# Preface

This document describes the high level approach to testing NVME compliant hardware devices according to specification compliance.

The tnvme directory contains the test application. The dnvme contains the test application driver. The qemu directory contains a virtualization layer to allow virtualized OSes to communicate with simulated NVME devices. The simulated world of NVME hardware is similar to real world hardware but the former does have tradeoffs. QEMU will only export at most a single NVME hardware device, whereas the real world can export many more.

There are also other limitations between the simulated world and the real world. For example, a request from dnvme will be blocked until the virtualized hardware executes all of the IO’s submitted in all SQ’s. This prevents the driver from being released and submitting new IO’s in other SQ’s. This limitation really means that certain tests and real world interaction is not attainable. For example filling up a 2nd SQ while the hardware is processing another SQ to test priority execution assigned to specific Q’s will be unattainable. This deficiency has its roots embedded in how QEMU is architected. It fails to be thread safe which could allow solutions to the aforementioned issue.

# Special Device Files

The NVME hardware devices will be exported into the file system as /dev/qnvmeX, where X = {0…n}. The test app can peruse the /dev directory to find all testable devices within a system. The Linux API open command can be used to allow issuing IOCTL’s of value into the kernel. These IO’s allow setting up the kernel level test environment to enable commands to be issued and weird scenarios to be tested.

# Strategy

The strategy of design is to allow invocations to test against specific revisions of the NVME specification. This becomes easier if more of the logic resides in user space, and this becomes possible when the logic in the dnvme, the test application driver, has generalized its interface correctly. The idea is to create building blocks in the dnvme such that they can be arranged to form a multitude of test cases within user space. Most of the test case logic will be pulled from the kernel and into user space to allow easier maintenance and the addition of future test cases.

# Overview

The following diagram illustrates the high level design concepts for tnvme, dnvme and QEMU’s NVME simulated hardware device.



# Revisions

QEMU will always attempt to conform to the latest released NVME speciation for hardware compliance. Thus the git repository will have a master branch whereby the latest work will be performed. The tnvme and dnvme will not have the luxury of such assumptions. It is expected that a test suite will need to target a specific release of the NVME specification due to removal of reserved bits, added features, and many other reasons. Therefore the test application will be written in C++ to take advantage of the polymorphic nature of running against various releases of the specification. The dnvme will be written in ANSI C. With more of the test logic being generalize within dnvme, it will be easier to address specification revision differences when compared to the prior test design.

# tnvme

The test application will attempt to place the test case description within the test itself to help promote upkeep of the information. Doxygen will be used to document the hierarchical nature of C++ to aid in understanding the design infrastructure. The best way to become superficially familiar with tnvme is the start it by requesting help of the command line, tnvme –h.

# dnvme

The test application driver can be best understood by thinking of it as a large kernel library, whereby user space test cases will choose the various mechanisms exported by IOCTL’s. tnvme can wire those IOCTL’s together to create various test scenarios. Of course not all testing will be able to utilize this methodology, thus specialized test cases which need to be close to the hardware will most certainly exist within dnvme. This hybrid approach should provide the greatest code reuse and flexibility in adding future test cases. The first approach, when adding new test logic, should be to ask whether it can be accomplished in user space and then design generalized functions within dnvme to support a tnvme user space test.

# Qemu Architecture

The following is old and no longer appies. It will be updated.

Designed a threaded architecture for QEMU, so that it would match the real hardware performance. Inside Qemu, each I/O admin command coming from the queue (Doorbell register writes) as wells as CQ register writes, spawn a new thread for their execution. But an internal hardware state is maintained which is protected by set of locks. Thus the handling of every command is done parallel but the execution of command and updating of state is serialized.

A big advantage of the threaded architecture is that the driver does not have to wait for the command to complete and thereby does not block the app in the user space. Also in the real world, the hardware wouldn’t block the driver for completion of a command and updating of state. Thus the threaded architecture was proposed. Moreover using the threaded architecture also improves the handling for asynchronous requests like controller resets. Whenever asynchronous requests are made, the threaded model signals the outstanding I/O admin command threads to exit and thus increases the responsiveness to asynchronous requests.

The following diagram illustrates the high level design concepts for QEMU’s emulated hardware device.

