Western Digital.

WHITE PAPER DECEMBER 2021

Trim Command – General Benefits for Hard Disk Drives

Author | Wes McMillen, Western Digital Corporation

Contents

Introduction

The purpose of a trim command is to allow the host to provide LBA/data usage information that allows the storage device to become more performant. The trim command provides input from the host to let the HDD know what sectors no longer contain user data. Thus, trimming data allows the HDD to minimize background data movement and other disk background activities that can impact host performance. Executing the trim command during opportune scenarios allows for additional host features such as additional cameras, auxiliary streams, picture streams, and playback streams.

Requested Trim Scenarios

HDD Initialization

Western Digital requests the host execute a TRIM across the entire range of the drive using the DATA SET MANGEMENT XL's TRIM operation when the system is going through an HDD initialization operation.

This step is critical if the user reuses a drive that was previously in an overstressed environment where its performance may have been degraded. The TRIM operation will reset and re-initialize the drive to a clean state.

Opening of an Allocation Unit (AU)

When the system opens a new allocation unit to write IPC data, Western Digital requests that the LBA range for the allocation unit be trimmed. If a sub-stream/auxiliary stream requires opening a new allocation unit, a TRIM of the range for the auxiliary/sub-stream is also required.

Executing the TRIM over allocation units before they are used allows the device to clean up any old data that was left over when the allocation unit is reused. This enables a significant reduction in background data movement which frees up more disk bandwidth for more features such as additional cameras, auxiliary streams, picture streams, and playback streams. Even when the device has been previously completely trimmed, any write to an allocation unit must first be preceded by a trim for the AU.

Summary of Trim XL and AU suggestions:

- All stream data writes in a host AU must be preceded by a Trim (XL) command.
- Host OS meta data areas do not need a trim.
- Trim (XL) the entire drive before performing high level system format.
- Trim LBA and range must be 4K aligned.
- Only use on range for the trim of the AU (not multiple ranges that combined equal the AU).

Additional host OS suggestions:

- Use a large Allocation Unit (AU).
  Recommend at least 1GiB/AU. 2GiB would be preferable.
- Always allocate a full AU per stream.
  Do not partially allocate AUs at the end of the HDD Lba space.
- AUs must represent the same Lba space after every wrap.
- Co-locate all video meta-data in the same video stream AU LBA space.
- Co-locate all system level meta-data in a confined area in the lower range of the LBA space.
- Locate video meta-data as a sequential stream (either forward or reverse).
- Do not backup and rewrite LBAs in an AU.
  Write the video streams purely sequential. It is understood that the video meta-data

streams will typically rewrite data and this practice is acceptable.
- Use a large and consistent block count sizes for the video stream write commands. This allows the HDD to identify this as a video record write command.
- Use significantly smaller block count sizes for meta-data write commands.
  This allows the HDD to identify this write data as meta-data.
- Write all streams (video/aux/picture) forward sequentially. The exception for this rule is a reverse video meta-data stream.
- Take care on placement of buzzers in the SVR box used for alerts. Improper placement or buzzer frequency could impact drive performance. In general, eliminate vibration as much as possible.
- Limit use of ATA Flush commands to the bare minimum.
- Incorporate the use of the ATA NOP command to inform the HDD of a system buffer under-run/over-run condition. This helps greatly with system validation and debugging.
- Incorporate the use of ATA security (password) for unlocking and system identification for the HDD.
- All data must be 4K aligned for optimum performance.
- NCQ_6G SATA support.
- Trim LBA and range must be 4K aligned.

TRIM DSM (Legacy) vs TRIM DSM XL

Trim is supported by either legacy TRIM Dataset Management (Opcode 06h) or TRIM Dataset Management XL (Opcode 07h). The main problem with the legacy TRIM DSM is that the TRIM RANGE LENGTH is limited to 16-bit vs 64-bit for TRIM DSM XL. Trimming large amount of data potentially requires hundreds if not thousand instances of legacy TRIM DSL commands to be issued vs one or a few for TRIM DSM XL. HDD processing of thousand instances of either trim ranges or legacy TRIM DSM can add tens of seconds to the trimming process. For this reason, WDC requests the host to implement TRIM with DSM XL.

RAID Rebuild

A RAID rebuild is the data reconstruction process that occurs when a hard disk drive needs to be replaced. A Trim command must be issued to restore original performance.

Implementation

The Trim functionality should be implemented in Linux Application Layer. In the Linux application boot-up, it should check if Data Set Management XL (DSM XL) trim is supported. The support flag for this drive should be stored in a global variable in the Application Layer.

Data Set Management XL (DSM XL) support bit is specified in ACS-4 documentation. To check if DSM XL is supported, query Log Address 30H (Identify Device Data Log), Page 03H (Supported Capabilities), Qword (byte offset 8–15), Bit 50.

Before the application sends a DSM XL Trim command to a drive, it should check the global variable to see if the drive supports trim.

At the end of this document is sample code for trim.
To check if the DSM XL Trim command is supported, please refer to the function **is_dsm_xl_supported()**.

Important: It is strongly recommended to check the feature support once when the application starts, rather than checking the feature support every time a trim is issued. This will reduce unnecessary overhead.

On a HDD hot-plug condition, the DSM XL support bit must be rechecked and the global variable re-initialized.

DSM XL is a non-queued command. If the OS is operating in an NCQ_ environment, all of the queued commands must complete before this non-queued command is issued. Otherwise, there will be a queue command intermix error.

Error Handling

Check for DSM XL support before issuing the command to prevent the OS from executing exception handling. This can occur due to an error condition generated for drives that do not support DSM XL Trim.

On a DSM XL Trim command error, the application layer may choose to dump failure information, such as if DSM XL Trim is supported, Trim Start LBA and Trim length.

Identify Device

7.1 Log Address 30H (Identify Device Data)

Log Address	Log Name	Feature Set	Support	R/W	Access
1Ah.. 1Fh	Reserved				
20h	Obsolette				
21h	Write Stream Error Log	Streaming	F	RO	GPL [b]
22h	Read Stream Error Log	Streaming	F	RO	GPL [b]
23h	Obsolete				
24h	Current Device Internal Status Data log	None	O	RO	GPL [b]
25h	Saved Device Internal Status Data log	None	O	RO	GPL [b]
26h.. 2Eh	Reserved				
2Fh	Set Sector Configuration	None		RO	GPL [b]
30h	IDENTIFY DEVICE data	None	M	RO	GPL, SL
31h.. 7Fh	Reserved				
80h.. 9Fh	Host Specific	SMART	M	R/W	GPL, SL
A0h.. DFh	Device Vendor Specific	SMART	O	VS	GPL, SL
E0h	SCT Command/Status	SCT	F	R/W	GPL, SL
E1h	SCT Data Transfer	SCT	F	R/W	GPL, SL
E2h.. FFh	Reserved				

7.2 Log Address 30H (Identify Device Data)

Page	Description	Required
00h	List of supported pages (see 9.10.2)	M
01h	Copy of IDENTIFY DEVICE data (see 7.13.6)	M
02h	Capacity (see 9.10.4)	M
03h	Supported Capacbilityes (see 9.10.5)	M
04h	Current Settings (see 9.10.6)	M
05h	ATA Strings (see 9.10.7)	M
06h	Security (see 9.10.8)	M
07h	Parallel ATA (see 9.10.9)	P
08h	Serial ATA (see 9.10.9)	S
09h	Reserved for ZAC	See ZAC
0Ah.. FFh	Reserved	

7.3 Supported Capabilities Page (DSM XL Support Bit)

Offset	Type	Content	
8.. 15	QWord	Supported Capabilities	
		Bit	**Description**
		63	Shall be set to one
		62:55	Reserved
		54	ADVANCED BACKGROUND OPERATION SUPPORTED bit
		53	PERSISTENT SENSE DATA REPORTING bit
		52	SFF-8447 REPORTING bit
		51	DEFINITIVE ENDING PATTERN SUPPORTED bit
		50	DATA SET MANAGEMENT XL SUPPORTED bit
		49	SET SECTOR CONFIGURATION SUPPORTED bit
		48	ZERO EXT SUPPORTED bit
		47	SUCCESSFUL NCQ COMMAND SENSE DATA SUPPORTED bit
		46	DLC SUPPORTED bit
		45	REQUEST SENSE DEVICE FAULT SUPPRORTED bit

Data Set Management XL

The Data Set Management XL command (**ATA Opcode 07h**) provides information (e.g., file system information) that the device may use to optimize its operations.

Inputs

See the following table for a Data Set Management XL Trim Command:

Field	Description	
FEATURE	**Bit**	**Description**
	15:8	DSM Function Field
	7:1	Reserved
	0	TRIM bit
COUNT	Number of 512-byte blocks to be transferred. The value zero is reserved.	
LBA	If the TRIM bit is set to one, reserved. If the TRIM bit is cleared to zero, defined by the DSM FUNCTION field.	
DEVICE	**Bit**	**Description**
	7	Obsolete
	6	N/A
	5	Obsolete
	4	Transport Dependent
	3:0	Reserved
COMMAND	7:0	07h

Host Inputs to Device Data Structure

Data Set Management XL Request Data is a list of one or more XL LBA Range Entry pages (see table below).

If the TRIM bit is set to one, individual XL LBA Range Entries may specify LBA ranges that overlap and are not required to be sorted.

XL LBA Range Entry

Offset	Type	Description	
0.. 15	DQWord	Entry 0	
		Bit	**Description**
		127:64	RANGE LENGTH field
		63:48	Reserved
		74:0	LBA VALUE field
16.. 31	DQWord	Entry 1	
		Bit	**Description**
		127:64	RANGE LENGTH field
		63:48	Reserved
		47:0	LBA VALUE field
...		...	
496..511	DQWord	Entry 31	
		Bit	**Description**
		127:64	RANGE LENGTH field
		63:48	Reserved
		47:0	LBA VALUE field

RANGE LENGTH Field

The RANGE LENGTH field specifies the number of logical sectors in the LBA range. If the RANGE LENGTH field is set to 0h, the entry shall be ignored.

LBA VALUE Field

The LBA Value field specifies the starting LBA of the LBA range. If the LBA VALUE plus the range is greater than the accessible capacity, the device shall return command aborted.

Sample Source Code

/* Sample code how to check if DSM XL is supported. It issues a DSM XL to trim the entire HDD if DSM XL is supported.

The function is_dsm_xl_supported() returns 1 (if DSM XL is supported) and 0 (if DSM XL is not supported). This state is then saved. Next time a trim issued, the application will check against the saved state versus issuing the Read Log command again.

The check for DSM XL supported should be part of the Application Bootup process.

*/

Sample Source Code:

```
#include <ctype.h>
#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <scsi/sg.h>
#include <unistd.h>

// See ACS-4 DATA SET MANAGEMENT XL command
typedef struct dsm_trim_xl_range_entry_t {
    uint64_t lba_value; // LBA VALUE field - little
endian
    uint64_t range_length; // RANGE LENGTH field -
little endian
} dsm_trim_xl_range_entry;

#define DSM_TRIM_XL_ENTRY_SIZE (16)
#define ATA_SECTOR_SIZE (512)
#define SGIO_TIMEOUT (1000 * 60)
#define MAX_DEVICES (1024)

// global variables to store info per device
struct device_data_t {
    char path[FILENAME_MAX];
    uint64_t total_lbas;
    uint8_t supported;
};

static struct device_data_t _devices[MAX_DEVICES];
static uint32_t _device_count = 0;
```

```
/**
 * Swap bytes from NULL terminated string.
   Useful for ATA IDENTIFY DEVICE strings
 * parameters:
 *  str         NULL terminated string
 * returns:
 *  byte swapped string
 */

char *byteswap(char *str) {
    size_t i;

    if (!str) {
        return 0;
    }

    for (i = 0; i <= strlen(str) && i + 1 <= strlen(str);
i += 2) {
        char a, b;

        a = str[i];
        b = str[i + 1];
        str[i] = b;
        str[i + 1] = a;
    }

    return str;
}

/**
 * trim leading/trailing spaces from string
 * parameters:
 *  str         NULL terminated string
 * returns:
 *  trimmed string
 */

char *trim(char *str) {
    size_t len;
    int i;

    if (!str) {
        return 0;
    }

    for (i = 0; i <= (int)strlen(str) && str[i] == ' ';
++i);

    len = strlen(str + i);
    memmove(str, str + i, len);
    str[len] = 0;

    for (i = (int)strlen(str); i >= 0; i--) {
        char *c = str + i;
        if (isspace(*c) || *c == '\0') {
            *c = 0;
```

```
        } else {
            break;
        }
    }

    return str;
}
/**
 * Write unsigned 64-bit value to array - little endian
 * parameters:
 *  value       The value to put in buffer
 *  buffer      The address to store 64-bit value
 */
void le _ u64 _ to _ array(uint64 _ t value, uint8 _ t* buffer)
{
    int i;
    for (i = 0; i < 8; i++) {
        *buffer++ = value & 0xFF;
        value >>= 8;
    }
}

/**
 * Read 16-bit unsigned value from device data (IDENTIFY
DEVICE, READ LOG EXT etc)
 * parameters:
 *  word            Word offset to read
 *  buffer          Device data buffer
 *  size            Size of device data buffer
 * returns:
 *  16-bit unsigned value from the buffer
 */
uint16 _ t array _ to _ le _ u16(uint8 _ t word, const
uint8 _ t * const buffer, size _ t size) {
    if (!buffer || size == 0 || word * 2 + 1 >= size) {
        fprintf(stderr, "invalid parameter(s)\n");
        return 0;
    }
    return (uint16 _ t)buffer[word * 2]
        | ((uint16 _ t)buffer[word * 2 + 1] << 8);
}
/**
 * Read 32-bit unsigned value from device data (IDENTIFY
DEVICE, READ LOG EXT etc)
 * parameters:
 *  word            Word offset to read
 *  buffer          Device data buffer
 *  size            Size of device data buffer
 * returns:
 *  32-bit unsigned value from the buffer
 */
uint32 _ t array _ to _ le _ u32(uint8 _ t dword, const
uint8 _ t * const buffer, size _ t size) {
    if (!buffer || size == 0 || dword * 4 + 3 >= size) {
        fprintf(stderr, "invalid parameter(s)\n");
        return 0;
    }
    return (uint32 _ t)buffer[dword * 4]
        | ((uint32 _ t)buffer[dword * 4 + 1] << 8)
        | ((uint32 _ t)buffer[dword * 4 + 2] << 16)
        | ((uint32 _ t)buffer[dword * 4 + 3] << 24);
}

/**
 * Read 64-bit unsigned value from device data (IDENTIFY
DEVICE, READ LOG EXT etc)
 * parameters:
 *  word            Word offset to read
 *  buffer          Device data buffer
 *  size            Size of device data buffer
 * returns:
 *  64-bit unsigned value from the buffer
 */
uint64 _ t array _ to _ le _ u64(uint8 _ t qword, const
uint8 _ t * const buffer, size _ t size) {
    if (!buffer || size == 0 || qword * 8 + 7 >= size) {
        fprintf(stderr, "invalid parameter(s)\n");
        return 0;
    }
    return (uint64 _ t)buffer[qword * 8]
        | ((uint64 _ t)buffer[qword * 8 + 1] << 8)
        | ((uint64 _ t)buffer[qword * 8 + 2] << 16)
        | ((uint64 _ t)buffer[qword * 8 + 3] << 24)
        | ((uint64 _ t)buffer[qword * 8 + 4] << 32)
        | ((uint64 _ t)buffer[qword * 8 + 5] << 40)
        | ((uint64 _ t)buffer[qword * 8 + 6] << 48)
        | ((uint64 _ t)buffer[qword * 8 + 7] << 56);
}
/**
 * Use SG _ IO to send a SCSI command
 * https://www.tldp.org/HOWTO/SCSI-Generic-HOWTO/sg _ io _
hdr _ t.html
 * parameters:
 *  fd          Send command to this file descripton
 *  direction   SG _ IO transfer direction, usually: SG _
DXFER _ NONE, SG _ DXFER _ TO _ DEV or SG _ DXFER _ FROM _ DEV
 *  cdb         CDB buffer
 *  cdb _ sz      Size of CDB buffer
 *  buffer      Returns device data (0 for SG _ DXFER _
NONE)
 *  buffer _ sz   Size of buffer (0 for SG _ DXFER _ NONE)
 *  sense       Returns sense data
 *  sense _ sz    Size of sense buffer
 * returns:
 *  error       0 upon success
 */
int scsi _ command(int fd, int direction, uint8 _ t* cdb,
size _ t cdb _ sz,
    uint8 _ t* buffer, size _ t buffer _ sz, uint8 _ t* sense,
size _ t sense _ sz) {
    int ret;
    int i;
```

```
    sg_io_hdr_t sgio;

    // clear sense data
    memset(sense, 0, sense_sz);

    // Setup SG_IO pass-through
    memset(&sgio, 0, sizeof(sgio));
    sgio.interface_id = 'S';
    sgio.dxfer_direction = direction;
    sgio.dxfer_len = buffer_sz;
    sgio.dxferp = buffer;
    sgio.cmd_len = cdb_sz;
    sgio.cmdp = cdb;
    sgio.mx_sb_len = sense_sz;
    sgio.sbp = sense;
    sgio.timeout = SGIO_TIMEOUT;

    ret = ioctl(fd, SG_IO, &sgio);

    if (ret) {
        fprintf(stderr, "ioctl failed (ret = %d)\n", ret);
    }
    else if (sgio.status & 0x01 || sgio.driver_status) {
        fprintf(stderr, "device error (status = 0x%x;
driver_status = 0x%x)\nsense:",
            sgio.status, sgio.driver_status);
        for (i = 0; i < sgio.sb_len_wr; i++) {
            fprintf(stderr, " %02x", sense[i]);
        }
        fprintf(stderr, "\n");
        ret = 1;
    }
   return ret;
}

/**
 * Use SG_IO to send READ LOG EXT command to ATA
device.
 * parameters:
 *  fd              Send command to this file descripton
 *  log_address     ATA spec: log address
 *  page_number     ATA spec: page number
 *  buffer          Returns id_buffer device data
 *  buffer_sz       Size of buffer (should be a multiple
of 512)
 * returns:
 *  error           0 upon success
 */
int ata_read_log_ext(int fd, uint8_t log_address,
uint16_t page_number,
    uint8_t* buffer, size_t buffer_sz) {
    uint8_t cdb[16];
    uint8_t sense[32];
    uint16_t features = 0;
    uint16_t log_page_count = buffer_sz / 512;
    uint64_t lba = (((uint64_t)page_number & 0xFF00)
<< 24)
        | ((page_number & 0x00FF) << 8)
        | (log_address);

    if (fd < 0 || !buffer || buffer_sz < ATA_SECTOR_
SIZE) {
        fprintf(stderr, "invalid parameter(s)\n");
        return -1;
    }

    // Set CDB - SAT spec, section 12.2.2.3
    cdb[0] = 0x85;  // ATA pass-through (16)
    cdb[1] = 0x09;  // PROTOCOL (bit 4:1 = PIO Data-In
[4]); EXT command (bit 0 = 1)
    cdb[2] = 0x0E;  // OFFLINE (bit 7:6 = 0); CK_COND
(bit 5 = 0); T_TYPE (bit 4 = 0); T_DIR (bit 3 = 1);
BYTE_BLOCK (bit 2 = 1); T_LENGTH (bit 1:0 = 2)
    cdb[3] = (features >> 8) & 0xFF;        // FEATURES
(15:8)
    cdb[4] = features & 0xFF;               // FEATURES
(7:0
    cdb[5] = (log_page_count >> 8) & 0xFF;  // COUNT
(15:8)
    cdb[6] = log_page_count & 0xFF;         // COUNT
(7:0)
    cdb[7] = (lba >> 24) & 0xFF;            // LBA (31:24)
    cdb[8] = lba & 0xFF;                    // LBA (7:0)
    cdb[9] = (lba >> 32) & 0xFF;            // LBA (39:32)
    cdb[10] = (lba >> 8) & 0xFF;            // LBA (15:8)
    cdb[11] = (lba >> 40) & 0xFF;           // LBA (47:40)
    cdb[12] = (lba >> 16) & 0xFF;           // LBA (23:16)
    cdb[13] = 0x00;                         // DEVICE
    cdb[14] = 0x2f;                         // COMMAND =
READ LOG EXT
    cdb[15] = 0x00;

    return scsi_command(fd, SG_DXFER_FROM_DEV, cdb,
sizeof(cdb), buffer, buffer_sz, sense, sizeof(sense));
}

/**
 * Use SG_IO to send IDENTIFY DEVICE command to ATA
device.
 * parameters:
 *  fd          Send command to this file descripton
 *  buffer      Returns id_buffer device data
 *  buffer_sz   Size of buffer (must be at least 512
bytes)
 *
 * returns:
 *  error       0 upon success
 */
int ata_identify_device(int fd, uint8_t* buffer,
size_t buffer_sz) {
    uint8_t cdb[12];
    uint8_t sense[32];
```

```
    if (fd < 0 || !buffer || buffer _ sz < ATA _ SECTOR _
SIZE) {
        fprintf(stderr, "invalid parameter(s)\n");
        return -1;
    }

    // Set CDB - SAT spec, section 12.2.2.2
    cdb[0] = 0xA1;  // ATA pass-through (12)
    cdb[1] = 0x08;  // PROTOCOL (bits 4:1 = PIO Data-In[4])
    cdb[2] = 0x0E;  // T _ DIR (bit 3 = 1); BYTE _ BLOCK
(bit 2 = 1); T _ LENGTH (bits 1:0 = 2)
    cdb[3] = 0x00;  // FEATURES (7:0)
    cdb[4] = 0x01;  // COUNT (7:0)
    cdb[5] = 0x00;  // LBA (7:0)
    cdb[6] = 0x00;  // LBA (15:8)
    cdb[7] = 0x00;  // LBA (23:16)
    cdb[8] = 0x00;  // DEVICE
    cdb[9] = 0xEC;  // COMMAND = IDENTIFY DEVICE
    cdb[10] = 0x00; // reserved
    cdb[11] = 0x00; // CONTROL

    return scsi _ command(fd, SG _ DXFER _ FROM _ DEV, cdb,
sizeof(cdb), buffer, buffer _ sz, sense, sizeof(sense));
}

/**
 * Use SG _ IO to send ATA data set management trim XL
command to ATA device.
 * parameters:
 *  fd          Send command to this file descripton
 *  entry       Pointer to a dsm _ trim _ xl _ range _ entry
array
 *  entry _ size  The number of valid entries
 *
 * returns:
 *  error       0 upon success
 */
int ata _ trim _ xl(int fd, dsm _ trim _ xl _ range _ entry *
entry, int entry _ size) {
    int ret;
    int i;
    uint8 _ t cdb[16];
    uint8 _ t sense[32];
    uint8 _ t* buffer;
    size _ t buffer _ sz;
    uint64 _ t lba = 0;
    uint16 _ t count;

    if (fd < 0 || !entry || !entry _ size) {
        fprintf(stderr, "invalid parameter(s)\n");
        return -1;
    }

    // Prepare buffer for DSM TRIM XL
    count = ((entry _ size * DSM _ TRIM _ XL _ ENTRY _ SIZE)
+ ATA _ SECTOR _ SIZE - 1) / ATA _ SECTOR _ SIZE;
    buffer _ sz = count * ATA _ SECTOR _ SIZE;
    buffer = malloc(buffer _ sz);
    if (!buffer) {
        fprintf(stderr, "out of memory\n");
        return -1;
    }
    memset(buffer, 0, buffer _ sz);

    for (i = 0; i < entry _ size; i++) {
        le _ u64 _ to _ array(entry[i].lba _ value, buffer + i
* DSM _ TRIM _ XL _ ENTRY _ SIZE);
        le _ u64 _ to _ array(entry[i].range _ length, buffer
+ sizeof(uint64 _ t) + i * DSM _ TRIM _ XL _ ENTRY _ SIZE);
    }

    // Set CDB - SAT spec, section 12.2.2.3
    cdb[0] = 0x85;  // ATA pass-through (16)
    cdb[1] = 0x0D;  // PROTOCOL (bit 4:1 = DMA[6]); EXT
command (bit 0 = 1)
    cdb[2] = 0x02;  // OFFLINE (bit 7:6 = 0); CK _ COND
(bit 5 = 0); T _ TYPE (bit 4 = 0); T _ DIR (bit 3 = 0);
BYTE _ BLOCK (bit 2 = 0); T _ LENGTH (bit 1:0 = 2)
    cdb[3] = 0x00;  // FEATURES (bit 15:8 = 0)
    cdb[4] = 0x01;  // FEATURES (bit 0 = TRIM[1])
    cdb[5] = (count >> 8) & 0xFF;    // COUNT (15:8)
    cdb[6] = count & 0xFF;           // COUNT (7:0)
    cdb[7] = (lba >> 24) & 0xFF;     // LBA (31:24)
    cdb[8] = lba & 0xFF;             // LBA (7:0)
    cdb[9] = (lba >> 32) & 0xFF;     // LBA (39:32)
    cdb[10] = (lba >> 8) & 0xFF;     // LBA (15:8)
    cdb[11] = (lba >> 40) & 0xFF;    // LBA (47:40)
    cdb[12] = (lba >> 16) & 0xFF;    // LBA (23:16)
    cdb[13] = 0x00; // DEVICE
    cdb[14] = 0x07; // COMMAND = DATA SET MANAGEMENT XL
    cdb[15] = 0x00;

    ret = scsi _ command(fd, SG _ DXFER _ TO _ DEV, cdb,
sizeof(cdb), buffer, buffer _ sz, sense, sizeof(sense));
    free(buffer);
    return ret;
}

/**
 * Query device to determine if DATA SET MANAGEMENT XL
command is supported
 * parameters. It shall read log address 0x30, log page
3, and parse QWord 1,
 * bit 50.
 *  fd          Send command to this file descripton
 * returns:
 *  supported   1 if supported, 0 if not supported
 */
uint8 _ t is _ dsm _ xl _ supported(int fd) {
    uint8 _ t supported = 0;
    uint8 _ t buffer[ATA _ SECTOR _ SIZE] = { 0 };
```

```
    uint64_t supported_capabilities;

    if (!ata_read_log_ext(fd, 0x30, 3, buffer, ATA_
SECTOR_SIZE)) {
        supported_capabilities = array_to_le_u64(1,
buffer, ATA_SECTOR_SIZE);
        supported = (supported_capabilities >> 50) & 1;
    }
    return supported;
}

/**
 * Trim entire LBA range of device path
 * parameters
 *  data        device_data_t structure
 * return
 *  error       0 upon success
 *              1 if DSM XL trim command fails
 *              2 if path could not be opened
 *              -1 if DSM XL is not supported
 */
int trim_entire_drive(struct device_data_t *data) {
    dsm_trim_xl_range_entry entry[1];
    int ret = 0;
    int fd;
    int i;

    if (data->supported) {
        fd = open(data->path, O_RDONLY);
        if (fd < 0) {
            fprintf(stderr, "failed to open %s\n", data-
>path);
            return 2;
        }

        entry[0].lba_value = 0;
        entry[0].range_length = data->total_lbas;

        printf("Trim XL on %s\n", data->path);
        printf("START LBA      RANGE LENGTH\n");
        printf("-------------- --------------\n");
        for (i = 0; i < sizeof(entry) / sizeof(entry[0]);
i++) {
            printf("0x%012lx 0x%012lx\n", entry[i].lba_
value, entry[i].range_length);
        }

        if (!ata_trim_xl(fd, entry, sizeof(entry) /
sizeof(entry[0]))) {
            printf("DSM Trim XL ok.\n");
        } else {
            ret = 1;
        }
        close(fd);
    } else {
        printf("DSM Trim XL is not supported.\n");
        ret = -1;
    }
    return ret;
}

/**
 * Open path and fetch total LBAs and DSM XL supported
 * parameters
 *  path        The device path name (eg /dev/sda)
 * returns
 *  error       0 upon success
 *              1 failed to open path
 *              2 failed to get LBAs/DSM XL support
 *              3 Too many devices queried
 */
int query_device(const char* const path) {
    int ret;
    int fd;
    int found;
    uint32_t i;
    uint8_t id_buffer[ATA_SECTOR_SIZE] = { 0 };
    uint64_t lbas;
    char model[41] = { 0 };
    char serial[21] = { 0 };
    uint8_t supported;

    fd = open(path, O_RDONLY);
    if (fd < 0) {
        fprintf(stderr, "failed to open %s\n", path);
        return 1;
    }

    ret = ata_identify_device(fd, id_buffer,
sizeof(id_buffer));
    if (ret) {
        fprintf(stderr, "Error: identify device failed
(%s)\n", path);
        close(fd);
        return 2;
    }


    // serial number at word 10, 20 bytes
    memcpy(serial, id_buffer + 10 * 2, 20);
    serial[20] = 0;
    byteswap(serial);
    trim(serial);
    // model number at word 27, 40 bytes
    memcpy(model, id_buffer + 27 * 2, 40);
    model[40] = 0;
    byteswap(model);
    trim(model);
    // total LBAs at word 100, 8 bytes
    lbas = array_to_le_u64(0, id_buffer + 100 * 2, 8);

    printf("Path:   %s\n", path);
```

```
    printf("Model:  %s\n", model);
    printf("Serial: %s\n", serial);
    printf("LBAs:   %lu\n", lbas);

    supported = is_dsm_xl_supported(fd);
    close(fd);

    found = 0;
    for (i = 0; i < _device_count; i++) {
        if (strcmp(_devices[i].path, path) == 0) {
            found = 1;
            _devices[i].supported = supported;
            _devices[i].total_lbas = lbas;
            break;
        }
    }
    if (!found) {
        // add it, if and only if there's enough space
        if (_device_count < MAX_DEVICES) {
            memset(_devices[_device_count].path, 0,
FILENAME_MAX);
            strncpy(_devices[_device_count].path,
path, FILENAME_MAX - 1);
            _devices[_device_count].supported =
supported;
            _devices[_device_count].total_lbas =
lbas;
            ++_device_count;
        } else {
            fprintf(stderr, "too many devices (max is
%u)\n", MAX_DEVICES);
            return 3;
        }
    }
    return 0;
}

/**
 * Trim all lbas on one device (passed in via command
line argument).
 */
int main(int argc, char* argv[]) {
    int ret;
    if (argc < 2) {
        fprintf(stderr, "missing device path\n");
        return 128;
    }
```

```
    ret = query_device(argv[1]);
    if (!ret) {
        ret = trim_entire_drive(&_devices[0]);
    }
    if (ret) {
        return 1;
    }
    return 0;
}
```

Western Digital.

5601 Great Oaks Parkway
San Jose, CA 95119, USA
US (Toll-Free): 800.801.4618
International: 408.717.6000

www.westerndigital.com