SDC 20

BY Developers FOR Developers

Storage Developer Conference
September 22-23, 2020

Unlocking the New Performance and QoS Capabilities of the **Software-Enabled Flash™** API

Rory Bolt
Principal Architect and Sr. Fellow
KIOXIA America, Inc.

01

What is Software-Enabled Flash?

Software-Enabled Flash™ Technology

Fundamentally redefines the relationship between host and solid-state storage

- Brings control of media to the host
- Host applications have complete control over storage functionality and behavior
- Solves legacy overhead problems and enables new features
- Maximizes flash flexibility, performance and parallelism…

in other words, its value.

A Software Enabled API

1. A software enabling technology built around an open source flash-native API
2. It delivers a rich interface of functions and tools to simplify storage innovation
3. The API abstracts flash details enabling future generations of flash to work without code changes
4. Allows any flash vendor to build and optimize their flash to the API
5. KIOXIA will provide sample source code and libraries in the future

Software-Enabled Flash Technology

is not software but it is **software enabled**

Hardware and Software Working Together

Purpose-built, media-centric flash controllers focused on hyperscaler requirements.

- Flash vendor/generational abstraction
- Advanced die time scheduling
- Access to entire media
- Host CPU offload
- Flexible DRAM configurations

Open sourced API and libraries providing functionality hyperscalers demand.

- Data placement
- Workload / tenant isolation
- Latency control
- Buffer management
- Adaptable to new workloads

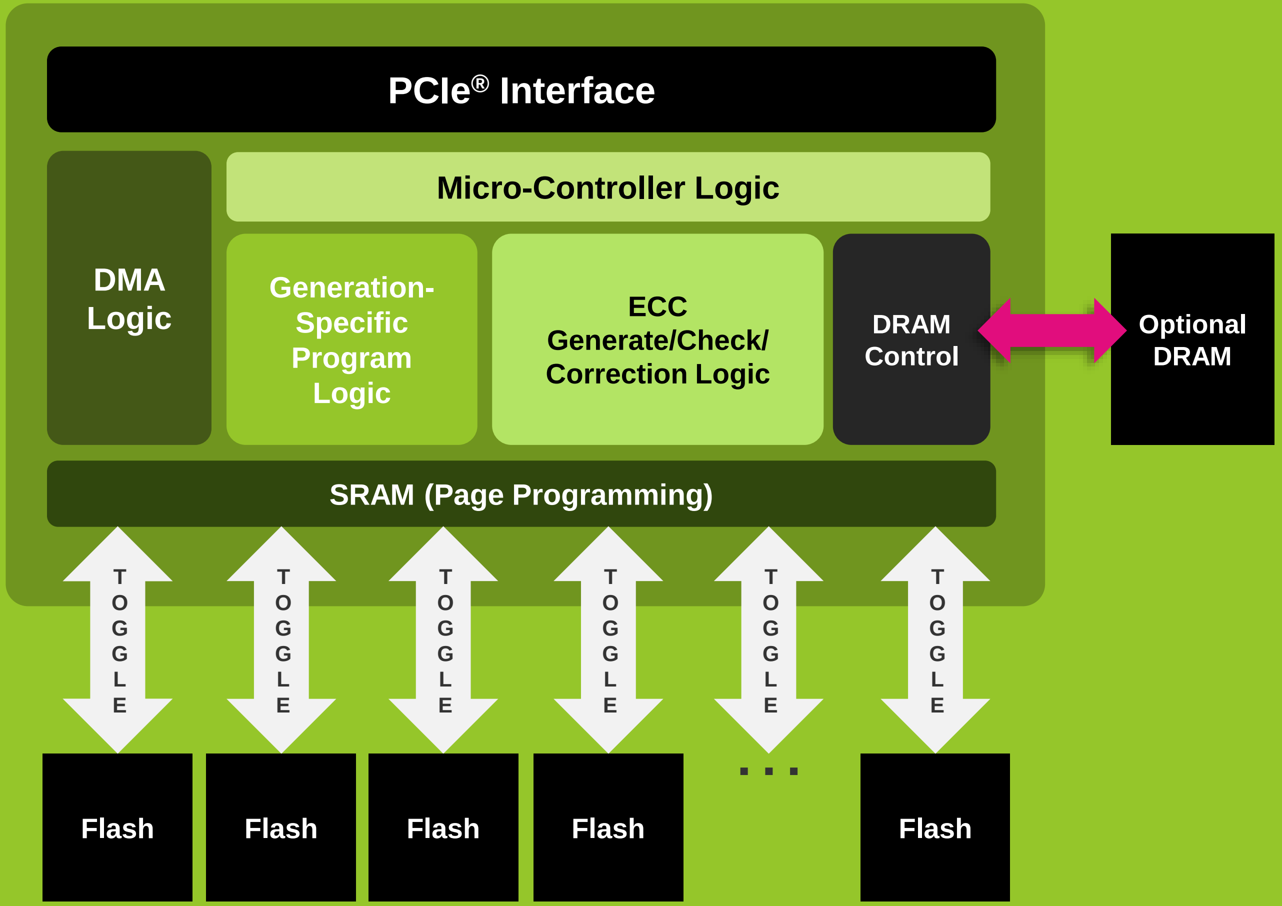
KIOXIA

Provided By the API

Hardware Structure	Buffer Management	Programming Algorithm	Error Management

• Channels/Dies
• Parallelism
• Flash structure
  • Plane/Block/Page
  • Defect management
  • Lifetime management
	• Write buffer
	• Number of open blocks
		• Physical placement
		• Number of passes
		• Voltage threshold
		• Thermal management
			• Error reduction
			• ECC deadline

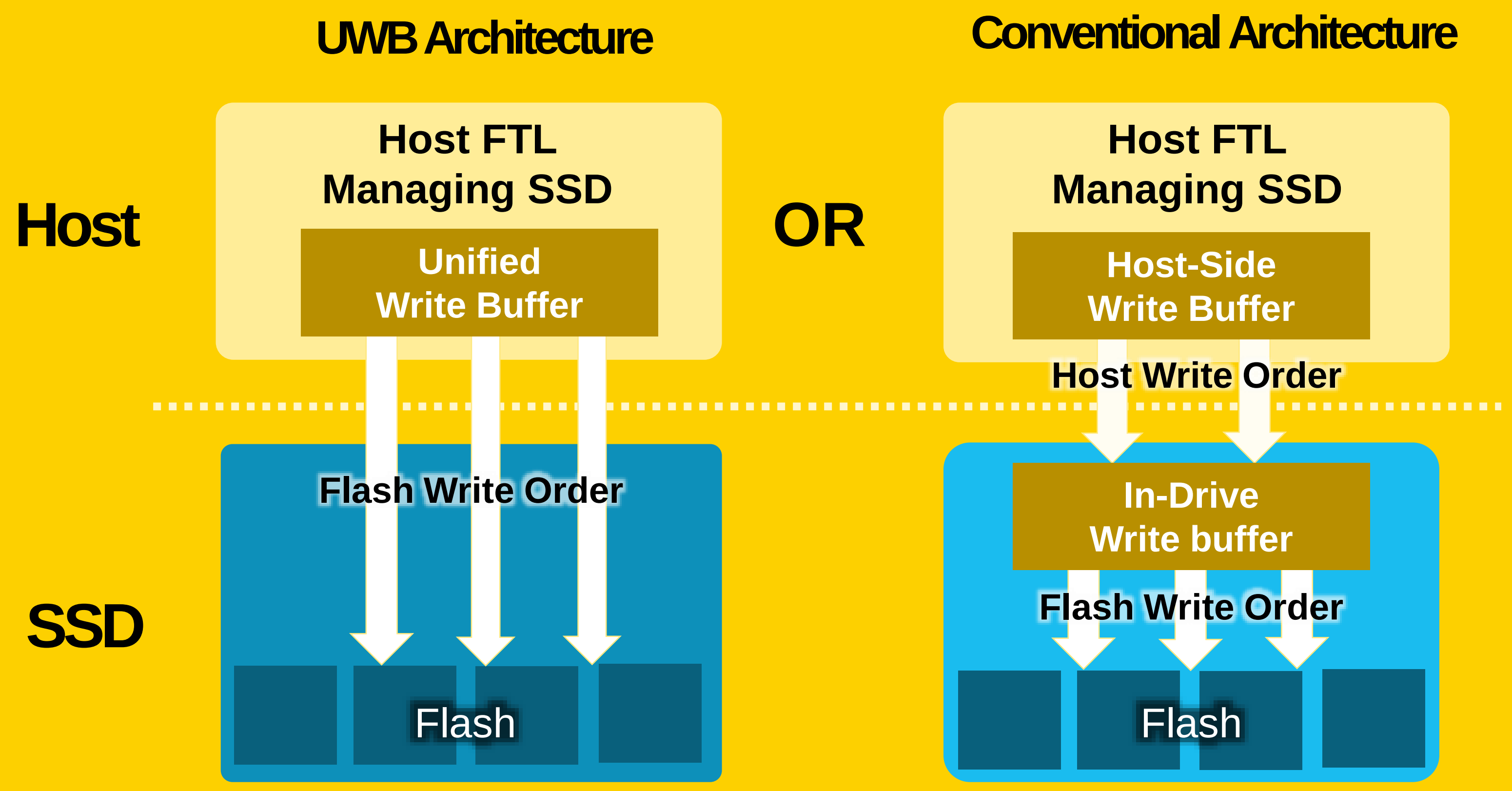
Purpose-Built Hardware to Manage Media



PCIe is a registered trademark of PCI-SIG.

- Controller handles all media details
  - Programming algorithms
  - Lifetime extension
  - Defect management
- Multiple prioritized and weighted queues per die
- Copy offload functionality
- ECC hardware
- Zero DRAM configurations (SRAM only)

Unified Write Buffers (UWB) offers Flexible use of DRAM



Supports both architectures as well as hybrid implementations!

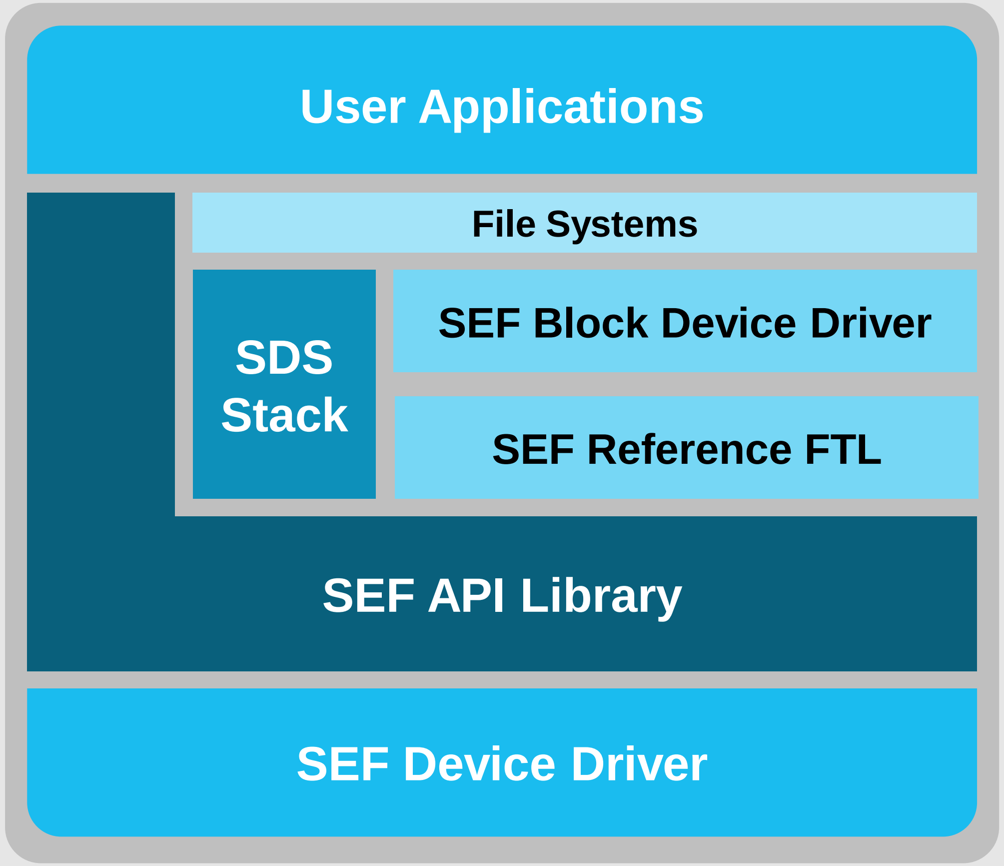
Unified Write Buffer Benefits:

- ✓ Can dynamically adapt to changing workloads
- ✓ Can optimize DRAM at the system level

Conventional Architecture

- ✓ Appropriate for systems that rely on drive power loss protection

Software Components and Layering



A Software Development Kit (SDK)

- Provides an Open Source Linux® reference block driver
- Provides an Open Source Linux® reference FTL

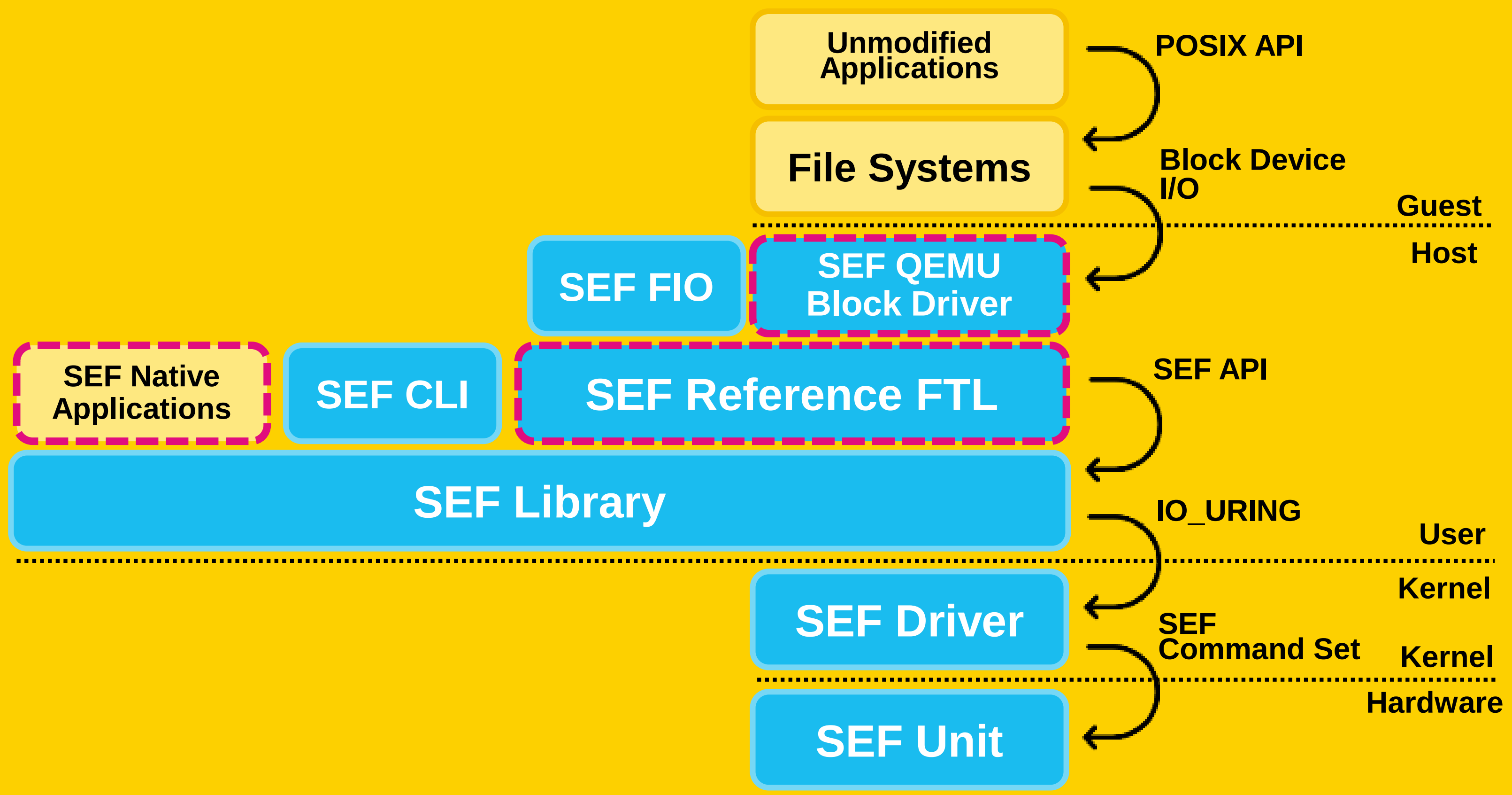
Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.

An API Library

- Open Source Linux interface library that implements the API (Application Programming Interface)
- C language with C++ wrappers, data structures, enumerated types, utility functions
- User space and kernel implementations
- Hides flash generational differences such as programming algorithms
- Controls advanced scheduling

The Software-Enabled Flash Device Driver

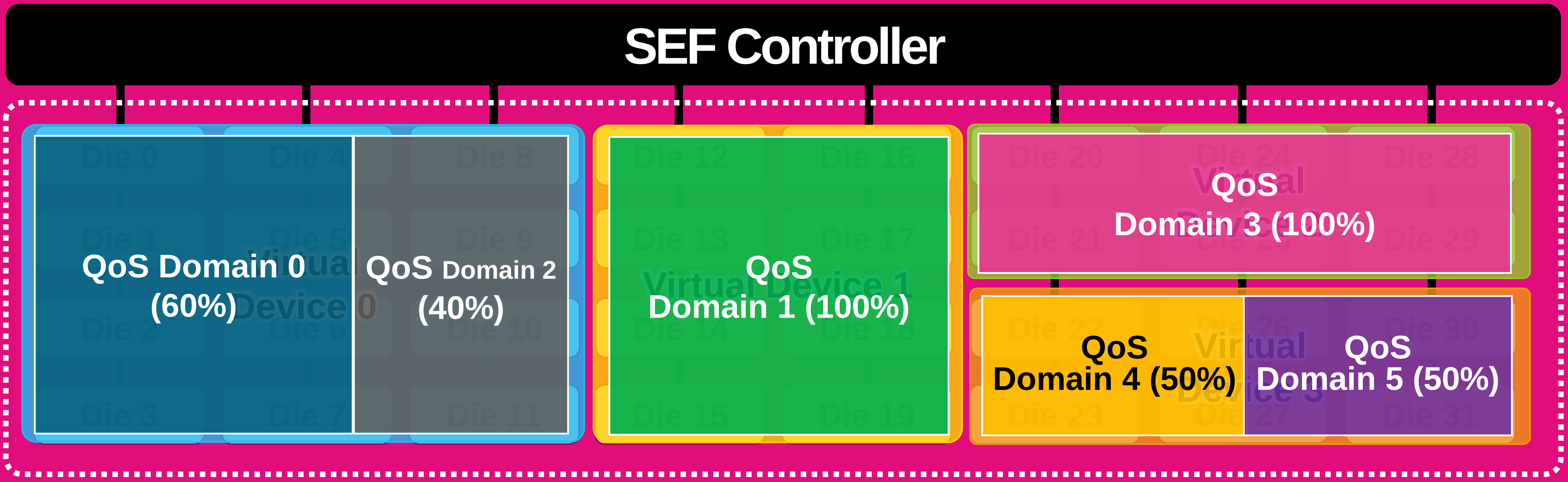
A Possible Software-Enabled Flash deployment



02

Software-Enabled Flash Concepts and Technologies

Placement Control and Isolation



CONCEPTS

Virtual Devices – a set of one or more flash die providing hardware isolation

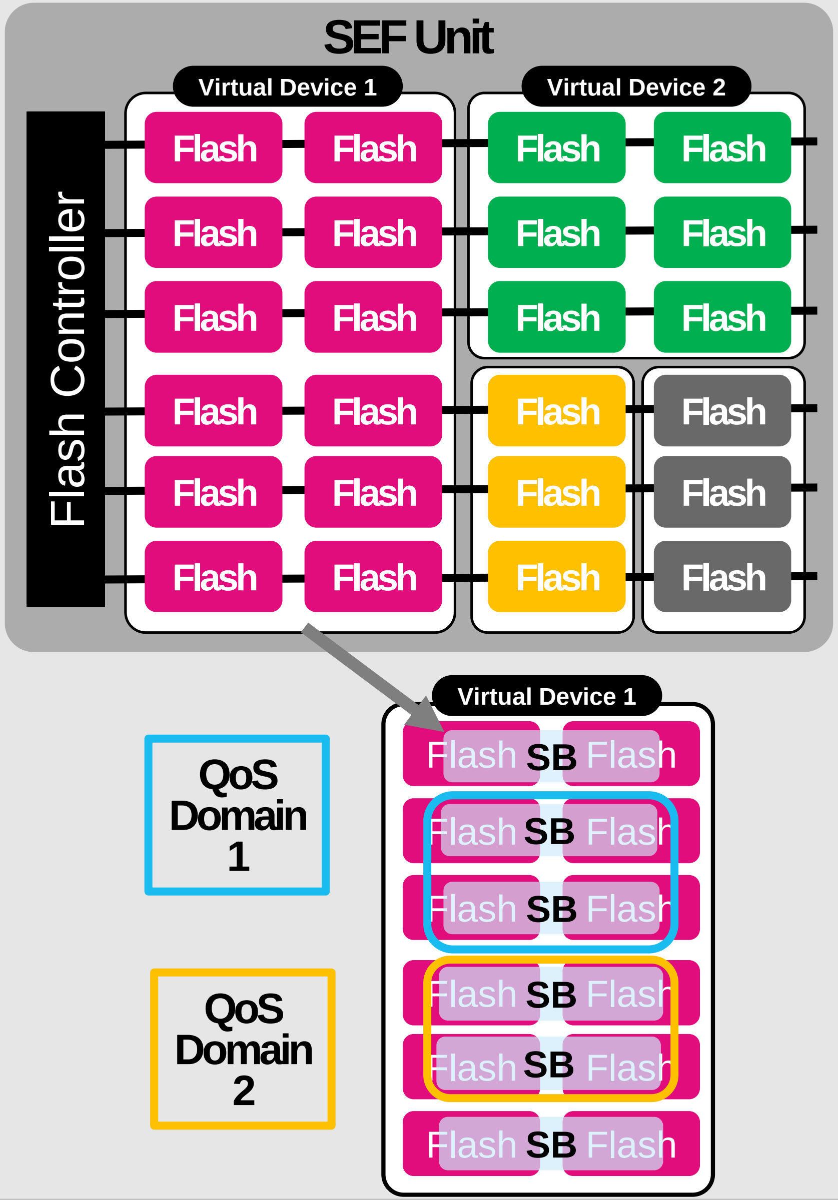
QoS Domains – a mechanism to impose capacity quotas and scheduling policy to provide software based isolation

Placement IDs – a mechanism to group data at the superblock level

KIOXIA

Data Placement Mechanism for Data Isolation

Isolation is done via the data placement mechanisms through definition of virtual devices and QoS domains.



Virtual Device: physical isolation at die level

- Capacity and I/O throughput can be defined
- Configures "Superblock (*)" which is the minimum data set size for internal data management.

  (*) Superblocks(SB) are striped across each die in the virtual device

QoS Domain: logical isolation at flash block level

- QoS Domain is a data container within the virtual device
- Superblocks are assigned to a QoS domain
- API manages life-cycle of superblocks

Isolation Capability

Die Level Isolation (Virtual Devices)

Most effective, least scalable.

- Full isolation is possible for small numbers (tens) of tenants
- Practical number of die is limited, so flash device cannot service large numbers of tenants with die level isolation

Block Level Isolation (QoS Domains)

Most scalable, not total isolation.

- To enable predictable performance and extend lifetime, block level isolation is very effective for large number (thousands) of tenants

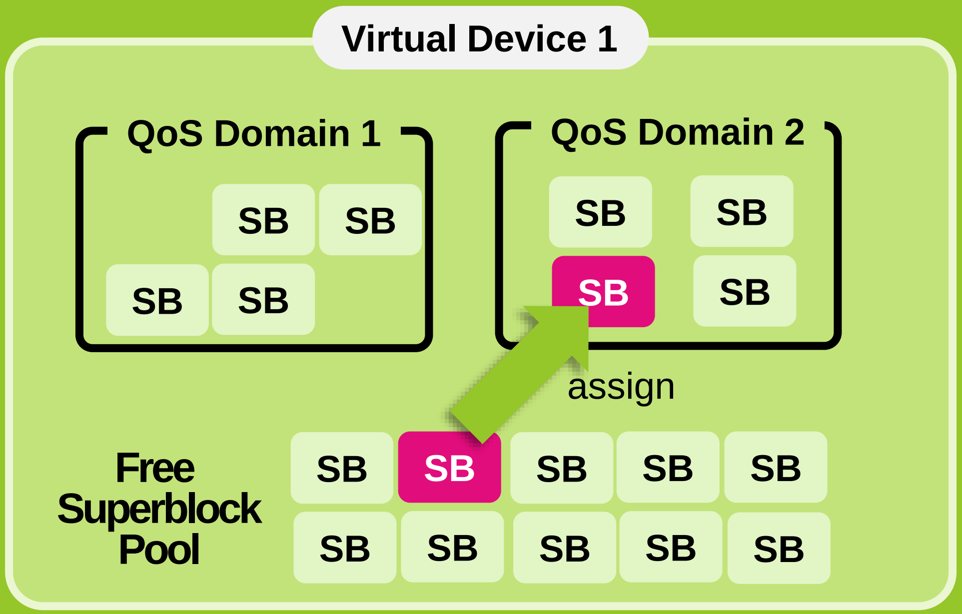
Nameless Write Mechanism (1) - "Automatically assigns a block to write"

Host wants control over data placement to minimize write amplification (WAF).

Host Dilemma

Optimal block selection is best left to the device to ensure media life and health.

- SEF Unit will automatically select most appropriate superblock and assign it to the QoS domain (applies to both manual and automatic allocation)
- Ownership of the superblock may change after it is released to the free pool

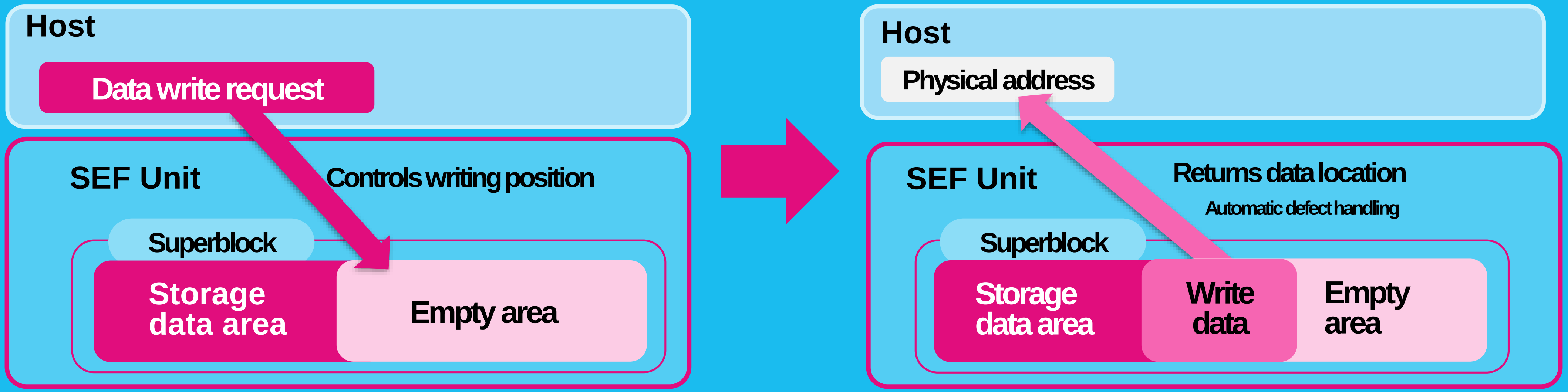


Nameless Write Mechanism (2) "Bounded Automatic Data Placement"

Write Operation:

QoSDomainID and **PlacementID** or **FlashAddress** bounds selection of superblock

- Host does not specify physical address for write
- Physical address returned upon write completion



Read Operation:

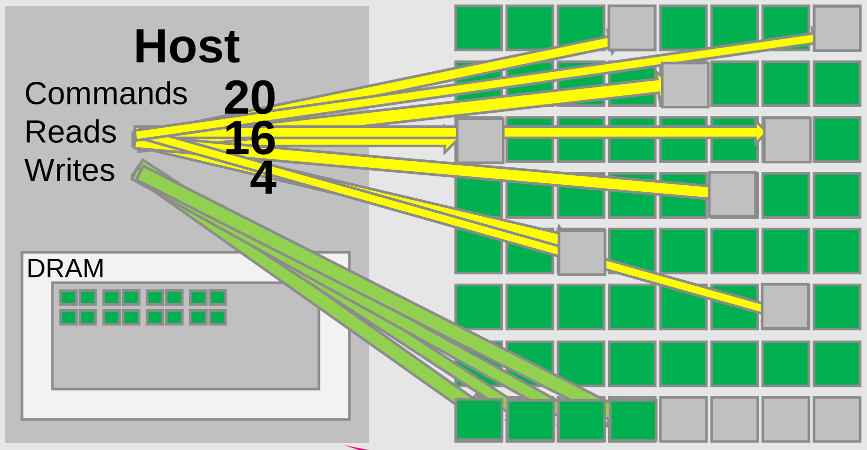
Direct physical address read

- Logical-physical address conversion process is not required by device
- Minimal read overhead and latency

Nameless Copy Mechanism – "Offloading Garbage Collection(GC)"

"Nameless Copy" is a copy offload function that implements internal data copy from die to die without the host moving data. Garbage collection becomes fast and efficient.

Manual Copy Operation

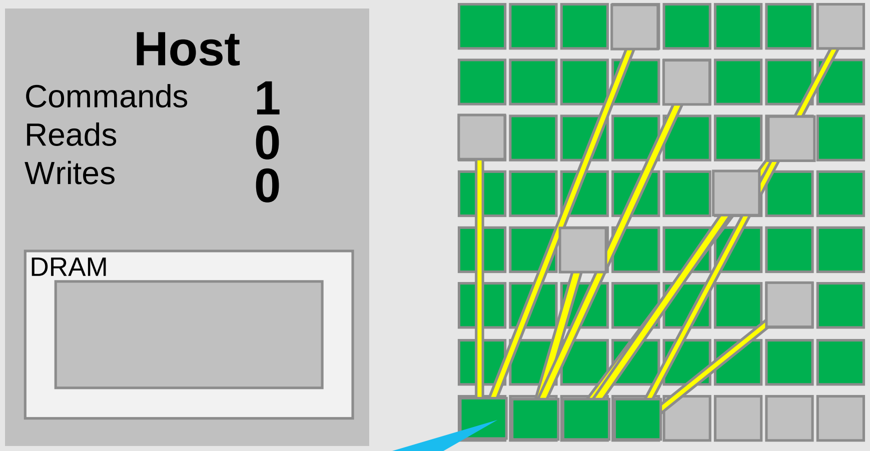


Fragmented Block

Valid Data Block

Free Block

Nameless Copy Operation



Garbage collection "read and write" data flows through the host

Host controls when to garbage collect, data is copied internally by device

Set-up

Copy Comparison

KIOXIA

Start Test

Manual Copy

CPU Utilization

User Workload

Garbage Collection

Nameless Copy

Manual Copy

User Workload	
User Write Data	0
Elapsed Time	0
CPU Utilization (Sec)	0
Garbage Collection	
Copied SuperBlocks	0
Released SuperBlocks	0
Copy Commands	0
GC Read Commands	0
GC Write Commands	0

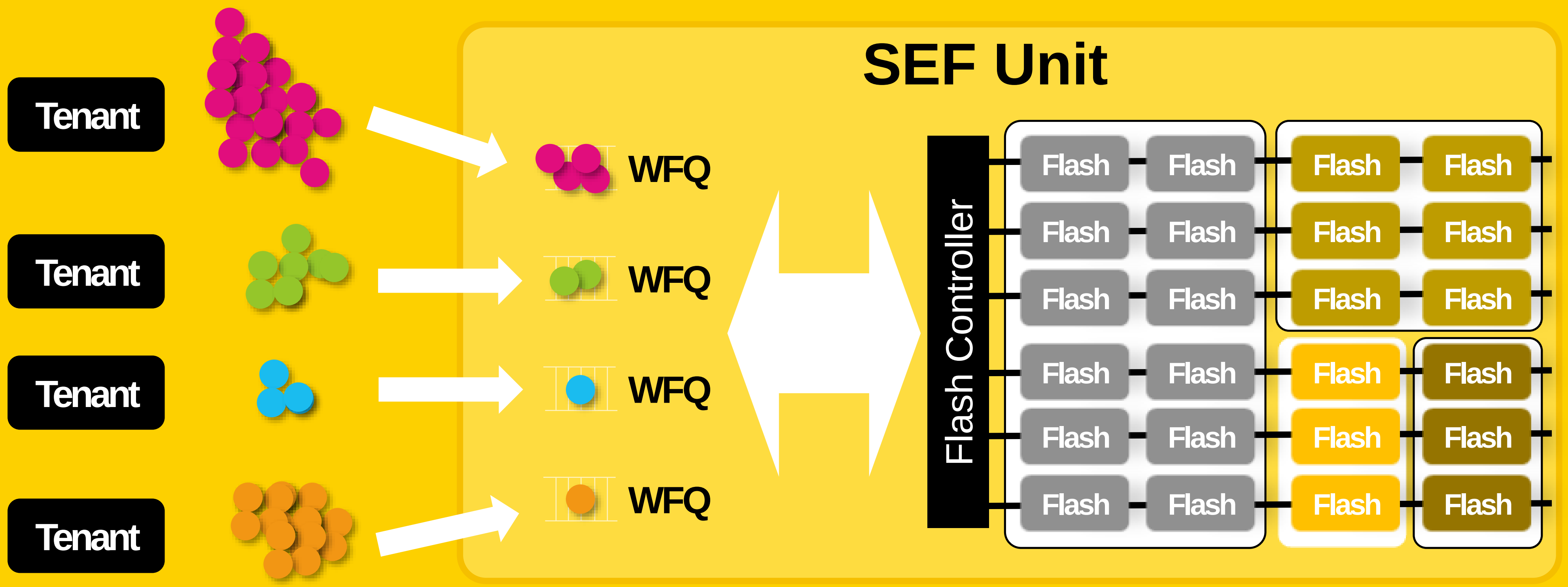
Nameless Copy

User Workload	
User Write Data	0
Elapsed Time	0
CPU Utilization (Sec)	0
Garbage Collection	
Copied SuperBlocks	0
Released SuperBlocks	0
Copy Commands	0
GC Read Commands	0
GC Write Commands	0

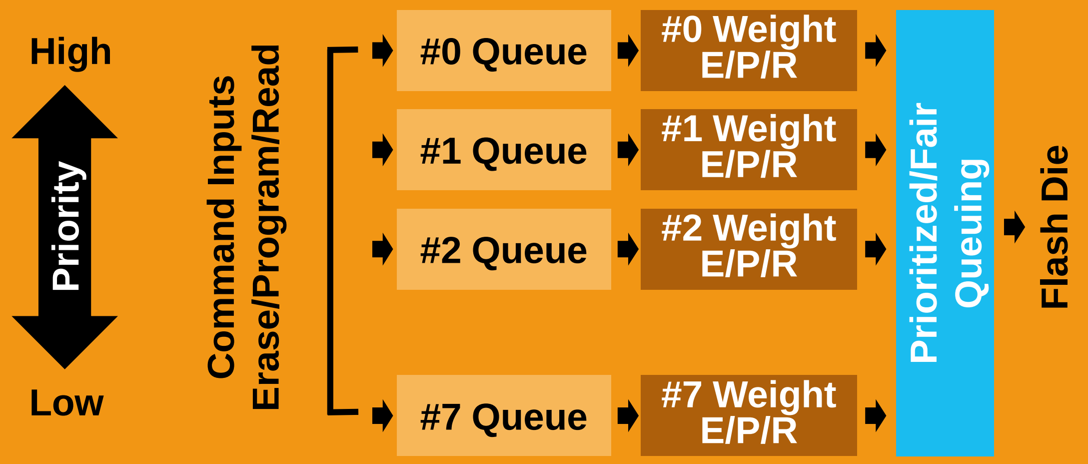
Weighted Fair Queuing (WFQ)

Control of processing fairness provides adjustable service levels for each tenant

- Host can control the priority ratio of write and read requests per QoS domain
- SEF unit implements a die-based IO fairness control



Controller Architecture: Die Scheduling and Queuing



Functionality

- When all weights are 0, it works as an 8-level priority scheduler
- When all weights are same value , it works as a round-robin scheduler
- When weights are unique, it works as a die time weighted scheduler

FEATURES

- Each virtual device has 8 FIFO command queues (up to 1:1 ratio with die!)
- Device scheduler handles suspend/resume for program and erase commands
- Host can specify a specific queue for each flash access command (read, program, erase) per QoS domain
- Every queue can specify die time weighting for read, program, and erase commands
- Host can override defaults queues and weights for individual commands

Set-up **Weighted Fair Queuing Demo**

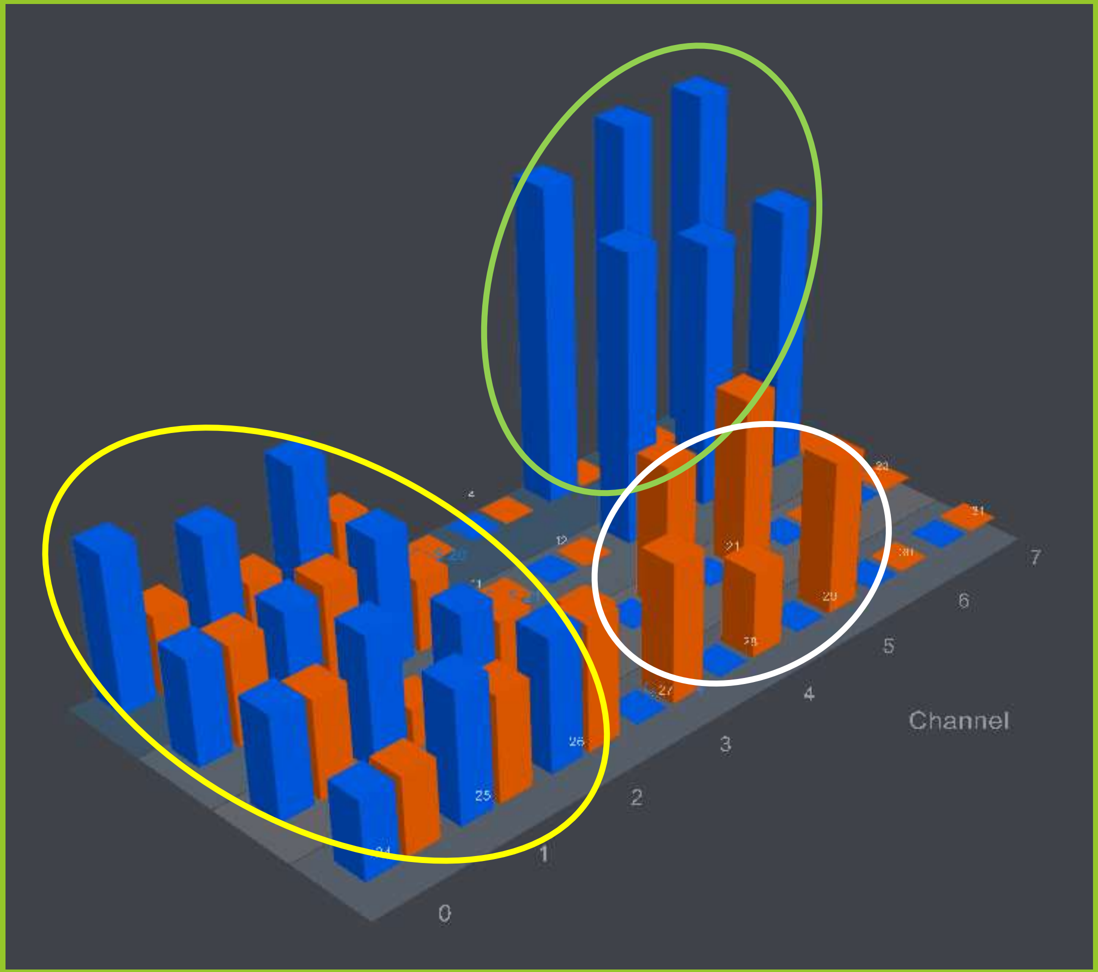
KIOXIA

Start Test

Read Weight

QoS Domain 1	QoS Domain 2
(198)	(202)

Software-Enabled Flash Adaptability… Demonstrated



- A single device/SKU that is customizable and reconfigurable
- ZNS, standard SSD, Software-Enabled Flash native RocksDB, and a custom hyperscale FTL all running simultaneously on same device
- As technologies, applications, and workloads change over time, Software-Enabled Flash can be adapted to changing needs

03

API Examples

Sef-cli

```
===========================     General  Usage     ===========================
./sef-cli <target-specific-action> <target> -[hv] [target-specific-parameters]
sef-cli supports many different engines and each engine support multiple actions.
In order to print which arguments are supported by each target use the verbose flag.

===========================  General  Arguments  ===========================
The following arguments are general and are applied to all SEF Cli Targets
   -h, --help               Print help and exit
   -,  --version            Print version number and git commit hash and exit
   -V, --verbose            Explain what is being done or show additional information regarding the action being taken

===========================        Target        ===========================
The following targets are supported by your sef-cli software:
     virtual-device         Perform actions towards Virtual Devices
     superblock             Perform actions towards superblocks
     shell                  A Python Shell that is extended with SEF functionalities in order to interact with the device
     sef-unit               Perform actions towards mounted SEF units
     sdk                    Perform actions towards SEF SDK (block layer)
     root-pointer           Perform actions towards QoS Domain's root pointers
     qos-domain             Perform actions towards QoS Domains
     adu                    Perform direct IO towards the SEF
```

Practical Example: Creating a Virtual Device

```
#!/bin/bash
# Practical Example: Mapping SEF unit 0 to a Single Virtual Device
# The following script assumes an initially un-configured SEF unit
#
# Note that this script should be run with sudo permissions

/usr/local/bin/sef-cli create virtual \
  -s0 \
  --start-channel=0 \
  --num-channel=4 \
  --start-bank=0 \
  --num-bank=4 \
  --virtual-device-id=0 \
  --max-qos-domains=1

# To see all the virtual devices on unit 0:
/usr/local/bin/sef-cli list virtual -s
```

KIOXIA

Practical Example: Creating a QoS Domain

```
#!/bin/bash
# Practical Example: Create a QoS Domain on Virtual Device 0 on SEF unit 0
#
# Note that this script should be run with sudo permissions

/usr/local/bin/sef-cli create qos \
  -s0 \
  --virtual-device-id=0 \
  --qos-domain-id=2 \
  --flash-capacity=3000000 \
  --adu-size=4096 \
  --num-root-pointers=4 \
  --num-placement-id=4

# To see all QoS Domains on SEF Unit 0:
/usr/local/bin/sef-cli list qos -s0
```

KIOXIA

API Example: Nameless Write

```
// Get the handle for the SEF Unit
sefHandle = SEFGetHandle(sefUnitIndex);

// Open QoS domain
status = SEFOpenQoSDomain(sefHandle, qosDomainId, notifyFunc, context, key, &qosHandle);

// Write data to SEF
status = SEFWriteWitoutPhysicalAddress1(qosHandle, SEFAutoAllocate, placementId, userAddress,
numADU, &iov, 1, &permanentAddress, &distanceToEndOfSuperblock, &overrides);
```

API Example: Asynchronous Event Handler

```
// Open QoS domain
status = SEFOpenQoSDomain(sefHandle, qosDomainId, HandleSEFNotification, context, key, &qosHandle);

void HandleSEFNotification(void *context, struct SEFQoSNotification event) {
    FTLContext *ctxt = (FTLContext *) context;
    switch (event.type) {
        case kAddressUpdate:
            SEFFTLUpdate(ctxt, SEFGetUserAddressLba(event.changedUserAddress),
                         event.oldFlashAddress, event.newFlashAddress);
            break;
     case kSuperblockStateChanged:
            SSBStateEvent(ctxt, event.changedFlashAddress, kSSBEventClosed);
            break;
     ...
    }
}
```

KIOXIA

Lockless Lookup Tables – Address Update

```
// This is the non-authoritative way to update the lut[].  First writer wins
// implies the data has been moved and not updated

int SEFFTLUpdate(FTLContext *ctxt, int64_t userLBA, struct SEFFlashAddress oldFlashAddress,
                 struct SEFFlashAddress newFlashAddress) {
    bool lutUpdated;
    uint64_t expectedBits = oldFlashAddress.bits;
    // Mark new flash address as used in superblock map
    // Update lut[] with the new flash address as long as the old was there
    // Mark old flash address as free in superblock map, if lut[] updated
    // Mark new flash address as free in superblock map, if lut[] failed update
    SSBSetAduValid(ctxt,newFlashAddress); // In case exchange works, must be set
    lutUpdated = atomic_compare_exchange_strong(&ctxt->lut[userLBA], &expectedBits, newFlashAddress.bits);
    if (lutUpdated)
    {
        assert(oldFlashAddress.bits);
        SSBClearAduValid(ctxt,oldFlashAddress);
        ctxt->lutDirty = true;
    }    else    {
        SSBClearAduValid(ctxt,newFlashAddress);
        // Undo it if it didn't
#if CC_DEBUG_COUNTERS
        atomic_fetch_add(&ctxt->ccCounter.lateUpdate, 1);
#endif
    }
    return 0;
}
```

KIOXIA

API Example: Direct Access Read

```
// Get the handle for the SEF Unit
sefHandle = SEFGetHandle(sefUnitIndex);

// Open QoS domain
status = SEFOpenQoSDomain(sefHandle, qosDomainId, notifyFunc, context, key, &qosHandle);

// Read data from the flash
status = SEFReadWithPhysicalAddress1(qosHandle, flashAddress, numADU, &iov, iovCount, iovOffset, userAddress, &overrides);

// NOTE: The error recovery mode for a QoS Domain is set at time of
// creation. This determines if the SEF unit will perform automatic
// error recovery on the QoS Domain. In manual mode SEFSetReadDeadline
// determines how quickly the SEF unit must respond when attempting
// recovery
```

Data Path Commands Have Asynchronous versions

```
struct SEFReadWithPhysicalAddressIOCB {
    struct SEFStatus status;                   /**< Library sets error field to a non-zero value to indicate ... */
    int16_t opcode;                            /**< Should never be accessed - for internal use by library */
    int16_t done;                              /**< Flag for polled I/O - library sets this field */
    ALIGN_FOR_POINTER(4);
    void *param1;                              /**< Ignored by the library; the caller can store context ... */
    void (*complete_func)(struct SEFReadWithPhysicalAddressIOCB *);
    const struct SEFReadOverrides *overrides;  /**< Override parameters for scheduling purposes, may be NULL */
    struct SEFFlashAddress flashAddress;       /**< Physical address for the read command */
    struct SEFUserAddress;                     /**< Contains LBA information */
    const struct iovec *iov;                   /**< A pointer to the scatter gather list */
    uint32_t iovOffset;                        /**< Starting byte offset into iov array */
    uint32_t numADU;                           /**< Number of ADUs to be read, maximum is superblockCapacity */
    uint16_t iovcnt;                           /**< Number of elements in the scatter gather list */
};

void SEFReadWithPhysicalAddress1Async(void *SEFHandle, struct SEFReadWithPhysicalAddressIOCB *iocb);
```

04

Software-Enabled Flash Summary & Where to Learn More

Summary

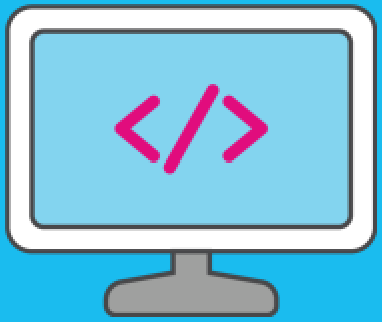
Software-Enabled Flash™ technology fundamentally redefines the relationship between host and solid-state storage.

- Purpose-built hardware for managing flash media under host control
- Open-source, flash-native API
- Industry standards and protocols
- May be used as a building block for multiple drive protocols *(block, ZNS, custom)*
- Combines full host control with ease of use

For More Information on Software-Enabled Flash™ Technology

1 

2 

3 

microsite

www.softwareenabledflash.com

white paper

Available on the microsite

API

http://github.com/kioxiaamerica

KIOXIA