commands executed in init()

if (!nvme::scan(EC))
{
    c::util::logPrintf(logFile, "Error: Fail to scan; error %d\n", EC);
    goto Return;
}

nDevs = nvme::getDevNum();
if (nDevs <= 0)
{
    c::util::logPrintf(logFile, "Error: No device is detected!");
    goto Return;
}
c::util::logPrintf(logFile, "Detect %d devices.\n\n", nDevs);

//
// initialize each device (physical function)
//
bInitError = 0;
for (n = 0; n < nDevs; ++n)
{
    cntlHandle = nvme::getCntlHandle(EC, n, 0);
    if (!nvme::init(EC, cntlHandle))
    {
        c::util::logPrintf(logFile, "Error: Device%d: EC = %d\n", n, EC);
        bInitError = 1;
    }
```

```
    else
        nvme::dumpTrace(EC, cntlHandle, logFile, 0);
}
```

cleanTrace()

| | | |
|---|---|---|
| Prototype | `cleanTrace(&EC, sqctHdle)` | |
| Description | Clean the command and event records. | |
| Return Value | 0 : failed<br>1 : successful | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | sqctHdle | An SQ handle or a controller handle. |
| Note | If sqctHdle is a controller handle, all SQ's command records and all event records will be cleaned up.<br>If sqctHdle is an SQ handle, only the command reords of this SQ will be cleaned up. | |
| See Also | | |

```
nvme::dumpTrace(EC, cntlHandle, 0, 0);
nvme::ioc::flush(EC, sq1Handle, 0xFFFFFFFF);
nvme::ioc::read(EC, sq2Handle, buf, 0, 1);
nvme::dumpTrace(EC, sq1Handle, 0, 0);
nvme::dumpTrace(EC, sq2Handle, 0, 0);
nvme::dumpTrace(EC, cntlHandle, 0, 0);
```

getAllCmdIssueNum()
getAllCmdComplNum()

| | | |
|---|---|---|
| Prototype | `getAllCmdIssueNum(&EC, cntlHdle)`<br>`getAllCmdComplNum(&EC, cntlHdle)` | |
| Description | Get the number of issued/completed commands of all SQs (the Admin SQ and all IO SQs) of the controller. | |
| Return Value | -1 : failed<br>Otherwise : The number of issued commands. | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | cntlHdle | A controller handle. |
| Note | | |
| See Also | | |

```
iNum = nvme::getAllCmdIssueNum(EC, cntlHandle);
cNum = nvme::getAllCmdComplNum(EC, cntlHandle);

if (iNum == cNum)
    c::printf("No command pended.\n");
else if (iNum > cNum)
    c::printf("Total %d commands pended\n",  iNum – cNum);
else
    c::printf("Error!\n");
```

getAllIOSQCmdIssueNum()
getAllIOSQCmdComplNum()

| | | |
|---|---|---|
| Prototype | `getAllIOSQCmdIssueNum(&EC, cntlHdle)`<br>`getAllIOSQCmdIssueNum(&EC, cntlHdle, opcode)`<br>`getAllIOSQCmdComplNum(&EC, cntlHdle)`<br>`getAllIOSQCmdComplNum(&EC, cntlHdle, opcode)` | |
| Description | Get the number of issued/completed commands of all IO SQs of the controller. | |
| Return Value | -1 : failed<br>Otherwise : The number of issued commands. | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | cntlHdle | A controller handle. |
| | opcode (optional) | The opcode of the command. |
| Note | With opcode specified, only returns the number of issued/completed commands of the opcode.<br>Without opcode specified, returns the number of issued/completed commands of any opcode. | |
| See Also | | |

```
var stopFlag_g = 0;

randReadTask(sqHandle)
{
    var EC;
    var cntlHandle = nvme::getCntlHandleBySQHandle(EC, sqHandle);

    ...
    while (!stopFlag_g)
    {
        // iteratively issue read commands to the IOSQ (sqHandle)
        ...
    }

    // wait command completions
    while (nvme::getSQCmdComplNum(EC, sqHandle) !=
           nvme::getSQCmdIssueNum(EC, sqHandle))
    {
        sleep(1);
    }
}

main()
{
    ...
    // invoke 1 task to issue read commands through each IOSQ
    idx = 0;
    for (sqid = 0, sqid = nvme::getNextIOSQId(EC, cntlHandle, sqid);
         sqid > 0;
```

```
          sqid = nvme::getNextIOSQId(EC, cntlHandle, sqid))
    {
        sqHandle = getSQHandle(EC, cntlHandle, sqid);
        tids[idx++] = task_create(randReadTask, sqHandle);
    }
    num = idx;

    N0 = nvme:: getAllIOSQCmdComplNum(EC, cntlHandle);
    stime = c::time();
    while (c::time() - stime < 60)
    {
        n0 = nvme::getAllIOSQCmdComplNum(EC, cntlHandle);
        sleep(1000);
        n1 = nvme::getAllIOSQCmdComplNum(EC, cntlHandle);
        c::printf("IOPS = %d\n", n1 - n0);
    }

    stopFlag_g = 1;
    for (idx = 0; idx < num; ++idx)
        task_join(tids[idx]);
    N1 = getAllIOSQCmdComplNum(EC, cntlHandle);
    etime = c::time();

    c::printf("Average IOPS = %d\n", (N1 - N0) / (etime - stime));
    ...
}
```

getSQCmdIssueNum()
getSQCmdComplNum()

| | | |
|---|---|---|
| Prototype | `getSQCmdIssueNum(&EC, sqHdle)`<br>`getSQCmdIssueNum(&EC, sqHdle, opcode)`<br>`getSQCmdComplNum(&EC, sqHdle)`<br>`getSQCmdComplNum(&EC, sqHdle, opcode)` | |
| Description | Get an SQ's number of issued/completed commands. | |
| Return Value | -1 : failed<br>Otherwise : The number of issued commands. | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is -1. |
| | sqHdle | An SQ (an Admin SQ or an IO SQ) handle. |
| | opcode (optional) | The opcode of the command. |
| Note | With opcode specified, only returns the number of issued/completed commands of the opcode.<br>Without opcode specified, returns the number of issued/completed commands of any opcode. | |
| See Also | | |

```
var stopFlag_g = 0;

randReadTask(sqHandle)
{
    var EC;
    var cntlHandle = nvme::getCntlHandleBySQHandle(EC, sqHandle);

    ...
    while (!stopFlag_g)
    {
        // iteratively issue read commands to the IOSQ (sqHandle)
        ...
    }

    // wait command completions
    while (nvme::getSQCmdComplNum(EC, sqHandle) !=
           nvme::getSQCmdIssueNum(EC, sqHandle))
    {
        sleep(1);
    }
}

main()
{
    ...
    sqHandle = getSQHandle(EC, cntlHandle, sqid=1);
    tid = task_create(randReadTask, sqHandle);

    stime = c::time();
```

```
    while (c::time() - stime < 60)
    {
        n0 = nvme::getSQCmdComplNum(EC, sqHandle);
        sleep(1000);
        n1 = nvme::getSQCmdComplNum(EC, sqHandle);
        c::printf("IOPS = %d\n", n1 - n0);
    }

    stopFlag_g = 1;
    task_join(tid);

    ...
}
```

pspp()

| | | |
|---|---|---|
| Prototype | `pspp(cmdHdle)`<br>`pspp(cmdHdle, &exetime)` | |
| Description | Poll command status, do post processing, and retrieve command execution time. | |
| Return Value | 0 : not completed and not timeout; no status at all<br>Otherwise : a 64-bit status value which contains: 1-bit completion flag, 1-bit timeout flag, 15-bit Status Field value coming from the completion entry DW3, and 32-bit Command Specific value coming from the completion entry DW0. Each information can be retrieved by bitwise operations with the following constants: nvme::cmdCplFlag, nvme::cmdToFlag, nvme::cmdStsMask, nvme::cmdStsLowBit, nvme::cmdSpcStsMask, and nvme::cmdSpcLowBit. | |
| Parameter | cmdHdle | A command handle. |
| | exetime (optional) | This parameter is a call-by-reference one. The argument should be an elementary variable. Command execution time in micro seconds is updated into this varable for the caller. |
| Note | | |
| See Also | pspp2() | |

```
showStatus(stsVal)
{
    var Status, SCTDescp, SCDescp, nCmdSpec;

    Status = (stsVal & nvme::cmdStsMask) >> nvme::cmdStsLowBit;
    if (Status) {
        c::printf("  stsVal = %Xh, Status = %04Xh, ", stsVal, Status);
        GetNVMeCmdCplStatusDescription(Status, SCTDescp, SCDescp);
        c::printf("  SCT: %s, SC: %s\n", SCTDescp, SCDescp);
    }

    nCmdSpec = (stsVal & nvme::cmdSpcStsMask) >> nvme::cmdSpcStsLowBit;
    if (nCmdSpec)
        c::printf("  Command specific status %08Xh\n", nCmdSpec);

    if (stsVal & nvme::cmdToFlag)
        c::printf("  Comand timeout\n");
}

main()
{
    ...
    // launch a read; expect "Invalid Field in Command" replied
    buf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 1024);
    cmdHandle = nvme::ioc::read(EC, sqHandle, buf, 0xFFFFFFFFFFFF, 0x1000);
    do
        stsVal = nvme::pspp(cmdHandle);
    while (stsVal == 0);

    if (stsVal != nvme::cmdCplFlag)
        showStatus(stsVal);
}
```

pspp2()

| | | |
|---|---|---|
| Prototype | `pspp2(cmdHdle, &CQEDWs[])`<br>`pspp2(cmdHdle, &CQEDWs[], &exetime)` | |
| Description | Poll command status, do post processing, retrive completion queue entry values, and retrieve command execution time. | |
| Return Value | 0 : not completed and not timeout; no status at all<br>Otherwise : a 64-bit status value which contains: 1-bit completion flag, 1-bit timeout flag, 15-bit Status Field value coming from the completion entry DW3, and 32-bit Command Specific value coming from the completion entry DW0. Each information can be retrieved by bitwise operations with the following constants: nvme::cmdCplFlag, nvme::cmdToFlag, nvme::cmdStsMask, nvme::cmdStsLowBit, nvme::cmdSpcStsMask, and nvme::cmdSpcLowBit. | |
| Parameter | cmdHdle | A command handle. |
| | CQEDWs | A call-by-reference array with at least 4 elements for storing the completion queue entry values. |
| | exetime (optional) | This parameter is a call-by-reference one. The argument should be an elementary variable. Command execution time in micro seconds is updated into this varable for the caller. |
| Note | | |
| See Also | | |

```
main()
{
    var CQEDWs[4];
    ...
    // launch a read; expect "Invalid Field in Command" replied
    buf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 1024);
    cmdHandle = nvme::ioc::read(EC, sqHandle, buf, 0xFFFFFFFFFFFF, 0x1000);
    do
        stsVal = nvme::pspp2(cmdHandle, CQEDWs);
    while (stsVal == 0);

    c::printf("CQE =");
    for (i = 0; i < 4; ++i)
        c::printf(" %08X", CQEDWs[i]);
    c::printf("\n");

    if (stsVal != nvme::cmdCplFlag)
        showStatus(stsVal);
}
```

getCId()

| | | |
|---|---|---|
| Prototype | `getCId(cmdHdle)` | |
| Description | Get command identifier. | |
| Return Value | -1 : failed<br>Otherwise : a 16-bit command Identifier used to send this command. | |
| Parameter | cmdHdle | A command handle. |
| Note | | |
| See Also | | |

```
buf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 4096);

cmdHandle = nvme::admc::identify(EC, cntlHandle, buf, 0, 1, 0);
c::printf("command Id = %04Xh\n", getCId(cmdHandle));
nvme::dumpTrace(EC, cntlHandle, 0, 1);
...
cmdHandle = nvme::ioc::read(EC, iosq1, buf, 0, 1);
c::printf("command Id = %04Xh\n", getCId(cmdHandle));
```

getCDWs()

| | | |
|---|---|---|
| Prototype | `getCDWs(cmdHdle, &cdws[])` | |
| Description | Get the 16 command Dwords. | |
| Return Value | 0 : failed<br>1 : successful | |
| Parameter | cmdHdle | A command handle. |
| | cdws | An 1-dimension array variable with at least 16 elements for storing the 16 command Dwords. |
| Note | | |
| See Also | | |

```
var CDWs[16];

cmdHandle = nvme::admc::identify(EC, cntlHandle, buf, 0, 1, 0);
nvme::getCDWs(cmdHandle, CDWs);
for (i = 0; i < 16; ++i)
    c::printf("CDW[%2d] = %08X\n", i, CDWs[i]);
```

setCMDTO()

| | | |
|---|---|---|
| Prototype | `setCMDTO(time_sec)` | |
| Description | Set command timeout value. | |
| Return Value | -1 : failed<br>Otherwise : the setting value | |
| Parameter | time_sec | The timeout value; in seconds. |
| Note | | |
| See Also | | |

```
// increase the command timeout value by 1
nvme::setCMDTO(nvme::getCMDTO() + 1);
```

getCMDTO()

| Prototype | `getCMDTO()` |
|---|---|
| Description | Get the command timeout value in seconds. |
| Return Value | Current command timeout value in seconds. |
| Parameter | N/A |
| Note | |
| See Also | |

```
// increase the command timeout value by 1
nvme::setCMDTO(nvme::getCMDTO() + 1);
```

strerror()

| | | |
|---|---|---|
| Prototype | `strerror(EC)` | |
| Description | Get the string describing the error code passed in the argument EC. | |
| Return Value | A string describing the error code. | |
| Parameter | EC | An error code. |
| Note | The error code is returned through the 1st parameter of the function If the function is devised to return the error code.<br>A string of below format is returned for an undefined error code, where ???? is the passed error code.<br>"Error code 0x???? is undefined." | |
| See Also | | |

```
print(nvme::strerror(0x12345678), "\n");

if (!nvme::scan(EC))
{
    print(nvme::strerror(EC), "\n");
    return;
}
```

# Extension functions in namespace nvme::admc

deleteIOSQ()

| | | |
|---|---|---|
| Prototype | `deleteIOSQ(&EC, cmdHdle, iosqId)` | |
| Description | Isse a Delete I/O Submission Queue command. | |
| Return Value | 0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | iosqId | An IO SQ Id. |
| Note | | |
| See Also | | |

```
// delete all IOSQs

sqid = 0;
while (1)
{
    sqid = getNextIOSQId(EC, cntlHandle, sqid);
    if (sqid <= 0)
        break;

    cmdHandle = nvme::admc::deleteIOSQ(EC, cntlHandle, sqid);
    ...
}
```

createIOSQ()

| | | |
|---|---|---|
| Prototype | `createIOSQ(&EC, ch, isq, icq)`<br>`createIOSQ(&EC, ch, isq, icq, size)`<br>`createIOSQ(&EC, ch, isq, icq, size, prio)`<br>`createIOSQ(&EC, ch, isq, icq, size, prio, nvmset)` | |
| Description | Issue a Create I/O Submission Queue command | |
| Return Value | 0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | ch | A controller handle. |
| | isq | Identifier of the created IO Submission queue. |
| | icq | Identifier of the paired IO Completion queue. |
| | size (optional) | Size of the created IO Submission queue; the number of entries. |
| | prio (optional) | Priority class to use for commands within this Submission Queue.<br>0 : Urgent<br>1 : High<br>2 : Medium<br>3 : Low |
| | nvmset (optional) | Identifier of the NVM Set to be associated with this Submission Queue. |
| Note | | |
| See Also | | |

```
// create IOSQs to which cqid is their paired completion queue Id

for (sqid = 1; sqid <= DEF_IOSQ_NUM; ++sqid)
    cmds[sqid-1] = nvme::admc::createIOSQ(EC, cntlHandle, sqid, cqid);

for (sqid = 1; sqid <= DEF_IOSQ_NUM; ++sqid)
{
    stsVal = nvme::pspp(cmds[sqid-1]);
    ...
}
```

getLogPage()

<table>
<tr><td>Prototype</td><td colspan="2">getLogPage(&EC, cntlHdle, buf, pgId, len, nsid)<br>getLogPage(&EC, cntlHdle, buf, pgId, len, nsid, csi)<br>getLogPage(&EC, cntlHdle, buf, pgId, len, nsid, csi, struct nvme::getLogPageOptParams_t optParam)</td></tr>
<tr><td>Description</td><td colspan="2">Issue a Get Log Page command.</td></tr>
<tr><td>Return Value</td><td colspan="2">0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully</td></tr>
<tr><td rowspan="8">Parameter</td><td>EC</td><td>This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0.</td></tr>
<tr><td>cntlHdle</td><td>A controller handle.</td></tr>
<tr><td>buf</td><td>A buffer handle. For receiving the data structure.</td></tr>
<tr><td>pgId</td><td>A 8-bit log page identifier.</td></tr>
<tr><td>len</td><td>Length of log page size; in bytes.</td></tr>
<tr><td>nsid</td><td>An namespace Id.</td></tr>
<tr><td>csi<br>(optional)</td><td>Command Set Identifier.<br>0 : NVM Command Set<br>1 : Key Value Command Set<br>2 : Zoned Namespace Command Set</td></tr>
<tr><td>optParam<br>(optional)</td><td>The argument should be a nvme::getLogPageOptParams_t structure variable. This structure is defined by the extension module as the following:<br><pre>struct getLogPageOptParams_t {
  var CDW10_RAE;
  var CDW10_LSP;
  var CDW11_LSId;
  var LogPageOffset;
  var CDW14_OffsetType;
  var CDW14_UUIDIndex;
  clean()
  {
    CDW10_RAE = 0;
    CDW10_LSP = 0;
    CDW11_LSId = 0;
    LogPageOffset = 0;
    CDW14_OffsetType = 0;
    CDW14_UUIDIndex = 0;
  }
};</pre>The CDW10_RAE is the Retain Asynchronous Event (RAE) field of the command DWORD 10 (CDW10).<br>The CDW10_LSP is the Log Specific Field of CDW10.<br>CDW11_LSId is the Log Specific Identifier field of CDW11.<br>LogPageOffset is the Log Page Offset value of CDW12 and CDW13.<br>CDW14_UUIDIndex is the UUIDIndex field of CDW14.<br>If there is argument for this parameter, value 0’s are set into</td></tr>
</table>

| | | |
|---|---|---|
| | | corresponding fields of the command entry. |
| Note | | |
| See Also | | |

```
// monitor one IOSQ’s IOPS performance and temperature

while (1)
{
    nCmdCplNum0 = nvme::getSQCmdComplNum(EC, sqHandle);
    sleep(1000);
    nCmdCplNum1 = nvme::getSQCmdComplNum(EC, sqHandle);

    cmdHandle =
        nvme::admc::getLogPage(EC, cntlHandle, buf, 2, 512, 0xFFFFFFFF);
    while (nvme::pspp(cmdHandle) == 0)
        ;
    nTemp = nvme::ddbReadUInt16(EC, buf, 1, 1 /*little endian*/);
    c::printf("%12d, %8.1f\n", nTemp, (nCmdCplNum1 - nCmdCplNum0) / 1000.0);
}
```

deleteIOCQ()

| | | |
|---|---|---|
| Prototype | `deleteIOCQ(&EC, cntlHdle, iocqid)` | |
| Description | Issue a Delete I/O Completion Queue command. | |
| Return Value | 0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | iocqid | An IO completion queue Id. |
| Note | | |
| See Also | | |

```
// delete iocq; before that, delete all its paired IOSQs

// delete iocq's paired IOSQs
sqid = 0;
while (1)
{
    sqid = nvme::getNextPairedIOSQId(EC, cntlHandle, cqid, sqid);
    if (sqid == 0)
        break;

    cmd = nvme::admc::deleteIOSQ(EC, cntlHandle, sqid);
    ...
}

// delete iocq
cmd = nvme::admc::deleteIOCQ(EC, cntlHandle, cqid);
...
```

createIOCQ()

| Prototype | `createIOCQ(&EC, cntlHdle, cqid)`<br>`createIOCQ(&EC, cntlHdle, cqid, size)` | |
|---|---|---|
| Description | Issue a Create I/O Completion Queue command. | |
| Return Value | 0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | cqid | The identifier of the IO completion queue. |
| | size | The size of the completion queue; the number of entries of the completion queue. |
| Note | | |
| See Also | | |

```
cmds[1] = createIOCQ(EC, cntlHandle, 1);
cmds[2] = createIOCQ(EC, cntlHandle, 5, 256);
```

identify()

| Prototype | identify(&EC, cntlHdle, buf, cns, nsid, CNTID)<br>identify(&EC, cntlHdle, buf, cns, nsid, CNTID, struct nvme::identifyOptParams_t optval) | |
|---|---|---|
| Description | Issue an Identify command. | |
| Return Value | 0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | buf | A buffer handle for receving the data structure. |
| | cns | Controller or Namespace Structure (CNS): This field specifies the information to be returned to the host. |
| | nsid | An namespace identifier. |
| | CNTID | Controller Identifier (CNTID): This field specifies the controller identifier used as part of some Identify operations. |
| | optval (optional) | Other CDW values of the Identify command. The argument should be a nvme:: identifyOptParams_t structure variable. This structure is defined by the extension module as the following:<br>`struct identifyOptParams_t`<br>`{`<br>`  var CDW11_CNSSpecID;`<br>`  var CDW11_CSI;`<br>`  var CDW14_UUIDIdx`<br>`  clean()`<br>`  {`<br>`  CDW11_CNSSpecID = 0;`<br>`  CDW11_CSI = 0;`<br>`  CDW14_UUIDIdx = 0;`<br>`  }`<br>`};` |
| Note | | |
| See Also | | |

```
  buf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 4096);

  // identify namespace data structure
  cmdHandle = nvme::admc::identify(EC, cntlHandle, buf, 0, nsid, 0);
  do
      stsVal = nvme::pspp(cmdHandle);
  while (stsVal == 0);
  nCap = nvme::ddbReadInt64(EC, buf, 8, 1);

  // identify controller data structure
  cmdHandle = nvme::admc::identify(EC, cntlHandle, buf, 1, 0, 0);
```

```
do
    stsVal = nvme::pspp(cmdHandle);
while (stsVal == 0);
nvme:: ddbReadASCIIString(EC, buf, FirmwareRevision, 64, 8);
```

abort()

| | | |
|---|---|---|
| Prototype | `abort(&EC, cntlHdle, cmdHdle)`<br>`abort(&EC, cntlHdle, sqid, cid)` | |
| Description | Issue an Abort command. | |
| Return Value | 0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | cmdHdle | A command handle; the command to be aborted. |
| | sqid | A submission queue identifier of the command to be aborted. |
| | cid | A command identifier of the command to be aborted. |
| Note | | |
| See Also | | |

```
for (nsid = 1; nsid <= nNSIdNum; ++nsid)
{
    do
        idchs[n] = nvme::admc::identify(EC, cntlHandle, buf[n], 0, nsid, 0);
    while (idchs[n] == 0);

    do
        abchs[n] = nvme::admc::abort(EC, cntlHandle, idchs[n]);
    while (abchs[n] == 0);
}
```

setFeatures()

| | | |
|---|---|---|
| Prototype | `setFeatures(&EC, cntlHdle, FId, SV, CDW11)`<br>`setFeatures(&EC, cntlHdle, FId, SV, CDW11, buf)`<br>`setFeatures(&EC, cntlHdle, FId, SV, CDW11, buf, struct nvme:: setFeaturesOptParams_t optVal)` | |
| Description | Issue a Set Features command. | |
| Return Value | 0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | FId | The feature Id. |
| | SV | 0 or 1.<br>1 = Specifies that the controller shall save the attribute so that the attribute persists through all power states and resets. |
| | CDW11 | The feature value. |
| | buf | A buffer handle coveying data structure value of the feature. 0 is specified if no data structure is defined for the feature Id. |
| | optVal (optional) | Other CDW values of the Set Features command. The argument should be a nvme:: setFeaturesOptParams_t structure variable. This structure is defined by the extension module as the following:<br>`struct setFeaturesOptParams_t {`<br>`  var NamespaceId;`<br>`  var CDW12;`<br>`  var CDW13;`<br>`  var CDW14;`<br>`  var CDW15;`<br>`  clean()`<br>`  {`<br>`  NamespaceId = 0;`<br>`  CDW12 = 0;`<br>`  CDW13 = 0;`<br>`  CDW14 = 0;`<br>`  CDW15 = 0;`<br>`  }`<br>`};` |
| Note | | |
| See Also | | |

```
  // enable Write Cache
  cmdHandle = nvme::setFeatures(EC, cntlHandle, 6, 0, 1);
```

getFeatures()

| | | |
|---|---|---|
| Prototype | `getFeatures(&EC, ch, FID, SEL)`<br>`getFeatures(&EC, ch, FID, SEL, buf)`<br>`getFeatures(&EC, ch, FID, SEL, buf, CDW11)`<br>`getFeatures(&EC, ch, FID, SEL, buf, CDW11, nsid)`<br>`getFeatures(&EC, ch, FID, SEL, buf, CDW11, nsid, CDW14)` | |
| Description | Issue a Get Features command. | |
| Return Value | 0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | ch | A controller handle. |
| | FID | The feature Id. |
| | SEL | Specifies which value of the attributes to return in the provided data.<br>0 : Current<br>1 : Default<br>2 : Saved<br>3 : Supported Capabilities |
| | buf (optional) | A buffer handle for storing structure data got. |
| | CDW11 (optional) | Value of command Dword 11. Refer to the NVMe spec. for the format. |
| | nsid (optional) | An namespace Id. |
| | CDW14 (optional) | Value of command Dword 14. Refer to the NVMe spec. for the format. |
| Note | | |
| See Also | | |

asyncEvtReq()

| Prototype | asyncEvtReq(&EC, cntlHdle) | |
|---|---|---|
| Description | Isse an Asynchronous Event Request command | |
| Return Value | 0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| Note | | |
| See Also | | |

```
const MAX_ASYNC_EVENTS = 128;

struct cntlInfo_t cntl_g;
var bExitAERMonitor_g = 0;
var AERCmdNum_g;
var AERCmdHdles_g[MAX_ASYNC_EVENTS];

AERMonitor(param)
{
    var EC;
    var i, CQE_DW0;
    var cntlHandle = cntl_g.m_CntlHandle;
    var stsVal;

    AERCmdNum_g = 0;
    for (i = 0; i < MAX_ASYNC_EVENTS; ++i)
        AERCmdHdles_g[i] = 0;

    AERCmdNum_g = MAX_ASYNC_EVENTS < cntl_g.m_AERL_0Based + 1 ?
                  MAX_ASYNC_EVENTS : cntl_g.m_AERL_0Based + 1;

    for (i = 0; i < AERCmdNum_g; ++i)
    {
        do
            AERCmdHdles_g[i] = nvme::admc::asyncEvtReq(EC, cntlHandle);
        while (AERCmdHdles_g[i] == 0);
    }

    while (!bExitAERMonitor_g)
    {
        sleep(1000);    // sleep for 1 second
        for (i = 0; i < AERCmdNum_g; ++i)
        {
            stsVal = nvme::pspp(AERCmdHdles_g[i]);

            // A command is processed only if it is completed.
            if (!(stsVal & nvme::cmdCplFlag))
```

```
                continue;

            CQE_DW0   = stsVal & nvme::cmdSpcStsMask;
            CQE_DW0 >>= nvme::cmdSpcStsLowBit;
            CQE_DW0  &= 0xFFFFFFFF;
            c::printf("CQE.DW0 = %08Xh\n", stsVal);

            // patch one request
            do
                AERCmdHdles_g[i] = nvme::admc::asyncEvtReq(EC, cntlHandle);
            while (AERCmdHdles_g[i] == 0);
        }
    }
}

main()
{
    ...

    cntl_g.Invalidate();
    cntl_g.RetrieveInfo(0, cntlHandle);

    tid = task_create(AERMonitor, 0);

    ...
}
```

nsMgmt()

| | | |
|---|---|---|
| Prototype | `nsMgmt(&EC, cntlHdle, SEL=0, buf)`<br>`nsMgmt(&EC, cntlHdle, SEL=0, buf, csi)`<br>`nsMgmt(&EC, cntlHdle, SEL=1, nsid)` | |
| Description | Issue a Namespace Management command. | |
| Return Value | 0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | SEL | Select the type of management operation to perform.<br>0 : Create<br>1 : Delete |
| | buf | A buffer handle; the buffer convey partial fields of identify namespace data structure for the namespace creation. Refer to the NVMe spec. for the detail.<br>A buffer handle should be specified when the type of management operation is 0 (Create). |
| | nsid | An namespace Id to be deleted.<br>An namespace Id should be specified when the type of management operation is 1 (Delete). |
| | csi (optional) | A 8-bit Command Set Identifier. This parameter is ignored if SEL is NOT 0.<br>0 : NVM Command Set<br>1 : Key Value Command Set<br>2 : Zoned Namespace Command Set |
| Note | | |
| See Also | | |

```
// return value: namespace id
createNamespace(fd, PCHandle, struct NSCreationParam_t &params)
{
    var EC;
    var buf;
    var bLE = 1;
    var cmdHandle, cmdStatus;
    var SEL, nsid;

    buf = nvme::ddbAlloc(EC, PCHandle, nvme::PRP, 4096);
    nvme::ddbFillPat8(EC, buf, 0);

    // Bytes 07:00 = Namespace Size
    nvme::ddbWriteInt64(EC, buf, 0, bLE, params.nNamespaceSize_lba);

    // Bytes 15:08 = Namespace Capacity
    nvme::ddbWriteInt64(EC, buf, 8, bLE, params.nNamespaceCapacity_lba);
```

```
    // Byte 26 = Formatted LBA Size
    nvme::ddbWriteUInt8(EC, buf, 26, params.nFormattedLBASize_idfy_byte26);

    // ... fill other fields

    SEL = 0;    // 0 = create
    cmdHandle = nvme::admc::nsMgmt(EC, PCHandle, SEL, buf);
    do
        cmdStatus = nvme::pspp(cmdHandle);
    while (cmdStatus == 0);

    nvme::ddbFree(EC, buf);

    nsid = ((cmdStatus & nvme::cmdSpcStsMask) >> nvme::cmdSpcStsLowBit) &
           0xFFFFFFFF;

    return (nsid);
}
```

fwCommit()

| | | |
|---|---|---|
| Prototype | `fwCommit(&EC, cntlHdle, slotNo, act)`<br>`fwCommit(&EC, cntlHdle, slotNo, act, bpid)` | |
| Description | Issue a Firmware Commit command. | |
| Return Value | 0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | slotNo | Firmware Slot (FS): Specifies the firmware slot that shall be used for the Commit Action, if applicable. If the value specified is 0h, then the controller shall choose the firmware slot (i.e., slot 1 to slot 7) to use for the operation. |
| | act | This field specifies the action that is taken (refer to section 8.1) on the image downloaded.<br>Refer to the NVMe spec. for the meanings of values set to Commit Action (CA) field of CDW10 when issuing a Firmware Commit command. |
| | bpid | Boot Partition ID. Specifies the Boot Partion that shall be used for the Commit action, if applicable.<br>Valid values: 0, 1 |
| Note | | |
| See Also | | |

fwImgDwnld()

| Prototype | `fwImgDwnld(&EC, cntlHdle, buf, size_dws, offset)` | |
|---|---|---|
| Description | Issue a Firmware Image Download command. | |
| Return Value | 0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | buf | A buffer handle; the buffer with firmware image content |
| | **size_dws** | **Size of the image content, in Dwords.**<br>**1 = 1 Dword.**<br>**2 = 2 Dwords.**<br>**N = N Dwords.** |
| | offset | Specifies the number of dwords offset from the start of the firmware image being downloaded to the controller. The offset is used to construct the complete firmware image when the firmware is downloaded in multiple pieces. |
| Note | | |
| See Also | | |

```
showStatus(stsVal)
{
    var Status, SCTDescp, SCDescp, nCmdSpec;

    Status = (stsVal & nvme::cmdStsMask) >> nvme::cmdStsLowBit;
    if (Status) {
        c::printf("  stsVal = %Xh, Status = %04Xh, ", stsVal, Status);
        GetNVMeCmdCplStatusDescription(Status, SCTDescp, SCDescp);
        c::printf("  SCT: %s, SC: %s\n", SCTDescp, SCDescp);
    }

    nCmdSpec = (stsVal & nvme::cmdSpcStsMask) >> nvme::cmdSpcStsLowBit;
    if (nCmdSpec)
        c::printf("  Command specific status %08Xh\n", nCmdSpec);

    if (stsVal & nvme::cmdToFlag)
        c::printf("  Comand timeout\n");
}

// return value:
//  1 : success
//  0 : fail
fw_download(logfd, cntlHandle, bufHandle, DWordsNum, offset)
{
    var EC;
    var cmdHandle = 0;
```

```
    var tries = 0;
    var stsVal;

    // send the command
    while (cmdHandle == 0 && tries < 5)
    {
        if (tries)
            sleep(50);
        cmdHandle = nvme::admc::fwImgDwnld(EC,
                                           cntlHandle,
                                           bufHandle,
                                           DWordsNum,
                                           offset);
        ++tries;
    }

    if (cmdHandle == 0)
        return (0);

    // poll the command status
    while (1)
    {
        stsVal = nvme::pspp(cmdHandle);
        if (stsVal == nvme::cmdCplFlag)
            break;
        else if (stsVal)
        {
            c::util::logPrintf(0, "Error: 'fwImgDwnld' command fail\n");
            showStatusErrors(stsVal);
            return (0);
        }
        sleep(2);
    }

    return (1);
}

main()
{

    ...
    nTxSize = PF.GetFWUpdateGranu();

    ddbBuf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, nTxSize);
    memHandle = c::malloc(nTxSize);
    nSrcMemAddr = c::util::getMemAddr(memHandle);

    c::fseek(fwImgFileHandle, 0, c::SEEK_SET);
    for (nAccReadSize = 0;
         nAccReadSize < fw_filesize;
         nAccReadSize += nThisReadSize)
    {
        nvme::ddbFillPat8(EC, ddbBuf, 0);
        if (fw_filesize - nAccReadSize > nTxSize)
            nThisReadSize = nTxSize;
        else
            nThisReadSize = fw_filesize - nAccReadSize;
```

```
        c::fread(memHandle, 1, nThisReadSize, fwImgFileHandle);
        nvme::ddbCopyFromMemAddr(EC, ddbBuf, 0, nSrcMemAddr, nThisReadSize);

        r = fw_download(logfd,
                        cntlHandle,
                        ddbBuf,
                        nTxSize/4,
                        nAccReadSize/4);
        if (!r)
        {
            nvme::ddbFree(EC, ddbBuf);
            break;
        }
    }
}
```

devSelfTest()

| | | |
|---|---|---|
| Prototype | `devSelfTest(&EC, cntlHdle, STC, nsid)` | |
| Description | Isse a Device Self-test command. | |
| Return Value | 0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | STC | The action taken by the Device Self-test command.<br>1 : Start a short device self-test operation.<br>2 : Start an extended device self-test operation.<br>Eh : Vendor specific<br>Fh : Abort device self-test operation. |
| | nsid | An namespace Id. |
| Note | | |
| See Also | | |

| |
|---|
| |

nsAtchmt()

| Prototype | nsAtchmt(&EC, cntlHdle, buf, SEL, nsid) | |
|---|---|---|
| Description | Issue a Namespace Attachment command. | |
| Return Value | 0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | buf | The buffer handle where the buffer contains a 4096-byte structure defing a list of controllers. Refer to the NVMe spec. for the structure format. |
| | SEL | Selects the type of attachment to perform.<br>0 : Controller Attach<br>1 : Controller Detach |
| | nsid | A namespace Id. |
| Note | | |
| See Also | | |

```
attachNamespace(fd, PCHandle, CNTLID, nsid)
{
    var EC;
    var buf;
    var bLE = 1;
    var cmdHandle, cmdStatus;
    var SEL;

    buf = nvme::ddbAlloc(EC, PCHandle, nvme::PRP, 4096);
    nvme::ddbFillPat8(EC, buf, 0);

    nvme::ddbWriteUInt16(EC, buf, 0, bLE, 1);
    nvme::ddbWriteUInt16(EC, buf, 2, bLE, CNTLID);

    SEL = 0;    // attachment
    cmdHandle = nvme::admc::nsAtchmt(EC, PCHandle, buf, SEL, nsid);
    do
        cmdStatus = nvme::pspp(cmdHandle);
    while (cmdStatus == 0);

    ...

    nvme::ddbFree(EC, buf);
}
```

keepAlive()

| | | |
|---|---|---|
| Prototype | `keepAlive(&EC, cntlHdle)` | |
| Description | Issue a Keep Alive command. | |
| Return Value | 0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| Note | | |
| See Also | | |

dirSend()

| | | |
|---|---|---|
| Prototype | `dirSend(&EC, cntlHdle, buf, nDWNum, struct nvme::dirSendParams_t sendParams)`<br>`dirSend(&EC, cntlHdle, buf, nDWNum, struct nvme::dirSendParams_t sendParams, nsid)` | |
| Description | Issue a Directive Send command. | |
| Return Value | 0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | buf | The data buffer. |
| | **nDWNum** | **Number of dwords to transfer.**<br>**1 = 1 dword**<br>**2 = 2 dwords**<br>**N = N dwords** |
| | sendParams | Other CDW values of the Directive Send command. The argument should be a nvme::dirSendParams_t structure variable. This structure is defined by the extension module as the following:<br>`struct dirSendParams_t {`<br>`  var CDW11_dirOp;`<br>`  var CDW11_dirType;`<br>`  var CDW11_dirSpec;`<br>`  var CDW12;`<br>`  var CDW13;`<br>`  clean() {`<br>`  CDW11_dirOp = 0;`<br>`  CDW11_dirType = 0;`<br>`  CDW11_dirSpec = 0;`<br>`  CDW12 = 0;`<br>`  CDW13 = 0;`<br>`  }`<br>`};` |
| | nsid (optional) | A namespace ID. |
| Note | | |
| See Also | | |

dirRecv()

| | | |
|---|---|---|
| Prototype | `dirRecv(&EC, cntlHdle, buf, nDWNum, struct nvme::dirRecvParams_t recvParams)`<br>`dirRecv(&EC, cntlHdle, buf, nDWNum, struct nvme::dirRecvParams_t recvParams, nsid)` | |
| Description | Issue a Directive Receive command. | |
| Return Value | 0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | buf | The data buffer. |
| | **nDWNum** | **Number of dwords to transfer.**<br>**1 = 1 dword**<br>**2 = 2 dwords**<br>**N = N dwords** |
| | recvParams | Other CDW values of the Directive Receive command. The argument should be a nvme::dirRecvParams_t structure variable. This structure is defined by the extension module as the following:<br>`struct dirRecvParams_t {`<br>`  var CDW11_dirOp;`<br>`  var CDW11_dirType;`<br>`  var CDW11_dirSpec;`<br>`  var CDW12;`<br>`  var CDW13;`<br>`  clean() {`<br>`  CDW11_dirOp = 0;`<br>`  CDW11_dirType = 0;`<br>`  CDW11_dirSpec = 0;`<br>`  CDW12 = 0;`<br>`  CDW13 = 0;`<br>`  }`<br>`};` |
| | nsid (optional) | A namespace ID. |
| Note | | |
| See Also | | |

virtMgmt()

| | | |
|---|---|---|
| Prototype | `virtMgmt(&EC, cntlHdle, ACT, RT, CNTLID, NR)` | |
| Description | Isse a Virtualization Management command. | |
| Return Value | 0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | ACT | This field indicates the operation for the command to perform.<br>1 : Primary Controller Flexible Allocation<br>7 : Secondary Controller Offline<br>8 : Secondary Controller Assign<br>9 : Secondary Controller Online |
| | RT | Resource type.<br>0 : VQ Resources<br>1 : VI Resources. |
| | CNTLID | A controller Id; indicates the controller for which the controller resources are to be modified. |
| | NR | Number of controller resources to allocate or assign |
| Note | | |
| See Also | | |

```
virtualizationManagement(PriCntlHandle, ACT, RT, cntid, NR)
{
    var EC;
    var cmdHandle, cmdStatus, Status, CSStatus;
    var SCTDescp, SCDescp;

    cmdHandle = nvme::admc::virtMgmt(EC, PriCntlHandle, ACT, RT, cntid, NR);
    do
        cmdStatus = nvme::pspp(cmdHandle);
    while ((cmdStatus & nvme::cmdCplFlag) == 0);

    Status = (cmdStatus & nvme::cmdStsMask) >> nvme::cmdStsLowBit;
    CSStatus = (cmdStatus & nvme::cmdSpcStsMask) >> nvme::cmdSpcStsLowBit;

    if (Status != 0) {
        c::printf("virt. management command Status   = %04Xh\n", Status);
        c::printf("virt. management command CSStatus = %08Xh\n", CSStatus);
        GetNVMeCmdCplStatusDescription(Status, SCTDescp, SCDescp);
        c::printf("\tSCTDescp:%s, SCDescp:%s\n", SCTDescp, SCDescp);
    }
}

assignVQFlexResource(PriCntlHandle, vf_cntid, NR)
{
```

```
    virtualizationManagement(PriCntlHandle, 8, 0 /*VQ*/, vf_cntid, NR);
}

assignVIFlexResource(PriCntlHandle, vf_cntid, NR)
{
    virtualizationManagement(PriCntlHandle, 8, 1 /*VI*/, vf_cntid, NR);
}

setSecondaryControllerOnline(PriCntlHandle, vf_cntid)
{
    virtualizationManagement(PriCntlHandle, 9, 0 /*0*/, vf_cntid, 0);
}

main()
{
    struct VFInfo_t vfInfo;
    struct nvme::queueInitParams_t queueInitParams;

    ...
    // PriCntlHandle : controller handle of the Primary Controller
    // PriCNTLID : controller ID of the Primary Controller

    vfInfo.InvalidateAll();
    vfInfo.RetrieveInfo(0, PriCntlHandle, PriCNTLID);

    DevNo = nvme::getDevNo(EC, PriCntlHandle);
    nVFNum = nvme::enableVF(EC, PriCntlHandle, maxVFNum);

    for (n = 1; n <= nVFNum; ++n)
    {
        nsid = createNamespace(0, PriCntlHandle, params);
        VFHandle = nvme::getCntlHandle(EC, DevNo, n);

        SecCtrlrId = vfInfo.LookUpSecondaryControllerId(n);
        attachNamespace(0, PriCntlHandle, SecCtrlrId, nsid);

        assignVQFlexResource(PriCntlHandle, SecCtrlrId, 2);
        assignVIFlexResource(PriCntlHandle, SecCtrlrId, 2);
        nvme::flReset(EC, VFHandle);
        sleep(500);
        setSecondaryControllerOnline(PriCntlHandle, SecCtrlrId);

        queueInitParams.setDef();
        queueInitParams.nIOSQNum = 1;
        nvme::init(EC, VFHandle, "", queueInitParams);
    }
}
```

capMgmt()

| | | |
|---|---|---|
| Prototype | `capMgmt(&EC, cntlHdle, op, elemId, cap)` | |
| Description | Issue a Capacity Management command. | |
| Return Value | 0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | op | The Operation field of the CDW 10. |
| | elemId | The Element Identifier value of the CDW 10. A value specific to the op parameter. |
| | cap | A 64-bit value specifying the capacity in bytes. |
| Note | | |
| See Also | | |

| |
|---|
| |

lockdown()

| | | |
|---|---|---|
| Prototype | `lockdown(&EC, cntlHdle, scope, prohibit, IFC, OFI)`<br>`lockdown(&EC, cntlHdle, scope, prohibit, IFC, OFI, UUIDIdx)` | |
| Description | Issue a Lockdown command. | |
| Return Value | 0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | scope | A 4-bit value corresponds to the Scope field of CDW 10. This field specifies the contents of the OFI parameter.<br>0h : The OFI parameter is an Admin command opcode<br>2h : The OFI parameter is a Set Features Feature ID<br>3h : The OFI parameter is a Management Interface Command Set opcode<br>4h : The OFI parameter is a PCIe Command Set opcode |
| | prohibit | To specify whether to prohibit or allow the command opcode or Set Features Feature ID.<br>1 : prohibit the execution<br>0 : allow the execution. |
| | IFC | The Interface.<br>00b : Admin Submission Queue<br>01b : Admin Submission Queue and out-of-band on a Management Endpoint<br>10b : Out-of-band on a Management Endpoint |
| | OFI | The command opcode or Set Features Feature ID. |
| | UUIDIdx (optional) | A 7-bit value corresponds to UUID Index field of CDW14. |
| Note | | |
| See Also | | |

formatNVM()

| | | |
|---|---|---|
| Prototype | `formatNVM(&EC, cntlHdle, nsid, LBAF, MSEL, PI, PIL, SES)` | |
| Description | Issue a Format NVM command. | |
| Return Value | 0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | nsid | A namespace Id |
| | LBAF | LBA format; Specifies the LBA format to apply. |
| | MSEL | Metadata Settings.<br>1 : the metadata is transferred as part of an extended data LBA<br>0 : the metadata is transferred as part of a separate buffer |
| | PI | Protection Information; specifies whether end-to-end data protection is enabled and the type of protection information.<br>0 : Protection information is not enabled<br>1 : Protection information is enabled, Type 1<br>2 : Protection information is enabled, Type 2<br>3 : Protection information is enabled, Type 3 |
| | PIL | Indicate the protection information location.<br>1 : protection information is transferred as the first eight bytes of metadata if protection information is enabled<br>0 : protection information is transferred as the last eight bytes of metadata if protection information is enabled |
| | SES | Secure Erase Settings; specifies whether to perform a secure erase as part of the format and specifies the type of the secure erase operation.<br>0 : No secure erase operation is requested<br>1 : Request User Data Erase as the secure erase<br>2 : Request Cryptographic Erase as the secure erase |
| Note | | |
| See Also | | |

secSend()

| Prototype | `secSend(&EC, cntlHdle, buf, NSSF, ComID_SPSP, SecP, len)` | |
|---|---|---|
| Description | Issue a Security Send command. | |
| Return Value | 0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | buf | The buffer handle of the buffer with the sent content. |
| | NSSF | The NVMe Security Specific Field; corresponding to NSSF of the CDW10. |
| | ComID_SPSP | ComID or Secure Protocol Specific value; corresponding to SPSP0 and SPSP1 of CDW10. That is corresponding to CDW10 bit 8 ~ 23. |
| | SecP | Security Protocol; corresponding to Security Protocol (SP) of CDW10. |
| | len | Transfer length in bytes.<br>1 = 1 byte.<br>2 = 2 bytes.<br>N = N bytes. |
| Note | | |
| See Also | | |

```
    var sbuf, rbuf; //send_buffer, receive_buffer
    var bufLen;
    struct ComPacketEnc_t CPE;

    bufLen = 4096;
    sbuf = dev::ddbAlloc(bufLen);
    rbuf = dev::ddbAlloc(bufLen);

    dev::ddbFillPat8(sbuf, 0);
    CPE.initBuf(sbuf, bufLen);
    CPE.start(session.ComID, 0);
    CPE.newPacket(0, 0, 0);
    CPE.newDataSubPacket();
    TCG_Packet_BuildStartSessionMethod(CPE, session); // start session
    CPE.finishDataSubPacket();
    CPE.finishPacket();
    nComPacketSize = CPE.finish();

    cmdHandle = nvme::admc::secSend(EC,
                                    cntlHandle,
                                    sbuf,
                                    0,
                                    session.ComID,
                                    1,
                                    bufLen);
```

secRecv()

| | | |
|---|---|---|
| Prototype | `secRecv(&EC, cntlHdle, buf, NSSF, ComID_SPSP, SecP)`<br>`secRecv(&EC, cntlHdle, buf, NSSF, ComID_SPSP, SecP, len)` | |
| Description | Issue a Security Receive command. | |
| Return Value | 0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | cntlHdle | A controller handle. |
| | buf | The buffer handle of the buffer with the sent content. |
| | NSSF | The NVMe Security Specific Field; corresponding to NSSF of the CDW10. |
| | ComID_SPSP | ComID or Secure Protocol Specific value; corresponding to SPSP0 and SPSP1 of CDW10. That is corresponding to CDW10 bit 8 ~ 23. |
| | SecP | Security Protocol; corresponding to Security Protocol (SP) of CDW10. |
| | len<br>(optional) | Transfer length in bytes.<br>1 = 1 byte.<br>2 = 2 bytes.<br>N = N bytes. |
| Note | If len is not specified, size of the buffer would be used for the requirement of the command. | |
| See Also | | |

```
...

// Level 0 discovery
buf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 4096);
cmdHandle = nvme::admc::secRecv(EC, cntlHandle, buf, 0, 1, 1, 4096);
do
    stsVal = nvme::pspp(cmdHandle);
while (stsVal == 0);

...
```

sanitize()

| | | |
|---|---|---|
| Prototype | `sanitize(&EC, ch, act, AUSE, OWPASS, OIPBP, bNoDeAlloc)`<br>`sanitize(&EC, ch, act, AUSE, OWPASS, OIPBP, bNoDeAlloc, pat32)` | |
| Description | To issue a Sanatize command to the device. | |
| Return Value | 0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | ch | A controller handle. |
| | act | Sanitize Action; corresponding to Sanitize Action field of CDW10.<br>1 : Exit Failure Mode<br>2 : Start a Block Erase sanitize operation<br>3 : Start an Overwrite sanitize operation<br>4 : Start a Crypto Erase sanitize operation |
| | AUSE | Corresponding to Allow Unrestricted Sanitize Exit field of CDW10.<br>1 : sanitize operation is performed in unrestricted completion mode<br>0 : sanitize operation is performed in restricted completion mode |
| | OWPASS | Corresponding to Overwrite Pass Count of CDW10. |
| | OIPBP | Corresponding to Overwrite Invert Pattern Between Passes.<br>1 : Overwrite Pattern (pat32) shall be inverted between passes<br>0 : overwrite pattern (pat32) shall not be inverted between passes |
| | bNoDeAlloc | Corresponding to No-Deallocate After Sanitize.<br>0 : deallocate logic blocks after the sanitize operation<br>1 : deallocate logic blocks if No-Deallocate Inhibited bit in Identify Controller Data Structure is 1.<br>1 : no logic block shall be deallocated if No-Deallocate Inhibited bit in Identify Controller Data Structure is 1. |
| | pat32 (optional) | Corresponding to CDW11.<br>The overwrite pattern used in sanitize operation if the act argument is 3. If no argument is specified to this parameter and the act argument is 3, the 0xCDCDCDCD value will be filled into CDW11. |
| Note | | |
| See Also | | |

getLBAStatus()

| | | |
|---|---|---|
| Prototype | `getLBAStatus(&EC, ch, buf, slba, nMNDW, nRL, act)`<br>`getLBAStatus(&EC, ch, buf, slba, nMNDW, nRL, act, nsid)` | |
| Description | | |
| Return Value | 0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | ch | A controller handle. |
| | buf | A buffer handle for data transfer. |
| | slab | A 64-bit starting LBA address to get the status. |
| | nMNDW | Specify the maximum number of dwords to return |
| | nRL | Range Length. |
| | act | Action Type. Corresponding to Action Type field of CDW 13.<br>10h : Perform a scan and return Untracked LBAs and Tracked LBAs in the specified range<br>11h : Return Tracked LBAs in the specified range |
| | nsid (optional) | An namespace Id. |
| Note | A namespace handle can be set as the default accessed one of an SQ by nvme::setSQAttr(). Without passing the nsid argument to this function, the nsid of the default namespace of the IO SQ will be used. | |
| See Also | | |

passThru()

<table>
<tr><td>Prototype</td><td colspan="2">passThru(&EC,<br>cntlHdle,<br>dataBuf,<br>dataLen_inBytes,<br>metaBuf,<br>nvme:: passThruParams_t passThruParams)</td></tr>
<tr><td>Description</td><td colspan="2">Except blocked op codes, this function assists to issue an arbitrary op-code command through the controller’s Admin Submission Queue.</td></tr>
<tr><td>Return Value</td><td colspan="2">0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully</td></tr>
<tr><td rowspan="6">Parameter</td><td>EC</td><td>This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0.</td></tr>
<tr><td>cntlHdle</td><td>A controller handle.</td></tr>
<tr><td>dataBuf</td><td>The data buffer handle; 0 shall be specified if data buffer is not needed.</td></tr>
<tr><td>dataLen_inBytes</td><td>Length of data in bytes.<br>1 = 1 Byte,<br>2 = 2 Bytes,<br>N = N Bytes</td></tr>
<tr><td>metaBuf</td><td>The metadata buffer handle; 0 shall be specified if metadata buffer is not needed.</td></tr>
<tr><td>passThruParams</td><td>Parameters for the command; should be a nvme::passThruParams_t structure. The structure is defined as the following. Value of the CSI (Command Set Identifier) field is ignored by this function.<br>struct passThruParams_t {<br>var CSI;<br>var CDW0_FUSE, CDW0_OPC;<br>var NamespaceId;<br>var CDW2,CDW3,CDW10,CDW11,CDW12,CDW13,CDW14,CDW15;<br>clean()<br>{<br>CSI = 0;<br>CDW0_FUSE = 0;<br>CDW0_OPC = 0;<br>NamespaceId = 0;<br>CDW2 = 0;<br>CDW3 = 0;<br>CDW10 = 0;<br>CDW11 = 0;<br>CDW12 = 0;<br>CDW13 = 0;<br>CDW14 = 0;<br>CDW15 = 0;<br>}<br>};</td></tr>
<tr><td>Note</td><td colspan="2">This function blocks sending commands with below OP code:</td></tr>
</table>

| | |
|---|---|
| | 00h (Delete I/O Submission Queue)<br>01h (Create I/O Submission Queue)<br>04h (Delete I/O Completion Queue)<br>05h (Create I/O Submission Queue)<br>0Ch (Asynchronous Event Request)<br>0Dh (Namespace Management)<br>15h (Namespace Attachment)<br>80h (Format NVM) |
| See Also | |

```
struct nvme::passThruParams_t getLogPage;

...
getLogPage.clean();
getLogPage.CDW0_OPC = 2;
getLogPage.NamespaceId = 0xFFFFFFFF;
getLogPage.CDW10 = 2 | ((512/4) << 16);  //page 2; length is 512 bytes

dbuf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 512);

cmdHdle = nvme::admc::passThru(EC, cntlHandle, dbuf, 512, 0, getLogPage);
do
    stsVal = nvme::pspp(cmdHdle);
while (stsVal == 0);
```

## Extension functions in namespace nvme::ioc

flush()

| | | |
|---|---|---|
| Prototype | `flush(&EC, iosqHdle)`<br>`flush(&EC, iosqHdle, nsid)` | |
| Description | Issue a Flush command through an IO SQ. | |
| Return Value | 0 : failed<br>1 : if the polling mode attribute of the IO SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | iosqHdle | An IO SQ handle. |
| | nsid (optional) | A namespace Id. |
| Note | A namespace handle can be set as the default accessed one of an SQ by nvme::setSQAttr(). Without passing the nsid argument to this function, the nsid of the default namespace of the IO SQ will be used. | |
| See Also | setSQAttr() | |

```
lastFlushTime = c::time();
while (1)
{
    ...
    lba = c::rand() % module;
    cmds[idx] = nvme::ioc::write(EC, iosq, dbuf, lba, n4k, mbuf);
    ...

    if (c::time() > lastFlushTime)
    {
        // flush per second
        flushCmdHandle = nvme::flush(EC, iosq);
        ...
        lastFlushTime = c::time();
    }
}
```

write()

<table>
<tr><td>Prototype</td><td colspan="2">write(&EC, qh, dbuf, slba, nlb)<br>write(&EC, qh, dbuf, slba, nlb, mbuf)<br>write(&EC, qh, dbuf, slba, nlb, mbuf, nsid)<br>write(&EC, qh, dbuf, slba, nlb, mbuf, nsid,<br>nvme::writeOptParams_t optParams)</td></tr>
<tr><td>Description</td><td colspan="2">Issue a Write command through an IO SQ.</td></tr>
<tr><td>Return Value</td><td colspan="2">0 : failed<br>1 : if the polling mode attribute of the IO SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully</td></tr>
<tr><td rowspan="8">Parameter</td><td>EC</td><td>This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0.</td></tr>
<tr><td>qh</td><td>An IO SQ handle.</td></tr>
<tr><td>dbuf</td><td>A buffer handle where the buffer contains LBA datum.</td></tr>
<tr><td>slba</td><td>The starting LBA.</td></tr>
<tr><td>nlb</td><td>Number of logic blocks.<br>1 = 1 logic block.<br>2 = 2 logic blocks.<br>N = N logic blocks.</td></tr>
<tr><td>mbuf<br>(optional)</td><td>A buffer handle where the buffer contains metadata. A meta deta buffer is required if each LBA contains metadata and metadata is set to be transferred in a separate buffer. Value 0 is specified while each LBA has no metadata.</td></tr>
<tr><td>nsid<br>(optional)</td><td>A namespace Id.</td></tr>
<tr><td>optParams<br>(optional)</td><td>For optional parameters and it should be a nvme::writeOptParams_t structure. The structure is defined as the following. Each structure field corresponds to the same name field of Write command entry.<br>struct writeOptParams_t<br>{<br>var CDW0_FUSE;<br>var CDW2;<br>var CDW3;<br>var CDW12_LR;<br>var CDW12_FUA;<br>var CDW12_PRINFO;<br>var CDW12_STC;<br>var CDW12_DTYPE;<br>var CDW13_DSPEC;<br>var CDW13_DSM;<br>var CDW14;<br>var CDW15_LBATM;<br>var CDW15_LBAT;<br>clean()<br>{<br>CDW0_FUSE = 0;</td></tr>
</table>

| | | |
|---|---|---|
| | | CDW2 = 0;<br>CDW3 = 0;<br>CDW12_LR = 0;<br>CDW12_FUA = 0;<br>CDW12_PRINFO = 0;<br>CDW12_STC = 0;<br>CDW12_DTYPE = 0;<br>CDW13_DSPEC = 0;<br>CDW13_DSM = 0;<br>CDW14 = 0;<br>CDW15_LBATM = 0;<br>CDW15_LBAT = 0;<br>}<br>}; |
| Note | A namespace handle can be set as the default accessed one of an SQ by nvme::setSQAttr(). Without passing the nsid argument to this function, the nsid of the default namespace of the IO SQ will be used.<br><br>This function automatically generate a 64-bit checksum value for each LBA data going to be written if the "AutoGenChecksumOnLBAData" attribute value of namespace is set as 1. The 64-bit checksum value is calculated by doing XOR operations on every 64-bit value from byte 8 to the end of the LBA data. The 64-bit checksum value is put at byte 0 of the LBA data. If metadata is enabled and is transferred immediately after each LBA data, metadata will be included in the checksum calculation.<br><br>When this function is called to issue a write command, the programmer should note that (1) whether metadata is contained and (2) the metadata is transferred immediately after the LBA data or separately. It is a good habit to write the script with metadata taken into consideration. | |
| See Also | nvme::setSQAttr(), nvme::ddbCalculateChecksum64() | |

```
struct cntlInfo_t cntl;
struct namespace_t ns;
var nlb, n4k;

...
cntl.Invalidate();
cntl.RetrieveInfo(0, cntlHandle);
...
cntl.CopyTheNthNamespace(0, ns);

nTotalLBANum = ns.GetCapacity_lba();
nLBADataSz = ns.GetLBADataSize_byte();
nLBAMetaSz = ns.GetLBAMetadataSize_byte();
nExtLBADataSz = ns.GetExtLBADataSize_byte();
n4K = 4096 / nLBADataSz;

dbuf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, n4K * nExtLBADataSz);
nvme::ddbFillPatRand(EC, dbuf);
if (nLBAMetaSz > 0 && nExtLBADataSz == nLBADataSz)
{
    // a separate buffer for metadata
    mbuf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, n4k * nLBAMetaSz);
```

```
        nvme::ddbFillPatRand(EC, mbuf);
    }
    else
        mbuf = 0;

    // generate "4KB random writes" workload
    iosq = nvme::getSQHandle(EC, cntlHandle, 1);
    module = nTotalLBANum - n4k + 1;
    while (1)
    {
        ...
        lba = c::rand() % module;
        cmds[idx] = nvme::ioc::write(EC, iosq, dbuf, lba, n4k, mbuf);
        ...
    }
```

read()

<table>
<tr><td>Prototype</td><td colspan="2">read(&EC, qh, dbuf, slba, nlb)<br>read(&EC, qh, dbuf, slba, nlb, mbuf)<br>read(&EC, qh, dbuf, slba, nlb, mbuf, nsid)<br>read(&EC, qh, dbuf, slba, nlb, mbuf, nsid, struct nvme::readOptParams_t optParams)</td></tr>
<tr><td>Description</td><td colspan="2">Isse a Read command through an IO SQ.</td></tr>
<tr><td>Return Value</td><td colspan="2">0 : failed<br>1 : if the polling mode attribute of the IO SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully</td></tr>
<tr><td rowspan="8">Parameter</td><td>EC</td><td>This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0.</td></tr>
<tr><td>qh</td><td>An IO SQ handle.</td></tr>
<tr><td>dbuf</td><td>A buffer handle for the LBA data read.</td></tr>
<tr><td>slba</td><td>The starting LBA.</td></tr>
<tr><td>nlb</td><td>Number of logic blocks.<br>1 = 1 logic block.<br>2 = 2 logic blocks.<br>N = N logic blocks.</td></tr>
<tr><td>mbuf (optional)</td><td>A buffer handle for read metadata. A meta deta buffer is required if each LBA contains metadata and metadata is set to be transferred in a separate buffer. Value 0 is specified while each LBA has no metadata.</td></tr>
<tr><td>nsid (optional)</td><td>A namespace Id.</td></tr>
<tr><td>optParams (optional)</td><td>For optional parameters and it should be a <code>nvme::readOptParams_t structure</code>. The structure is defined as the following. Each structure field corresponds to the same name field of Read command entry.<br><pre>struct readOptParams_t
{
  var CDW0_FUSE;
  var CDW2;
  var CDW3;
  var CDW12_LR;
  var CDW12_FUA;
  var CDW12_PRINFO;
  var CDW12_STC;
  var CDW13_DSM;
  var CDW14;
  var CDW15_ELBATM;
  var CDW15_ELBAT;
  clean()
  {
    CDW0_FUSE = 0;
    CDW2 = 0;
    CDW3 = 0;
    CDW12_LR = 0;
    CDW12_FUA = 0;</pre></td></tr>
</table>

| | | |
|---|---|---|
| | | CDW12_PRINFO = 0;<br>CDW12_STC = 0;<br>CDW13_DSM = 0;<br>CDW14 = 0;<br>CDW15_ELBATM = 0;<br>CDW15_ELBAT = 0;<br>}<br>}; |
| Note | A namespace handle can be set as the default accessed one of an SQ by nvme::setSQAttr(). Without passing the nsid argument to this function, the nsid of the default namespace of the IO SQ will be used.<br><br>When this function is called to issue a read command, the programmer should note that (1) whether metadata is contained and (2) the metadata is transferred immediately after the LBA data or separately. It is a good habit to write the script with metadata taken into consideration. | |
| See Also | | |

```
struct cntlInfo_t cntl;
struct namespace_t ns;
var nlb, n4k;

...
cntl.Invalidate();
cntl.RetrieveInfo(0, cntlHandle);
...
cntl.CopyTheNthNamespace(0, ns);

nTotalLBANum = ns.GetCapacity_lba();
nLBADataSz = ns.GetLBADataSize_byte();
nLBAMetaSz = ns.GetLBAMetadataSize_byte();
nExtLBADataSz = ns.GetExtLBADataSize_byte();
n4K = 4096 / nLBADataSz;

dbuf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, n4K * nExtLBADataSz);
if (nLBAMetaSz > 0 && nExtLBADataSz == nLBADataSz)
{
    // a separate buffer for metadata
    mbuf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, n4k * nLBAMetaSz);
}
else
    mbuf = 0;

// generate "4KB random reads" workload
iosq = nvme::getSQHandle(EC, cntlHandle, 1);
module = nTotalLBANum – n4k + 1;
while (1)
{
    ...
    lba = c::rand() % module;
    cmds[idx] = nvme::ioc::read(EC, iosq, dbuf, lba, n4k, mbuf);
    ...
}
```

writeUNC()

| | | |
|---|---|---|
| Prototype | `writeUNC(&EC, iosqHdle, slba, nlb)`<br>`writeUNC(&EC, iosqHdle, slba, nlb, nsid)` | |
| Description | Issue a Write Uncorrectable command through an IO SQ. | |
| Return Value | 0 : failed<br>1 : if the polling mode attribute of the IO SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | iosqHdle | An IO SQ handle. |
| | slba | The starting LBA. |
| | **nlb** | **Number of logic blocks.**<br>**1 = 1 logic block.**<br>**2 = 2 logic blocks.**<br>**N = N logic blocks.** |
| | nsid (optional) | A namespace Id. |
| Note | A namespace handle can be set as the default accessed one of an SQ by nvme::setSQAttr(). Without passing the nsid argument to this function, the nsid of the default namespace of the IO SQ will be used. | |
| See Also | | |

```
showStatus(stsVal)
{
    var Status, SCTDescp, SCDescp, nCmdSpec;

    Status = (stsVal & nvme::cmdStsMask) >> nvme::cmdStsLowBit;
    if (Status) {
        c::printf("  stsVal = %Xh, Status = %04Xh, ", stsVal, Status);
        GetNVMeCmdCplStatusDescription(Status, SCTDescp, SCDescp);
        c::printf("  SCT: %s, SC: %s\n", SCTDescp, SCDescp);
    }

    nCmdSpec = (stsVal & nvme::cmdSpcStsMask) >> nvme::cmdSpcStsLowBit;
    if (nCmdSpec)
        c::printf("  Command specific status %08Xh\n", nCmdSpec);

    if (stsVal & nvme::cmdToFlag)
        c::printf("  Comand timeout\n");
}

main()
{
    sqh = nvme::getSQHandle(EC, cntlHandle, 1);
    cmdHandle = nvme::ioc::writeUNC(EC, sqh, 0, 3);
    ... // poll command complete
    cmdHandle = nvme::ioc::write(EC, sqh, buf, 1, 1);
```

```
    ... // poll command complete
    for (lba = 0; lba < 3; ++lba)
    {
        cmdHandle = nvme::ioc::read(EC, sqh, buf, lba, 1);
        do {stsVal = nvme::pspp(cmdHandle);} while (stsVal == 0);
        if (stsVal == nvme::cmdCplFlag)
            c::printf("LBA-%d Status = OK\n", lba);
        else
        {
            c::printf("LBA-%d Status:\n", lba);
            showStatus(stsVal);
        }
    }
}
```

compare()

<table>
<tr><td>Prototype</td><td colspan="2">compare(&EC, qh, dbuf, slba, nlb)<br>compare(&EC, qh, dbuf, slba, nlb, mbuf)<br>compare(&EC, qh, dbuf, slba, nlb, mbuf, nsid)<br>compare(&EC, qh, dbuf, slba, nlb, mbuf, nsid, struct nvme::compareOptParams_t optParams)</td></tr>
<tr><td>Description</td><td colspan="2">Issue a Compare command through an IO SQ.</td></tr>
<tr><td>Return Value</td><td colspan="2">0 : failed<br>1 : if the polling mode attribute of the IO SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully</td></tr>
<tr><td rowspan="8">Parameter</td><td>EC</td><td>This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0.</td></tr>
<tr><td>qh</td><td>An IO SQ handle.</td></tr>
<tr><td>dbuf</td><td>A buffer handle where the buffer contains LBA datum.</td></tr>
<tr><td>slba</td><td>The starting LBA.</td></tr>
<tr><td>nlb</td><td>Number of logic blocks.<br>1 = 1 logic block.<br>2 = 2 logic blocks.<br>N = N logic blocks.</td></tr>
<tr><td>mbuf (optional)</td><td>A buffer handle where the buffer contains metadata. A meta deta buffer is required if each LBA contains metadata and metadata is set to be transferred in a separate buffer. Value 0 is specified while each LBA has no metadata.</td></tr>
<tr><td>nsid (optional)</td><td>A namespace Id.</td></tr>
<tr><td>optParams (optional)</td><td>For optional parameters and it should be a nvme::compareOptParams_t structure. The structure is defined as the following. Each structure field corresponds to the same name field of Compare command entry.<br>struct compareOptParams_t<br>{<br>var CDW0_FUSE;<br>var CDW2;<br>var CDW3;<br>var CDW12_LR;<br>var CDW12_FUA;<br>var CDW12_PRINFO;<br>var CDW12_STC;<br>var CDW14;<br>var CDW15_ELBATM;<br>var CDW15_ELBAT;<br>clean()<br>{<br>CDW0_FUSE = 0;<br>CDW2 = 0;<br>CDW3 = 0;<br>CDW12_LR = 0;</td></tr>
</table>

| | | |
|---|---|---|
| | | CDW12_FUA = 0;<br>CDW12_PRINFO = 0;<br>CDW12_STC = 0;<br>CDW14 = 0;<br>CDW15_ELBATM = 0;<br>CDW15_ELBAT = 0;<br>}<br>}; |
| Note | A namespace handle can be set as the default accessed one of an SQ by nvme::setSQAttr(). Without passing the nsid argument to this function, the nsid of the default namespace of the IO SQ will be used. | |
| See Also | | |

```
showStatus(stsVal)
{
    var Status, SCTDescp, SCDescp, nCmdSpec;

    Status = (stsVal & nvme::cmdStsMask) >> nvme::cmdStsLowBit;
    if (Status) {
        c::printf("  stsVal = %Xh, Status = %04Xh, ", stsVal, Status);
        GetNVMeCmdCplStatusDescription(Status, SCTDescp, SCDescp);
        c::printf("  SCT: %s, SC: %s\n", SCTDescp, SCDescp);
    }

    nCmdSpec = (stsVal & nvme::cmdSpcStsMask) >> nvme::cmdSpcStsLowBit;
    if (nCmdSpec)
        c::printf("  Command specific status %08Xh\n", nCmdSpec);

    if (stsVal & nvme::cmdToFlag)
        c::printf("  Comand timeout\n");
}

main()
{
    ...
    // write random values into LBA 100
    qh = nvme::getSQHandle(EC, cntlHandle, 1);
    buf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 512);
    nvme::ddbFillPatRand(EC, buf);
    cmdHandle = nvme::ioc::write(EC, qh, buf, 100, 1);
    ...

    // clean the buffer
    nvme::ddbFillPat8(EC, buf, 0);

    // read LBA 100
    cmdHandle = nvme::ioc::read(EC, qh, buf, 100, 1);
    ...

    // compare LBA 100
    cmdHandle = nvme::ioc::compare(EC, qh, buf, 100, 1);
    do
        stsVal = nvme::pspp(cmdHandle);
    while (stsVal == 0);
```

```
    if (stsVal == nvme::cmdCplFlag)
        c::printf("PASS\n");
    else
    {
        c::printf("FAIL\n");
        showStatus(stsVal);
    }
}
```

write0()

<table>
<tr><td>Prototype</td><td colspan="2">write0(&EC, iosqHdle, slba, nlb)<br>write0(&EC, iosqHdle, slba, nlb, nsid)<br>write0(&EC, iosqHdle, slba, nlb, nsid, struct nvme::write0OptParams_t optParam</td></tr>
<tr><td>Description</td><td colspan="2">Issue a Write Zeroes command.</td></tr>
<tr><td>Return Value</td><td colspan="2">0 : failed<br>1 : if the polling mode attribute of the IO SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully</td></tr>
<tr><td rowspan="6">Parameter</td><td>EC</td><td>This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0.</td></tr>
<tr><td>iosqHdle</td><td>An IO SQ handle.</td></tr>
<tr><td>slba</td><td>The starting LBA.</td></tr>
<tr><td>nlb</td><td>Number of logic blocks.<br>1 = 1 logic block.<br>2 = 2 logic blocks.<br>N = N logic blocks.</td></tr>
<tr><td>nsid<br>(optional)</td><td>A namespace Id.</td></tr>
<tr><td>optParam<br>(optional)</td><td>For specifying optional parameters and it should be a nvme::write0OptParams_t structure. The structure is defined as the following. Each structure field corresponds to the same name field of Write Zeroes command entry.<br>struct write0OptParams_t<br>{<br>var CDW0_FUSE;<br>var CDW2;<br>var CDW3;<br>var CDW12_LR;<br>var CDW12_FUA;<br>var CDW12_PRINFO;<br>var CDW12_DEAC;<br>var CDW12_STC;<br>var CDW14;<br>var CDW15_LBATM;<br>var CDW15_LBAT;<br>clean()<br>{<br>CDW0_FUSE = 0;<br>CDW2 = 0;<br>CDW3 = 0;<br>CDW12_LR = 0;<br>CDW12_FUA = 0;<br>CDW12_PRINFO = 0;<br>CDW12_DEAC = 0;<br>CDW12_STC = 0;<br>CDW14 = 0;<br>CDW15_LBATM = 0;</td></tr>
</table>

| | | |
|---|---|---|
| | | CDW15_LBAT = 0;<br>}<br>}; |
| Note | A namespace handle can be set as the default accessed one of an SQ by nvme::setSQAttr(). Without passing the nsid argument to this function, the nsid of the default namespace of the IO SQ will be used. | |
| See Also | | |

```
struct nvme::write0OptParams_t optParams;

nsid = 0;
nsid = nvme::getNextNsId(EC, cntlHandle, nsid);

sqid = 0;
sqid = nvme::getNextIOSQId(EC, cntlHandle, sqid);
sqHandle = nvme::getSQHandle(EC, cntlHandle, sqid);

optParams.clean();
optParams.FUA = 1;
optParams.DEAC = 1;

cmdHandle = nvme::ioc::write0(EC, sqHandle, 0, 64, nsid, optParams);
...
```

dsm()

| Prototype | `dsm(&EC, iosqHdle, buf, rangenum, attr_cdw11)`<br>`dsm(&EC, iosqHdle, buf, rangenum, attr_cdw11, nsid)` | |
|---|---|---|
| Description | Issue a Dataset Management command through an IO SQ. | |
| Return Value | 0 : failed<br>1 : if the polling mode attribute of the IO SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | iosqHdle | An IO SQ handle. |
| | buf | A buffer handle; contains range information. |
| | **rangenum** | **Number of ranges.**<br>**1 = 1 range**<br>**2 = 2 ranges**<br>**N = N ranges** |
| | attr_cdw11 | Attribute value. The value for CDW11. |
| | nsid (optional) | A namespace Id. |
| Note | A namespace handle can be set as the default accessed one of an SQ by nvme::setSQAttr(). Without passing the nsid argument to this function, the nsid of the default namespace of the IO SQ will be used. | |
| See Also | | |

```
nAccTrimSize = 0;
while (nAccTrimSize < nTotalLBANum)
{
    nThisTrimSize = 0x40000000;
    nvme::ddbFillPat8(EC, buf, 0);
    for (nNRIdx = 0; nNRIdx < 256 && nAccTrimSize < nTotalLBANum;
         nNRIdx++, nAccTrimSize += nThisTrimSize)
    {   // Each range uses 16 bytes
        //      Byte03:Byte00 = Context Attributes
        //      Byte07:Byte04 = Length in logical blocks (1-based)
        //      Byte15:Byte08 = Starting LBA
        if (nThisTrimSize + nAccTrimSize >= nTotalLBANum)
            nThisTrimSize = nTotalLBANum - nAccTrimSize;
        nvme::ddbWriteInt32(EC, buf, nNRIdx * 16 + 4, 1, nThisTrimSize);
        nvme::ddbWriteInt64(EC, buf, nNRIdx * 16 + 8, 1, nAccTrimSize);
    }

    cmdHandle = nvme::ioc::dsm(EC, SQ1Handle, buf, nNRIdx, 1 << 2);
    ...
}
```

verify()

| | | |
|---|---|---|
| Prototype | `verify(&EC, iosqHdle, slba, nlb)`<br>`verify(&EC, iosqHdle, slba, nlb, nsid)`<br>`verify(&EC, iosqHdle, slba, nlb, nsid, struct nvme::verifyOptParams_t optParam)` | |
| Description | Issue a Verify command through an IO SQ. | |
| Return Value | 0 : failed<br>1 : if the polling mode attribute of the IO SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | iosqHdle | An IO SQ handle. |
| | slba | The starting LBA. |
| | **nlb** | **Number of logic blocks.**<br>**1 = 1 logic block.**<br>**2 = 2 logic blocks.**<br>**N = N logic blocks.** |
| | nsid (optional) | A namespace Id. |
| | optParam (optional) | For specifying optional parameters and it should be a `nvme::verifyOptParams_t structure`. The structure is defined as the following. Each structure field corresponds to the same name field of Verify command entry.<br>`struct verifyOptParams_t`<br>`{`<br>`  var CDW0_FUSE;`<br>`  var CDW2;`<br>`  var CDW3;`<br>`  var CDW12_LR;`<br>`  var CDW12_FUA;`<br>`  var CDW12_PRINFO;`<br>`  var CDW12_STC;`<br>`  var CDW14;`<br>`  var CDW15_ELBATM;`<br>`  var CDW15_ELBAT;`<br>`  clean()`<br>`  {`<br>`  CDW0_FUSE = 0;`<br>`  CDW2 = 0;`<br>`  CDW3 = 0;`<br>`  CDW12_LR = 0;`<br>`  CDW12_FUA = 0;`<br>`  CDW12_PRINFO = 0;`<br>`  CDW12_STC = 0;`<br>`  CDW14 = 0;`<br>`  CDW15_ELBATM = 0;`<br>`  CDW15_ELBAT = 0;`<br>`  }` |

| | | |
|---|---|---|
| | | }; |
| Note | A namespace handle can be set as the default accessed one of an SQ by nvme::setSQAttr(). Without passing the nsid argument to this function, the nsid of the default namespace of the IO SQ will be used. | |
| See Also | | |

```
struct nvme::verifyOptParams_t optParams;

nsid = 0;
nsid = nvme::getNextNsId(EC, cntlHandle, nsid);

sqid = 0;
sqid = nvme::getNextIOSQId(EC, cntlHandle, sqid);
sqHandle = nvme::getSQHandle(EC, cntlHandle, sqid);

optParams.clean();
optParams.FUA = 1;
cmdHandle = nvme::ioc::verify(EC, sqHandle, 0, 64, nsid, optParams);
```

resvReg()

| | | |
|---|---|---|
| Prototype | `resvReg(&EC, iosqHdle, buf, act, ignrk, CPTPL)`<br>`resvReg(&EC, iosqHdle, buf, act, ignrk, CPTPL, nsid)` | |
| Description | Issue a Reservation Register command through an IO SQ. | |
| Return Value | 0 : failed<br>1 : if the polling mode attribute of the IO SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | iosqHdle | An IO SQ handle. |
| | buf | The data buffer. |
| | act | A 3-bit value standing for the Reservation Register Action parameter.<br>000b = Register Reservation Key<br>001b = Unregister Reservation Key<br>010b = Replace Reservation Key |
| | ignrk | A 1-bit value standing for the Ignore Existing Key parameter. |
| | CPTPL | A 2-bit value standing for the Change Persist Through Power Loss State (CPTPL) parameter. |
| | nsid (optional) | A namespace Id. |
| Note | A namespace handle can be set as the default accessed one of an SQ by nvme::setSQAttr(). Without passing the nsid argument to this function, the nsid of the default namespace of the IO SQ will be used. | |
| See Also | | |

resvRep()

| | | |
|---|---|---|
| Prototype | `resvRep(&EC, iosqHdle, buf, nDWNum, bExtDS)`<br>`resvRep(&EC, iosqHdle, buf, nDWNum, bExtDS, nsid)` | |
| Description | Issue a Reservation Report command through an IO SQ. | |
| Return Value | 0 : failed<br>1 : if the polling mode attribute of the IO SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | iosqHdle | An IO SQ handle. |
| | buf | The data buffer. |
| | **nDWNum** | **To specify the number of dwords of the Reservation Status data structure to transfer.**<br>**1 = 1 dword**<br>**2 = 2 dwords**<br>**N = N dwords** |
| | bExtDS | Specify the returned data structure.<br>1 = to return the Reservation Status Extended Data Structure<br>0 = to return the Reservation Status Data Structure |
| | nsid (optional) | A namespace Id. |
| Note | A namespace handle can be set as the default accessed one of an SQ by nvme::setSQAttr(). Without passing the nsid argument to this function, the nsid of the default namespace of the IO SQ will be used. | |
| See Also | | |

resvAcq()

| | | |
|---|---|---|
| Prototype | `resvAcq(&EC, iosqHdle, buf, act, ignrk, rtype)`<br>`resvAcq(&EC, iosqHdle, buf, act, ignrk, rtype, nsid)` | |
| Description | Issue a Reservation Acquire command through an IO SQ. | |
| Return Value | 0 : failed<br>1 : if the polling mode attribute of the IO SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | iosqHdle | An IO SQ handle. |
| | buf | The data buffer. |
| | act | A 3-bit value standing for the Reservation Acquire Action parameter.<br>000b = Acquire<br>001b = Preempt<br>010b = Preempt and Abort |
| | ignrk | A 1-bit value corresponds to bit 3 of the CDW 10. |
| | rtype | A 8-bit value corresponds to the Reservation Type field of CDW10. |
| | nsid (optional) | A namespace Id. |
| Note | A namespace handle can be set as the default accessed one of an SQ by nvme::setSQAttr(). Without passing the nsid argument to this function, the nsid of the default namespace of the IO SQ will be used. | |
| See Also | | |

resvRel()

| | | |
|---|---|---|
| Prototype | `resvRel(&EC, iosqHdle, buf, act, ignrk, rtype)`<br>`resvRel(&EC, iosqHdle, buf, act, ignrk, rtype, nsid)` | |
| Description | Issue a Reservation Acquire command through an IO SQ. | |
| Return Value | 0 : failed<br>1 : if the polling mode attribute of the IO SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | iosqHdle | An IO SQ handle. |
| | buf | The data buffer. |
| | act | A 3-bit value standing for the Reservation Release Action parameter.<br>000b = Release<br>001b = Clear |
| | ignrk | A 1-bit value corresponds to bit 3 of the CDW 10. |
| | rtype | A 8-bit value corresponds to the Reservation Type field of CDW10. |
| | nsid (optional) | A namespace Id. |
| Note | A namespace handle can be set as the default accessed one of an SQ by nvme::setSQAttr(). Without passing the nsid argument to this function, the nsid of the default namespace of the IO SQ will be used. | |
| See Also | | |

copy()

| | | |
|---|---|---|
| Prototype | `copy(&EC, iosqHdle, buf, sdlba, nr, dfmt)`<br>`copy(&EC, iosqHdle, buf, sdlba, nr, dfmt, nsid)`<br>`copy(&EC, iosqHdle, buf, sdlba, nr, dfmt, nsid, struct nvme::copyOptParams_t optParams)` | |
| Description | Issue a Copy command through an IO SQ. | |
| Return Value | 0 : failed<br>1 : if the polling mode attribute of the IO SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | iosqHdle | An IO SQ handle. |
| | buf | |
| | sdlba | |
| | nr | |
| | dfmt | |
| | nsid (optional) | A namespace Id. |
| | optParams (optional) | |
| Note | A namespace handle can be set as the default accessed one of an SQ by nvme::setSQAttr(). Without passing the nsid argument to this function, the nsid of the default namespace of the IO SQ will be used. | |
| See Also | | |

store()

| | | |
|---|---|---|
| Prototype | `store(&EC, iosqHdle, buf, valsz, struct nvme::kvKey_t key, storeOpt)`<br>`store(&EC, iosqHdle, buf, valsz, struct nvme::kvKey_t key, storeOpt, nsid)` | |
| Description | Issue a Store command through an IO SQ. | |
| Return Value | 0 : failed<br>1 : if the polling mode attribute of the IO SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | iosqHdle | An IO SQ handle. |
| | buf | The data buffer storing the KV value. |
| | valsz | The KV value size in bytes. |
| | key | The KV key which is a structure parameter. The structure is defined as the following. The KeyLen specifies the length of the key in bytes. Each of CDW2_Key, CDW3_Key, CDW14_Key, and CDW15_Key is a 32-bit value of the key and corresponds to Command Dword 2, 3, 14, and 15 respectively.<br>`struct kvKey_t {`<br>`  var KeyLen;`<br>`  var CDW2_Key;`<br>`  var CDW3_Key;`<br>`  var CDW14_Key;`<br>`  var CDW15_Key;`<br>`  clean()`<br>`  {`<br>`  KeyLen = 0;`<br>`  CDW2_Key = 0;`<br>`  CDW3_Key = 0;`<br>`  CDW14_Key = 0;`<br>`  CDW15_Key = 0;`<br>`  }`<br>`};` |
| | storeOpt | A 8-bit value specifying the store option. |
| | nsid (optional) | A namespace Id. |
| Note | A namespace handle can be set as the default accessed one of an SQ by nvme::setSQAttr(). Without passing the nsid argument to this function, the nsid of the default namespace of the IO SQ will be used. | |
| See Also | | |

retrieve()

<table>
<tr><td>Prototype</td><td colspan="2">retrieve(&EC, iosqHdle, buf, bufsz, struct nvme::kvKey_t key, retrOpt)<br>retrieve(&EC, iosqHdle, buf, bufsz, struct nvme::kvKey_t key, retrOpt, nsid)</td></tr>
<tr><td>Description</td><td colspan="2">Issue a Retrieve command through an IO SQ.</td></tr>
<tr><td>Return Value</td><td colspan="2">0 : failed<br>1 : if the polling mode attribute of the IO SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully</td></tr>
<tr><td rowspan="7">Parameter</td><td>EC</td><td>This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0.</td></tr>
<tr><td>iosqHdle</td><td>An IO SQ handle.</td></tr>
<tr><td>buf</td><td>The host buffer to which the data is transferred.</td></tr>
<tr><td>bufsz</td><td>The size of the buffer.</td></tr>
<tr><td>key</td><td>The KV key which is a structure parameter. The structure is defined as the following. The KeyLen specifies the length of the key in bytes. Each of CDW2_Key, CDW3_Key, CDW14_Key, and CDW15_Key is a 32-bit value of the key and corresponds to Command Dword 2, 3, 14, and 15 respectively.<br><pre>struct kvKey_t {
  var KeyLen;
  var CDW2_Key;
  var CDW3_Key;
  var CDW14_Key;
  var CDW15_Key;
  clean()
  {
    KeyLen = 0;
    CDW2_Key = 0;
    CDW3_Key = 0;
    CDW14_Key = 0;
    CDW15_Key = 0;
  }
};</pre></td></tr>
<tr><td>retrOpt</td><td>A 8-bit value specifying the retrieve option.</td></tr>
<tr><td>nsid (optional)</td><td>A namespace Id.</td></tr>
<tr><td>Note</td><td colspan="2">A namespace handle can be set as the default accessed one of an SQ by nvme::setSQAttr(). Without passing the nsid argument to this function, the nsid of the default namespace of the IO SQ will be used.</td></tr>
<tr><td>See Also</td><td colspan="2"></td></tr>
</table>

delete()

<table>
<tr><td>Prototype</td><td colspan="2">delete(&EC, iosqHdle, struct nvme::kvKey_t key)<br>delete(&EC, iosqHdle, struct nvme::kvKey_t key, nsid)</td></tr>
<tr><td>Description</td><td colspan="2">Issue a Delete command through an IO SQ.</td></tr>
<tr><td>Return Value</td><td colspan="2">0 : failed<br>1 : if the polling mode attribute of the IO SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully</td></tr>
<tr><td rowspan="4">Parameter</td><td>EC</td><td>This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0.</td></tr>
<tr><td>iosqHdle</td><td>An IO SQ handle.</td></tr>
<tr><td>key</td><td>The KV key which is a structure parameter. The structure is defined as the following. The KeyLen specifies the length of the key in bytes. Each of CDW2_Key, CDW3_Key, CDW14_Key, and CDW15_Key is a 32-bit value of the key and corresponds to Command Dword 2, 3, 14, and 15 respectively.<br><pre>struct kvKey_t {
  var KeyLen;
  var CDW2_Key;
  var CDW3_Key;
  var CDW14_Key;
  var CDW15_Key;
  clean()
  {
    KeyLen = 0;
    CDW2_Key = 0;
    CDW3_Key = 0;
    CDW14_Key = 0;
    CDW15_Key = 0;
  }
};</pre></td></tr>
<tr><td>nsid (optional)</td><td>A namespace Id.</td></tr>
<tr><td>Note</td><td colspan="2">A namespace handle can be set as the default accessed one of an SQ by nvme::setSQAttr(). Without passing the nsid argument to this function, the nsid of the default namespace of the IO SQ will be used.</td></tr>
<tr><td>See Also</td><td colspan="2"></td></tr>
</table>

exist()

<table>
<tr><td>Prototype</td><td colspan="2">exist(&EC, iosqHdle, struct nvme::kvKey_t key)<br>exist(&EC, iosqHdle, struct nvme::kvKey_t key, nsid)</td></tr>
<tr><td>Description</td><td colspan="2">Issue a Exist command through an IO SQ.</td></tr>
<tr><td>Return Value</td><td colspan="2">0 : failed<br>1 : if the polling mode attribute of the IO SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully</td></tr>
<tr><td rowspan="4">Parameter</td><td>EC</td><td>This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0.</td></tr>
<tr><td>iosqHdle</td><td>An IO SQ handle.</td></tr>
<tr><td>key</td><td>The KV key which is a structure parameter. The structure is defined as the following. The KeyLen specifies the length of the key in bytes. Each of CDW2_Key, CDW3_Key, CDW14_Key, and CDW15_Key is a 32-bit value of the key and corresponds to Command Dword 2, 3, 14, and 15 respectively.<br><pre>struct kvKey_t {
  var KeyLen;
  var CDW2_Key;
  var CDW3_Key;
  var CDW14_Key;
  var CDW15_Key;
  clean()
  {
    KeyLen = 0;
    CDW2_Key = 0;
    CDW3_Key = 0;
    CDW14_Key = 0;
    CDW15_Key = 0;
  }
};</pre></td></tr>
<tr><td>nsid<br>(optional)</td><td>A namespace Id.</td></tr>
<tr><td>Note</td><td colspan="2">A namespace handle can be set as the default accessed one of an SQ by nvme::setSQAttr(). Without passing the nsid argument to this function, the nsid of the default namespace of the IO SQ will be used.</td></tr>
<tr><td>See Also</td><td colspan="2"></td></tr>
</table>

list()

| Prototype | `list(&EC, sqh, buf, bufsz, struct nvme::kvKey_t key)`<br>`list(&EC, sqh, buf, bufsz, struct nvme::kvKey_t key,  nsid)` | |
|---|---|---|
| Description | Issue a List command through an IO SQ. | |
| Return Value | 0 : failed<br>1 : if the polling mode attribute of the IO SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | sqh | An IO SQ handle. |
| | buf | The host buffer to which data is transferred. |
| | bufsz | The size of the buffer. |
| | key | The KV key which is a structure parameter. The structure is defined as the following. The KeyLen specifies the length of the key in bytes. Each of CDW2_Key, CDW3_Key, CDW14_Key, and CDW15_Key is a 32-bit value of the key and corresponds to Command Dword 2, 3, 14, and 15 respectively.<br>`struct kvKey_t {`<br>`  var KeyLen;`<br>`  var CDW2_Key;`<br>`  var CDW3_Key;`<br>`  var CDW14_Key;`<br>`  var CDW15_Key;`<br>`  clean()`<br>`  {`<br>`  KeyLen = 0;`<br>`  CDW2_Key = 0;`<br>`  CDW3_Key = 0;`<br>`  CDW14_Key = 0;`<br>`  CDW15_Key = 0;`<br>`  }`<br>`};` |
| | nsid (optional) | A namespace Id. |
| Note | A namespace handle can be set as the default accessed one of an SQ by nvme::setSQAttr(). Without passing the nsid argument to this function, the nsid of the default namespace of the IO SQ will be used. | |
| See Also | | |

zoneMgmtSend()

| | | |
|---|---|---|
| Prototype | `zoneMgmtSend(&EC, sqh, buf, slba, bSelAll, act)`<br>`zoneMgmtSend(&EC, sqh, buf, slba, bSelAll, act, nsid)` | |
| Description | Issue a Zone Management Send command through an IO SQ. | |
| Return Value | 0 : failed<br>1 : if the polling mode attribute of the IO SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | sqh | An IO SQ handle. |
| | buf | The buffer where the data is. |
| | slba | To specify the lowest LBA of the zone on which the Zone Send Action is performed. |
| | bSelAll | An 1-bit value corresponding to bit 8 of the CDW13 of the Zone Management Send command. |
| | act | An 8-bit value standing for the Zone Send Action. |
| | nsid (optional) | A namespace Id. |
| Note | A namespace handle can be set as the default accessed one of an SQ by nvme::setSQAttr(). Without passing the nsid argument to this function, the nsid of the default namespace of the IO SQ will be used. | |
| See Also | | |

zoneMgmtRecv()

| | | |
|---|---|---|
| Prototype | `zoneMgmtRecv(&EC, sqh, buf, slba, nDWNum, CDW13)`<br>`zoneMgmtRecv(&EC, sqh, buf, slba, nDWNum, CDW13, nsid)` | |
| Description | Issue a Zone Management Send command through an IO SQ. | |
| Return Value | 0 : failed<br>1 : if the polling mode attribute of the IO SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully | |
| Parameter | EC | This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0. |
| | sqh | An IO SQ handle. |
| | buf | The buffer to which data is transferred. |
| | slba | To specify the lowest LBA of the zone on which the Zone Send Action is performed. |
| | nDWNum | To specify number of Dwords to return. |
| | CDW13 | The value of the Command Dword 13. |
| | nsid (optional) | A namespace Id. |
| Note | A namespace handle can be set as the default accessed one of an SQ by nvme::setSQAttr(). Without passing the nsid argument to this function, the nsid of the default namespace of the IO SQ will be used. | |
| See Also | | |

zoneAppend()

<table>
<tr><td>Prototype</td><td colspan="2">zoneAppend(&EC, sqh, dbuf, zslba, nlb, mbuf)<br>zoneAppend(&EC, sqh, dbuf, zslba, nlb, mbuf, nsid)<br>zoneAppend(&EC, sqh, dbuf, zslba, nlb, mbuf, nsid, struct nvme::zoneAppendOptParams_t optParams)</td></tr>
<tr><td>Description</td><td colspan="2">Issue a Zone Management Send command through an IO SQ.</td></tr>
<tr><td>Return Value</td><td colspan="2">0 : failed<br>1 : if the polling mode attribute of the IO SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully</td></tr>
<tr><td rowspan="8">Parameter</td><td>EC</td><td>This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0.</td></tr>
<tr><td>sqh</td><td>An IO SQ handle.</td></tr>
<tr><td>dbuf</td><td>The buffer where the data is.</td></tr>
<tr><td>zslba</td><td>To specify the lowest LBA of the zone on which the Zone Send Action is performed.</td></tr>
<tr><td>nlb</td><td>Number of logic blocks.</td></tr>
<tr><td>mbuf</td><td>The buffer where the metadata is.</td></tr>
<tr><td>nsid<br>(optional)</td><td>A namespace Id.</td></tr>
<tr><td>optParams<br>(optional)</td><td>A structure variable for additional optional parameters. The structure is defined as the following. Each structure field correspond to a Command Dword field or a full Command Dword.<br>struct zoneAppendOptParams_t {<br>var CDW0_FUSE;<br>var CDW2;<br>var CDW3;<br>var CDW12_LR;<br>var CDW12_FUA;<br>var CDW12_PRINFO;<br>var CDW12_PIREMAP;<br>var CDW12_STC;<br>var CDW12_DTYPE;<br>var CDW13_DSPEC;<br>var CDW14;<br>var CDW15_LBATM;<br>var CDW15_LBAT;<br>clean()<br>{<br>CDW0_FUSE = 0;<br>CDW2 = 0;<br>CDW3 = 0;<br>CDW12_LR = 0;<br>CDW12_FUA = 0;<br>CDW12_PRINFO = 0;<br>CDW12_PIREMAP = 0;<br>CDW12_STC = 0;</td></tr>
</table>

| | | |
|---|---|---|
| | | CDW12_DTYPE = 0;<br>CDW13_DSPEC = 0;<br>CDW14 = 0;<br>CDW15_LBATM = 0;<br>CDW15_LBAT = 0;<br>}<br>}; |
| Note | A namespace handle can be set as the default accessed one of an SQ by nvme::setSQAttr(). Without passing the nsid argument to this function, the nsid of the default namespace of the IO SQ will be used. | |
| See Also | | |

| |
|---|
| |

passThru()

<table>
<tr><td>Prototype</td><td colspan="2">passThru(&EC,<br>iosqHdle,<br>dataBuf,<br>dataLen_inBytes,<br>metaBuf,<br>nvme:: passThruParams_t passThruParams)</td></tr>
<tr><td>Description</td><td colspan="2">Issue an arbitrary op-code command through an IO SQ.</td></tr>
<tr><td>Return Value</td><td colspan="2">0 : failed and no command is issued<br>1 : if the polling mode attribute of the Admin SQ is disabled and the command is sent successfully<br>Otherwise : an command handle; command is sent successfully</td></tr>
<tr><td rowspan="6">Parameter</td><td>EC</td><td>This parameter is a call-by-reference one. The argument should be an elementary variable. An error code is stored on this variable if the return value is 0.</td></tr>
<tr><td>iosqHdle</td><td>An IO SQ handle.</td></tr>
<tr><td>dataBuf</td><td>The data buffer handle; 0 shall be specified if data buffer is not needed.</td></tr>
<tr><td>dataLen_inBytes</td><td>Length of data in bytes.<br>1 = 1 Byte,<br>2 = 2 Bytes,<br>N = N Bytes</td></tr>
<tr><td>metaBuf</td><td>The metadata buffer handle; 0 shall be specified if metadata buffer is not needed.</td></tr>
<tr><td>passThruParams</td><td>Parameters for the command; should be a nvme::passThruParams_t structure. The structure is defined as the following. The CSI field is used for specifying the Command Set Identifier: 0 for NVM Command Set; 1 for Key Value Command Set; and 2 for Zoned Namespace Command Set. Other fields correspond to a field of a Command Dword or a Command Dword.<br>struct passThruParams_t {<br>var CSI;<br>var CDW0_FUSE, CDW0_OPC;<br>var NamespaceId;<br>var CDW2,CDW3,CDW10,CDW11,CDW12,CDW13,CDW14,CDW15;<br>clean()<br>{<br>CSI = 0;<br>CDW0_FUSE = 0;<br>CDW0_OPC = 0;<br>NamespaceId = 0;<br>CDW2 = 0;<br>CDW3 = 0;<br>CDW10 = 0;<br>CDW11 = 0;<br>CDW12 = 0;<br>CDW13 = 0;<br>CDW14 = 0;<br>CDW15 = 0;</td></tr>
</table>

| | | |
|---|---|---|
| | | }<br>}; |
| Note | The namespace Id is retrieved from the passThruParams but not from the default accessed namespace handle set to the IO SQ. | |
| See Also | | |

```
main()
{
    struct nvme::passThruParams_t ptcmd;

    ...
    sqHandle = nvme::getSQHandle(EC, cntlHandle, 1);
    dbuf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 4096);
    mbuf = nvme::ddbAlloc(EC, cntlHandle, nvme::PRP, 4096);

    // Write
    ptcmd.clean();
    ptcmd.CSI = 0;
    ptcmd.CDW0_OPC = 1;  // opc 1 = write
    ptcmd.NamespaceId = nsid;
    ptcmd.CDW10 =  slba        & 0xFFFFFFFF;
    ptcmd.CDW11 = (slba >> 32) & 0xFFFFFFFF;
    ptcmd.CDW12 = (1 << 30) | (1 - 1);   // bit 30: FUA; 1 - 1: NLB
    cmdHandle = nvme::ioc::passThru(EC, sqHandle, dbuf, 4096, mbuf, ptcmd);

    // Flush
    ptcmd.clean();
    ptcmd.CSI = 0;
    ptcmd.CDW0_OPC = 0; // opc 1 = flush
    ptcmd.NamespaceId = 0xFFFFFFFF;
    cmdHandle = nvme::ioc::passThru(EC, sqHandle, 0, 0, 0, ptcmd);
}
```

## Extension structures in namespace nvme

struct queueInitParams_t

```
struct queueInitParams_t {
  var nASQSize;
  var nACQSize;
  var nIOSQNum;
  var nIOSQSize;
  var nIOSQNumPerIOCQ;
  var nIOCQSize;
  var nIOCQNumPerPCCT;
  setDef() {
      nASQSize = 0;
      nACQSize = 0;
      nIOSQNum = 0;
      nIOSQSize = 0;
      nIOSQNumPerIOCQ = 0;
      nIOCQSize = 0;
      nIOCQNumPerPCCT = 0;
  }
};
```

A variable of this structure is used to customize the creations of queues and is passed to init() function.

| | |
|---|---|
| nASQSize | Specify the size of the Admin Submission Queue. If 0 is specified, a default value will be applied. |
| nACQSize | Specify the size of the Admin Completion Queue. If 0 is specified, a default value will be applied. |
| nIOSQNum | Specify the number of I/O Submission Queues. If 0 is specified, a default value will be applied. |
| nIOSQSize | Specify the size of each I/O Submission Queue. If 0 is specified, a default value will be applied. |
| nIOSQNumPerIOCQ | Specify the method to pair I/O SQ and I/O CQ. Say this number is N. Then:<br>IOSQ-1 ~ IOSQ-N are paired with IOCQ-1.<br>IOSQ-N+1 ~ IOSQ-2*N are paired with IOCQ-2.<br>If 0 is specified, a default value will be applied. |
| nIOCQSize | Specify the size of each I/O Completion Queue. If 0 is specified, a default value will be applied. |
| nIOCQNumPerPCCT | Specify the number of I/O Completion Queues to be polled by a task. Say this number is M. Then:<br>IOCQ-1 ~ IOCQ-M are polled by one task.<br>IOCQ-M+1 ~ IOCQ-2*M are polled by another task. |

| | |
|---|---|
| | If 0 is specified, a default value will be applied. |

struct HMBInfo_t

```
struct HMBInfo_t {
  var nPageSize; /*in bytes*/
  var nTotalSize; /*in bytes*/
  var nDescpListPhyAddr;
  var nDescpCount;
  clean()
  {
    nPageSize = 0;
    nTotalSize = 0;
    nDescpListPhyAddr = 0;
    nDescpCount = 0;
  }
};
```

A variable of this structure is passed to ddbAlloc() for ddbAlloc() to update allocation information of the HMB buffer.

| | |
|---|---|
| nPageSize | The memory page size of the system in bytes. |
| nTotalSize | The memory allocation size in bytes. |
| nDescpListPhyAddr | The physical memory address of the Host Memory Descriptor List. |
| nDescpCount | The number of Host Memory Buffer Descriptors. |

struct identifyOptParams_t

```
struct identifyOptParams_t {
  var CDW11_CNSSpecID;
  var CDW11_CSI;
  var CDW14_UUIDIdx
  clean()
  {
    CDW11_CNSSpecID = 0;
    CDW11_CSI = 0;
    CDW14_UUIDIdx = 0;
  }
};
```

A variable of this structure is used to specify optional parameters and is passed to the identify() function to send an Identify command.

| | |
|---|---|
| CDW11_CNSSpecID | To specify bit0~bit15 value of CDW11. |
| CDW11_CSI | To specify the Command Set Identifier; bit24~bit31 value of CDW11. |
| CDW14_UUIDIdx | To specify the UUID Index; bit0~bit6 value of CDW14. |

struct setFeaturesOptParams_t

```
struct setFeaturesOptParams_t {
  var NamespaceId;
  var CDW12;
  var CDW13;
  var CDW14;
  var CDW15;
  clean()
  {
    NamespaceId = 0;
    CDW12 = 0;
    CDW13 = 0;
    CDW14 = 0;
    CDW15 = 0;
  }
};
```

A variable of this structure is used to specify optional parameters and is passed to the setFeatures() function to send a Set Features command.

| | |
|---|---|
| NamespaceId | The namespace Id. |
| CDW12 ~ CDW15 | To specify values for Command Dword12 ~ Dword15. |

struct getLogPageOptParams_t

```
struct getLogPageOptParams_t {
  var CDW10_RAE;
  var CDW10_LSP;
  var CDW11_LSId;
  var LogPageOffset;
  var CDW14_OffsetType;
  var CDW14_UUIDIndex;
  clean()
  {
    CDW10_RAE = 0;
    CDW10_LSP = 0;
    CDW11_LSId = 0;
    LogPageOffset = 0;
    CDW14_OffsetType = 0;
    CDW14_UUIDIndex = 0;
  }
};
```

A variable of this structure is used to specify optional parameters and is passed to the getLogPage() function to send a Get Log Page command.

| | |
|---|---|
| CDW10_RAE | To specify the bit15 (Retain Asynchronous Event) value of CDW10. |
| CDW10_LSP | To specify the bit8~bit11 (Log Specific Field) value of CDW10. |
| CDW11_LSId | To specify the bit16~bit31 (Log Specific Identifier) value of CDW11. |
| LogPageOffset | To specify the Log Page Offset value which is combined by CDW12 and CDW13. |
| CDW14_OffsetType | To specify the bit23 (Offset Type) value of CDW14. |
| CDW14_UUIDIndex | To specify the bit0~bit6 (UUID Index) value of CDW14. |

struct dirSendParams_t

```
struct dirSendParams_t {
  var CDW11_dirOp;
  var CDW11_dirType;
  var CDW11_dirSpec;
  var CDW12;
  var CDW13;
  clean()
  {
    CDW11_dirOp = 0;
    CDW11_dirType = 0;
    CDW11_dirSpec = 0;
    CDW12 = 0;
    CDW13 = 0;
  }
};
```

A variable of this structure is used to specify parameters and is passed to the dirSend() function to send a Directive Send command.

| | |
|---|---|
| CDW11_dirOp | To specify the bit0~7 (Directive Operation) value of CDW11. |
| CDW11_dirType | To specify the bit8~15 (Directive Type) value of CDW11. |
| CDW11_dirSpec | To specify the bit16~31 (Directive Specific) value of CDW11. |
| CDW12, CDW13 | To specify values for Command Dword12 and Dword13. |

struct dirRecvParams_t

```
struct dirRecvParams_t {
  var CDW11_dirOp;
  var CDW11_dirType;
  var CDW11_dirSpec;
  var CDW12;
  var CDW13;
  clean()
  {
    CDW11_dirOp = 0;
    CDW11_dirType = 0;
    CDW11_dirSpec = 0;
    CDW12 = 0;
    CDW13 = 0;
  }
};
```

A variable of this structure is used to specify parameters and is passed to the dirRecv() function to send a Directive Receive command.

| | |
|---|---|
| CDW11_dirOp | To specify the bit0~7 (Directive Operation) value of CDW11. |
| CDW11_dirType | To specify the bit8~15 (Directive Type) value of CDW11. |
| CDW11_dirSpec | To specify the bit16~31 (Directive Specific) value of CDW11. |
| CDW12, CDW13 | To specify values for Command Dword12 and Dword13. |

struct passThruParams_t

```
struct passThruParams_t {
  var CSI;
  var CDW0_FUSE, CDW0_OPC;
  var NamespaceId;
  var CDW2, CDW3, CDW10, CDW11, CDW12, CDW13, CDW14, CDW15;
  clean()
  {
    CSI = 0;
    CDW0_FUSE = 0;
    CDW0_OPC = 0;
    NamespaceId = 0;
    CDW2 = 0;
    CDW3 = 0;
    CDW10 = 0;
    CDW11 = 0;
    CDW12 = 0;
    CDW13 = 0;
    CDW14 = 0;
    CDW15 = 0;
  }
};
```

A variable of this structure is used to specify parameters and is passed to the nvme::ioc::passThru() or nvme::admc::passThru() function to send a pass-through command.

| | |
|---|---|
| CSI | To specify the Command Set Identifier. Value of this field is not passed to as a CDW field to the device.<br>0 : NVM Command Set<br>1 : Key Value Command Set<br>2 : Zoned Namespace Command Set.<br>This value should be set as 0 when nvme::admc::passThru() is called. |
| CDW0_FUSE | To specify bit8~9 (Fused Operation) value of CDW0. |
| CDW0_OPC | The command opcode. A 8-bit value. |
| NamespaceId | The namespace ID and is filled into CDW1. |
| CDW2, …, CDW15 | To specify values to corresponding Command Dwords. |

struct compareOptParams_t

```
struct compareOptParams_t {
  var CDW0_FUSE;
  var CDW2;
  var CDW3;
  var CDW12_LR;
  var CDW12_FUA;
  var CDW12_PRINFO;
  var CDW12_STC;
  var CDW14;
  var CDW15_ELBATM;
  var CDW15_ELBAT;
  clean()
  {
    CDW0_FUSE = 0;
    CDW2 = 0;
    CDW3 = 0;
    CDW12_LR = 0;
    CDW12_FUA = 0;
    CDW12_PRINFO = 0;
    CDW12_STC = 0;
    CDW14 = 0;
    CDW15_ELBATM = 0;
    CDW15_ELBAT = 0;
  }
};
```

A variable of this structure is used to specify optional parameters and is passed to the compare() function to send a Compare command.

| | |
|---|---|
| CDW0_FUSE | To specify bit8~9 (Fused Operation) value of CDW0. |
| CDW2, CDW3 | To specify values for Command Dword2 and Dword3. |
| CDW12_LR | To specify bit31 (Limited Retry) value of CDW12. |
| CDW12_FUA | To specify bit30 (Force Unit Access) value of CDW12. |
| CDW12_PRINFO | To specify bit26~19 (Protection Information Field) value of CDW12. |
| CDW12_STC | To specify bit24 (Storage Tag Check) value of CDW12. |
| CDW14 | To specify value for Command Dword14. |
| CDW15_ELBATM | To specify bit16~31 (Expected Logical Block Application Tag Mask) value of CDW15. |
| CDW15_ELBAT | To specify bit0~15 (Expected Logical Block Application Tag) value of CDW15. |

struct verifyOptParams_t

```
struct verifyOptParams_t {
  var CDW0_FUSE;
  var CDW2;
  var CDW3;
  var CDW12_LR;
  var CDW12_FUA;
  var CDW12_PRINFO;
  var CDW12_STC;
  var CDW14;
  var CDW15_ELBATM;
  var CDW15_ELBAT;
  clean()
  {
    CDW0_FUSE = 0;
    CDW2 = 0;
    CDW3 = 0;
    CDW12_LR = 0;
    CDW12_FUA = 0;
    CDW12_PRINFO = 0;
    CDW12_STC = 0;
    CDW14 = 0;
    CDW15_ELBATM = 0;
    CDW15_ELBAT = 0;
  }
};
```

A variable of this structure is used to specify optional parameters and is passed to the verify() function to send a Verify command.

| | |
|---|---|
| CDW0_FUSE | To specify bit8~9 (Fused Operation) value of CDW0. |
| CDW2, CDW3 | To specify values for Command Dword2 and Dword3. |
| CDW12_LR | To specify bit31 (Limited Retry) value of CDW12. |
| CDW12_FUA | To specify bit30 (Force Unit Access) value of CDW12. |
| CDW12_PRINFO | To specify bit26~19 (Protection Information Field) value of CDW12. |
| CDW12_STC | To specify bit24 (Storage Tag Check) value of CDW12. |
| CDW14 | To specify value for Command Dword14. |
| CDW15_ELBATM | To specify bit16~31 (Expected Logical Block Application Tag Mask) value of CDW15. |
| CDW15_ELBAT | To specify bit0~15 (Expected Logical Block Application Tag) value of CDW15. |

struct writeOptParams_t

```
struct writeOptParams_t {
  var CDW0_FUSE;
  var CDW2;
  var CDW3;
  var CDW12_LR;
  var CDW12_FUA;
  var CDW12_PRINFO;
  var CDW12_STC;
  var CDW12_DTYPE;
  var CDW13_DSPEC;
  var CDW13_DSM;
  var CDW14;
  var CDW15_LBATM;
  var CDW15_LBAT;
  clean()
  {
    CDW0_FUSE = 0;
    CDW2 = 0;
    CDW3 = 0;
    CDW12_LR = 0;
    CDW12_FUA = 0;
    CDW12_PRINFO = 0;
    CDW12_STC = 0;
    CDW12_DTYPE = 0;
    CDW13_DSPEC = 0;
    CDW13_DSM = 0;
    CDW14 = 0;
    CDW15_LBATM = 0;
    CDW15_LBAT = 0;
  }
};
```

A variable of this structure is used to specify optional parameters and is passed to the write() function to send a Write command.

| | |
|---|---|
| CDW0_FUSE | To specify bit8~9 (Fused Operation) value of CDW0. |
| CDW2, CDW3 | To specify values for Command Dword2 and Dword3. |
| CDW12_LR | To specify bit31 (Limited Retry) value of CDW12. |
| CDW12_FUA | To specify bit30 (Force Unit Access) value of CDW12. |
| CDW12_PRINFO | To specify bit26~19 (Protection Information Field) value of CDW12. |
| CDW12_STC | To specify bit24 (Storage Tag Check) value of CDW12. |
| CDW12_DTYPE | To specify bit20~23 (Directive Type) value of CDW12. |
| CDW13_DSPEC | To specify bit16~31 (Directive Specific) value of CDW13. |
| CDW13_DSM | To specify bit0~7 (Dataset Management) value of CDW13. |
| CDW14 | To specify value for Command Dword14. |

| | |
|---|---|
| CDW15_LBATM | To specify bit16~31 (Logical Block Application Tag Mask) value of CDW15. |
| CDW15_LBAT | To specify bit0~15 (Logical Block Application Tag) value of CDW15. |

struct write0OptParams_t

```
struct write0OptParams_t {
  var CDW0_FUSE;
  var CDW2;
  var CDW3;
  var CDW12_LR;
  var CDW12_FUA;
  var CDW12_PRINFO;
  var CDW12_DEAC;
  var CDW12_STC;
  var CDW14;
  var CDW15_LBATM;
  var CDW15_LBAT;
  clean()
  {
    CDW0_FUSE = 0;
    CDW2 = 0;
    CDW3 = 0;
    CDW12_LR = 0;
    CDW12_FUA = 0;
    CDW12_PRINFO = 0;
    CDW12_DEAC = 0;
    CDW12_STC = 0;
    CDW14 = 0;
    CDW15_LBATM = 0;
    CDW15_LBAT = 0;
  }
};
```

A variable of this structure is used to specify optional parameters and is passed to the write0() function to send a Write Zeroes command.

| | |
|---|---|
| CDW0_FUSE | To specify bit8~9 (Fused Operation) value of CDW0. |
| CDW2, CDW3 | To specify values for Command Dword2 and Dword3. |
| CDW12_LR | To specify bit31 (Limited Retry) value of CDW12. |
| CDW12_FUA | To specify bit30 (Force Unit Access) value of CDW12. |
| CDW12_PRINFO | To specify bit26~19 (Protection Information Field) value of CDW12. |
| CDW12_DEAC | To specify bit25 (Deallocate) value of CDW12. |
| CDW12_STC | To specify bit24 (Storage Tag Check) value of CDW12. |
| CDW14 | To specify value for Command Dword14. |
| CDW15_LBATM | To specify bit16~31 (Logical Block Application Tag Mask) value of CDW15. |
| CDW15_LBAT | To specify bit0~15 (Logical Block Application Tag) value of CDW15. |

struct copyOptParams_t

```
struct copyOptParams_t {
  var CDW0_FUSE;
  var CDW2;
  var CDW3;
  var CDW12_LR;
  var CDW12_FUA;
  var CDW12_PRINFOW;
  var CDW12_STCW;
  var CDW12_DTYPE;
  var CDW12_PRINFOR;
  var CDW13_DSPEC;
  var CDW14;
  var CDW15_LBATM;
  var CDW15_LBAT;
  clean()
  {
    CDW0_FUSE = 0;
    CDW2 = 0;
    CDW3 = 0;
    CDW12_LR = 0;
    CDW12_FUA = 0;
    CDW12_PRINFOW = 0;
    CDW12_STCW = 0;
    CDW12_DTYPE = 0;
    CDW12_PRINFOR = 0;
    CDW13_DSPEC = 0;
    CDW14 = 0;
    CDW15_LBATM = 0;
    CDW15_LBAT = 0;
  }
};
```

A variable of this structure is used to specify optional parameters and is passed to the copy() function to send a Copy command.

| | |
|---|---|
| CDW0_FUSE | To specify bit8~9 (Fused Operation) value of CDW0. |
| CDW2, CDW3 | To specify values for Command Dword2 and Dword3. |
| CDW12_LR | To specify bit31 (Limited Retry) value of CDW12. |
| CDW12_FUA | To specify bit30 (Force Unit Access) value of CDW12. |
| CDW12_PRINFOW | To specify bit26~19 (Protection Information Write) value of CDW12. |
| CDW12_STCW | To specify bit24 (Storage Tag Check Write) value of CDW12. |
| CDW12_DTYPE | To specify bit20~23 (Directive Type) value of CDW12. |
| CDW13_DSPEC | To specify bit16~31 (Directive Specific) value of CDW13. |
| CDW14 | To specify value for Command Dword14. |
| CDW15_LBATM | To specify bit16~31 (Logical Block Application Tag Mask) value of CDW15. |

| | |
|---|---|
| CDW15_LBAT | To specify bit0~15 (Logical Block Application Tag) value of CDW15. |

struct kvKey_t

```
struct kvKey_t {
  var KeyLen;
  var CDW2_Key;
  var CDW3_Key;
  var CDW14_Key;
  var CDW15_Key;
  clean()
  {
    KeyLen = 0;
    CDW2_Key = 0;
    CDW3_Key = 0;
    CDW14_Key = 0;
    CDW15_Key = 0;
  }
};
```

A variable of this structure is used to specify a KV key and is passed to the store(), retrieve(), delete(), exist(), and list() functions to send a Store, Retrieve, Delete, Exist, and List commands.

| | |
|---|---|
| KeyLen | A 8-bit value to specify the length of the KV key in bytes. |
| CDW2_Key,<br>CDW3_Key,<br>CDW14_Key,<br>CDW15_Key | Key contents in CDW2, CDW3, CDW14, and CDW15 respectively. |

struct zoneAppendOptParams_t

```
struct zoneAppendOptParams_t {
  var CDW0_FUSE;
  var CDW2;
  var CDW3;
  var CDW12_LR;
  var CDW12_FUA;
  var CDW12_PRINFO;
  var CDW12_PIREMAP;
  var CDW12_STC;
  var CDW12_DTYPE;
  var CDW13_DSPEC;
  var CDW14;
  var CDW15_LBATM;
  var CDW15_LBAT;
  clean()
  {
    CDW0_FUSE = 0;
    CDW2 = 0;
    CDW3 = 0;
    CDW12_LR = 0;
    CDW12_FUA = 0;
    CDW12_PRINFO = 0;
    CDW12_PIREMAP = 0;
    CDW12_STC = 0;
    CDW12_DTYPE = 0;
    CDW13_DSPEC = 0;
    CDW14 = 0;
    CDW15_LBATM = 0;
    CDW15_LBAT = 0;
  }
};
```

A variable of this structure is used to specify optional parameters and is passed to the zoneAppend() function to send a Zone Append command.

| | |
|---|---|
| CDW0_FUSE | To specify bit8~9 (Fused Operation) value of CDW0. |
| CDW2, CDW3 | To specify values for Command Dword2 and Dword3. |
| CDW12_LR | To specify bit31 (Limited Retry) value of CDW12. |
| CDW12_FUA | To specify bit30 (Force Unit Access) value of CDW12. |
| CDW12_PRINFO | To specify bit26~19 (Protection Information Field) value of CDW12. |
| CDW12_PIREMAP | To specify bit25 (Protection Information Remap) value of CDW12. |
| CDW12_STC | To specify bit24 (Storage Tag Check) value of CDW12. |
| CDW12_DTYPE | To specify bit20~23 (Directive Type) value of CDW12. |
| CDW13_DSPEC | To specify bit16~31 (Directive Specific) value of CDW13. |
| CDW14 | To specify value for Command Dword14. |

| | |
|---|---|
| CDW15_LBATM | To specify bit16~31 (Logical Block Application Tag Mask) value of CDW15. |
| CDW15_LBAT | To specify bit0~15 (Logical Block Application Tag) value of CDW15. |