The Jasmine OpenSSD Platform

Version 1.4

Technical Reference Manual

Revision history

Date	Author	Description	Rev.
2011-04-27	Sangphil Lim (Sungkyunkwan university)	Initial release	1.0
2011-04-28	Sangphil Lim (Sungkyunkwan university)	fix typo & reflect review comments	1.1
2011-05-20	Sangphil Lim (Sungkyunkwan university)	Section 2.1, 2.3, 2.5 update	1.2
2011-06-01	Sangphil Lim (Sungkyunkwan university)	fix errata (Section 4.4 & 4.5)	1.3
2012-01-11	Sangphil Lim (Sungkyunkwan university)	Fix errata & Update some additional information (Section 2.5, 3.3.2)	1.4
2016-08-22	Preethika Kasu, Donghyun Kang, Heerak Lim (Ajou university)	Translated to English	1.4

Contents

The Jasmine OpenSSD Platform: Technical Reference Manual

PREFACE **4**

About this document 4
Contents 4
Further reading 4
Feedback 4

CHAPTER 1. JASMINE OPENSSD PLATFORM SPECIFICATION **5**

1.1. Overview 5
1.2. Jasmine Board 5

CHAPTER 2. INDILINX BAREFOOT™ SSD CONTROLLER SPECIFICATION **8**

2.1. Hardware Architecture 8
2.2. Memory Map 11
2.3. NAND Flash Controller 15
2.4. SATA Controller 18
2.5. DRAM Host Buffer & Buffer Manager 19
2.6. Memory Utility 22
2.7. Interrupt Controller 23

CHAPTER 3. JASMINE OPENSSD PLATFORM FIRMWARE ARCHITECTURE **25**

3.1. Firmware Overview 25
3.2. Host Interface Layer 25
3.3. Flash Translation Layer 26
3.4. Flash Interface Layer 30

CHAPTER 4. JASMINE OPENSSD PLATFORM SOFTWARE SPECIFICATION **32**

4.1. Source File Description 32
4.2. Installer Function 35
4.3. FTL Protocol API 35
4.4. LLD API 37
4.5. Memory Utility API 40

Preface

About this Document

This document describes about the Barefoot™ controller based Jasmine OpenSSD platform's hardware, software, and SSD software development. The main contents of this document are as follows:

- Characteristics of the Jasmine board which is the reference board for the Jasmine OpenSSD platform
- Architecture of Indilinx Barefoot™ SSD controller
- SSD firmware's characteristics on Jasmine OpenSSD platform
- Jasmine OpenSSD platform's SSD software structures and API introduction

Contents

Chapter 1. Jasmine OpenSSD Platform Specification

This chapter explains the Jasmine board's hardware architecture.

Chapter 2. Indilinx Barefoot™ SSD Controller Specification

This chapter explains the hardware architecture of the Barefoot controller and internal controllers.

Chapter 3. Jasmine OpenSSD Platform Firmware Architecture

This chapter explains the SSD firmware of the Jasmine board.

Chapter 4. Jasmine OpenSSD Platform Software Specification

This chapter explains the structure and core API of the SSD software.

Further Reference

- Refer the documents at www.arm.com for the ARM7 processor architecture.
- Refer OpenSSD Project for detailed information about the Jasmine OpenSSD platform.
- Refer OpenSSD project manual for technical information.

Feedback

- OpenSSD project website (http://www.openssd-project.org/)

Chapter 1.
Jasmine OpenSSD Platform Specification

This chapter includes the hardware architecture specifications of the Jasmine OpenSSD platform

- ✓ Jasmine OpenSSD Platform overview
- ✓ Jasmine Board

1.1. Overview

Jasmine OpenSSD platform includes the reference board (Jasmine board) loaded with Indilinx's high-performance Barefoot™ SSD controller, and SSD firmware (Jasmine firmware) which supports SATA 2.0.

1.2. Jasmine Board

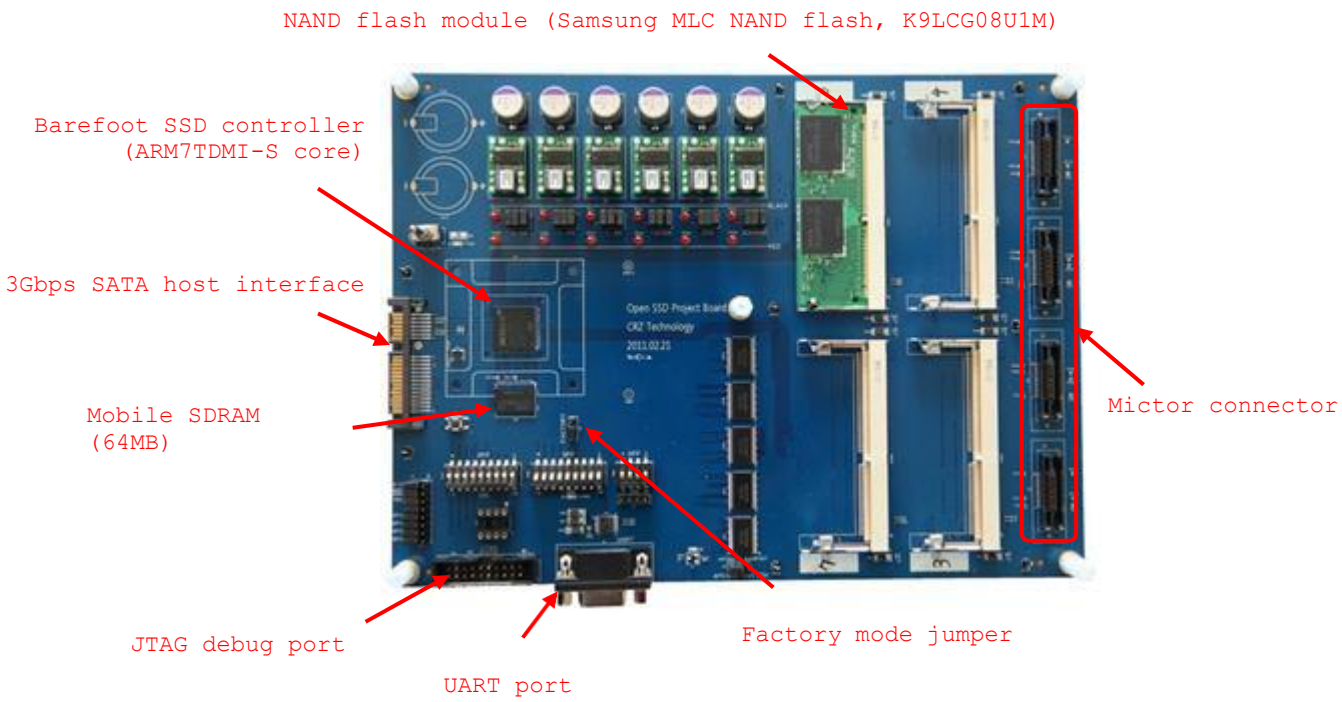

Figure 1) Jasmine OpenSSD Platform: Jasmine board containing Barefoot SSD Controller

The below are the hardware features of the Jasmine board:

- Indilinx Barefoot™ SSD controller
  - ARM7TDMI-S core running at 87.5Mhz
  - DRAM access bus, flash/SATA control running at 175Mhz
  - 96KB internal RAM
  - SATA 2.0 host interface (3Gbps) with NCQ support
  - Mobile SDRAM controller up to 64MB (running at up to 175MHz)
  - NAND flash BCH 8/12/16 bit correction per sector
  - SDRAM Reed Solomon 2 byte correction per 128 + 4 byte
  - NAND flash controller up to 64 CE's (4 channels, 16 bits/channel, 8 banks/channel)
  - Separate DRAM access bus for transferring data between NAND flash memory and DRAM buffer
  - Supports various NAND flash memory chips from different vendors such as Samsung, Hynix, Toshiba, Micron, etc.
  - Specialized hardware for buffer management and memory utility functions
  - Debugging/monitoring aids
    - JTAG
    - UART
    - 1 LED and 6 GPIO pins
    - Mictor connector to NAND flash signals for logic analyzer
    - Separate current measurement points for core, I/O, SDRAM, and NAND
- Mobile SDRAM
  - 64MB from Samsung (subject to change)
- 8 NAND flash memory slots (DIMM)
  - 64GB from Samsung (subject to change)

1.2.1. Indilinx Barefoot™ SSD controller

Chapter 2 describes the specifications of the Indilinx Barefoot SSD controller.

1.2.2. Factory Mode Jumper

Jasmine board contains the Factory mode jumper, connected to the Barefoot controller GPIO #0 pin. Once the power is on, the CPU address space 0 mapped to the ROM initiates the ROM code. The ROM code initializes the hardware and checks the condition of GPIO #0. Depending on the GPIO pin value, the next instruction can differ as below:

- If the value of the GPIO #0 equals 0 (Jumper placed at Normal mode)

ROM code identifies the address and numbers of the installed NAND flash, reads the firmware image from block #0 and finally loads it to SRAM. Then, address remap operation called as 'jump' moves the CPU address space 0 from ROM to SRAM. Thus, initiates the firmware. Firmware loading halts and the board goes to the Factory mode, if the contents of block #0 is missed or corrupted or if the page #4 signature (0xC0C2E003) of the block #0 is lost.

- If the value of the GPIO #0 equals 1 (Jumper placed at Factory mode)

  ROM code initiates the SATA interface and waits for the commands from install.exe.

Chapter 2.
Indilinx Barefoot™ SSD Controller Specification

OpenSSD platform's Indilinx Barefoot™ controller is ARM based SATA controller, which is currently loaded on various SSD products. This chapter specifies the hardware architecture of the Barefoot Controller. This chapter describes the following elements:

- ✓ Indilinx Barefoot controller architecture
- ✓ Memory map
- ✓ NAND flash controller (FCP, WR & BSP)
- ✓ SATA controller (NCQ, SATA event queue)
- ✓ DRAM host buffer & Buffer manager
- ✓ Memory utility
- ✓ Interrupt controller

2.1. Hardware Architecture

Figure 2 shows Jasmine platform's hardware architecture as a diagram.

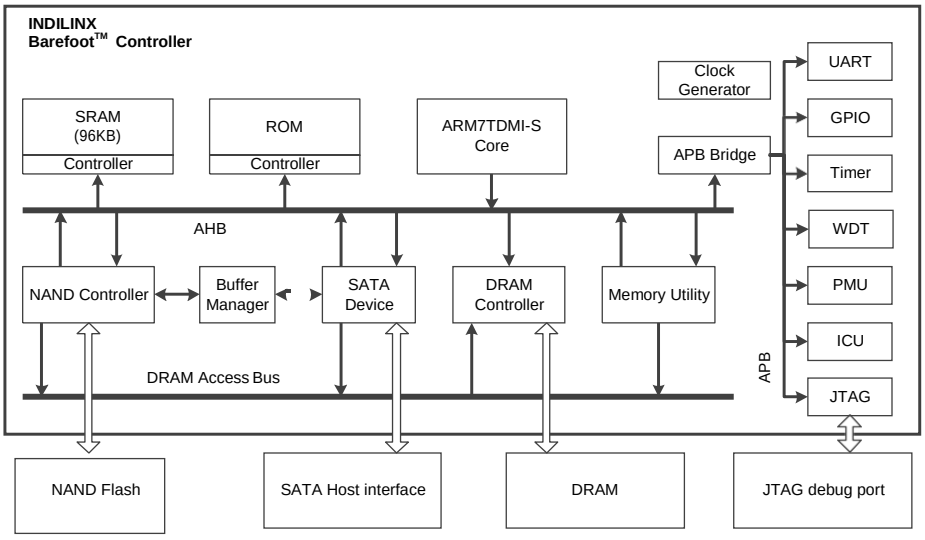

Figure 2) Jasmine OpenSSD Platform architecture

Barefoot controller is an ARM based SATA-compatible SSD controller loaded on various SSD products. Barefoot controller containing the 16/32-bit ARM7TDMI-S RISC microprocessor of the ARM Ltd. ARM7TDMIS-S is implemented as von Neumann architecture and also contains an AMBA bus inside.

Barefoot controller is composed of 96KB SRAM, system manager (SDRAM/NAND controller), buffer manager, SATA 2.0 host interface, clock generator, UART, timer, WDT (Watchdog timer), PMU (Power Management Unit), ICU (Interrupt Control Unit), and JTAG. Barefoot controller controls various external components (SLC/MLC NAND flash, Mobile DRAM, and more).

2.1.1. NAND Flash Architecture

NAND flash architecture of the Barefoot controller is designed as multichannel and in a multi-way, to provide high bandwidth of SSD. Because, each channel connects to a NAND flash memory chip in each way to perform IO operations at the bank level.

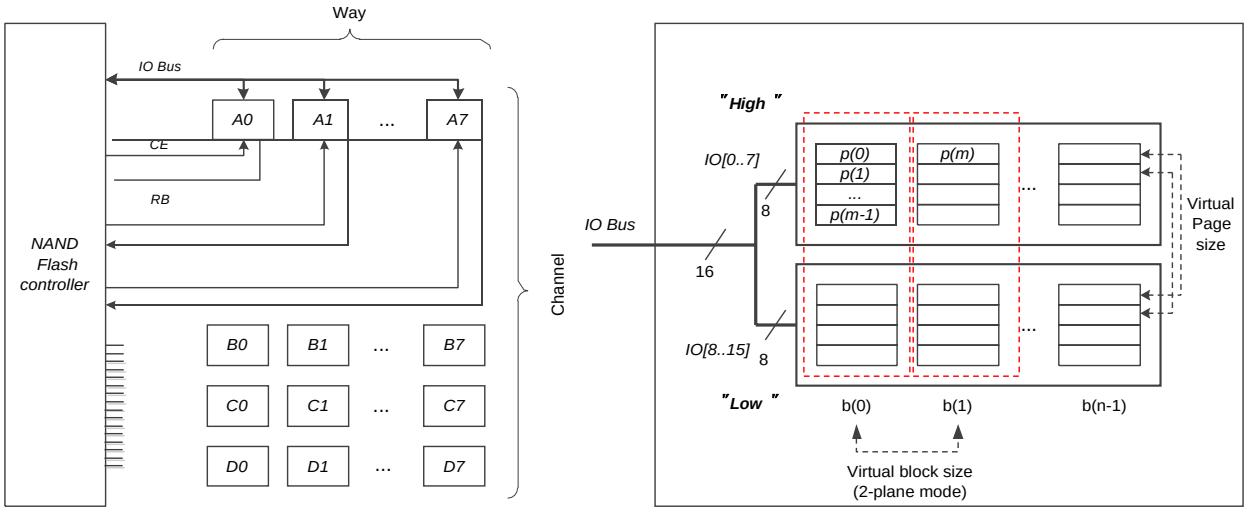

Figure 3) NAND flash architecture of the Jasmine OpenSSD platform

As shown in the left side of the figure 3, NAND flash architecture of the Jasmine platform contains four channels, which can operate in an independently parallel way. There are eight banks in each channel, composed of two hardware connected NAND flash chips and input and output of these banks comes through 16-bit IO bus. Flash controller controls the NAND flash through some control pins including CE input pin, R/B output pin, etc.,

---

NOTE: Actually, CE pin has two pins, which controls two chips. Both the pins should either be enabled or disabled at a time (c.f it is possible to enable/disable a specific chip only. Refer the FO_H, FO_L options in Table1).

Eight banks connected to one channel shares the IO bus, but the IO operation can be performed on two banks at the same time. However, internal cell operation of the NAND flash chip can perform parallel IO operations on all the banks.

The flash controller has only one R/B output pin per 2 banks which resides in one channel, maximum of 4 banks can perform interleaving IO at the same time. For example, in case of channel A, one R/B output pin is connected with A0/A4, A1/A5, A2/A6 and A3/A7 banks. Therefore, the cell operation is unable to perform at the same time for the bank A0 and A4.

Figure 4 shows the process of logging 512B data to a specific bank. The chip transmitting upper byte of 2 bytes named as 'high'. Contrary, 'low' is the name of the chip transmitting the lower byte of two bytes. From the perspective of the firmware, these two chips are considered as virtual, and the virtualization is done by Barefoot controller.

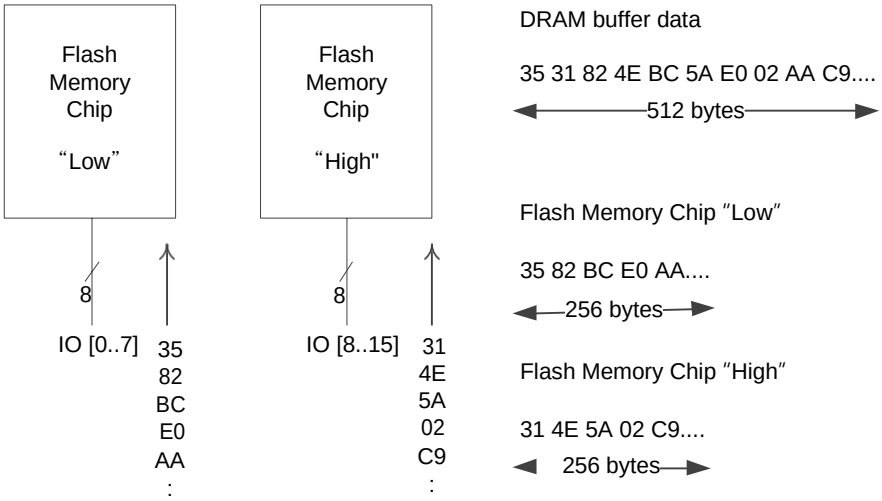

Figure 4) Bank interleaved operation

The command unit of the firmware to hardware is a page and the page(s) virtualized by the hardware called as a Virtual page (VPAGE). As explained, one page of the low chip and one page of the high chip always receives the read/write operation at the same time, so the size of the virtual page is 2 times the size of the physical page.

Almost every flash memory supports '2-plane mode' feature, which is an acceleration feature that ties two pages as a single page making read/write on two pages at the same time. From the perspective of the firmware, the number of page decreases to half and the size of one page becomes twice (because two pages from different blocks tied as one page and the number of pages per block remains the same. Only the total number of blocks decreases to half). Barefoot controller also supports 2-plane mode. As a result, on Jasmine platform, virtual page size is doubled the physical page size in 1-plane mode. Where as in 2-plane mode, size of virtual page is four times the physical page size.

Block erase operation, same as the read/write is also the object of virtualization. Therefore, in 1-plane mode, virtual block size is twice the physical block size and in 2-plane mode, virtual block size is four times the physical block size.

In order to maximize the IO parallelism on Jasmine platform, enable both of the NAND flash chips. However, by selective enabling of these chips (FO_L, FO_H), the platform is forced to ignore the command/input signal from flash controller. For example, from Figure 4, activates the high chip when the flash command delivers FO_H option and the data logging occurs only on Low chip. In case of read operation, unpredictable values can be read from high chip and is ignored for a write operation.

ECC/CRC engine will not perform its operation to a specific chip, IO operation with activated FO_L or FO_H option will cause "uncorrectable data corruption" interrupt. To avoid this kind of interrupt, skip the FO_E option while delivering flash command.

NOTE: Refer 'Jasmine board schematic' tab and 'NAND flash module schematic' tab in the OpenSSD Project website for the detailed explanation of hardware design of the Jasmine board.

NOTE: Refer 2.3.1 section of this document for information about Flash Command Port (FCP).

2.2. Memory Map

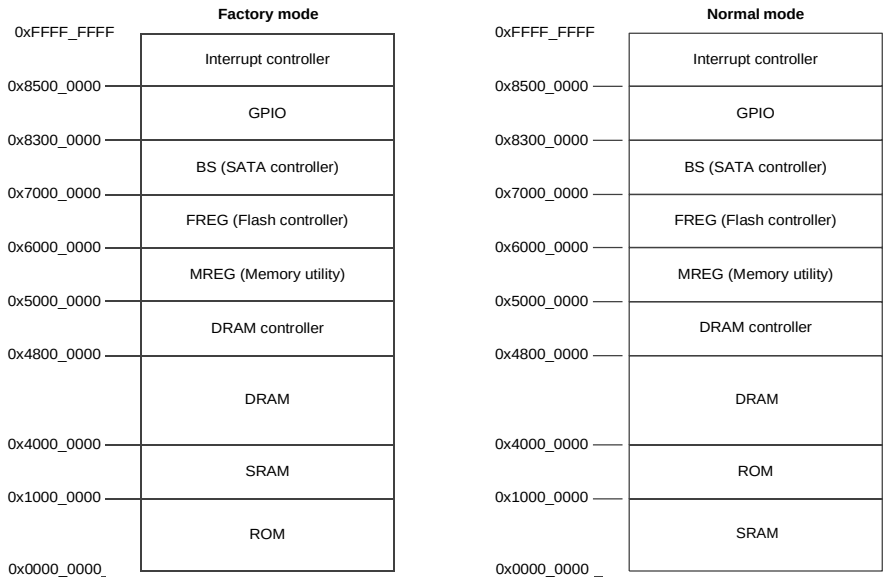

Figure 5) Memory Map of the Barefoot Controller

ROM has Various codes for 'Factory mode' booting.

NOTE: *Factory mode* is a firmware installation mode on Jasmine board. Booting with Factory mode will execute the code saved in ROM, and the user can install binary image of the firmware and metadata for firmware operation. Jasmine firmware logs this information on VBLK #0. If the Jasmine board power is on, after firmware installation, bootloader will load the logged firmware image from VBLK #0 and then performs the firmware operations.

In this mode, ROM maps to the address 0x00000_0000 and proceeds with codes for firmware installation. The '*Normal mode (Non-factory mode)*' interchanges the ROM memory address and SRAM memory address.

In *Factory mode*, SRAM is mapped to the address 0x1000_0000 and loads the ROM code. Where as in *Normal mode,* bootloader, main firmware image and ZI/RO/RW data of the firmware are loaded.

On the other hand, SDRAM, as shown in Figure 6, splits into buffer space and FTL metadata space. DRAM buffer, divided into SATA read/write buffer and copy buffer. The data inside each buffer becomes input/output of the flash memory by DMA controller. SATA read/write buffer stores the user data request upon receiving at SATA event queue. Copy buffer, used for copy-back and modified copy-back command.

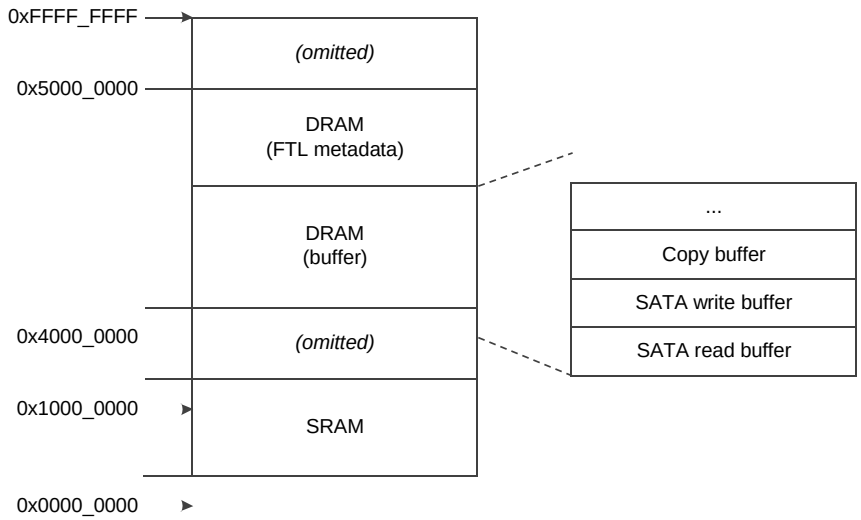

Figure 6) Memory map (DRAM segmentation)

SATA Read buffer

SATA Read buffer contains the user data, which reads from the NAND flash. This buffer is loaded with the data requested by the hosts and the SATA controller delivers the data to the corresponding host.

This buffer only stores user data and not used to store any kind of internal data like mapping table.

SATA Write buffer

SATA controller loads the user data into this buffer, FTL writes the user data on to the NAND flash.

Copy buffer

`FC_CPBACK` and `FC_MODIFIED_CPBACK` commands are used by this buffer to copy page X's data to page Y, which are existing on the same plane of the NAND flash memory chip.

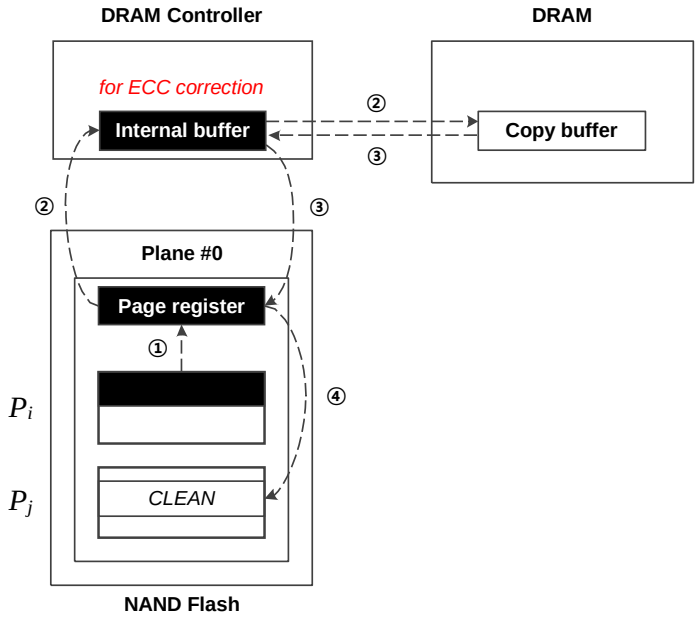

Figure 7) Copy buffer for copy-back operation

Copy buffer for page copy operation works the same way as the primitive operation of the NAND flash memory. However, using Copy buffer check the ECC validity right before the page copy. Figure 7 shows the `FC_CPBACK` operation. To copy a page from P_i to P_j, which are in the different blocks but on the same plane, first loads the data to the internal page register. Then, performs ECC check to validate the data in the internal temporary buffer and copy buffer of the DRAM. If ECC correction is required, then create a new ECC for Copy buffer and, copy to P_j in order of ③, ④. In case if ECC correction is not needed then skip the step ③ and directly copy the data in the page register from P_i to P_j by step ④.

NOTE: Apart from the SATA read/write and Copy buffers, Firmware can use the remaining space in the DRAM.

Address space 0x4800_0000 mapped to the DRAM memory controller register and the address space 0x5000_0000 mapped to the memory utility register set (MREG). DMA works based on the register value mentioned in this section.

NOTE: The hardware memory utility in the Barefoot controller allows the Barefoot controller to support DMA memory copying between DRAM and SRAM and quick memory searching feature with its H/W engine.

SATA controller register set (BS) mapped to the address space 0x7000_0000 and address 0x8300_0000 is allocated for GPIO pin mapping.

The Interrupt register set mapped to the address space 0x8400_0000. Interrupt controller handles the interrupts from the SATA/Flash/DRAM controller and from the components connected to the APB (Advanced Peripheral Bus).

Finally, flash controller register set mapped to the address space 0x6000_0000. Flash controller uses special architectures like FCP, WR, and BSP, in order to perform IO commands on flash memory. Based on the register feature, each register mapped to this area respectively. Especially, as shown in Figure 8, Barefoot controller has extra space for 32 banks, in order to supply large size SSDs. Register sets are described in detail in Chapter 2.3.

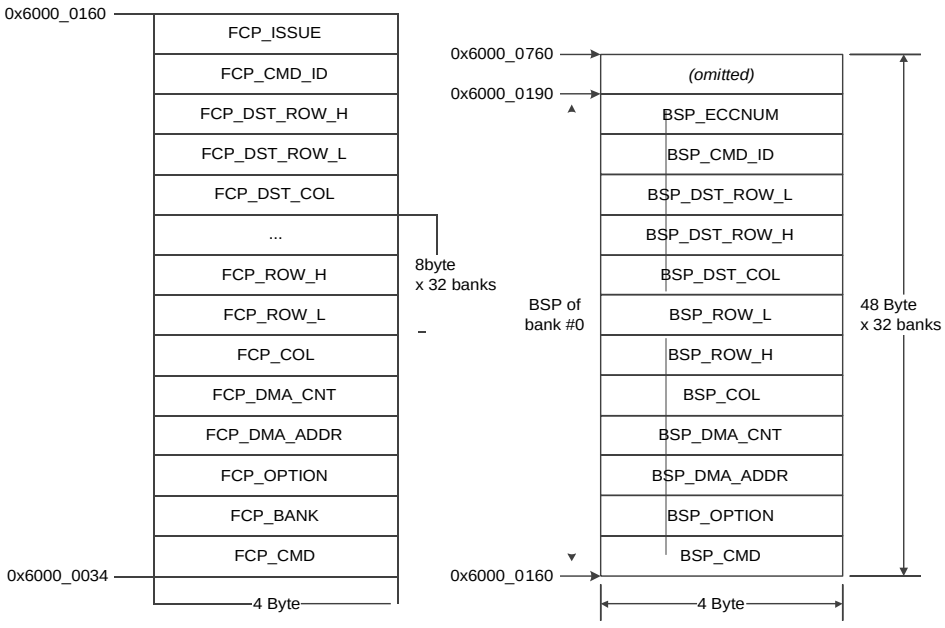

Figure 8) Memory map (FCP & BSP)

2.3. NAND Flash Controller

NAND flash controller uses FCP (Flash Command Port), WR (Waiting Room), and BSP (Bank Status Port) to deliver its IO requests efficiently in multiple bank environment.

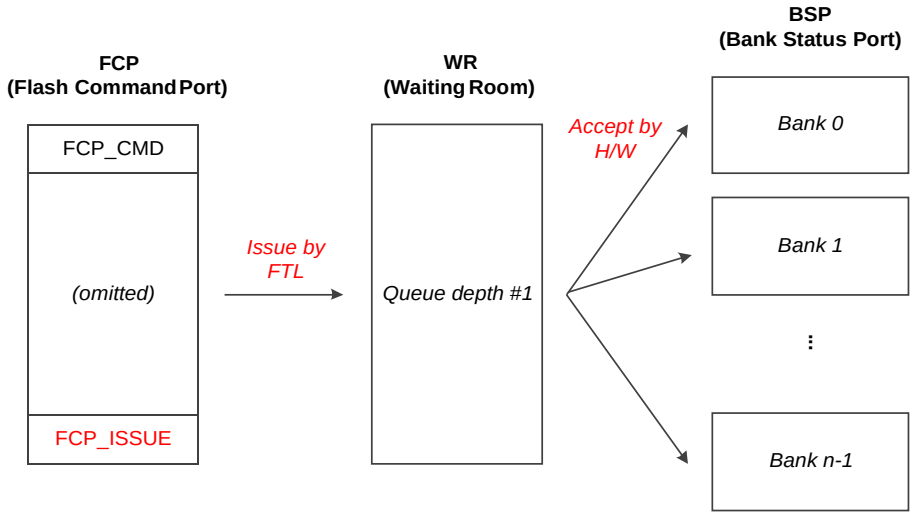

Figure 9) FCP, WR and BSP

The flow of requesting an IO operation is as follows: First, set the IO commands like page read/write or block erase on FCP. Then, WR receives the commands issued by the FTL. Hardware delivers the IO command waiting in the WR to the corresponding bank. If the state of the corresponding bank is busy then the command waits in the WR until the previous IO command completes. BSP logs the IO command delivered to the flash. BSP_INTR register logs the state information, if any interrupt occurs.

2.3.1. FCP (Flash Command Port)

Set the FCP to deliver the commands like page read/write or block erase on flash memory. FCP register sets have the options: flash command, bank number, row/column address, buffer address and flag. First, Set the FCP to deliver the commands like page read/write or block erase on flash memory. FCP registers to be set first have options like flash command, bank number, row/column address, buffer address and flag. After setting the FCP registers, the issue performed will be deliverd to WR with the corresponding FCP.

FCP register set is explained in the below Table 1. FCP register set configures only when FTL requests IO related commands to flash memory in LLD (Low-Level device Driver) level, which is known as flash memory interface.

Table 1) FCP register set

Register	Address	Description
FCP_CMD	0x6000_0034	NAND flash command configuration 0x00 = FC_WAIT 0x01 = FC_COL_ROW_IN_PROG 0x02 = FC_COL_ROW_IN 0x03 = FC_IN 0x04 = FC_IN_PROG 0x09 = FC_PROG 0x0a = FC_COL_ROW READ_OUT 0x0b = FC_COL_ROW_READ 0x0c = FC_OUT 0x0f = FC_COL_OUT 0x10 = FC_READ_ID 0x12 = FC_COPYBACK 0x14 = FC_ERASE 0x15 = FC_GENERIC 0x16 = FC_GENERIC_ADDR 0x17 = FC_MODIFY_COPYBACK **Note**: Refer to ./include/flash.h for the operation sequence of the above flash command codes.
FCP_BANK	0x6000_0038	Bank number (max: NUM_BANKS) 0x3F = *auto-select mode*(AUTO_SEL) **Note:** If set to AUTO_SEL, once issues the FCP command, the first idle bank will take the contents of WR. This supports parallelism.
FCP_OPTION	0x6000_003C	FCP option flag configuration. 0x001 = FC_P 0x006 = FO_E 0x008 = FO_SCRAMBLE 0x010 = FO_L 0x020 = FO_H 0x040 = FO_B_W_DRDY 0x080 = FO_B_SATA_W 0x100 = FO_B_SATA_R FO_P: 2-plane mode FO_E: ECC & CRC hardware enable FO_SCRAMBLE: enable data scrambler FO_L: disable LOW chip FO_H: disable HIGH chip FO_B_W_DRDY: ready data to write in write buffer FO_B_SATA_W: release write buffer when FCP command is completed FO_B_SATA_R: release read buffer when FCP command is completed **note**: Refer to ./include/flash.h, for detail explanation of above flash option flags.
FCP_DMA_ADDR	0x6000_0040	Buffer address for DRAM to flash or flash to DRAM. **Note:** Buffer address must be a multiple of 512B.
FCP_DMA_CNT	0x6000_0044	Data size (Unit: Byte, must be a multiple of 512B)
FCP_COL	0x6000_0048	Location of the starting column (Unit: Byte, In case using FO_E, must be a multiple of 512B, max: SECTORS_PER_PAGE - 1)
FCP_ROW_L FCP_ROW_H	0x6000_0048 0x6000_004C	Target page number, based on total virtual page numbers. (max: PAGES_PER_VBLK - 1) Number of this register is exactly same as the number of the banks. H/L chip means 'High' and 'Low' respectively. (Generally, both H/L chip uses same row number) **Note**: The reason for using the same number of banks and registers in *auto-select mode is* to assign each bank's target page location.
FCP_DST_COL	0x6000_0118	Starting column location of the destination virtual page. (Set when using 'copy-back', which is internal command of NAND flash)
FCP_DST_ROW_L FCP_DST_ROW_H	0x6000_0150 0x6000_0154	Virtual page number, which will be the destination. (Set when using 'copy-back', which is internal command of NAND flash)

FCP_CMD_ID	0x6000_0158	FCP command id Register used in debugging mode, when delivered to WR, hardware numbers its command sequence automatically.
FCP_ISSUE	0x6000_015C	A register to deliver FCP command to WR When a value is written on this register, corresponding FCP command is issued to WR.

2.3.2. WR (Waiting Room)

Commands related to WR, before delivered to the flash memory waits here and contains the same information as FCP. Flash controller checks the state of the target bank, if the state is idle then delivers the content of WR to the bank, if the state is busy then waits on WR. Figure 9 shows that WR can handle only one FCP request at a time.

Table 2) WR register set

Register	Address	Description
WR_STAT	0x6000_002C	Register to check the state of the WR. **Note**: The below condition shows that WR is empty (WR_STAT & 0x0000_0001 == 0)
WR_BANK	0x6000_0030	In case delivered FCP to NAND flash with Auto-select mode, this register takes that command and checks the bank number

NOTE: If a new FCP command is issued while a FCP command already exists in WR, H/W is not aware weather the delivered command is performed or not. Therefore, firmware first checks WR_STAT register, if the register is in idle state then only delivers new flash command.

2.3.3. BSP (Bank Status Port)

As shown in Figure 8, BSP is composed of the FCP issued flash command information and added information during that issue i.e., BSP contains information from WR. BSP exists in all the 32 banks, from 0x6000_0160 to 0x6000_070, with a size of 48 bytes. BSP contains the information of the last performed operation and is very useful for debugging.

BSP_INTR register and BSP_FSM register stores the interrupts occurred during internal flash operation. BSP_INTR stores the cause of the interrupt occurrence as shown in Table 3, and BSP_FSM stores the current condition.

NOTE: Only firmware can clear the BSP_INTR register.

Table 3) BSP_INTR & BSP_FSM register

Register	Address	Description
BSP_INTR	0x6000_0760: 0x6000_0780	Bank interrupt info. (1Byte) separately exists in every Bank 0x01 = FIRQ_CORRECTED 0x02 = FIRQ_CRC_FAIL 0x04 = FIRQ_MISMATCH 0x08 = FIRQ_BADBLK_L 0x10 = FIRQ_BADBLK_H 0x20 = FIRQ_ALL_FF 0x80 = FIRQ_ECC_FAIL 0x82 = FIRQ_DATA_CORRUPT
BSP_FSM	0x6000_0780: 0x6000_0800	Bank FSM (Finite State Machine) info. (1Byte) Same as BSP_INTR, separately exists in every bank. 0x0 = idle others = no idle

2.4. SATA Controller

Barefoot controller itself contains the SATA controller, which manages the data communication between host and device. SATA controller manages a separate command queue (event queue) different from NCQ (Native Command Queuing), which delivers the IO commands efficiently to FTL.

NOTE: Refer to OpenSSD community for more technical information about SATA Operation.

2.4.1. SATA Protocol

The IO commands sent to the SSD by the hosts are overtook by SATA protocol and the SATA protocol sends a response to the host automatically. If the command is a write, then the firmware requests to start the data transfer and the hardware handles this data transfer process automatically.

If any exception arises while performing the IO operation, the hardware will forward the same automatically. If the IO operation is success without exception, then the hardware will send the commit message automatically.

2.4.2. SATA NCQ

SATA NCQ is a command protocol for SATA, which can receive multiple commands simultaneously from one drive.

NCQ of the Barefoot controller with SATA 2.0 can receive maximum of 32 host commands and follows FIFO structure. Actually, the commands delivered to NCQ are not transferred. From the host perspective, unfinished commands wait in NCQ. These commands are moved to the SATA event queue and are delivered to the FTL.

2.4.3. SATA Event Queue

SATA event queue, from the host perspective, manages only the queueing feature of the already finished SATA data transfer commands. As the actual IO operation has not yet occurred, these commands wait for the transfer process from DRAM to NAND flash.

As the below Figure 10 shows, SATA event queue can contain maximum of 128 commands same as the SATA NCQ which operates FIFO order. FIQ interrupt delivers the host commands and are removed when FTL take the corresponding commands.

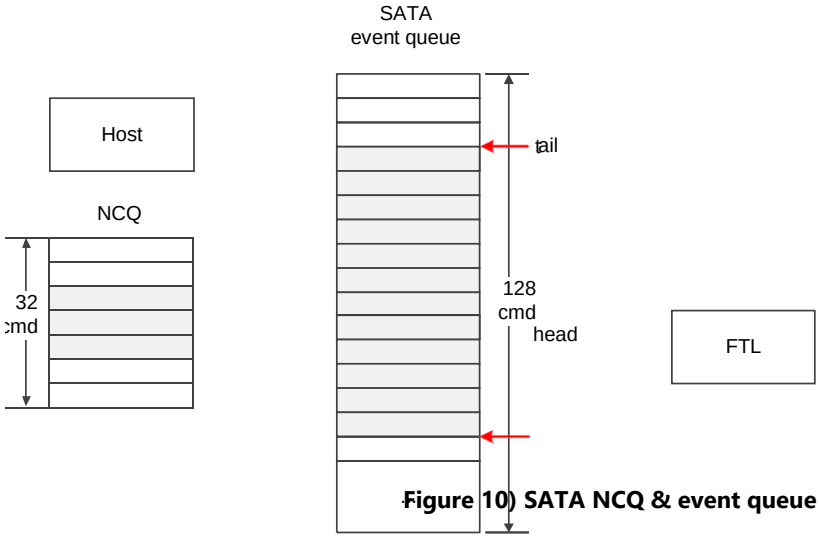

Figure 10) SATA NCQ & event queue

The below mentioned issues may occur when SATA event queue operates in FIFO order:

1. Host waiting time for the Read command can be longer.
2. During a Read command, while the data is transferring from DRAM to host, there is a high probability that FTL and NAND flash memory could be in idle state degrading the performance.

To overcome these issues, the SATA event queue performs the READ operation first and then the WRITE operation. However, this approach can lead to data coherence problem for "*read-after-write*" operation on some addresses. For example, while `<WRITE, lsn=3>` command is stored in the SATA event queue, if the FTL handles the `<READ, lsn=3>` command first, old data is retrieved from the NAND. Therefore, hardware applies the history log search to ensure the data coherence. To avoid these kind of issues, hardware handles the same in the sector duplication case.

2.5. DRAM Host Buffer & Buffer Manager

DRAM host buffer, buffers the user data from the IO operations of the SATA event queue. It is branched into SATA read buffer and SATA write buffer. SATA read buffer buffers the data from the flash memory during the host read request. SATA write buffer buffers the data from the flash memory during the host write request. These buffers operate in a circular format. Hardware buffer manager controls the flow control of the DRAM host buffer and the SATA controller.

2.5.1. SATA Read/Write Buffer

The basic frame unit size of the SATA read/write buffer is same as the VPAGE size (e.g., 4~32KB). Each buffer size differs by the FTL metadata inside the DRAM. Normally, SATA read buffer uses 1MB and SATA write buffer uses few tens of MB.

NOTE: DRAM host buffer space depends on the FTL metadata size. Therefore, listed in the FTL header file. (`ftl.h`)

Table 4) SATA register for read/write buffer

Register	Address	Description
`SATA_BUF_PAGE_SIZE`	0x7000_00B4	SATA buffer frame size (default size = BYTES_PER_PAGE)
`SATA_WBUF_BASE` `SATA_RBUF_BASE`	0x7000_0170: 0x7000_0174	base address of SATA write/read buffer
`SATA_WBUF_SIZE` `SATA_RBUF_SIZE`	0x7000_0178 0x7000_017C	Number of SATA write/read buffer frames
`SATA_RESET_WBUF_PTR` `SATA_RESET_RBUF_PTR`	0x7000_0184 0x7000_0188	Firmware can directly use the pointer reset register of the SATA write/read buffer frame for buffer management.
`SATA_WBUF_PTR` `SATA_RBUF_PTR`	0x7000_0194 0x7000_0198	Pointer of the SATA write/read buffer (id #)
`SATA_WBUF_FREE`	0x7000_019C	Number of free buffer frames of the SATA write buffer
`SATA_RBUF_PENDING`	0x7000_01A0	Number of pending buffer frames of the SATA read buffer

The signal communication between SATA, Buffer Manager and NAND Controller handles the actual management of the SATA read/write buffer. In case if needed, firmware can also manage the SATA read/write buffer.

Refer to the chapter 2.5.2 for more information about SATA read/write buffer management.

2.5.2. Buffer management

DRAM host buffer is managed by the adjustment between SATA, hardware buffer manager and pointers of FTL (i.e., `sata_xxx_ptr`, `bm_xxx_limit`, `ftl_xxx_ptr`). Meanwhile, as explained in the previous chapter, DRAM host buffer works in the circular buffer format, each buffer frame pointers must grow in an ascending order.

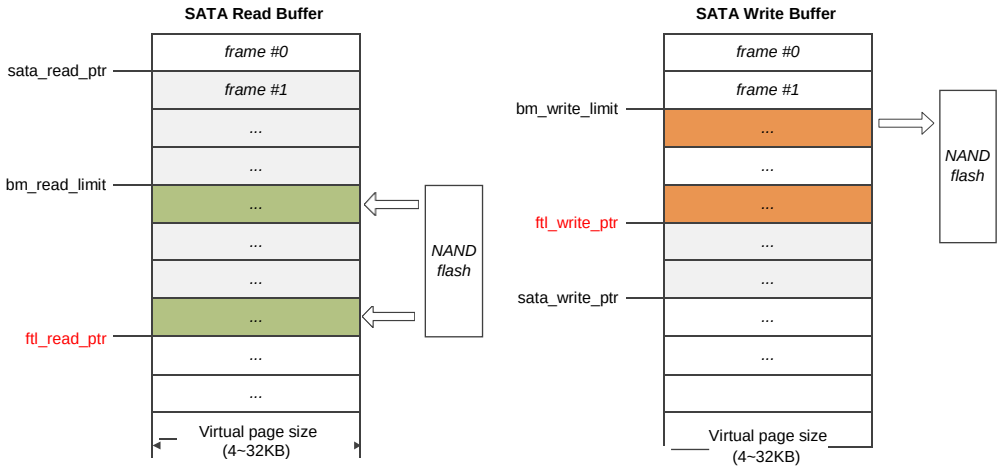

Figure 11) DRAM Host Buffer Management

The order of the SATA read buffer management is mentioned below:

1. First, current buffer frame pointed by the `ftl_read_ptr` is an address. When flash controller delivers the read operation to the NAND flash, DMA loads the data on the target page of the NAND flash.
2. Firmware increments `ftl_read_ptr` and handles next IO operation.
3. SATA controller passes the data loaded by the FTL to the host sequentially by incrementing the `sata_read_ptr`.

Because of the bandwidth gap between flash memory and SATA, it is possible that `sata_read_ptr` overtakes `ftl_read_ptr` and causes wrong data delivery to the host. To avoid this problem, the pointer `bm_read_limit` managed by the buffer manager, ensures the data copied completely from the NAND flash to the SATA read buffer is delivered to the host. In other words, when the flash controller issues host manager that a read operation of a specific bank is completed, then buffer manager increments `bm_read_limit` pointer, which allows SATA controller to send the data to the host resulted from the read operation.

NOTE: When the speed of the read operation from FTL to flash controller is too fast, `ftl_read_ptr` can take over `sata_read_ptr`, resulting in a overwriting of the undelivered host data. In order to prevent this problem, Firmware should control the pointers. Make `ftl_read_ptr` not to take over `sata_read_ptr`. (c.f. `ftl_read() in ./ftl_tutorial/ftl.c`)

SATA write buffer operation is same as SATA read buffer and is as follows:

1 SATA controller increments `sata_write_ptr`, the write request of the host is queued on the event queue and the host data is delivered to the SATA write buffer. Then DMA delivers the data to NAND flash.

2. FTL receives write request from the event queue and delivers it to the flash controller and increments `ftl_write_ptr`.
3. On completion of the data delivery to SATA write buffer, SATA controller issues buffer manager, which makes the flash controller to log the host data on the NAND flash.

There is a problem of potential overwrite due to the slow programming speed of the flash memory by *tPROG* (The user data which is not yet written to the NAND flash could be overwritten by SATA, for other write request). To overcome this, the `bm_write_limit` pointer on the SATA write buffer managed by the hardware buffer manager, ensures the write operation by preventing `sata_write_ptr` to outstrip the `bm_write_limit`.

Table 5, below shows the register set of the Buffer Manager. Flow control of the Buffer Manager is done by intercommunicating with NAND controller, when required can be done by firmware using these registers.

Table 5) Buffer Manager Register

Register	Address	Description
`BM_WRITE_LIMIT` `BM_READ_LIMIT`	0x5000_0000: 0x5000_0004	SATA read/write limit pointer (*READ-ONLY*) of the Buffer Manager 0| Actual flow control is done by this register value (Refer to `./ftl_dummy/ftl.c` for example)

`BM_STACK_RESET`	0x5000_0008	Resets BM read/write limit pointer
		`0x01 = reset BM write limit to BM_SATA_WRSET`
		`0x02 = reset BM read limit to BM_SATA_RDSET`

`BM_STACK_WRSET`	0x5000_0028	Register to reset the BM write limit Logs SATA write buffer id on this register
`BM_STACK_RDSET`	0x5000_002C	Register to reset the BM read limit Logs SATA read buffer id on this register

2.6. Memory Utility

Memory utility handles the data communication between SRAM and DRAM. Also performs iterative memory operations like memory initialization (e.g., `mem_set`) and acceleration of searching on specific memory.

Barefoot controller uses separate ECC engine to increase the integrity of DRAM data. 4 bytes of DRAM ECC information is created per `DRAM_ECC_UNIT` (128 Byte). Below is the sequence of data logging from SRAM to DRAM.

1. Reads 132B (128Byte data + 4Byte ECC parity info) from DRAM and stores at the internal temporary memory of the Barefoot controller.

2. In point 1, when needed performs ECC data correction on that temporary memory. (c.f. In fact, DRAM is a memory of high-integrity with a very low ECC correction probability)
3. Reads the data from SRAM and corrects the content of the temporary memory
4. Creates new ECC parity for 128 Byte and logs 132Byte on DRAM.

NOTE: When CPU directly modifies the DRAM data, there is a possibility of ECC information corruption or loss by memory utility. So, the communication between SRAM and DRAM must done via memory utility (`./include/mem_util.h`).

`SDRAM_INTSTATUS` register of the SDRAM controller records the interrupts occurred during memory utility operation.

Table 6) Memory Utility Register Set

Register	Address	Description
`MU_SRC_ADDR`	0x5000_0010	Source memory address. Sets when reads the data from DRAM
`MU_DST_ADDR`	0x5000_0014	Destination memory address. Sets when writes the data to DRAM
`MU_VALUE`	0x5000_0018	Logs new data
`MU_SIZE`	0x5000_001C	Size of the memory space to be set or searched **Note**: In case of `mem_search`, max 32768 Byte
`MU_RESULT`	0x5000_0020	Result value of the memory operation 0xFFFFFFFF means currently performing a memory operation
`MU_CMD`	0x5000_0024	Memory utility command code

`MU_UNITSTEP`	0x5000_0030	Set with unit 'step' in case of iterative memory operation.
		note: `SETREG(MU_UNITSTEP, MU_UNIT_8 | 1);`
		`SETREG(MU_UNITSTEP, MU_UNIT_16 | 2);`
		`SETREG(MU_UNITSTEP, MU_UNIT_32 | 4);`

Table 7) SDRAM Controller Register Set

Register	Address	Description

`SDRAM_INTSTATUS`	0x4800_001C	Interrupt state information caused by DRAM controller
		`0x01 = ECC fail` `0x02 = ECC correction`
		`0x04 = Address Overflow` `0x08 = Deadlock`

2.7. Interrupt Controller

Interrupt controller receives interrupt requests from external devices and internal devices like SATA, flash memory, DRAM, UART, Timer, WDT. Then, issues CPU to handle these interrupts.

Below Table 8, shows register set of interrupt controller. Firmware configures the devices detected on `APB_ICU_CON` register. When a hardware interrupt occurs during runtime, refers `APB_INT_STS` to know from where the interrupt came from. Then it performs the corresponding interrupt handling operation.

Table 8) Memory Utility Register Set

Register	Address	Description
`APB_ICU_CON`	0x8500_0000	FIQ interrupt configuration

`APB_INT_STS`	0x8500_0004	Interrupt state information. From this register, firmware can find the interrupt caused device.
		`0x001 = INTR_SATA` `0x002 = INTR_FLASH`
		`0x004 = INTR_SDRAM` `0x008 = INTR_UART_TX`
		`0x010 = INTR_TIMER_4` `0x020 = INTR_TIMER_3`
		`0x040 = INTR_TIMER_2` `0x080 = INTR_TIMER_2`
		`0x100 = INTR_TIMER_1` `0x200 = INTR_WATCH_DOG`
		`0x400 = INTR_EXT`

`APB_INT_MSK`	0x8500_000C	IRQ interrupt configuration **Note**: When configured with a value, allows interrupts from that device
`APB_PRI_SET1`	0x8500_0054	-
`APB_PRI_SET2`	0x8500_0058	-

Chapter 3.
Jasmine OpenSSD Platform Firmware Architecture

This chapter specify the internal structure of SSD firmware on Jasmine OpenSSD platform.

3.1. Firmware Overview

Main firmware of the Jasmine OpenSSD platform is composed of three main components, HIL (Host Interface Layer), FTL (Flash Translation Layer) and FIL (Flash Interface Layer).

HIL manages SATA host commands and buffer management. The IO requests issued by the host to the SATA controller are pushed to the SATA event queue and are handled later by the FTL sequentially.

FTL is a software layer, views flash memory as a block device like hard disk. Generally, FTL supports address translation, garbage collection and wear leveling features. FTL handles the IO request by passing it to the flash memory. Various kinds of the existing FTLs optimizes the FTL to improve the performance and safety. On the Jasmine firmware, Tutorial FTL, Greedy FTL and Dummy FTL are implemented.

FIL layer handles flash memory. Operations of the delivered flash commands are performed by LLD (Low-level device driver) and the exceptions occurred during the normal operation are detected by the interrupt controller and finally, FTL handles the interrupts.

3.2. Host Interface Layer

3.2.1. Hardware Event Queue

On Jasmine firmware, SATA event queue (Refer to 2.4.3) is implemented as a hardware event queue. READ/WRITE ATA commands are managed by hardware event queue and are handled in HIL (Host Interface Layer) main function and delivers to FTL. The procedure of handling the ATA command by the event queue is described in the below steps.

1. Transfers the ATA command from host to the Jasmine board
2. Calls the FIQ handler if any FIQ interrupt occurs from SATA host interface
3. Reads the corresponding command from FIS (Frame Information Structure). Extract the `cmd_type, lba, sector_count` values from the requested command.
4. In case of READ/WRITE operation (CCL_FTL_D2H/CCL_FTL_H2D), add the corresponding command (`handle_got_cfis() from ./sata/sata_isr.c`) to the hardware event queue.
5. For the rest of the cases, e.g. slow commands like TRIM, save at the variable named `g_sata_context.slow_cmd`.

6. Handles the command in firmware main function (`Main()` of `./sata/sata_main.c`)
   - If there exist a read/write request in the event queue, pick out and handle one by one (`eventq_get()` of `./sata/sata_main.c`)
   - If there is no read/write request available, just handle the command saved in the variable named `slow_cmd`

3.3. Flash Translation Layer

FTL (Flash Translation Layer) is a software layer that helps the host to see the flash memory as a hard disk. FTL retrieves the read/write request from the hardware event queue and passes IO commands to flash controller sequentially.

3.3.1. FTL Protocol Interface

FTL protocol interface is a set of functions to communicate using messages with SATA host interface. This chapter describes the 4 main protocol interface functions (`ftl_open, ftl_read/ ftl_write, ftl_flush`).

NOTE: Refer to chapter 4.2 for FTL protocol API function specifications.

---

ftl_open

This function loads the FTL to receive the host IO requests, after the Jasmine board initializes.

1. When the firmware installation initiates, reads the scan list logged on VBLK #0 to check the initial bad blocks.
2. Sets NAND flash memory to base condition to handle the host IO requests. It also includes the erase operation of the entire blocks to log user data.
3. Loads FTL metadata on SRAM or DRAM from flash memory that has installed along with firmware or metadata logged at the power-off time.
4. Initialize the metadata managed by FTL including volatile variable.
5. Set the interrupt options of the flash controller.

ftl_read

An API to handle host read request delivered by the event queue. Through the mapping information, transfers the read request to the flash memory chip. This flash memory chip has valid pages with its unit size as virtual page size and modifies the SARA read buffer pointer managed by the FTL.

ftl_write

An API to handle host write request delivered by the event queue. First, scan for the free pages and write the new data. After write, modifies the mapping information with the new page.

ftl_flush

This API logs the FTL metadata for POR/SPOR on NAND flash memory. When SATA is in idle/standby state calls this API to maintain FTL metadata consistency and makes POR possible.

3.3.2. Tutorial FTL

Tutorial FTL has implemented to help understanding the FTL operation principles on Jasmine platform architecture. This FTL has very simple structure that uses page-level mapping table and does not include FTL features like garbage correction, wear leveling and POR. This chapter mainly explains the handling of read/write requests by FTL using DRAM.

DRAM Usage

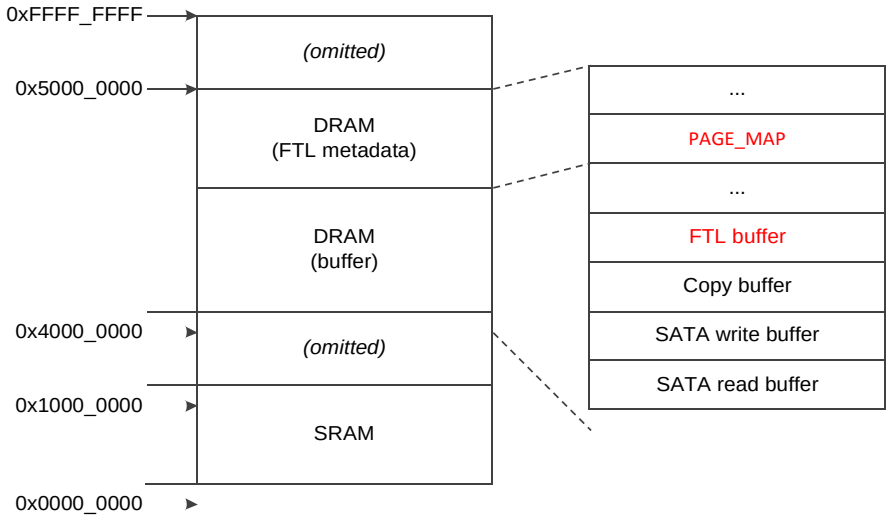

Figure 12) DRAM usage of Tutorial FTL

Dummy FTL allocates a separate 'FTL buffer' inside DRAM buffer space. This FTL buffer is used as a temporary buffer when logging modified metadata on the Flash memory or while reading logged metadata from flash memory.

DRAM FTL metadata space manages page mapping table (`PAGE_MAP`). `PAGE_MAP` contains mapping information of page address to physical page address.

NAND Structure

Tutorial FTL manages flash memory as shown in the below Figure 13. VBLK #0 is written by firmware installation function (`install()` of `./installer/install.c`) during firmware installation. This space stores the data (i.e., bad blk scan list, firmware binary image) needed while FTL operates the firmware. Tutorial FTL uses the rest of the space to sequentially log the user data excluding VBLK #0.

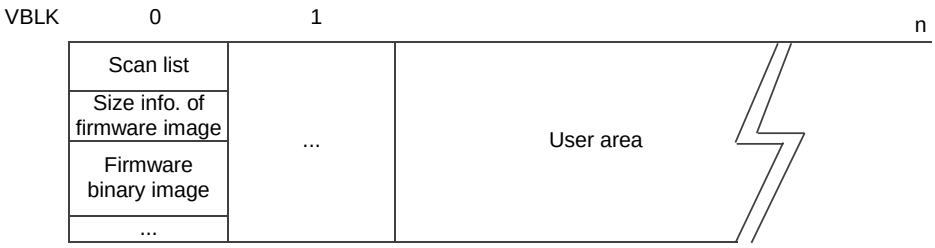

Figure 13) NAND structure of Tutorial FTL

Address Mapping

Generally, FTL performs IO with VPAGE size as its unit, through `<lsn, sector_size>` information delivered by the host. As discussed already, Tutorial FTL manages mapping table in page-level. User data on flash memory can be accessed with direct mapping without any searching process.

Below Figure 14, shows the address mapping sequence of Tutorial FTL. First, FTL receives the read request for LPN 2 from SATA event queue. Then, receives the mapping information of LPN through page mapping table (i.e., `PAGE_MAP`). This information contains corresponding bank number of the LPN and VPN. Finally, FTL transfers corresponding PPN user data by sending flash IO request to FIL.

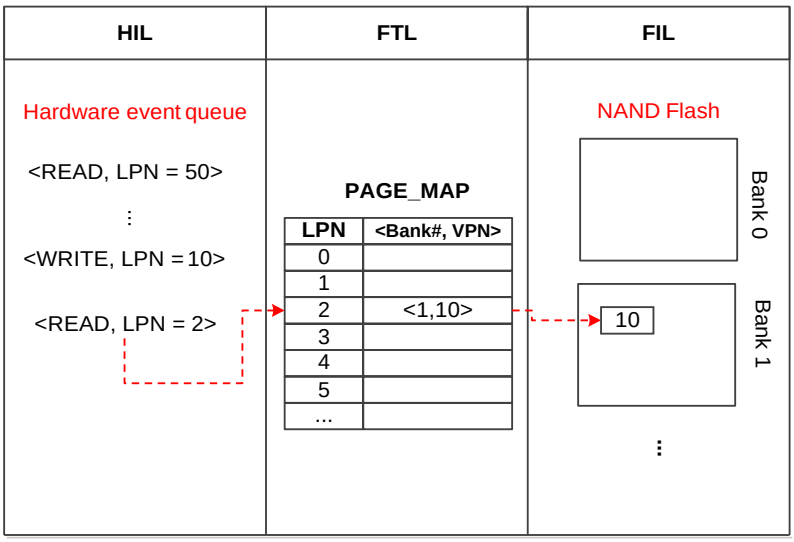

Figure 14) Address Mapping in Tutorial FTL

Read Operation

Tutorial FTL, as explained above, locates the physical location of the user data through address mapping phase. In figure 15, step ① sends the read operation to the flash controller and finally delivers user data to SATA read buffer. After that, hardware notifies the buffer manager about the completion of the user data transfer. Then buffer manager transfers the data in the DRAM buffer to SATA host, as shown in step ②.

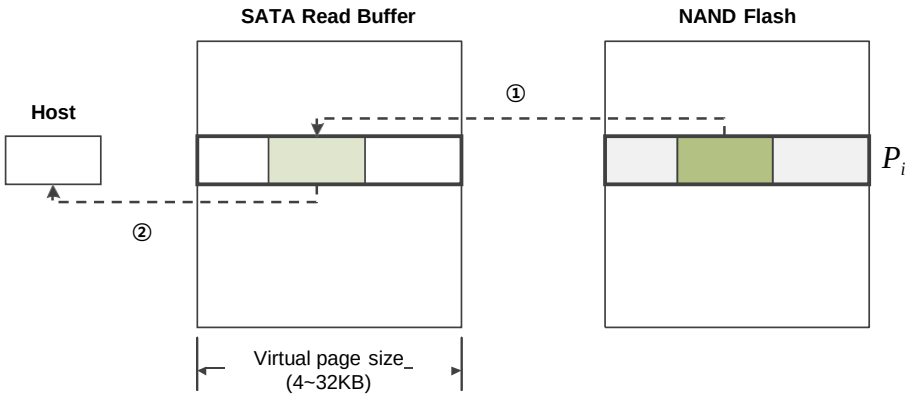

Figure 15) Read operation in Tutorial FTL

Write Operation

Write operation comes with an additional read operation by its size. If an IO operation performs with smaller size than `VPAGE` size, then loads old data from flash memory to temporary buffer and merges it. Then finally performs the write operation.

Below Figure 16, shows write operation with smaller size than `VPAGE` size. First, same as read operation, FTL receives the write operation by polling hardware event queue. Then access the page-mapping table to check whether the corresponding page LPN data exists or not. If data exists for the corresponding LPN, loads rest of the data excluding user data to SATA write buffer (like step ①). Like step ②, hardware waits for new user data from the host. When new data comes, writes on a new page. If no data exists or have to perform a write operation with only page size, skip step ①.

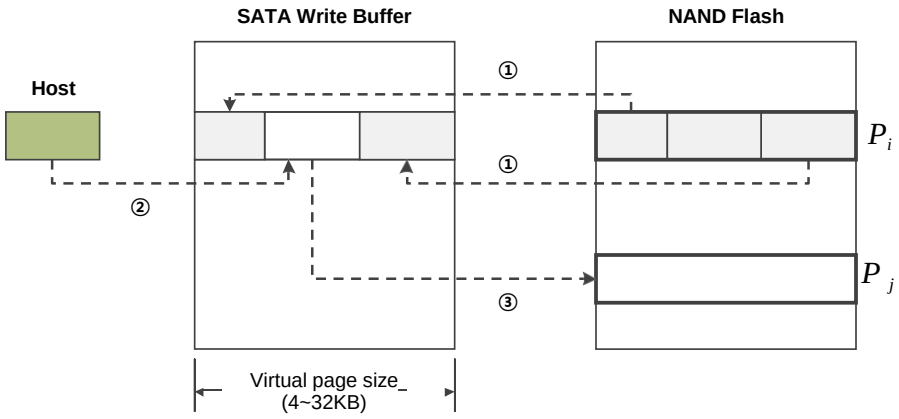

Figure 16) Write operation in Tutorial FTL

The process of allocating free pages to write new data during Tutorial FTL write operation improves parallelism as high as possible. Write operation on a LPN is not fixed to a specific bank, the bank number is incremented by one after each write and logs the bank number and VPN information on the address mapping table (This kind of write operation can degrades the parallelism performance).

Tutorial FTL does not logs modified metadata on flash memory during write operation. However, in order to perform error recovery, it should have enough metadata space inside flash memory to log modified contents of the metadata periodically.

BSP Interrupt Handling

FTL handles the exceptions (e.g., ECC error, runtime bad block occurs, ETC) as hardware interrupts encountered while flash memory handling the requests.

Issues a BSP interrupt when internal error occurs during flash memory handling the request. Then flash controller issues corresponding interrupt to the hardware interrupt controller. After that hardware interrupt controller issues ARM with IRQ interrupt and then ARM calls IRQ interrupt handler (`irq_handler()` in `./target_spw/misc.c`).

If the interrupt is from flash controller, IRQ interrupt calls FTL interrupt service routine (`ftl_isr`). Then, FTL checks interrupt register (`BSP_INTR`) for the reason of the interrupt occurrence. Finally, FTL executes the corresponding exception handling code.

3.3.3. Dummy FTL

Dummy FTL is a virtual FTL to check the speed of SATA and DRAM, implemented with minimal code to handle host request from event queue and can never access flash memory.

Actual Dummy FTL (`./ftl_dummy/ftl.c`) implementation does not perform read/write from/to memory (in case of `ftl_read/ftl_write`). Dummy FTL includes only the SATA read/write buffer pointer-moving feature, in order to make SATA and FTL work normally.

3.4. Flash Interface Layer

FIL delivers the actual IO commands to flash memory from FTL. LLD sets the flash commands (e.g. FCP) and delivers the IO commands to the flash memory.

3.4.1. Flash Command Issue

IO commands delivered to LLD are executed by the flash command delivery function internally (`flash_issue_cmd()` in `./target_spw/flash.c`). This function delivers the configured FCP at LLD to WR.

This firmware provides 3 types of synchronous/asynchronous IO issues, as shown in below Table 9. Using these issues, SSD IO handling increases the parallelism by delivering IO to flash controller.

Table 9) FCP issue type

Type	Value	Description
RETURN_ON_ISSUE	0x0	When WR is empty, delivers FCP contents to WR and return
RETURN_ON_ACCEPT	0x1	Bank gets the flash command delivered to WR and do polling until its start and return
RETURN_WHEN_DONE	0x2	Waits until corresponding bank is done with flash command and return

NOTE: FTL delivers the flash command. Do not deliver another flash command while WR already contains waiting commands.

3.4.2. LLD (Low-level device driver)

LLD is an abstracted interface to set the flash commands on FCP and to deliver that configured FCP to WR. This interface eases the firmware developers to implement because there is no need to implement the configuration of flash controller register every time while delivering the flash commands to flash memory. It makes the flash memory viewed as a simple logical architecture composed of logical page and logical block.

LLD performs flash commands with both VPAGE and VBLK, as its unit size. Some typical examples are (partial) page read/program, block erase, simple copy-back, and modified copy-bac, which are in API.

NOTE: Refer to chapter 4.4 for detailed usage of LLD API types.

Chapter 4.
Jasmine OpenSSD Platform Software Specification

This chapter specify the firmware software of Jasmine OpenSSD platform. Firmware software is composed of firmware source file, build script and firmware installation program. This chapter details about the core modules of firmware API and includes following contents.

- ✓ Jasmine firmware file structure
- ✓ Jasmine Firmware APIs (FTL protocol API/LLD API/Memory utility, etc.)
- ✓ Some important macros

4.1. Source File Description

4.1.1. File Hierarchy

Firmware header files and source files of the Jasmine OpenSSD platform are mentioned in the below table.

Table 10) File hierarchy *(some files are omitted)*

Location	File List	Description
./	release_lock.inc COPYING, HISTORY, README	Debugging script file, Intellectual property rights and revision history and build overview of the Jasmine firmware
./build_gnu	Makefile, ld_script build.bat	Firmware build script using Code Sourcery G++ toolchain
./build_rvds	armlink_opt.via file_list.via build.bat	RVDS compile environment configuration file and firmware build script
./ftl_tutorial	ftl.h, ftl.c	Tutorial FTL related file folder
./ftl_greedy	ftl.h, ftl.c	Greedy FTL related file folder
./ftl_dummy	ftl.h, ftl.c	Dummy FTL related file folder
./sata	sata_cmd.c sata_identifiy.c sata_isr.c sata_main.c sata_table.c	SATA interface related source file folder
./installer	ata_7.h, installer.c, installer.sln, ntddstor.h	Firmware install solution file. Creates install.exe
./include	jasmine.h mem_util.h flash.h, peri.h rom.h, *etc.*	Firmware source code related header file
./target_spw	init_gnu.s, init_rvds.s flash_warpper.c initialize.c target.h mem_util.c, flash.c, *uart.c etc.*	Firmware startup code/Firmware initialization/ Memory utility/LLD API/UART/Timer utility, ETC.
./tc	tc_synth.c	Folder includes the test case for FTL code validation

4.1.2. Header File

Table 11) Jasmine firmware header file *(some files are omitted)*

Location	File	Description
./include	jasmine.h	Barefoot controller header file - Defines page/block size following NAND spec - Defines DRAM segmentation
	mem_util.h	Memory utility header file
	flash.h	Flash controller header file - Defines flash controller register set including FCP/WR/BSP - Defines Flash command code - Defines Bank interrupt flag
	peri.h	Defines Peripheral device register set - Interrupt/DRAM controller, Buffer Manager - GPIO/PMU/UART/Watch-dog/Timer/Clock, ETC.
	hi.h	Defines host interface data structure Defines SATA spec command
	ftl.h	Defines FTL metadata data structure FTL public function (include Protocol API) declaration
	sata.h	SATA related header file Defines ATA command list and SATA command structure declaration
	sata_cmd.h	ATA command related header file
./target_spw	target.h	Target device related header file - Defines memory map - Defines NAND/PLL/Timer/Clock cycle speed
	misc.h	Other useful functions (LED, Time measurement) are declared

4.1.3. Source code File

Table 12) Jasmine firmware Source file *(some files are omitted)*

Location	File	Description
./ftl_tutorial	ftl.c	Tutorial FTL source code (no GC)
./ftl_greedy	ftl.c	Greedy FTL source code (simple GC)
./ftl_dummy	ftl.c	Dummy FTL source code (*no access to NAND flash*)
./sata	sata_cmd.c	Defines ATA command function
	sata_isr.c	Defines FIQ and SATA interrupt service routine function
	sata_main.c	SATA hardware initialization and definition of firmware main function

./installer	installer.c	Defines functions for firmware installation - enable factory mode - scan bad block list - install block 0 - install FTL metadata
	installer.sln	Firmware installer solution file - Build installer using Visual C++ 2010 Express

./target_spw	initialize.c	Defines firmware initialization function - PLL/interrupt/Barefoot controller initialization
	mem_util.c	Defines memory utility function
	flash.c	Flash memory initialization and defines some FCP functions
	flash_wrapper.c	Defines LLD function - Page read/write/copy function - Block erase function
	misc.c	Defines other useful functions - LED/NAND block test/Time measurement
	init_gnu.s init_rvds.s	Firmware startup code (GNU/RVDS)
./tc	tc_synth.c	Defines Synthetic test case function

4.1.4. Miscellaneous File

Table 13) Jasmine firmware miscellaneous file *(some files are omitted)*

Location	File	Description
./	release_lock.inc	Script file which opens JTAG debug port
	README	Shows the build process and execution guidelines of the Jasmine firmware.
	HISTORY	Jasmine firmware revision log
	COPYING	Jasmine OpenSSD Platform license - GPL ver. 3
./build_rvds	armcc_opt.via armlink_opt.via	Set RVDS compile option
	file_list.via	Target file list compiled during firmware build - Includes the new file, if any exists.
	scatter.scl	Code/ZI/RW data memory address allocating scripts to compile with RVDS.
./build_gnu	ld_script	code/ZI/RW data memory address allocating scripts to compile with GNU tool

4.2. Installer Function

Installer function builds the base condition of FTL operation from Jasmine firmware installer (`./installer/installer.c`).

4.2.1. Install

```
void install(void)
```

Function:

Performs firmware installation on Jasmine board and includes the following:

- load scan list
- load firmware image
- install block #0
- install FTL metadata

NOTE: Current firmware version erases the NAND block completely when `ftl_open`, `format operation invokes`. However, there is a possibility to have *response time out error* while initializing Jasmine board in normal mode, so to avoid this, format the board in install function. Installing base condition FTL metadata for later Jasmine board booting paves a way for POR by `ftl_open` while loading metadata logged on flash memory.

4.2.2. ftl_install_mapping_table

```
void ftl_install_mapping_table(void)
```

Function:

This function logs various metadata base condition including mapping table for FTL operation to NAND flash. During firmware installation, called by install function in 'Factory mode'.

The metadata logged in this function should be loaded during FTL initialization process (`ftl_open`) while booting the Jasmine board. Therefore, the developer must modify and implement `ftl_open` function too.

4.3. FTL Protocol API

FTL protocol API initializes FTL and manages the features of handling the delivered commands by host, which SATA calls. These functions are core of this API, which makes SATA and FTL work.

This chapter will specify the FTL public functions of (`./ftl_tutorial/ftl.h`), in Tutorial FTL.

4.3.1. ftl_open

```
void ftl_open(void)
```

Function:

Performs FTL initialization. First function called by firmware main function when device turns on.

(`init_jasmine` of `./target_spw/initialize.c`) function performs following operations

- Build scan list
- FTL format
- FTL metadata initialization

4.3.2. ftl_read

```
void ftl_read(UINT32 const lba, UINT32 const num_sectors)
```

Function:

Performs FTL read request. Actual IO unit size is VPAGE size.

Parameter:

lba – Logical block address given by the host

num_sectors – Number of sectors (Unit sector size is 512B)

4.3.3. ftl_write

```
void ftl_write(UINT32 const lba, UINT32 const num_sectors)
```

Function:

Performs FTL write operation. Actual IO unit size is VPAGE size

Parameter:

lba – Logical block address given by the host

num_sectors – Number of sectors (Unit sector size is 512B)

4.3.4. ftl_flush

```
void ftl_flush(void)
```

Function:

Logs FTL metadata on flash memory. Called periodically when SATA controller is in idle state. (`ata_flush_cache`, `ata_idle`, `ata_standby` from `./sata/sata_cmd.c`)

4.3.5. ftl_isr

```
void ftl_isr(void)
```

Function:

Responsible for BSP interrupt handling feature. IRQ interrupt handler (`irq_handler`, from `./target_spw/misc.c`) calls this function if interrupt occurs during IO request to flash memory.

Checks `BSP_INTR` register and handles each of the corresponding interrupt event

4.4. LLD API

This chapter specifies the interface LLD (Low-Level device Driver) API. LLD interface manages the delivery of flash commands to flash controller to handle actual IO after FTL internal operations.

NOTE: LLD API (`./target_spw/flash_wrapper.c`) eases the implementation of FTL for FTL developers. In order to increase parallelism, developers have to implement separate LLD API or flash command through FCP register setting.

4.4.1. nand_page_read

```
void nand_page_read(UINT32 const bank, UINT32 const vblock,
UINT32 const page_num, UINT32 const buf_addr)
```

Function:

Performs certain page read operations on flash memory by using FCP command (`FC_NORMAL_READ_OUT`). RETURN_WHEN_DONE delivers the FCP command to WR.

Parameter:

bank – bank number (max: `NUM_BANKS`)

vblock – VBLK number (max: `VBLKS_PER_BANK`)

page_num – VPAGE number (max: `PAGES_PER_BLK`)

buf_addr – FTL internal buffer address loads the read page data

4.4.2. nand_page_ptread

```
void nand_page_ptread(UINT32 const bank, UINT32 const vblock,
UINT32 const page_num, UINT32 const sect_offset, UINT32 const
num_sectors, UINT32 const buf_addr, UINT32 const issue_flag)
```

Functions:

Performs certain page read operation on flash memory by using FCP command (`FC_NORMAL_READ_OUT`). This main purpose of this function are:

- When reads the logged metadata on flash memory (RETURN_WHEN_DONE)
- When reads the certain column user data to handle partial page program (RETURN_ON_ISSUE)

Parameter:

bank – Bank number (max: `NUM_BANKS`)

vblock –VBLK number (max: `VBLKS_PER_BANK`)

page_num – VPAGE number (max: `PAGES_PER_BLK`)

sect_offset – VPAGE starting sector offset (max: `SECTORS_PER_VPAGE`)

num_sectors – Number of sectors (Unit sector size is 512B)

buf_addr – FTL internal buffer address loads the read page data

issue_flag – Sets the FCP issue flag. (Refer to Table 9, from chapter 3.4.1)

4.4.3. nand_page_read_to_host

```
void nand_page_read_to_host(UINT32 const bank, UINT32 const
vblock, UINT32 const page_num)
```

Function:

Reads a certain page on flash memory by using FCP command (FC_NORMAL_READ_OUT) and delivers it to SATA host interface. After read operation, modifies SATA read buffer pointer

Parameter:

bank – Bank number (max: NUM_BANKS)

vblock – VBLK number (max: VBLKS_PER_BANK)

page_num – VPAGE number (max: PAGES_PER_BLK)

4.4.4. nand_page_ptread_to_host

```
void nand_page_ptread_to_host(UINT32 const bank, UINT32 const
vblock, UINT32 const page_num, UINT32 const sect_offset, UINT32
const num_sectors)
```

Function:

Reads few sectors (i.e., partial page read) of a certain page on flash memory by using FCP command (FC_NORMAL_READ_OUT) and delivers it to SATA host interface. After read operation, modifies SATA read buffer pointer.

Parameter:

bank – Bank number (max: NUM_BANKS)

vblock – VBLK number (max: VBLKS_PER_BANK)

page_num – VPAGE number (max: PAGES_PER_BLK)

sect_offset – VPAGE starting sector offset for partial read (max: SECTORS_PER_VPAGE)

num_sectors – Number of sectors (Unit sector size is 512B)

4.4.5. nand_page_program

```
void nand_page_program(UINT32 const bank, UINT32 const vblock,
UINT32 const page_num, UINT32 const buf_addr)
```

Function:

Logs FTL internal buffer data on certain page of flash memory by using FCP command (FC_NORMAL_IN_PROG) (Frequently used for metadata logging).

Parameter:

bank – Bank number (max: NUM_BANKS)

vblock – VBLK number (max: VBLKS_PER_BANK)

page_num – VPAGE number (max: PAGES_PER_BLK)

buf_addr – FTL internal buffer address where write page data exists and should be VPAGE-align.

4.4.6. nand_page_program_from_host

```
void nand_page_program_from_host(UINT32 const bank, UINT32 const
vblock, UINT32 const page_num)
```

Function:

This function records some sectors in the certain page of NAND Flash memory by using FCP command (FC_NORMAL_IN_PROG) (i.e., partial program)

After page writing operation, adjusts SATA write buffer.

Parameter:

bank – number of banks (max: NUM_BANKS)
vblock – number of VBLK (max: VBLKS_PER_BANK)
page_num – number of VPAGE (max: PAGES_PER_BLK)
sect_offset – start offset of VPAGE for partial (max: SECTORS_PER_VPAGE)
num_sectors – number of sectors (sector size unit: 512B)

NOTE: Check NOP (Number of Program) of NAND Flash, to do partial programming more than two times at one page.

4.4.1. nand_page_copyback

```
void nand_page_copyback(UINT32 const bank, UINT32 const
src_vblock, UINT32 const src_page, UINT32 const dst_vblock,
UINT32 const dst_page)
```

Function:

This function copies pages that are in the same bank using FCP command (FC_COPYBACK). The copying speed is quite fast because DRAM buffer is not involved.

This function writes the pages after reading the pages in FTL internal buffer because internal copy-back is impossible as in the following cases

- If NAND flash memory does not support internal copy-back operation.
- if each page is located in different planes.

Parameter:

bank – number of banks (max: NUM_BANKS)
src_vblock – original VBLK number (max: VBLKS_PER_BANK)
src_page – original VPAGE number (max: PAGES_PER_BLK)
dst_vblock – target VBLK number (max: VBLKS_PER_BANK)
dst_page – target VPAGE number (max: PAGES_PER_BLK)

4.4.2. nand_page_modified_copyback

```
void nand_page_modified_copyback(UINT32 const bank, UINT32 const
src_vblock, UINT32 const src_page, UINT32 const dst_vblock,
UINT32 const dst_page, UINT32 const sect_offset, UINT32 dma_addr,
UINT32 const dma_count)
```

Function:

This function edits data of specific pages in the same bank and copies them

Note: Page write operation should be followed by page reading in FTL internal buffer because internal copy-back is not possible as in the following cases

- If the NAND Flash Memory does not support copy-back operation
- If each page is located in different planes

If `FC_MODIFY_COPYBAC`K is not possible, modify the copy- back operation as in the following steps:

1. Read the original page and copy it to DRAM copy buffer.
2. Send some of the existing data (left hole) to object page of Flash memory using FC_NORMAL_IN command
3. Send new data to object page of Flash memory using FC_IN command
4. Send the rest of the existing data (right hole) using FC_IN_PROG command and program it with the data that is sent in step 2,3 at Flash memory.

Parameter:

bank – bank number (max: `NUM_BANKS`)
src_vblock – original VBLK number (max: `VBLKS_PER_BANK`)
src_page – original VPAGE number (max: `PAGES_PER_BLK`)
dst_vblock – target VBLK number (max: `VBLKS_PER_BANK`)
dst_page – target VPAGE number (max: `PAGES_PER_BLK`)
sect_offset – sector offset changed to new data (max: `SECTORS_PER_PAGE`)
dma_addr – buffer address containing the new data
dma_count – size of new data (Unit: Byte)

4.4.3. nand_block_erase

```
void nand_block_erase(UINT32 const bank, UINT32 const vblock)
```

Function:

Erase specific VBLK using FCP command (`FC_ERASE`)

Note: Erases 2 physical blocks in 1-plane mode and 4 physical blocks in 2-plane mode.

Parameter:

bank – bank number (max: `NUM_BANKS`)
vblock – VBLK number (max: `VBLKS_PER_BANK`)

4.5. Memory Utility API

Memory utility API allows memory communication between SRAM and DRAM. Calls these functions when read/write/search operations requires a memory address. (`./target_spw/mem_util.c`) defines the Memory utility functions

4.5.1. _mem_set_sram

```
void _mem_set_sram(void* const addr, UINT32 const val, UINT32 const num_bytes)
```

Macro – `#define mem_set_sram (ADDR, VAL, BYTES)`

Function:

Sets memory value of SRAM

Parameter:

ADDR – Memory address in SRAM field. The memory address should be 4 Byte. (Max: 0x1000_0000)

VAL – Value to set

BYTES – Byte field to set. It must be 4 Byte unit.

4.5.2. _mem_set_dram

```
void _mem_set_dram(void* const addr, UINT32 const val, UINT32 const num_bytes)
```

Macro – `#define mem_set_dram(ADDR, VAL, BYTES)`

Function:

Sets memory value of DRAM

Parameter:

ADDR – Memory address in DRAM field (min: 0x4000_0000)

VAL – Value to set

BYTES – Byte field to set. It must be multiple of `DRAM_ECC_UNIT`.

4.5.3. _mem_copy

```
void _mem_copy(void* const dst, const void* const src, UINT32 const num_bytes)
```

Macro – `#define mem_copy(DST, SRC, BYTES)`

Function:

Performs memory copy to DMA.

Note: Improves the performance, if the frequently referenced metadata in DRAM is copied to and accessed from SRAM.

Parameter:

DST – Destination memory address. It must be 4 Bytes

SRC – Source memory address. It must be 4 Bytes

BYTES – Byte field to set. It must be 4 Bytes unit (max: 32768)

(In case of DRAM-to-DRAM, it must be a multiple of DRAM_`ECC_UNIT`)

4.5.4. _mem_bmp_find_sram

```
UINT32 _mem_bmp_find_sram(const void* const bitmap, UINT32 const num_bytes, UINT32 const val)
```

Macro – `#define mem_bmp_find_sram(BMP, BYTES, VAL)`

Function:

Checks if a specific value exists in SRAM bitmap memory.

Parameter:

BMP – Bitmap memory address. It must be 4 Bytes (max: 0x1000_0000)

BYTES – Byte field to set. It must be 4Byte unit (max: UNIT * SIZE = 32768)
VAL – Value to find (0 or 1)

Returns:

On success returns the index number of the matched bitmap. On failure returns BYTES * 8.

4.5.5. _mem_bmp_find_dram

```
UINT32 _mem_bmp_find_dram(const void* const bitmap, UINT32 const num_bytes, UINT32 const val)
```

Macro – `#define mem_bmp_find_dram(BMP, BYTES, VAL)`

Function:

Checks if a specific value exists in DRAM bitmap memory field.

Parameter:

BMP – Bitmap memory address. It must be 4 Bytes (min: 0x4000_0000)

BYTES – Byte field to set. It must be 4 Byte unit (max: UNIT * SIZE = 32768)
VAL – Value to find (0 or 1)

Returns:

On success returns the index number of the matched bitmap. On failure returns BYTES * 8.

4.5.6. _mem_search_min_max

```
UINT32 _mem_search_min_max(const void* const addr, UINT32 const bytes_per_item, UINT32 const size, UINT32 const cmd)
```

Macro – `#define mem_search_min_max(ADDR, UNIT, SIZE, CMD)`

Function:

Finds the maximum value or minimum value.

Parameter:

ADDR – memory address to start searching. It must be 4Bytes.

UNIT – unit to search (e.g., 1, 2, 4 Byte)
SIZE – index range to search (max: UNIT * SIZE = 32768)
CMD – MU command code
(e.g., `MU_CMD_SEARCH_MAX_SRAM`, `MU_CMD_SEARCH_MIN_SRAM`, `MU_CMD_SEARCH_MAX_DRAM`, `MU_CMD_SEARCH_MIN_DRAM`)

Returns:

On success returns the index number of the maximum value or minimum value if exists. On failure returns the SIZE

Example code:

```
UINT32 zxcv[100] = {0};
UINT32 idx;

zxcv[4] = 0xFFFFFFFF

// search max value in array 'zxcv'
idx = mem_search_min_max(zxcv, sizeof(UINT32), 100,
                         MU_CMD_SEARCH_MAX_SRAM); // ret 4
```

4.5.7. _mem_search_equ

```
UINT32 _mem_search_equ(const void* const addr, UINT32 const
bytes_per_item, UINT32 const size, UINT32 const cmd, UINT32
const val)
```

Macro – `#define mem_search_equ(ADDR, UNIT, SIZE, CMD, VAL)`

Function:

Finds if a specific value exists in memory field.

Parameter:

ADDR – memory address to start searching. It must be 4Bytes.

UNIT – unit to search (e.g., 1, 2, 4 Byte)

SIZE – index range to search (max: UNIT * SIZE = 32768)

CMD – MU command code

(e.g., `MU_CMD_SEARCH_EQU_SRAM`, `MU_CMD_SEARCH_EQU_DRAM`)

VAL – value to find

Returns:

On success returns the index number of the matched value. On failure returns BYTES * 8

Example code:

```
UINT32 zxcv[100] = {0};
UINT32 idx;

zxcv[4] = 0xFFFFFFFF;    // Write data to buffer

// search 0xFFFFFFFF in array 'zxcv'
idx = mem_search_equ(zxcv, sizeof(UINT32), 100,
                     MU_CMD_SEARCH_EQU_SRAM, 0xFFFFFFFF); // ret 4
idx = mem_search_equ(zxcv, sizeof(UINT32), 100,
                     MU_CMD_SEARCH_EQU_SRAM, 0x80808080); // ret 100
```

Notes:

Searching a specific 4Byte value in 32KB DRAM field using DMA (Memory utility) takes about 180u (i.e., 177MB/s. However, if the SATA or NAND access the DRAM data, it would be slower)

4.5.8. _mem_cmp_sram

```
UINT32 _mem_cmp_sram(const void* const addr1, const void* const
addr2, const UINT32 num_bytes)
```

Macro – `#define mem_cmp_sram(ADDR1, ADDR2, BYTES)`

Function:

Compares the two memory fields in SRAM

Parameter:

ADDR1 – First memory address. It must be 4Bytes. (Max: 0x1000_0000)

ADDR2 – Second memory address. It must be 4Bytes. (Max: 0x1000_0000)

BYTES– Compares the range

Returns:

If same returns 0. If not returns -1 or 1

4.5.9. _mem_cmp_dram

```
UINT32 _mem_cmp_dram(const void* const addr1, const void* const
addr2, const UINT32 num_bytes)
```

Macro – `#define mem_cmp_dram(ADDR1, ADDR2, BYTES)`

Function:

Compares the two memory fields in DRAM. In this function, CPU (not the DMA) directly reads DRAM data and compares them

Parameter:

ADDR1 – Start memory address. It must be 4Bytes. (Min: 0x4000_0000)

ADDR2 – Start memory address. It must be 4Bytes. (Min: 0x4000_0000)

BYTES – Compares the range

Returns:

If same returns 0. If not returns -1 or 1

4.5.10. _read_dram_8

```
UINT8 _read_dram_8(UINT32 const addr)
```

Macro – `#define read_dram_8(ADDR)`

Function:

CPU directly reads 8 bits (1 Byte) memory from DRAM to SRAM

Parameter:

ADDR – DRAM memory address to read

Returns:

8bit memory data read from DRAM

4.5.11. _read_dram_16

```
UINT8 _read_dram_16(UINT32 const addr)
```

Macro – `#define read_dram_16(ADDR)`

Function:

CPU directly reads 16-bit (2 Byte) memory from DRAM to SRAM

Parameter:

ADDR – DRAM memory address to read

Returns:

16-bit memory data read from DRAM

4.5.12. _read_dram_32

```
UINT8 _read_dram_32(UINT32 const addr)
```

Macro – `#define read_dram_32(ADDR)`

Function:

CPU directly reads 32-bit (4 Byte) memory from DRAM to SRAM

Parameter:

ADDR – DRAM memory address to read. It must be 4Bytes

Returns:

32-bit memory data read from DRAM

4.5.13. _write_dram_8

```
void _write_dram_8(UINT32 const addr, UINT32 const val)
```

Macro – `#define write_dram_8(ADDR, VAL)`

Function:

Writes 8-bit (1 Byte) data in DRAM memory using `mem_copy` function (debug cautiously because it uses an aligned relative address)

Parameter:

ADDR – DRAM memory address to write data (min: 0x4000_0000)

VAR – 8-bit data

4.5.14. _write_dram_16

```
void _write_dram_16(UINT32 const addr, UINT32 const val)
```

Macro – `#define write_dram_16(ADDR, VAL)`

Function:

Writes 16-bit (2 Byte) data in DRAM memory using `mem_copy` function (Debug cautiously because it uses an aligned relative address).

Parameter:

ADDR – DRAM memory address to write data (min: 0x4000_0000)

VAR – 16-bit data

4.5.15. _write_dram_32

```
void _write_dram_32(UINT32 const addr, UINT32 const val)
```

Macro – `#define write_dram_32(ADDR, VAL)`

Function:

Writes 16-bit (2 Byte) data in DRAM memory using `mem_copy` function.

Parameter:

ADDR – DRAM memory address to write data. It must be 4Bytes (min: 0x4000_0000)

VAR – 32-bit data

4.5.16. _set_bit_dram

```
void _set_bit_dram(UINT32 const base_addr, UINT32 const bit_offset)
```

Macro – `#define set_bit_dram(BASE_ADDR, BIT_OFFSET)`

Function:

Sets 1bit specific offset of DRAM memory

Parameter:

BASE_ADDR – DRAM base address

BIT_OFFSET – bit offset of BASE_ADDR

4.5.17. _clr_bit_dram

```
void _clr_bit_dram(UINT32 const base_addr, UINT32 const bit_offset)
```

Macro – `#define clr_bit_dram(BASE_ADDR, BIT_OFFSET)`

Function:

Unsets (clear) 1bit specific offset of DRAM memory

Parameter:

BASE_ADDR – DRAM base address

BIT_OFFSET – bit offset of BASE_ADDR

4.5.18. _tst_bit_dram

```
BOOL32 _tst_bit_dram(UINT32 const base_addr, UINT32 const bit_offset)
```

Macro – `#define tst_bit_dram(BASE_ADDR, BIT_OFFSET)`

Function:

Tests 1-bit specific offset of DRAM memory

Parameter:

BASE_ADDR – DRAM base address

BIT_OFFSET – bit offset of BASE_ADDR

Returns:

If the test target bit_offset is 0 then returns 0. If not returns some value other than 0.