pynvme

Release 2.3.2

cranechu@gmail.com

Sep 28, 2023

TABLE OF CONTENTS

Pynvme is an user-space NVMe test driver with Python API. It is an open, fast, and extensible solution for SSD developers and test engineers to build their own tests intuitively.

Quick Start in 3 Steps!

1. Clone pynvme from GitHub: https://github.com/pynvme/pynvme

```
git clone https://github.com/pynvme/pynvme
```

2. Build pynvme:

```
cd pynvme
./install.sh
```

3. Run pynvme tests.

```
make setup
make test TESTS="driver_test.py::test_ioworker_iops_multiple_queue[1]"
```

In this document, we explain the design and usage of pynvme, and you can soon build your own tests of NVMe devices.

CHAPTER

ONE

INTRODUCTION

1.1 Background

Storage is important to client computers, data centers and enterprise servers. NVMe is a fast growing standard of storage in the era of Solid State Storage. However, we were still using traditional testing tools which are not designed for NVMe compatibility, performance, reliability and rapid evolution. These tools became the bottleneck of the project. First, most of the existed tools do not provide source code or API, so it is very difficult to be integrited into the automatic test flow, which is critical to Agile project management. Second, many testing tools are developed in HDD/SATA era, so it cannot consistently benchmark the super-fast NVMe SSD. Last but not least, NVMe specification is keeping growing, but the testing tools cannot be changed rapidly. Then, pynvme comes.

1.2 Design

Pynvme is an user-space PCIe/NVMe test driver with Python API. It is open, fast, and extensible.

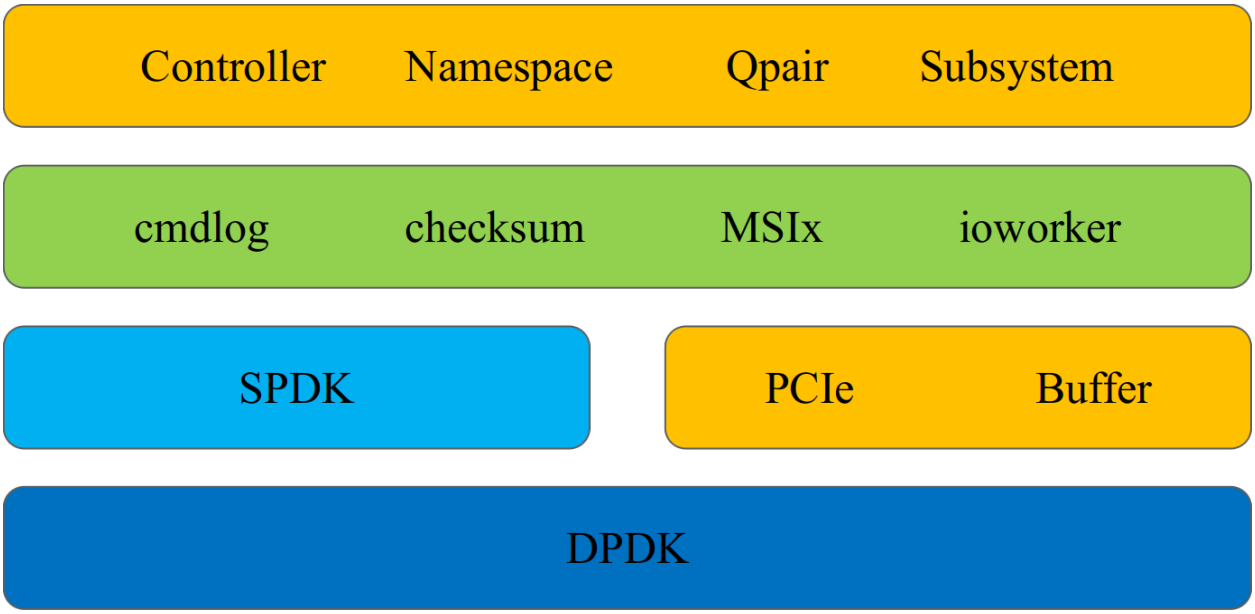
Open. Every people and team can use it and make contributions to pynvme. Instead of the GUI-based applications, pynvme is designed as an open Python module. With pynvme, as well as the whole Python ecosystem, engineers can efficiently develop and deploy their own test scripts.

Fast. Pynvme is an user-space software. It bypasses the whole Kernel and accesses PCIe/NVMe hardware resources directly to eliminate system-call overhead. Pynvme is also a poll mode driver (PMD), which further eliminates the cost of interrupts. So its performance is very high and consistent.

Extensible. Pynvme provides API to access PCIe and physical memory. As the result, a client can read/write PCIe configuration space, PCIe memory space (aka BAR space), and physical (instead of logical) memory space. These low-level capabilites open doors to many more innovations.

We did not build pynvme from scratch. We build it based on the SPDK, a reliable software stack from Intel. We extended SPDK NVMe driver with several testing-purpose functions in pynvme:

1. Interrupts. SPDK is a polling mode driver, so it does not support interrupts, like MSIx and MSI. We implemented a software interrupt host controller to enable and check interrupt signals.
2. Checksum. Storage cares data integrity. Pynvme verifies each LBA block with CRC32 checksum, without any penalty on performance.
3. Cmdlog. Pynvme traces every command and completion dwords. When any problem happens, users can check the trace data to debug the issue.
4. IOWorker. It is slow to send each IO in test scripts, therefore pynvme provides an agent to send IOes in separated processes. Users can create multiple IOWorkers with little overhead.



We then wrap all these SPDK and pynvme functions in a Python module. Users can use Python classes (e.g. Controller, Namespace, ...) to test NVMe devices. Here is an identical script as SPDK's 400-line example: https://github.com/spdk/spdk/blob/master/examples/nvme/hello_world/hello_world.c

```
import pytest
import nvme as d

def test_hello_world(nvme0, nvme0n1, qpair):
    # prepare data buffer and IO queue
    read_buf = d.Buffer(512)
    write_buf = d.Buffer(512)
    write_buf[10:21] = b'hello world'

    # send write and read command
    def write_cb(cdw0, status1):  # command callback function
        nvme0n1.read(qpair, read_buf, 0, 1)
    nvme0n1.write(qpair, write_buf, 0, 1, cb=write_cb)

    # wait commands complete and verify data
    assert read_buf[10:21] != b'hello world'
    qpair.waitdone(2)
    assert read_buf[10:21] == b'hello world'
```

Now, let's install pynvme to execute test scripts.

CHAPTER

TWO

INSTALL

2.1 All-in-One

Users can build pynvme by simply running *install.sh* after getting source code from github. This is the recommended method to compile pynvme.

```
git clone https://github.com/pynvme/pynvme
cd pynvme
./install.sh
```

Now, it generates binary python module *nvme.so*. Users can import it in Python scripts:

```
import nvme as d
nvme0 = d.Controller(d.Pcie("01:00.0"))
```

We will describe more details of installation below. You can skip to the next chapter if you have already got the binary *nvme.so*.

2.2 System Requirements

Pynvme is a software-defined NVMe test solution, so users can install and use pynvme in most of the commodity computers with some requirements:

1. CPU: Intel x86_64, 4-core or more.
2. Memory: 4GB or larger.
3. OS: Linux. Fedora is recommanded.
4. Python3: Python2 is not supported.
5. sudo privilege is required.
6. NVMe: NVMe SSD is the devices to be tested. Backup your data!
7. RAID mode (Intel® RST): shall be disabled in BIOS.
8. Secure boot: shall be disabled in BIOS.
9. IOMMU: (a.k.a. VT for Direct I/O) shall be disabled in BIOS.
10. VMD: shall be disabled in BIOS.
11. NUMA: recommend to be disabled in BIOS.
12. SATA: recommend to install OS and pynvme in a SATA drive.

2.3 Source Code

We clone the pynvme source code from GitHub. We recommend to checkout the latest stable release.

```
git clone https://github.com/pynvme/pynvme
cd pynvme
git checkout tags/2.3.2
```

Or you can clone the code from the mirror repository:

```
git clone https://gitee.com/pynvme/pynvme
```

2.4 Prerequisites

Then, we need to fetch all required dependencies.

```
# fedora-like
sudo dnf install -y make redhat-rpm-config python3-devel python3-pip

# ubuntu-like
sudo apt install -y python3-setuptools python3-dev python3-pip

# get SPDK and its submodules
git submodule update --init --recursive

# install SPDK required packages
sudo ./spdk/scripts/pkgdep.sh

# install python packages required by pynvme
sudo python3 -m pip install -r requirements.txt
```

2.5 SPDK

We need to compile and test SPDK first.

```
# use the according pynvme-modified SPDK code
cd spdk
git checkout pynvme_2.3

# configure SPDK
./configure --without-isal;

# compile SPDK
cd ..
make spdk

# compile pynvme
make
```

Now, we can find a generated binary file *nvme.so*.

2.6 Test

After compilation, let's first verify if SPDK works in your platform with SPDK applications. Before moving forward, check and backup your data in the NVMe SSD.

```
# setup SPDK runtime environment
make setup

# run pre-built SPDK application
sudo ./identify_nvme
```

This application lists identify data of your NVMe SSD. If it works, let's move ahead to run pynvme tests!

```
cd ~/pynvme
make setup
make test TESTS="driver_test.py::test_ioworker_iops_multiple_queue[1]"
```

After the test, we can find the file *test.log* in pynvme directory, which keeps more debug logs than that in the standard output. When you meet any problem, please submit issues with this *test.log*.

make setup allocates hugepages and reserves NVMe devices for SPDK runtime environment. When you want to release memory and NVMe devices back to kernel, execute this command:

```
make reset
```

2.7 pip

As an alternative way, we can also install pynvme with pip in the latest Fedora Linux.

```
pip install pynvme
cd /usr/local/pynvme
make setup
make test TESTS="driver_test.py::test_ioworker_iops_multiple_queue[1]"
```

It installs a prebuilt pynvme binary module. But we still recommend to clone pynvme source code and compile it by *install.sh*.

CHAPTER

THREE

PYTEST

The pytest framework helps developers writing test scripts from simple to complex. Pynvme is integrated with pytest, but it can also be used standalone.

3.1 Fixtures

Pytest's fixture is a powerful way to create and free resources in the test. Pynvme provides following fixtures:

fixture	scope	notes
pciaddr	session	PCIe BDF address of the DUT, pass in by argument –pciaddr
pcie	session	the object of the PCIe device.
nvme0	session	the object of default NVMe controller
nvme0n1	session	the object of first Namespace of the default controller
verify	function	declare this fixture in test cases where data CRC is to be verified.

In the following example, the fixture *nvme0n1* initializes the namespace for the test. Then, we can use it to start an ioworker directly. Super intuitive!

```
def test_ioworker_simplified_context(nvme0n1):
    with nvme0n1.ioworker(io_size=8, lba_align=16,
                          lba_random=True, qdepth=16,
                          read_percentage=0, time=2):
        pass
```

We no longer need to repeat nvme0n1 creation in every test case as usual way:

```
nvme0 = d.Controller(b'01:00.0')
nvme0n1 = d.Namespace(nvme0, 1)
```

3.2 Execution

With pytest and pre-defined makefile, we can execute tests in many flexible ways, like these:

```
# all tests under the directory
make test TESTS=scripts

# all tests in one file
```

(continues on next page)

(continued from previous page)

```
make test TESTS=scripts/demo_test.py

# a specific test function
make test TESTS=scripts/utility_test.py::test_download_firmware

# a test function with a specified parameter
make test TESTS="driver_test.py::test_ioworker_iops_multiple_queue[1]"

# start tests on multiple drives with 1GB reserved memory space each
make test TESTS=scripts/stress/endurance_test.py::test_replay_jedec_client_trace␣
↪pciaddr=01:00.0
make test TESTS=scripts/stress/endurance_test.py::test_replay_jedec_client_trace␣
↪pciaddr=02:00.0
make test TESTS=scripts/stress/endurance_test.py::test_replay_jedec_client_trace␣
↪pciaddr=172.168.5.44
```

Without specified pciaddr commandline parameter, *make test* automatically uses the last PCIe NVMe device in the lspci list. Pynvme supports multiple test sessions with different NVMe devices, or even NVMe over TCP targets, specified in the command line.

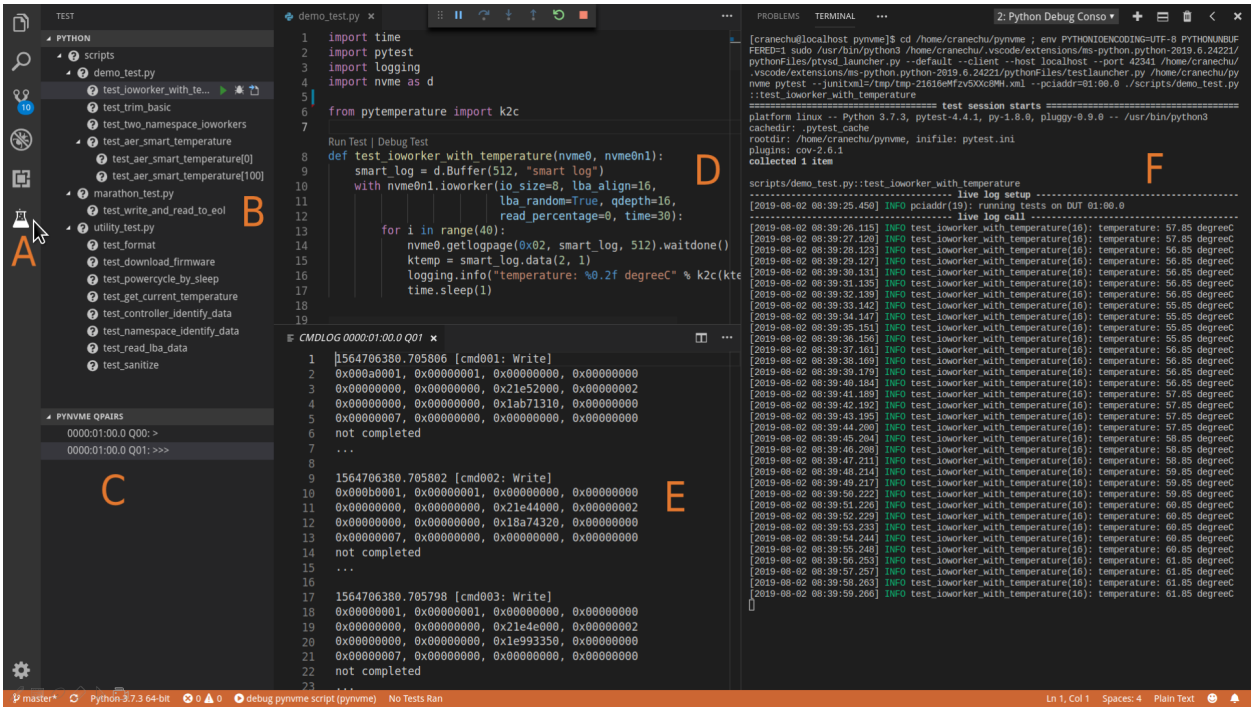
CHAPTER

FOUR

VSCODE

VSCode is an excellent source code editor and IDE. It supports python and pytest with an offical extension: https://github.com/microsoft/vscode-python. We developed another extension for pynvme to make VSCode more friendly to SSD test.

4.1 Layout



A. Activity Bar: you can select the last Test icon for pytest and pynvme extensions.

B. pytest panel: collects all test files and cases in scripts directory.

C. pynvme panel: displays all active qpairs in all controllers. Click qpair to open or refresh its cmdlog viewer. Click the icon in the upper right corner to open a performance gauge.

D. editor: edit test scripts here.

E. cmdlog viewer: displays the latest 128 commands and completion dwords in one qpair.

F. log viewer: displays pytest log.

4.2 Setup

1. First of all, install vscode here: https://code.visualstudio.com/
2. Root user is not recommended in vscode, so just use your ordinary non-root user. It is required to configure the user account to run sudo without a password.

```
sudo visudo
```

3. In order to monitor qpairs status and cmdlog along the progress of testing, user can install vscode extension pynvme-console. The extension provides DUT status and cmdlogs in VSCode UI.

```
code --install-extension pynvme-console-2.0.0.vsix
```

4. Before start vscode, modify .vscode/settings.json with the correct pcie address (bus:device.function listed in *lspci* command) of your DUT device.

```
lspci
# 01:00.0 Non-Volatile memory controller: Lite-On Technology Corporation Device␣
↪2300 (rev 01)
```

5. Then in pynvme folder, we can start vscode to edit, debug and run scripts:

```
make setup; code .  # make sure to enable SPDK nvme driver before starting vscode
```

6. Users can add their own script files under *scripts* directory. Import following packages in new test script files.

```
import pytest
import logging
import nvme as d    # import pynvme's python package
```

Now, we can edit, debug and run test scripts in VSCode! It time to learn more pynvme features and build your own tests.

CHAPTER

FIVE

FEATURES

In order to fully test NVMe devices for functionality, performance and even endurance, pynvme supports many features about NVMe controller, namespace, as well as PCIe and otheir essential parts in the system.

5.1 PCIe

NVMe devices are firstly PCIe devices, so we need to management the PCIe resources. Pynvme can access NVMe device's PCI memory space and configuration space, including all capabilities.

```
pcie = d.Pcie('3d:00.0')
hex(pcie[0:4])                  # Byte 0/1/2/3
pm_offset = pcie.cap_offset(1)  # find Power Management Capability
pcie.reset()
pcie.aspm = 2                   # set ASPM control to enable L1 only
pcie.power_state = 3            # set PCI PM power state to D3hot
```

Actually, pynvme can also test non-NVMe PCIe devices.

5.2 Buffer

In order to transfer data with NVMe devices, users need to allocate and provide the *Buffer* to IO commands. In this example, it allocates a 512-byte buffer, and get identify data of the controller in this buffer. We can also give a name to the buffer.

```
buf = d.Buffer(512, "part of identify data")
nvme.identify(buf).waitdone()
# now, the buf contains the identify data
print(buf[0:4])
del buf  # delete the `Buffer` after the commands complete.
```

5.2.1 data pattern

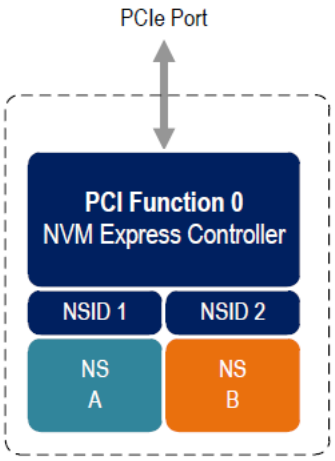
Users can identify the data pattern of the *Buffer*. Pynvme supports following different data patterns by argument *pvalue* and *ptype*.

data pattern	ptype	pvalue
all 0	0	0
all 1	0	1
repeated dwords	32	32-bit data
random data	0xBEEF	compression percentage rate

Users can also specify argument *pvalue* and *ptype* in *Namespace.ioworker()* in the same manner.

The first 8-byte and the last 8-byte of each LBA are not filled by the data pattern. The first 8-byte is the LBA address, and the last 8-byte is a token which changes on every LBA written.

5.3 Controller



To access the NVMe device, scripts have to create *Pcie* object first, and then create the *Controller* object from this *Pcie* object. It is required to close *Pcie* object when it is not used. For example:

```
import nvme as d
pcie = d.Pcie('01:00.0')
nvme0 = d.Controller(pcie)
# ...
pcie.close()
```

It uses Bus:Device:Function address to specify a PCIe DUT. Then, We can access NVMe registers and send admin commands to the NVMe device.

```
csts = nvme0[0x1c]  # CSTS register, e.g.: '0x1'
nvme0.setfeatures(0x7, cdw11=(15<<16)+15).waitdone()
```

5.3.1 NVMe Initialization

When creating controller object, pynvme implements a default initialization process defined in NVMe specification 7.6.1 (v1.4). However, scripts can define its own initialization function, which has one parameter *Controller*. Here is an example:

```
def test_init_nvme_customerized(pcie):
    def nvme_init(nvme0):
        nvme0[0x14] = 0
        while not (nvme0[0x1c]&0x1) == 0: pass
        nvme0.init_adminq()
        nvme0[0x14] = 0x00460000
        nvme0[0x14] = 0x00460001
        while not (nvme0[0x1c]&0x1) == 1: pass
        nvme0.identify(d.Buffer(4096)).waitdone()
        nvme0.init_ns()
        nvme0.setfeatures(0x7, cdw11=0x00ff00ff).waitdone()
        nvme0.getfeatures(0x7).waitdone()
        aerl = nvme0.id_data(259)+1
        for i in range(aerl):
            nvme0.aer()

    nvme0 = d.Controller(pcie, nvme_init_func=nvme_init)
```

5.3.2 Admin Commands

We set the feature number of queues (07h) above, and now we try to get the configuration data back with admin command *Controller.getfeatures()*.

```
nvme0.getfeatures(7)
```

Pynvme sends the commands asynchronously, and so we can sync and wait for the commands completion by API *Controller.waitdone()*.

```
nvme0.waitdone(1)
```

Also, *Controller.waitdone()* returns dword0 of the latest completion data structure. So, we can get the feature data in one line:

```
assert (15<<16)+15 == nvme0.getfeatures(0x7).waitdone()
```

Pynvme supports all mandatory admin commands defined in the NVMe spec, as well as most of the optional admin commands.

5.3.3 Command Callback

Scripts can specify one callback function for every command call. After the command completes, pynvme calls the specified callback function. Here is an example:

```
def getfeatures_cb1(cpl):
    logging.info(cpl)
nvme0.getfeatures(7, cb=getfeatures_cb1).waitdone()

def getfeatures_cb2(cdw0, status1):
    logging.info(status1)
nvme0.getfeatures(7, cb=getfeatures_cb2).waitdone()
```

Pynvme provides two forms of callback function. 1. single parameters: *cpl*. Pynvme shall pass the whole 16-byte completion data structure to the single parameter callback funciton. This is recommended form. 2. two parameters: *cdw0* and *status1*. Pynvme shall pass the dword0 and higher 16-bit of dword3 of Completion Queue Entry to the two-parameter callback function. *status1* is a 16-bit integer, which includes both **Phase Tag** and Status Field. This is the obsoleted form for back-compatibility only.

5.3.4 Identify Data

Here is an usual way to get controller's identify data:

```
buf = d.Buffer(4096, 'controller identify data')
nvme0.identify(buf, 0, 1).waitdone()
logging.info("model number: %s" % buf[24:63, 24])
```

Scripts shall call *Controller.waitdone()* to make sure the *buf* is filled by the NVMe device with identify data. Moving one step forward, because identify data is so frequently used, pynvme provides another API *Controller.id_data()* to get a field of the controller's identify data more easily:

```
logging.info("model number: %s" % nvme0.id_data(63, 24, str))
logging.info("vid: 0x%x" % nvme0.id_data(1, 0))
```

It retrieves bytes from 24 to 63, and interpret them as a *str* object. If the third argument is omitted, they are interpreted as an *int*. Users can refer to NVMe specification to get the fields of the data.

5.3.5 Generic Commands

Pynvme provides API for all mandatory admin commands and most of the optional admin commands listed in the NVMe specification. However, pynvme also provides the API to send the generic admin commands, *Controller.send_cmd()*. This API can be used for: 1. pynvme un-supported admin commands, 2. Vendor Specific admin commands 3. illegal Admin Commands

```
nvme0.send_cmd(0xff).waitdone()

def getfeatures_cb_2(cdw0, status1):
    logging.info(status1)
nvme0.send_cmd(0xa, nsid=1, cdw10=7, cb=getfeatures_cb_2).waitdone()
```

5.3.6 Utility Functions

Besides admin commands, class *Controller* also provides some utility functions, such as *Controller.reset()* and *Controller.downfw()*. Please refer to the last chapter for the full list of APIs.

```
nvme0.downfw('path/to/firmware_image_file')
nvme0.reset()
```

Please note that, these utility functions are not NVMe admin commands, so we do not need to reap them by *Controller.waitdone()*.

5.3.7 Timeout

The timeout duration is configurable, and the default time is 10 seconds. Users can change the timeout setting for those expected long-time consuming commands.

```
nvme0.timeout=30000  # the unit is milli-second
nvme0.format().waitdone()  # format may take long time
nvme0.timeout=10000  # recover to usual timeout configuration
```

When a command timeout happens, pynvme notifies user scripts in two ways. First, pynvme will throw a timeout warning. Second, pynvme completes (not abort) the command by itself with an all-1 completion dwords returned.

5.3.8 Asynchronous Event Request

AER is a special NVMe admin command. It is not applicable to timeout setting. In default NVMe initialization process, pynvme sends only one AER command for those unexpected AER events during the test. However, scripts can replace this default initializaiton process with which sends more AER commands or none. When one AER is completed, a warning is raised. Pynvme driver internally calls *waitdone* to reap this AER's CQE, and send one more AER command. Scripts can also give callback functions for AER commands as the usual commands.

Here is an example of AER with sanitize operations.

```
def test_aer_with_multiple_sanitize(nvme0, nvme0n1, buf):  #L8
    if nvme0.id_data(331, 328) == 0:  #L9
        pytest.skip("sanitize operation is not supported")  #L10

    logging.info("supported sanitize operation: %d" % nvme0.id_data(331, 328))

    for i in range(3):
        nvme0.sanitize().waitdone()  #L13

        # check sanitize status in log page
        with pytest.warns(UserWarning, match="AER notification is triggered"):
            nvme0.getlogpage(0x81, buf, 20).waitdone()  #L17
            while buf.data(3, 2) & 0x7 != 1:  #L18
                time.sleep(1)
                nvme0.getlogpage(0x81, buf, 20).waitdone()  #L20
                progress = buf.data(1, 0)*100//0xffff
                logging.info("%d%%" % progress)
```

AER completion is triggered when sanitize operation is finished. We can find the UserWarning for the AER notification in the test log below. The first AER command is sent by pynvme initialization process, while the remaining AER commands are sent by pynvme driver internally.

```
cmd: sudo python3 -B -m pytest --color=yes --pciaddr=3d:00.0 'scripts/test_examples.
↪py::test_aer_with_multiple_sanitize'

================================ test session starts ================================
platform linux -- Python 3.8.3, pytest-5.4.2, py-1.8.1, pluggy-0.13.1
rootdir: /home/cranechu/pynvme, inifile: pytest.ini
plugins: cov-2.9.0
collected 1 item

scripts/test_examples.py::test_aer_with_multiple_sanitize
---------------------------------- live log setup ----------------------------------
[2020-06-07 22:57:09.934] INFO script(65): setup random seed: 0xb56b1bda
---------------------------------- live log call -----------------------------------
[2020-06-07 22:57:10.334] INFO test_aer_with_multiple_sanitize(580): supported sanitize␣
↪operation: 2
[2020-06-07 22:57:13.139] INFO test_aer_with_multiple_sanitize(592): 10%
[2020-06-07 22:57:14.140] WARNING test_aer_with_multiple_sanitize(590): AER triggered,␣
↪dword0: 0x810106, status1: 0x1
[2020-06-07 22:57:14.140] INFO test_aer_with_multiple_sanitize(592): 100%
[2020-06-07 22:57:16.967] INFO test_aer_with_multiple_sanitize(592): 10%
[2020-06-07 22:57:17.968] WARNING test_aer_with_multiple_sanitize(590): AER triggered,␣
↪dword0: 0x810106, status1: 0x1
[2020-06-07 22:57:17.969] INFO test_aer_with_multiple_sanitize(592): 100%
[2020-06-07 22:57:20.777] INFO test_aer_with_multiple_sanitize(592): 10%
[2020-06-07 22:57:21.779] WARNING test_aer_with_multiple_sanitize(590): AER triggered,␣
↪dword0: 0x810106, status1: 0x1
[2020-06-07 22:57:21.780] INFO test_aer_with_multiple_sanitize(592): 100%
PASSED                                                                    [100%]
--------------------------------- live log teardown ---------------------------------
[2020-06-07 22:57:21.782] INFO script(67): test duration: 11.848 sec


================================= 1 passed in 12.30s ================================
```

5.3.9 Multiple Controllers

Users can create as many controllers as they have, even mixed PCIe devices with NVMe over TCP targets in the test.

```
nvme0 = d.Controller(b'01:00.0')
nvme1 = d.Controller(b'03:00.0')
nvme2 = d.Controller(b'10.24.48.17')
nvme3 = d.Controller(b'127.0.0.1:4420')
for n in (nvme0, nvme1, nvme2, nvme3):
    logging.info("model number: %s" % n.id_data(63, 24, str))
```

One script can be executed multiple times with different NVMe drives' BDF address in the command line.

```
laptop:~ sudo python3 -m pytest scripts/cookbook.py::test_verify_partial_namespace -s --
↪pciaddr=01:00.0
laptop:~ sudo python3 -m pytest scripts/cookbook.py::test_verify_partial_namespace -s --
↪pciaddr=02:00.0
```

5.4 Qpair

In pynvme, we combine a Submission Queue and a Completion Queue as a Qpair. The Admin *Qpair* is created within the *Controller* object implicitly. However, we need to create IO *Qpair* explicitly for IO commands. We can specify the queue depth for IO Qpairs. Scripts can delete both SQ and CQ by calling *Qpair.delete()*.

```
qpair = d.Qpair(nvme0, 10)
# ...
qpair.delete()
```

Similar to Admin Commands, we use *Qpair.waitdone()* to wait IO commands complete.

5.4.1 Interrupt

Pynvme creates the IO Completion Queues with interrupt (e.g. MSIx or MSI) enabled. However, pynvme does not check the interrupt signals on IO Qpairs. We can check interrupt signals through a set of API *Qpair.msix_*()* in the scripts. Here is an example.

```
q = d.Qpair(nvme0, 8)
q.msix_clear()
assert not q.msix_isset()
nvme0n1.read(q, buf, 0, 1) # nvme0n1 is the Namespace of nvme0
time.sleep(1)
assert q.msix_isset()
q.waitdone()
```

Interrupt is supported only for testing. Pynvme still reaps completions by polling, without checking the interrupt signals. Users can check the interrupt signal in test scripts when they need to test this function of the DUT. The interrupt of Admin Qpair of the Controller is handled in a different way by pynvme: pynvme does check the interrupt signals in each time of *Controller.waitdone()* function call. Only when the interrupt of Admin Commands is presented, pynvme would reap Admin Commands. Interrupts associated with the Admin Completion Queue cannot be delayed by coalescing (specified in 7.5 Interrupts, NVMe specification 1.4).

5.4.2 Cmdlog

Pynvme traces recent thousands of commands in the cmdlog, as well as the completion dwords, for each Qpair. API *Qpair.cmdlog()* lists the cmdlog of the Qpair. With pynvme's VSCode plugin, users can also get the cmdlog in IDE's GUI windows.

5.4.3 Notice

The Qpair object is created with a Controller object. So, users create the Qpair after the Controller. On the other side, users should free Qpair before the Controller. We recommend to use pytest and its fixture *nvme0*. It always creates controller before qpairs, and deletes controller after any qpairs.

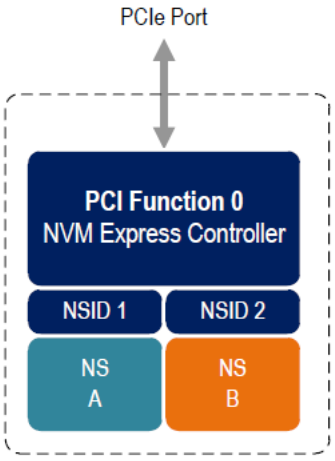
Qpair objects may be reclaimed by Python Garbage Collection, when they are not used in the script. So, qpairs would be deleted and qid would be reused. If you really want to keep qpairs alive, remember to keep their references, for example, in a list:

```
def test_create_many_qpairs(nvme0):
    qlist = []  # container to reference all qpairs
    for i in range(16):
        qlist.append(d.Qpair(nvme0, 8))
    del qlist   # delete all 16 qpairs
```

5.5 Namespace

We can create a Namespace and attach it to a Controller. It is required to close *Namespace* object when it is not used.

```
nvme0n1 = d.Namespace(nvme0, nsid=1)
# ...
nvme0n1.close()
```



For most Client NVMe SSD, we only need to use the fixture *nvme0n1* to declare the single namespace. Pynvme also supports callback functions of IO commands.

```
def write_cb(cdw0, status1):
    nvme0n1.read(qpair, read_buf, 0, 1)
nvme0n1.write(qpair, data_buf, 0, 1, cb=write_cb).waitdone(2)
```

In the above example, the waitdone() function-call reaps two commands. One is the write command, and the other is the read command which was sent in the write command's callback function. The function-call waitdone() polls commands Completion Queue, and the callback functions are called within this waitdone() function.

```
def test_invalid_io_command_0xff(nvme0n1):
    logging.info("controller0 namespace size: %d" % nvme0n1.id_data(7, 0))
```

As you see, we use API *Namespace.id_data()* to get a field of namespace identify data.

5.5.1 IO Commands

With *Namespace*, *Qpair*, and *Buffer*, we can send IO commands to NVMe devices.

```
def test_write_lba_0(nvme0, nvme0n1):
    buf = d.Buffer(512)
    qpair = d.Qpair(nvme0, 16)
    nvme0n1.write(qpair, buf, 0).waitdone()
```

Pynvme inserts LBA and calculates CRC data for each LBA to write. On the other side, pynvme checks LBA and CRC data for each LBA to read. It verifies the data integrity on the fly with ultra-low CPU cost.

5.5.2 Trim

Dataset Management (e.g. deallocate, or trim) is another commonly used IO command. It needs a prepared data buffer to specify LBA ranges to trim. Users can use API *Buffer.set_dsm_range()* for that.

```
nvme0 = d.Controller(b'01:00.0')
buf = d.Buffer(4096)
qpair = d.Qpair(nvme0, 8)
nvme0n1 = d.Namespace(nvme0)
buf.set_dsm_range(0, 0, 8)
buf.set_dsm_range(1, 8, 64)
nvme0n1.dsm(qpair, buf, 2).waitdone()
```

5.5.3 Generic Commands

We can also send any IO commands through generic commands API *Namespace.send_cmd()*:

```
nvme0n1.send_cmd(5|(1<<8), q, b, 1, 8, 0, 0)
nvme0n1.send_cmd(1|(1<<9), q, b, 1, 8, 0, 0)
q.waitdone(2)
```

It is actually a fused operation of compare and write in the above script.

5.5.4 Data Verify

We mentioned earlier that pynvme verifies data integrity on the fly of data IO. However, the controller is not responsible for checking the LBA of a Read or Write command to ensure any type of ordering between commands. See explanation from NVMe specification:

> For all commands which are not part of a fused operation (refer to section 4.12), or for which the write size is greater than AWUN, each command is processed as an independent entity without reference to other commands submitted to the same I/O Submission Queue or to commands submitted to other I/O Submission Queues. Specifically, the controller is not responsible for checking the LBA of a Read or Write command to ensure any type of ordering between commands. For example, if a Read is submitted for LBA x and there is a Write also submitted for LBA x, there is no guarantee of the order of completion for those commands (the Read may finish first or the Write may finish first). If there are ordering requirements between these commands, host software or the associated application is required to enforce that ordering above the level of the controller.

For example, when two IOWorkers write the same LBA simultaneously, the order of these writes is not defined. Similarly, in a read/write mixed IOWorker, when both read and write IO happen on the same LBA, their order is also not defined. So, it is impossible for host to determine the data content of the read.

To avoid data conflict, we can start IOWorkers one after another. Otherwise, when we have to start multiple IOWorkers in parallel, we can separate them to different LBA regions. Pynvme maintains a lock for each LBA, so within a single ioworker, pynvme can detect and resolve the LBA conflication mention above, and thus make the data verification possible and reliable in one ioworker. For those conflict-free scripts, we can enable the data verify by the fixture *verify*.

```
def test_ioworker_write_read_verify(nvme0n1, verify):
    assert verify

    nvme0n1.ioworker(io_size=8, lba_align=8, lba_random=False,
                     region_start=0, region_end=100000
                     read_percentage=0, time=2).start().close()

    nvme0n1.ioworker(io_size=8, lba_align=8, lba_random=False,
                     region_start=0, region_end=100000
                     read_percentage=100, time=2).start().close()
```

Another consideration on data verify is the memory space. During Namespace initialization, only if pynvme can allocate enough memory to hold the CRC data for each LBA, the data verify feature is enabled on this Namespace. Otherwise, the data verify feature cannot be enabled. Take a 512GB namespace for an example, it needs about 4GB memory space for CRC data. However, scripts can specify a limited scope to enable verify function with limited DRAM usage.

```
def test_verify_partial_namespace(nvme0):
    region_end=1024*1024*1024//512  # 1GB space
    nvme0n1 = d.Namespace(nvme0, 1, region_end)
    assert True == nvme0n1.verify_enable(True)

    nvme0n1.ioworker(io_size=8,
                     lba_random=True,
                     region_end=region_end,
                     read_percentage=50,
                     time=30).start().close()
```

5.6 IOWorker

It is inconvenient and expensive to send each IO command in Python scripts. Pynvme provides the low-cost high-performance *IOWorker* to send IOs in separated processes. IOWorkers make full use of multi-core CPU to improve IO test performance and stress. Scripts create the *IOWorker* object by API *Namespace.ioworker()*, and start it. Then scripts can do anything else, and finally close it to wait the IOWorker process finish and get its result data. Each IOWorker occupies one Qpair in runtime. Here is an IOWorker randomly writing 4K data for 2 seconds.

```
r = nvme0n1.ioworker(io_size=8, lba_align=8, lba_random=True,
                     read_percentage=0, time=2).start().close()
logging.info(r)
```

5.6.1 Return Data

The IOWorker result data includes these information:

item	type	explanation
io_count_read	int	total read IO in the IOWorker
io_count_nonread	int	total write and other non-read IO in the IOWorker
io_count_write	int	total write IO in the IOWorker
mseconds	int	IOWorker duration in milli-seconds
cpu_usage	int	the percentage of CPU time used by ioworker
latency_max_us	int	maximum latency in the IOWorker, unit is micro-seconds
latency_average_us	int	average latency in the IOWorker, unit is micro-seconds
error	int	error code of the IOWorker

Here are ioworker's error code:

- 0: no erro
- -1: generic error
- -2: io_size is larger than MDTS
- -3: io timeout
- -4: ioworker timeout

5.6.2 Output Parameters

To get more result of the ioworkers, we should provide output parameters.

- output_io_per_second: when an empty list is provided to output_io_per_second, ioworker will fill the io count of every seconds during the whole test.
- output_percentile_latency: when a dict, whose keys are a series of percentiles, is provided to output_percentile_latency, ioworker will fill the latency of these percentiles as the values of the dict.
- output_cmdlog_list: when a list is provided, ioworker fills the last completed commands information.

With these detail output data, we can test IOPS consistency, latency QoS, and etc. Here is an example:

```
def test_ioworker_output_io_per_latency(nvme0n1, nvme0):
    output_io_per_second = []
    output_percentile_latency = dict.fromkeys([10, 50, 90, 99, 99.9, 99.99, 99.999, 99.
↪99999])
    r = nvme0n1.ioworker(io_size=8, lba_align=8,
                         lba_random=False, qdepth=32,
                         read_percentage=0, time=10,
                         output_io_per_second=output_io_per_second,
                         output_percentile_latency=output_percentile_latency).start().
↪close()
    assert len(output_io_per_second) == 10
    assert output_percentile_latency[99.999] < output_percentile_latency[99.99999]
```

5.6.3 Concurrent

We can simultaneously start as many ioworkers as the IO Qpairs NVMe device provides.

```
with nvme0n1.ioworker(lba_start=0, io_size=8, lba_align=64,
                      lba_random=False,
                      region_start=0, region_end=1000,
                      read_percentage=0,
                      iops=0, io_count=1000, time=0,
                      qprio=0, qdepth=9), \
     nvme0n1.ioworker(lba_start=1000, io_size=8, lba_align=64,
                      lba_random=False,
                      region_start=0, region_end=1000,
                      read_percentage=0,
                      iops=0, io_count=1000, time=0,
                      qprio=0, qdepth=9), \
     nvme0n1.ioworker(lba_start=8000, io_size=8, lba_align=64,
                      lba_random=False,
                      region_start=0, region_end=1000,
                      read_percentage=0,
                      iops=0, io_count=1000, time=0,
                      qprio=0, qdepth=9), \
     nvme0n1.ioworker(lba_start=8000, io_size=8, lba_align=64,
                      lba_random=False,
                      region_start=0, region_end=1000,
                      read_percentage=0,
                      iops=0, io_count=10, time=0,
                      qprio=0, qdepth=9):
    pass
```

We can even start IOWorkers on different Namespaces in one script:

```
def test_two_namespace_ioworkers(nvme0n1, nvme0):
    nvme1 = d.Controller(b'03:00.0')
    nvme1n1 = d.Namespace(nvme1)
    with nvme0n1.ioworker(io_size=8, lba_align=16,
                          lba_random=True, qdepth=16,
                          read_percentage=0, time=100), \
         nvme1n1.ioworker(io_size=8, lba_align=16,
                          lba_random=True, qdepth=16,
                          read_percentage=0, time=100):
        pass
```

Scripts can send NVMe commands accompanied with IOWorkers. In this example, the script monitors SMART temperature value while writing NVMe device in an IOWorker.

```
def test_ioworker_with_temperature(nvme0, nvme0n1):
    smart_log = d.Buffer(512, "smart log")
    with nvme0n1.ioworker(io_size=8, lba_align=16,
                          lba_random=True, qdepth=16,
                          read_percentage=0, time=30):
        for i in range(40):
            nvme0.getlogpage(0x02, smart_log, 512).waitdone()
            ktemp = smart_log.data(2, 1)
```

(continues on next page)

(continued from previous page)

```
            logging.info("temperature: %0.2f degreeC" % k2c(ktemp))
            time.sleep(1)
```

Scripts can also make a reset or power operation when iowrokers are active. But before these kinds of operations, scripts need to wait for seconds before ioworkers are started. In these way, we can inject abnormal events into the IO workload like dirty power cycle.

```
def test_power_cycle_dirty(nvme0n1, subsystem):
    with nvme0n1.ioworker(io_size=256, lba_align=256,
                          lba_random=False, qdepth=64,
                          read_percentage=0, time=30):
        time.sleep(10)
        subsystem.power_cycle()
```

5.6.4 Performance

The performance of *IOWorker* is super high and super consistent because pynvme is an user-space driver. We can use it extensively in performance tests and stress tests. For example, we can get the 4K read IOPS in the following script.

```
@pytest.mark.parametrize("qcount", [1, 2, 4, 8, 16])
def test_ioworker_iops_multiple_queue(nvme0n1, qcount):
    l = []
    io_total = 0
    for i in range(qcount):
        a = nvme0n1.ioworker(io_size=8, lba_align=8,
                             region_start=0, region_end=256*1024*8, # 1GB space
                             lba_random=False, qdepth=16,
                             read_percentage=100, time=10).start()
        l.append(a)

    for a in l:
        r = a.close()
        io_total += (r.io_count_read+r.io_count_nonread)

    logging.info("Q %d IOPS: %dK" % (qcount, io_total/10000))
```

5.6.5 Input Parameters

IOWorker can also accurately control the IO pressure by the input parameter *iops*.

```
def test_ioworker_output_io_per_second(nvme0n1, nvme0):
    output_io_per_second = []
    nvme0n1.ioworker(io_size=8, lba_align=16,
                     lba_random=True, qdepth=16,
                     read_percentage=0, time=7,
                     iops=1234,
                     output_io_per_second=output_io_per_second).start().close()
    logging.info(output_io_per_second)
    assert len(output_io_per_second) == 7
    assert output_io_per_second[0] != 0
```

(continues on next page)

(continued from previous page)

```
    assert output_io_per_second[-1] >= 1233
    assert output_io_per_second[-1] <= 1235
```

The result of the IOWorker shows that it tests for 7 seconds, and sends 1234 IOs in each second. In this way, we can measure the latency against different IOPS pressure.

Scripts can create an ioworker up to 24 hours. We can also specify different data pattern in the IOWorker with arguments pvalue and ptype, which are the same definition as that in class Buffer.

Scripts can send different size IO in an ioworker through parameter io_size, which accepts different types of input: int, range, list, and dict.

type	explanation	example
int	fixed io size	1, send all io with size of 512 Byte.
range	a range of different io size	range(1, 8), send io size of 512, 1024, 1536, 2048, 2560, 3072, and 3584.
list	a list of different io size	[8, 16], send io size of 4096, and 8192.
dict	identify io size, as well as the ratio	{8: 2, 16: 1}, send io size of 4096 and 8192, and their IO count ratio is 2:1.

We can limit ioworker sending IO in a region specified by parameter *region_start* and *region_end*. Furthermore, we can do a further fine granularity control of IO distribution across the LBA space by parameter *distribution*. It evenly divides LBA space into 100 regions, and we specify how to identify 10000 IOs in these 100 regions.

Here is an example to display how ioworker implements JEDEC workload by these parameters:

```
def test_ioworker_jedec_workload(nvme0n1):
    # distribute 10000 IOs to 100 regions
    distribution = [1000]*5 + [200]*15 + [25]*80

    # specify different IO size and their ratio of io count
    iosz_distribution = {1: 4,
                         2: 1,
                         3: 1,
                         4: 1,
                         5: 1,
                         6: 1,
                         7: 1,
                         8: 67,
                         16: 10,
                         32: 7,
                         64: 3,
                         128: 3}

    # implement JEDEC workload in a single ioworker
    nvme0n1.ioworker(io_size=iosz_distribution,
                     lba_random=True,
                     qdepth=32,
                     distribution = distribution,
                     read_percentage=0,
                     ptype=0xbeef, pvalue=100,
                     time=10).start().close()
```

lba_random is the percentage of random IO, while *read_percentage* defines the percentage of read IO. *op_percentage*

can specify any IO opcodes as the keys of the dict, and the values are the percentage of that IO. So, we can send any kind of IO commands in ioworker, like Trim, Write Zeroes, Compare, and even VU commands.

```
def test_ioworker_op_dict_trim(nvme0n1):
    nvme0n1.ioworker(io_size=2,
                     lba_random=30,
                     op_percentage={2: 40, 9: 30, 1: 30},
                     time=2).start().close()
```

For more details on these input parameters, please refer to the lastest chapter of API documents, we well as the examples in the file: https://github.com/pynvme/pynvme/blob/master/scripts/test_examples.py

5.7 Miscellaneous

Besides functions described above, pynvme provides more facilities to make your tests more simple and powerful.

5.7.1 Power

Without any addtional equipment, pynvme can power off NVMe devices through S3 power state, and use RTC to wake it up. We implemented this process in API *Subsystem.power_cycle()*.

```
subsystem = d.Subsystem(nvme0)
subsystem.power_cycle(15)  # power off, sleep for 15 seconds, and power on
```

We can check if the hardware and OS supports S3 power state in the command line:

```
> sudo cat /sys/power/state
freeze mem disk
> sudo cat /sys/power/mem_sleep
s2idle [deep]
```

Scripts can send a notification to NVMe device before turn power off, and this is so-called clean power cycle in SSD testing:

```
subsystem = d.Subsystem(nvme0)
subsystem.shutdown_notify()
subsystem.power_cycle()
```

Pynvme also supports third-party hardware power module. Users provides the function of poweron and poweroff when creating subsystem objects, and pynvme calls them in *Subsystem.poweron()* and *Subsystem.poweroff()*.

```
def test_quarch_defined_poweron_poweroff(nvme0):
    import quarchpy

    def quarch_poweron():
        logging.info("power off by quarch")
        pwr = quarchpy.quarchDevice("SERIAL:/dev/ttyUSB0")
        pwr.sendCommand("run:power up")
        pwr.closeConnection()

    def quarch_poweroff():
        logging.info("power on by quarch")
```

(continues on next page)

(continued from previous page)

```
        pwr = quarchpy.quarchDevice("SERIAL:/dev/ttyUSB0")
        pwr.sendCommand("signal:all:source 7")
        pwr.sendCommand("run:power down")
        pwr.closeConnection()

    s = d.Subsystem(nvme0, quarch_poweron, quarch_poweroff)
```

It is required to call *Controller.reset()* after *Subsystem.power_cycle()* and *Subssytem.poweron()*.

5.7.2 Reset

Pynvme provides different ways of reset:

```
nvme0.reset()     # reset controller by its CC.EN register. We can also reset the NVMe␣
↪device as a PCIe device:

pcie.reset()      # PCIe hot reset
nvme0.reset()

subsystem.reset() # use register NSSR.NSSRC
nvme0.reset()
```

It is required to call *Controller.reset()* after *Pcie.reset()* and *Subsystem.reset()*.

5.7.3 Random Number

Before every test item, pynvme sets a different random seed to get different serie of random numbers. When user wants to reproduce the test with the identical random numbers, just manually set the random seed in the beginning of the test scripts. For example:

```
def test_ioworker_iosize_inputs(nvme0n1):
    # reproduce the test with the same random seed, and thus the identical random␣
↪numbers generated by host
    d.srand(0x58e7f337)

    nvme0n1.ioworker(io_size={1: 2, 8: 8}, time=1).start().close()
```

5.7.4 Python Space Drive

Based on SPDK, pynvme provides a high performance NVMe driver for product test. However, it lacks of flexibility to test every details defined in the NVMe Specification. Here are some of the examples:

1. Multiple SQ share one CQ. Pynvme abstracts CQ and SQ as the Qpair.
2. Non-contiguous memory for SQ and/or CQ. Pynvme always allocates contiguous memory when creating Qpairs.
3. Complicated PRP tests. Pynvme creates PRP with some reasonable limitations, but it cannot cover all corner cases in protocol tests.

In order to cover these considerations, pynvme provides an extension of **Python Space Driver** (PSD). It is an NVMe driver implemented in pure Python based on two fundamental pynvme classes:

1. DMA memory allocation abstracted by class *Buffer*.

2. PCIe configuration and memory space provided by class *Pcie*.

PSD implements NVMe data structures and operations in the module *scripts/psd.py* based on Buffer:

1. PRP: alias of Buffer, and the size is the memory page by default.
2. PRPList: maintain the list of PRP entries, which are physical addresses of *Buffer*.
3. IOSQ: create and maintain IO Submission Queue.
4. IOCQ: create and maintain IO Completion Queue.
5. SQE: submission queue entry for NVMe commands dwords.
6. CQE: completion queue entry for NVMe completion dwords.

Here is an example:

```
# import psd classes
from psd import IOCQ, IOSQ, PRP, PRPList, SQE, CQE

def test_send_cmd_2sq_1cq(nvme0):
    # 2 SQ share one CQ
    cq = IOCQ(nvme0, 1, 10, PRP())
    sq1 = IOSQ(nvme0, 1, 10, PRP(), cqid=1)
    sq2 = IOSQ(nvme0, 2, 16, PRP(), cqid=1)

    # write lba0, 16K data organized by PRPList
    write_cmd = SQE(1, 1)  # write to namespace 1
    write_cmd.prp1 = PRP() # PRP1 is a 4K page
    prp_list = PRPList()   # PRPList contains 3 pages
    prp_list[0] = PRP()
    prp_list[1] = PRP()
    prp_list[2] = PRP()
    write_cmd.prp2 = prp_list   # PRP2 points to the PRPList
    write_cmd[10] = 0           # starting LBA
    write_cmd[12] = 31          # LBA count: 32, 16K, 4 pages
    write_cmd.cid = 123;        # verify cid later

    # send write commands in both SQ
    sq1[0] = write_cmd          # fill command dwords in SQ1
    write_cmd.cid = 567;        # verify cid later
    sq2[0] = write_cmd          # fill command dwords in SQ2
    sq2.tail = 1                # ring doorbell of SQ2 first
    time.sleep(0.1)             # delay to ring SQ1,
    sq1.tail = 1                #  so command in SQ2 should comple first

    # wait for 2 command completions
    while CQE(cq[1]).p == 0: pass

    # check first cpl
    cqe = CQE(cq[0])
    assert cqe.sqid == 2
    assert cqe.sqhd == 1
    assert cqe.cid == 567

    # check second cpl
```

(continues on next page)

(continued from previous page)

```
    cqe = CQE(cq[1])
    assert cqe.sqid == 1
    assert cqe.sqhd == 1
    assert cqe.cid == 123

    # update cq head doorbell to device
    cq.head = 2

    # delete all queues
    sq1.delete()
    sq2.delete()
    cq.delete()
```

Pynvme opens quite many APIs of low-level resources, so people are free to make innovations with pynvme in user scripts.

CHAPTER

SIX

EXAMPLES

In this chapter, we will review several typical NVMe test scripts. You can find more example scripts here: https://github.com/pynvme/pynvme/blob/master/scripts/test_examples.py

6.1 Ex1: hello world

```python
# import packages
import pytest
import nvme as d

# define the test case in a python function
# list fixtures used in the parameter list
# specify the class type of the fixture, so VSCode can give more docstring online
def test_hello_world(nvme0, nvme0n1: d.Namespace):
    # create the buffers and fill data for read/write commands
    read_buf = d.Buffer(512)
    data_buf = d.Buffer(512)
    data_buf[10:21] = b'hello world'

    # create IO Qpair for read/write commands
    qpair = d.Qpair(nvme0, 16)

    # Define the callback function for write command.
    # The argument *status1* of the callback is a 16-bit
    # value, which includes the Phase-bit.
    def write_cb(cdw0, status1):
        nvme0n1.read(qpair, read_buf, 0, 1)

    # execute the write command with the callback function
    nvme0n1.write(qpair, data_buf, 0, 1, cb=write_cb)

    # wait the write command, and the read command in its callback, to be completed
    qpair.waitdone(2)

    # check the data in read buffer
    assert read_buf[10:21] == b'hello world'
```

6.2 Ex2: sanitize

```python
# import more package for GUI programming
import PySimpleGUI as sg

# define another test function, use the default buffer created by the fixture
def test_sanitize(nvme0, nvme0n1, buf):
    # check if sanitize is supported by the device
    if nvme0.id_data(331, 328) == 0:
        pytest.skip("sanitize operation is not supported")

    # start sanitize operation
    logging.info("supported sanitize operation: %d" % nvme0.id_data(331, 328))
    nvme0.sanitize().waitdone()

    # polling sanitize status in its log page
    nvme0.getlogpage(0x81, buf, 20).waitdone()
    while buf.data(3, 2) & 0x7 != 1:  # sanitize is not completed
        progress = buf.data(1, 0)*100//0xffff
        # display the progress of sanitize in a GUI window
        sg.OneLineProgressMeter('sanitize progress', progress, 100,
                                'progress', orientation='h')
        nvme0.getlogpage(0x81, buf, 20).waitdone()
        time.sleep(1)
```

6.3 Ex3: parameterized tests

```python
# create a parameter with a argument list
@pytest.mark.parametrize("qcount", [1, 2, 4, 8, 16])
def test_ioworker_iops_multiple_queue(nvme0n1, qcount):
    l = []
    io_total = 0

    # create multiple ioworkers for read performance test
    for i in range(qcount):
        a = nvme0n1.ioworker(io_size=8, lba_align=8,
                             region_start=0, region_end=256*1024*8, # 1GB space
                             lba_random=False, qdepth=16,
                             read_percentage=100, time=10).start()
        l.append(a)

    # after all ioworkers complete, calculate the IOPS performance result
    for a in l:
        r = a.close()
        io_total += (r.io_count_read+r.io_count_nonread)
    logging.info("Q %d IOPS: %dK" % (qcount, io_total/10000))
```

6.4 Ex4: upgrade and reboot the drive

```
# this test function is actually a utility to upgrade SSD firmware
def test_download_firmware(nvme0, subsystem):
    # open the firmware binary image file
    filename = sg.PopupGetFile('select the firmware binary file', 'pynvme')
    if filename:
        logging.info("To download firmware binary file: " + filename)

        # download the firmware image to SSD
        nvme0.downfw(filename)

        # power cycle the SSD to activate the upgraded firmware
        subsystem.power_cycle()

        # reset controller after power cycle
        nvme0.reset()
```

6.5 Ex5: write drive and monitor temperature

```
# a temperature calculation package
from pytemperature import k2c

def test_ioworker_with_temperature(nvme0, nvme0n1):
    smart_log = d.Buffer(512, "smart log")

    # start the ioworker for sequential writing in secondary process
    with nvme0n1.ioworker(io_size=256, lba_align=256,
                          lba_random=False, qdepth=16,
                          read_percentage=0, time=30):
        # meanwhile, monitor SMART temperature in primary process
        for i in range(40):
            nvme0.getlogpage(0x02, smart_log, 512).waitdone()

            # the K temperture from SMART log page
            ktemp = smart_log.data(2, 1)
            logging.info("temperature: %0.2f degreeC" % k2c(ktemp))
            time.sleep(1)
```

6.6 Ex6: multiple ioworkers on different namespaces and controllers

```
def test_multiple_controllers_and_namespaces():
    # address list of the devices to test
    addr_list = ['3a:00.0', '10.24.48.17']

    # create the list of controllers and namespaces
    nvme_list = [d.Controller(d.Pcie(a)) for a in addr_list]
    ns_list = [d.Namespace(n) for n in nvme_list]
```

(continues on next page)

(continued from previous page)

```
    # operations on multiple controllers
    for nvme in nvme_list:
        logging.info("device: %s" % nvme.id_data(63, 24, str))

    # start multiple ioworkers
    ioworkers = {}
    for ns in ns_list:
        a = ns.ioworker(io_size=8, lba_align=8,
                        region_start=0, region_end=256*1024*8, # 1GB space
                        lba_random=False, qdepth=16,
                        read_percentage=100, time=10).start()
        ioworkers[ns] = a

    # test results of different namespaces
    for ns in ioworkers:
        r = ioworkers[ns].close()
        io_total = (r.io_count_read+r.io_count_nonread)
        logging.info("capacity: %u, IOPS: %.3fK" %
                     (ns.id_data(7, 0), io_total/10000))
```

6.7 Ex7: format and fused operations

```
# fused operation is not directly supported by pynvme APIs
def test_fused_operations(nvme0, nvme0n1):
    # format the namespace to 4096 block size. Use Namespace.format(), instead
    # of Controller.format(), for pynvme to update namespace data in the driver.
    nvme0n1.format(4096)

    # create qpair and buffer for IO commands
    q = d.Qpair(nvme0, 10)
    b = d.Buffer()

    # separate compare and write commands
    nvme0n1.write(q, b, 8).waitdone()
    nvme0n1.compare(q, b, 8).waitdone()

    # implement fused compare and write operations with generic commands
    # Controller.send_cmd() sends admin commands,
    # and Namespace.send_cmd() here sends IO commands.
    nvme0n1.send_cmd(5|(1<<8), q, b, 1, 8, 0, 0)
    nvme0n1.send_cmd(1|(1<<9), q, b, 1, 8, 0, 0)
    q.waitdone(2)
```

CHAPTER

SEVEN

DEVELOPMENT

You are always warmly welcomed to join us to develop pynvme, as well as the test scripts, together! Here are some fundamental information for pynvme development.

7.1 Files

Pynvme makes use a bunch of 3-rd party tools. Here is a collection of key source code and configuration files and directories.

files	notes
spdk	pynvme is built on SPDK
driver_wrap.pyx	pynvme uses cython to bind python and C. All python classes are defined here.
cdriver.pxd	interface between python and C
driver.h	interface of C
driver.c	the core part of pynvme, which extends SPDK for test purpose
setup.py	cython configuration for compile
Makefile	it is a part of SPDK makefiles
driver_test.py	pytest cases for pynvme test. Users can develop more test cases for their NVMe devices.
conftest.py	predefined pytest fixtures. Find more details below.
pytest.ini	pytest runtime configuration
install.sh	build pynvme for the first time
.vscode	vscode configurations
pynvme-console-1.x.x.vsix	pynvme plugin of VSCode
requirements.txt	python packages required by pynvme
.gitlab-ci.yml	pynvme's CI configuration file for gitlab.com

7.2 Repository

Our official repository is at: https://github.com/pynvme/pynvme. We built pynvme based on SPDK/DPDK. In order to extend them with test-purpose features, we forked SPDK and DPDK respectively at:

- SPDK: https://github.com/cranechu/spdk
- DPDK: https://github.com/cranechu/dpdk

The pynvme modified code is in the branch of pynvme_1.x. We will regularly rebase pynvme modifications to the latest SPDK/DPDK stable release.

7.3 CI

We defined 3 different tests for pynvme in GitLab's CI:

1. checkin test: it is executed automatically when any new commit pushed onto master or any other branches. It should be finished in 3 minutes.
2. stress test: it is executed automatically in every weekend. Furthermore, before we merge any code onto master, we should also start and pass this stress test. We can start it here: https://gitlab.com/cranechu/pynvme/pipeline_schedules. It should be finished in 3 hours.
3. manual test: we can start any test scripts via web: https://gitlab.com/cranechu/pynvme/pipelines/new. For example, when we need to run performance tests, we can set variable key to "SCRIPT_PATH", and set its varaiable value to "scripts/performance". Then, CI starts the tests as below:

```
make test TESTS="scripts/performance"
```

We can find CI test status, logs and reports here: https://gitlab.com/cranechu/pynvme/pipelines.

The CI runner is setup on Fedora 30 Server.

7.4 Debug

1. assert: We leave a lot of assert in SPDK and pynvme code, and they are all enabled in the compile time by default.
2. log: logs include driver's log and script's log. All logs are captured/hidden by pytest in default. You can find them in file *test.log* even in the run time of tests. Users can change log levels for driver and scripts.
    1. driver: spdk_log_set_print_level in driver.c, for SPDK related logs. Set it to SPDK_LOG_DEBUG for more debug output, while it makes very heavy overhead.
    2. scripts: log_cli_level in pytest.ini, for python/pytest scripts.
3. gdb: when driver crashes or any unexpected behave observed, we can collect debug information through gdb.
    1. core dump: sudo coredumpctl debug
    2. generate core dump in dead loop: CTRL-\
    3. test within gdb (e.g. realgud with Emacs)

```
sudo gdb --args python3 -m pytest --color=yes --pciaddr=01:00.0 "driver_test.
↪py::test_create_device"
```

If you meet any problem, please report it to Issues with the *test.log* file uploaded.

7.5 Socket

Pynvme replaced Kernel's NVMe driver, so usual user space utilities (e.g. nvme-cli, iostat, etc) are not aware of pynvme and its tests. Pynvme provides an unix socket to solve these problems. Scripts and third-party tools can use this socket to get the current status of the testing device. Currently, we support 3 commands of jsonrpc call:

1. list_all_qpair: get the status of all created qpair, like its qid and current outstanding IO count.
2. get_iostat: get current IO performance of the testing device.
3. get_cmdlog: get the recent commands and completions in the specified qpair.

Here is an example of scripts access this socket in Python, but it can be accessed by any other tools (e.g. typescript which is used by pynvme's VSCode plugin).

```
def test_jsonrpc_list_qpairs(pciaddr):  #L1
    import json
    import socket  #L3

    # create the jsonrpc client
    sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
    sock.connect('/var/tmp/pynvme.sock')  #L7

    def jsonrpc_call(sock, method, params=[]):  #L9
        # create and send the command
        req = {}
        req['id'] = 1234567890
        req['jsonrpc'] = '2.0'
        req['method'] = method
        req['params'] = params
        sock.sendall(json.dumps(req).encode('ascii'))

        # receive the result
        resp = json.loads(sock.recv(4096).decode('ascii'))
        assert resp['id'] == 1234567890
        assert resp['jsonrpc'] == '2.0'
        return resp['result']

    # create controller and admin queue
    nvme0 = d.Controller(d.Pcie(pciaddr))  #L25

    result = jsonrpc_call(sock, 'list_all_qpair')  #L27
    assert len(result) == 1  #L28
```

CHAPTER

EIGHT

NVME

8.1 Buffer

```
Buffer()
```

Buffer allocates memory in DPDK, so we can get its physical address for DMA. Data in buffer is clear to 0 in initialization.

Parameters

- **size (int)**: the size (in bytes) of the buffer. Default: 4096
- **name (str)**: the name of the buffer. Default: 'buffer'
- **pvalue (int)**: data pattern value. Default: 0
- **ptype (int)**: data pattern type. Default: 0

data patterns

```
|ptype    | pvalue                                                        |
|---------|---------------------------------------------------------------|
|0        | 0 for all-zero data, 1 for all-one data                       |
|32       | 32-bit value of the repeated data pattern                     |
|0xbeef   | random data compressed rate (0: all 0; 100: fully random)     |
|others   | not supported                                                 |
```

Examples

```
>>> b = Buffer(1024, 'example')
>>> b[0] = 0x5a
>>> b[1:3] = [1, 2]
>>> b[4:] = [10, 11, 12, 13]
>>> b.dump(16)
example
00000000  5a 01 02 00 0a 0b 0c 0d  00 00 00 00 00 00 00 00   Z...............
>>> b[:8:2]
b'Z\x02\n\x0c'
>>> b.data(2) == 2
True
>>> b[2] == 2
True
>>> b.data(2, 0) == 0x02015a
```

(continues on next page)

(continued from previous page)

```
True
>>> len(b)
1024
>>> b
<buffer name: example>
>>> b[8:] = b'xyc'
example
00000000  5a 01 02 00 0a 0b 0c 0d  78 79 63 00 00 00 00 00   Z.......xyc.....
>>> b.set_dsm_range(1, 0x1234567887654321, 0xabcdef12)
>>> b.dump(64)
buffer
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000010  00 00 00 00 12 ef cd ab  21 43 65 87 78 56 34 12  ........!Ce.xV4.
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
```

8.1.1 data

```
Buffer.data(byte_end, byte_begin, type)
```

get field in the buffer. Little endian for integers.

Parameters

- **byte_end (int)**: the end byte number of this field, which is specified in NVMe spec. Included.
- **byte_begin (int)**: the begin byte number of this field, which is specified in NVMe spec. It can be omitted if begin is the same as end when the field has only 1 byte. Included. Default: None, means only get 1 byte defined in byte_end
- **type (type)**: the type of the field. It should be int or str. Default: int, convert to integer python object

Returns

(int or str): the data in the specified field

8.1.2 dump

```
Buffer.dump(size)
```

get the buffer content

Parameters

- **size (int)**: the size of the buffer to print. Default: None, means to print the whole buffer

8.1.3 phys_addr

physical address of the buffer

8.1.4 set_dsm_range

```
Buffer.set_dsm_range(index, lba, lba_count, attr)
```

set dsm ranges in the buffer, for dsm/deallocation (a.k.a. trim) commands

Parameters

- **index (int)**: the index of the dsm range to set
- **lba (int)**: the start lba of the range
- **lba_count (int)**: the lba count of the range
- **attr (int)**: context attributes of the range

8.2 Controller

```
Controller()
```

Controller class. Prefer to use fixture "nvme0" in test scripts.

Parameters

- **pcie (Pcie)**: Pcie object, or Tcp object for NVMe TCP targets
- **nvme_init_func (callable, bool, None)**: True: no nvme init process, None: default process, callable: user defined process function

Example

```
>>> n = Controller(Pcie('01:00.0'))
>>> hex(n[0])     # CAP register
'0x28030fff'
>>> hex(n[0x1c])  # CSTS register
'0x1'
>>> n.id_data(23, 4, str)
'TW0546VPLOH007A6003Y'
>>> n.supports(0x18)
False
>>> n.supports(0x80)
True
>>> id_buf = Buffer()
>>> n.identify().waitdone()
>>> id_buf.dump(64)
buffer
00000000  a4 14 4b 1b 54 57 30 35  34 36 56 50 4c 4f 48 30  ..K.TW0546VPLOH0
00000010  30 37 41 36 30 30 33 59  43 41 33 2d 38 44 32 35  07A6003YCA3-8D25
00000020  36 2d 51 31 31 20 4e 56  4d 65 20 4c 49 54 45 4f  6-Q11 NVMe LITEO
00000030  4e 20 32 35 36 47 42 20  20 20 20 20 20 20 20 20   N 256GB
>>> n.cmdlog(2)
```

(continues on next page)

(continued from previous page)

```
driver.c:1451:log_cmd_dump: *NOTICE*: dump qpair 0, latest tail in cmdlog: 1
driver.c:1462:log_cmd_dump: *NOTICE*: index 0, 2018-10-14 14:52:25.533708
nvme_qpair.c: 118:nvme_admin_qpair_print_command: *NOTICE*: IDENTIFY (06) sqid:0 cid:0␣
↪nsid:1 cdw10:00000001 cdw11:00000000
driver.c:1469:log_cmd_dump: *NOTICE*: index 0, 2018-10-14 14:52:25.534030
nvme_qpair.c: 306:nvme_qpair_print_completion: *NOTICE*: SUCCESS (00/00) sqid:0 cid:95␣
↪cdw0:0 sqhd:0142 p:1 m:0 dnr:0
driver.c:1462:log_cmd_dump: *NOTICE*: index 1, 1970-01-01 07:30:00.000000
nvme_qpair.c: 118:nvme_admin_qpair_print_command: *NOTICE*: DELETE IO SQ (00) sqid:0␣
↪cid:0 nsid:0 cdw10:00000000 cdw11:00000000
driver.c:1469:log_cmd_dump: *NOTICE*: index 1, 1970-01-01 07:30:00.000000
nvme_qpair.c: 306:nvme_qpair_print_completion: *NOTICE*: SUCCESS (00/00) sqid:0 cid:0␣
↪cdw0:0 sqhd:0000 p:0 m:0 dnr:0
```

8.2.1 abort

```
Controller.abort(cid, sqid, cb)
```

abort admin commands

Parameters

- **cid (int)**: command id of the command to be aborted
- **sqid (int)**: sq id of the command to be aborted. Default: 0, to abort the admin command
- **cb (function)**: callback function called at completion. Default: None

Returns

self (Controller)

8.2.2 aer

```
Controller.aer(cb)
```

asynchorous event request admin command.

Not suggested to use this command in scripts because driver manages to send and monitor aer commands. Scripts should register an aer callback function if it wants to handle aer, and use the fixture aer.

Parameters

- **cb (function)**: callback function called at completion. Default: None

Returns

self (Controller)

8.2.3 cap

64-bit CAP register of NVMe

8.2.4 cmdlog

```
Controller.cmdlog(count)
```

print recent commands and their completions.

Parameters

- **count (int)**: the number of commands to print. Default: 0, to print the whole cmdlog

8.2.5 cmdname

```
Controller.cmdname(opcode)
```

get the name of the admin command

Parameters

- **opcode (int)**: the opcode of the admin command

Returns
(str): the command name

8.2.6 downfw

```
Controller.downfw(filename, slot, action)
```

firmware download utility: by 4K, and activate in next reset

Parameters

- **filename (str)**: the pathname of the firmware binary file to download
- **slot (int)**: firmware slot field in the command. Default: 0, decided by device
- **cb (function)**: callback function called at completion. Default: None

Returns

8.2.7 dst

```
Controller.dst(stc, nsid, cb)
```

device self test (DST) admin command

Parameters

- **stc (int)**: selftest code (stc) field in the command
- **nsid (int)**: nsid field in the command. Default: 0xffffffff
- **cb (function)**: callback function called at completion. Default: None

Returns
self (Controller)

8.2.8 format

```
Controller.format(lbaf, ses, nsid, cb)
```

format admin command

Notice
This Controller.format only send the admin command. Use Namespace.format to maintain pynvme internal data!

Parameters

- **lbaf (int)**: lbaf (lba format) field in the command. Default: 0
- **ses (int)**: ses field in the command. Default: 0, no secure erase
- **nsid (int)**: nsid field in the command. Default: 1
- **cb (function)**: callback function called at completion. Default: None

Returns
self (Controller)

8.2.9 fw_commit

```
Controller.fw_commit(slot, action, cb)
```

firmware commit admin command

Parameters

- **slot (int)**: firmware slot field in the command
- **action (int)**: action field in the command
- **cb (function)**: callback function called at completion. Default: None

Returns
self (Controller)

8.2.10 fw_download

```
Controller.fw_download(buf, offset, size, cb)
```

firmware download admin command

Parameters

- **buf (Buffer)**: the buffer to hold the firmware data
- **offset (int)**: offset field in the command
- **size (int)**: size field in the command. Default: None, means the size of the buffer
- **cb (function)**: callback function called at completion. Default: None

Returns
self (Controller)

8.2.11 getfeatures

```
Controller.getfeatures(fid, sel, buf, cdw11, cdw12, cdw13, cdw14, cdw15,
                       cb)
```

getfeatures admin command

Parameters

- **fid (int)**: feature id
- **cdw11 (int)**: cdw11 in the command. Default: 0
- **sel (int)**: sel field in the command. Default: 0
- **buf (Buffer)**: the buffer to hold the feature data. Default: None
- **cb (function)**: callback function called at completion. Default: None

Returns
self (Controller)

8.2.12 getlogpage

```
Controller.getlogpage(lid, buf, size, offset, nsid, cb)
```

getlogpage admin command

Parameters

- **lid (int)**: Log Page Identifier
- **buf (Buffer)**: buffer to hold the log page
- **size (int)**: size (in byte) of data to get from the log page,. Default: None, means the size is the same of the buffer
- **offset (int)**: the location within a log page
- **nsid (int)**: nsid field in the command. Default: 0xffffffff
- **cb (function)**: callback function called at completion. Default: None

Returns
self (Controller)

8.2.13 id_data

```
Controller.id_data(byte_end, byte_begin, type, nsid, cns, cntid, csi,
                   nvmsetid)
```

get field in controller identify data

Parameters

- **byte_end (int)**: the end byte number of this field, which is specified in NVMe spec. Included.
- **byte_begin (int)**: the begin byte number of this field, which is specified in NVMe spec. It can be omitted if begin is the same as end when the field has only 1 byte. Included. Default: None, means only get 1 byte defined in byte_end
- **type (type)**: the type of the field. It should be int or str. Default: int, convert to integer python object

Returns
(int or str): the data in the specified field

8.2.14 identify

```
Controller.identify(buf, nsid, cns, cntid, csi, nvmsetid, cb)
```

identify admin command

Parameters

- **buf (Buffer)**: the buffer to hold the identify data
- **nsid (int)**: nsid field in the command. Default: 0
- **cns (int)**: cns field in the command. Default: 1
- **cb (function)**: callback function called at completion. Default: None

Returns
self (Controller)

8.2.15 init_adminq

```
Controller.init_adminq()
```

used by NVMe init process in scripts

8.2.16 init_ns

```
Controller.init_ns()
```

used by NVMe init process in scripts

8.2.17 init_queues

```
Controller.init_queues(cdw0)
```

used by NVMe init process in scripts

8.2.18 latest_cid

cid of latest completed command

8.2.19 latest_latency

latency of latest completed command in us

8.2.20 mdts

max data transfer bytes

8.2.21 mi_receive

```
Controller.mi_receive(opcode, dword0, dword1, buf, mtype, cb)
```

NVMe MI receive

Parameters

- **opcode (int)**: MI opcode
- **dword0 (int)**: MI request dword0
- **dword1 (int)**: MI request dword1
- **buf (Buffer)**: buffer to hold the response data
- **mtype (int)**: MI message type. Default:1, MI command set
- **cb (function)**: callback function called at completion. Default: None

Returns
self (Controller)

8.2.22 mi_send

```
Controller.mi_send(opcode, dword0, dword1, buf, mtype, cb)
```

NVMe MI Send

Parameters

- **opcode (int)**: MI opcode
- **dword0 (int)**: MI request dword0
- **dword1 (int)**: MI request dword1
- **buf (Buffer)**: buffer to hold the request data
- **mtype (int)**: MI message type. Default:1, MI command set
- **cb (function)**: callback function called at completion. Default: None

Returns
self (Controller)

8.2.23 reset

```
Controller.reset()
```

controller reset: cc.en 1 => 0 => 1

Notice
Test scripts should delete all io qpairs before reset!

8.2.24 sanitize

```
Controller.sanitize(option, pattern, cb)
```

sanitize admin command

Parameters

- **option (int)**: sanitize option field in the command
- **pattern (int)**: pattern field in the command for overwrite method. Default: 0x5aa5a55a
- **cb (function)**: callback function called at completion. Default: None

Returns
self (Controller)

8.2.25 security_receive

```
Controller.security_receive(buf, spsp, secp, nssf, size, cb)
```

admin command: security receive

Parameters

- **buf (Buffer)**: buffer of the data received
- **spsp**: SP specific 0/1, 16bit filed
- **secp**: security protocal, default 1, TCG
- **nssf**: NVMe security specific field: default 0, reserved
- **size**: size of the data to receive, default the same size of the buffer
- **cb (function)**: callback function called at cmd completion

8.2.26 security_send

```
Controller.security_send(buf, spsp, secp, nssf, size, cb)
```

admin command: security send

Parameters

- **buf (Buffer)**: buffer of the data sending
- **spsp**: SP specific 0/1, 16bit filed
- **secp**: security protocal, default 1, TCG

- **nssf**: NVMe security specific field: default 0, reserved
- **size**: size of the data to send, default the same size of the buffer
- **cb (function)**: callback function called at cmd completion

8.2.27 send_cmd

```
Controller.send_cmd(opcode, buf, nsid, cdw10, cdw11, cdw12, cdw13, cdw14,
                    cdw15, cb)
```

send generic admin commands.

This is a generic method. Scripts can use this method to send all kinds of commands, like Vendor Specific commands, and even not existed commands.

Parameters

- **opcode (int)**: operate code of the command
- **buf (Buffer)**: buffer of the command. Default: None
- **nsid (int)**: nsid field of the command. Default: 0
- **cb (function)**: callback function called at completion. Default: None

Returns
self (Controller)

8.2.28 setfeatures

```
Controller.setfeatures(fid, sv, buf, cdw11, cdw12, cdw13, cdw14, cdw15,
                       cb)
```

setfeatures admin command

Parameters

- **fid (int)**: feature id
- **cdw11 (int)**: cdw11 in the command. Default: 0
- **sv (int)**: sv field in the command. Default: 0
- **buf (Buffer)**: the buffer to hold the feature data. Default: None
- **cb (function)**: callback function called at completion. Default: None

Returns
self (Controller)

8.2.29 supports

```
Controller.supports(opcode)
```

check if the admin command is supported

Parameters

- **opcode (int)**: the opcode of the admin command

Returns
(bool): if the command is supported

8.2.30 timeout

timeout value of this controller in milli-seconds.

It is configurable by assigning new value in milli-seconds.

8.2.31 waitdone

```
Controller.waitdone(expected)
```

sync until expected admin commands completion

Notice
Do not call this function in commands callback functions.

Parameters

- **expected (int)**: expected commands to complete. Default: 1

Returns
(int): cdw0 of the last command

8.3 Namespace

```
Namespace()
```

Namespace class.

Parameters

- **nvme (Controller)**: controller where to create the queue
- **nsid (int)**: nsid of the namespace. Default 1
- **nlba_verify (long)**: number of LBAs where data verificatoin is enabled. Default 0, the whole namespace

8.3.1 capacity

bytes of namespace capacity

8.3.2 close

```
Namespace.close()
```

close to explictly release its resources instead of del

8.3.3 cmdname

```
Namespace.cmdname(opcode)
```

get the name of the IO command

Parameters

- **opcode (int)**: the opcode of the IO command

Returns
(str): the command name

8.3.4 compare

```
Namespace.compare(qpair, buf, lba, lba_count, io_flags, cb)
```

compare IO command

Notice
buf cannot be released before the command completes.

Parameters

- **qpair (Qpair)**: use the qpair to send this command
- **buf (Buffer)**: the data buffer of the command, meta data is not supported.
- **lba (int)**: the starting lba address, 64 bits
- **lba_count (int)**: the lba count of this command, 16 bits. Default: 1
- **io_flags (int)**: io flags defined in NVMe specification, 16 bits. Default: 0
- **cb (function)**: callback function called at completion. Default: None

Returns
qpair (Qpair): the qpair used to send this command, for ease of chained call

Raises

- `SystemError`: the command fails

8.3.5 dsm

```
Namespace.dsm(qpair, buf, range_count, attribute, cb)
```

data-set management IO command

Notice
buf cannot be released before the command completes.

Parameters

- **qpair (Qpair)**: use the qpair to send this command
- **buf (Buffer)**: the buffer of the lba ranges. Use buffer.set_dsm_range to prepare the buffer.
- **range_count (int)**: the count of lba ranges in the buffer
- **attribute (int)**: attribute field of the command. Default: 0x4, as deallocation/trim
- **cb (function)**: callback function called at completion. Default: None

Returns
qpair (Qpair): the qpair used to send this command, for ease of chained call

Raises

- `SystemError`: the command fails

8.3.6 flush

```
Namespace.flush(qpair, cb)
```

flush IO command

Parameters

- **qpair (Qpair)**: use the qpair to send this command
- **cb (function)**: callback function called at completion. Default: None

Returns
qpair (Qpair): the qpair used to send this command, for ease of chained call

Raises

- `SystemError`: the command fails

8.3.7 format

```
Namespace.format(data_size, meta_size, ses)
```

change the format of this namespace

Notice
Namespace.format() not only sends the admin command, but also updates driver to activate new format immediately. Recommend to use this API to do format. Close and re-create namespace when lba format is changed.

Parameters

- **data_size (int)**: data size. Default: 512

- **meta_size (int)**: meta data size. Default: 0
- **ses (int)**: ses field in the command. Default: 0, no secure erase

Returns
int: cdw0 of the format admin command

8.3.8 get_lba_format

```
Namespace.get_lba_format(data_size, meta_size)
```

find the lba format by its data size and meta data size

Parameters

- **data_size (int)**: data size. Default: 512
- **meta_size (int)**: meta data size. Default: 0

Returns
(int or None): the lba format has the specified data size and meta data size

8.3.9 id_data

```
Namespace.id_data(byte_end, byte_begin, type, cns, csi, cntid)
```

get field in namespace identify data

Parameters

- **byte_end (int)**: the end byte number of this field, which is specified in NVMe spec. Included.
- **byte_begin (int)**: the begin byte number of this field, which is specified in NVMe spec. It can be omitted if begin is the same as end when the field has only 1 byte. Included. Default: None, means only get 1 byte defined in byte_end
- **type (type)**: the type of the field. It should be int or str. Default: int, convert to integer python object

Returns
(int or str): the data in the specified field

8.3.10 ioworker

```
Namespace.ioworker(io_size, lba_step, lba_align, lba_random,
                   read_percentage, op_percentage, time, qdepth,
                   region_start, region_end, iops, io_count, lba_start,
                   qprio, distribution, ptype, pvalue, io_sequence,
                   fw_debug, output_io_per_second,
                   output_percentile_latency, output_cmdlog_list)
```

workers sending different read/write IO on different CPU cores.

User defines IO characteristics in parameters, and then the ioworker executes without user intervesion, until the test is completed. IOWorker returns some statistic data at last.

User can start multiple IOWorkers, and they will be binded to different CPU cores. Each IOWorker creates its own Qpair, so active IOWorker counts is limited by maximum IO queues that DUT can provide.

Each ioworker can run upto 24 hours.

Parameters

- **io_size (short, range, list, dict)**: IO size, unit is LBA. It can be a fixed size, or a range or list of size, or specify ratio in the dict if they are not evenly distributed. 1base. Default: 8, 4K
- **lba_step (short)**: valid only for sequential read/write, jump to next LBA by the step. Default: None, same as io_size, continous IO.
- **lba_align (short)**: IO alignment, unit is LBA. Default: None: means 1 lba.
- **lba_random (int, bool)**: percentage of radom io, or True if sending IO with all random starting LBA. Default: True
- **read_percentage (int)**: sending read/write mixed IO, 0 means write only, 100 means read only. Default: 100. Obsoloted by op_percentage
- **op_percentage (dict)**: opcode of commands sent in ioworker, and their percentage. Output: real io counts sent in ioworker. Default: None, fall back to read_percentage
- **time (int)**: specified maximum time of the IOWorker in seconds, up to 1000*3600. Default:0, means no limit
- **qdepth (int)**: queue depth of the Qpair created by the IOWorker, up to 1024. 1base value. Default: 64
- **region_start (long)**: sending IO in the specified LBA region, start. Default: 0
- **region_end (long)**: sending IO in the specified LBA region, end but not include. Default: 0xffff_ffff_ffff_ffff
- **iops (int)**: specified maximum IOPS. IOWorker throttles the sending IO speed. Default: 0, means no limit
- **io_count (long)**: specified maximum IO counts to send. Default: 0, means no limit
- **lba_start (long)**: the LBA address of the first command. Default: 0, means start from region_start
- **qprio (int)**: SQ priority. Default: 0, as Round Robin arbitration
- **distribution (list(int))**: distribute 10,000 IO to 100 sections. Default: None
- **pvalue (int)**: data pattern value. Refer to data pattern in class `Buffer`. Default: 100 (100%)
- **ptype (int)**: data pattern type. Refer to data pattern in class `Buffer`. Default: 0xbeef (random data)
- **io_sequence (list)**: io sequence of captured trace from real workload. Ignore other input parameters when io_sequence is given. Default: None
- **output_io_per_second (list)**: list to hold the output data of io_per_second. Default: None, not to collect the data
- **output_percentile_latency (dict)**: dict of io counter on different percentile latency. Dict key is the percentage, and the value is the latency in micro-second. Default: None, not to collect the data
- **output_cmdlog_list (list)**: list of dwords of lastest commands completed in the ioworker. Default: None, not to collect the data

Returns

ioworker instance

8.3.11 nsid

id of the namespace

8.3.12 read

```
Namespace.read(qpair, buf, lba, lba_count, io_flags, dword13, dword14,
               dword15, cb)
```

read IO command

Notice
buf cannot be released before the command completes.

Parameters

- **qpair (Qpair)**: use the qpair to send this command
- **buf (Buffer)**: the data buffer of the command, meta data is not supported.
- **lba (int)**: the starting lba address, 64 bits
- **lba_count (int)**: the lba count of this command, 16 bits. Default: 1
- **io_flags (int)**: io flags defined in NVMe specification, 16 bits. Default: 0
- **dword13 (int)**: command SQE dword13
- **dword14 (int)**: command SQE dword14
- **dword15 (int)**: command SQE dword15
- **cb (function)**: callback function called at completion. Default: None

Returns
qpair (Qpair): the qpair used to send this command, for ease of chained call

8.3.13 send_cmd

```
Namespace.send_cmd(opcode, qpair, buf, nsid, cdw10, cdw11, cdw12, cdw13,
                   cdw14, cdw15, cb)
```

send generic IO commands.

This is a generic method. Scripts can use this method to send all kinds of commands, like Vendor Specific commands, and even not existed commands.

Parameters

- **opcode (int)**: operate code of the command
- **qpair (Qpair)**: qpair used to send this command
- **buf (Buffer)**: buffer of the command. Default: None
- **nsid (int)**: nsid field of the command. Default: 0
- **cdw1x (int)**: command SQE dword10 - dword15
- **cb (function)**: callback function called at completion. Default: None

Returns
qpair (Qpair): the qpair used to send this command, for ease of chained call

8.3.14 supports

```
Namespace.supports(opcode)
```

check if the IO command is supported

Parameters

- **opcode (int)**: the opcode of the IO command

Returns
(bool): if the command is supported

8.3.15 verify

```
Namespace.verify(qpair, lba, lba_count, io_flags, cb)
```

verify IO command

Parameters

- **qpair (Qpair)**: use the qpair to send this command
- **lba (int)**: the starting lba address, 64 bits
- **lba_count (int)**: the lba count of this command, 16 bits. Default: 1
- **io_flags (int)**: io flags defined in NVMe specification, 16 bits. Default: 0
- **cb (function)**: callback function called at completion. Default: None

Returns
qpair (Qpair): the qpair used to send this command, for ease of chained call

Raises

- `SystemError`: the read command fails

8.3.16 verify_enable

```
Namespace.verify_enable(enable)
```

enable or disable the inline verify function of the namespace

Parameters

- **enable (bool)**: enable or disable the verify function

Returns
(bool): if it is enabled successfully

8.3.17 write

```
Namespace.write(qpair, buf, lba, lba_count, io_flags, dword13, dword14,
                dword15, cb)
```

write IO command

Notice
buf cannot be released before the command completes.

Parameters

- **qpair (Qpair)**: use the qpair to send this command
- **buf (Buffer)**: the data buffer of the write command, meta data is not supported.
- **lba (int)**: the starting lba address, 64 bits
- **lba_count (int)**: the lba count of this command, 16 bits
- **io_flags (int)**: io flags defined in NVMe specification, 16 bits. Default: 0
- **dword13 (int)**: command SQE dword13
- **dword14 (int)**: command SQE dword14
- **dword15 (int)**: command SQE dword15
- **cb (function)**: callback function called at completion. Default: None

Returns
qpair (Qpair): the qpair used to send this command, for ease of chained call

8.3.18 write_uncorrectable

```
Namespace.write_uncorrectable(qpair, lba, lba_count, cb)
```

write uncorrectable IO command

Parameters

- **qpair (Qpair)**: use the qpair to send this command
- **lba (int)**: the starting lba address, 64 bits
- **lba_count (int)**: the lba count of this command, 16 bits. Default: 1
- **cb (function)**: callback function called at completion. Default: None

Returns
qpair (Qpair): the qpair used to send this command, for ease of chained call

Raises

- `SystemError`: the command fails

8.3.19 write_zeroes

```
Namespace.write_zeroes(qpair, lba, lba_count, io_flags, cb)
```

write zeroes IO command

Parameters

- **qpair (Qpair)**: use the qpair to send this command
- **lba (int)**: the starting lba address, 64 bits
- **lba_count (int)**: the lba count of this command, 16 bits. Default: 1
- **io_flags (int)**: io flags defined in NVMe specification, 16 bits. Default: 0
- **cb (function)**: callback function called at completion. Default: None

Returns

qpair (Qpair): the qpair used to send this command, for ease of chained call

Raises

- `SystemError`: the command fails

8.4 Pcie

```
Pcie()
```

Pcie class to access PCIe configuration and memory space

Parameters

- **addr (str)**: BDF address of PCIe device

8.4.1 aspm

config new ASPM Control:

Parameters

- **control**: ASPM control field in Link Control register:
- **b00**: ASPM is disabled
- **b01**: L0s
- **b10**: L1
- **b11**: L0s and L1

8.4.2 cap_offset

```
Pcie.cap_offset(cap_id)
```

get the offset of a capability

Parameters

- **cap_id (int)**: capability id

Returns

(int): the offset of the register, or None if the capability is not existed

8.4.3 close

```
Pcie.close()
```

close to explictly release its resources instead of del

8.4.4 power_state

config new power state:

Parameters

- **state**: new state of the PCIe device:
- **0**: D0
- **1**: D1
- **2**: D2
- **3**: D3hot

8.4.5 register

```
Pcie.register(offset, byte_count)
```

access registers in pcie config space, and get its integer value.

Parameters

- **offset (int)**: the offset (in bytes) of the register in the config space
- **byte_count (int)**: the size (in bytes) of the register. Default: 4, dword

Returns

(int): the value of the register

8.4.6 reset

```
Pcie.reset(rst_fn)
```

reset this pcie device with hot reset

Notice
call Controller.reset() to re-initialize controller after this reset

8.5 Qpair

```
Qpair()
```

Qpair class. IO SQ and CQ are combinded as qpairs.

Parameters

- **nvme (Controller)**: controller where to create the queue
- **depth (int)**: SQ/CQ queue depth
- **prio (int)**: when Weighted Round Robin is enabled, specify SQ priority here
- **ien (bool)**: interrupt enabled. Default: True
- **iv (short)**: interrupt vector. Default: 0xffff, choose by driver

8.5.1 cmdlog

```
Qpair.cmdlog(count)
```

print recent IO commands and their completions in this qpair.

Parameters

- **count (int)**: the number of commands to print. Default: 0, to print the whole cmdlog

8.5.2 delete

```
Qpair.delete()
```

delete qpair's SQ and CQ

8.5.3 latest_cid

cid of latest completed command

8.5.4 latest_latency

latency of latest completed command in us

8.5.5 sqid

submission queue id in this qpair

8.5.6 waitdone

```
Qpair.waitdone(expected)
```

sync until expected IO commands completion

Notice
Do not call this function in commands callback functions.

Parameters

- **expected (int)**: expected commands to complete. Default: 1

Returns
(int): cdw0 of the last command

8.6 srand

```
srand(seed)
```

manually setup random seed

Parameters

- **seed (int)**: the seed to setup for both python and C library

8.7 Subsystem

```
Subsystem()
```

Subsystem class. Prefer to use fixture "subsystem" in test scripts.

Parameters

- **nvme (Controller)**: the nvme controller object of that subsystem
- **poweron_cb (func)**: callback of poweron function
- **poweroff_cb (func)**: callback of poweroff function

8.7.1 power_cycle

```
Subsystem.power_cycle(sec)
```

power off and on in seconds

Notice
call Controller.reset() to re-initialize controller after this power cycle

Parameters

- **sec (int)**: the seconds between power off and power on

8.7.2 poweroff

```
Subsystem.poweroff()
```

power off the device by the poweroff function provided in Subsystem initialization

8.7.3 poweron

```
Subsystem.poweron()
```

power on the device by the poweron function provided in Subsystem initialization

Notice
call Controller.reset() to re-initialize controller after this power on

8.7.4 reset

```
Subsystem.reset()
```

reset the nvme subsystem through register nssr.nssrc

Notice
call Controller.reset() to re-initialize controller after this reset

8.7.5 shutdown_notify

```
Subsystem.shutdown_notify(abrupt)
```

notify nvme subsystem a shutdown event through register cc.shn

Parameters

- **abrupt (bool)**: it will be an abrupt shutdown (return immediately) or clean shutdown (wait shutdown completely)

8.8 Tcp

```
Tcp()
```

Tcp class for NVMe TCP target

Parameters

- **addr (str)**: IP address of TCP target
- **port (int)**: the port number of TCP target. Default: 4420