The Jasmine OpenSSD Platform

Version 1.4

Technical Reference Manual

Revision history

Date	Author	Description	Rev.
2011-04-27	임상필 (성균관대)	Initial release	1.0
2011-04-28	임상필 (성균관대)	fix typo & reflect review comments	1.1
2011-05-20	임상필 (성균관대)	Section 2.1, 2.3, 2.5 update	1.2
2011-06-01	임상필 (성균관대)	fix errata (Section 4.4 & 4.5)	1.3
2012-01-11	임상필 (성균관대)	Fix errata & Update some additional information (Section 2.5, 3.3.2)	1.4

Contents

The Jasmine OpenSSD Platform: Technical Reference Manual

Preface

About this document

본 문서는 (주)인디링스에서 개발한 Barefoot™ 컨트롤러 기반의 Jasmine OpenSSD 플랫폼의 하드웨어 및 소프트웨어에 대한 기술 정보와 SSD 소프트웨어 개발을 위한 가이드 목적으로 작성되었다. 이 문서에 포함된 주된 내용은 아래와 같다.

- Jasmine OpenSSD 플랫폼의 레퍼런스 보드인 Jasmine 보드의 특징
- Indilinx Barefoot™ SSD 컨트롤러 구조
- Jasmine OpenSSD 플랫폼의 SSD 펌웨어의 특징
- Jasmine OpenSSD 플랫폼의 SSD 소프트웨어 구성 및 API 소개

Contents

본 문서는 다음과 같은 순서로 작성되어 있다.

Chapter 1. Jasmine OpenSSD Platform Specification

이 장에서는 Jasmine 보드의 하드웨어 구조에 대해 설명한다.

Chapter 2. Indilinx Barefoot™ SSD Controller Specification

이 장에서는 Barefoot 컨트롤러의 하드웨어 아키텍처 및 내부 컨트롤러에 대해 명세 한다.

Chapter 3. Jasmine OpenSSD Platform Firmware Architecture

이 장에서는 Jasmine 보드에 탑재되는 SSD 펌웨어에 대해 명세한다.

Chapter 4. Jasmine OpenSSD Platform Software Specification

이 장에서는 Jasmine 보드와 함께 제공되는 SSD 소프트웨어의 구성과 중요 API 에 대해 명세한다.

Further reading

- ARM7 프로세서 아키텍처에 대한 자세한 설명은 www.arm.com 에서 제공하는 관련 문서를 참고하도록 한다.
- Jasmine OpenSSD 플랫폼 관련 모든 정보는 OpenSSD 프로젝트 위키 페이지를 참고하도록 한다.
  - http://www.openssd-project.org/wiki/The_OpenSSD_Project
- 좀더 기술적인 내용에 대해서는 OpenSSD 프로젝트 위키 페이지의 영문 매뉴얼을 참고하도록 한다.
  - http://www.openssd-project.org/wiki/Barefoot_Technical_Reference

Feedback

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

Chapter 1.
Jasmine OpenSSD Platform Specification

본 장에서는 Jasmine OpenSSD 플랫폼의 하드웨어 전반 사항에 대해 명세한다.

- ✓ Jasmine OpenSSD Platform overview
- ✓ Jasmine board

1.1. Overview

Jasmine OpenSSD 플랫폼은 인디링스의 고성능 Barefoot™ SSD 컨트롤러가 탑재된 레퍼런스 보드(Jasmine 보드)와 SATA 2.0 을 지원하는 SSD 펌웨어(Jasmine 펌웨어)를 포함한다.

1.2. Jasmine Board

아래 Figure 1 은 Barefoot SSD 컨트롤러가 탑재된 Jasmine 보드이다.

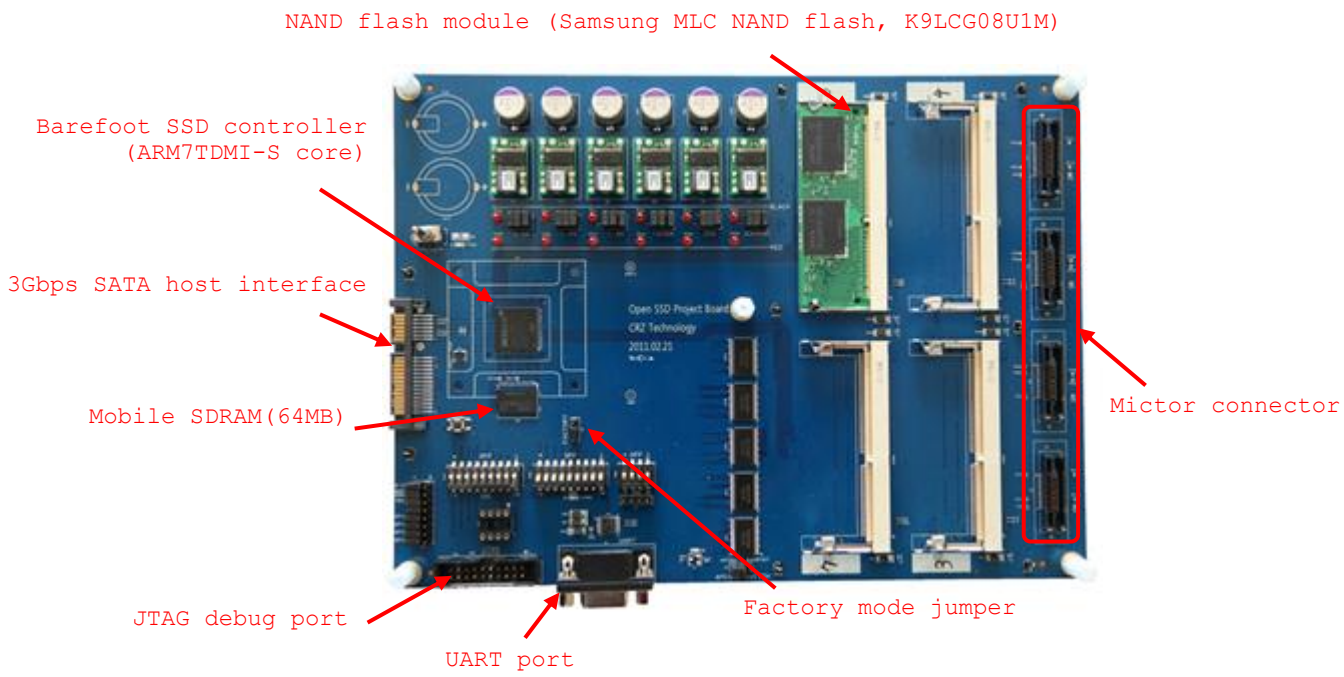

Figure 1) Jasmine OpenSSD Platform: Jasmine board

다음은 Jasmine board 의 하드웨어 Key feature 이다.

- 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

인디링스 Barefoot SSD 컨트롤러의 명세는 본 문서의 Chapter 2 를 참고할 것.

1.2.2. Factory mode jumper

Jasmine 보드는 Factory mode 점퍼를 가지고 있는데, 이 점퍼는 Barefoot 컨트롤러의 GPIO #0 핀에 연결되어 있다. 보드에 전원이 인가되면 CPU 주소 공간의 0 번지는 ROM 에 맵핑되어 있으므로 ROM code 가 실행되기 시작한다. ROM code 는 간단한 하드웨어 초기화 과정을 거치고 나서 GPIO #0 의 상태를 검사한다. 이 핀의 상태에 따라 다음과 같이 후속 동작이 달라진다.

- GPIO #0 의 값이 0 인 경우 (점퍼의 위치가 Normal 인 경우)
  ROM code 는 장착된 NAND 플래시의 위치와 개수를 파악한 뒤에, 0 번 블럭의 펌웨어 이미지를 읽어서 SRAM 에 로딩하고, address remap 동작에 의하여 CPU 주소 공간의 0 번지가 ROM 이 아닌 SRAM 이 되도록 한 뒤에 다시 0 번지로 점프한다. 이와 같은 방식으로 펌웨어가 시작된다. 만일 0 번 블럭의 내용이 실수로 인하여 지워지거나 손상된 경우, 또는 기타 이유로 인하여 0 번 블럭의 4 번 페이지에서 signature (0xC0C2E003)가 발견되지 않는 경우에는 펌웨어 로딩 과정이 중단되고 Factory mode 에 진입한다.
- GPIO #0 의 값이 1 인 경우 (점퍼의 위치가 Factory mode 인 경우)
  ROM code 는 SATA 인터페이스를 초기화하고 install.exe 로부터의 명령을 기다린다.

Chapter 2.
Indilinx Barefoot™ SSD Controller Specification

OpenSSD 플랫폼에 탑재된 인디링스 Barefoot™ 컨트롤러는 ARM 기반의 SATA 컨트롤러로써, 현재 다양한 SSD 제품 군에 탑재되고 있다. 이 장에서는 Barefoot 컨트롤러의 하드웨어 아키텍처에 대해 살펴본다. 본 장은 다음 내용을 주로 포함하고 있다.

- ✓ 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

아래 Figure 2 는 Jasmine 플랫폼의 H/W 아키텍처를 블록 다이어그램으로 나타낸 것이다.

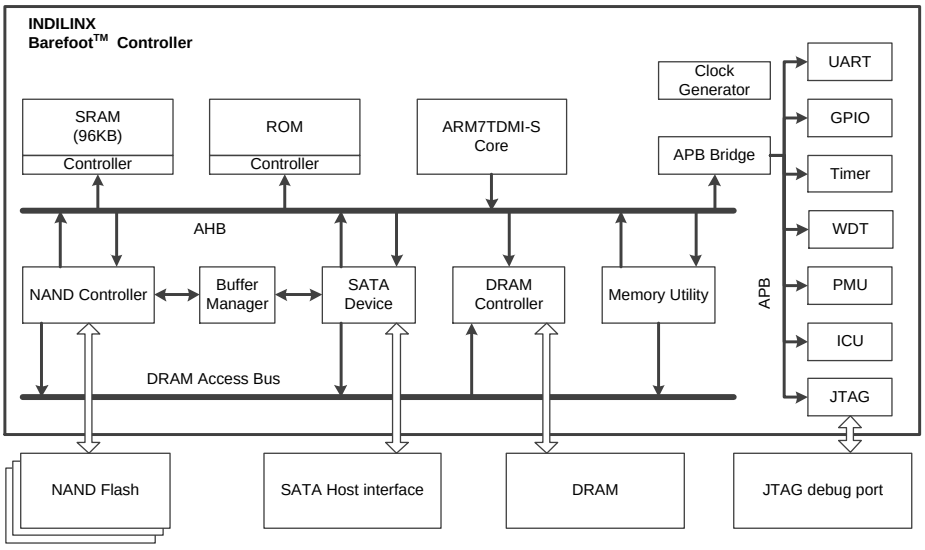

Figure 2) Jasmine OpenSSD Platform architecture

Barefoot 컨트롤러는 ARM 기반의 SATA-compatible SSD 컨트롤러로써, 많은 SSD 제품 군에 탑재되고 있다. Barefoot 컨트롤러는 ARM, Ltd 사의 16/32-bit ARM7TDMI-S RISC 마이크로 프로세서를 내장하고 있다. ARM7TDMIS-S 는 내부적으로 AMBA 버스와 ARMv4T 폰노이만 아키텍처 방식으로 구현되어 있다.

Barefoot 컨트롤러는 기본적으로 96KB SRAM, system manager(SDRAM/NAND controller), buffer manager 와 SATA 2.0 host interface, clock generator, UART, timer, WDT(Watch-dog timer), PMU(Power Management Unit), ICU(Interrupt Control Unit), JTAG 등으로 구성되어 있다. Barefoot 컨트롤러는 외부 컴포넌트인 SLC/MLC NAND 플래시, Mobile DRAM 등을 제어한다.

2.1.1. NAND flash architecture

Barefoot 컨트롤러의 NAND 플래시 아키텍처는 SSD 의 높은 대역폭을 제공하기 위해 다중 channel 및 다중 way 기반으로 설계되었다. 각 channel 과 way 에 연결된 NAND 플래시 메모리 칩(들)은 펌웨어에 의해 bank 단위로 IO 를 수행할 수 있다.

Figure 3 는 Jasmine OpenSSD 플랫폼의 NAND 플래시 아키텍처를 보여주고 있다.

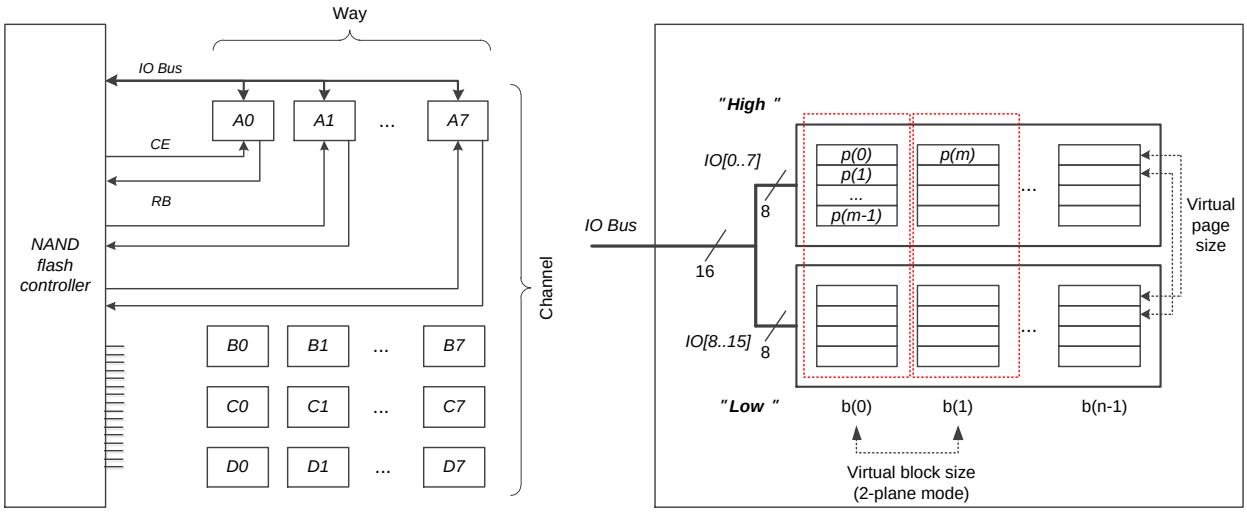

Figure 3) NAND flash architecture

Figure 3 의 왼쪽 그림처럼 Jasmine 플랫폼의 NAND 플래시 아키텍처는 완전히 병렬적이며 독립적으로 동작하는 4 channel 로 설계되어 있다. 각 channel 은 8 개의 bank 가 존재하며, 이러한 bank 는 Figure 3 의 오른쪽 그림처럼 하드웨어적으로 연결된 두 개의 NAND 플래시 칩으로 구성된다. 또한 각 bank 는 16bit IO 버스(Data bus)를 통해 동시에 두 칩에서 데이터를 입출력하며, CE(Chip Enable) 입력 핀과 R/B(Ready/Busy) 출력 핀을 포함하는 몇 개의 제어 핀(Control pin)을 통해 플래시 컨트롤러가 NAND 플래시를 제어한다.

NOTE: CE pin 은 사실상 두 개의 칩을 제어하기 위해 분리되어 있는데, 보통은 한꺼번에 enable/disable 된다 (c.f 특정 칩만을 enable/disable 하는 것이 가능한데, 이는 Table 1 의 `FO_H`, `FO_L` 옵션을 참고하기 바란다).

한편, 하나의 channel 에 연결된 8 개의 bank 는 IO 버스를 공유하기 때문에 IO 연산 자체는 여러 bank 에 대해 동시에 이루어 질 수 없다. 하지만 NAND 플래시 칩 내부 cell

operation 은 병렬적으로 가능하다. 다만, 하드웨어 설계상 플래시 컨트롤러는 하나의 channel 내 2 개의 bank 에 대해 하나의 R/B 출력 핀을 갖기 때문에 동시에 8 개의 bank 가 아닌, 최대 4 개의 bank 에서 interleaving IO 가 가능하다. 즉 channel A 의 경우, A0 와 A4, 그리고 A1/A5, A2/A6, A3/A7 bank 가 하나의 R/B 출력 핀으로 묶여있다. 따라서, A0 과 A4 bank 에 대해서 동시에 cell operation 은 불가능하다.

Figure 4 는 특정 bank 에 512B 데이터가 기록되는 과정을 보여준다. 한번에 전송되는 두 바이트 중에서 하위 바이트가 전송되는 칩을 low 라고 정의하고, 상위 바이트가 전송되는 칩을 high 라고 정의한다. 펌웨어의 관점에서 두 칩은 하나의 가상적인 칩으로 간주되며, Barefoot 컨트롤러에 의하여 이러한 가상화가 이루어진다.

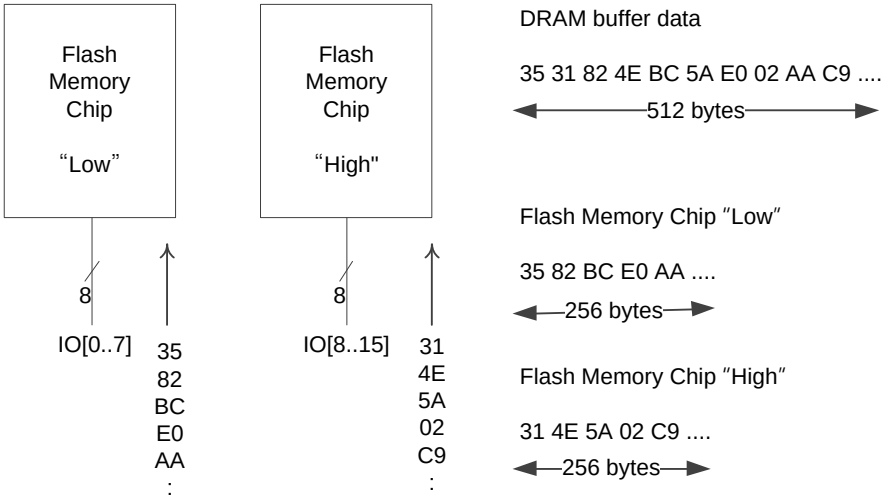

Figure 4) Bank interleaved operation

펌웨어가 하드웨어에게 보내는 읽기/쓰기 명령의 단위는 페이지인데, 하드웨어에 의하여 어느 정도의 가상화가 이루어지므로 이를 가상 페이지(Virtual Page, 또는 VPAGE)라고 정의한다. 앞서 설명한 바와 같이 low 칩의 한 페이지와 high 칩의 한 페이지가 동시에 읽히거나 쓰여지므로, 가상 페이지의 크기는 물리 페이지 크기의 2 배가 된다.

대부분의 플래시 메모리는 2-plane mode 라고 하는 기능을 지원한다. 두 페이지를 하나의 페이지처럼 묶어서 동시에 읽거나 쓰는 가속 기능으로서, 이는 펌웨어의 관점에서 보았을 때에 페이지의 크기를 두 배로 늘이고 페이지의 전체 개수를 반으로 줄이는 결과를 가져온다. (서로 다른 블록에 속하는 두 페이지가 하나로 묶이므로, 블록당 페이지 수는 변함이 없고 블록의 전체 개수가 반으로 줄어든다.) Barefoot 컨트롤러는 2-plane mode 에 참여하는 두 페이지가 하나의 페이지처럼 보이게 하는 가상화를 제공한다. 그러므로, Jasmine 플랫폼에서 1-plane mode 를 사용할 때에는 가상 페이지의 크기가 물리 페이지 크기의 2 배이고, 2-plane mode 를 사용할 때에는 가상 페이지의 크기가 물리 페이지 크기의 4 배이다.

블록 지우기 동작도 읽기/쓰기와 마찬가지로 가상화의 대상이므로, 1-plane mode 를 사용할 때에는 가상 블록의 크기가 물리 블록 크기의 2 배이고, 2-plane mode 를 사용할 때에는 가상 블록의 크기가 물리 블록 크기의 4 배이다.

한편, Jasmine 플랫폼에서 IO 병렬성을 최대화 하기 위해서는 기본적으로 bank 내 두 NAND 플래시 칩을 모두 활성화 하지만, 선택적으로 플래시 명령을 전달할 때 (`FO_L`/`FO_H`) 옵션을 활성화함으로써 플래시 컨트롤러로부터 전달되는 command/input signal 을 무시하도록 만들 수 있다. 위 Figure 4 로 예를 들면, `FO_H` 옵션을 활성화하여 플래시 명령을 전달하게 되면, High 칩이 비활성화되어 Low 칩에만 [35 82 BC ...] 데이터가 기록되게 된다. 쓰기 연산의 경우는 단순히 무시될 뿐이지만, 읽기 연산의 경우는 High 칩에서 예기치 못한 값들이 읽혀질 수 있으므로 유의하여야 한다.

하지만 ECC/CRC 엔진은 특정 칩에 선택적으로 수행되지 않기 때문에, `FO_L` 또는 `FO_H` 옵션이 활성화 되면, IO 를 수행 시 "uncorrectable data corruption" 인터럽트가 발생하게 된다. 참고로 이러한 인터럽트 발생을 생략하려면 플래시 명령 전달 시 `FO_E` 옵션을 생략하도록 한다.

NOTE: Jasmine 보드의 하드웨어 설계에 관한 좀더 자세한 설명은 OpenSSD Project 홈페이지의 Jasmine board schematic 과 NAND flash module schematic 을 참고하기 바란다.
(http://www.openssd-project.org/wiki/Downloads#Jasmine_Hardware_Schematics)

NOTE: 플래시 명령(FCP)에 관한 설명은 본 문서의 2.3.1 절을 참고하도록 한다.

2.2. Memory Map

Barefoot 컨트롤러의 메모리 맵은 아래 Figure 5 과 같다.

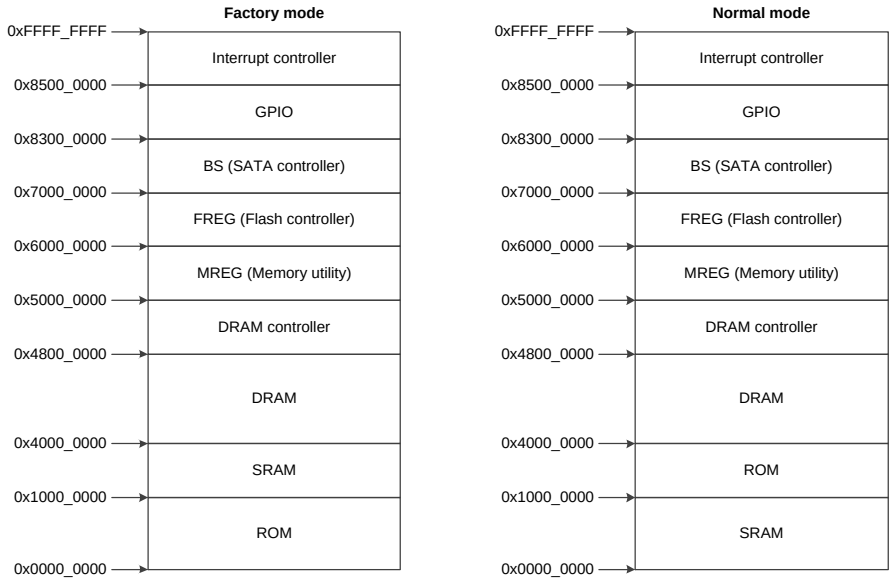

Figure 5) Memory map

먼저, ROM 에는 '*Factory mode*'로 부팅하기 위한 코드들이 저장되어 있다.

NOTE: *Factory mode* 는 Jasmine 보드에 펌웨어를 설치하기 위한 모드이다. 이 모드로 부팅하면 ROM 에 저장된 코드가 수행되며, 사용자에 의해 펌웨어 바이너리 이미지, 펌웨어 동작을 위한 메타 정보들을 플래시 메모리에 설치할 수 있다. 기본적으로 Jasmine 펌웨어는 VBLK #0 에 이러한 정보들을 기록하며, 펌웨어 설치 이후에 Jasmine 보드에 전원을 인가하면 부트로더가 VBLK #0 에 기록된 펌웨어 이미지를 SRAM 로드하고, 이후 펌웨어 동작을 수행하게 된다.

이 모드에서는 ROM 이 0x0000_0000 번지에 맵핑되어, 펌웨어 설치를 위한 ROM 에 저장된 코드가 수행되고, '*Normal(Non-factory) mode*'에서는 ROM 메모리 주소가 SRAM 메모리 주소와 서로 변경(remap)된다.

SRAM 에는 *Factory mode* 에서 0x1000_0000 번지에 맵핑되어 ROM 코드가 로드되며, *Normal mode* 에서는 부트로더 및 메인 펌웨어 이미지, 그리고 펌웨어의 ZI/RO/RW 데이터들이 로드된다.

한편, SDRAM 은 Figure 6 에서 보이는 바와 같이 버퍼 영역과 FTL 메타데이터를 저장하기 위한 영역으로 분할된다.

DRAM 버퍼는 SATA read/write 버퍼, 그리고 copy 버퍼로 나뉘며, 버퍼 내 데이터는 DMA 컨트롤러에 의해 플래시 메모리로 입출력 된다. SATA read/write 버퍼는 SATA 이벤트 큐에 전달된 사용자 요청의 데이터를 버퍼링하고, copy 버퍼는 simple copy-back 과 modified copy-back 명령 시 사용된다.

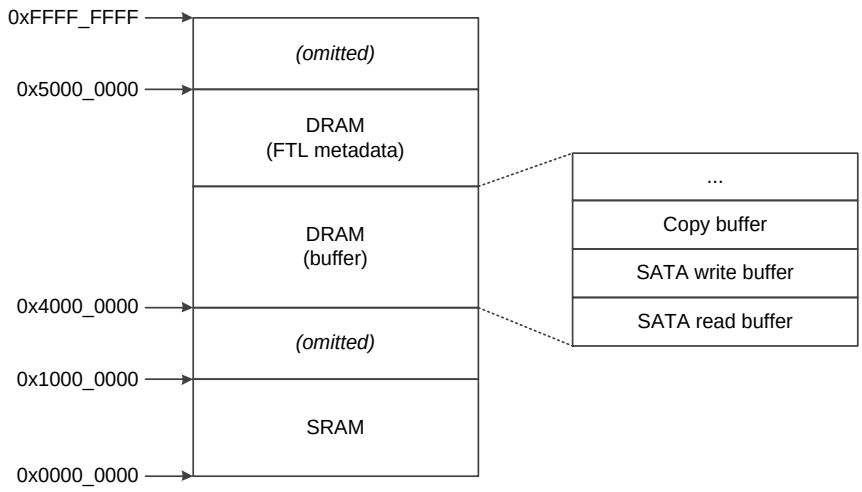

Figure 6) Memory map (DRAM segmentation)

SATA Read buffer

NAND 플래시로부터 읽혀진 사용자 데이터를 담고 있는 버퍼이다. 호스트가 요청한 데이터를 모두 읽어 이 버퍼에 올려 놓으면, SATA 컨트롤러가 호스트에 전달하

게 된다. 이 버퍼는 사용자 데이터만을 저장한다. 맵핑 테이블을 비롯한 펌웨어 내부 데이터를 읽어낼 때에는 이 버퍼 공간을 사용하지 않는다.

SATA Write buffer

FTL 로 하여금, 호스트가 전달한 사용자 데이터를 NAND 플래시에 쓰게 하기 위해, SATA 컨트롤러가 사용자 데이터를 올려 놓는 버퍼이다.

Copy buffer

`FC_CPBACK` 과 `FC_MODIFIED_CPBACK` 명령을 처리할 때 사용되는 버퍼로써, NAND 플래시메모리 칩 내 동일 플레인(plane) 에 존재하는 페이지 X 의 데이터를 페이지 Y 로 복사할 때 사용된다.

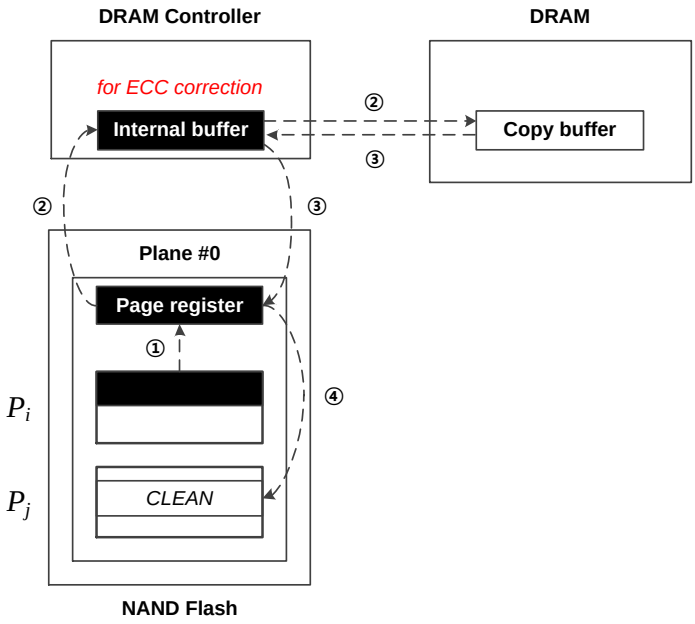

Figure 7) Copy buffer for copy-back operation

그런데 어떤 독자는, 일반적으로 페이지 복사 연산은 NAND 플래시 메모리의 primitive operation 으로 내부에서 처리할 수 있는데, 왜 별도의 Copy 버퍼로 복사연산을 수행하는지 궁금할 것이다. 이는 바로 페이지 복사 전 ECC 검증 과정이 필요하기 때문이다. 위 Figure 7 는 `FC_CPBACK` 연산 과정을 보여준다. 동일 plane 내 페이지 P_i 를 다른 블록의 P_j 로 복사하기 위해서 먼저 내부 page register 에 데이터를 로드한 후, 데이터의 신뢰성 여부를 확인하기 위해 DRAM 컨트롤러 내부 임시 버퍼와 DRAM 의 Copy 버퍼에 올린 후 ECC 검증 과정을 수행한다. 만약 ECC 정정이 필요할 경우에는 Copy 버퍼에 대해 새로운 ECC 를 생성하기 위해 과정 ③, ④의 순서로 P_j 에 복사하게 된다. 반면에 ECC 정정이 필요 없다고 판단될 경우에는 과정 ③을 수행하지 않고, 바로 과정 ④에 의해 page register 에 있는 P_i 데이터를 P_j 에 복사하게 된다.

NOTE: DRAM 공간은 위 3 가지 버퍼(SATA read/write & Copy 버퍼)를 제외한 나머지 여유 공간을 펌웨어에 의해 자유롭게 사용될 수 있다.

계속해서, 0x4800_0000 번지에는 DRAM 메모리 컨트롤러 레지스터들이 맵핑되어 있고, 0x5000_0000 번지부터는 메모리 유틸리티 레지스터 셋(MREG)을 위해 할당되어 있다. 이 영역에 설정된 레지스터 값을 통해 DMA 가 동작하게 된다.

NOTE: Barefoot 컨트롤러는 하드웨어 메모리 유틸리티를 탑재하여 DMA 에 의한 DRAM 과 SRAM 사이의 메모리 복사와 H/W 엔진을 활용한 빠른 메모리 탐색 기능 등을 지원한다.

SATA 컨트롤러 레지스터 셋(BS)은 0x7000_0000 번지에 맵핑되어 있고, 0x8300_0000 번지부터는 GPIO 핀들이 맵핑된다.

인터럽트 레지스터 셋은 0x8400_0000 번지에 맵핑되어 있으며, 인터럽트 컨트롤러는 기본적으로 SATA/Flash/DRAM 컨트롤러 및 APB(Advanced Peripheral Bus)에 연결된 컴포넌트들이 요청하는 인터럽트를 처리한다.

마지막으로, 플래시 컨트롤러 레지스터 셋(FREG)은 0x6000_0000 번지에 맵핑되어 있다. 플래시 컨트롤러는 플래시 메모리에 IO 명령을 수행하기 위해, FCP, WR, 그리고 BSP 와 같은 특별한 아키텍처를 사용하는데, 각각의 기능과 관련된 레지스터들이 이 영역에 맵핑된다. 특히, Figure 8 에서 보이는 바와 같이, Barefoot 컨트롤러는 대용량 SSD 를 지원하기 위해, 32 개 bank 를 위한 BSP 메모리 주소가 확보되어 있다. 한편, 이러한 FCP, WR & BSP 와 Figure 8 의 고유 레지스터 셋에 대해서는 다음 2.3 장에서 자세히 다루도록 한다.

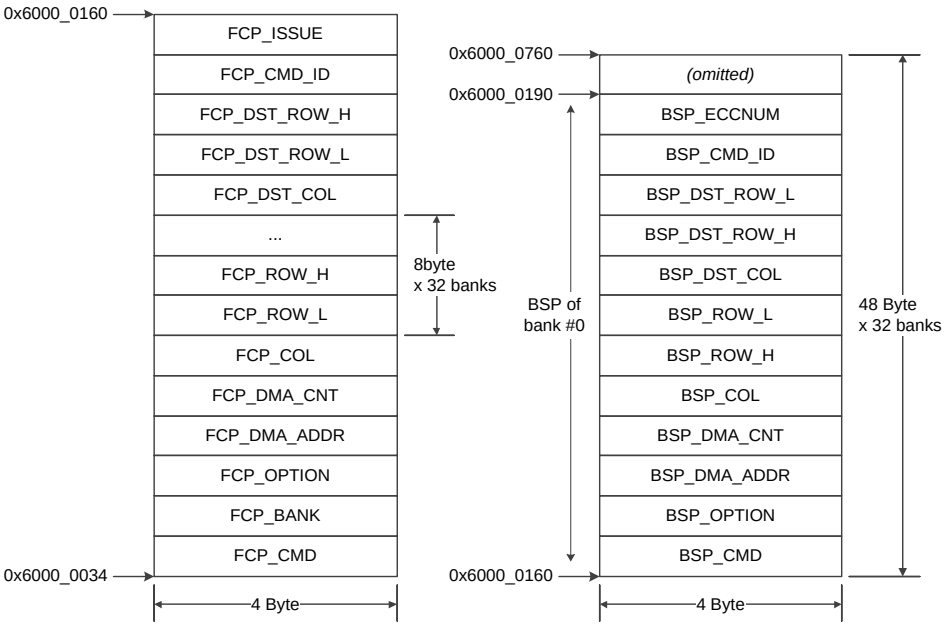

Figure 8) Memory map (FCP & BSP)

2.3. NAND Flash Controller

NAND 플래시 컨트롤러는 다중 bank 환경에서 플래시 메모리에 IO 요청을 효율적으로 전달하기 위해, FCP(Flash Command Port), WR(Waiting Room), 그리고 BSP(Bank Status Port)를 사용한다.

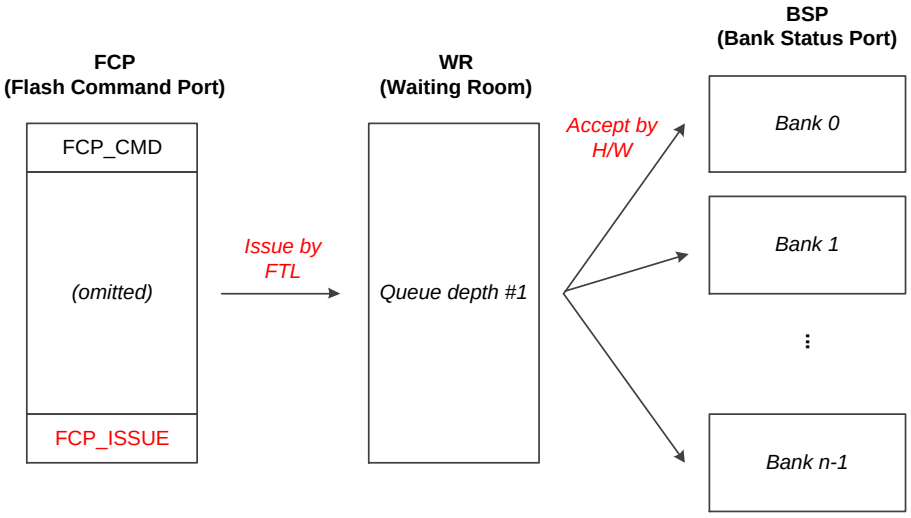

Figure 9) FCP, WR and BSP

플래시메모리에 IO 요청을 하는 과정은 다음과 같다. 우선, 플래시메모리에 요청할 페이지 읽기/쓰기 또는 블록 삭제와 같은 IO 명령을 FCP 에 설정한다. 이후, FTL 이 해당 명령을 issue 하게 되면, WR 로 전달된다. WR 에 대기하고 있는 IO 명령은 하드웨어에 의해 해당 bank 에 명령을 전달된다. 만약, 해당 bank 가 busy 상태라면, 이전 명령이 완료될 때까지, WR 에 계속 대기하게 된다. 한편, 플래시메모리에 전달된 IO 명령은 BSP 에 기록되며, 해당 명령을 수행하다 인터럽트가 발생하면 `BSP_INTR` 레지스터에 상태정보가 남게 된다.

2.3.1. FCP (Flash Command Port)

플래시 메모리에 페이지 읽기/쓰기 또는 블록 삭제와 같은 명령을 전달하기 위해서는, 먼저 FCP 를 설정해야 한다. 설정해야 할 FCP 레지스터 들은 플래시 명령, bank 번호, row/column 주소, 버퍼 주소, flag 옵션 등이 존재한다. 이러한 레지스터 셋 들을 설정하고 issue 를 수행하면, 해당 FCP 가 WR 에 전달된다.

FCP 레지스터 셋에 대한 설명은 아래 Table 1 과 같다. FCP 레지스터 셋은 FTL 이 플래시 메모리에 실제 IO 관련 요청을 할 때, 즉 플래시 메모리인터페이스가 되는 LLD(Low-Level device Driver)단에서 설정하게 된다.

Table 1) FCP register set

Register	Address	Description
FCP_CMD	0x6000_0034	NAND 플래시 명령 설정 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**: 위 플래시 명령 코드에 대한 동작순서는 ./include/flash.h 를 참고할 것.
FCP_BANK	0x6000_0038	Bank 번호 (max: NUM_BANKS) 0x3F = *auto-select mode*(AUTO_SEL) **note:** AUTO_SEL 로 설정하면, FCP command issue 후에 가장 먼저 idle 상태가 되는 bank 가 WR 의 내용을 가져감으로써, 병렬성을 높일 수 있음
FCP_OPTION	0x6000_003C	FCP option flag 설정 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**: 위 플래시 옵션 flag 에 대한 설명은 ./include/flash.h 를 참고할 것.
FCP_DMA_ADDR	0x6000_0040	DRAM to flash 또는 flash to DRAM 을 위한 버퍼 주소 **note:** 버퍼 주소는 512B 의 배수가 되어야 함
FCP_DMA_CNT	0x6000_0044	데이터 크기 (단위: Byte, 512B 의 배수여야 함
FCP_COL	0x6000_0048	시작 column 위치 (단위: Byte, FO_E 를 사용하는 경우에는 512B 의 배수여야 함, max: SECTORS_PER_PAGE - 1)
FCP_ROW_L FCP_ROW_H	0x6000_0048 0x6000_004C	전체 가상 페이지 개수 기준의 target 페이지 번호 (max: PAGES_PER_VBLK - 1) 이 레지스터는 bank 개수만큼 존재하며, H/L 은 각각 상위/하위 칩을 의미 (일반적으로 H/L 칩 모두 동일한 row 값을 설정) **note**: bank 개수만큼 존재하는 이유는 *auto-select mode* 로 동작할 때, 각 bank 의 target 페이지 위치를 지정하기 위함 **note**: copy-back 명령일 경우는 source row 가 됨
FCP_DST_COL	0x6000_0118	Destination 이 될 가상 페이지의 시작 column 위치 (NAND 플래시 칩 내부 명령인 copy-back 을 사용할 때 설정)
FCP_DST_ROW_L FCP_DST_ROW_H	0x6000_0150 0x6000_0154	Destination 이 될 가상 페이지 번호 (NAND 플래시 칩 내부 명령인 copy-back 을 사용할 때 설정)

FCP_CMD_ID	0x6000_0158	FCP command id 디버깅 용도로 사용하기 위한 레지스터로써, WR 에 전달될 때 하드웨어가 자동적으로 명령 순서 번호를 부여함
FCP_ISSUE	0x6000_015C	FCP command 를 WR 에 전달하기 위한 레지스터 이 레지스터에 임의 값을 써주면 WR 에 해당 FCP 명령이 전달 (i.e., issue) 됨

2.3.2. WR (Waiting Room)

WR 해당 명령이 플래시메모리에 전달되기 전에 잠시 대기하는 장소이며 FCP 와 동일한 정보를 갖는다. 플래시 컨트롤러는 요청된 명령이 수행될 bank 의 상태를 확인하여 유휴 상태(idle)일 때, 해당 bank 에 WR 의 내용을 전달하게 된다. 만약 목표 bank 가 busy 상태일 경우에는 WR 에서 계속 대기하게 된다. Figure 9 에서 보이는 바와 같이, WR 은 단 하나의 FCP 요청만 전달할 수 있다.

Table 2) WR register set

Register	Address	Description
WR_STAT	0x6000_002C	WR 의 상태를 관찰하기 위한 레지스터 **note**: 아래 조건은 WR 이 빈 상태를 의미 (WR_STAT & 0x0000_0001 == 0)
WR_BANK	0x6000_0030	Auto-select mode 로 FCP 를 NAND 플래시에 전달하였을 경우, 해당 명령을 가져가 bank 번호를 확인하기 위한 레지스터

NOTE: WR 에 이미 다른 FCP 명령이 전달되어 있는 상태에서 새로운 FCP 명령이 issue 될 경우, 전달된 명령 수행여부를 H/W 가 예측할 수 없다. 따라서 펌웨어는 WR_STAT 레지스터를 확인하여 반드시 WR 이 비어있는 상태일 때 새로운 플래시 명령을 전달하도록 한다.

2.3.3. BSP (Bank Status Port)

BSP 는 Figure 8 에서 보다시피 FCP 에서 WR 로 issue 된 플래시 명령정보와 WR 로 issue 되면서 추가된 정보들로 구성된다(즉, WR 에서 전달된 정보를 그대로 담고 있다). BSP 는 각 bank 마다 존재하며, 0x6000_0160 부터 48 바이트씩 총 32 bank BSP 가 0x6000_070 까지 연속되어 있다. 기본적으로 BSP 는 해당 bank 에서 마지막으로 수행된 명령에 대한 정보를 담고 있기 때문에 디버깅 목적으로 매우 용이하게 사용될 수 있다.

한편, BSP 정보 외에 플래시 내부 연산이 수행되는 과정에서 발생한 오류는 Table 3 의 인터럽트 원인 정보가 기록되는 BSP_INTR 레지스터와 해당 bank 의 현재 동작 상태 정보를 담고 있는 BSP_FSM 레지스터를 통해 확인할 수 있다.

NOTE: BSP_INTR 레지스터는 펌웨어가 clear 하기 전에는 스스로 clear 되지 않는다.

Table 3) BSP_INTR & BSP_FSM register

Register	Address	Description
BSP_INTR	0x6000_0760: 0x6000_0780	Bank 인터럽트 정보. (1Byte) Bank 마다 별도로 존재 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) 정보. (1Byte) BSP_INTR 레지스터와 마찬가지로 bank 마다 별도로 존재 0x0 = idle others = no idle

2.4. SATA Controller

Barefoot 컨트롤러 내부에는 호스트와 장치 사이의 데이터 전송을 담당하는 SATA 컨트롤러가 존재한다. SATA 컨트롤러는 NCQ(Native Command Queuing)과 별도의 명령 큐(이벤트 큐)를 관리하여 FTL 에 효율적으로 IO 를 전달할 수 있도록 한다.

NOTE: SATA Operation 에 관한 좀더 기술적인 내용은 OpenSSD 커뮤니티의 영문 메뉴얼을 참고하기 바란다.

2.4.1. SATA protocol

호스트가 SSD 에 IO 요청을 보내면, SATA 프로토콜에 의해 해당 명령이 접수되어 자동으로 호스트에 응답을 전송한다. 만약 쓰기 명령일 경우, 펌웨어가 데이터 전송 시작을 요청하면, 해당 명령에 대한 모든 전송이 완료될 때까지의 과정이 하드웨어에 의해 자동으로 처리된다.

한편 IO 수행 도중에 예외 상황이 발생하지 않았다면, 해당 명령에 대한 최종 응답을 하드웨어가 자동으로 전송한다.

2.4.2. SATA NCQ

SATA NCQ 는 동시에 한 드라이브 내에서 여러 개의 명령을 연속적으로 받아드릴 수 있는 SATA 용 명령 프로토콜이다.

Barefoot 컨트롤러의 NCQ 는 SATA 2.0 에 따라, 32 개의 호스트 명령을 수용할 수 있으며 FIFO 방식으로 동작하도록 설계되었다. 한편 NCQ 에 전달된 명령들은 아직 SATA 데이터 전송이 이루어지지 않은 명령들이다. 즉, 호스트 입장에서는 아직 완료되지 않은 명령들이 NCQ 에 대기하게 된다. 이러한 명령들은 이후 SATA 이벤트 큐로 이동되어 FTL 에 전달된다.

2.4.3. SATA event queue

SATA 이벤트 큐는 호스트 관점에서 이미 SATA 데이터 전송이 완료된 명령들을 큐잉하는 역할을 담당한다. 이러한 명령들은 실제 IO 가 수행되지 않아 DRAM 에서 NAND 플래시로의 전송을 기다리고 있는 상태이다.

아래 Figure 10 처럼 SATA 이벤트 큐는 최대 128 개의 명령을 수용하며, SATA NCQ 와 마찬가지로 FIFO 방식으로 동작한다. 호스트 명령은 FIQ 인터럽트에 의해 전달되며, FTL 이 해당 명령을 가져갈 때마다 제거된다.

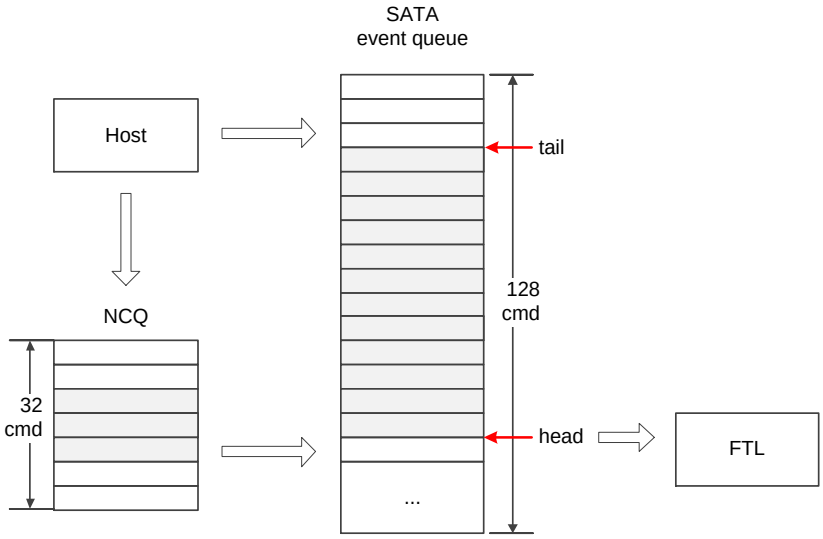

Figure 10) SATA NCQ & event queue

한편, SATA 이벤트 큐를 FIFO 방식으로 처리하는 경우에는 아래와 같이 몇 가지 문제점이 발생할 수 있다.

1. Read 명령에 대한 호스트 대기시간이 길어질 수 있다.
2. Read 시 DRAM 에서 호스트로 데이터 전송이 진행되는 동안, FTL 및 NAND 플래시 메모리가 유휴(idle) 상태에 있을 가능성이 높아, 성능 측면에서 불리할 수 있다.

이러한 문제점을 해결하기 위해 SATA 이벤트 큐는 읽기 명령을 쓰기 명령보다 우선해서 처리하도록 설계되었다. 하지만 이 방법은 특정 주소에 대해 "*read-after-write*" 연산에 대해서 data coherence 문제가 발생할 가능성이 있기 때문에 (예를 들어, `<WRITE, lsn=3>` 명령이 SATA 이벤트 큐에 쌓인 상태에서 `<READ, lsn=3>` 명령을 FTL 이 먼저 처리할 경우, old 데이터를 NAND 로부터 꺼내게 됨), 하드웨어에 의한 history log search 를 통해 이러한 data coherence 를 보장하게 된다. 뿐만 아니라, 섹터가 겹치는 경우에 대해서도 하드웨어로 해결한다.

2.5. DRAM Host Buffer & Buffer Manager

DRAM 호스트 버퍼는 SATA 이벤트 큐의 IO 요청에 대한 실제 사용자 데이터를 버퍼링하기 위해 사용되며, 이는 호스트 읽기 요청에 의해 플래시 메모리로부터 전달된 데이터를 버퍼링하기 위한 'SATA read 버퍼'와 쓰기 데이터를 버퍼링하기 위한 'SATA write 버퍼'

로 나뉘어져 관리된다. 이러한 버퍼는 기본적으로 원형 버퍼(circular buffer)의 형태로 동작하며, DRAM 호스트 버퍼와 SATA 컨트롤러의 flow control 은 하드웨어 버퍼 매니저에 의해 이루어진다.

2.5.1. SATA read/write buffer

SATA read/write 버퍼의 기본 버퍼 프레임 단위는 VPAGE 크기(e.g., 4~32KB)이다. 각 버퍼 공간의 크기는 DRAM 내 FTL 메타데이터 크기에 따라 좌우되는데, 보통 SATA read 버퍼를 약 1MB, 그리고 SATA write 버퍼를 수십 MB 정도 사용하게 된다.

NOTE: DRAM 호스트 버퍼 공간은 FTL 메타데이터 크기에 의해 결정되므로, FTL 헤더 파일에 (`ftl.h`)에 작성되어 있다.

아래는 SATA read/write 버퍼와 관련된 레지스터이다.

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	SATA write/read buffer 의 base address
`SATA_WBUF_SIZE` `SATA_RBUF_SIZE`	0x7000_0178 0x7000_017C	SATA write/read buffer frame 의 개수
`SATA_RESET_WBUF_PTR` `SATA_RESET_RBUF_PTR`	0x7000_0184 0x7000_0188	SATA write/read buffer frame 포인터 reset 레지스터 펌웨어가 직접 buffer management 를 하기 위해서 사용될 수 있음
`SATA_WBUF_PTR` `SATA_RBUF_PTR`	0x7000_0194 0x7000_0198	SATA write/read buffer 포인터 (id #)
`SATA_WBUF_FREE`	0x7000_019C	SATA write buffer 의 free buffer frame 개수
`SATA_RBUF_PENDING`	0x7000_01A0	SATA read buffer 의 pending buffer frame 개수

한편, SATA read/write 버퍼의 실질적인 관리는 기본적으로 SATA, Buffer Manager, NAND controller 간의 signal 전달로 이루어지게 되는데, 원할 경우 펌웨어가 직접적으로 관여하여 이를 조정할 수 있다.

SATA read/write 버퍼의 관리에 대한 설명은 다음 2.5.2 장을 참고하기 바란다.

2.5.2. Buffer management

DRAM 호스트 버퍼는 SATA, 하드웨어 버퍼 매니저, 그리고 FTL 간의 포인터(i.e., `sata_xxx_ptr`, `bm_xxx_limit`, `ftl_xxx_ptr`) 조정에 의해 관리된다. 한편, 앞 절에서 설명한 대로, DRAM 호스트 버퍼는 원형 버퍼로 동작되기 때문에 각각의 버퍼 프레임 포인터는 항상 오름차순으로 증가하도록 관리되어야 한다.

Figure 11 은 DRAM 호스트 버퍼의 관리 정책을 그림으로 표현한 것이다.

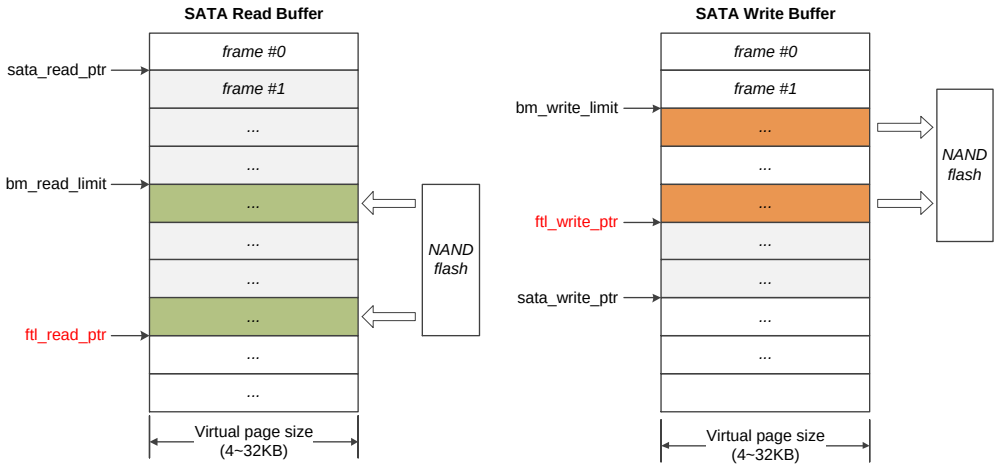

Figure 11) DRAM host buffer

아래는 SATA read 버퍼 관리 순서이다.

1. 먼저, 현재 `ftl_read_ptr` 이 가리키는 버퍼 프레임을 주소로 하여 플래시 컨트롤러가 읽기 명령을 NAND 플래시에 전달하면, DMA 는 NAND 플래시의 타겟 페이지에서 해당 버퍼에 데이터를 로드한다.
2. 펌웨어는 `ftl_read_ptr` 을 증가시키고, 다음 IO 요청을 처리한다.
3. SATA 컨트롤러는 FTL 이 로드한 사용자 데이터를 `sata_read_ptr` 을 증가시키면서 순차적으로 호스트에 전달한다.

한편, 플래시 메모리와 SATA 간의 bandwidth gap 으로 인해 `sata_read_ptr` 가 `ftl_read_ptr` 을 추월하여, 잘못된 데이터를 호스트에 전달하는 문제가 발생할 수 있다. 이러한 문제가 발생되는 것을 방지하기 위해 버퍼 매니저가 관리하는 포인터인, `bm_read_limit` 을 두어, 항상 NAND 플래시에서 SATA read 버퍼로 데이터 전송이 완료된 데이터만을 호스트에 전달하는 것을 보장한다. 즉, 플래시 컨트롤러는 특정 bank 의 읽기 연산이 완료되었다는 사실을 버퍼매니저에 통보하면, 버퍼매니저는 `bm_read_limit` 포인터를 증가시킴으로써 SATA 컨트롤러로 하여금 호스트에 읽기 데이터를 전송할 수 있도록 한다.

NOTE: 추가적으로 FTL 이 읽기 명령을 플래시 컨트롤러에 전달하는 속도가 너무 빠르면, `ftl_read_ptr` 이 `sata_read_ptr` 을 추월할 수 있게 된다. 이는 아직 호스트에 전달되지 않은 데이터에 새로운 읽기 데이터를 덮어써버릴 수 있는 가능성을 의미하기 때문에, `ftl_read_ptr` 이 `sata_read_ptr` 을 추월하지 않도록 펌웨어 단에서 이를 제어해야 한다 (c.f. `ftl_read() in ./ftl_tutorial/ftl.c`).

계속해서 SATA write 버퍼 동작의 경우도 SATA read 버퍼와 유사한 방식으로 관리된다.

1. SATA 컨트롤러는 sata_write_ptr 을 증가시키면서 event queue 에 큐잉된 쓰기 요청에 대한 호스트 데이터를 SATA write 버퍼에 전달하여, DMA 가 NAND 플래시에 데이터를 전달할 수 있도록 한다.
2. FTL 은 event queue 에서 쓰기 요청을 가져와 플래시 컨트롤러에 쓰기 명령을 전달하고, ftl_write_ptr 을 증가시킨다.
3. SATA 컨트롤러는 쓰려고 하는 데이터가 SATA write 버퍼에 전달이 완료될 때, 버퍼 매니저에게 이 사실을 통보하고, 버퍼매니저는 플래시 컨트롤러가 NAND 플래시에 호스트 데이터를 기록할 수 있도록 한다.

하지만 플래시 메모리의 비교적 느린 프로그래밍 속도(*tPROG*)로 인해, 미처 NAND 플래시에 기록되지 않은 사용자 데이터를 SATA 가 다른 쓰기 요청에 대한 새로운 데이터로 덮어써버릴 수 있다. 따라서 이러한 문제를 해결하기 위해 SATA write 버퍼에도 하드웨어 버퍼 매니저가 관리하는 bm_write_limit 포인터를 두어, sata_write_ptr 가 bm_write_limit 을 추월하는 현상을 방지하여 호스트 쓰기 연산의 정상 동작을 보장한다.

아래 Table 5 는 Buffer Manager 의 레지스터 셋을 보여주고 있다. 기본적으로 Buffer Manager 의 flow control 은 NAND controller 와의 상호작용에 의해 이루어지지만, 펌웨어가 직접 flow control 을 해야 할 필요가 있을 경우에는 이 레지스터를 활용하면 된다.

Table 5) Buffer Manager register

Register	Address	Description
BM_WRITE_LIMIT BM_READ_LIMIT	0x5000_0000: 0x5000_0004	Buffer Manager 의 SATA read/write limit pointer (*READ-ONLY*) 이 레지스터 값을 통해 실질적인 flow control 이 이루어짐 (실제 사용 예는 ./ftl_dummy/ftl.c 참고)
BM_STACK_RESET	0x5000_0008	BM read/write limit pointer 를 reset 0x01 = reset BM write limit to BM_SATA_WRSET 0x02 = reset BM read limit to BM_SATA_RDSET
BM_STACK_WRSET	0x5000_0028	BM write limit 을 reset 하기 위한 레지스터. 이 레지스터에 SATA write buffer id 를 기록함
BM_STACK_RDSET	0x5000_002C	BM read limit 을 reset 위한 레지스터 이 레지스터에 SATA read buffer id 를 기록함

2.6. Memory Utility

Memory utility 는 기본적으로 SRAM 과 DRAM 간의 데이터 전송을 담당한다. 그리고 메모리 초기화(e.g., mem_set)과 같은 반복적인 메모리 작업이나 특정 메모리 영역의 탐색 연산을 빠르게 하는 기능도 수행한다.

한편 Barefoot 컨트롤러는 DRAM 데이터의 신뢰성을 높이기 위해 별도의 하드웨어 ECC 엔진을 사용하며, DRAM ECC 정보는 DRAM_ECC_UNIT(128 Byte)당 4 Byte 가 생성된다. 다음은 SRAM 데이터가 DRAM 에 기록되는 과정을 보여준다.

1. DRAM 에서 한번에 132B(128 Byte 데이터+ 4 Byte ECC 패리티 정보)씩 읽어서 Barefoot 컨트롤러의 내부 132 Byte 임시 메모리에 저장

2. 위 1 번 과정에서 ECC 정정이 필요한 경우, 임시 메모리를 대상으로 데이터 정정 연산 수행 (c.f. 사실상 DRAM 은 고신뢰성 메모리이기 때문에 ECC 정정 연산 발생 확률이 매우 낮음)
3. SRAM 에서 원하는 데이터를 읽어 임시 메모리의 내용 수정
4. 새로운 128 Byte 에 대해 새로운 ECC 패리티를 생성하고, 총 132 Byte 를 DRAM 에 기록

NOTE: CPU 가 DRAM 데이터를 직접 변경하게 되면 memory utility 에 의한 ECC 정보가 손실되거나 무의미해질 수 있기 때문에, SRAM 과 DRAM 사이의 데이터 전송은 반드시 memory utility(`./include/mem_util.h`)를 사용함으로써 이루어져야 한다.

Memory utility 관련 레지스터 셋은 아래 Table 6 과 같으며, memory utility 동작 과정에서 발생하는 인터럽트는 SDRAM 컨트롤러의 `SDRAM_INTSTATUS` 레지스터를 확인해 봄으로써 발생 원인을 알아낼 수 있다.

Table 6) Memory utility register set

Register	Address	Description
`MU_SRC_ADDR`	0x5000_0010	복사할(source) 메모리 주소. DRAM 데이터를 읽을 때 설정
`MU_DST_ADDR`	0x5000_0014	복사될(destination) 메모리 주소. DRAM 에 데이터를 쓸 때 설정
`MU_VALUE`	0x5000_0018	기록할 새 데이터
`MU_SIZE`	0x5000_001C	설정 또는 탐색할 메모리 영역의 크기 **note**: `mem_search` 의 경우, max 32768 Byte 영역
`MU_RESULT`	0x5000_0020	메모리 연산 결과값 0xFFFFFFFF = 현재 메모리 연산 수행 중 을 의미
`MU_CMD`	0x5000_0024	Memory utility 명령 코드

`MU_UNITSTEP`	0x5000_0030	반복적인 메모리 작업 시, step 단위 설정
		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	DRAM 컨트롤러가 발생시킨 인터럽트 상태 정보
		`0x01 = ECC fail  0x02 = ECC correction`
		`0x04 = Address Overflow  0x08 = Deadlock`

2.7. Interrupt Controller

인터럽트 컨트롤러는 내부 주변 장치인 SATA, 플래시 메모리, DRAM, UART, Timer, WDT 와 외부 주변 장치에 의한 인터럽트 처리 요구를 수신하고, CPU 로 하여금 각각의 주변장치에서 발생한 인터럽트를 처리하게 한다.

아래 Table 8 는 인터럽트 컨트롤러의 레지스터 셋을 보여준다. 펌웨어는 부팅 시, 감지하고자 할 주변장치를 APB_ICU_CON 레지스터에 설정해야 한다. 그리고 런타임에 하드웨어 인터럽트가 발생하면, APB_INT_STS 레지스터를 확인하여 어떠한 주변장치에서 발생한 인터럽트인지 알아낸 후 적합한 인터럽트 핸들링 연산을 수행해야 한다.

Table 8) Memory utility register set

Register	Address	Description
APB_ICU_CON	0x8500_0000	FIQ 인터럽트 설정
APB_INT_STS	0x8500_0004	인터럽트 상태 정보. 펌웨어는 이 레지스터를 통해 인터럽트를 발생시킨 주변장치를 확인할 수 있음 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 인터럽트 설정 **note**: 설정할 경우, 해당 주변장치의 인터럽트 허용
APB_PRI_SET1	0x8500_0054	-
APB_PRI_SET2	0x8500_0058	-

Chapter 3.
Jasmine OpenSSD Platform Firmware Architecture

본 장에서는 Jasmine OpenSSD 플랫폼의 SSD 펌웨어의 내부 구조에 대해 명세한다.

3.1. Firmware Overview

Jasmine OpenSSD 플랫폼의 메인 펌웨어는 크게 HIL(Host Interface Layer), FTL(Flash Translation Layer), FIL(Flash Interface Layer)로 구조화되어 있다.

HIL 은 주로 SATA 호스트 명령과 버퍼 관리를 담당하는 계층으로써 호스트로부터 SATA 컨트롤러에 IO 요청이 전달되면, 해당 명령을 SATA 이벤트 큐에 삽입하고 이후 FTL 이 순차적으로 해당 요청을 처리하도록 돕는다.

다음으로 FTL 은 플래시 메모리를 하드디스크와 같은 블록 디바이스로 인식할 수 있도록 하는 소프트웨어 계층이다. FTL 은 일반적으로 1)주소 맵핑 기능, 2) 가비지 콜렉션, 3) 마모 균등화 기능 등을 담당하며, IO 요청의 실체 처리는 FTL 이 플래시 메모리에 해당 연산을 전달함으로써 이루어진다. FTL 은 성능 및 안정성을 높이기 위해 다양한 기법들이 존재하는데, Jasmine 펌웨어에는 Tutorial FTL, Greedy FTL, 그리고 Dummy FTL 이 구현되어 있다.

FIL 계층은 플래시 메모리를 담당하며, FTL 로부터 전달된 플래시 명령의 실제 동작은 LLD(Low-level device driver)에 의해 수행되며, 정상동작 과정에서 발생한 예외 상황(e.g., runtime bad block, data corruption)등은 인터럽트 컨트롤러에 의해 탐지된 후, FTL 이 핸들링한다.

3.2. Host Interface Layer

3.2.1. Hardware event queue

Jasmine 펌웨어에서 SATA 이벤트 큐(2.4.3 절 참고)는 하드웨어 이벤트 큐로 구현되어 있다. SATA 인터페이스로 전달되는 read/write ATA 커맨드는 하드웨어 이벤트 큐에 의해 관리되고, HIL(Host Interface Layer)의 메인 함수에서 FTL 에 전달되어 처리된다. 아래는 ATA 커맨드가 이벤트 큐에 의해 처리되는 과정을 보여준다.

1. 호스트에서 Jasmine 보드로 ATA 커맨드를 전달
2. SATA 호스트 인터페이스 에서 FIQ 인터럽트 발생 및 FIQ 핸들러 호출
3. FIS(Frame Information Structure)에서 해당 명령을 읽음. 요청 명령의 `cmd_type`, `lba`, `sector_count` 추출
4. 읽기/쓰기 연산(CCL_FTL_D2H/CCL_FTL_H2D)인 경우, 하드웨어 이벤트 큐에 해당 명령 추가 (`./sata/sata_isr.c` 의 `handle_got_cfis()`)
5. 읽기/쓰기 외의 명령(e.g. TRIM 과 같은 slow command)의 경우, `g_sata_context.slow_cmd` 변수에 저장

6. 펌웨어 메인 함수(`./sata/sata_main.c` 의 `Main()`)에서 명령 처리
   - 이벤트 큐에 읽기/쓰기 요청이 있을 경우, 하나씩 꺼내 우선적으로 처리 (`./sata/sata_main.c` 의 `eventq_get()`)
   - 만약 read/write 요청이 없을 경우, `slow_cmd` 변수에 저장된 명령 처리

3.3. Flash Translation Layer

FTL(Flash Translation Layer)은 호스트가 플래시 메모리를 하드디스크처럼 보일 수 있도록 하는 소프트웨어 계층이다. FTL 은 하드웨어 이벤트 큐에 삽입된 읽기/쓰기 요청을 가져와 순차적으로 처리하며, 플래시 컨트롤러에 IO 명령을 전달한다.

3.3.1. FTL protocol interface

FTL 프로토콜 인터페이스는 SATA 호스트 인터페이스와 메시지를 주고받기 위한 함수들을 의미한다. 본 절에서는 4 가지 핵심 프로토콜 인터페이스(`ftl_open`, `ftl_read/ftl_write`, `ftl_flush`) 함수에 대해 어떠한 기능들이 구현되어야 하는지를 중심으로 설명하도록 한다.

NOTE: FTL protocol API 의 함수 명세는 본 문서의 4.2 절을 참고할 것

---

ftl_open

이 함수는 Jasmine 보드의 초기화 과정을 마친 후, 호스트 IO 요청을 받아들이기 위해 FTL 을 초기화하는 기능들을 수행해야 한다.

1. 펌웨어 인스톨 시, 작성된 VBLK #0 에 기록된 scan list 를 읽어 초기 bad block 을 확인한다.
2. NAND 플래시 메모리를 호스트 IO 요청 처리를 위한 초기 상태로 만든다. 사용자 데이터를 기록하기 위해 전체 블록을 삭제(format)해주는 연산이 포함될 수 있다.
3. 펌웨어 인스톨 시 플래시 메모리에 설치한 FTL 메타데이터 또는 이전 power-off 시점에 기록한 메타데이터 정보를 SRAM 또는 DRAM 에 로드한다.
4. volatile 변수를 포함한 FTL 이 관리하는 기타 메타데이터 정보를 초기화 한다.
5. 플래시 컨트롤러의 인터럽트 옵션 등을 설정한다.

ftl_read

Event queue 로부터 전달된 호스트 읽기 요청을 처리하기 위한 API 이다. 맵핑사상 정보를 통해, virtual page 단위로 valid 페이지가 속한 플래시 메모리 칩에 읽기 명령을 전달한다. 그리고 FTL 이 관리하는 SATA read 버퍼 포인터를 조정한다.

ftl_write

Event queue 로부터 전달된 호스트 쓰기 요청을 처리하기 위한 API 이다. 새로운 데이터를 쓰기 위한 여유 페이지(free pace)를 찾고 새로운 위치에 데이터를 쓰면 기존 맵핑 사상정보를 새 페이지 정보로 변경하도록 한다.

ftl_flush

POR/SPOR 을 위해 반드시 보존되어야 할 FTL 메타데이터 정보를 NAND 플래시 메모리에 기록하는 API 이다. 이 API 는 SATA idle/standby 시 주기적으로 호출되어 FTL 메타데이터의 일관성 및 POR 을 가능하게 한다.

3.3.2. Tutorial FTL

Tutorial FTL 은 Jasmine 플랫폼 아키텍쳐 상에서 FTL 의 동작 원리를 간단하게 이해시키기 위한 목적으로 구현되었다. 이 FTL 은 페이지 기반의 주소 사상 기법을 사용하며, FTL 의 핵심 기능인 가비지 콜렉션/마모 균등화/POR 등을 포함하지 않는 매우 간단한 구조로 구현되어 있다. 본 절에서는 Tutorial FTL 의 DRAM 활용과 읽기/쓰기 요청을 어떻게 처리하는 지를 중심으로 설명하도록 한다.

DRAM usage

아래 Figure 12 은 Tutorial FTL 의 DRAM 사용을 보여준다.

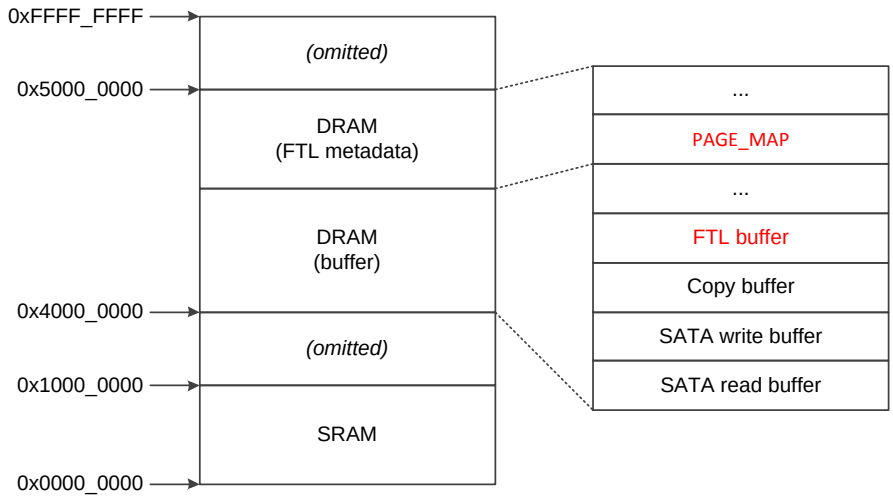

Figure 12) DRAM usage of Tutorial FTL

Dummy FTL 은 DRAM 버퍼 공간에 별도의 'FTL buffer'를 할당하여 사용한다. 이러한 FTL 버퍼는 변경된 메타데이터를 플래시 메모리에 기록할 때 또는 이미 기록된 메타 데이터를 플래시 메모리로부터 읽어올 때 임시 버퍼 용도로 사용한다.

또한, DRAM FTL 메타데이터 공간에는 페이지 사상 테이블(`PAGE_MAP`)을 관리한다. `PAGE_MAP` 은 페이지 주소에 대해 물리 페이지 주소 맵핑을 담당하는 페이지 맵핑 테이블이다.

NAND Structure

Tutorial FTL 은 아래 Figure 13 와 같은 구조로 플래시 메모리를 관리한다. VBLK #0 는 펌웨어 설치 시 펌웨어 인스톨러 함수(`./installer/install.c` 의 `install()`)에 의해 작성된다. 이 블록에는 펌웨어가 FTL 을 동작시키기 위한 정보(i.e., bad blk scan list, firmware binary image 등)들이 저장된다. Tutorial FTL 은 VBLK #0 을 제외한 나머지 영역을 사용자 데이터를 순차적으로 기록하는 용도로 사용한다.

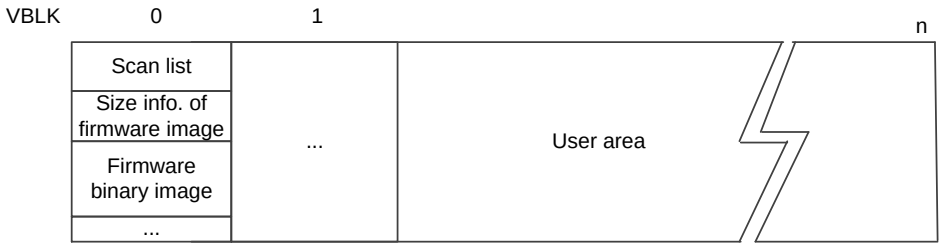

Figure 13) NAND structure of Tutorial FTL

Address mapping

FTL 은 일반적으로 호스트로부터 전달된 <lsn, sector_size> 정보를 통해, VPAGE 단위로 IO 를 수행한다. Tutorial FTL 은 페이지 단위로 주소 맵핑 테이블을 관리한다. 즉, 플래시 메모리 상의 사용자 데이터를 별도의 탐색과정 없이 직접 사상(direct mapping)으로 접근하게 된다.

아래 Figure 14 는 Tutorial FTL 의 주소 맵핑 과정을 보여준다. 먼저, FTL 은 SATA event queue 에서 LPN 2 에 대한 읽기 요청을 가져온다. 그리고 페이지 맵핑 테이블(i.e., PAGE_MAP)을 통해 LPN 에 대한 사상 정보를 얻는다. 이 정보에는 해당 LPN 이 기록된 bank 번호와 VPN 정보가 함께 기록되어 있다. 이후 FTL 은 플래시 IO 명령을 FIL 에 전달하여 해당 PPN 에 대한 사용자 데이터를 호스트에 전달한다.

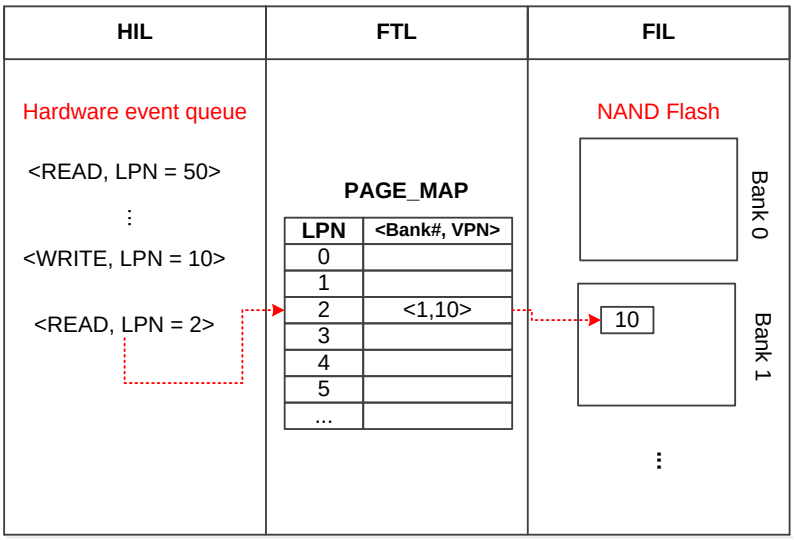

Figure 14) Address mapping in Tutorial FTL

Read operation

읽기 연산은 아래 Figure 15 과 같은 순서로 진행된다. Tutorial FTL 은 위 0 절에서 설명한 방식으로 주소 맵핑 과정을 거쳐 읽고자 하는 사용자 데이터의 물리적 위치를 알아낸다. 그리고 과정 ①처럼 플래시 컨트롤러에 읽기 명령을 전달하여, SATA read 버퍼에 사용자 데이터를 전달한다. 이후 하드웨어가 사용자 데이터 전송이 마쳤다는 사실을 버퍼 매니

저에 알려주면, 버퍼 매니저는 과정 ②와 같이 DRAM 버퍼의 데이터를 SATA 호스트로 전달한다.

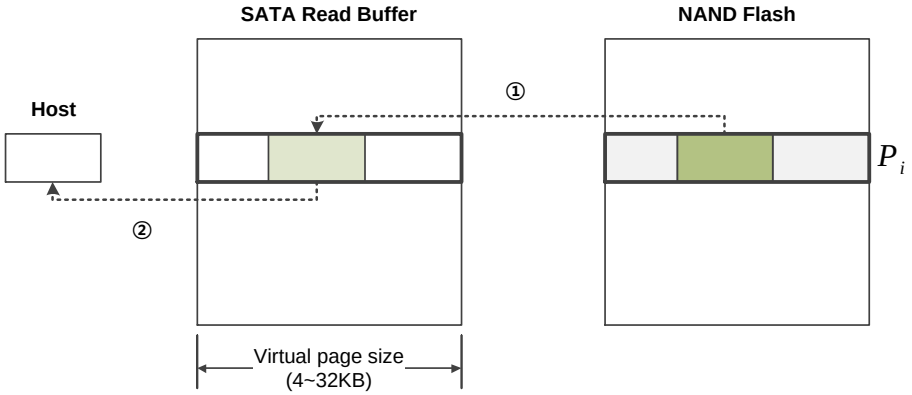

Figure 15) Read operation in Tutorial FTL

Write operation

쓰기 연산은 쓰기 단위에 따라 별도의 읽기 연산이 추가로 발생한다 즉, VPAGE 크기보다 작은 단위로 IO 를 수행해야 하면, 플래시 메모리 상에 기록된 기존 데이터를 임시 버퍼에 읽어와 병합한 후 쓰기 연산을 수행한다.

아래 Figure 16 는 VPAGE 크기 보다 작은 단위의 쓰기 요청에 대한 연산과정을 보여준다. 먼저, 읽기 연산과 마찬가지로 FTL 은 hardware event queue 를 폴링(polling)하여 쓰기 요청을 가져온다. 그리고 주소 맵핑 테이블에 접근하여 기존 데이터가 존재하는지 확인한다. 만약, 이미 동일 LPN 에 대한 기존 데이터가 존재한다면 과정 ①처럼 해당 페이지 내의 새로운 섹터를 제외한 나머지 사용자 데이터(이를 left hole 및 right hole sectors 라 부름)를 SATA write 버퍼에 올린다. 그리고 과정 ②와 같이 하드웨어로 하여금 호스트로부터 새로운 사용자 데이터 도착을 기다렸다가 빈 페이지에 함께 기록한다. 만약, 기존 데이터가 존재하지 않거나, 페이지 크기만큼의 쓰기 연산을 수행해야 한다면 과정 ①은 생략된다.

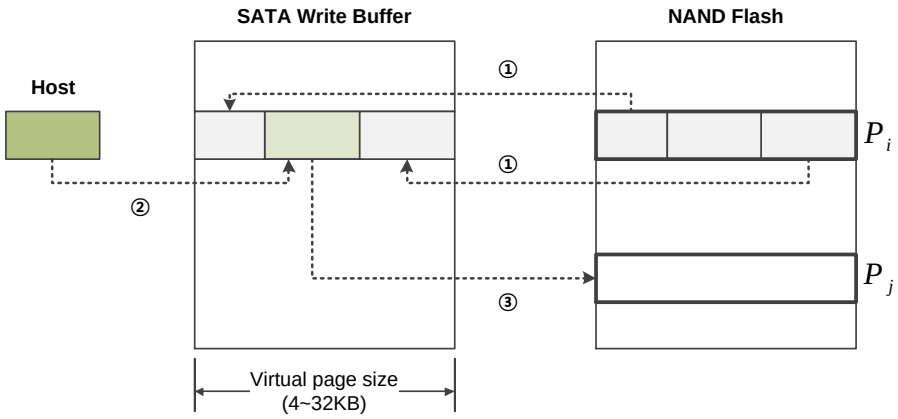

Figure 16) Write operation in Tutorial FTL

Tutorial FTL 의 write 연산에서 새로운 데이터를 쓸 free page 를 할당하는 과정은 병렬성을 최대한 높이기 위한 방식으로 구현되어 있다. 즉, 각 LPN 에 대한 쓰기연산은 특정 bank 에 고정되어 수행되지 않고, 이전에 쓰여진 bank 의 바로 다음 bank 에 연속해서 기록되게 된다. 그리고 기록된 bank 번호와 VPN 정보를 address mapping table 에 함께 기록한다 (물론, 이러한 쓰기연산 방식은 순차읽기 시 병렬성 측면에서 손해볼 수 있다).

한편, Tutorial FTL 은 쓰기 연산 시, 변경된 메타 데이터를 별도로 플래시 메모리에 기록하지 않지만, 고장 회복을 위해서는 플래시 메모리 내 메타 영역을 확보하여 주기적으로 메타 정보의 변경 내용을 기록해주어야 한다.

BSP Interrupt handling

FTL 은 플래시 메모리에 전달된 명령이 처리되는 과정 중에 발생하는 예외상황(e.g., ECC 오류, runtime bad block 발생 등)을 하드웨어 인터럽트로 처리한다.

플래시 메모리 동작 중 내부 오류가 발생하면 BSP 인터럽트가 발생하게 되는데, 플래시 컨트롤러가 하드웨어 인터럽트 컨트롤러에 해당 인터럽트를 통보한다. 하드웨어 인터럽트 컨트롤러는 ARM 에 IRQ 인터럽트가 발생했다는 사실을 알리면, ARM 은 IRQ 인터럽트 핸들러(`./target_spw/misc.c` 의 `irq_handler()`)를 호출한다.

IRQ 인터럽트 핸들러는 플래시 컨트롤러에 의한 인터럽트(`INTR_FLASH`)일 경우, FTL 인터럽트 서비스 루틴(`ftl_isr`) 함수를 호출한다. 이후, FTL 은 인터럽트 레지스터(`BSP_INTR`)를 확인하여 오류 발생 원인을 찾고 적합한 예외 처리 코드를 수행한다.

3.3.3. Dummy FTL

Dummy FTL 은 SATA 와 DRAM 속도를 측정하기 위한 가상 FTL 로써, 플래시 메모리에 접근하지 않고 event queue 로부터 전달된 호스트 요청을 처리하기 위한 최소한의 코드만 구현되어 있다.

Dummy FTL 의 실제 구현(`./ftl_dummy/ftl.c`)은, `ftl_read/ftl_write` 함수의 경우, 플래시 메모리에서 데이터를 읽거나 쓰는 연산을 수행하지 않는다. 즉, SATA 와 FTL 의 정상적인 동작을 위해 SATA read/write 버퍼의 버퍼 포인터만 조정하는 연산만 수행하도록 구현되어 있다.

3.4. Flash Interface Layer

FIL 은 FTL 이 실제 IO 명령을 플래시 메모리에 전달하기 위한 계층으로써, LLD 에 의해 플래시 명령(i.e. FCP)을 설정하고 플래시 메모리에 IO 요청을 전달한다.

3.4.1. Flash command issue

LLD 에 전달된 IO 명령은 내부적으로 플래시 명령 전달 함수(`./target_spw/flash.c` 의 `flash_issue_cmd()`)에 의해 이루어 진다. 이 함수는 기본적으로 LLD 에서 설정한 FCP 를 WR 에 전달하는 기능을 수행한다.

한편, 본 펌웨어는 아래 Table 9 처럼 세 가지 타입의 동기/비동기 IO 방법을 제공한다. 이러한 이슈 타입을 잘 활용하여 플래시 컨트롤러에 IO 를 전달하면 SSD 의 IO 처리 병렬성을 높일 수 있다.

Table 9) FCP issue type

Type	Value	Description
RETURN_ON_ISSUE	0x0	WR 이 비면 FCP 내용을 WR 에 전달하고 리턴
RETURN_ON_ACCEPT	0x1	WR 에 전달된 플래시 명령을 bank 가 가져가서 시작할 때까지 대기(polling)한 후 리턴
RETURN_WHEN_DONE	0x2	해당 bank 에서 플래시 명령이 완료할 때까지 대기한 후 리턴

NOTE: 단, 플래시 명령을 전달하는 주최는 FTL 이므로 WR 에 대기하고 있는 명령이 있는 상태에서 다른 플래시 명령을 전달하지 않도록 주의해야 한다.

3.4.2. LLD(Low-level device driver)

LLD 는 플래시 명령을 FCP 에 설정하고, 설정된 FCP 를 WR 에 전달하는 추상화된 인터페이스이다. 이러한 인터페이스는 펌웨어 개발자로 하여금 구현 용이성을 제공할 수 있다. 즉, 플래시 메모리에 IO 명령을 전달하기 위해 매번 플래시 컨트롤러 레지스터를 설정하는 코드를 작성할 필요 없이, 플래시 메모리를 논리적으로 페이지와 블록의 단순한 논리 구조로 바라보게 할 수 있다.

LLD 는 VPAGE 및 VBLK 단위로 플래시 명령을 수행할 수 있도록 구현되어 있다. 대표적으로 (partial) page read/program 및 block erase, 그리고 simple copy-back 및 modified copy-back 의 기능을 담당하는 API 들을 제공한다.

NOTE: LLD API 종류와 자세한 사용법은 본 문서의 4.4 절을 참고할 것

Chapter 4.
Jasmine OpenSSD Platform Software Specification

본 장에서는 Jasmine OpenSSD 플랫폼의 펌웨어 소프트웨어에 대해 명세 한다. 펌웨어 소프트웨어는 크게 펌웨어 소스 파일 및 build 스크립트 그리고 펌웨어 설치 프로그램 등으로 구성된다. 이 장은 펌웨어의 핵심 모듈에 대해 API 중심으로 기술되어 있으며, 다음과 같은 내용을 포함한다.

- ✓ 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

Jasmine OpenSSD 플랫폼의 펌웨어 헤더파일 및 소스파일들은 아래와 같이 구성되어 있다.

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

Location	File List	Description
./	release_lock.inc COPYING, HISTORY, README	디버깅 스크립트 파일, 지적재산권 및 버전 히스토리, Jasmine 펌웨어 build 과정에 대한 전반적인 설명
./build_gnu	Makefile, ld_script build.bat	Code Sourcery G++ toolchain 을 이용한 펌웨어 build 스크립트
./build_rvds	armlink_opt.via file_list.via build.bat	RVDS 컴파일 환경 설정 파일 및 펌웨어 build 스크립트
./ftl_tutorial	ftl.h, ftl.c	Tutorial FTL 관련 파일 폴더
./ftl_greedy	ftl.h, ftl.c	Greedy FTL 관련 파일 폴더
./ftl_dummy	ftl.h, ftl.c	Dummy FTL 관련 파일 폴더
./sata	sata_cmd.c sata_identifiy.c sata_isr.c sata_main.c sata_table.c	SATA 인터페이스 관련 소스 파일 폴더
./installer	ata_7.h, installer.c, installer.sln, ntddstor.h	펌웨어 설치 솔루션 파일. install.exe 파일을 생성
./include	jasmine.h mem_util.h flash.h, peri.h rom.h, *etc.*	펌웨어 소스코드 관련 헤더 파일
./target_spw	init_gnu.s, init_rvds.s flash_warpper.c initialize.c target.h mem_util.c, flash.c, *uart.c etc.*	펌웨어 startup 코드/펌웨어 초기화/ 메모리 유틸리티/LLD API/UART/타이머 유틸리티 등
./tc	tc_synth.c	FTL 코드 검증을 위한 테스트 케이스가 정의된 폴더

4.1.2. Header file

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

Location	File	Description
./include	jasmine.h	Barefoot 컨트롤러 헤더파일 - NAND 스펙에 따른 페이지/블록 크기 정의 - DRAM segmentation 정의
	mem_util.h	Memory utility 헤더파일
	flash.h	플래시 컨트롤러 헤더파일 - FCP/WR/BSP 을 포함한 플래시 컨트롤러 레지스터 셋 정의 - Flash command code 정의 - Bank interrupt flag 정의
	peri.h	Peripheral 디바이스 레지스터 셋 정의 - 인터럽트/DRAM 컨트롤러, Buffer Manager - GPIO/PMU/UART/Watch-dog/Timer/Clock 등
	hi.h	호스트 인터페이스 자료구조 정의 SATA spec command 정의
	ftl.h	FTL 메타데이터 자료구조 정의 FTL public function(Protocol API 포함) 선언
	sata.h	SATA 관련 헤더파일 ATA 커맨드 리스트 정의 및 SATA 명령 structure 선언
	sata_cmd.h	ATA 커맨드 관련 헤더파일
./target_spw	target.h	Target 디바이스 관련 헤더파일 - 메모리 맵 정의 - NAND/PLL/Timer/Clock cycle speed 정의
	misc.h	기타 유용한 함수(LED, Time measurement 등) 선언

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 소스코드 (no GC)
./ftl_greedy	ftl.c	Greedy FTL 소스코드 (simple GC)
./ftl_dummy	ftl.c	Dummy FTL 소스코드 (*not access NAND flash at all*)
./sata	sata_cmd.c	ATA 커맨드 함수 정의
	sata_isr.c	FIQ 및 SATA 인터럽트 서비스 루틴 함수 정의
	sata_main.c	SATA 하드웨어 초기화 및 펌웨어 메인 함수 정의
./installer	installer.c	펌웨어 인스톨을 위한 함수 정의 - enable factory mode - scan bad block list - install block 0 - install FTL metadata
	installer.sln	펌웨어 인스톨러 솔루션 파일 - Visual C++ 2010 Express 로 인스톨러 build

./target_spw	initialize.c	펌웨어 초기화 함수 정의 - PLL/인터럽트/Barefoot 컨트롤러 초기화
	mem_util.c	Memory utility 함수 정의
	flash.c	플래시 메모리 초기화 및 일부 FCP 함수 정의
	flash_wrapper.c	LLD 함수 정의 - 페이지 읽기/쓰기/복사 함수 - 블록 삭제 함수 등
	misc.c	기타 유용한 함수 정의 - LED/NAND block test/Time measurement
	init_gnu.s init_rvds.s	펌웨어 startup 코드(GNU/RVDS)
./tc	tc_synth.c	Synthetic 테스트 케이스 함수 정의

4.1.4. Miscellaneous file

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

Location	File	Description
./	release_lock.inc	JTAG 디버그 포트를 open 하는 스크립트 파일
	READE	Jasmine 펌웨어 build 과정과 실행 방법에 대한 가이드라인 제공
	HISTORY	Jasmine 펌웨어 revision 로그
	COPYING	Jasmine OpenSSD Platform 의 license - GPL ver. 3
./build_rvds	armcc_opt.via armlink_opt.via	RVDS 컴파일 옵션 설정
	file_list.via	펌웨어 build 시, 컴파일 대상이 되는 파일 리스트 - 새로 작성한 파일이 있다면 포함시킬 것
	scatter.scl	RVDS 로 컴파일 시, code/ZI/RW 데이터 메모리 주소 할당 스크립트
./build_gnu	ld_script	GNU 툴로 컴파일 시, code/ZI/RW 데이터 메모리 주소 할당 스크립트

4.2. Installer Function

Installer 함수는 Jasmine 펌웨어 인스톨러(./installer/installer.c)에서 FTL 동작을 위한 초기 상태를 구축하는 기능을 수행한다.

4.2.1. install

```
void install(void)
```

Function:

Jasmine 보드에 펌웨어 인스톨 수행. 이 함수에서 수행하는 연산은 다음과 같음.

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

NOTE: 현재 펌웨어 버전에서는 `ftl_open` 시 NAND 블록 전체를 삭제-즉, `format-` 연산을 수행한다. 하지만 Normal mode 에서 Jasmine 보드를 초기화할 때 *response time out error* 가 발생할 수 있기 때문에, 사실상 이 install 함수 내에서 format 과정을 수행해야 한다. 또한 초기상태의 FTL metadata 를 설치하여, 추후 Jasmine 보드 부팅 시, `ftl_open` 함수에 의해 플래시 메모리에 기록된 메타데이터를 로드함으로써 POR 이 가능하도록 해야 한다.

4.2.2. ftl_install_mapping_table

```
void ftl_install_mapping_table(void)
```

Function:

FTL 동작을 위해 맵핑 테이블을 포함한 각종 메타데이터의 초기 상태를 NAND 플래시에 기록. 이 함수는 펌웨어를 설치하는 모드인 'Factory mode'에서 install 함수에 의해 호출됨.

한편, 이 함수에서 기록한 메타데이터는 Jasmine 보드 부팅 시 FTL 이 초기화될 때 (`ftl_open`) 로드되어야 하므로, FTL 개발자는 `ftl_open` 함수도 함께 수정 및 구현 해야 함.

4.3. FTL Protocol API

FTL protocol API 는 FTL 을 초기화하고 호스트로부터 전달된 명령을 처리하는 기능들을 담당하며, SATA 로부터 호출되는 함수이다. 이 함수들은 SATA 와 FTL 이 서로 동작하기 위해 반드시 구현되어야 할 핵심 API 들이다.

본 절에서는 Tutorial FTL (`./ftl_tutorial/ftl.h`) 의 FTL public function 들에 대해 명세한다.

4.3.1. ftl_open

```
void ftl_open(void)
```

Function:

FTL 초기화 수행. 디바이스에 전원이 들어온 후 가장 먼저 호출되는 FTL 함수. 펌웨어 메인 함수(`./target_spw/initialize.c` 의 `init_jasmine`)에 의해 호출됨. 이 함수에서는 다음 연산들을 수행함

- Build scan list
- FTL format
- FTL metadata initialization

4.3.2. ftl_read

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

Function:

FTL 읽기 요청을 수행. 실제 IO 는 VPAGE 단위로 수행.

Parameter:

lba –호스트로부터 주어진 논리 섹터 주소
num_sectors – 섹터 개수 (512B 섹터 기준)

4.3.3. ftl_write

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

Function:

FTL 쓰기 요청을 수행. 실제 IO 는 VPAGE 단위로 수행

Parameter:

lba – 호스트로부터 주어진 논리 섹터 주소
num_sectors – 섹터 개수 (512B 섹터 기준)

4.3.4. ftl_flush

```
void ftl_flush(void)
```

Function:

FTL 메타 데이터를 플래시메모리에 기록. SATA 컨트롤러가 idle 상태일 때 주기적으로 호출됨(`./sata/sata_cmd.c` 의 `ata_flush_cache , ata_idle, ata_standby` 등)

4.3.5. ftl_isr

```
void ftl_isr(void)
```

Function:

BSP 인터럽트 핸들러 기능을 담당하며, 플래시 메모리에 IO 요청 도중에 BSP 인터럽트 발생하면 이 함수가 호출됨. 이 함수는 IRQ 인터럽트 핸들러 (`./target_spw/misc.c` 의 `irq_handler`)에 의해 호출됨.

`BSP_INTR` 레지스터를 확인하여 각각의 인터럽트 이벤트에 대한 처리 담당

4.4. LLD API

이 절에서는 FTL 이 내부 연산을 마친 후, 실제 IO 를 처리하기 위해 플래시 컨트롤러에 플래시 명령을 전달하기 위해 사용하는 인터페이스인 LLD(Low-Level device Driver) API 에 대해 명세 한다.

NOTE: 제공되는 LLD API (`./target_spw/flash_wrapper.c`)는 FTL 개발자로 하여금 구현의 편의성을 제공하기 위하여 작성되었으며, 병렬성을 높이기 위한 최적화를 위해서는 별도의 LLD API 또는 FCP 레지스터 설정을 통한 플래시 명령을 FTL 내부에 직접 구현하도록 한다.

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:

FCP 커맨드(FC_NORMAL_READ_OUT)를 사용하여 플래시 메모리에 특정 페이지 읽기 수행. FCP 커맨드는 RETURN_WHEN_DONE 로 WR 에 전달됨

Parameter:

bank – bank 번호 (max: NUM_BANKS)

vblock – VBLK 번호 (max: VBLKS_PER_BANK)

page_num – VPAGE 번호 (max: PAGES_PER_BLK)

buf_addr – 읽은 페이지 데이터가 올라갈 FTL 내부 버퍼 주소

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:

FCP 커맨드(FC_NORMAL_READ_OUT)를 사용하여 플래시 메모리의 특정 페이지 부분 읽기 수행. 본 함수는 주로 아래와 같은 목적으로 사용될 수 있다

- 플래시 메모리에 기록된 메타 데이터를 읽어올 때 (RETURN_WHEN_DONE)
- Partial page program 을 처리하기 위해 특정 column 사용자 데이터 읽기가 필요할 때 (RETURN_ON_ISSUE)

Parameter:

bank – bank 번호 (max: NUM_BANKS)

vblock –VBLK 번호 (max: VBLKS_PER_BANK)

page_num – VPAGE 번호 (max: PAGES_PER_BLK)

sect_offset – VPAGE 의 시작 섹터 오프셋 (max: SECTORS_PER_VPAGE)

num_sectors – 섹터 개수 (단일 섹터 크기는 512B 기준)

buf_addr – 읽은 페이지 데이터가 올라갈 FTL 내부 버퍼 주소

issue_flag – FCP 이슈 플래그 설정. (본 문서 3.4.1 절의 Table 9 참고할 것)

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:

FCP 커맨드(FC_NORMAL_READ_OUT)를 사용하여 플래시 메모리에서 특정 페이지를 읽어 호스트 인터페이스인 SATA 에 전달

페이지 읽기 연산 후, SATA read 버퍼포인터 조정 수행

Parameter:

bank – bank 번호 (max: NUM_BANKS)

vblock – VBLK 번호 (max: VBLKS_PER_BANK)

page_num – VPAGE 번호 (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:

FCP 커맨드(FC_NORMAL_READ_OUT)를 사용하여 플래시 메모리에서 특정 페이지의 일부 섹터만을 읽어(i.e., partial page read), host 인터페이스인 SATA 에 전달

페이지 읽기 연산 후, SATA read 버퍼포인터 조정 수행

Parameter:

bank – bank 번호 (max: NUM_BANKS)
vblock – VBLK 번호 (max: VBLKS_PER_BANK)
page_num – VPAGE 번호 (max: PAGES_PER_BLK)
sect_offset – partial read 를 위한 VPAGE 의 시작 섹터 오프셋 (max: SECTORS_PER_VPAGE)
num_sectors – 섹터 개수 (단일 섹터 크기는 512B 기준)

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:

FCP 커맨드(FC_NORMAL_IN_PROG)를 사용하여 플래시 메모리에서 특정 페이지에 FTL 내부 버퍼의 데이터를 기록 (주로 메타데이터 기록 시에 사용될 수 있음).

Parameter:

bank – bank 번호 (max: NUM_BANKS)
vblock – VBLK 번호 (max: VBLKS_PER_BANK)
page_num – VPAGE 번호 (max: PAGES_PER_BLK)
buf_addr – 쓰기 페이지 데이터가 존재하는 FTL 내부 버퍼 주소. VPAGE-align 되어 있어야 함

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:

FCP 커맨드(FC_NORMAL_IN_PROG)를 사용하여 플래시 메모리에서 특정 페이지에 SATA write 버퍼 데이터를 기록

페이지 쓰기 연산 후, SATA write 버퍼 포인터 조정

Parameter:

bank – bank 번호 (max: NUM_BANKS)
vblock – VBLK 번호 (max: VBLKS_PER_BANK)
page_num – VPAGE 번호 (max: PAGES_PER_BLK)

4.4.7. nand_page_ptprogram_from_host

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

Function:

FCP 커맨드(FC_NORMAL_IN_PROG)를 사용하여 플래시 메모리의 특정 페이지내 일부 섹터만을 기록(i.e., partial program)

페이지 쓰기 연산 후, SATA write 버퍼 조정

Parameter:

bank – bank 번호 (max: NUM_BANKS)
vblock – VBLK 번호 (max: VBLKS_PER_BANK)
page_num – VPAGE 번호 (max: PAGES_PER_BLK)
sect_offset – partial program 을 위한 VPAGE 의 시작 섹터 오프셋 (max: SECTORS_PER_VPAGE)
num_sectors – 섹터 개수 (단일 섹터 크기는 512B 기준)

NOTE: 동일 페이지에 1 회 이상의 partial programming 을 수행하기 위해서는, NAND 플래시의 NOP(Number of Program)를 확인할 것.

4.4.8. 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:

FCP 커맨드(FC_COPYBACK)를 사용하여 동일 bank 내 에서 페이지를 복사하는 연산을 수행. DRAM 버퍼를 거치지 않기 때문에 빠른 페이지 복사가 가능함

참고로, 아래와 같은 경우에는 내부 copy-back 이 불가능 하므로 FTL 내부 버퍼에 페이지 읽기를 수행한 후, 페이지 쓰기 연산을 수행함

- NAND 플래시 메모리가 internal copy-back operation 을 지원하지 않는 경우
- 각각의 페이지가 서로 다른 plane 인 경우

Parameter:

bank – bank 번호 (max: NUM_BANKS)
src_vblock – 원본 VBLK 번호 (max: VBLKS_PER_BANK)
src_page – 원본 VPAGE 번호 (max: PAGES_PER_BLK)
dst_vblock – 복사될 VBLK 번호 (max: VBLKS_PER_BANK)
dst_page – 복사될 VPAGE 번호 (max: PAGES_PER_BLK)

4.4.9. 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:

FCP 커맨드(FC_MODIFY_COPYBACK)를 사용하여 동일 bank 내 에서 특정 페이지의 데이터를 수정하여, 복사하는 연산을 수행.

note: 아래와 같은 경우, internal modified copy-back 이 불가능 하므로 FTL 내부 버퍼에 페이지 읽기를 수행한 후, 새 데이터와 함께 페이지 쓰기 연산을 수행함

- NAND 플래시 메모리가 internal copy-back operation 을 지원하지 않는 경우
- 각각의 페이지가 서로 다른 plane 인 경우

FC_MODIFY_COPYBACK 이 불가능 한 경우, 아래와 같은 순서로 modified copy-back 연산을 수행한다.

1. 본 페이지를 읽어, DRAM copy 버퍼에 올림
2. 기존 데이터 일부(left hole)를 목적 페이지에 FC_NORMAL_IN 명령을 사용하여 플래시 메모리에 전달
3. 새 데이터를 목적 페이지에 FC_IN 명령으로 플래시 메모리에 전달
4. 나머지 기존 데이터를(right hole)를 FC_IN_PROG 명령으로 전달하여, 2, 3 단계에서 전달한 데이터와 함께 플래시 메모리에 programming 수행

Parameter:

bank – bank 번호 (max: NUM_BANKS)
src_vblock – 원본 VBLK 번호 (max: VBLKS_PER_BANK)
src_page – 원본 VPAGE 번호 (max: PAGES_PER_BLK)
dst_vblock – 복사될 VBLK 번호 (max: VBLKS_PER_BANK)
dst_page – 복사될 VPAGE 번호 (max: PAGES_PER_BLK)
sect_offset – 새로운 데이터로 교체될 섹터 오프셋 (max: SECTORS_PER_PAGE)
dma_addr – 새 데이터를 담고 있는 버퍼 주소
dma_count – 새 데이터 크기 (단위: Byte)

4.4.10. nand_block_erase

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

Function:

FCP 커맨드(FC_ERASE)를 사용하여 해당 bank 의 특정 VBLK 을 삭제

note: 1-plane mode 일 경우, 2 개의 물리불록을 삭제. 2-plane mode 일 경우, 4 개의 물리 블록을 삭제

Parameter:

bank – bank 번호 (max: NUM_BANKS)
vblock – VBLK 번호 (max: VBLKS_PER_BANK)

4.5. Memory Utility API

Memory utility API 는 SRAM 과 DRAM 간의 메모리 작업을 위한 함수이며, 이 함수들은 메모리 값 읽기/쓰기/탐색 등이 필요할 경우 사용된다.

Memory utility 함수는 ./target_spw/mem_util.c 에 정의되어 있다.

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:

SRAM 의 메모리 값 설정

Parameter:

ADDR – SRAM 영역 내 메모리 주소이며, 4 Byte-align 되어 있어야 함 (max: 0x1000_0000)

VAL – 설정 값

BYTES – 설정하고자 하는 Byte 영역이며, 4Byte 단위여야 함

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:

DRAM 의 메모리 값 설정

Parameter:

ADDR – DRAM 영역 내 메모리 주소 (min: 0x4000_0000)

VAL – 설정 값

BYTES – 설정하고자 하는 Byte 영역을 의미하며, `DRAM_ECC_UNIT` 의 배수여야 함

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:

DMA 로 메모리 복사 수행. DRAM 에서 자주 참조되는 메타데이터는 SRAM 에 복사해서 사용하면 성능을 높일 수 있음.

Parameter:

DST – 목적지 메모리 주소. 4 Byte-align 되어 있어야 함

SRC – 원본 메모리 주소. 4 Byte-align 되어 있어야 함

BYTES – 설정하고자 하는 byte 영역이며, 4Byte 단위여야 함 (max: 32768) (단, DRAM-to-DRAM 의 경우는 `DRAM_ECC_UNIT` 의 배수가 되어야 함)

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:

SRAM 비트맵 메모리 영역에서 특정 값이 존재하는 지 탐색

Parameter:

BMP – 비트맵 메모리 주소. 4 Byte-align 되어 있어야 함 (max: 0x1000_0000)

BYTES – 설정하고자 하는 byte 영역이며, 4 Byte 단위여야 함 (max: UNIT * SIZE = 32768)

VAL – 찾고자 하는 값 (반드시 0 또는 1)

Returns:

성공 시, 매칭되는 값이 존재하는 bitmap 의 인덱스 번호

실패 시, 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:

DRAM 비트맵 메모리 영역에서 특정 값이 존재하는 지 탐색

Parameter:

BMP – 비트맵 메모리 주소. 4 Byte-align 되어 있어야 함 (min: 0x4000_0000)

BYTES – 설정하고자 하는 Byte 영역이며, 4 Byte 단위여야 함 (max: UNIT * SIZE = 32768)

VAL – 찾고자 하는 값. (0 또는 1)

Returns:

성공 시, 매칭되는 값이 존재하는 bitmap 의 인덱스 번호

실패 시, 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:

메모리 영역에서 최소 또는 최대 값 탐색

Parameter:

ADDR – 탐색 시작 메모리 주소. 4 Byte-align 되어 있어야 함

UNIT – 탐색 단위 (e.g., 1, 2, 4 Byte)

SIZE – 탐색 인덱스 범위 (max: UNIT * SIZE = 32768)

CMD – MU 커맨트 코드

(e.g., `MU_CMD_SEARCH_MAX_SRAM`, `MU_CMD_SEARCH_MIN_SRAM`, `MU_CMD_SEARCH_MAX_DRAM`, `MU_CMD_SEARCH_MIN_DRAM`)

Returns:

성공 시, 최소 또는 최대 값이 존재하는 인덱스 번호

실패 시, 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:

메모리 영역에서 특정 값이 존재하는지 탐색

Parameter:

ADDR – 탐색 시작 메모리 주소. 4 Byte-align 되어 있어야 함

UNIT – 탐색 단위 (e.g., 1,2, 4 Byte)

SIZE – 탐색 인덱스 범위 (max: UNIT * SIZE = 32768)

CMD – MU 커맨트 코드

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

VAL – 찾고자 하는 값

Returns:

성공 시, 매칭되는 값이 존재하는 인덱스 번호

실패 시, SIZE

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:

Memory utility(DMA)를 활용하여, DRAM 32KByte 영역에서 4Byte 특정 값 탐색 시, 최대 약 180us 소요 (i.e., 177MB/s. 단, SATA 또는 NAND 가 DRAM 데이터를 access 할 경우, 더 느려질 수 있음)

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:

SRAM 의 두 메모리 영역 비교 수행

Parameter:

ADDR1 – 첫 번째 메모리 주소. 4 Byte-align 되어 있어야 함 (max: 0x1000_0000)

ADDR2 – 두 번째 메모리 주소. 4 Byte-align 되어 있어야 함 (max: 0x1000_0000)

BYTES– 비교 범위

Returns:

같은 경우, 0 을 리턴

다를 경우, -1 또는 1 을 리턴

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:

DRAM 의 두 메모리 영역 비교 수행. 이 함수는 DMA 가 아닌 CPU 가 직접 DRAM 데이터를 읽어와서 비교

Parameter:

ADDR1 – 시작메모리 주소. 4 Byte-align 되어 있어야 함 (min: 0x4000_0000)

ADDR2 – 시작메모리 주소. 4 Byte-align 되어 있어야 함 (min: 0x4000_0000)

BYTES – 비교 범위

Returns:

같은 경우, 0 을 리턴

다를 경우, -1 또는 1 을 리턴

4.5.10. _read_dram_8

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

Macro – `#define read_dram_8(ADDR)`

Function:

DRAM 에서 SRAM 으로 8 bit(1 Byte)를 CPU 가 직접 해당 메모리 주소에서 읽어옴.

Parameter:

ADDR – 읽어올 DRAM 메모리 주소

Returns:

DRAM 에서 읽어온 8bit 메모리 데이터 값

4.5.11. _read_dram_16

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

Macro – `#define read_dram_16(ADDR)`

Function:

DRAM 에서 SRAM 으로 16 bit(2 Bytes)를 CPU 가 직접 해당 메모리 주소에서 읽어옴.

Parameter:

ADDR – 읽어올 DRAM 메모리 주소

Returns:

DRAM 에서 읽어온 16 bit 메모리 데이터 값

4.5.12. _read_dram_32

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

Macro – `#define read_dram_32(ADDR)`

Function:

DRAM 에서 SRAM 으로 32 bit(4 Byte)를 CPU 가 직접 해당 메모리 주소에서 읽어옴.

Parameter:

ADDR – 읽어올 DRAM 메모리 주소. 4 Byte-align 되어 있어야 함

Returns:

DRAM 에서 읽어온 32bit 메모리 데이터 값

4.5.13. _write_dram_8

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

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

Function:

`mem_copy` 함수를 사용하여 DRAM 상의 메모리 주소에 8 bit(1 Byte) 데이터를 기록. (내부적으로는 align 된 상대 주소를 사용하므로 메모리 디버깅 시 유의할 것)

Parameter:

ADDR – 데이터를 쓸 DRAM 메모리 주소 (min: 0x4000_0000)
VAR – 8 bit 데이터

4.5.14. _write_dram_16

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

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

Function:

mem_copy 함수를 사용하여 DRAM 상의 메모리 주소에 16 bit(2 Bytes) 데이터를 기록. (내부적으로는 align 된 상대 주소를 사용하므로 메모리 디버깅 시 유의할 것)

Parameter:

ADDR – 데이터를 쓸 DRAM 메모리 주소 (min: 0x4000_0000)
VAR – 16 bit 데이터

4.5.15. _write_dram_32

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

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

Function:

mem_copy 함수를 사용하여 DRAM 상의 메모리 주소에 32 bit(4 Bytes) 데이터를 기록

Parameter:

ADDR – 데이터를 쓸 DRAM 메모리 주소. 4 Byte-align 되어 있어야 함 (min: 0x4000_0000)
VAR – 32 bit 데이터

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:

DRAM 메모리 상의 특정 오프셋을 one bit SET

Parameter:

BASE_ADDR – DRAM base 주소
BIT_OFFSET – BASE_ADDR 기준의 bit offset

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:

DRAM 메모리 상의 특정 오프셋을 one bit UNSET(CLEAR)

Parameter:

BASE_ADDR – DRAM base 주소
BIT_OFFSET – BASE_ADDR 기준의 bit offset

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:

`DRAM` 메모리 상의 특정 오프셋(one bit)에 대해 테스트 수행

Parameter:

BASE_ADDR – DRAM base 메모리 주소

BIT_OFFSET – BASE_ADDR 기준의 bit offset

Returns:

테스트하고자 하는 bit_offset 정보가 0 일 경우, 0 을 리턴

아닐 경우, 0 이 아닌 값을 리턴