Software Development Kit (SDK) User Guide

Software-Enabled Flash™ (SEF)

Version: 1.00

SEF-SDK-01-00

LEGAL DISCLAIMER

THIS DOCUMENT AND THE INFORMATION CONTAINED HEREIN IS PROVIDED ON AN "AS IS" BASIS. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, THE SOFTWARE-ENABLED FLASH PROJECT, THE LINUX FOUNDATION, AND THE CONTRIBUTORS TO THIS DOCUMENT HEREBY DISCLAIM ALL REPRESENTATIONS, WARRANTIES AND/OR COVENANTS, EITHER EXPRESS OR IMPLIED, STATUTORY OR AT COMMON LAW, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, VALIDITY, AND/OR NONINFRINGEMENT.

All product names, trademarks, registered trademarks, and/or servicemarks may be claimed as the property of their respective owners.

DEFINITIONS AND CLARIFICATIONS

Definition of capacity: we define a megabyte (MB) as 1,000,000 bytes, a gigabyte (GB) as 1,000,000,000 bytes and a terabyte (TB) as 1,000,000,000,000 bytes. A computer operating system, however, reports storage capacity using powers of 2 for the definition of 1GB $= 2^{30} =$ 1,073,741,824 bytes and therefore shows less storage capacity. Available storage capacity (including examples of various media files) will vary based on file size, formatting, settings, software and operating system, such as Microsoft Operating System and/or pre-installed software applications, or media content. Actual formatted capacity may vary.

KiB: A kibibyte (KiB) means 2^{10}, or 1,024 bytes, a mebibyte (MiB) means 2^{20}, or 1,048,576 bytes, and a gibibyte (GiB) means 2^{30}, or 1,073,741,824 bytes.

Read and write speed may vary depending on the host device, read and write conditions, and file size.

TRADEMARKS

NVM Express and NVMe are registered or unregistered marks of NVM Express, Inc. in the United States and other countries.

PCI Express, PCIe and PCI are trademarks or registered trademarks of PCI-SIG.

Linux is a registered trademark of Linus Torvalds in the U.S. and other countries.

Ubuntu is a trademark of Canonical Ltd.

Other company names, product names and service names may be trademarks of third-party companies.

Contents

1 Revision History

Version	Date	Description of change(s)
1.00	2023.11.30	Initial version of the document

Part I

Getting Started

2 Intended Audience

This document is targeted toward engineers responsible for developing storage infrastructure and applications that directly access and manage storage. The document assumes that the reader is familiar with the Linux™ operating system, the C programming language, the basic architecture of solid state drives, the architecture of device drivers, flash translation layers (FTLs), and flash media.

3 Overview

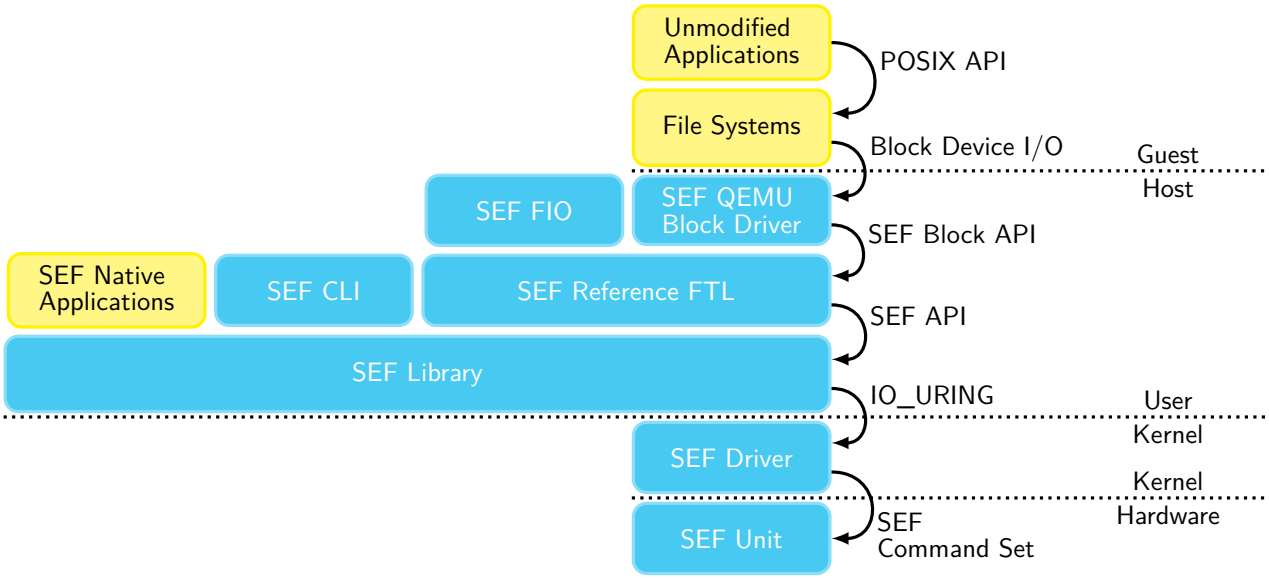
The Software-Enabled Flash™ (SEF) Software Development Kit (SDK) is distributed in source code format and consists of the following major components:

- SEF API: The SEF API is the Application Programming Interface, it represents a contract between an application and the implementation. The implementation is free to change as long as the API remains constant.
- SEF Library: The SEF Library is a user mode library with a set of functions that may be used to implement the SEF API, translating the functions defined in the API into the actual commands issued to the device. To do so, the library also implements many "helper functions" that are not defined by the API, but are of general utility in either implementing the API or working around device specific restrictions/limitations. There is no contract between the library and the application for anything that is not a part of the API, and these helper functions are free to change without affecting applications. An example of a helper function is a function that manipulates or copies a scatter gather list, assisting in the cases where a single command from the perspective of the API needs to be broken up into multiple device commands due to device specific limitations. The SEF Library encompasses more than just the API; the SEF Library provides quite a bit of information on how to use and interact with the SEF Command Set.
- SEF Reference FTL: A user mode FTL built on the SEF Library. It exposes the asynchronous SEF Block API.
- SEF CLI: A command line tool for listing and configuring most aspects of an SEF Unit. It supports multiple target types, such as SEF Units, Virtual Devices, QoS Domains, and SEF FTL block devices.
- SEF FIO: Provides an SEF FTL block device `ioengine` to the Linux I/O testing tool fio.
- SEF QEMU Block Driver: Provides three SEF-backed block device types to QEMU: a virtual block device, a block NVMe™ device, and a ZNS NVMe device.
- SEF Kernel Driver: Builds the sef.ko module

- SEF Linux: Includes the hooks required to use an SEF Unit
- SEF NVME-CLI: Provides support for SEF subcommands.

Figure 3.1 shows how the components fit together.

Figure 3.1: SEF Components Diagram



The goal of the SEF SDK is to provide a software set that enables the immediate evaluation of SEF technology as well as to illustrate the use of the SEF API and SEF Library to build real world storage applications. **Although the SEF SDK has been extensively tested, it is not intended for commercial use in a production environment.** The SEF API is documented in the *Software Enabled Flash™ (SEF) API Specification*.

This document is NOT intended to cover the design and implementation of the SEF Library, modifications to the NVMe device driver, the SEF Unit driver or the SEF utility programs. The source code for all of these components is available as part of the SDK.

4 Definitions and Acronyms

Table 4.1: Definitions and Acronyms

Terms/Acronyms	Definition
Software-Enabled Flash™ (SEF)	A flash memory-based storage hardware platform that is driven by software. Pronounced as ess ē ef.
SEF Unit	A PCIe® flash memory storage device. Contains one or more flash memory dies and provides flash memory service functions. The SEF Unit command set consists of a subset of the NVMe command set with extensions.
Flash Translation Layer (FTL)	A mapping of Logical Block Addresses (LBA) to flash memory addresses providing a block based API on top of a flash memory API.
Virtual Device (VD)	A set of flash memory dies. A Virtual Device occupies one or more flash memory dies and provides one or more QoS Domains and wear leveling service between QoS Domains. Flash memory dies can only be assigned to one Virtual Device; they are never shared between Virtual Devices. Virtual devices provide true hardware-based isolation.
Pseudo Single-Level Cell (pSLC)	SEF Units may optionally support programming flash memory as if it's SLC for increased endurance and performance.
QoS Domain (QD)	A logical construct exposed to the host and enumerated as an SEF Unit node. QoS Domains are created within a single Virtual Device and draw Super Blocks from a common pool within the Virtual Device. One or more QoS Domains may be created within a single Virtual Device. QoS Domains provide software-based isolation, impose quotas on capacity, and are comprised of a set of Super Blocks within a Virtual Device. Super blocks are not shared between QoS Domains. Read/write commands are issued to a specific QoS Domain.

Super Block	A set of flash memory blocks spanning all of the dies in a Virtual Device. All flash memory blocks in a Super Block can be programmed and read in parallel.
Logical Block Address (LBA)	Represents one component of an optional user-visible addressing interface implemented by an FTL.
ADU	Atomic data unit. An SEF-defined internal representation of abstract storage that is the minimum read/write quantum (analogous to the block size of a traditional block device). An SEF Unit may support multiple ADU sizes, and the ADU size is specified when creating a QoS domain. The minimum ADU size is 4096 bytes.
User Address (UA)	Eight bytes of metadata that is stored with an ADU. It is typically used to record the LBA of the ADU. It's auto incremented when writing to multiple ADUs and can be verified when reading. A portion of the UA is a user defined tag of 24-bits for the application's use.
Placement ID	A placement ID is used when writing data to a QoS Domain. It's used to group data of a similar lifetime together. ADUs written with the same placement ID are stored in the same Super Blocks.
Root Pointer	Provides a bootstrapping mechanism to retrieve metadata from a QoS domain.
API	An Application Programming Interface is a set of rules and protocols that allows different software applications to communicate. The API does not indicate the implementation and various libraries can implement an API differently.

5 | Security Model

NOTE: This section will be added in a future release.

6 Build Instructions

6.1 General Build Requirements

This chapter will cover how to get the Software Development Kit (SDK) source, build and install different components. Specific requirements and dependencies are listed for each component along with the install command. The Software-Enabled Flash™ (SEF) SDK components and tools were designed to run on 32-bit or 64-bit systems and are endian agnostic.

NOTE: The build instructions below match how the SDK is built on Ubuntu™ 20.04

6.2 Source Code

The SEF SDK will soon be available to clone from GitHub. The following command can be used to clone with SEF SDK:

```
$ git clone https://github.com/SoftwareEnabledFlash/SEF-SDK.git
```

Note: Pre-release distributions will be distributed as compressed tar images which may be extracted as follows:

```
$ tar -xzvf SEFSDK.tar.gz
```

6.3 Linux™ Kernel

SEF SDK requires a modified Linux Kernel 5.19. This section covers how to download, modify, build and install the required kernel. To learn more about SEF Linux and how it works refer to Chapter 7.

Note: Cloning the Linux repository as part of the build will take several minutes.

Before getting started, ensure the following prerequisites have been installed:

- libncurses-dev
- gawk

- flex
- bison
- openssl
- libssl-dev
- libudev-dev
- dwarves
- zstd
- libelf-dev
- libpci-dev
- libiberty-dev
- autoconf
- dkms
- bc

The following command can be used to install these dependencies:

```
$ sudo apt update
$ sudo apt install -y libncurses-dev gawk flex bison openssl \
 libssl-dev dwarves zstd libelf-dev libpci-dev libiberty-dev \
 autoconf dkms bc
```

Because this is a custom kernel, you may receive an error about a bad signature, which requires Secure Boot disabled in BIOS.

We provide a kioxia.config built from a server install of Ubuntu™ 20.04's config. However, you can create your own .config file with the 'make menuconfig' command.

To build and install the SEF Linux, issue the following commands:

```
$ cd <path to SEF_SDK>/linux
$ ./apply_patch.sh
$ cd official-linux
$ cp ./arch/x86/configs/kioxia.config .config
$ make -j`nproc`
$ sudo make -j`nproc` INSTALL_MOD_STRIP=1 modules_install
$ sudo make install
```

```
$ sudo reboot now
```

6.4 SEF Driver

It is important to note that the SEF Driver and the Linux kernel must be built on the same system to ensure compatibility. The sef.ko kernel module requires a system running SEF Linux in order to access the SEF Unit. To learn more about SEF Driver and how it works refer to Chapter 7.

Note: If the driver fails to load with a message regarding unknown symbols, it is because nvme-core.ko and nvme_ko were not previously loaded by the system. Also, please check if your SEF Unit was properly installed using lspci command.

Before getting started, ensure the following prerequisites have been installed:

- build-essential
- libudev-dev
- cmake

The following command can be used to install these dependencies:

```
$ sudo apt install -y build-essential libudev-dev cmake
```

To build and install the SEF Driver so that it will load across reboots, first build Linux, then issue the following commands:

```
$ cd <path to SEF_SDK>
$ export SEF_KERNEL_SRC=/<path to SEF_SDK>/linux/official-linux/
$ sudo SEF_KERNEL_SRC=${SEF_KERNEL_SRC} ./build.sh -o driver -i
$ sudo modprobe sef
```

6.5 SEF Library, Flash Translation Layer (FTL) and Command Line Interface (CLI)

The SEF SDK uses the cmake and make tool set in order to prepare, build, and install the SDK components. The Software Development Kit can be downloaded and installed in one easy step. The master SEF SDK project includes all the SDK modules.

To learn more about SEF Library, FTL, and CLI and how they work, refer to Chapter 8, Chapter 9, Chapter 18 respectively.

Before getting started, ensure the following prerequisites have been installed:

- cmake
- python3-dev (python3.6m or higher)

The following command can be used to install these dependencies:

```
$ sudo apt install -y cmake python3-dev
```

To build the SEF Library, FTL, and CLI, issue the following commands:

```
$ cd <path to SEF_SDK>
$ sudo ./build.sh -i
```

As part of the installation, the process will automatically install the CLI's man page and auto-complete script.

6.6 FIO

To learn more about SEF FIO Engine and how it works refer to Chapter 21.

In addition to the dependencies of FIO, ensure the following SEF prerequisites have been installed:

- SEF Reference FTL library
- SEF library
- Header and libraries for pthread

Before building FIO, it should be configured. The SEF Engine is enabled by default. To configure, build, and install FIO, issue the following commands:

```
$ cd <path to SEF_SDK>/fio
$ ./configure
$ make
$ sudo make install
```

6.7 SEF QEMU

Although `QEMU` can be used to test software that has not be changed to support SEF natively, it is not required to use or test SEF. To learn more about SEF QEMU and how it works refer to Chapter 22.

In addition to the dependencies of QEMU, ensure the following SEF prerequisites have been installed:

- SEF Reference FTL library

- SEF library
- Header and libraries for pthread
- git
- libglib2.0-dev
- libfdt-dev
- libpixman-1-dev
- zlib1g-dev
- ninja-build
- pkg-config

The following command can be used to install these dependencies:

```
$ sudo apt install -y git libpthread-stubs0-dev libglib2.0-dev \
 libfdt-dev libpixman-1-dev zlib1g-dev ninja-build pkg-config
```

SEF QEMU supports an SEF-backed virtual block device, block NVMe device, ZNS NVMe device or FDP.

To build and install the SEF QEMU, issue the following commands:

```
$ cd <path to SEF_SDK>
$ sudo -v; ./build.sh -o qemu -e -i
```

6.8 NVME-CLI

Although `NVME-CLI` is a powerful tool that can be used to interact with and configure an SEF Unit, it is not required. To learn more about SEF NVME-CLI and how it works refer to Chapter 23.

Ensure the following prerequisites have been installed:

- libjson-c-dev
- pthread
- meson
- ninja-build

The following command can be used to install these dependencies:

```
$ sudo apt install -y meson ninja-build
```

To build and install the NVME-CLI, issue the following commands:

```
$ cd <path to SEF_SDK>/nvme-cli
$ ./apply_patch.sh
$ cd official-nvme
$ meson .build
$ ninja -C .build
$ sudo make install
```

6.9 Validating Installation

Once the kernel, driver, libraries, and applications are built and installed, the following sequence of commands can verify communications with an installed SEF Unit.

The following commands check if the SEF Unit can be detected by listing all SEF Units, create a Virtual Device, create a QoS Domain, configure the FTL, and runs a short FIO job. The script assumes the SEF Unit is installed at index 0. The commands use the `--force / -f` flag to force the actions without asking for confirmation and the `--verbose / -V` flag to use the verbose mode. To learn more about the commands being executed, read Chapter 18.

Note: The following instructions assume the driver has been loaded. Prerequisite: Should have an empty SEF Unit. Delete Virtual Devices following directions in Section 19.2.2

```
$ sudo sef-cli list sef
SEF Unit Index    Vendor  FW Version  HW Version     Channels   Dies
SEF Unit 0        KIOXIA  1LMSS528    1              8          192
SEF Unit Count 1

$ sudo sef-cli create virtual-device --sef-index 0 \
 --virtual-device-id 1 --super-block 1:32 --force --verbose
Die Map:
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
```

```
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1
The Virtual Devices for SEF Unit 0 was successfully created

$ sudo sef-cli create qos-domain --sef-index 0 --label \
 "613006, 88877" --virtual-device-id 1 --force --defect-strategy \
 "kPacked" --verbose --flash-capacity-percent 5
The QoS Domain 1 was successfully created

$ sudo sef-cli configure ftl --sef-index 0 --label "613006, 88877" \
 --force --verbose
The QoS Domain 1 was successfully configured by SDK

$ sudo fio <path to SEF_SDK>/fio/official-fio/examples/sef.fio
```

Part II

Driver

7 | SEF Linux™ Driver

Although an Software-Enabled Flash™ (SEF) Unit reports as an NVMe 2.0 device, it requires its own `sef.ko` device driver and a customized `nvme.ko` driver. With a kernel version of 6.3, the only customization is to recognize an SEF Unit's non-standard PCI™ storage class. This requirement is expected to be dropped in future firmware.

The `sef.ko` driver is stacked on the `nvme.ko` and `nvme-core.ko` drivers. It enumerates the NVMe devices `/dev/nvme0` through `/dev/nvme255`. For every `/dev/nvmeX` that supports the SEF alternate command set, it creates a corresponding `/dev/sefX` device. A `/dev/sefXnY` is also created for every `/dev/ngXnY` as well.

The following functionality is implemented by the SEF Unit driver.

- Translates NVMe device udev asynchronous-event-notifications (AENs) into namespace-AEN udev events.
- Keeps paired commands on the same submit queue.
- Triggers refresh of QoS Domain and Virtual Device information when affected by SEF admin commands.
- Publishes device and namespace information via sysfs so discovery does not require root privileges.
- Adds support for metadata-only reads.
- Flushes delayed writes before the nvme-core driver's namespace enumeration freezes and flushes a device's queues.

The SDK always uses the SEF driver but either device driver can be used to communicate with an SEF Unit. Sending admin commands to the NVMe device may cause `sef-cli` to return out-of-date information. `sef.ko` is required for its extended functionality, and in specific cases, it is more efficient to use.

Part III

SEF Library

8 SEF Library

8.1 Design Environment

The Software-Enabled Flash™ (SEF) Library runs on a Linux™ host. It only supports user mode and does not support forked processes. The SEF Library API is defined by SEFAPI.h and implemented in libsef.so. It converts interface calls into NVMe SEF command set commands and submits them to the SEF kernel driver.

8.2 Design Strategy

The SEF Library makes every effort to execute what it's asked to do. It does not enforce any policies. Limits are the responsibility of the caller. I/O requests that are larger than the device's max data transfer size are split up by the library in a way to simulate a single device request. For split writes and copies, requests for flash addresses are serialized to keep the addresses sequential when possible. The write commands and split reads run concurrently with no limit placed on how many requests are generated. Large I/Os and large I/O depths can overwhelm the device queues, causing the driver to reject requests. These are automatically retried after a delay. This makes the API simple to use but can lead to significant latency and may render weighted fair queueing I/O policies ineffective.

8.3 Threading Model

A threading model was chosen that prevents lost I/O completion and deadlock. I/O uring is used to send all asynchronous requests to the SEF driver. The submitting and completion of asynchronous SEF driver requests is handled by an internal, statically sized thread pool based on the number of CPUs. When possible, synchronous requests use `ioctl()` issued from the caller's thread. Pair commands, which can't use `ioctl()`, use a dedicated thread. This design allows the lifetime of the caller's thread to be independent of the I/O it issues and prevents deadlock when a synchronous API call is made from an asynchronous I/O completion callback. New I/O issued from a completion routine will be submitted using the completion thread, avoiding the overhead of switching to an I/O thread. There are two additional threads started by the SEF Library to handle notifications. One handles udev events sent

by the SEF driver, and the other monitors closing Super Blocks and issues a close notification once all write I/O has completed.

Use of the SEF API will never internally deadlock, but issuing a synchronous request or waiting for a resource owned by another completion may add latency to any I/O issued by the blocked thread. Deadlock may still occur externally if a completion routine blocks on resources that require another completion routine to execute.

Part IV

Reference FTL

9 | SEF Reference Flash Translation Layer (FTL)

The Software-Enabled Flash™ (SEF) Reference FTL implements an asynchronous user mode block API using the SEF Library. The implementation serves as an example of how to use the SEF Library and demonstrates the following:

- Managing a Flash Translation Layer (FTL) with multiple placement IDs
- Persisting and recovering FTL metadata
- Background garbage collection using I/O priority control
- Handling and accounting for device wear

The code is implemented with simplicity in mind. Parts of the Reference FTL can be used in other projects or expanded and enhanced based on project requirements and needs.

The SEF Reference FTL is implemented as a shared library. The public, `C-11` API is defined in `sef-block-module.h`. It implements an asynchronous block API using a host-based flash translation layer (FTL). Garbage collection is handled by the Reference FTL and supports a QoS Domain configured with multiple placement IDs. Over-provisioning is set with a one-time configuration of a fresh QoS Domain by calling `SEFBlockConfig()`. Once configured, a context for a QoS Domain is obtained by calling `SEFBlockInit()`. The returned context is required when issuing I/O with `SEFBlockIO()`.

I/O is only asynchronous and inherits the threading model of the SEF Library. The FTL is designed to have no locking for reads, and, given adequate over-provisioning, limited locking for writes. Reads are never queued, and writes are only queued by the FTL when there are no available Super Blocks in the QoS Domain. The writes are queued until GC can recover a free Super Block. Garbage collection runs when the end of the over-provisioning is encountered. Blocked writes waiting for GC can be canceled by calling `SEFBlockCancel()`.

The FTL's LBA-to-flash address lookup table (LUT) is kept entirely in RAM. It is loaded in `SEFBlockInit()` and saved in `SEFBlockCleanup()`. In the case of a client/system crash, the saved LUT

will be out of date but marked as dirty. When this happens, `SEFBlockInit()` will fail. The LUT can be both checked and repaired by calling `SEFBlockCheck()`. The current implementation rebuilds the LUT entirely from Super Block metadata, but the internal function that does the work is capable of updating a stale LUT by only processing the metadata of Super Blocks written after the LUT was last saved.

The SDK is made up of 7 components:

- Block Layer
- Flash Translation Layer
- Super Block State Management
- Garbage Collection
- Persistence
- Instrumentation
- Logging

These sections document the internals and theory of operation for each layer.

10 | Block Layer

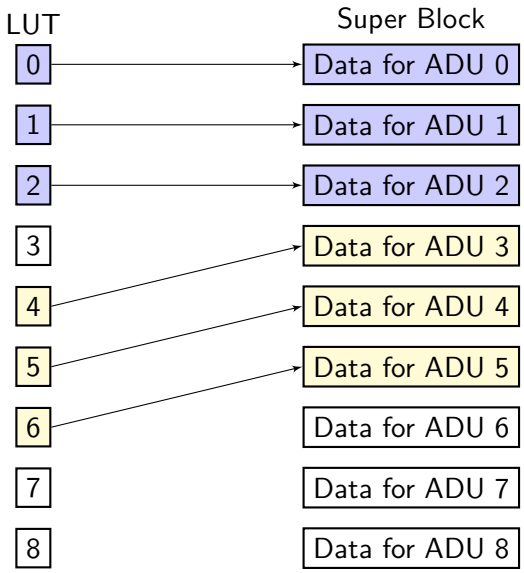
The block layer is the public API for the Reference FTL and implements an asynchronous block API. It uses the flash translation layer (FTL) to map user LBAs to device flash addresses. The properties of a configured QoS Domain are read using `SEFBlockGetInfo()`. The `deferMount` member in `struct SEFBlockOption` can be set to `true` to create a context to quickly access properties without having to perform a full initialization.

10.1 I/O

Block I/O is issued by calling `SEFBlockIO()`. It places no restriction on the size of a write, but the SEF Library will break large writes up into multiple smaller writes based on device and operating system limits. There is also no restriction on the size of a read, but large reads will be broken up into smaller reads of contiguous runs of flash addresses. Typically, if data was written with one write, it can be read with one read until it crosses a Super Block boundary. However, after portions are rewritten by the client or have been moved on flash as part of a garbage collection (GC), runs can be shortened. More SEF read requests will be required to read the same data. This is handled automatically by the block layer for the caller.

As an example, this is a section of the LUT and a newly erased Super Block after two writes. The first was a write to LBA 0 of 3 ADUs in blue and the second was a write to LBA 4 of 3 ADUs in yellow.

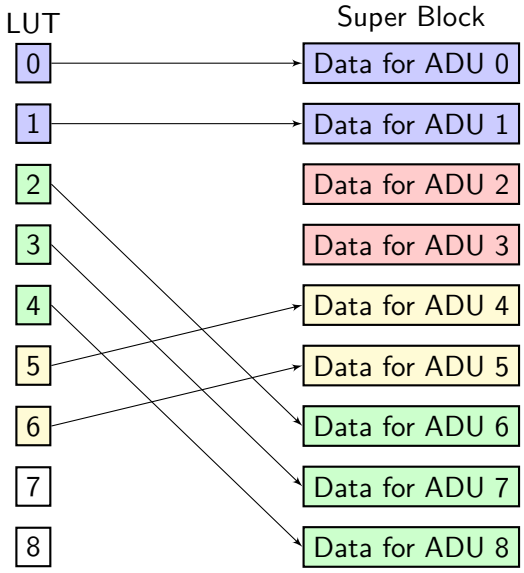
Figure 10.1: Write Example 1



When a client issues a read of LBA 0 through 6, internally the block layer will need to issue 3 calls to `sbmWholeRead()`. The flash addresses are contiguous, but the LBA addresses are not. The first call will issue a read for LBA0/ADU0 of 3 ADUs, the second call will not issue a read, but will fill the buffer for LBA 3 with zeros and the third call will issue a read for LBA4/ADU3 of 3 ADUs.

After the gap at LBA 3 is filled by a third write in green to LBA 2 of 3 ADUs, this is how the LUT and Super Block appear:

Figure 10.2: Write Example 2



The gap at LBA 3 has been filled. ADUs 2 and 3 are colored red because they are unreachable through the LUT and have been marked as invalid in the Super Block layer. GC skips ADUs marked as invalid. Another result of the write is the flash addresses are no longer contiguous. A client reading LBAs 0 through 6 will result in 3 calls to `sbmWholeRead()`. The first call will issue a read for LBA0/ADU0 of 2 ADUs, the second call will issue a read for LBA2/ADU6 of 3 ADUs and the third call will issue a read for LBA5/ADU4 of 2 ADUs.

Reads can be issued as soon as the LUT is updated, but because an SEF Unit can delay the write of data to flash, when a bad block is encountered, a previously returned tentative flash address will be asynchronously updated via the QoS Domain's notification function. When the read of a tentative flash address has been issued against an updated flash address, it will fail and needs to be retried with the updated flash address. The block layer handles re-issuing the read using the updated flash address. See section 10.3 Delayed Writes for details.

10.2 I/O priority

The I/O priority used for reads and writes is inherited from the default weights configured for the QoS Domain and read FIFOs from the Virtual Device. However, when a garbage collect is active, the write and copy weights are adjusted by the block layer such that the throughput of read stays unchanged. The priority of GC copies vs. writes is adjusted by `sbmAdjustIOWeights` at the start of every GC-issued copy command.

Even though priority can be adjusted for every copy command, the current priority scheme is fixed and based on the worst-case WAF, which can be calculated from the amount of over-provisioning.

$$WAF <= 1/OP$$

Every user write requires GC to write $WAF - 1$ more. That is, the ratio of copy to write weights has to be at least $WAF - 1$ to not be overrun by user writes. At the same time, the sum of the weights must be equal to the default program weight to not affect read die time.

$$programWeight = defaultWeight * WAF$$

$$copyWeight = defaultWeight * (WAF - 1)/WAF$$

At the end of a GC cycle, the program weight is restored to the default weight.

10.3 Delayed Writes

An SEF Unit uses delayed writes to quickly respond with a flash address but remove the burden of the client handling the program size of the device. This requires the buffer lifetime to be longer than completion of `SEFBlockIO()`. This introduces extra overhead in the form of a buffer allocation and data copy. The impact of delayed writes can be eliminated by the client taking responsibility for buffer management.

When the flag kSEFBlockIOFlagNotifyBufferRelease is set in a `struct SEFMultiContext` I/O request, the caller's buffer is used for delayed writes. That is, the buffer passed in must continue to be valid, even after the I/O's completion routine has been called. Once the second phase of write is complete, the FTL client will receive a `struct SEFBlockNotify` through the notify routine supplied in `struct SEFBlockOption` when `SEFBlockInit()` was called. The iov and iovcnt members describe the portion of the write buffer that is now committed to flash. A delayed write typically completes in milliseconds but can take tens of minutes to complete when there is insufficient data to fill the device's internal write buffer. The SDK includes a sample buddy allocator that can allocate a large buffer with minimal fragments where portions of the buffer are freed at different times.

10.4 Crash Recovery

The LUT is kept entirely in DRAM by the FTL layer. If the FTL client process were to crash, the LUT and data saved on the device would be out of sync. When this happens, `SEFBlockInit()` will fail. This can be fixed by repairing the LUT using `SEFBlockCheck()`. Enough metadata is kept with each write to reconstruct the LUT, even when it is entirely lost.

11 | Flash Translation Layer

The FTL manages the LBA lookup table (LUT). It uses the LUT to translate LBA-based block I/O into SEF API flash address-based I/O on behalf of the block layer. It also processes events from the SEF Library to keep the LUT and Super Block state up to date as user I/O and GC I/O complete. The FTL keeps the LUT in DRAM, only persisting it when the block layer is closed. The LUT uses 8 bytes of DRAM for each LBA and is by far the largest consumer of DRAM with 2GiB required for a 1TiB QoS Domain. It can be reconstructed or repaired from the written user data in the case of an unclean shutdown.

The FTL exposes functions to update and use the LUT. `SFTLookupForRead()` returns the flash address for a given LBA. The call will fail if the owning Super Block is not in the Open or Closed state. Additionally, it increments the reader count for the Super Block. This prevents the Super Block from being released until `SFTReleaseForRead()` is called. When the Super Block is known to have a non-zero reader count, `SFTLookup()` can be used, which does not modify the reader count or validate the Super Block is in a state where reads are allowed. Entries are placed in the LUT using `SFTSet()` and `SFTUpdate()`. `SFTSet()` is the authoritative way to update the LUT. It overwrites whatever is in the LUT entry for a given LBA, whereas `SFTUpdate()` will only update when the current entry is unchanged.

The FTL also includes the function `SFTRebuildLut()` that can be used to validate or repair the LUT. It can be invoked with SEF-CLI in the case of an unclean shutdown or to rebuild the LUT after a domain has been restored from a backup.

12 | Super Block State Management

Super block state management tracks the states of Super Blocks in use by the FTL. It mirrors the state of the blocks inside the SEF Unit so decisions can be driven by information in DRAM rather than a constant stream of device requests. The Super Block state machine is driven by a combination of notifications from the SEF Unit, the results of I/O operations, and operations performed by garbage collection (GC).

The state of each Super Block is tracked in an array indexed by Super Block ID. As a result, memory consumption is affected by the size of the Virtual Device, 16 bytes per Super Block. A bitmap is allocated to track which ADUs are valid, requiring `8+super_block_capacity_adus/8` bytes for each Super Block assigned to the QoS Domain. The bits are set and cleared using `SSBSetAduValid()` and `SSBClearAduValid()`, respectively. A snapshot of the bits is read using `SSBCopyValidBits()`. As an example, for a 1TiB Virtual Device with 64Kibi ADUs per Super Block and a maximally sized QoS Domain, the memory required is 64KiB for an empty QoS Domain and just over 32MiB when it is full.

$$numSB = \frac{1\,\mathrm{TiB}}{64\,\frac{\mathrm{KiAdu}}{\mathrm{SB}} * 4\,\frac{\mathrm{KiB}}{\mathrm{Adu}}} = 4\,\mathrm{KiSB}$$

$$sbStateMem = 16\,\frac{\mathrm{B}}{\mathrm{SB}} * numSB = 64\,\mathrm{KiB}$$

$$aduBitMem = numSB * (8\,\frac{\mathrm{B}}{\mathrm{SB}} + \frac{64\,\frac{\mathrm{KiAdu}}{\mathrm{SB}}}{8\,\frac{\mathrm{Adu}}{\mathrm{B}}}) \approx 32\,\mathrm{MiB}$$

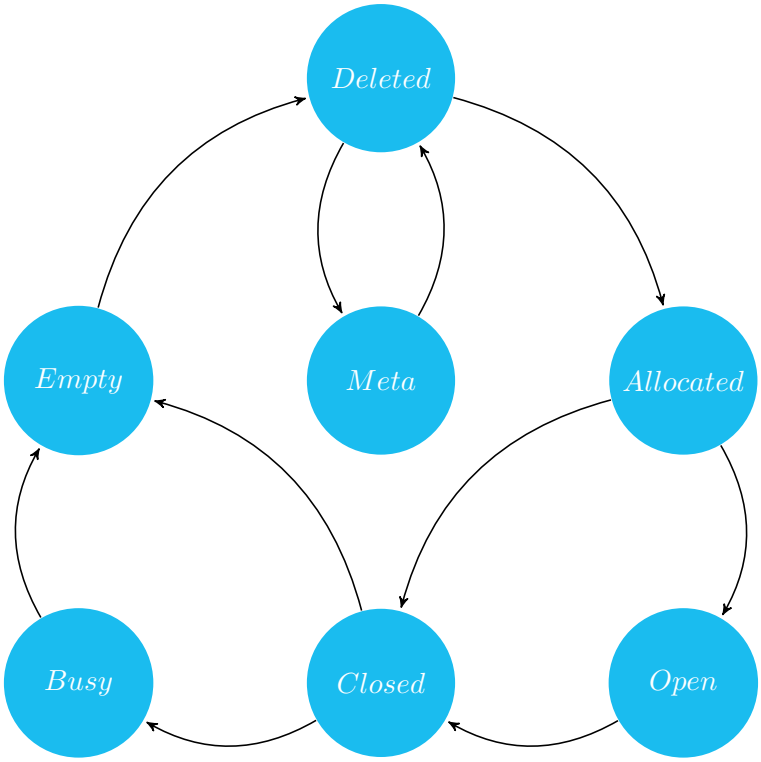
Super blocks can be in 1 of 7 states. The Super Block states are described in Table 12.1 Super Block States, and transitions are shown in Figure 12.1.

Table 12.1: Super Block States

State	Description

Deleted	Not in use by the QoS Domain.
Allocated	Owned by the QoS Domain but is not yet available for read or write. A Super Block that is the destination for nameless copy is in this state until it is filled.
Open	Mirrors the open state in the device. The Super Block is available for read and write.
Closed	Mirrors the closed state in the device. The Super Block is available only for read.
Busy	Valid ADU count is zero, but there are still reads in-flight. No new reads are allowed.
Empty	Valid ADU count is zero and the Super Block is being released back to the Virtual Device Super Block pool.
Meta	Holds FTL data, not user data. These are allocated/released when the block layer shuts down and persists the LUT.

Figure 12.1: Super Block State Transitions



The state value also contains a reader count. When a read is issued against a Super Block, that count is incremented by `SSBMarkBusy()` to prevent it from being deleted. `SSBMarkNotBusy()` decrements the count once the read completes. There is also a reference count that is incremented by `SSBSetAduValid()` for each valid ADU and decremented by `SSBClearAduValid()` for each invalid ADU. Two extra counts are held for the allocated and open states. The extra counts are removed by calls to `SSBRemoveRef()` and by `SSBClosed()`. Table 12.2 Super Block Transitions provides additional detail on when a Super Block will transition to a new state.

Table 12.2: Super Block Transitions

Transition	Triggers
Deleted to Allocated	There are two events that cause this transition: when a flash address is returned from a write for a Super Block in the deleted state and when garbage collection allocates a destination Super Block.
Allocated to Open	When a flash address is returned from a write transitioning the Super Block to the allocated state, it moves immediately to the open state.
Allocated/Open to Closed	A close notification from the device is processed.
Closed to Busy	The reference count for the Super Block has become zero, and the active reader count is non-zero. The reference count is decremented when an LBA is rewritten, GC moves an LBA, or the Super Block transitions out of Open.
Closed to Empty	The reference count for the Super Block has become zero, and the active reader count is zero The reference count is decremented when an LBA is rewritten, GC moves an LBA, or the Super Block transitions out of Open
Busy to Empty	The busy count has become zero.
Empty to Deleted	The release of the Super Block is issued and moves immediately to Deleted.
Meta to/from Deleted	At FTL shutdown, the persistence layer has allocated space to save the LUT and has released the old LUT back to the Virtual Device Super Block pool.

Super blocks are released asynchronously as they become empty. This can be prevented by marking a block as busy with `SSBMarkBusy()`. The returned instance ID can be used to verify the block hasn't been released and reallocated between enumeration with `SSBEnumBlocks()` or other calls to `SSBMarkBusy()`.

13 | Garbage Collection

Garbage collection (GC) is a single background thread that coalesces partially invalidated Super Blocks into a new Super Block so the partially invalid ones can be released. This makes them available for future writes for the QoS Domains in the same Virtual Device. It uses the Super Block layer to know which blocks are available as sources and for allocating the destination Super Block. The actual data movement is performed by the SEF Unit, but the decisions of when and what to move are made by GC. The priority used for copy I/O is selected externally from the garbage collection. At the start of each source-block copy, a notification is sent allowing the block layer to modify the queue-weight overrides for the copy. The garbage collection thread also has other related housekeeping functions for wear leveling, running patrols and releasing blocks marked for maintenance.

Garbage collection is started by calling `GCTrigger()`. This signals the GC thread to attempt a garbage collection. It loops garbage collecting the domain until the number of allocated ADUs is below the GC trigger value. The GC trigger value is near the end of the over-provisioning set by `SEFBlockConfig()`. When a patrol has been indicated for the device from the SEF Library, a call to `SEFCheckSuperBlock()` is issued for each Super Block returned by `SEFGetCheckList()`. With the collection cycle complete, the GC thread waits for the next trigger.

Garbage collection is performed by `gcQoSDomain()`. It starts by enumerating all the collectable blocks allocated to the QoS Domain using the function `SSBEnumBlocks()`. Enumerated blocks are sorted by placement ID and ranked by `gcBuildListOfCandidates()`. The rank of a candidate source Super Block is simply the number of valid ADUs it has. The rank is hard-coded to zero for Super Blocks marked for maintenance. This gives them highest priority within a placement ID. Next, the placement ID to collect is selected by `gcSelectPlacementIdToCollect()`. For a placement ID to be collected, it must have a Super Block's worth of invalid ADUs. It does not need to have a Super Block's worth of valid ADUs. When this occurs, the destination Super Block is closed before it's full. Placement IDs with Super Blocks marked for maintenance are processed first. Otherwise, the placement ID with the highest ratio of invalid ADUs to allocated blocks is selected. During the process of selecting a placement ID for GC, information about the top source candidates is cached in the `sbList` member of the `struct gcPidStats` in the `struct gcContext`. The amount cached is set at compile time by the value of `GC_SB_LIST_SIZE`. It is a compromise between using the most up-to-date information

and the CPU required to re-enumerate the blocks in a QoS Domain. Because the cache can be smaller than the number of source blocks required to fill the destination, the `struct gcContext` has a bitmap of which blocks have already been used as a source in its member `srcBitmap`. This is used to prevent a Super Block from being used twice for the same destination.

With a placement ID selected for collection, `gcToASuperBlock()` issues a nameless copy to the SEF Unit for each source Super Block in rank order until the number of ADUs requested to copy is greater than the number of writable ADUs in the destination Super Block. As each nameless copy completes, it is queued to be processed after all the copies have completed and the destination Super Block is closed. `gcPostProcessCopyIOCB()` processes the completed nameless copies with `SEFProcessAddressChangeRequests()`, which will generate `kAddressUpdate` notification events for each copied ADU. The FTL's QoS Domain notification handler, `HandleSEFNotification()`, calls `SFTUpdate()` to do a non-authoritative update of the LUT so a new flash address will only be updated if unchanged since copied. This is done so that whenever there is a race between an LBA being rewritten and garbage collect to update the LUT, the rewrite always wins. Once `gcToASuperBlock()` completes, the cycle starts again, evaluating the need for GC, selecting a placement ID to collect, and performing the garbage collection.

14 | Persistence

The SEF Reference FTL uses the persistence layer to store the flash translation layer's operating metadata, such as the lookup table, and Super Block information to the flash memory. The persistence layer manages that in two parts: Data Management and Data Tree storage.

14.1 Data Management

The data management code, which is located in `persistence.c`, is used to manage the data to be stored, stored data, and the root pointer.

14.1.1 Data to Be Stored

After the persistence layer is initialized, individual components of the Reference FTL can queue data to be flushed. The data is stored and identified by a unique key and index. While queuing the data, the caller may provide a callback function that will be called when the data is flushed to the flash memory or when the persistence cleanup is called. This callback function may be used to clean up allocated memory or perform other follow-up actions. Moreover, given the return value of isFlushed, appropriate action may be taken considering whether the data was flushed.

Queuing the data does not ensure the data is stored to the flash memory. In order to store the data on the flash memory, the queued data should be flushed by calling `PDLFlushToFlash()`. The caller is responsible for freeing the data previously stored on the flash memory. In other words, the previously written data should be cleared manually by calling `PDLFreeFlash()` to avoid leaks.

14.1.2 Stored Data

In order to keep track of the flushed data, a metadata table is used. The metadata table is an array of stored objects and their unique keys, unique indexes, sizes, and CRC-32. The array of metadata is flushed at the time of storage and is used to identify and read the stored data.

As part of persistence initialization, the persistence layer uses the root pointers to locate the metadata table and find out about the stored data. A cyclic redundancy check (CRC-32) is used in order to verify the stored metadata. Moreover, given that multiple code bases may use the persistence layer independent

of the Reference FTL, each code base should use a Globally Unique Identifier (GUID). The GUID is checked at the time of persistence initialization.

To read the stored data, the caller uses a unique key and index. The stored data is also verified using the CRC-32.

In order to separate the user data from the metadata stored by persistence, the persistence layer stores the data on isolated Super Blocks. The persistence layer manually allocates Super Blocks that are only used by persistence to store the queued data.

14.1.3 Root Pointer Management

The root pointers are used to keep track of the persisted data. They are used as pointers into the device where the metadata table is stored. If the root pointer is null, it is assumed that the device is empty and no persistence data has been flushed; however, a null root pointer could also indicate that the device is not empty and the persistence layer was not able to store the metadata or update the root pointer. In order to avoid ambiguity, the caller should mark the root pointer as dirty after data has been written to the device.

Given that the Reference FTL may experience unexpected shutdowns, the root pointer is an important tool for verifying the freshness of the read data. To detect an unclean shutdown, the root pointer is checked at initialization time. A dirty root pointer denotes that the software experienced an unexpected shutdown.

The root pointer is set to clean either automatically, after data is flushed to the flash memory, or manually, by being marked as clean.

The root pointer is marked as dirty by creating a Flash Address that has its QoS Domain ID set to zero. The root pointer can be marked as clean and reconstructed by setting the QoS Domain ID of the Flash Address back to the QoS Domain ID of the loaded QoS Domain. The Reference FTL uses this mechanism in order to detect an unexpected shutdown.

In summary, the root pointer is checked when the Reference FTL is initialized, and a dirty root pointer denotes the occurrence of an unexpected shutdown. If the root pointer is clean, the Reference FTL marks the root pointer as dirty when the first write is performed. When the Reference FTL goes through the expected shutdown process, the data is flushed and a clean root pointer is set, denoting a clean shutdown.

14.2 Data Tree Object

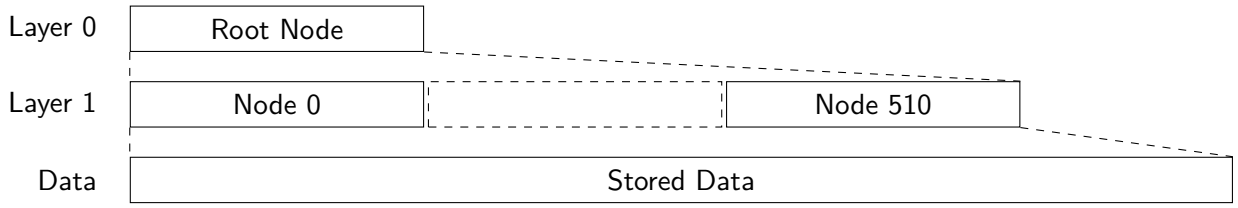
The Data Tree Object (DTO) is a self-contained data structure used to store contiguous data. The need for a DTO becomes apparent given SEF's support for various defect-management methods that do

not guarantee contiguous data placement. The kFragmented method, for example, may not store data contiguously.

The DTO root is always 1 ADU in size and stores the flash addresses of the next tree nodes. For example, if an ADU is 4096 bytes and the DTO node structure is 16 bytes, then 510 flash addresses (8 bytes each) can be used to store either the locations of the next layer's nodes or the location of data. Given this structure, the root node can store the flash addresses for up to 2 megabytes of data.

Figure 14.1 illustrates the structure of an example DTO. In this example, just by having a two-layer tree, persistence is able to keep track of an object with a maximum size of 1065 megabytes.

Figure 14.1: Data Tree Object Structure



The DTO offers functions to read, write, free, and get the size of an entire tree in order to simplify use of the given tree. Moreover, the host can get the Super Blocks used by a given tree. Given the structure of the DTO, it must be read as a whole and not partially.

15 | Instrumentation

The SEF Reference FTL has a unique, built-in instrumentation system that lets the user monitor and change the FTL's internal properties at runtime. Because the FTL is implemented in user mode, this instrumentation system is the only way to obtain performance metrics for the SEF Unit. The instrumentation system is exposed via a Unix Socket. It was designed with modularity in mind. In other words, individual components of the Reference FTL can register and manage their own communication systems.

15.1 Basic Usage

The instrumentation socket is created by default at `/tmp/SEFFTLDomain<sef-unit>.<qos-domain>` where `<sef-unit>` is the SEF Unit number and `<qos-domain>` is the QoS Domain ID. While the Reference FTL is running, ncat can be used to communicate with instrumentation via Unix Sockets. The following is a simple invocation to communicate with the SDK running on SEF Unit 0's QoS Domain ID 2:

```
$ncat -U --udp /tmp/SEFFTLDomain.0.2
```

The communication is two-way; the opened socket is treated as an interactive shell. A typical command for the instrumentation shell is <action> [options] where <action> is registered by a component of the Reference FTL. To view a list of all the registered actions and how to use them, use the `help` action. The Instrumentation Actions Table includes the built-in actions and their individual descriptions.

Table 15.1: Instrumentation Actions

Actions	Options	Description
cc	[json]	Dump all code coverage data
dump	[json][reset][info]	Dump all instrumentation counters
gc	[<low water>]	Show status of and configure garbage collect's low water value- triggers GC
gc	[dynamic|static]	Switch from dynamic to static weighting for gc
getlog		Returns log level

setlog	<level Number>	Sets the log level. Possible values are 0 to 5
setlog	<level Name>	Sets the log level. Possible values are "trace", "debug", "info", "error", "fatal", "nolog"
state	[json][info]	Dump all I/O state data or set weight
help		Prints help

15.2 Counters

One of the built-in actions for the instrumentation system is counters. The built-in counters keep track of various instrumentation data, such as numbers of reads and writes. Similar to the instrumentation system, individual components of the Reference FTL can register counters. The `dump` and `state` actions expose counter information. These actions can return data in a JSON format when given the `json` option. For a detailed list of the registered counters and their descriptions, both actions support an `info` option. Moreover, `dump` supports the `reset` option to reset the counters' values.

15.3 Extending Functionality

Extensibility is one of the main design features of the instrumentation system. The instrumentation can be extended by registering new actions and counters after initializing it. The registered actions are called using their callback functions after the registered keyword is received via the Unix Socket. The registered counters are managed by the instrumentation system; however, an optional update function can be used to update the values when the corresponding action is called.

16 | Logging

The logging service was built with customization in mind. In other words, the user can provide their own logging system while initializing the block layer or use the built-in file logger. The default file logger appends log messages from each caller to a shared log file. By default, these files are located at `/tmp/SEFLog.<unit>.<domain>` where `<unit>` is an SEF Unit number and `<domain>` is a QoS Domain ID.

16.1 Basic Usage

The supported log levels include the following, in order from most verbose to least: Trace, Debug, Info, Error, Fatal, or no log. By default, each instance is set to log messages at levels Trace and above. The level for logged messages can be changed after initialization.

The level of logging for an already-initialized instance should be accessed only using the public setters and getters, by calling either `LogSetLevel()` or `LogSetLevelByName()`. In the case of calling the public setters and getters, the dispatch function to change the log level is called and the local copy of the log level is changed.

To create new log entries, use the following function calls with their respective levels:

```
LogTrace(LogHandle, ...)
LogDebug(LogHandle, ...)
LogInfo(LogHandle, ...)
LogError(LogHandle, ...)
LogFatal(LogHandle, ...)
```

The logging macros will not call the log dispatch if the log level is higher than the current level. However, the dispatch function can be called directly to avoid the local level check.

16.2 Custom Logger

A custom logger can be used instead of the built-in file logger. In other words, to use a custom log function, the caller should provide a log handle with an `iLogger` dispatch. While changing the log level or logging, the provided dispatch functions are called in order to update their log level or add a log.

16.3 Default File Logger

The default file logger is initialized when the logHandle is not passed in while initializing the SEF Reference FTL. Moreover, an instance of the built-in file logger can be passed in while initializing if a custom log path is needed.

Each line in an output log file holds timestamped information about the level of the logging call (Trace, Debug, Info, Error, or Fatal), the calling function within a source code file and its line number, and the logging message itself.

Log files are not deleted automatically; information from previous calls will persist in log files inside of the `/tmp` directory (by default) until these files are manually removed and new files generate or are auto-generated.

17 | SEF FTL Public API

17.1 SEFBlockIOType

Type of i/o in a SEFMultiContext.

Table 17.1: Members of SEFBlockIOType

Member	Description
kSEFRead	Performs a read operation
kSEFWrite	Performs a write operation

17.2 SEFBlockIOFlags

I/O flags in a SEFMultiContext.

Table 17.2: Members of SEFBlockIOFlags

Member	Description
kSEFBlockIOFlagNotifyBufferRelease	Buffer lifetime is controlled by caller

17.3 SEFBlockNotifyType

Event types for SEFBlockNotify.

Table 17.3: Members of SEFBlockNotifyType

Member	Description
kSefBlockNotifyGCDown	GC can no longer make progress, write I/O will fail
kSefBlockNotifyBufferRelease	A portion of an I/O buffer can be released

17.4 SEFBlockConfig

```
struct SEFStatus SEFBlockConfig(uint16_t SEFUnitIndex, struct
    SEFQoSDomainID QoSDomainID, struct SEFBlockConfig *config)
```

This function is used to configure the SEF Block Module.
It configures freshly created QoS domains so they can used as a SEF Block Module domain. It sets the number of QoS domains and the amount of over provisioning to use. Once these values are set, they cannot be changed without erasing or recreating the QoS domains.
A QoS domain must be configured as a SEF Block Module domain before calling SEFBlockModuleInit().
See Also: SEFBlockInit

Table 17.4: Parameters of SEFBlockConfig

Type	Name	Direction	Description
uint16_t	SEFUnitIndex	In	The index of the SEF Unit; the index is zero-based
struct SEFQoSDomainID	QoSDomainID	In	QoS Domain ID
struct SEFBlockConfig*	config	In	A pointer to an instance of SEFBlockConfig, for config settings

Table 17.5: Return value of SEFBlockConfig

Type	Description
struct SEFStatus	Status and info summarizing result.

Table 17.6: Return values of SEFBlockConfig

Error Value	Description
-EINVAL	Invalid over provisioning percentage
-EEXIST	The device is already configured, can not reconfigure a pre-configured device
-EBADF	The block module was not shutdown cleanly; Consider running Check Disk

17.5 SEFBlockInit

```
struct SEFStatus SEFBlockInit(uint16_t SEFUnitIndex, struct
    SEFQoSDomainID QoSDomainID, struct SEFBlockOption *options,
    SEFBlockHandle *blockHandle)
```

This function is used to get a SEFBlockModuleHandle for issuing I/O.

The QoS domain must be configured as a SEF Block Module domain using SEFBlockConfig() before calling SEFBlockInit(). Configuration only needs to be done once.
See Also: SEFBlockConfig

Table 17.7: Parameters of SEFBlockInit

Type	Name	Direction	Description
uint16_t	SEFUnitIndex	In	The index of the SEF Unit; the index is zero-based
struct SEFQoSDomainID	QoSDomainID	In	QoS Domain ID
struct SEFBlockOption*	options	In	A pointer to an instance of SEFBlockOption, for runtime options
SEFBlockHandle *	blockHandle	In	A pointer to the SEF Block handle to be used for access to the block instance

Table 17.8: Return value of SEFBlockInit

Type	Description
struct SEFStatus	Status and info summarizing result.

Table 17.9: Return values of SEFBlockInit

Error Value	Description
-EINVAL	Invalid log level
-EBADF	The block module was not shutdown cleanly; Consider running Check Disk
-ENOENT	Block layer has not been configured with SEFBlockConfig()

17.6 SEFBlockMount

```
struct SEFStatus SEFBlockMount(SEFBlockHandle blockHandle)
```

This function mounts the FTL configured domain if not already mounted.
It's required to be called before calling SEFBlockIO() when SEFBlockInit() was called with the option delayMount set to true. Calling it when it's not required has no effect.
See Also: SEFBlockInit, SEFBlockIO

Table 17.10: Parameters of SEFBlockMount

Type	Name	Direction	Description

SEFBlockHandle	blockHandle	In	SEF Block handle to be used for access to the block instance

Table 17.11: Return value of SEFBlockMount

Type	Description
struct SEFStatus	Status and info summarizing result.

17.7 SEFBlockGetInfo

```
void SEFBlockGetInfo(SEFBlockHandle blockHandle, struct
    SEFBlockInfo *info)
```

This function returns size information about a block device.
The function requires the block layer to be initialized before it is called.

Table 17.12: Parameters of SEFBlockGetInfo

Type	Name	Direction	Description
SEFBlockHandle	blockHandle	In	SEF Block handle to be used for access to the block instance
struct SEFBlockInfo*	info	Out	SEFBlockInfo struct to fill with data

17.8 SEFBlockIO

```
struct SEFStatus SEFBlockIO(struct SEFMultiContext *context)
```

This function is used to perform I/O commands.

Table 17.13: Parameters of SEFBlockIO

Type	Name	Direction	Description
struct SEFMultiContext*	context	In	A pointer to an instance of SEFMultiContext

Table 17.14: Return value of SEFBlockIO

Type	Description
struct SEFStatus	Status and info summarizing result.

Table 17.15: Return values of SEFBlockIO

Error Value	Description
-EINVAL	The input parameters are invalid; either Scatter gather list is too small or I/O exceeds the device capacity
-ENOPROTOOPT	FTL is either not initialized or shutting down, failing I/O
-EBUSY	I/o context p appears to be in use
-ECANCELED	GC shutdown, inflight writes canceled

17.9 SEFBlockTrim

```
struct SEFStatus SEFBlockTrim(SEFBlockHandle blockHandle, int64_t
    lba, int32_t lbc)
```

This function is used to discard or TRIM LBAs that are no longer needed by the application.

Table 17.16: Parameters of SEFBlockTrim

Type	Name	Direction	Description
SEFBlockHandle	blockHandle	In	SEF Block handle to be used for access to the block instance
int64_t	lba	In	Logical block address
int32_t	lbc	In	Logical block count

Table 17.17: Return value of SEFBlockTrim

Type	Description
struct SEFStatus	Status and info summarizing result.

Table 17.18: Return values of SEFBlockTrim

Error Value	Description
-ENOTBLK	The Block context is not valid
-EINVAL	The LBA or LBC is not valid
-EIO	Was unable to trim; the Info would indicate LBC that wasn't completed

17.10 SEFBlockCancel

```
struct SEFStatus SEFBlockCancel(struct SEFMultiContext *context)
```

Requests the passed in I/O be canceled.
It's the caller's responsibility to ensure that the context is valid for the duration of the call. Because of internal race conditions, the call may fail and yet still cancel the I/O. Also possible is the function returns success, yet the I/O completes without error instead of a canceled I/O error.

Table 17.19: Parameters of SEFBlockCancel

Type	Name	Direction	Description
struct SEFMultiContext*	context	In	a pointer to an instance of SEFMultiContext

Table 17.20: Return value of SEFBlockCancel

Type	Description
struct SEFStatus	Status and info summarizing result.

Table 17.21: Return values of SEFBlockCancel

Error Value	Description
-ENOENT	The I/O was not found in the any queues

17.11 SEFBlockCheck

```
struct SEFStatus SEFBlockCheck(SEFBlockHandle blockHandle, int
    shouldRepair)
```

This function is used to check if the stored data and metadata match.
Moreover, it is capable of repairing the device. To create a blockHandle for a device that won't mount, call SEFBlockInit() with the delayMount option set to true.

Table 17.22: Parameters of SEFBlockCheck

Type	Name	Direction	Description
SEFBlockHandle	blockHandle	In	A pointer to the SEF Block handle to be used for access to the block instance.
int	shouldRepair	In	Should the block module be repaired if it is faulty? 1 denotes true

Table 17.23: Return value of SEFBlockCheck

Type	Description
struct SEFStatus	Status and info summarizing result. An error is returned if the device is faulty.

17.12 SEFBlockCleanup

```
struct SEFStatus SEFBlockCleanup(SEFBlockHandle *blockHandle)
```

This function is used to clean up the SEF Block Module by freeing up memory and stopping all related functionality.

Table 17.24: Parameters of SEFBlockCleanup

Type	Name	Direction	Description
SEFBlockHandle *	blockHandle	In	A pointer to the SEF Block handle to be used for access to the block instance

Table 17.25: Return value of SEFBlockCleanup

Type	Description
struct SEFStatus	Status and info summarizing result.

Table 17.26: Return values of SEFBlockCleanup

Error Value	Description
-ENODATA	The input block handle is not valid

17.13 SEFMultiContext

I/O request for SEFBlockIO()

Table 17.27: Members of SEFMultiContext

Type	Name	Description
SEFBlockHandle	blockHandle	SEF Block handle to be used for access to the block instance
struct SEFMultiContext*	parent	A pointer to an instance of SEFMultiContext used for compound operations

void(*)(struct SEFMultiContext *)	completion	The function that is called when the transaction is completed
void *	arg	A pointer that can be used by caller for any reason
uint64_t	lba	Logical block address
uint32_t	lbc	Logical block count
uint8_t	ioType	enum SEFBlockIOType that needs to be performed
uint8_t	flags	I/O flags enum SEFBlockIOFlags
uint8_t	readQueue	kSEFRead queue override when valid
uint16_t	ioWeight	I/O weight to override when non-zero
struct iovec*	iov	A pointer to the scatter/gather list
size_t	iovOffset	Starting byte offset into iov array
int	iovcnt	The number of elements in the scatter/gather list
uint16_t	qosIndex	0 based, used for multi-domain FTL
struct SEFPlacementID	placementID	Placement ID for writes
uint32_t	numLbl	Num logical blocks left in the super block
atomic_int	transferred	Counter denoting number of bytes transferred for the transaction
atomic_int	count	Reference count, I/O is completed -> 0
atomic_int	error	First error for the transaction
int	cancel	Set to indicate cancel in progress

17.14 SEFBlockNotify

Event data sent to a client's notification function.
The notification function is set when SEFBlockInit() is called.

Table 17.28: Members of SEFBlockNotify

Type	Name	Description
enum SEFBlockNotifyType	type	Type of notification
const struct iovec*	iov	Vector of buffers to release
int16_t	iovcnt	Count of buffers in iov

17.15 SEFBlockOption

Init time options.

Initialization time options supplied to SEFBlockInit()

Table 17.29: Members of SEFBlockOption

Type	Name	Description
int	logLevel	Initial log level (Trace = 0 , Debug = 1, Info = 2, Error = 3, Fatal = 4)
LogHandle	logHandle	Use an external logger
const char *	instrumentationPath	The location for the Unix Domain Socket. Will replace the first two format specifier with UnitIndex and QoSDomainId. Defaults to /tmp/SEFFTLDomain.[UnitIndex].[QoSDomainId]
void *	notifyContext	User context passed to notifyFunc()
void(*)(struct SEFBlockNotify event, void *notifyContext)	notifyFunc	Set to receive notifications
bool	delayMount	When true, SEFBlockInit() delays mounting until SEFBlockMount() is called

17.16 SEFBlockConfig

Configuration when calling SEFBlockConfig()

Table 17.30: Members of SEFBlockConfig

Type	Name	Description
int	overprovisioning	Percentage of over-provisioning (e.g., 20 for 20 percent)
int	unused0	Unused member
int	numDomains	Number of domains to use for a block-device. 0 indicates all defined domains
struct SEFBlockOption	blockOption	Run time options while configuring

17.17 SEFBlockInfo

Information about a block-module device.

Table 17.31: Members of SEFBlockInfo

Type	Name	Description
struct SEFADUsize	aduSize	Size of ADU and Meta
int16_t	numPlacementIDs	Number of Placement IDs
int	superBlockSize	Maximum number of ADUs in a super block
int	superPageSize	Number of ADUs in a super page
int	flashWriteSize	Number of ADUs to write to avoid padding (e.g., sync write)
uint64_t	capacity	Capacity in ADUs
int	overprovisioning	Percentage of over-provisioning (e.g., 20 for 20 percent)
int	numDomains	Number of domains the FTL covers

Part V

Command Line Interface (CLI)

18 | Command Line Interface (CLI)

`SEF-CLI` is a versatile command line tool provided as a means of performing both SEF Unit configuration and administration and as a debugging tool. `SEF-CLI` requires superuser permissions to interact with and configure SEF Units.

The `SEF-CLI` uses a target-and-action paradigm. In other words, an action is applied to the given target. When using the command line interface, the first input is an action, and the second input is the target the action is applied to. For example, `list sef` applies the `list` action to the `sef` target and will list the available SEF Units.

The shortest unique prefix may be used to invoke an action on a target. For example, `l se` may be used to list SEF Units, since `l` and `se` each correspond to a unique action and target.

A list of all actions and targets can be printed using the `--help` or `-h` flag. The help functionality may be viewed in greater detail using the `--verbose` or `-V` flag. The detailed help function prints targets, actions, and options for each target. It also prints the default value for each option. All targets support the `help` action.

The SEF CLI Targets and Actions Table enumerates the built-in targets and their corresponding actions.

Table 18.1: SEF CLI Targets and Actions

Target	Action	Action Description
sef-unit	list	Print list of available SEF Units. Additional info is printed in verbose mode
	info	Prints SEF Unit's information
	help	Print verbose help for the target
virtual-device	list	Print list of created Virtual Devices. Commands to recreate same configuration is printed in verbose
	draw	Print a map of created Virtual Devices
	info	Print a Virtual Device's information; Prints usage data in verbose mode

	create	Create a new Virtual Device
	delete	Delete Virtual Device. Virtual Device can only be deleted if it is empty
	set-pslc	Sets the number of pSLC Super Blocks for a Virtual Device
	set-suspend-config	Sets the suspend configuration for a Virtual Device
	help	Print verbose help for the target
qos-domain	list	Print list of available QoS Domains. Commands to recreate same configuration is printed in verbose
	info	Prints QoS Domain's information
	create	Create a new QoS Domain
	delete	Delete QoS Domain. All storage allocated to the QoS Domain is returned to the Virtual Device
	format	Format QoS Domain. All storage allocated to the QoS Domain is returned to the Virtual Device, but the QoS Domain will not be removed
	resize	Change size of existing QoS Domain
	label	Set the label for an existing QoS Domain
	backup	Backup QoS Domain's stored data to a set of files
	restore	Restore a set of backed up files to the QoS Domain
	help	Print verbose help for the target
ftl	info	Prints the SEF Block Module's information/configuration
	configure	Configure the SEF Block Module
	check	Check the SEF Block Module's health, and repair if broken
	help	Print verbose help for the target
shell	interactive	An Interactive shell to interact with the device in real time
	execute	Execute a Python script to interact with the device
	help	Print verbose help for the target

19 | SEF CLI Targets

The examples provided in this chapter assume an SEF Unit with 192 dies. In order to keep the examples easy to understand, the long versions of the options are used; however, most common options have short versions available. A confirmation prompt appears following many of the available commands, but using the `--force` flag (present in most examples below) allows the user to bypass this confirmation.

19.1 SEF Unit

The `sef-unit` target is used to list and get information about the installed SEF Units using `list` and `info` actions.

The following example uses the `list` action to list all the SEF Units installed on the machine.

```
$ sudo sef-cli list sef-unit
SEF Unit Index   Vendor   FW Version   HW Version   Channels   Banks
SEF Unit 0               1LMSS529                  8          24
SEF Unit Count 1
$
```

Additional information about a specific SEF Unit can be acquired using the `info` action. The following example prints the information for an SEF Unit installed at index 0.

```
$ sudo sef-cli info sef-unit \
    --sef-index 0
vendor: /dev/sef0
vendor:
serialNumber: P21L23E00193        1LMSS529
FWVersion: 1LMSS529
HWVersion:
supported Options:
* kPackedSupported
* kSuperBlockSupported
* kAutomaticSupported
```

```
* kFastestSupported
* kTypicalSupported
* kLongSupported
* kHeroicSupported
* kStopSupported
* kPSLCSupported
* kDeleteVirtualDeviceSupported
unitNumber: 0
APIVersion: 270
supportedOptions: 2614402
maxQoSDomains: 512
maxRootPointers: 8
maxPlacementIDs: 16
maxOpenSuperBlocks: 0
numReadQueues: 384
numVirtualDevices: 4
numQoSDomains: 0
numBanks: 24
numChannels: 8
numPlanes: 2
pageSize: 16384
numPages: 1792
numBlocks: 3294
totalBandWidth: 0
readTime: 0
programTime: 0
eraseTime: 0
minReadWeight: 32
minWriteWeight: 256
openExpirationPeriod: 0
ADUsize(1):
* data:meta
* 4096:16
$
```

19.2 Virtual Device Target

The `virtual-device` target is used to create, list, delete, configure, and get information about Virtual Devices.

19.2.1 Create Virtual Devices

The `create` action is used to create all Virtual Devices within an SEF Unit. This action supports the `--force / -f` flag to create the Virtual Devices without asking for confirmation. Moreover, the `--verbose / -V` flag, when provided, prints the configuration to be created.

Virtual Devices can be created using either auto place or die map.

Auto Placement of Virtual Devices

Auto place positions Virtual Devices at the best location given the input configuration. Auto place requires `channel-num`, `bank-num`, and `repeat-num` in lists of space-separated numbers denoting the number of channels and banks for each Virtual Device. The `repeat-num` option denotes how many times each channel and bank configuration should be repeated. If the proposed configuration does not fill the device, the user will be asked to confirm the proposed configuration; this confirmation is skipped when the `--force / -f` flag is given. The `--virtual-device-id / -v` option is used as the starting Virtual Device ID, and the remaining Virtual Device IDs are incremented by one.

The auto configuration algorithm fills the device with Virtual Devices, from largest to smallest. These Virtual Devices are placed across Channels and then Banks.

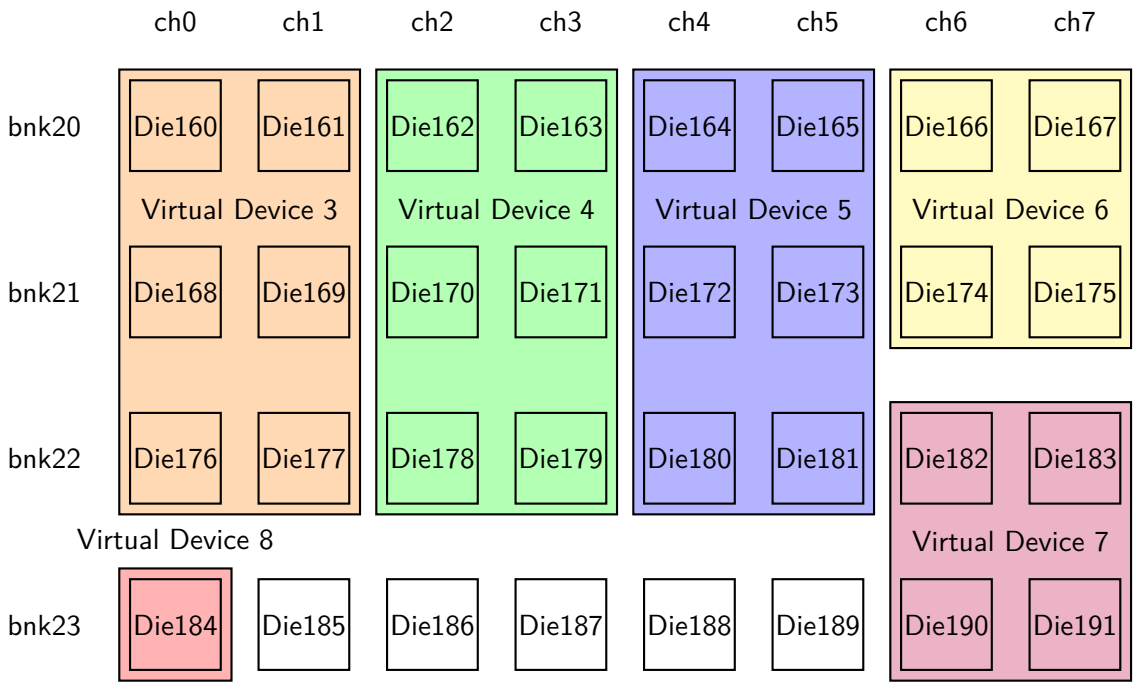
Prerequisite: Should have an empty SEF Unit. Delete Virtual Devices following directions in Section 19.2.2

```
$ sudo sef-cli create virtual   \
    --sef-index 0               \
    --virtual-device-id 2       \
    --channel-num   "2 2 1 8"   \
    --bank-num      "2 3 1 20"  \
    --repeat-num    "2 3 1 1"   \
    --verbose                   \
    --force
Die Map:
2    2    2    2    2    2    2    2
2    2    2    2    2    2    2    2
2    2    2    2    2    2    2    2
2    2    2    2    2    2    2    2
2    2    2    2    2    2    2    2
2    2    2    2    2    2    2    2
2    2    2    2    2    2    2    2
2    2    2    2    2    2    2    2
2    2    2    2    2    2    2    2
2    2    2    2    2    2    2    2
```

```
2     2     2     2     2     2     2     2
2     2     2     2     2     2     2     2
2     2     2     2     2     2     2     2
2     2     2     2     2     2     2     2
2     2     2     2     2     2     2     2
2     2     2     2     2     2     2     2
2     2     2     2     2     2     2     2
2     2     2     2     2     2     2     2
2     2     2     2     2     2     2     2
2     2     2     2     2     2     2     2
3     3     4     4     5     5     6     6
3     3     4     4     5     5     6     6
3     3     4     4     5     5     7     7
8     0     0     0     0     0     7     7
The Virtual Devices for SEF Unit 0 was successfully created
$
```

The above example command results in the Virtual Device configuration in Figure 19.1. The figure only shows the last 32 dies for simplicity. The first 160 dies are configured with a single Virtual Device as a filler to simplify the example.

Figure 19.1: Virtual Device Configuration Auto Placement



Using a Die Map for Virtual Devices

The die map creation method provides complete control over how each die is assigned to a Virtual Device. To use this method, pass in a space-separated list of Virtual Device IDs using the `die-map` option. The Virtual Device IDs should be passed in bank by bank, from the lowest channel to the highest channel per bank. The Virtual Device ID of 0 leaves a die unused and unassigned.

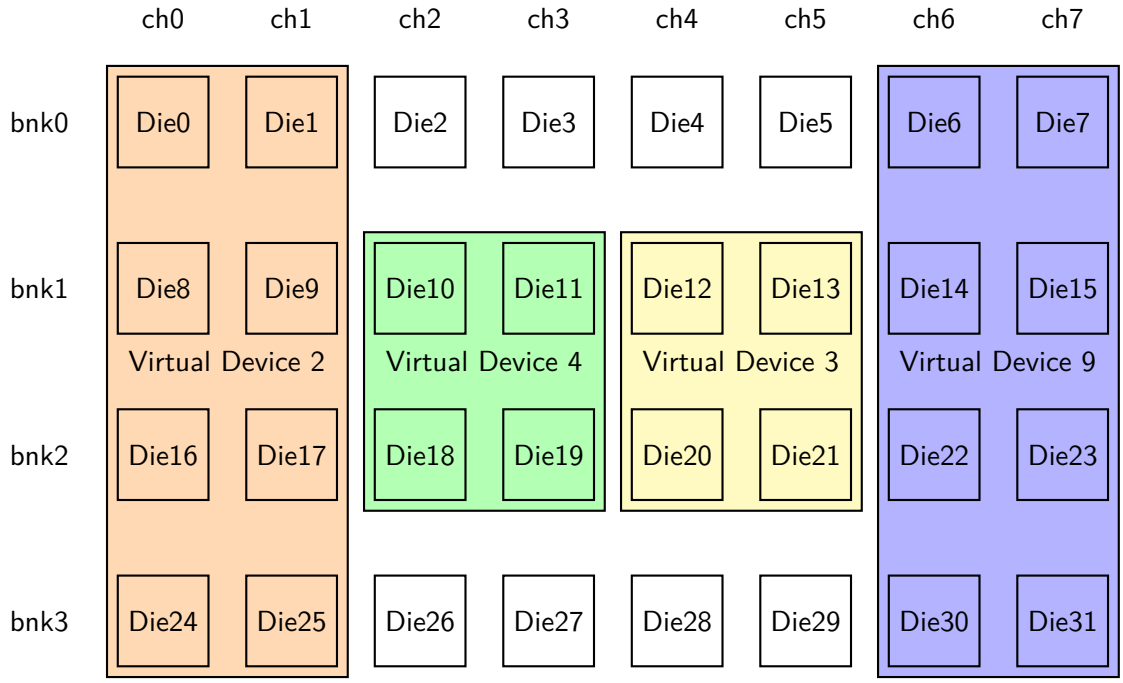
Prerequisite: Should have an empty SEF Unit. Delete Virtual Devices following directions in Section 19.2.2

```
$ sudo sef-cli create virtual  \
    --sef-index 0              \
    --verbose                  \
    --force                    \
    --die-map "2 2 0 0 0 0 9 9 \
        2 2 4 4 3 3 9 9        \
        2 2 4 4 3 3 9 9        \
        2 2 0 0 0 0 9 9        \
        0 0 0 0 0 0 0 0        \
        0 0 0 0 0 0 0 0        \
        0 0 0 0 0 0 0 0        \
        0 0 0 0 0 0 0 0        \
        0 0 0 0 0 0 0 0        \
        0 0 0 0 0 0 0 0        \
        0 0 0 0 0 0 0 0        \
        0 0 0 0 0 0 0 0        \
        0 0 0 0 0 0 0 0        \
        0 0 0 0 0 0 0 0        \
        0 0 0 0 0 0 0 0        \
        0 0 0 0 0 0 0 0        \
        0 0 0 0 0 0 0 0        \
        0 0 0 0 0 0 0 0        \
        0 0 0 0 0 0 0 0        \
        0 0 0 0 0 0 0 0        \
        0 0 0 0 0 0 0 0        \
        0 0 0 0 0 0 0 0        \
        0 0 0 0 0 0 0 0        \
        0 0 0 0 0 0 0 0"
Die Map:
2    2    0    0    0    0    9    9
2    2    4    4    3    3    9    9
```

```
2       2       4       4       3       3       9       9
2       2       0       0       0       0       9       9
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0
The Virtual Devices for SEF Unit 0 was successfully created
$
```

The above example command results in the Virtual Device configuration in Figure 19.2. The figure only shows the first 32 dies for simplicity; other dies are not configured.

Figure 19.2: Virtual Device Configuration Die Map



Creation Time Configuration Additional configurations can be applied while creating a Virtual Device.

Option `--super-block` is an array of tuples which is used to set the number of dies per Super Block. The default dies per Super Block is the number of dies in the Virtual Device. The supplied value should be divisible by the number of dies in the Virtual Device. Another configuration that can be applied is option `--read-weight` which is an array of tuples that is used to override the default weights for read operations for each possible read queue.

The example below creates a set of Virtual Devices similar to the previous example but overrides the number of dies per Super Block and default read weights.

Prerequisite: Should have an empty SEF Unit. Delete Virtual Devices following directions in Section 19.2.2

```
$ sudo sef-cli create virtual  \
    --sef-index 0              \
    --verbose                  \
    --force                    \
    --die-map "2 2 0 0 0 0 9 9 \
        2 2 4 4 3 3 9 9        \
        2 2 4 4 3 3 9 9        \
        2 2 0 0 0 0 9 9        \
```

```
          0  0  0  0  0  0  0  0         \
          0  0  0  0  0  0  0  0         \
          0  0  0  0  0  0  0  0         \
          0  0  0  0  0  0  0  0         \
          0  0  0  0  0  0  0  0         \
          0  0  0  0  0  0  0  0         \
          0  0  0  0  0  0  0  0         \
          0  0  0  0  0  0  0  0         \
          0  0  0  0  0  0  0  0         \
          0  0  0  0  0  0  0  0         \
          0  0  0  0  0  0  0  0         \
          0  0  0  0  0  0  0  0         \
          0  0  0  0  0  0  0  0         \
          0  0  0  0  0  0  0  0         \
          0  0  0  0  0  0  0  0         \
          0  0  0  0  0  0  0  0         \
          0  0  0  0  0  0  0  0         \
          0  0  0  0  0  0  0  0         \
          0  0  0  0  0  0  0  0         \
          0  0  0  0  0  0  0  0"        \
     --super-block "2:1 9:4"    \
     --read-weight "2:500 2:800 \
          2:1000 3:500"
Warning: The read weight for the Virtual Device 9 was not specified
Warning: The read weight for the Virtual Device 4 was not specified
Die Map:
2     2     0     0     0     0     9     9
2     2     4     4     3     3     9     9
2     2     4     4     3     3     9     9
2     2     0     0     0     0     9     9
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
```

```
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
The Virtual Devices for SEF Unit 0 was successfully created
$
```

The above example results in the same Virtual Device layout as Figure 19.2. However, Super Blocks in Virtual Device 2 will be striped across a single die, and Super Blocks in Virtual Device 9 will be striped across 4 dies. Moreover, the weights for the first three read queues for Virtual Device 2 will be overridden, and the value for the first read queue for Virtual Device 3 is also overridden.

19.2.2 Deleting Virtual Devices

Virtual Devices can only be deleted all at once. The `delete` action is used to delete the Virtual Devices within an SEF Unit. The Virtual Devices should be empty before they can be deleted. In other words, if a QoS Domain is created on a Virtual Device they should be deleted before deleting all Virtual Devices. Directions for deleting a QoS Domain is found in 19.3.2.

The following command deletes the Virtual Devices in SEF Unit 0.

Prerequisite: Should have an SEF Unit with created Virtual Devices. Create Virtual Devices following directions in Section 19.2.1

```
$ sudo sef-cli delete virtual  \
    --sef-index 0              \
    --force                    \
    --verbose
All Virtual Devices in SEF Unit 0 were successfully deleted
$
```

19.2.3 Common Actions

Prerequisite: All actions in this section should have an SEF Unit with created Virtual Devices. Create Virtual Devices following directions in Section 19.2.1

The `list` action lists the Virtual Devices. Moreover, if the `--verbose / -V` flag is provided, `list` will print the command to recreate the same Virtual Device configuration.

The following command shows how to list all created Virtual Devices in SEF Unit 0.

```
$ sudo sef-cli list virtual-device    \
    --sef-index 0                     \
    --verbose
Example output
Virtual Device ID       Flash Cap      Flash Avail    SB Cap     SB
   Dies
Virtual Device 2        356679680      356565000      114688     8
Virtual Device 3        181608448      181551112      57344      4
Virtual Device 4        182411264      182353928      57344      4
Virtual Device 9        350715904      350601224      114688     8
Virtual Device Count 4
Recreate Command:
sef-cli create virtual-device -s 0 --die-map "2 2 0 0 0 0 9 9 2 2 4
   4 3 3 9 9 2
 2 4 4 3 3 9 9 2 2 0 0 0 0 9 9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0"
$
```

The `draw` action prints the current configuration of the Virtual Devices within the SEF Unit. `draw` is useful when trying to picture the current location of the created Virtual Devices.

The following command draws current configuration of Virtual Devices in SEF Unit 0.

```
$ sudo sef-cli draw virtual-device    \
    --sef-index 0
Die Map:
2    2    0    0    0    0    9    9
2    2    4    4    3    3    9    9
2    2    4    4    3    3    9    9
2    2    0    0    0    0    9    9
```

```
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
0     0     0     0     0     0     0     0
$
```

The `info` action prints detailed information about the Virtual Device.

The following command prints Virtual Device information for Virtual Device 2 at SEF Unit 0

```
$ sudo sef-cli info virtual-device      \
    --sef-index 0                       \
    --virtual-device-id 2               \
    --verbose
VirtualDeviceId: 2
flashCapacity: 356679680
flashAvailable: 356565000
pSLCFlashCapacity: 286720
pSLCFlashAvailable: 172040
superBlockCapacity: 114688
pSLCSuperBlockCapacity: 28672
maxOpenSuperBlocks: 4294967295
numPSLCSuperBLocks: 10
suspendConfig:
* maxTimePerSuspend: 0
```

```
* minTimeUntilSuspend: 0
* maxSuspendInterval: 0
superBlockDies: 8
aduOffsetBitWidth: 24
superBlockIdBitWidth: 24
Read Queues Weights (1):
* Read Weight 32
QoS Domains: None
Dies (8):
* Die 0
* Die 1
* Die 8
* Die 9
* Die 16
* Die 17
* Die 24
* Die 25
eraseCount: 0
numUnallocatedSuperBlocks: 3110
numSuperBlocks: 0
numUnallocatedPSLCSuperBlocks: 10
numPSLCSuperBlocks: 0
averagePEcount: 0
maxPEcount: 0
patrolCycleTime: 26416
$
```

19.2.4 Configuring a Virtual Device

The Virtual Device engine provides `set-pslc` and `set-suspend-config` actions to further configure a Virtual Device.

The `set-pslc` action is used to set the number of pSLC Super Blocks for the Virtual Device.

The following command sets the pSLC Super Blocks for Virtual Device 1.

Prerequisite: Should have an SEF Unit with created Virtual Devices. Create Virtual Devices following directions in Section 19.2.1

```
$ sudo sef-cli set-pslc virtual-device  \
    --sef-index 0                       \
    --virtual-device-id 2               \
```

```
    --num-pslc-super-block 50           \
    --force --verbose
The number of pSLC Super Blocks for Virtual Devices 2 was
   successfully set
$
```

The `set-suspend-config` action is used to set the configuration for the suspend functionality of the Virtual Device. Not all options are required when setting the suspend configuration.

Prerequisite: Should have an SEF Unit with created Virtual Devices. Create Virtual Devices following directions in Section 19.2.1

```
$ sudo sef-cli set-suspend-config virtual-device    \
    --sef-index 0                                   \
    --virtual-device-id 2                           \
    --suspend-max-time-per 100                      \
    --suspend-min-time-until 100                    \
    --suspend-max-interval 100                      \
    --force                                         \
    --verbose
$
```

19.3 QoS Domain Target

The `qos-domain` target is used to list, create, delete, backup, restore, modify, and get information about QoS Domains.

19.3.1 Creating a QoS Domain

The `create` action of the `qos-domain` target is used to create a QoS Domain. This action supports the `--force / -f` flag, for creating the QoS Domain without asking for confirmation.

When creating a QoS Domain, the QoS Domain ID, a unique identifier of the QoS Domain, is returned by the SEF Unit. This can make scripting difficult, given that the returned ID is non-deterministic. A QoS Domain's label is a 128-bit unique signature used to identify the QoS Domain. These labels can be used to find and perform operations on QoS Domains. The label is stored on the last two root pointers of the QoS Domain. The `--label / -l` option is used to label a QoS Domain. The QoS Domain can be relabeled using the `label` action.

Prerequisite: Should have an SEF Unit with created empty Virtual Devices. Create Virtual Devices following directions in Section 19.2.1

```
$ sudo sef-cli create qos                          \
    --sef-index 0                                  \
    --virtual-device-id 1                          \
    --label "61300673829223 888776657572036"       \
    --flash-capacity 1024000                       \
    --defect-strategy "kPacked"                    \
    --verbose                                      \
    --force
Warning: The value of the erase weight 0 is below the recommended
   number of 256
Warning: The value of the program weight 0 is below the recommended
   number of
 256
The QoS Domain 1 was successfully created
$
```

The example above creates a QoS Domain with label of `61300673829223 888776657572036` on the SEF Unit 0 and Virtual Device ID of 1 with a size of 1024000 ADUs. In other words, the created QoS Domain has a size of 4,194,304 MB, assuming the default ADU size of 4096 bytes. If the Virtual Device does not have enough space available, the QoS Domain is not created, and an error is returned. This action assumes many default options, such as those pertaining to WFQ queues.

Creation Time Configuration

Additional configurations can be applied while creating a QoS Domain.

The option `--read-queue` is used to set the default read queue that should be used for read operations. The options `--program-weight` and `--erase-weight` are used to set the default program and erase weights.

The example below creates a similar QoS Domain to the previous example but overrides the assigned queues and the default values for program and erase weights.

Should have an SEF Unit with created empty Virtual Devices. Create Virtual Devices following directions in Section 19.2.1

```
$ sudo sef-cli create qos                          \
    --sef-index 0                                  \
    --virtual-device-id 2                          \
    --label "61300673829223 888776657572036"       \
    --flash-capacity 1024000                       \
    --read-queue 1                                 \
```

```
    --program-weight 500                                \
    --erase-weight 500                                  \
    --defect-strategy "kPacked"                         \
    --verbose                                           \
    --force
The QoS Domain 1 was successfully created
$
```

19.3.2 Deleting a QoS Domain

QoS Domains can be deleted one at a time. The `delete` action is used to delete a specific QoS Domain.

The following command deletes the QoS Domain with specified label.

Prerequisite: Should have a Virtual Device with created a QoS Domain. Create QoS Domain following directions in Section 19.3.1

```
$ sudo sef-cli delete qos-domain                    \
    --sef-index 0                                   \
    --label "61300673829223 888776657572036"        \
    --force                                         \
    --verbose
The QoS Domain was successfully deleted
$
```

19.3.3 Common Actions

Prerequisite: All actions in this section should have a Virtual Device with created a QoS Domain. Create QoS Domain following directions in Section 19.3.1

The `list` action lists the QoS Domains defined in an SEF Unit. Moreover, if the `--verbose` or `-V` flag is provided, `list` will print the command to recreate the same QoS Domain configuration.

The following command lists all the created QoS Domains in SEF Unit 0.

```
$ sudosef-cli list qos-domain  \
    --sef-index 0              \
    --verbose
QoS Domain ID       VD ID    Capacity     Label
QoS Domain 1        2        1024000
   [61300673829223,888776657572036]
QoS Domain Count 1
```

```
Recreate Command:
sef-cli create qos-domain -s 0 -v 2 --flash-capacity 1024000
 --max-open-super-blocks 2 --defect-strategy "kPacked"
$
```

The `info` action prints detailed information about the QoS Domain.

The following command prints QoS Domain Information for QoS Domain with specified label in SEF Unit 0

```
$ sudo sef-cli info qos-domain                          \
    --sef-index 0                                       \
    --label "61300673829223 888776657572036"            \
    --verbose
qosDomainId: 1
qosDomainLabel: [61300673829223, 888776657572036]
virtualDeviceID: 2
numPlacementIDs: 1
maxOpenSuperBlocks: 2
encryption: Enabled
recoveryMode: Automatic
defectStrategy: Packed
api: SuperBlock
flashCapacity: 1024000
flashQuota: 1024000
flashUsage: 0
pSLCFlashCapacity: 0
pSLCFlashQuota: 0
pSLCFlashUsage: 0
ADUSize:
* Data: 4096
* Meta: 0
superBlockCapacity: 114688
pSLCSuperBlockCapacity: 28672
maxOpenSuperBlocks: 2
defectMapSize: 0
deadline: Typical
defaultReadQueue: 0
Weights:
* erase: 256
* program: 256
```

```
* rootPointer (0): 0x0
* rootPointer (1): 0x0
* rootPointer (2): 0x0
* rootPointer (3): 0x0
* rootPointer (4): 0x0
* rootPointer (5): 0x0
* rootPointer (6): 0x37c0ace35967
* rootPointer (7): 0x3285670f590c4
$
```

19.3.4 QoS Domain Label

A QoS Domain can either be labeled when it is created or using the `label` action. The `label` action can override a QoS Domain that is already labeled if the `--force / -f` flag is provided.

Prerequisite: Should have a Virtual Device with created a QoS Domain. Create QoS Domain following directions in Section 19.3.1

The following command labels a created QoS Domain with QoS Domain Id 1 on SEF Unit 0.

```
$ sudo sef-cli label QoS-Domain                \
    --sef-index 0                              \
    --qos-domain-id 1                          \
    --label "61300673829223 888776657572036"   \
    --verbose --force
QoS Domain 1 was successfully labeled
$
```

19.3.5 Resizing a QoS Domain

Prerequisite: All actions in this section should have a Virtual Device with created a QoS Domain. Create QoS Domain following directions in Section 19.3.1

The `resize` action is used to modify an existing QoS Domain's size. The action can resize both QLC and pSLC capacity and quota; however, the operations have to be done using two separate commands.

```
$ sudo sef-cli resize qos-domain               \
    --sef-index 0                              \
    --label "61300673829223 888776657572036"   \
    --flash-capacity 1024000                   \
    --flash-quota    1024000                   \
    --force --verbose
QoS Domain 1 was successfully resized
```

```
$ sudo sef-cli resize qos-domain              \
    --sef-index 0                             \
    --label "61300673829223 888776657572036"  \
    --pslc-capacity 1024000                   \
    --pslc-quota    1024000                   \
    --force --verbose
QoS Domain 1 was successfully resized
$
```

The `--flash-quota` option is shared for both resize types and will be used accordingly.

19.3.6 Formatting a QoS Domain

The `format` action is used to delete all the data stored on the QoS Domain. In other words, the action does not delete the QoS Domain but just the data that is stored, including root pointers and all the metadata.

The `--relabel` flag will make sure to retain the current label of the QoS Domain.

Prerequisite: Should have a Virtual Device with created a QoS Domain. Create QoS Domain following directions in Section 19.3.1

```
$ sudo sef-cli format qos-domain              \
    --sef-index 0                             \
    --label "61300673829223 888776657572036"  \
    --relabel                                 \
    --force --verbose
QoS Domain 1 was successfully formatted
$
```

19.3.7 Backing Up and Restoring QoS Domains

`SEF-CLI` provides actions to back up and restore the data and metadata stored within a QoS Domain. The data is backed up and restored as-is. In other words, the data and structures stored on the QoS Domain are neither updated nor modified while backing up or restoring. For example, if the data has structures that point to each other, the references and addresses are not changed and will break.

The data is stored in separate files. Each file contains the data for a single Super Block.

The `--path` option points to the directory where the backed up data is stored. The directory is made by the backup process. The command below backs up the data of the QoS Domain in its entirety in the directory.

Prerequisite: Should have a Non-empty QoS Domain. Create QoS Domain following directions in Section 19.3.1. Populate the QoS Domain using FIO explained in Chapter 21 or Python Sell explained in Section 19.5

```
$ sudo sef-cli backup qos                          \
    --sef-index 0                                  \
    --label "61300673829223 888776657572036"       \
    --path "/tmp/SefBackup"
$
```

The following command will restore a backed up QoS Domain's data and metadata from the generated files.

Prerequisite: Should have a empty QoS Domain. Create QoS Domain following directions in Section 19.3.1 or format an existing domain following directions in Section 19.3.6

```
$ sudo sef-cli restore qos                         \
    --sef-index 0                                  \
    --label "61300673829223 888776657572036"       \
    --path "/tmp/SefBackup"
$
```

The above command restores a backed up QoS Domain's data stored in the sub-directory named `/tmp/SefBackup` into another QoS Domain.

Use of both the user address `allow-list` and `block-list` options allows for customization of what is backed up and restored. The `allow-list` and `block-list` options expect a space-separated list of hex flash addresses. The flash addresses can denote a single address or a range of addresses separated by a colon.

Prerequisite: Should have a Non-empty QoS Domain. Create QoS Domain following directions in Section 19.3.1. Populate the QoS Domain using FIO explained in Chapter 21 or Python Sell explained in Section 19.5

```
$ sudo sef-cli backup qos                          \
    --sef-index 0                                  \
    --label "61300673829223 888776657572036"       \
    --path "/tmp/SefBackup"                        \
    --block-list "AAAA BBBB:CCCC DDDD"
$
```

The above command skips the flash addresses AAAA and DDDD and the range of addresses from BBBB to CCCC while backing up the QoS Domain. *Note: The values AAAA, BBBB, CCCC, and DDDD are used for simplicity and should be replaced with valid hex flash addresses.*

19.4 FTL Target

The `ftl` target offers three actions: `info`, `check`, and `configure`. The `configure` action is used to configure the Reference FTL by setting the over-provisioning percentage. Only an empty QoS Domain can be configured. This action supports the `--force / -f` flag to configure the reference FTL without asking for confirmation.

The following command configures the reference FTL for QoS Domain 2 with 20% of over-provisioning.

```
$ sudo sef-cli configure ftl                    \
    --sef-index 0                               \
    --label "61300673829223 888776657572036"    \
    --overprovisioning 20                       \
    --force --verbose
The QoS Domain 1 was successfully configured by SDK
$
```

The `info` action can be used to print detailed information about a configured QoS Domain.

```
$ sudo sef-cli info FTL                         \
    --sef-index 0                               \
    --label "61300673829223 888776657572036"    \
    --verbose
aduSize: 4096
numPlacementIDs: 1
superBlockSize: 28672
superPageSize: 16
capacity: 335868
overprovisioning: 20
numDomains: 1
Recreate Command:
sef-cli configure sdk -s 0 -q 1 --overprovisioning 20
$
```

The health of the reference FTL may be checked using the `check` action. The `check` action can only be performed on a configured QoS Domain. If the FTL is found to be faulty, this action may be used to repair the device when given the `--should-repair` flag. The below command checks the health of the reference FTL. Keep in mind that the reference FTL should not be running while executing the following:

```
$ sudo sef-cli check FTL                        \
    --sef-index 0                               \
```

```
    --label "61300673829223 888776657572036"  \
    --verbose
The SEF SDK was not broken and does not need repair
$
```

19.5 Shell Target: Using an SEF Unit Interactively

A unique feature of the `SEF-CLI` is the interactive Python 3 shell. `SEF-CLI` expands the Python functionality by defining a custom Python module used to interact with the SEF Unit.

The `sefCliHelp` function may be used to print the help function within the shell.

The following is a simple example of how to interact with the built-in Python 3 shell.

```
$ sudo sef-cli interactive shell
>>> sefCli has been loaded, use sefCliHelp() to learn about its use
   cases
>>> use function quit() to exit the interactive mode
[<qos>@<sef>] >>> listSEF()       #Lists sef units
[0, 1, 2]
[<qos>@<sef>] >>> selectSEF(0)    #Selects sef unit 0 to interact
   with
[<qos>@    0] >>> listQoS()       #Lists QoS Domains and if they
   are open
[(3, False)]
[<qos>@    0] >>> openQoS(3)      #Opens QoS Domain to interact with
[    3@    0] >>> quit()          #Shuts down the shell
$
```

Moreover, the Python shell can be used to execute a Python script to interact with SEF Units using the `execute` action. The following command executes a Python 3 script stored in the file `script.py`.

```
$ sudo sef-cli execute shell --python-script script.py
```

20 | Extending SEF CLI

SEF-CLI was created with extensibility and improvement in mind; its functionality may be easily extended by adding new targets and by adding new actions to existing targets. Similarly, the functionality of existing actions may be easily updated or augmented. A good use case for the extensibility of SEF-CLI is configuring custom code without the need for creating a separate app. For example, SEF-CLI has been extended by adding an ftl target to interact with the reference FTL module. In the same way, SEF-CLI may be extended to interact with user-defined targets.

To get started, begin by examining the existing targets. The targets are located in the engines directory. Moreover, an example null target engines/null.c has been added, which can be used as a model or template for building another desired custom target.

20.1 DevTools

The devtool directory includes development-only code to generate the man page and the auto-complete script. After making any changes to SEF-CLI, these tools may be used to generate a new man page and auto-complete script. In order to use the DevTools, the DEVTOOL macro should be defined. CMake option sef_cli_enable_devtool will define the DEVTOOL macro and enable use of the DevTools.

The following commands enable the DevTools while building sef-cli and shows how to generate the man page and the auto-complete script.

```
$ cd <path to SEF_SDK>/cli
$ cmake -Dsef_enable_devtool=ON ..
$ make

$ ./sef-cli --man-page -V > ../sef-cli.1

$ ./sef-cli --auto-complete > ../sef-cli_completion.sh
```

Part VI

Flexible I/O Tester (FIO)

21 | Flexible I/O Tester (FIO)

The Software-Enabled Flash™ (SEF) patch for `fio` adds SEF as an I/O engine. The SEF I/O engine uses the SEF Block Module to issue block I/O against a defined QoS Domain. In other words, the QoS Domain which is used should be defined before the job is executed. If the QoS Domain is not configured, the engine will configure it using default values.

The SEF FIO Engine Options Table enumerates the options accepted by the SEF I/O engine. `qos_domain_id` or `qos_domain_label` option should be provided to identify the target QoS Domain. If both options are provided, `qos_domain_id` is used as default. The `qos_domain_label` should be unique across the SEF Unit or the job will fail.

Table 21.1: SEF FIO Engine Options

Option	Type	Required	Description
sef_unit_index	int (16 bit)	Yes	The index of the SEF Unit, the index is zero based
qos_domain_id	int (16 bit)	Yes*	The Id of the QoS Domain
qos_domain_label	uint (Two 64 bits)	Yes*	The label of the QoS Domain
sef_zero_copy	bool	No	Use zero copy buffers
read_queue	int (16 bit)	No	kSEFRead queue override when valid
read_weight	int (16 bit)	No	Read weight to override when non-zero
write_weight	int (16 bit)	No	Write weight to override when non-zero
placement_id	int (16 bit)	No	Placement ID for writes

Listing 21.1 is an example job file that runs for 10 minutes. It contains two jobs, both operating on SEF Unit 0. The first randomly writes to QoS Domain 2, and the second reads and writes to a QoS Domain labeled as `61300673829223,888776657572036`.

It's important to note the `thread` option. It's required because the SEF Block Module and SEF libraries do not currently support forked processes.

Listing 21.1: Example SEF Engine FIO Job

```
[global]
ioengine=sef
sef_unit_index=0
bs=4k
time_based=1
runtime=600
verify=md5
iodepth=16
thread

[sef-write]
qos_domain_id=2
rw=randwrite
size=100%

[sef-read-write]
qos_domain_label=61300673829223,888776657572036
rw=randrw
size=100%
```

21.1 Buddy Allocator

The `sef_zero_copy` option of the SEF FIO Engine uses the Buddy Allocator to write the data.

The Buddy Allocator works by associating fragmented memory with the incoming I/O requests and freeing them as I/O get completed. The buddy allocator allocates blocks of memory in multiples of a fixed size—in this case, the ADU size. A pool of memory is pre-allocated by the buddy allocator in a power-of-2 of the fixed size. Free lists are maintained for each power-of-2 from the fixed size to the size of the whole pool. The buddy allocator begins searching for free blocks at the smallest power-of-2 which is larger than the requested allocation. If space can't be found there, the next largest level is checked until space is found. The block there will be split, and the unallocated portion of the block will be assigned to the lower power-of-2 free list. When blocks are freed, a "buddy" is looked for in the matching power-of-2 list, and if found, the blocks are combined and moved to the higher power-of-2 list.

Unlike a regular heap, it is possible to use blocks from within a previous allocation.

For the SEF FIO engine, we allocate a pool that is larger than the size of the outstanding I/O, and the blocks are returned to the buddy allocator when the nameless write completes.

Part VII

QEMU

22 | QEMU

The Software-Enabled Flash™ (SEF) Software Development Kit (SDK) includes QEMU storage devices that allow non-SEF enabled software to be tested with an SEF Unit. The storage devices translate block APIs into SEF commands storing the data in an SEF Unit. The storage devices included are:

- An SEF-Backed Virtual Driver
- An SEF-Backed NVMe Driver with FDP support
- An SEF-Backed ZNS Device

To use them, an SEF Unit and SEF QoS Domain are supplied as parameters. The storage devices then present a block device to the QEMU guest based on the SEF QoS Domain's configuration. See each sub-section for details on how to use them.

22.1 SEF-Backed Virtual Driver

The SEF patches for QEMU add support for a virtual driver called sef-aio. Instead of using a file for a backing store, it uses an SEF QoS Domain. The QoS Domain must be unconfigured or configured as an SEF Block FTL domain. An unconfigured domain will be auto configured as an SEF Block FTL domain when it's mounted. The file option selects the unit and QoS Domain and has the following format: `file=<unit>:<domain>`. Below is an example of a QEMU command line with options for an SEF Virtual Device on SEF Unit 0 and QoS Domain 3.

```
$sudo qemu -name sefAio -m 4G -smp 8 -enable-kvm                    \
  -drive file=system-debian.qcow2,cache=none,format=qcow2,if=virtio\
  -netdev user,id=vnet,hostfwd=tcp:127.0.0.1:2222-:22               \
  -device virtio-net-pci,netdev=vnet -nographic                     \
  -display none -cpu host                                           \
  -drive driver=sef-aio,file=0:3,cache=none,if=virtio
```

To verify the virtual drive is present, run the command `lsblk`. In this case, it's the device vdb.

```
$sudo lsblk
```

```
NAME    MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
fd0       2:0    1    4K  0 disk
sr0      11:0    1 1024M  0 rom
vda     254:0    0  100G  0 disk
+-vda1  254:1    0   96G  0 part /
+-vda2  254:2    0    1K  0 part
+-vda5  254:5    0    4G  0 part [SWAP]
vdb     254:16   0  4.3G  0 disk
```

FIO can be used for testing the device from within the virtual machine. Below is an example that randomly reads and writes to the device.

```
$sudo fio -ioengine=libaio -filename=/dev/vdb -name=rw -randrw
```

22.2 SEF-Backed ZNS Device

The standard NVMe device for QEMU supports ZNS. Using it requires that the guest OS has a Linux kernel of 5.9 or later. It's enabled with the option zoned=true. The SEF patches for QEMU add support for using an SEF QoS Domain as a backing store. Table 22.1 SEF-specific ZNS Device Options shows the SEF-specific options. The SEF QoS Domain used must be unconfigured or configured as an SEF ZNS domain. An unconfigured domain will be auto configured as an SEF ZNS domain.

Table 22.1: SEF-specific ZNS Device Options

Option	Default Value	Description
sef	false	If true, switches from a file as a backing store to an SEF QoS Domain. In this mode, a zone is an SEF Super Block, which dictates the default zone. Setting the zone size smaller is supported but will waste space.
sef_unit	0	Unit number of the SEF Unit to use as the backing store.
sef_qos_domain	2	QoS Domain ID to use as the backing store.

Below is an example of a QEMU command line with options for an SEF-backed ZNS device on QoS Domain 3. Although unused, it's still required to supply a valid backing file, even if it's zero-sized. The example is using the default of 0 for the SEF Unit number and explicitly configuring QoS Domain 3.

```
$sudo qemu -name sefZns -m 4G -smp 8 -enable-kvm                    \
  -drive file=system-debian.qcow2,cache=none,format=qcow2,if=virtio\
  -netdev user,id=vnet,hostfwd=tcp:127.0.0.1:2222-:22               \
```

```
  -device virtio-net-pci,netdev=vnet -nographic                    \
  -display none -cpu host                                          \
  -drive if=none,id=nvme1,file=nvme3.raw,format=raw                \
  -device nvme,serial=654321                                       \
  -device nvme-ns,drive=nvme1,nsid=1,sef=true,zoned=true,          \
   sef_qos_domain=3
```

To verify the ZNS device is present, run the command `lsblk -z`.

```
$sudo lsblk -z

NAME    ZONED
fd0     none
sr0     none
vda     none
+-vda1  none
+-vda2  none
+-vda5  none
nvme0n1 host-managed
```

The indication "host managed" tells you that device nvme0n1 is a ZNS device.

FIO can be used for testing ZNS devices from within the virtual machine. Below is an example that randomly reads and writes 8 gigabytes of data using 128k I/O requests. The request types are evenly split between reads and writes, yielding 4 gigabytes of reads and 4 gigabytes of writes.

```
$sudo fio --aux-path=/tmp --allow_file_create=0 --name=job1 \
  --filename=/dev/nvme0n1 --rw=randrw --direct=1            \
  --zonemode=zbd --bs=128k --size=8G
```

When run only once, it will not fill the device since half of the size is used for reads, and size will be rounded down to the device size if larger. To see how much data has been written, run `blkzone report /dev/nvme0n1` for the write pointer value of each zone. By running the fio job more than once, it will eventually fill a zone, which will then be reset/erased to allow for more writes.

```
$sudo blkzone report /dev/nvme0n1 | head -2

  start: 0x000000000, len 0x010000, cap 0x010000, wptr 0x009800
     reset:0 non-seq:0, zcond: 2(oi) [type: 2(SEQ_WRITE_REQUIRED)]
  start: 0x000010000, len 0x010000, cap 0x010000, wptr 0x00aa00
     reset:0 non-seq:0, zcond: 2(oi) [type: 2(SEQ_WRITE_REQUIRED)]
```

22.3 SEF-Backed NVMe Device

The SEF patches for QEMU add support for using an SEF QoS Domain as a backing store for a QEMU NVMe device. Table 22.2 SEF-specific NVMe Device Options shows the SEF-specific options. The QoS Domain must be unconfigured or configured as an SEF Block FTL domain. An unconfigured domain will be auto configured as an SEF Block FTL domain when it's mounted.

Table 22.2: SEF-specific NVMe Device Options

Option	Default Value	Description
sef	false	If true, switches from a file as a backing store to an SEF QoS Domain. In this mode, the SEF reference FTL is used to provide block I/O for the QEMU NVMe device.
sef_unit	0	Unit number of the SEF Unit to use as the backing store.
sef_qos_domain	2	QoS Domain ID to use as the backing store.

Below is an example of a QEMU command line with options for an SEF-backed NVMe device on QoS Domain 2. Although unused, it's still required to supply a valid backing file, even if it's zero-sized. The example is using the default of 0 for the SEF Unit number and the default of 2 for the QoS Domain.

```
$sudo qemu -name sefNvme -m 4G -smp 8 -enable-kvm                    \
  -drive file=system-debian.qcow2,cache=none,format=qcow2,if=virtio\
  -netdev user,id=vnet,hostfwd=tcp:127.0.0.1:2222-:22                \
  -device virtio-net-pci,netdev=vnet -nographic                      \
  -display none -cpu host                                            \
  -drive if=none,id=nvme1,file=nvme2.raw,format=raw                  \
  -device nvme,serial=123456                                         \
  -device nvme-ns,drive=nvme1,nsid=1,sef=true
```

To verify the virtual drive is present, run the command `lsblk`. In this case, it's the device vdb.

```
$sudo lsblk

NAME   MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
fd0      2:0    1    4K  0 disk
sr0     11:0    1 1024M  0 rom
vda    254:0    0  100G  0 disk
+-vda1 254:1    0   96G  0 part /
+-vda2 254:2    0    1K  0 part
+-vda5 254:5    0    4G  0 part [SWAP]
nvme0n1 259:0    0  4.3G  0 disk
```

FIO can be used for testing the device from within the virtual machine. Below is an example that randomly reads and writes to the device.

```
$sudo fio –ioengine=libaio –filename=/dev/nvme0n1 –name=rw –randrw
```

Part VIII

NVMe-CLI

23 | NVMe-CLI

The `nvme-cli` has been modified to support SEF Units with the addition of the `Kioxia` plugin.

The `nvme-cli` allows the user to interact with a SEF Unit at a low level without using the library; however, it does not provide as much functionality or user friendliness as SEF-CLI.

After installing the SEF `nvme-cli` command, you will be able to execute the following commands.

```
usage: nvme kioxia <command> [<device>] [<args>]

The following are all implemented sub-commands:
  sef-info              SEF Unit Information
  create-vd             Create a Virtual Device
  create-cap-cfg        Create a Capacity Configuration
  sel-cap-cfg           Select a Capacity Configuration
  delete-vd             Delete a Virtual Device
  create-qosd           Create a QoS Domain
  delete-qosd           Delete a QoS Domain
  setroot-qosd          Set Rootpointer to QoS Domain
  change-cap-qosd       Change the capacity of a  QoS Domain
  change-weights-qosd   Change the weights of a QoS Domain
  read-deadline-qosd    Change the read deadline for a QoS Domain
  max-open-sb-qosd      Change the max number of open Super Blocks
     on a QoS Domain
  def-rq-qosd           Change the read queue for a QoS Domain
  vd-info               Virtual Device Information
  qd-info               QosD Info
  cap-cfg-list          Capacity Config List
  sb-info               Super Block Information
  sb-list               Super Block List
  ua-list               User Address List
  aco-list              Address Change Order
```

```
  rf-list-log           Read Fifo List
  rf-attach             Attach Read Fifo to virtual domain
  rf-detach             Detach Read Fifo from virtual domain
  vd-set-ndies          Set the number of dies for a virtual domain
  vd-set-pslc           Set the pSLC blocks for a virtual domain
  read                  Submit Read command, return results
  write                 Submit FLA Request and NLW command, return
     results
  copy                  Submit NLC command, return results
  sb-manage             Super Block Management
  aen-rq                Asynchronous event change request
```

23.1 SEF Info

Get the SEF Unit Information.

```
$ nvme kioxia sef-info /dev/nvme0
SEF Capability Supports:
        Packed ADU Offset
        List indicated Nameless Copy command
        Super Block operated with Nameless Write and Nameless Copy
           commands
        pSLC Super Block
Number of Channels: 4
Number of Banks: 4
Number of Blocks per Die: 1958
Number of Pages per Block: 384
Number of Planes per Page: 6
Plane Data Size: 14
Expiration Open Period: 43200
Number of Placement IDs: 8
Max Number of Open Super Blocks: 2048
Number of Read FIFOs: 256
Number of Pages to Secure Integrity: 0

----DYNAMIC INFORMATION----
Number of Read FIFOs in Use: 0
Number of Virtual Devices: 0
Number of QoS Domains: 0
$
```

23.2 Virtual Device

23.2.1 Create

The create-vd command requires a Virtual Device descriptor file For example, this will create 1 Virtual Device spanning 4 dies:

```
cd <path to SEF_SDK>/lib
devtool/script_for_cli/gen_create_cap_cfg.sh -f VDFD.out -d "1 1 1
   1"
nvme kioxia create-vd /dev/nvme0 -f
   devtool/script_for_cli/data/VDFD.out
```

23.2.2 Info

Get the Virtual Device Information.

```
$ nvme kioxia vd-info /dev/nvme0 -v 1 -n 16
VDID              1
Critical Warning  0x0
CWC               7
CESN              0
MAX_PE            0
AVG_PE            0
NSBDIE            16
SBBW              23
AOBW              23
RPC               0
MXOSB             0
DRFID             1
NRF               1
NPBLK             0
VDCAP             72105984
VDGCAP            0
VDUCAP            0
ASS_QOSD          0
NDIE              15
DIES -
        000000 000001 000002 000003 000004 000005 000006 000007
        000008 000009 00000a 00000b 00000c 00000d 00000e
00000f
$
```

- VDID: Virtual Device ID
- CWC: Critical Warning Control
  - 7:3 - Reserved
  - 2 - Capacity Reduction Event
  - 1 - Out of pSLC Super Block Event
  - 0 - Out of general Super Block Event
- CESN: Current Erase Serial Number
- MAX_PE: Maximum Program/Erase Count
- AVG_PE: Average Program/Erase Count
- NSBDIE: Number of Dies per Super Block
- SBBW: SuperBlock Bit Width
- AOBW: ADU Offset Bit Width
  - SBBW and AOBW determine the appearance of the SEFFlashAddress
  - Together they can add up to 64 or no less than 32-bits.
  - For example: SBBW = 16, AOBW = 32
  - Super Block bits: x, ADU Offset bits: y
  - SEFFlashAddress: 0000 xxxx yyyy yyyy
- RPC: Recommended Patrol Cycle
- MAXOSB: Maximum Number of Open Super Blocks
- DRFID: Default Read FIFO ID
- NRF: Number of Read FIFOs
- NPBLK: Number of pSLC Blocks
- VDCAP: Virtual Device Total Capacity
- VDGCAP: Virtual Device Guaranteed Capacity
- VDUCAP: Virtual Device In-use Capacity
- ASS_QOSD: Number of QoS Domains assigned

- NDIE: Number of Dies
- DIES: List of Die IDs

23.2.3 Delete Virtual Device

Delete specific Virtual Device by ID or use `0xffff` to delete all Virtual Devices.

```
nvme kioxia delete-vd /dev/nvme0 -v 0xffff
```

23.2.4 Create Capacity Configuration

Create a capacity configuration that assigns a Virtual Device ID to each die on the SEF Unit.

```
cd <path to SEF_SDK>/lib
devtool/script_for_cli/gen_create_cap_cfg.sh -f CAPCFG.ld2 -d "
        1 1 2 2 2 2 0 0 \
        1 1 2 2 2 2 0 0 \
        1 1 3 3 3 3 0 0 \
        1 1 3 3 3 3 0 0 \
        0 0 0 0 0 0 0 0 \
        0 0 0 0 0 0 0 0 \
        0 0 0 0 0 0 0 0 \
        0 0 0 0 0 0 0 0 \
        0 0 0 0 0 0 0 0 \
        0 0 0 0 0 0 0 0 \
        0 0 0 0 0 0 0 0 \
        0 0 0 0 0 0 0 0 \
        0 0 0 0 0 0 0 0 \
        0 0 0 0 0 0 0 0 \
        0 0 0 0 0 0 0 0 \
        0 0 0 0 0 0 0 0 \
        0 0 0 0 0 0 0 0 \
        0 0 0 0 0 0 0 0 \
        0 0 0 0 0 0 0 0 \
        0 0 0 0 0 0 0 0 \
        0 0 0 0 0 0 0 0 \
        0 0 0 0 0 0 0 0 \
        0 0 0 0 0 0 0 0";
nvme kioxia create-cap-cfg /dev/nvme0 -f
   devtool/script_for_cli/data/CAPCFG.ld2
```

23.2.5 Cap Config List

```
$ nvme kioxia cap-cfg-list /dev/nvme0
CCI                 1
DI                  0
EGCN        16
  ENDGID            1
  CAF               0
  EGSETS            0
  EGCHANS           1
        CI                          -1
        CHMUS               1
          MU        0,   ML          0
...
$
```

- CCI: Capacity Configuration Index
- DI: Domain Identifier
- EGCN: Endurance Group Channel Number
- ENDGID: Endurance Group ID
- CAF: Capacity Adjustment Factor
- EGSETS: Number of NVM Sets
- EGCHANS: Number of Channels
- CI: Channel Identifier
- CHMUS: Number of Channel Media Units
- MU: Media Unit Identifier
- ML: Media Unit Descriptor Length

23.2.6 Select Cap Config

After creating a secondary capacity configuration, swap between them.

```
nvme kioxia sel-cap-cfg /dev/nvme0 -c 2
```

23.2.7 Set Number of Dies for Virtual Device

```
nvme kioxia vd-set-ndies /dev/nvme0 -v1 -n8
```

23.2.8 Set Number of pSLC blocks for a Virtual Device

```
nvme kioxia vd-set-pslc /dev/nvme0 -v1 -n10
```

23.3 QoS Domain

A Virtual Device should be created before the following actions can be performed.

23.3.1 Create QoS Domain

```
cd <path to SEF_SDK>/lib
devtool/script_for_cli/gen_create_qosd_bin.sh -v 1 -s 0 -p 8 -c
   50000 -f QOSFD.out
nvme kioxia create-qosd /dev/nvme0 -q 1 -f
   devtool/script_for_cli/data/QOSFD.out
```

23.3.2 Delete QoS Domain

```
nvme detach-ns /dev/nvme0 -n1 -c1
nvme kioxia delete-qosd /dev/nvme0 -q 1
```

23.3.3 QoS Domain Info

```
$ nvme kioxia qd-info /dev/nvme0 -q 1
QoS Domain Size:         288276504
QoS Domain Capacity:     288276504
QoS Domain Utilization:  0
Virtual Device ID:       1
Read Deadline:           0
Defect Management Type:  0
Number of PLIDs(0's):    0
Max Open SuperBlocks:    0
Write Weight:            0
Erase Weight:            1
Default Read FIFO ID:    257
SuperBlock Statistics
  Max Planes:           18449696256
  Guaranteed Planes:    18449696256
  Current Planes:       0
  Active SuperBlocks:   0
  Open SB for NLW:      0
  Open SB for Erase:    0
pSLC SuperBlock Statistics
```

```
  Max Planes:           0
  Guaranteed Planes:    0
  Current Planes:       0
  Active SuperBlocks:   0
  Open SB for NLW:      0
  Open SB for Erase:    0
Root Pointer 0:          0x0
...
$
```

23.3.4 QoS Domain Capacity

```
nvme kioxia change-cap-qosd /dev/nvme0 -q1 -t0 -g4096
```

23.3.5 QoS Domain Weights (Read/Program)

```
nvme kioxia change-weights-qosd /dev/nvme0 -q1 -e100 -w100
```

23.3.6 Change QoS Domain Read Deadline

```
nvme kioxia read-deadline-qosd /dev/nvme0 -q1 -r2
```

23.3.7 Maximum Number of Open Super Blocks

```
nvme kioxia max-open-sb-qosd /dev/nvme0 -q1 -m256
```

23.3.8 QoS Domain Read Queue

```
nvme kioxia def-rq-qosd /dev/nvme0 -q1 -r8
```

23.3.9 Set Root QoS Pointer

```
nvme kioxia setroot-qosd /dev/nvme0 -q1 -f 0x0002000100000000 -i0
```

23.4 FIFO

A Virtual Device should be created before the following actions can be performed.

23.4.1 Attach a read FIFO

```
nvme kioxia rf-attach /dev/nvme0 -v 1 -f 2 -w 0x32
```

23.4.2 Detach a read FIFO

```
nvme kioxia rf-detach /dev/nvme0 -v 1 -f 2 -w 0x32
```

23.4.3 List available read FIFOs

```
$ nvme kioxia rf-list /dev/nvme0 -v1 -n128
Fifo Id = 0000000001, VD Id = 000001, Weight = 000032
Fifo Id = 0000000002, VD Id = 000002, Weight = 000064
...
$
```

23.4.4 List a specific read FIFO

```
$ nvme kioxia rf-list-log /dev/nvme0 -v 1 -f 1 -l 8
Fifo Id = 0000000001, VD Id = 000001, Weight = 000032
$
```

23.5 Super Block

An available Super Block is required to perform the following commands.

23.5.1 Super Block Info

```
$ nvme kioxia sb-info /dev/nvme0 -v 1 -s 10
VDID            : 1
ST              : <State>
SBS             : NLW (04)
SBID            : 0x00000001
PID             : 0
QOSDID          : 1
PECI            : 192
DII             : Reserved (55)
TL              : 0
CAP             : 147456
ADUPTR          : 00000018
ESN             : 00000001
DGID            : 0
NDPL            : 0
DBM                     :
...
$
```

- VDID: Vritual Device ID
- ST: State Transition
- SBS: Super Block State

- SBID: Super Block ID
- PID: Placement ID
- QOSDID: QoS Domain ID
- PECI: Program/Erase Count Index
- DII: Data Integrity Index
- TL: Time Left
- CAP: Capacity
- ADUPTR: ADU Pointer
- ESN: Erase Serial Number
- DGID: Die Group ID
- NDPL: Number of Defective Planes
- DBM: Defect Bitmap

23.5.2 Superblock List

```
$ nvme kioxia sb-list /dev/nvme0 -q1 -l 0 -s 40 -o0
SB-1, PECI = 1, DII = 4
...
$
```

- SB: Super Block ID
- PECI: Program/Erase Count Index
- DII: Data Integrity Index

23.5.3 Superblock management

Operation:

- 0: Erase
- 1: Close
- 2: Free
- 3: Patrol
- 4: Flush

```
nvme kioxia sb-manage -q 1 -s 10 -o 1 /dev/nvme0
```

23.5.4 User Address List

```
$ nvme kioxia ua-list /dev/nvme0 -q1 -s 10 -l4096 -o0
00000000: UA-00000000, METADATA = 00000001 00000002
00000001: UA-00000003, METADATA = ffffffffffffffff ffffffffffffffff
...
$
```

23.5.5 Address Change Order

Note: If there are no ACO's, the output will be blank.

```
nvme kioxia aco-list /dev/nvme0 -q1 -s0 -l4096 -o0
```

23.6 Write

Prerequisite: Data file that is the size of <# of ADUs> * <ADU Size> Note: An open Super Block must be closed in order to delete a QoS Domain.

```
cd <path to SEF_SDK>/lib
nvme kioxia write -a 4 -n 1 -f -D
   devtool/nvme-cli-sef/sefWriteData_16K.txt /dev/nvme0
OPENSB=$(nvme kioxia sb-list -q1 -l3 -s8 -o0 /dev/nvme0 | cut -d
   "-" -f2 | cut -d "," -f1)
nvme kioxia sb-manage -q 1 -s $OPENSB -o 1 /dev/nvme0
```

23.7 Read

Prerequisite: Issue a write command Note: A read command requires the source flash address

```
cd <path to SEF_SDK>/lib
nvme kioxia write /dev/nvme0 -a 4 -n 1 -f -D
   devtool/nvme-cli-sef/sefWriteData_16K.txt > WRITE.log
SUA=$(cat WRITE.log | grep "SFLA:" | cut -d ":" -f2 | xargs)
SUA=0x$SUA
OPENSB=$(nvme kioxia sb-list -q1 -l3 -s8 -o0 /dev/nvme0 | cut -d
   "-" -f2 | cut -d "," -f1)
nvme kioxia sb-manage -q 1 -s $OPENSB -o 1 /dev/nvme0
nvme kioxia read /dev/nvme0 -f $SUA -a 4 -d sefReadData_16K.txt -u 0
```

23.8 Copy

Prerequisite: Issue a write command Note: A copy command requires the source flash address The flash address 0xffffffff indicates automatic allocation.

```
cd <path to SEF_SDK>/lib
nvme kioxia write /dev/nvme0 -a 4 -n 1 -f -D
   devtool/nvme-cli-sef/sefWriteData_16K.txt > WRITE.log
SUA=$(cat WRITE.log | grep "SFLA:" | cut -d ':' -f2 | xargs)
SUA=0x$SUA
OPENSB=$(nvme kioxia sb-list -q1 -l3 -s8 -o0 /dev/nvme0 | cut -d
   '-' -f2 | cut -d ',' -f1)
nvme kioxia sb-manage -q 1 -s $OPENSB -o 1 /dev/nvme0
nvme kioxia sb-manage -q 1 -s 0xffffffff -o 0 /dev/nvme0
OPENSB=$(nvme kioxia sb-list -q1 -l3 -s16 -o0 /dev/nvme0 | cut -d
   '-' -f2 | cut -d ',' -f1)
IFS="
"
OPENSBS=( $OPENSB )
echo open superblocks 1:${OPENSBS[0]} 2:${OPENSBS[1]}
devtool/script_for_cli/gen_copy_bitmap_bin.sh -a 4 -b 0 -c 0 -d 0
   -e $SUA -z COPY.out
nvme kioxia copy /dev/nvme0  -q 1 -s ${OPENSBS[1]} -n 256 -b
   devtool/script_for_cli/data/COPY.out -B
   devtool/nvme-cli-sef/sefNlcOBitmap.bin -A
   devtool/nvme-cli-sef/sefNlcACR.bin -R 0xffffffff
```

23.9 Get Log

```
cd <path to SEF_SDK>/lib
nvme kioxia write /dev/nvme0 -a 4 -n 1 -f -D
   devtool/nvme-cli-sef/sefWriteData_16K.txt
nvme kioxia get-log /dev/nvme0 -n 1 -i 0xd5 -l1024 -o0 -s0 -B0 -a0
```

23.10 Asynchronous event change request

Prerequisite: An event must be triggered

```
Usage: nvme kioxia aen-rq <device> [OPTIONS]

Wait on async event notification request
```

```
Options:
  [  --get-log, -l ]                    --- read the log after
     getting the
                                            event
  [  --del-log, -d ]                    --- delete the log entry
     after
                                            reading
  [  --num-events=<NUM>, -n <NUM> ]     --- number of events to
     wait for
  [  --timeout=<NUM>, -t <NUM> ]        --- NVME command timeout in
     ms
```