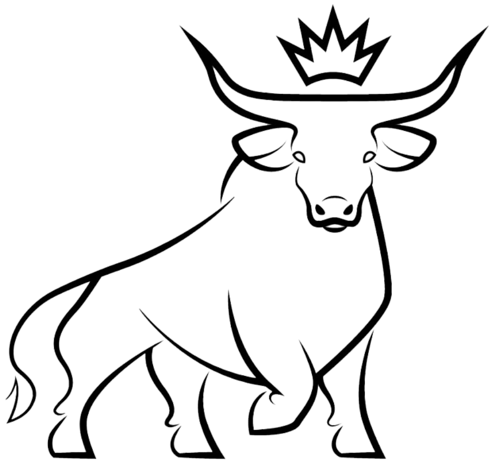
IT University of Copenhagen



OX Controller 1.0

Revision 1.0

October, 2016

Table of Contents

Introduction **2**

Open-Channel SSDs **2**

Figure 1. Solid-State Drive Geometry 2

LightNVM **2**

Figure 2. LightNVM Ecosystem (Matias Bjørling, 2016) 3

OX Controller **3**

Figure 3. OX Controller Layers 4

OX Layers in the C code 5

NVMe to MMGR I/O Command Translation 5

Figure 4. NVMe to MMGR I/O Translation 7

Table 1. nvm_io_cmd Fields 8

Table 2. nvm_mmgr_io_cmd Fields 9

I/O Command Flow 9

Figure 5. OX I/O Command Flow 9

DFC Implementation 10

Table 3. Latest DFC firmware tested with OX 11

DFC OX Installation 11

OX Command Line 12

Start and Debug 12

Admin 13

Tests 13

OX Layer Registration **14**

OX Logs **14**

1. Introduction

OX is a controller solution for programmable devices like the DFC. OX exposes the device as a LightNVM compatible Open-Channel SSD. OX has been developed to work as a hybrid controller, potentially supporting different FTL responsibilities (e.g. write buffering or ECC) or a full-fledged FTL. FTLs are registered within the OX core, enabling applications to select channels to be managed by a specific FTL. Within a device we may have different channels managed by different FTLs.

2. Open-Channel SSDs

SSDs are composed of tens of flash chips wired in parallel to a controller through channels. Flash memory is organized as shown in Figure 1. An embedded FTL abstracts the non-volatile memory and exposes a flat address space to the host. This arbitration shields the host from dealing with physical media.

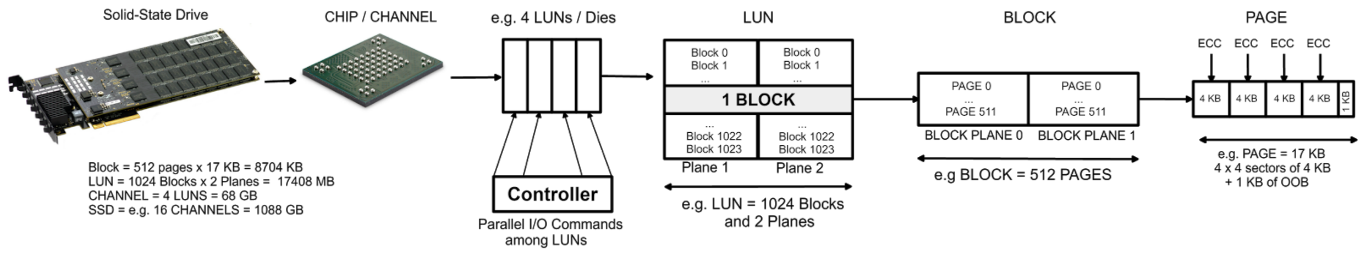


Figure 1. Solid-State Drive Geometry

A new generation of SSDs known as Open-Channel are emerging in the storage market. These new devices expose their media geometry to the host and allow applications to decide how to place and manage data among their chips, dies, blocks, and pages. FTL responsibilities such as wear leveling (WL), logical to physical mapping (L2P), garbage collection (GC), bad block management (BB), error correction codes (ECC), among others, may be implemented on the host side or be kept in the device controller. These decisions depend on the application needs and will result in different generations of Open-Channel SSDs. **Open-Channel SSDs implements the PPA I/O interface**. Please refer to <http://openchannelssd.readthedocs.io/en/latest/> for further details.

3. LightNVM

LightNVM is the open-channel SSD subsystem in the Linux kernel. LightNVM identifies Open-Channel SSDs through the lightnvm-enable NVMe driver. Responsibilities like WL, L2P, GC, BB and ECC may be implemented within the stack. Figure 2 depicts the LightNVM ecosystem.

The device driver is responsible to identify and register the OpenChannel SSD within LightNVM core. The targets are responsible to implement responsibilities such as WL, L2P and GC. Finally, applications can mount a target (expose a block device) on top of an Open-Channel SSD.

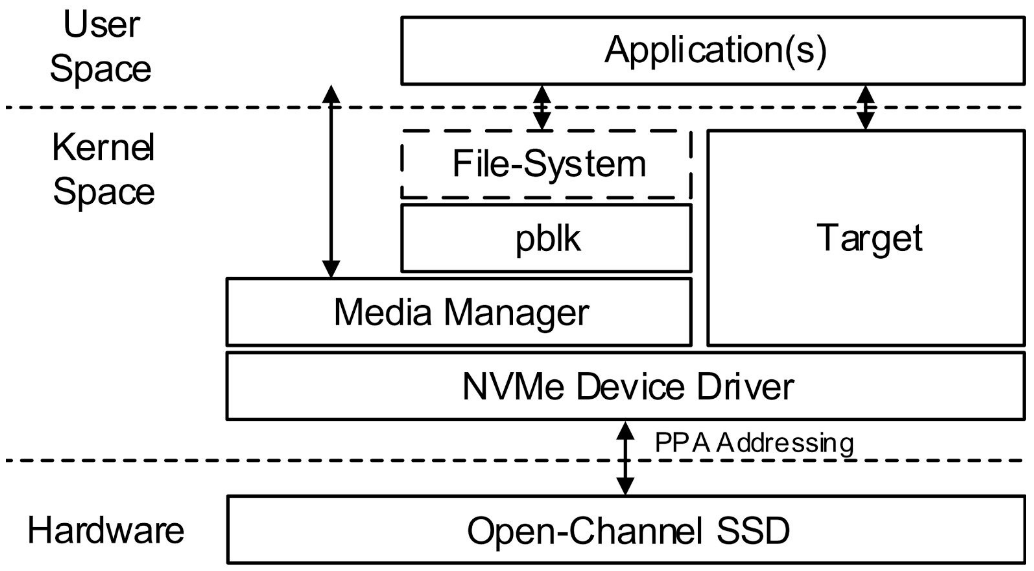


Figure 2. LightNVM Ecosystem (Matias Bjørling, 2016)

4. OX Controller

OX is a LightNVM-enabled SSD controller that exposes the device, either as Open-Channel or as a standard block device. OX implements the NVMe interface and comprises several layers running their own responsibilities. Each layer registers an entry point for global functions, avoiding redundant calls. Figure 3 depicts the OX layers.

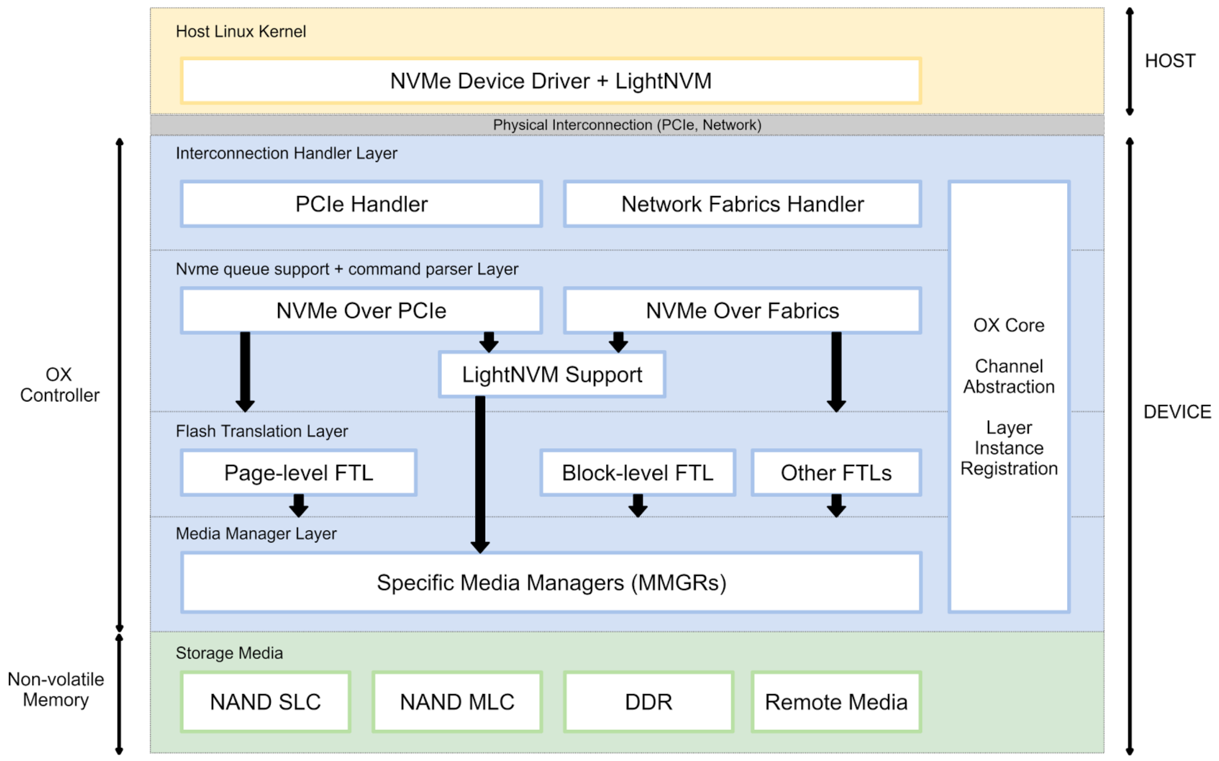


Figure 3. OX Controller Layers

Media managers (MMGR): Identifies the non-volatile memory, and registers a channel abstraction with its geometry to be available to the stack. Media managers implement read/write/erase commands to specific pages and blocks. OX may have several media managers exposing channels from different sources.

Flash Translation Layers (FTL): Manages MMGR channels, accepts IO commands (a command may have several pages) and sends page-level commands to the media managers. It implements responsibilities such as WL, L2P, GC, BB and ECC. Several FTLs may run within OX and be responsible to manage different MMGR channels.

Interconnect handlers (ICH): Communication between controller and host. This layer implements responsibilities such as NVMe register mapping and MSI interruptions. It gathers data from the host and sends it to the command parser layer.

NVMe queue support: Read/write NVMe registers mapped by the ICH, manage admin/ IO NVMe queue(s) and perform the DMA data transfer.

Command parsers (CP): Parses commands coming from the ICH and call the right FTL instance to handle it.

4.1. OX Layers in the C code

OX is organized in layers and represented in the code as follow (all paths start in the root directory):

Core Files:
core.c, cmd_args.c, include/ssd.h

NVMe, LightNVM support, command parsers:
nvme.c, nvme_cmd.c, lightnvm.c, include/nvme.h, include/lightnvm.h

DFC PCIe Interconnect Handler:
pcie_dfc/<pci_dfc.c, pcie_dfc.h>

DFC NAND Media Manager:
mmgr/dfcnand/<dfc_nand.c, dfc_nand.h, nand_dma.a, nand_dma.c, nand_dma.h>

For Open-Channel SSD, OX implements a raw FTL (ftl_lnvm) containing only the NVMe to MMGR I/O Command Translation (other FTL responsibilities may be implemented in the device. For now, the full-fledged FTL must be implemented in the host. Refer to 'pblk' in LightNVM for a full FTL implementation in the host side). See section 6.2 for further details.

LightNVM FTL (ftl_lnvm):
ftl/lnvm/<ftl_lnvm.c, ftl_lnvm.h>

Tests:
test/<test_core.c, test_admin.c, test_lightnvm.c, test_mmgr.c, include/tests.h>

4.2. NVMe to MMGR I/O Command Translation

OX Controller parses NVMe commands for Block or for PPA I/O interfaces. Both types are represented in the same internal data structure (struct nvm_io_cmd) and must be translated to the media specific command (struct nvm_mmgr_io_cmd) for each physical page. Please refer to Code 1 and Code 2 for the respective data structures.

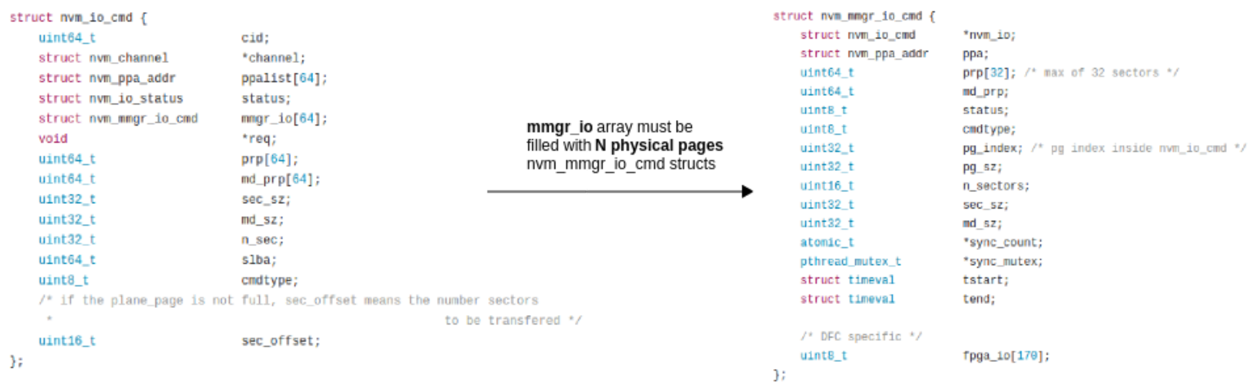
```
struct nvm_io_cmd {
    uint64_t                    cid;
    struct nvm_channel          *channel;
    struct nvm_ppa_addr         ppalist[64];
    struct nvm_io_status        status;
    struct nvm_mmgr_io_cmd      mmgr_io[64];
    void                        *req;
    uint64_t                    prp[64];
    uint64_t                    md_prp[64];
    uint32_t                    sec_sz;
    uint32_t                    md_sz;
    uint32_t                    n_sec;
    uint64_t                    slba;
    uint8_t                     cmdtype;
    /* if the plane_page is not full, sec_offset means the number sectors
     *                                                to be transfered */
    uint16_t                    sec_offset;
};
```

Code 1. OX I/O command data structure (struct nvm_io_cmd)

```
struct nvm_mmgr_io_cmd {
    struct nvm_io_cmd       *nvm_io;
    struct nvm_ppa_addr     ppa;
    uint64_t                prp[32]; /* max of 32 sectors */
    uint64_t                md_prp;
    uint8_t                 status;
    uint8_t                 cmdtype;
    uint32_t                pg_index; /* pg index inside nvm_io_cmd */
    uint32_t                pg_sz;
    uint16_t                n_sectors;
    uint32_t                sec_sz;
    uint32_t                md_sz;
    atomic_t                *sync_count;
    pthread_mutex_t         *sync_mutex;
    struct timeval          tstart;
    struct timeval          tend;

    /* DFC specific */
    uint8_t                 fpga_io[170];
};
```

Code 2. OX Media specific I/O command data structure (struct nvm_mmgr_io_cmd)



```
struct nvm_io_cmd {
    uint64_t                    cid;
    struct nvm_channel          *channel;
    struct nvm_ppa_addr         ppalist[64];
    struct nvm_io_status        status;
    struct nvm_mmgr_io_cmd      mmgr_io[64];
    void                        *req;
    uint64_t                    prp[64];
    uint64_t                    md_prp[64];
    uint32_t                    sec_sz;
    uint32_t                    md_sz;
    uint32_t                    n_sec;
    uint64_t                    slba;
    uint8_t                     cmdtype;
    /* if the plane_page is not full, sec_offset means the number sectors
     *                                                   to be transfered */
    uint16_t                    sec_offset;
};
```

mmgr_io array must be filled with **N physical pages** nvm_mmgr_io_cmd structs

```
struct nvm_mmgr_io_cmd {
    struct nvm_io_cmd       *nvm_io;
    struct nvm_ppa_addr     ppa;
    uint64_t                prp[32]; /* max of 32 sectors */
    uint64_t                md_prp;
    uint8_t                 status;
    uint8_t                 cmdtype;
    uint32_t                pg_index; /* pg index inside nvm_io_cmd */
    uint32_t                pg_sz;
    uint16_t                n_sectors;
    uint32_t                sec_sz;
    uint32_t                md_sz;
    atomic_t                *sync_count;
    pthread_mutex_t         *sync_mutex;
    struct timeval          tstart;
    struct timeval          tend;

    /* DFC specific */
    uint8_t                 fpga_io[170];
};
```

Figure 4. NVMe to MMGR I/O Translation

NVMe commands when translated to media specific commands may contain several physical pages (each page is represented by a nvm_mmgr_io_cmd instance. Refer to Figure 4). These pages must be mapped according to the incoming I/O command. The translation must occur within the FTL. We call this translation as NVMe to MMGR I/O translation.

Block I/O and PPA I/O differ each other and must be translated according the Table 2.

Table 1 shows the nvm_io_cmd fields:

Type	**Field**	**Description**
uint64_t	cid	NVMe command id
struct nvm_channel *	channel	Related nvm_channel pointer
struct nvm_ppa_addr [ ]	ppalist	List of physical addresses (per sector). Not used for Block I/O commands.
struct nvm_io_status	status	Command status
struct nvm_mmgr_io_cmd [ ]	mmgr_io	MMGR commands per page, up to 64
void *	req	Pointer to NVMe queue specific data structure
uint64_t [ ]	prp	Array of PRP addresses (per sector)
uint64_t [ ]	md_prp	Array of metadata addresses (per sector)
uint32_t	sec_sz	Sector size
uint32_t	md_sz	Metadata size (total for all sectors)
uint32_t	n_sec	PPA IO: number of PPA sectors Block IO: number of logical blocks

uint64_t	slba	Start logical block address. Not used for PPA I/O commands.
uint8_t	cmdtype	Command type: READ, WRITE, ERASE (not used for Block I/O commands).
uint16_t	sec_offset	PPA IO: if > 0, last LightNVM page has sec_offset sectors Block IO: if > 0, last logical block is smaller

Table 1. nvm_io_cmd Fields

Table 2 shows the nvm_mmgr_io_cmd fields:

Type	**Field**	**Description**
	Fields to be filled by FTL	
struct nvm_ppa_addr	ppa	Physical address (page granularity)
uint64_t [ ]	prp	Array of PRP addresses (per physical sector)
uint64_t	md_prp	Page metadata pointer
uint32_t	pg_index	Index of nvm_io_cmd->mmgr_io[ ]
uint32_t	pg_sz	physical page size (must follow global pg_size)
uint16_t	n_sectors	number of physical sectors (must be equal of global sectors per page)
uint32_t	sec_sz	physical sector size (must be equal global sector size)
uint32_t	md_sz	Page Metadata size
atomic_t *, pthread_mutex_t	sync_count, sync_mutex	Set these pointers to use synchronous IO commands for static wear leveling. Keep it NULL for NVMe I/O commands.
	Fields automatically set by OX	
struct nvm_io_cmd *	nvm_io_cmd	Pointer to the parent I/O command
uint8_t	status	Command status, can be checked in the completion
uint8_t	cmdtype	WRITE / READ / ERASE

struct timeval struct timeval	tstart, tend	command times in microseconds. Can be checked after completion.

Table 2. nvm_mmgr_io_cmd Fields

4.3. I/O Command Flow

OX has been designed to execute IO commands in parallel. Figure 5 shows the IO command flow within the controller. Colors represent thread responsibilities.

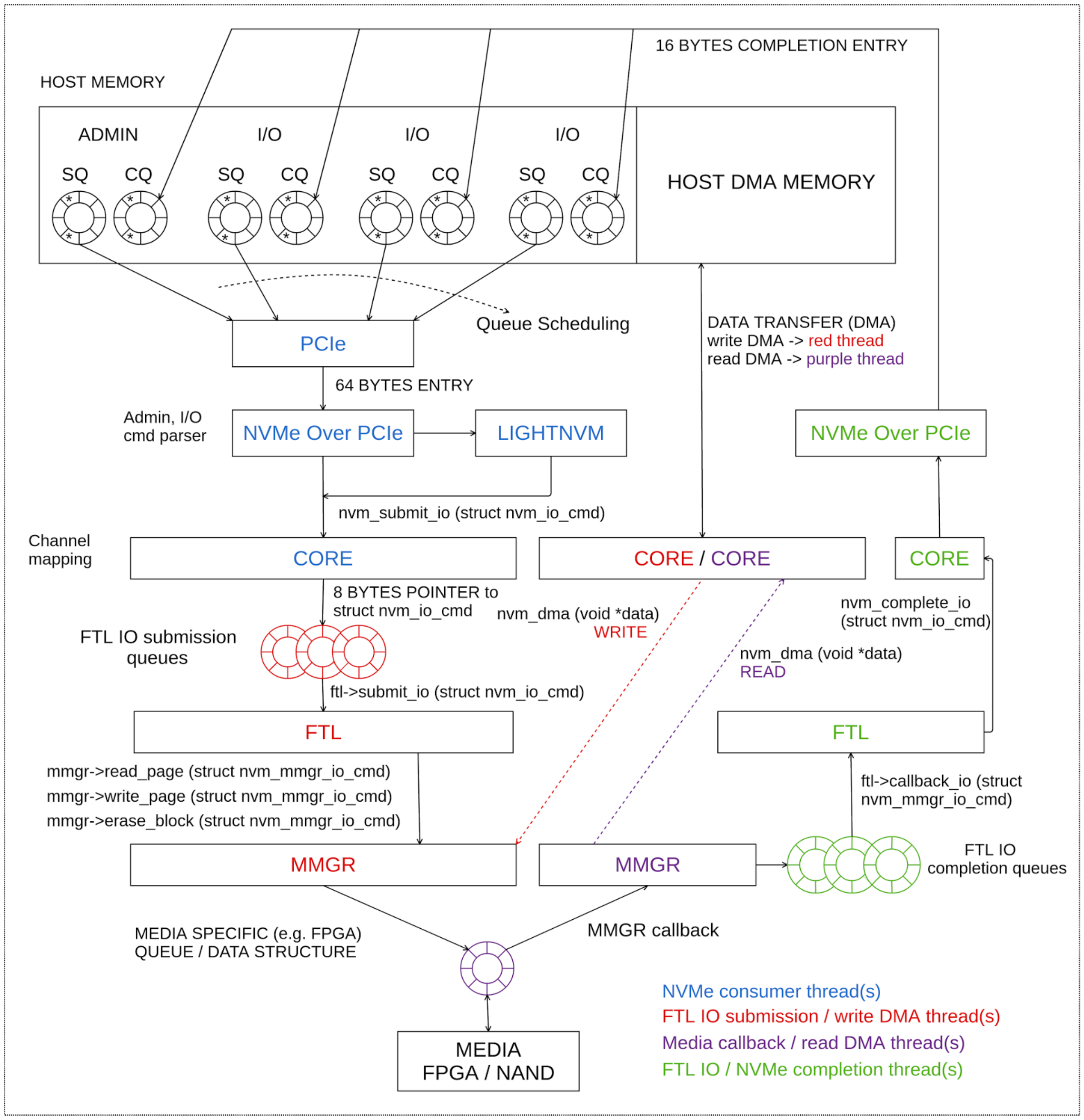


Figure 5. OX I/O Command Flow

Blue threads are responsible to consume the NVMe queue located in the host. These threads cross the Interconnection Layer (PCIe), NVMe and command parser Layer. After dequeuing the NVMe command from the host memory queue, the blue thread uses the library located within the OX core to map the correct channel and FTL. Then, it posts the I/O command to the FTL queue (red).

Red threads are responsible to consume the FTL queues located in the device memory space. All FTL processing occurs in these threads. **Each FTL queue has its own thread consumer. Since the number of queues per FTL is configurable, the FTL is natively multi-threaded.** These threads cross the FTL Layer, Media Manager Layer and Write (read from host) DMA process. Within the Media Manager, the red threads post the IO command to the media specific queue (purple). In case of the DFC, this queue is located in the FPGA interface library.

Purple threads are responsible for the media I/O completion. These threads checks for completed I/O commands in the purple queues and post the completion to the related FTL completion queue (green). These threads cross the Media Manager Layer and Read (write to host) DMA process.

Green threads are responsible for the NVMe I/O completion. These threads checks for completed I/O commands in the green queues and are responsible for FTL command completion. The NVMe completion process takes place in this stage, when the threads post a completion NVMe queue entry in the host memory. These threads cross the FTL layer and NVMe Layer.

4.4. DFC Implementation

We have implemented a PCIe interconnection handler, a media manager to expose 8 channels, NVMe and LightNVM support. The DFC-based OX controller is a layered architecture that enables future extensions such as support for other devices and Fabrics/RDMA interconnection.

OX does **not have a FTL for the standard Block I/O interface**, for now OX only works as Open-Channel SSD Controller. Future releases may contain full FTL implementations, fabrics and RDMA interconnection.

Table 3 shows the latest DFC stable firmware tested with OX. **Please be sure of your FPGA version.** OX is current implemented for FPGA 03.01.00.

Image	Version
PBL	03.00.00
DPC	03.00.00

MC	03.00.00
Kernel	05.00.00
U-boot	03.00.00
rootfs	03:00:00
CPLD_NIC	A12
CPLD_SSD	A02
FPGA	03.01.00

Table 3. Latest DFC firmware tested with OX

4.5. DFC OX Installation

For now, OX only works with 2 NAND modules installed in the slots M1 and M3. A DDR Media Manager may be included in the next releases. For proper DFC setup, please refer to LS2_iSSD_USER_MANUAL_A00-06.pdf available in the EMC iSSD Community repository under the folder iSSD_release_V02.02.01/Documents.

OX must be cross-compiled using the SDK available in the EMC iSSD Community repository under the folder iSSD_release_V03.00.00. However, the OX code contains the binaries already compiled in the follow paths:

```
<OX_root>/bin/dfc_fpga_3_01_00/ox-ctrl
<OX_root>/bin/dfc_fpga_3_01_00/ox-ctrl-test
<OX_root>/bin/dfc_fpga_3_01_00/rootfs_03.00.00_ox.sqsh
```

The Makefile creates 2 binaries (ox-ctrl and ox-ctrl-test). ox-ctrl is a essential-only binary for a production environment. ox-ctrl-test is a complete binary including all tests and administration mode. The rootfs is also included. WE RECOMMEND THE USE OF ROOTFS. After upgrading the rootfs firmware using the upgrade.sh script included in the last rootfs, you can run OX only typing 'ox-ctrl' or 'ox-ctrl-test’ in the DFC prompt.

IMPORTANT: If you decide to copy the OX binaries to the DFC and run it, please make sure the last issd-nvme controller is not started (the normal boot with the last rootfs starts the issd-nvme).

4.6. OX Command Line

OX accepts arguments in the command line. Please run the OX binary in the DFC prompt or make a script to start it automatically after the kernel boots.

You can use the follow command to see all available arguments:

$ ox-ctrl --help

```
Available commands:
  start             Start controller with standard configuration
  debug             Start controller and print Admin/IO commands
  test              Start controller, run tests and close
  admin             Execute specific tasks within the controller
```

4.6.1. Start and Debug

Use ‘start’ to run the controller with standard settings:

$ ox-ctrl start

Use ‘debug’ to run the controller in debug mode (verbose):

$ ox-ctrl debug

When an I/O command is posted to the NVMe submission queue by the host, OX will print the information related to the I/O command flow in Figure 5. See a printed I/O debug example below:

```
[659] IO CMD 0x92, nsid: 1, cid: 0
 Number of sectors: 8
 DMA size: 32768 (data) + 0 (meta) = 32768 bytes
 [ppa(0): ch: 0, lun: 0, blk: 220, pl: 0, pg: 0, sec: 0]
 [ppa(1): ch: 0, lun: 0, blk: 220, pl: 0, pg: 0, sec: 1]
 [ppa(2): ch: 0, lun: 0, blk: 220, pl: 0, pg: 0, sec: 2]
 [ppa(3): ch: 0, lun: 0, blk: 220, pl: 0, pg: 0, sec: 3]
 [ppa(4): ch: 0, lun: 0, blk: 220, pl: 1, pg: 0, sec: 0]
 [ppa(5): ch: 0, lun: 0, blk: 220, pl: 1, pg: 0, sec: 1]
 [ppa(6): ch: 0, lun: 0, blk: 220, pl: 1, pg: 0, sec: 2]
 [ppa(7): ch: 0, lun: 0, blk: 220, pl: 1, pg: 0, sec: 3]
 [prp(0): 0x00000000961a3000
 [prp(1): 0x0000000096234000
 [prp(2): 0x0000000096235000
 [prp(3): 0x000000009ba23000
 [prp(4): 0x0000000096237000
```

```
[prp(5): 0x0000000096238000
[prp(6): 0x0000000096239000
[prp(7): 0x000000009623a000
[meta_prp(0): 0x0000000000000000
CMD cid: 0, type: 0x1 submitted to FTL.
 Channel: 0, FTL queue 2
[IO CALLBK. CMD 0x1. mmgr_ch: 0, lun: 0, blk: 220, pl: 0, pg: 0]
[IO CALLBK. CMD 0x1. mmgr_ch: 0, lun: 0, blk: 220, pl: 1, pg: 0]
[NVMe cmd 0x92. cid: 0 completed. Status: 0]
```

4.6.2. Admin

All commands related to configuration and storage administration are implemented under 'admin' argument. For now, only 'erase-blk' is implemented (LightNVM writes and reads the first page in the namespace, if the block is not erased properly, LightNVM will fail to initialize the device. If it happens, please erase the block 2 in the LUN 0 in channel 0. This process needs to be done only the first time you initialize the device with LightNVM). Use the follow commands to see all admin available arguments:

```
$ ox-ctrl-test admin --help
Use this command to run specific tasks within the controller.
 Examples:
  Show all available Admin tasks:
    ox-ctrl admin -l
  Run a specific admin task:
    ox-ctrl admin -t <task_name>
```

```
$ ox-ctrl-test admin -l
Available OX Admin Tasks:
  - 'erase-blk': erase specific blocks.
     eg. ox-ctrl admin -t erase-blk (it will ask the block address)
  - 'erase-lun': erase specific LUNs.
     eg. ox-ctrl admin -t erase-lun (not implemented)
  - 'erase-ch': erase specific channels.
     eg. ox-ctrl admin -t erase-ch (not implemented)
```

4.6.3. Tests

OX tests are implemented in the 'ox-ctrl-test' binary. It will run a sequence of I/O commands to test the Media Manager, LightNVM FTL and LightNVM command parser. To test the PCIe interconnect handler and NVMe queue support, please run tests from the host PC. Use the follow command to see all available test arguments:

$ ox-ctrl-test test --help

```
Examples:
  Show all available set of tests + subtests:
    ox-ctrl test -l
  Run all available tests:
    ox-ctrl test -a
  Run a specific test set:
    ox-ctrl test -s <set_name>
  Run a specific subtest:
    ox-ctrl test -s <set_name> -t <subtest_name>
```

5. OX Layer Registration

Besides the DFC, OX is extensible to other devices through the layer registration interface. The registration can also be used to register multiple Media Managers and FTLs. In order to use OX in other platforms, the follow layers are required:

- **Interconnect handler:** Here you map the NVMe registers according to your platform, or may implement a Fabrics interconnection. See pcie_dfc implementation for an example.
- **Media Manager:** Here you abstract the chips of your non-volatile memory as channels, to be used by all OX stack. See dfc_nand implementation for an example.
- **FTL:** You can use the same FTL at any platform, but you can develop a new one and register it within the OX core (each channel has an internal data structure written in the first page in the media manager reserved block. This structure contains the FTL_ID related to this channel. Set this value to your specific FTL). See ftl_lnvm implementation for an example.

The respective global functions related to layer registration is listed below:

```
int  nvm_register_mmgr(struct nvm_mmgr *);
int  nvm_register_pcie_handler(struct nvm_pcie *);
int  nvm_register_ftl (struct nvm_ftl *);
```

Please refer to <OX_root>/include/ssd.h for the data structures details.

6. OX Logs

OX creates a log file in the startup and writes to it during the runtime. Please access the file /var/log/nvme.log in the DFC file system to check the log. After the OX startup you should get a log as follow:

```
<date>: [nvm: OX Controller is starting...]
<date>:  [nvm: Media Manager registered: DFC_NAND]
<date>:   [nvm: FTL Queue (/nvm_mq0) started.]
```

```
<date>:   [nvm: FTL Queue (/nvm_mq1) started.]
<date>:   [nvm: FTL Queue (/nvm_mq2) started.]
<date>:   [nvm: FTL Queue (/nvm_mq3) started.]
<date>:   [nvm: FTL Queue (/nvm_mq4) started.]
<date>:   [nvm: FTL Queue (/nvm_mq5) started.]
<date>:   [nvm: FTL Queue (/nvm_mq6) started.]
<date>:   [nvm: FTL Queue (/nvm_mq7) started.]
<date>:  [nvm: FTL (FTL_LNVM)(1) registered.]
<date>:  [FTL_LNVM cap: Get Bad Block Table]
<date>:  [FTL_LNVM cap: Set Bad Block Table]
<date>:  [FTL_LNVM Bad block table type: Byte array. 1 byte per blk.]
<date>:   [lnvm: channel 0 started with 5 bad blocks.
<date>:   [lnvm: channel 1 started with 5 bad blocks.
<date>:   [lnvm: channel 2 started with 5 bad blocks.
<date>:   [lnvm: channel 3 started with 5 bad blocks.
<date>:   [lnvm: channel 4 started with 5 bad blocks.
<date>:   [lnvm: channel 5 started with 5 bad blocks.
<date>:   [lnvm: channel 6 started with 5 bad blocks.
<date>:   [lnvm: channel 7 started with 5 bad blocks.
<date>:  [pci: outbound addr: 0x1400000000  host size: 0x200000000]
<date>:  [pci: host io_mem_addr : 0xfffd77800000]
<date>:  [pci: host msix_mem_addr : 0xffff91a2f000]
<date>:  [pci: bar4:0xfffd777e0000 icr:0xfffd777e8004 table_sz:0xfffd777e8008
csr:0xfffd777e8000 table:0xffff7c212000
<date>:  [nvm: PCI Handler registered: PCI_LS2085]
<date>:  [pci: NVMe thread alive.]
<date>:   [lnvm: Channels: 1]
<date>:   [lnvm: LUNs per Channel: 32]
<date>:   [lnvm: Blocks per LUN: 1024]
<date>:   [lnvm: Pages per Block: 512]
<date>:   [lnvm: Planes: 2]
<date>:   [lnvm: Total Blocks: 32768]
<date>:   [lnvm: Reserved/Bad Blocks: 40]
<date>:   [lnvm: Total Pages: 33554432]
<date>:   [lnvm: Page size: 16384 bytes]
<date>:   [lnvm: Plane Page size: 32768 bytes]
<date>:   [lnvm: Total: 524288 MB]
<date>:   [lnvm: Total Available: 523648 MB]
<date>:  [nvm: LightNVM is registered]
<date>:  [nvm: NVME standard registered]
<date>: [nvm: OX Controller ready.]
<date>:  [nvm: Active pci handler: PCI_LS2085]
<date>:  [nvm: 1 media manager(s) up, total of 8 channels]
```

```
<date>:   [channel: 0, FTL id: 1 ns_id: 1, ns_part: 0, pg: 4194304, in_use: 60]
<date>:   [channel: 1, FTL id: 1 ns_id: 1, ns_part: 1, pg: 4194304, in_use: 60]
<date>:   [channel: 2, FTL id: 1 ns_id: 1, ns_part: 2, pg: 4194304, in_use: 60]
<date>:   [channel: 3, FTL id: 1 ns_id: 1, ns_part: 3, pg: 4194304, in_use: 60]
<date>:   [channel: 4, FTL id: 1 ns_id: 1, ns_part: 4, pg: 4194304, in_use: 60]
<date>:   [channel: 5, FTL id: 1 ns_id: 1, ns_part: 5, pg: 4194304, in_use: 60]
<date>:   [channel: 6, FTL id: 1 ns_id: 1, ns_part: 6, pg: 4194304, in_use: 60]
<date>:   [channel: 7, FTL id: 1 ns_id: 1, ns_part: 7, pg: 4194304, in_use: 60]
<date>:  [nvm: namespace size: 549755813888 bytes]
<date>:  [nvm: total pages: 33554432]
```