

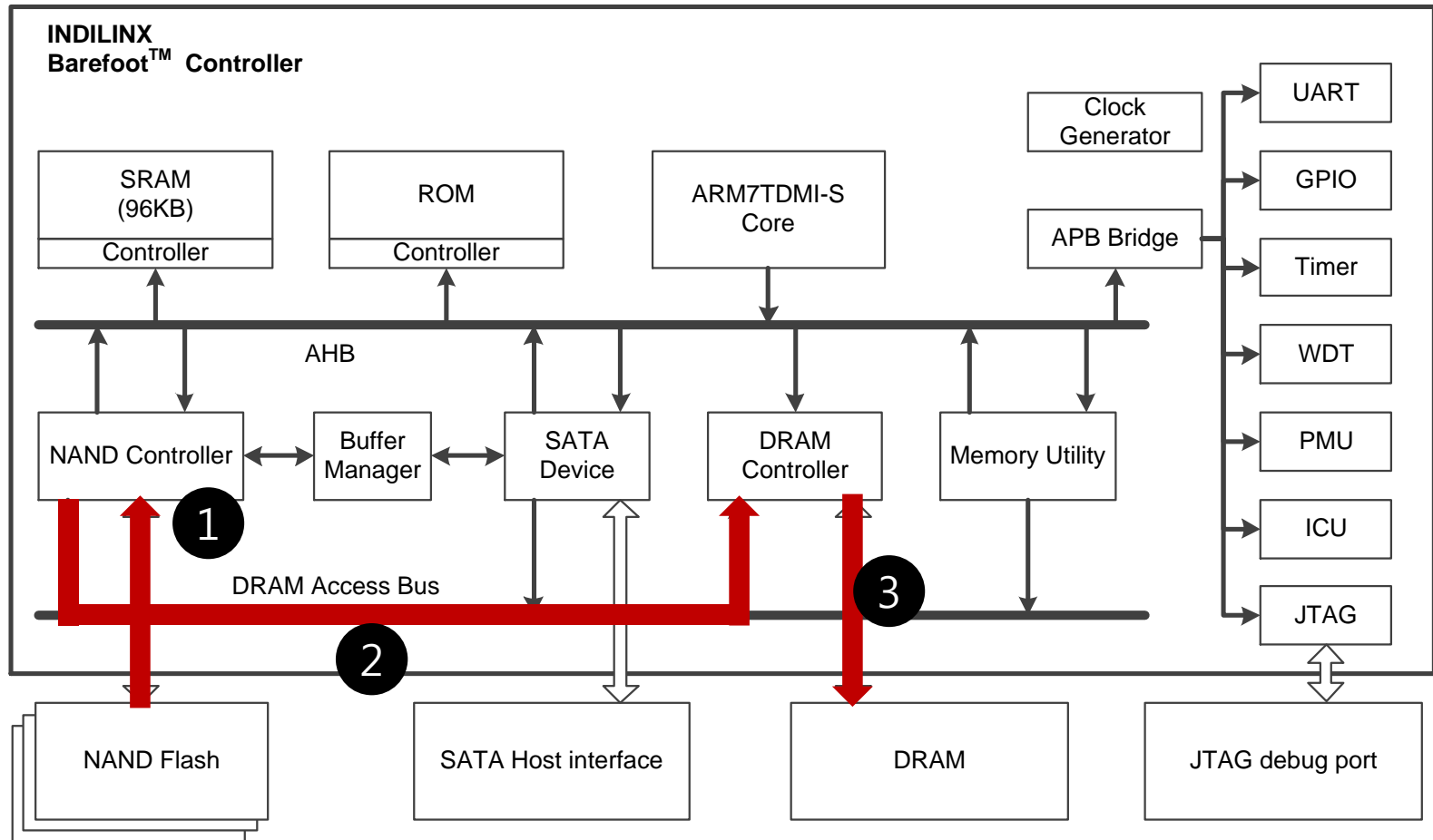
# Dummy FTL

Sejun Kwon  
Computer Systems Laboratory  
Sungkyunkwan University  
<http://csl.skku.edu>

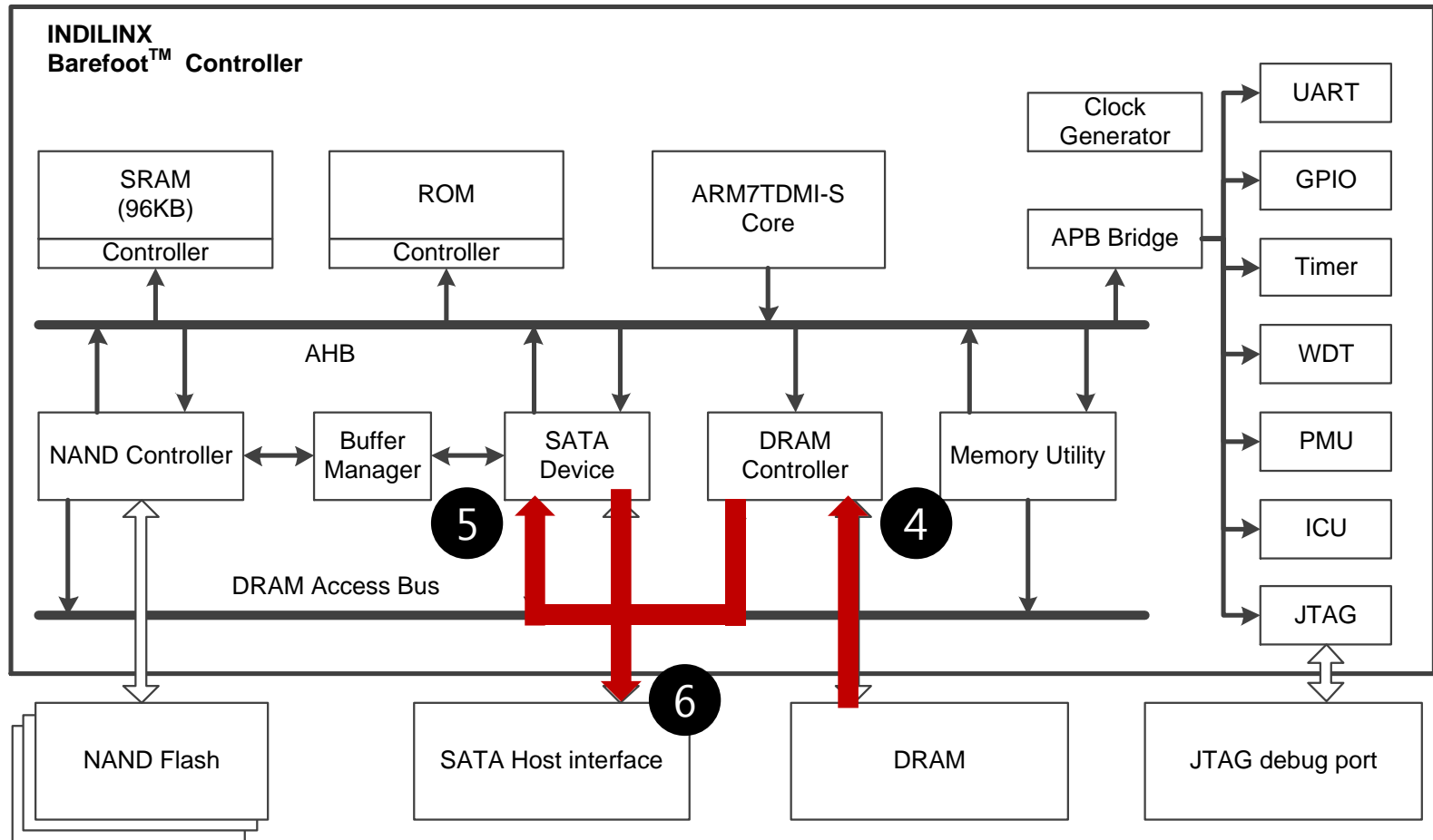
# Contents

- Read/Write Command
- SATA Controller
- Buffer Manager
- Dummy FTL
- Appendix. IOmeter

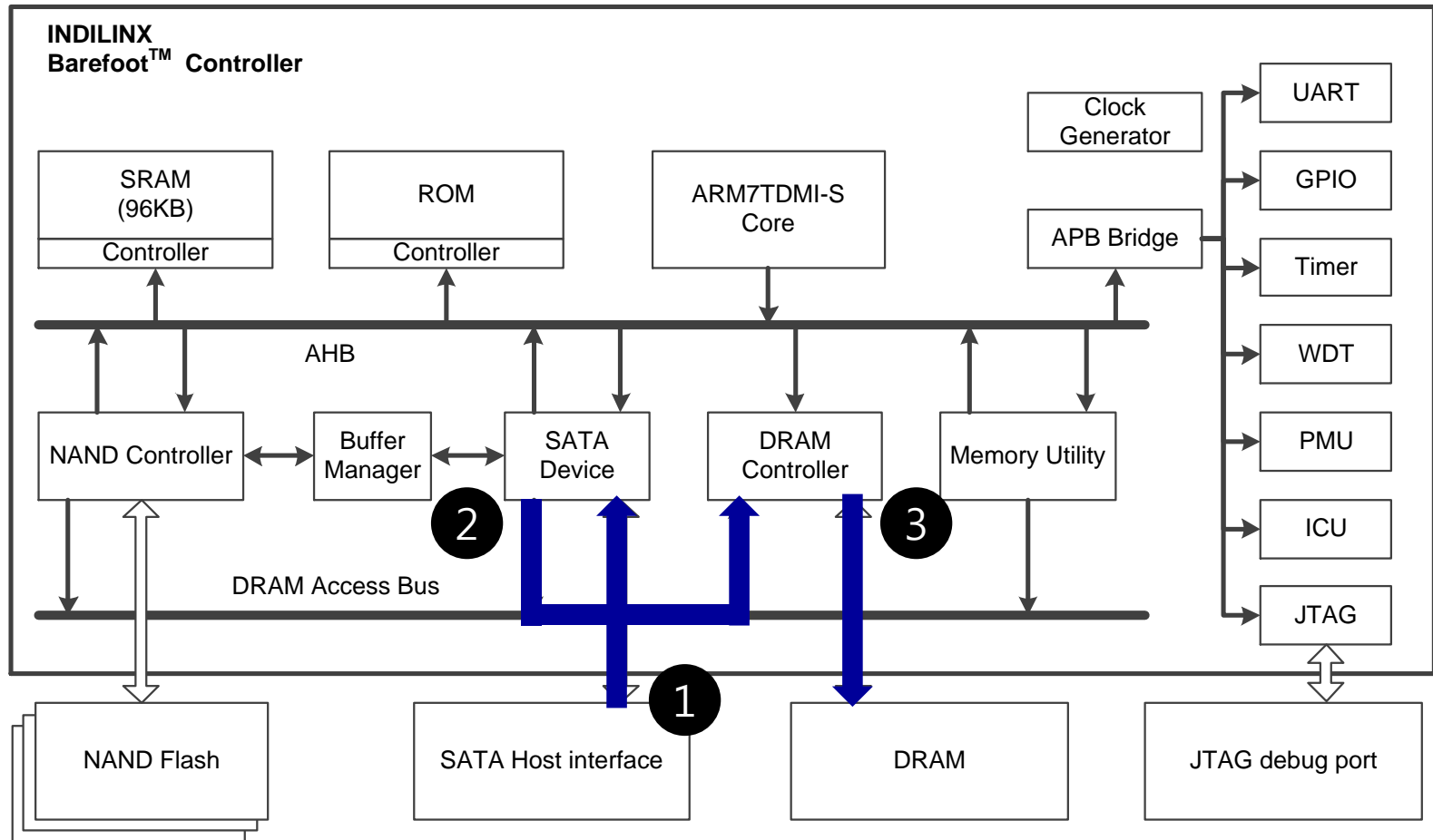
# Read Command



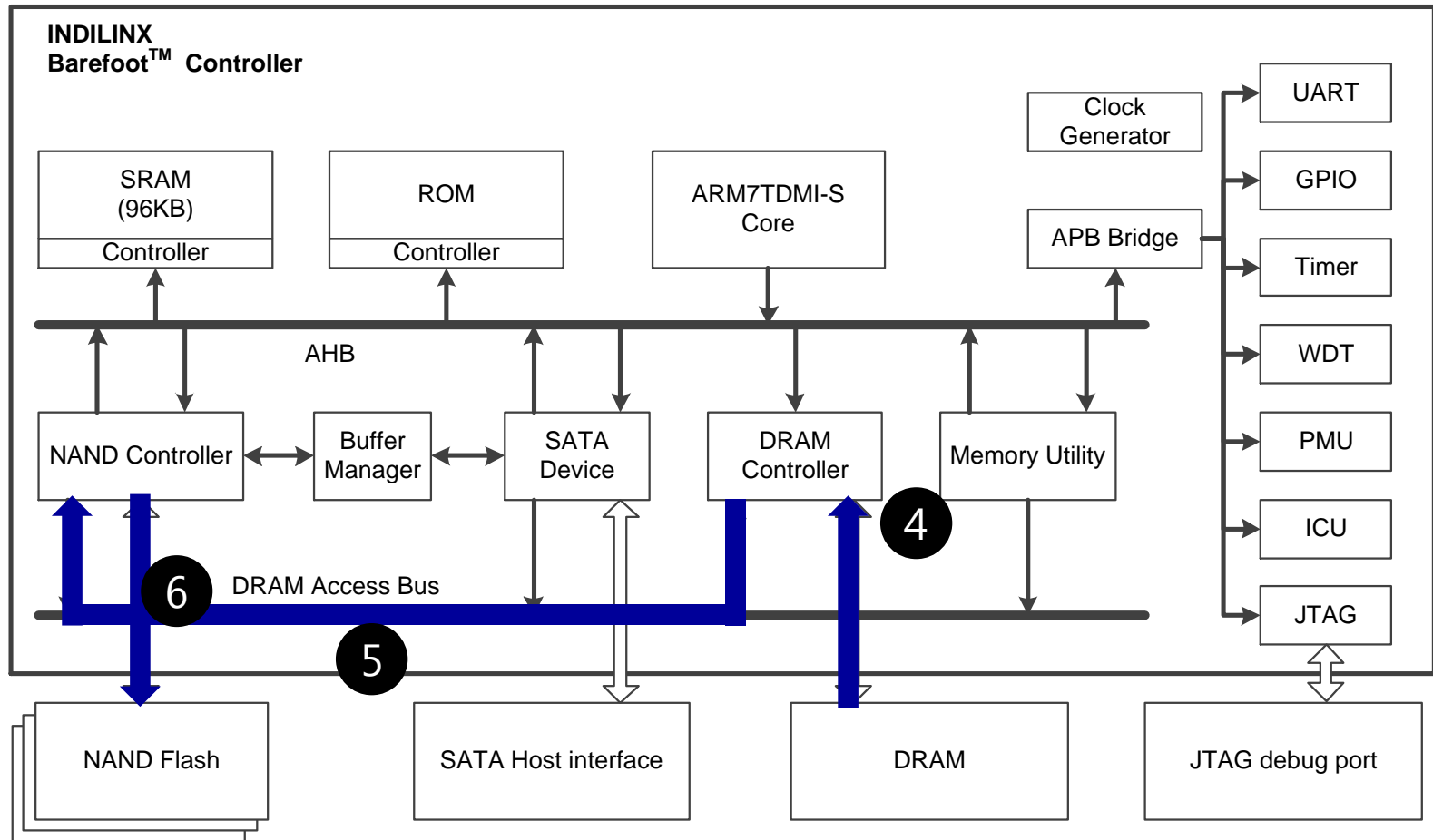
# Read Command



# Write Command



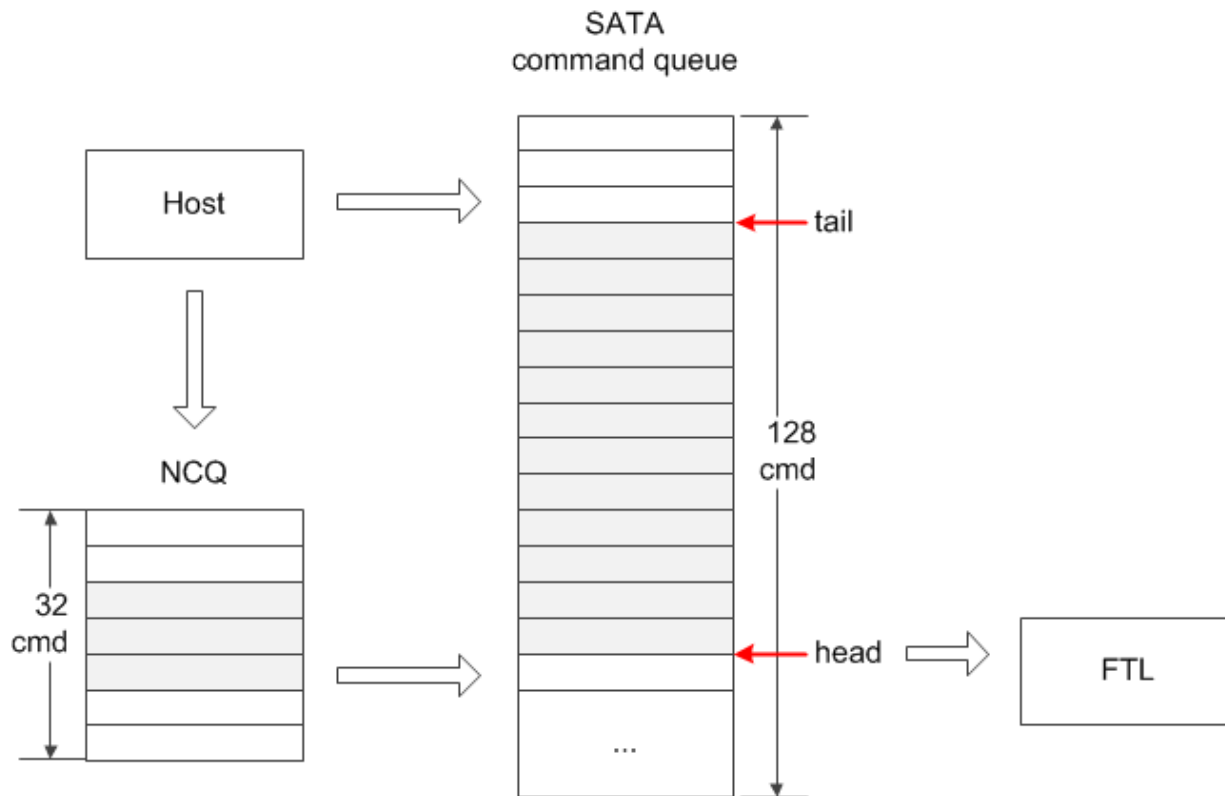
# Write Command



# SATA Controller

- **SATA Event Queue**
  - 128 slots for SATA commands
  - An entry is inserted by ISR upon command reception
  - An entry is removed by FTL top level loop and processed
- **NCQ**
  - 32 slots for NCQ commands
  - Currently, NCQ is disabled in Jasmine firmware

# SATA Controller

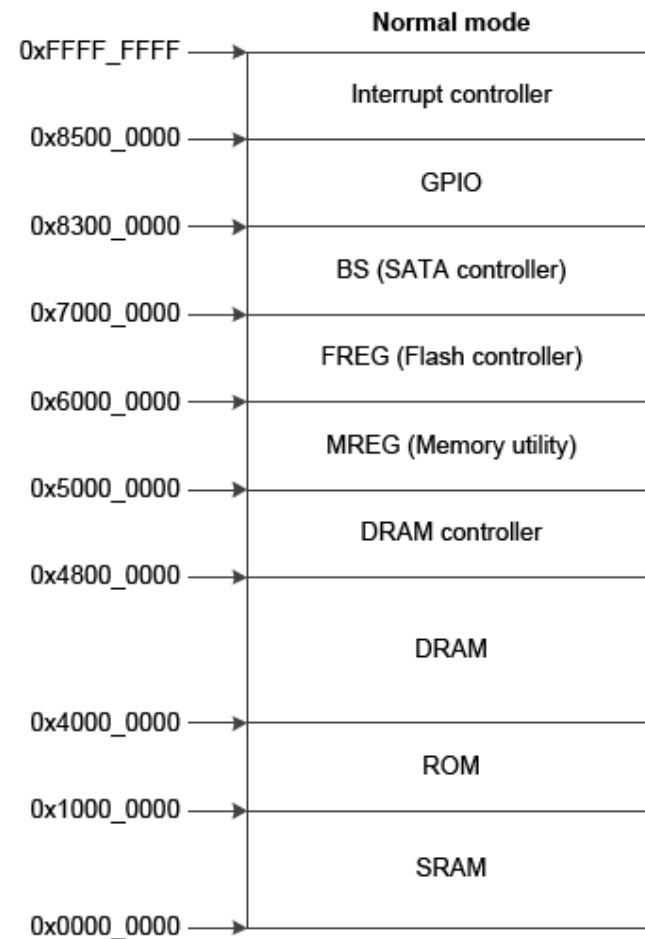
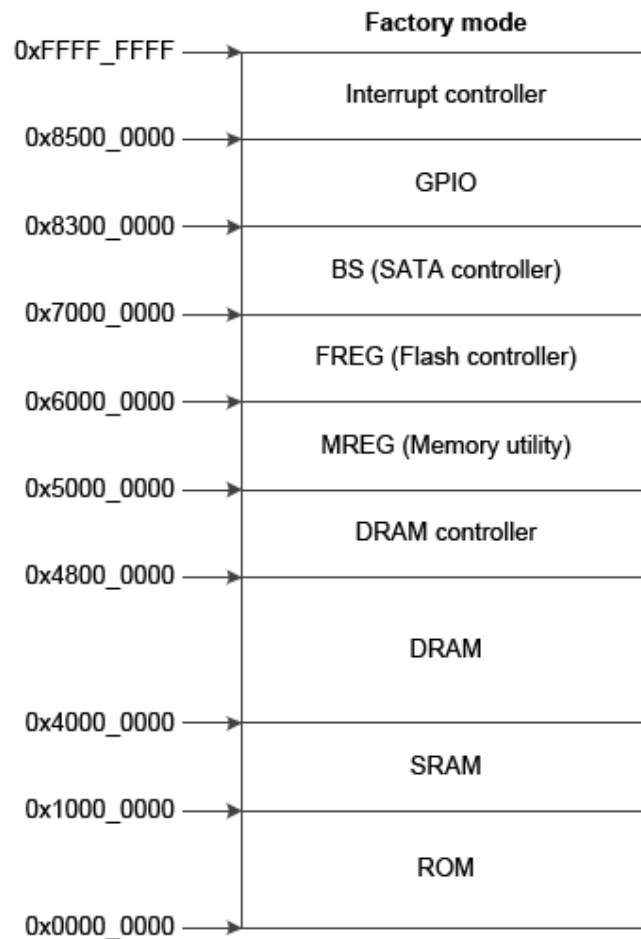




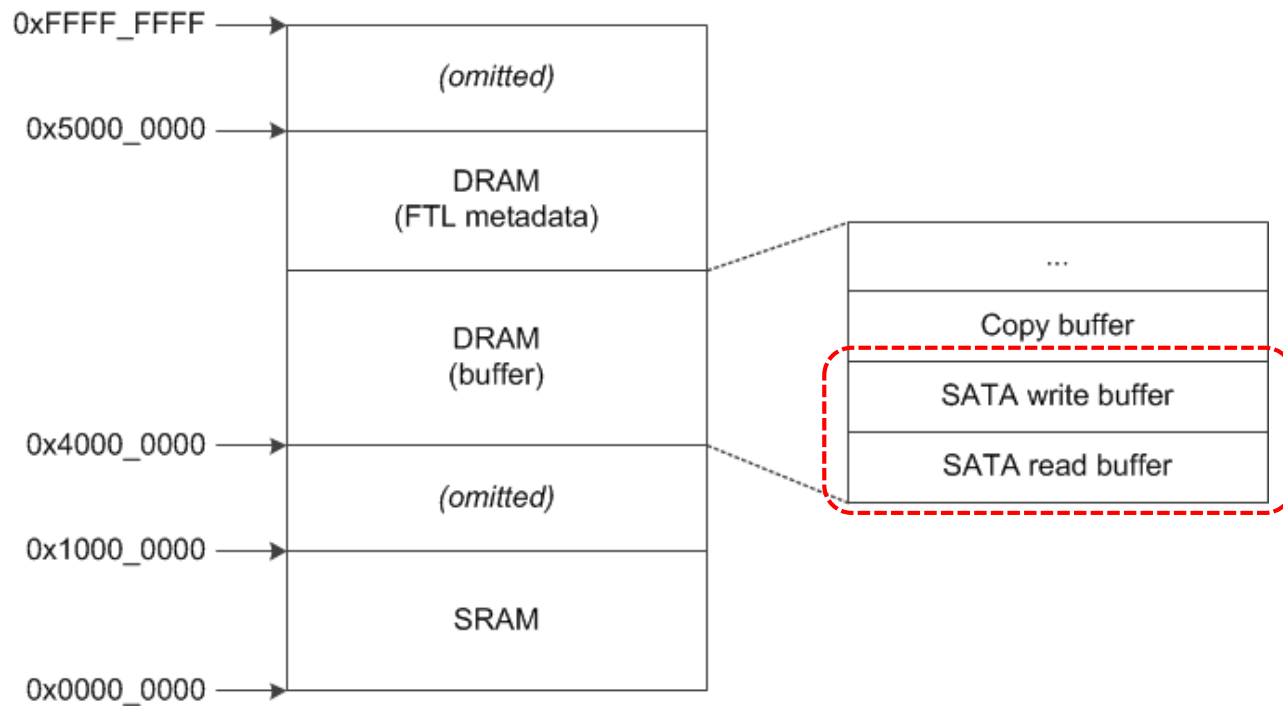
# Buffer Manager

- SATA data is buffered in DRAM
- Separate read and write buffer space
- Buffer space consists of multiple buffers
  - 4 ~ 32 KB per buffer
  - Must be identical to virtual flash page size

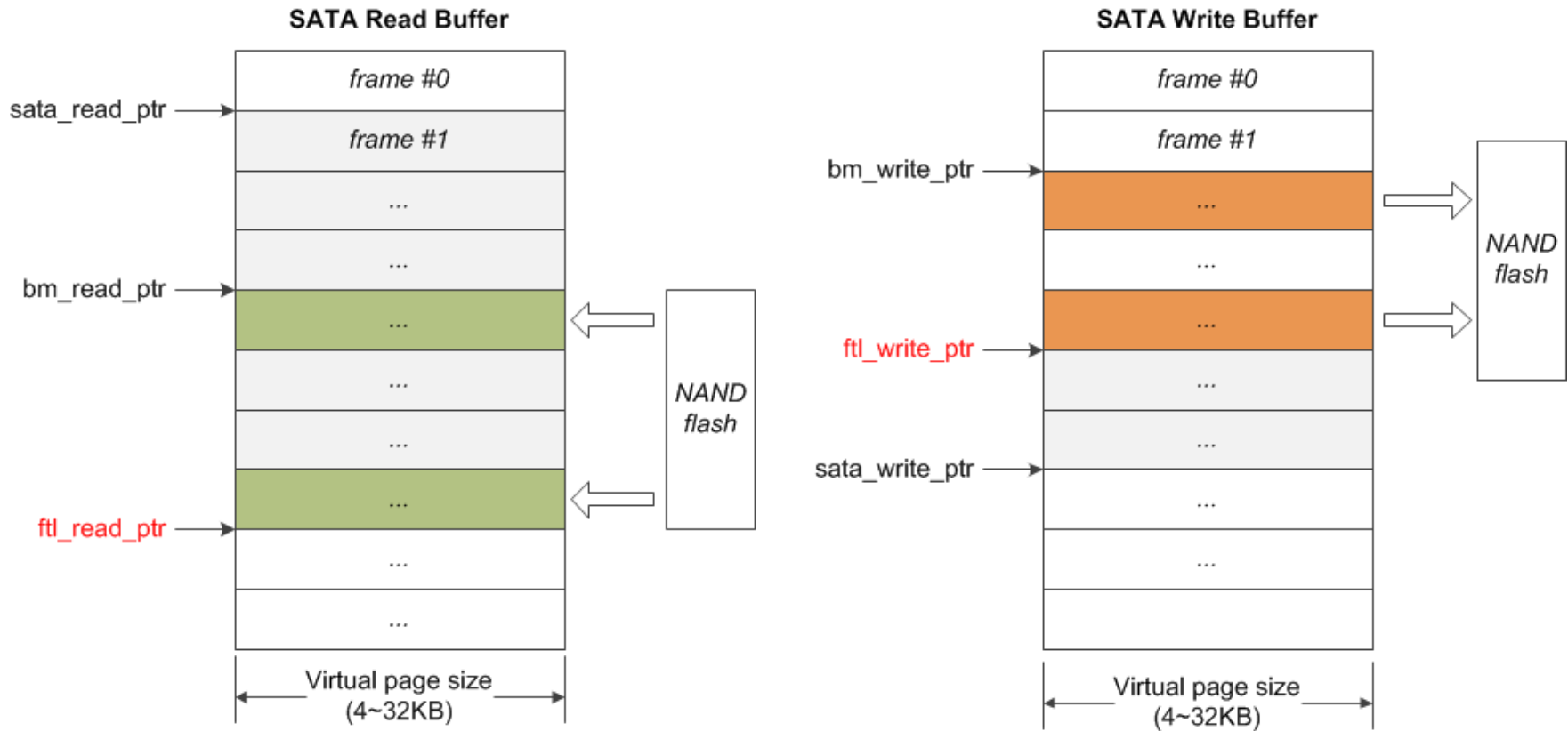
# Memory Map



# Memory Map



# Buffer Manager



# Buffer Manager

- `ftl_read_ptr(variable)`
  - Buffer ID maintained by firmware
  - Firmware reads data from NAND to `ftl_read_ptr`
- `sata_read_ptr`
  - Buffer ID which SATA read transfer is being done
- `bm_read_ptr`
  - Buffer ID which NAND read transfer is being done
  - `sata_read_ptr` does not run ahead of `bm_read_ptr`

# Buffer Manager

- `ftl_write_ptr(variable)`
  - Buffer ID maintained by firmware
  - Firmware writes data from `ftl_write_ptr` to NAND
- `sata_write_ptr`
  - Buffer ID which SATA write transfer is being done
- `bm_write_ptr`
  - Buffer ID which NAND write transfer is being done
  - `sata_write_ptr` does not run ahead of `bm_write_ptr`

# Dummy FTL

- `./ftl_dummy`
  - `ftl.c, ftl.h`
- Literally, Dummy FTL is not a real FTL
  - Not access NAND flash at all
  - write data to DRAM buffer
  - read data from DRAM buffer

# Makefile

- `./build_gnu/Makefile`

```
Makefile - 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
FTL = tutorial
PREFIX = arm-none-eabi-
CC = $(PREFIX)gcc
AS = $(PREFIX)as
LD = $(PREFIX)ld
OBJCOPY = $(PREFIX)objcopy
RM = del

INCLUDES = -I../include -I../ftl_$(FTL) -I../sata -I../target_spw|
CFLAGS = -mcpu=arm7tdmi-s -mthumb-interwork -ffreestanding -nostdlib -std=c99 -O2 -g -DPROGRAM_MAIN_FW -Wall
ASFLAGS = -R -mcpu=arm7tdmi-s
LDFLAGS = -static -nostartfiles -ffreestanding -T ld_script -Wl,-O1,-Map=list.txt
LIBS = -lgcc
VPATH = ../ftl_$(FTL);../sata;../target_spw

SRCS = ftl.c sata_identify.c sata_cmd.c sata_isr.c sata_main.c sata_table.c initialize.c mem_util.c flash.c flash_wrapper.c misc.c uart.c
INITSRC = ../target_spw/init_smc.c
OBJS = $(SRCS:.c=.o) init.o
DEPS = $(SRCS:.c=.d)
TARGET = firmware
TARGETELF = $(TARGET).elf
TARGETBIN = $(TARGET).bin
```



# Start-up

- `./target_spw/init_gnu.s`
  - Call `init_jasmine()`
  - Call `Main()`
- `init_jasmine()`
  - Initialize H/W configuration
- `Main()`
  - FTL top level loop

# Start-up

- `./sata/sata_main.c`

```
void Main(void)
{
    while (1)
    {
        if (eventq_get_count())
        {
            CMD_T cmd;

            eventq_get(&cmd);

            if (cmd.cmd_type == READ)
            {
                ftl_read(cmd.lba, cmd.sector_count);
            }
            else
            {
                ftl_write(cmd.lba, cmd.sector_count);
            }
        }
        else if (g_sata_context.slow_cmd.status == SLOW_CMD_STATUS_PENDING)
        {
            void (*ata_function)(UINT32 lba, UINT32 sector_count);

            slow_cmd_t* slow_cmd = &g_sata_context.slow_cmd;
            slow_cmd->status = SLOW_CMD_STATUS_BUSY;

            ata_function = search_ata_function(slow_cmd->code);
            ata_function(slow_cmd->lba, slow_cmd->sector_count);

            slow_cmd->status = SLOW_CMD_STATUS_NONE;
        }
        else
        {
            // idle time operations
        }
    }
} // end while 1 ?
} // end Main ?
```

# Read Operation

- `./ftl_dummy/ftl.c`

```
void ftl_read(UINT32 const lba, UINT32 const total_sectors)
{
    UINT32 num_sectors_to_read;

    UINT32 lpage_addr      = lba / SECTORS_PER_PAGE; // logical page address
    UINT32 sect_offset     = lba % SECTORS_PER_PAGE; // sector offset within the page
    UINT32 sectors_remain  = total_sectors;

    while (sectors_remain != 0) // one page per iteration
    {
        if (sect_offset + sectors_remain < SECTORS_PER_PAGE)
        {
            num_sectors_to_read = sectors_remain;
        }
        else
        {
            num_sectors_to_read = SECTORS_PER_PAGE - sect_offset;
        }

        
            UINT32 next_read_buf_id = (g_ftl_read_buf_id + 1) % NUM_RD_BUFFERS;

            while (next_read_buf_id == GETREG(SATA_RBUF_PTR)); // wait if the read buffer is full (slow host)

            SETREG(BM_STACK_RDSET, next_read_buf_id); // change bm_read_limit
            SETREG(BM_STACK_RESET, 0x02); // change bm_read_limit

            g_ftl_read_buf_id = next_read_buf_id;
        

        sect_offset = 0;
        sectors_remain -= num_sectors_to_read;
        lpage_addr++;
    } // ? end while sectors_remain != 0 ?
} // ? end ftl_read ?
```

# Write Operation

- `./ftl_dummy/ftl.c`

```
void ftl_write(UINT32 const lba, UINT32 const total_sectors)
{
    UINT32 num_sectors_to_write;

    UINT32 sect_offset = lba % SECTORS_PER_PAGE;
    UINT32 remain_sectors = total_sectors;

    while (remain_sectors != 0)
    {
        if (sect_offset + remain_sectors >= SECTORS_PER_PAGE)
        {
            num_sectors_to_write = SECTORS_PER_PAGE - sect_offset;
        }
        else
        {
            num_sectors_to_write = remain_sectors;
        }

        while (g_ftl_write_buf_id == GETREG(SATA_WBUF_PTR)); // bm_write_limit should not outpace SATA_WBUF_PTR
        g_ftl_write_buf_id = (g_ftl_write_buf_id + 1) % NUM_WR_BUFFERS; // Circular buffer

        SETREG(BM_STACK_WRSET, g_ftl_write_buf_id); // change bm_write_limit
        SETREG(BM_STACK_RESET, 0x01); // change bm_write_limit

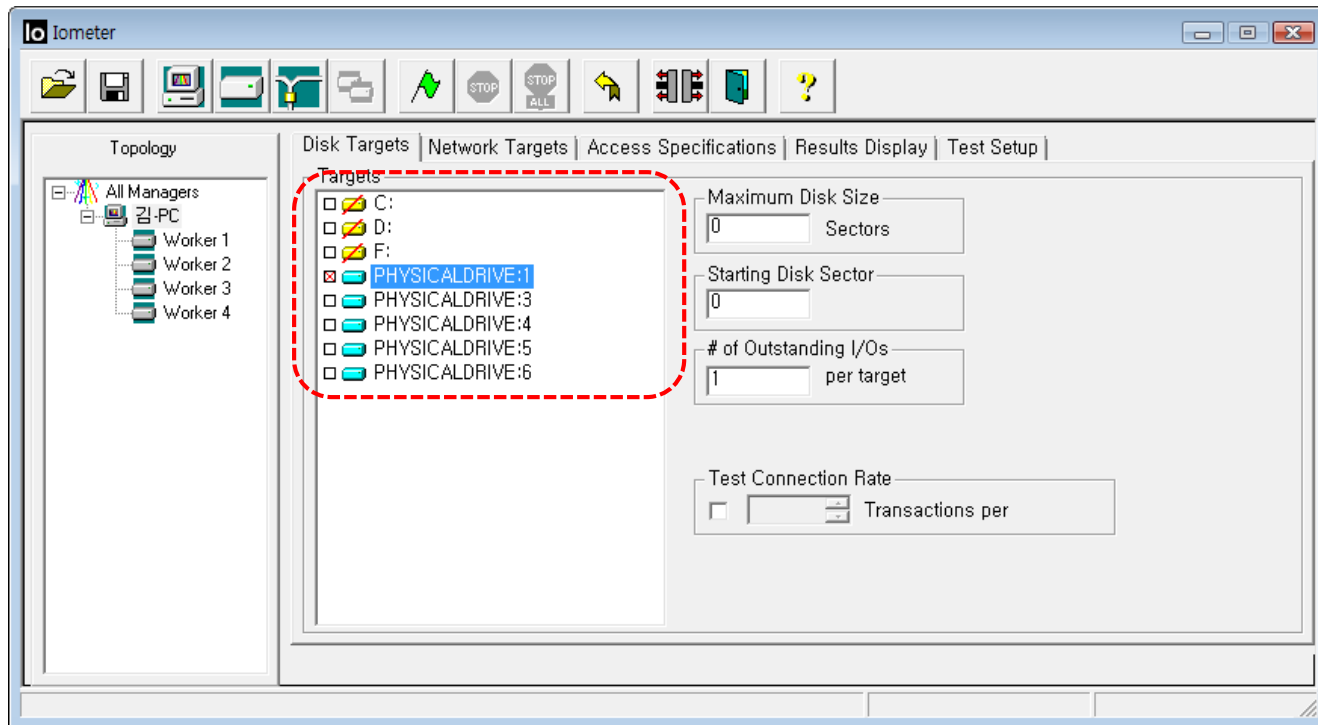
        sect_offset = 0;
        remain_sectors -= num_sectors_to_write;
    } ? end while remain_sectors != 0 ?
} ? end ftl_write ?
```

# IOmeter

- Performance measurement tool for storage system
  - Perform I/Os accessing a file or block device
  - <http://www.iometer.org>
- Performance is measured in
  - IOPS (IOs Per Second)
  - MB/s (Mega Bytes Per Second)
  - Response time

# Example

- Install & Run Iometer
- Select disk target



# Example

- Make new access specification
  - Access Specifications -> New

**Edit Access Specification**

Name: Sequential Write 128KB | Default Assignment: None

Size	% Access	% Read	% Random	Delay	Burst	Alignment	Reply
0MB 128KB 0B	100	0	0	0	1	sector	none

Transfer Request Size: 0 Megabytes, 128 Kilobytes, 0 Bytes

Percent of Access Specification: 100 Percent

Percent Read/Write Distribution: 100% Write, 0% Read

Percent Random/Sequential Distribution: 100% Sequential, 0% Random

Burstiness: Transfer Delay: 0 ms, Burst Length: 1 I/Os

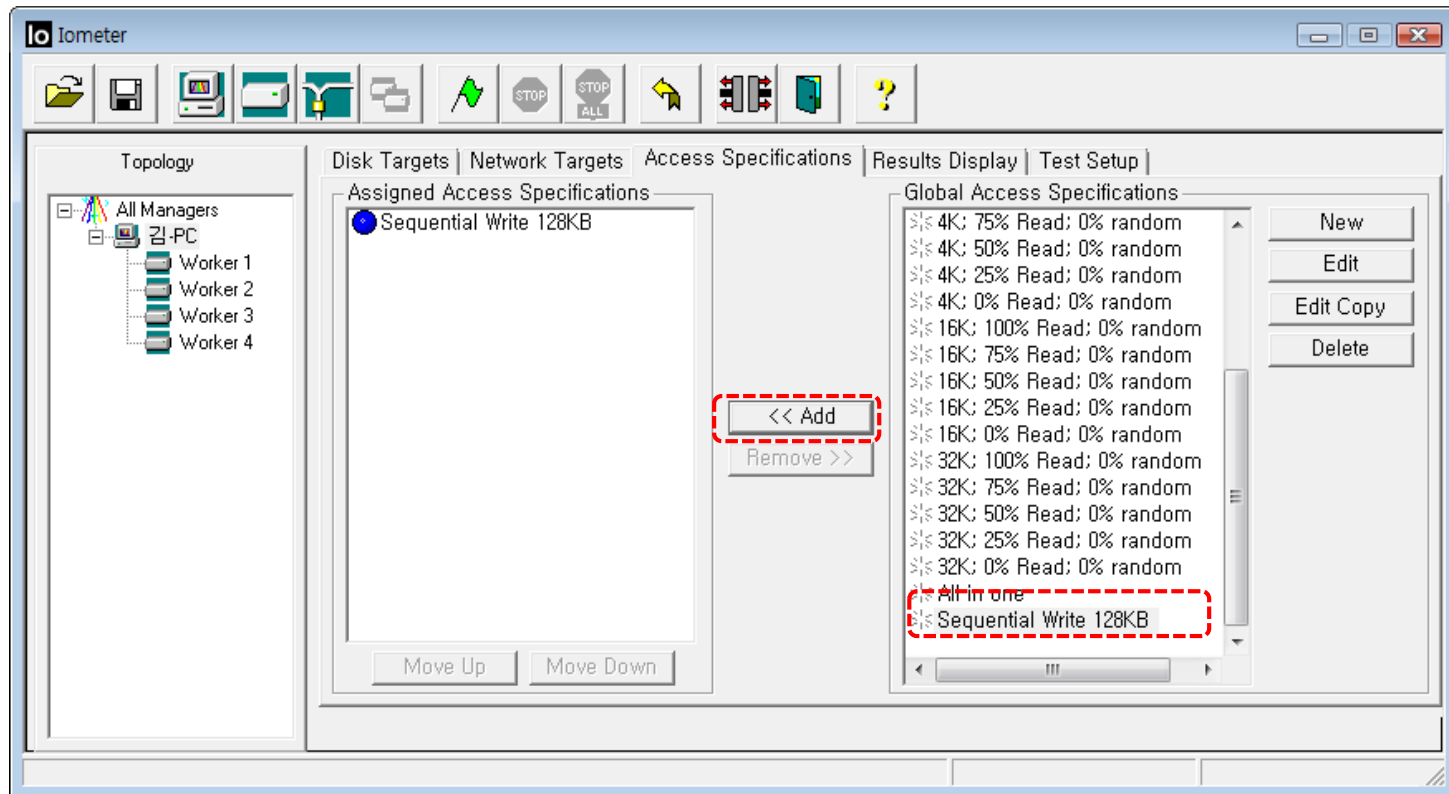
Align I/Os on:  Sector Boundaries,  0 Megabytes,  0 Kilobytes,  512 Bytes

Reply Size:  No Reply,  0 Megabytes,  128 Kilobytes,  0 Bytes

Buttons: Insert Before, Insert After, Delete, OK, Cancel

# Example

- Assign access specification
  - Select access specification and “Add”





# Example

- Start Tests
  - Results Display -> Click 'Flag Icon'

The screenshot shows the Iometer software interface. The 'Results Display' tab is active, showing a table of performance metrics for 'All Managers'. A red dashed box highlights the 'Flag Icon' in the toolbar and the 'Results Since' and 'Update Frequency' controls in the Results Display tab.

Display	All Managers		
Total I/Os per Second	1864,40	10000	>
Total MBs per Second	233,05	1000	>
Average I/O Response Time (ms)	0,5358	10	>
Maximum I/O Response Time (ms)	0,5802	10	>
% CPU Utilization (total)	3,85 %	10 %	>
Total Error Count	0	10	>

Run 1 of 1

Any Questions?