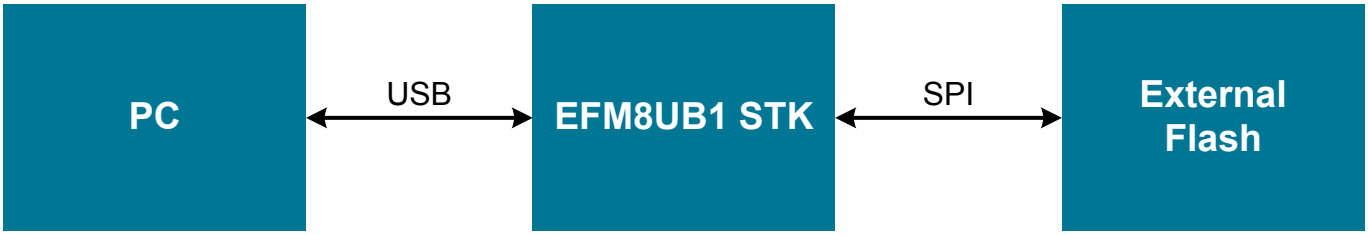
AN1047: EFM8 NOR Flash USB MSD Reference Design

This application note describes the implementation of a USB Mass Storage Device with an external NOR-Flash memory.

A method of logical to physical address mapping is also implemented in this reference design. The full source code is included, which contains information about USB Mass Storage Device class specification, SCSI, and the logical to physical transition for EFM8UB1.

KEY POINTS

- USB Mass Storage Device Class
- Logical to Physical address mapping
- NOR-Flash application



1. Introduction

The USB Mass Storage Device (MSD) class is one of the most widely supported device classes among the USB device classes natively supported by popular operating systems. A USB device that supports this class can use the built-in drivers provided by the operating system without the need to install or maintain any custom device drivers. The USB MSD Reference Design utilizes this widespread support by providing device firmware for Silicon Labs USB microcontrollers that complies with the MSD class specification. In addition, as any page of the NOR-Flash to be programmed must be erased first, a method of logical to physical address mapping is implemented in the firmware to improve the performance and reliability of NOR-Flash arrays.

This document describes the various components of the device firmware in detail.

Note: This example builds upon the USB Mass Storage Reference Design described in *AN282: USB Mass Storage Device Reference Design Programmer's Guide*. See this document for additional information.

2. References

The following specifications or standards were used as references for this design:

- *AN282: USB Mass Storage Device Reference Design Programmer's Guide*
- *Universal Serial Bus Specification*—Revision 2.0, December 21, 2000.
- *Universal Serial Bus Mass Storage Device Class Bulk-Only Transport*—Revision 1.0, September 31, 1999
- *SCSI Architecture Model - 3 (SAM-3)*—Revision 9, September 12, 2003
- *SCSI Block Commands - 2 (SBC-2)*—Revision 10, September 13, 2003
- *SCSI Primary Commands - 3 (SPC-3)*—Revision 17, January 28, 2004

3. Basic Overview

The MSD reference design hardware consists of two boards: a EFM8UB1 Starter Kit (STK) board and the NOR-Flash Memory Expansion board. The first figure below shows the EFM8UB1 STK board. The second figure shows a block diagram with the connections between the hardware components.

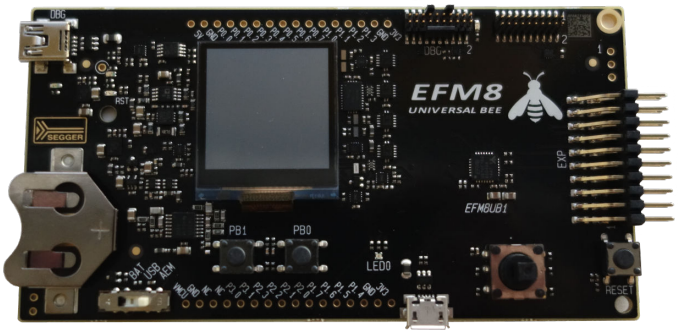


Figure 3.1. EFM8UB1-SLSTK2000A

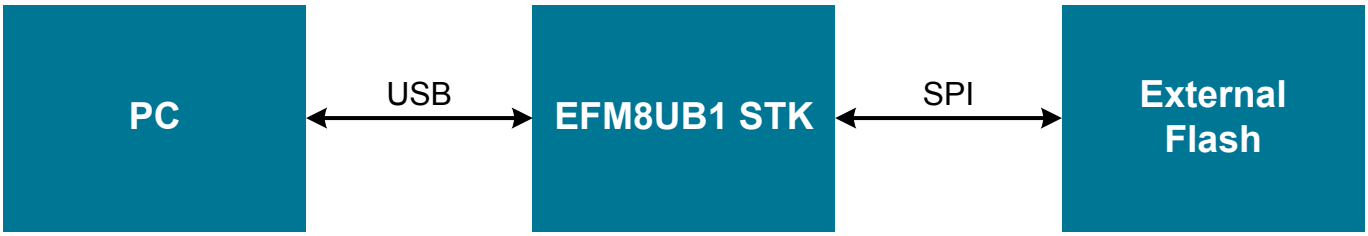


Figure 3.2. Hardware Overview

The USB MSD RD firmware consists of many distinct blocks that work together. The overall system architecture is shown in the figure below.

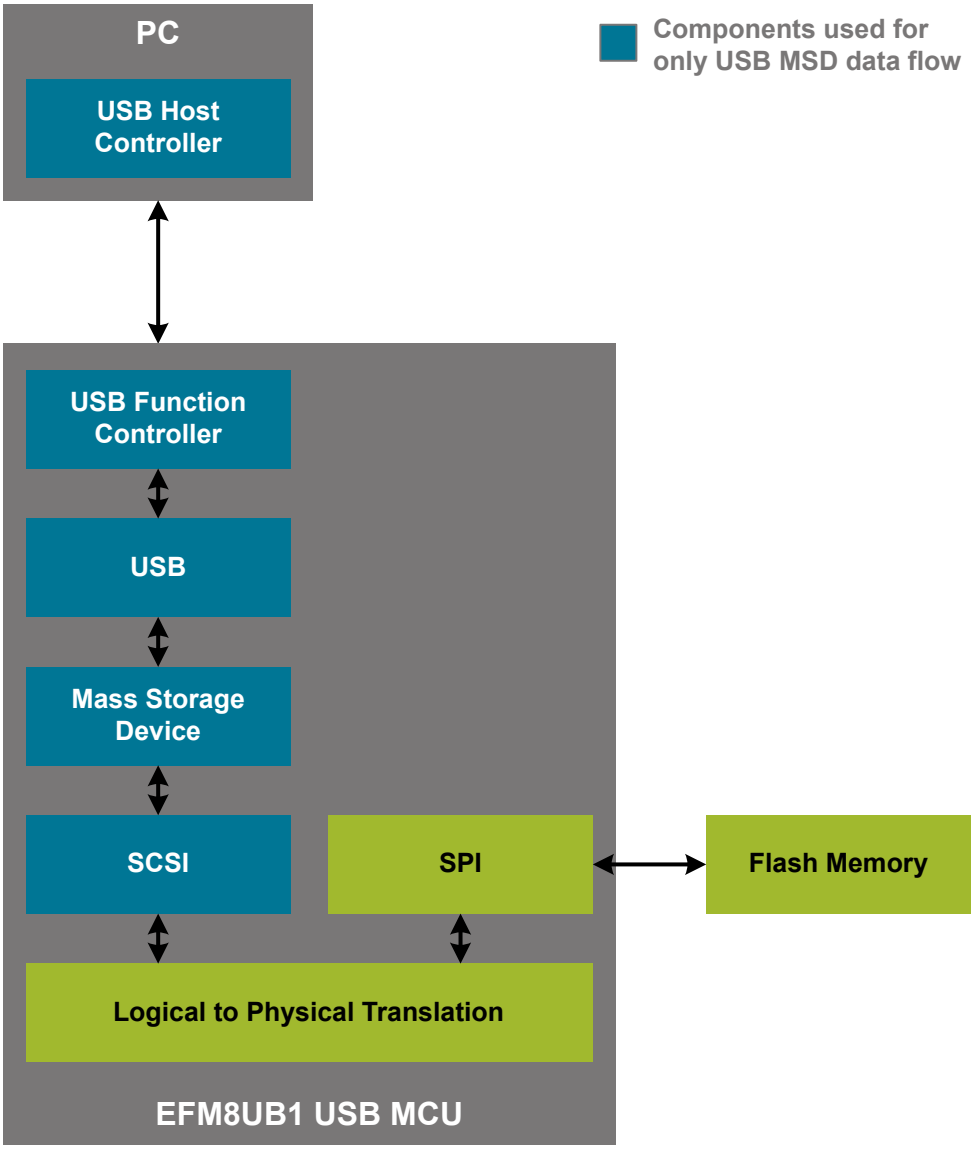


Figure 3.3. USB MSD RD System Architecture

The firmware is designed using USB Mass Storage Device class specification, and the device appears as a USB Mass Storage Device on the PC after connecting to the PC via USB. The firmware uses USB, Mass Storage Device, SCSI, and Logical to Physical Transition components. Each component is explained in detail in the following sections.

4. USB MSD RD Firmware Components

4.1 USB Low-level Interface

The USB descriptor is an important component of the USB low-level interface. USB enumeration is initiated when a USB device is plugged into a USB host, during which USB descriptors are requested by the host to determine the capabilities and requirements of the device. The information contained in the descriptor allows the host to load the appropriate device drivers and allocate power to the device, if requested. See 8. Appendix A—MSD RD USB Descriptor Details for more information about the USB descriptors used in the USB MSD RD firmware.

The tasks and capabilities of the USB Low-level Interface are listed below:

- Loads data into the IN endpoint FIFO.
- Reads data from the OUT endpoint FIFO.
- Handles USB bus conditions: Suspend, Resume, and Reset.
- Sends a STALL when the host sends an unsupported command.
- Handles the USB Standard Requests that are listed below:
  - GET_STATUS
  - CLEAR_FEATURE
  - SET_FEATURE
  - SET_ADDRESS
  - GET_DESCRIPTOR
  - GET_CONFIGURATION
  - SET_CONFIGURATION
  - GET_INTERFACE
  - SET_INTERFACE

The USB low-level interface functions are internal functions to the MSD RD and need not be called directly from the Application-level firmware. Refer to the `lib\efm8_usb\src` module for these functions and their descriptions.

4.2 MSD Class Command Interpreter

The Mass Storage Device Class Command Interpreter communicates directly with the hardware USB data endpoints (IN and OUT) that are managed by the USB Low-level Interface code. Note that the USB control endpoint traffic is handled by the USB Low-level Interface and is not seen by the MSD Class command interpreter.

The *MSD Class - Bulk Only Transport* specification defines two structures that are used for reliable Command Transport and Status Transport.

4.2.1 Command Block Wrapper (CBW)

The Command Block Wrapper (CBW) is defined as a packet containing a command block and the associated information. See Figure 4.1 MSD Class—Command/Data/Status Flow on page 6 for the format of this structure.

Table 4.1. Command Block Wrapper (CBW) Format

	7	6	5	4	3	2	1	0
0–3	dCBWSignature = 0x43425355							
4–7	dCBWTag							
8–11	dCBWDataTransferLength							
12	bmCBWFlags (including direction bit)							
13	Reserved (0)					bCBWLUN		
14	Reserved (0)					bCBWCBLength (1..16)		
15–30	CBWCB (contains SCSI command)							

4.2.2 Command Status Wrapper (CSW)

The Command Status Wrapper (CSW) is defined as a packet containing the status of a command block.

Table 4.2. Command Status Wrapper (CSW) Format

	7	6	5	4	3	2	1	0
0–3	dCSWSignature (= 0x53425355)							
4–7	dCSWTag (=identical as dCBWTag)							
8–11	dCSWDataResidue (=dCBWDataTransferLength – number of bytes processed)							
12	bCSWStatus (=Good, Fail or Phase Error)							

4.2.3 Command Interpreter State Machine

The command interpreter implements a simple state machine that is very similar to the Command/Data/Status Flow shown in the figure below. In the state machine implementation, the 'Data - In' and 'Data - Out' stages in this figure have been combined into one state.

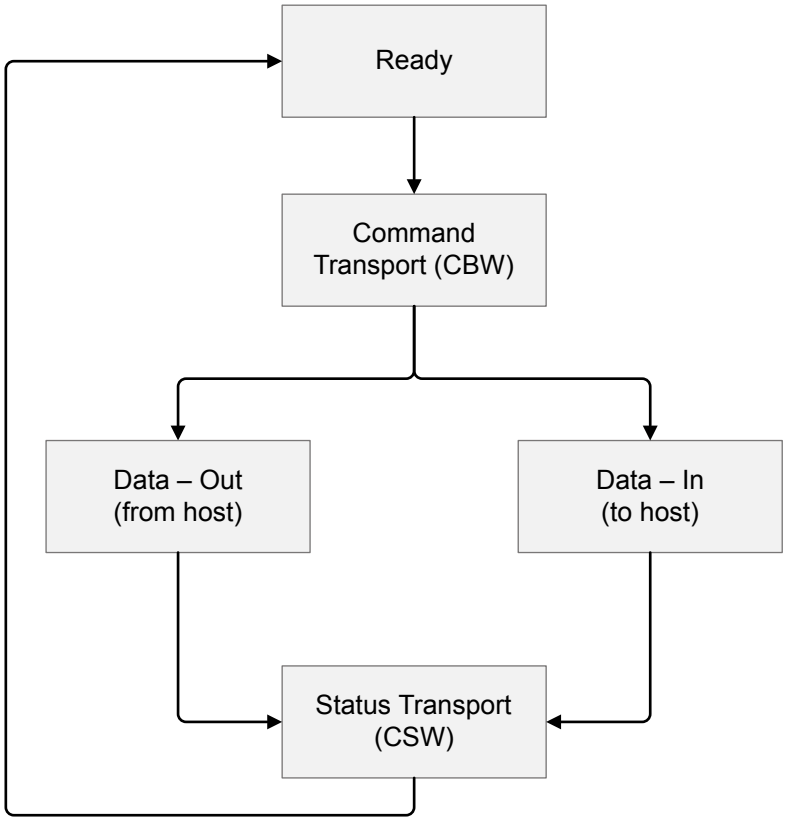


Figure 4.1. MSD Class—Command/Data/Status Flow

- MSD_READY—The state machine is in this state most of the time. In this state, the command interpreter receives data via the data OUT endpoint and checks whether it is a valid and meaningful CBW. If it is determined to be a valid and meaningful CBW, then it calls the SCSI command interpret function. If the SCSI command is write10, it moves to the next state, which is MSD_DATA_OUT. For other commands, it handles them in the SCSI command interpret function and then returns in a CSW.
- MSD_DATA_OUT—In this state, the Data out transfer happens, and the data is read from the USB FIFO and then written into NOR-Flash. Once all data has been received and written into NOR-Flash, it returns the transfer status in the CSW. The state machine transitions to the MSD_READY state.

4.3 SCSI Command Interpreter

Small Computer System Interface - 2 (SCSI-2) is a standard primarily used by hard disk drives and optical drives that defines an I/O bus for interconnecting computers and peripherals. The USB MSD class specification is written such that SCSI commands can be embedded inside the MSD class structures CBW and CSW. This allows Flash-based memory cards to be connected via USB and appear as disk drives within the operating system.

The USB MSD RD implements a SCSI Command Interpreter to process and respond to the SCSI commands sent by the MSD block. This is responsible for the parsing and handling of 10 different SCSI commands as listed in the table below. Unknown commands are also properly handled.

Table 4.3. SCSI Commands, Codes and Responses

SCSI command	SCSI code	Response
SCSI_TEST_UNIT_READY	0x00	“Passed”

SCSI_INQUIRY	0x12	`0x00, // Peripheral qualifier & device type`
		`0x80, // Removable medium`
		`0x05, // Version of the standard (5=SPC-3)`
		`0x02, // No NormACA, No HiSup, data format=2`
		`0x1F, // No extra parameters`
		`0x00, // No flags`
		`0x80, // Basic Task Management supported`
		`0x00, // No flags`
		`'S','i','L','a','b','s',' ',' ',`
		`'M','a','s','s',' ',`
		`'S','t','o','r','a','g','e'`

SCSI_MODE_SENSE_6	0x1A	`0x03,0,0,0` `// No mode sense parameters`
SCSI_START_STOP_UNIT	0x1B	“Passed”
SCSI_PREVENT_ALLOW_MEDIUM_REMOVAL	0x1E	“Passed”
SCSI_READ_CAPACITY_10	0x25	`0x00,0x00,0x3D,0x07, // Last block address` `0x00,0x00,0x02,0x00 // Block length`
SCSI_READ_10	0x28	Read a number of sectors from the NOR-Flash and send those via the USB IN bulk endpoint.
SCSI_WRITE_10	0x2A	Receive a number of sectors via the USB OUT bulk endpoint and write these sectors to the NOR-Flash.
SCSI_VERIFY_10	0x2F	“Passed” (this command is used when the host PC formats the filesystem).

The MSD Class Command Interpreter calls the SCSI Command Interpreter whenever a valid and meaningful CBW is received from the host.

5. Logical To Physical Transition

5.1 Introduction

This USB MSD reference design is using an 8051-based USB MCU and an external 64 Mb NOR-Flash with 4 kB per sector. The NOR-Flash memory can be read or programed byte-by-byte; however, every 256-byte page to be programmed should first be erased, and the Flash can be erased in sectors. This sector erase operation applies to an entire sector (setting all 4 kB bytes in the sector to 0xFF). The status of a page can therefore be complex to manage.

In order to improve the performance and reliability of NOR-Flash arrays, a method of logical to physical address mapping is implemented in the firmware. When overwriting the existing page, the data will be written into a new free physical page without erasing the existing one by updating the logical to physical address mapping array. That helps greatly on read/write performance. In addition, wear-leveling algorithms can be implemented here to average the write/erase times for each page.

The logical to physical transition is popular in NAND flash implementations of mass storage devices. Compared with NAND flash, the NOR Flash advantage is that it doesn't need ECC correction and bad block management. Because of this, it is much easier to maintain the address transition and wear leveling.

5.2 Design Method

Hardware used:

- EFM8UB1, 16 kB Flash, 256 bytes standard 8051 RAM, and 1024 bytes on-chip XRAM
- NOR Flash MX25L6433F, 64 Mb size, 4 kB sector erase operation

This is a resource-constrained system, and the firmware cannot make all logical to physical mappings in the relatively small XRAM. Instead, the firmware can divide the entire NOR-Flash into multiple regions, and each region keeps a logical block to physical block mapping address. The block size should be same as the erase size, which is 4 kB. Since 4 kB is still much bigger than the size of the MCU XRAM, so the block is further divided into multiple sector blocks of 128 bytes. The figure below shows the memory layout used in the USB RD firmware.

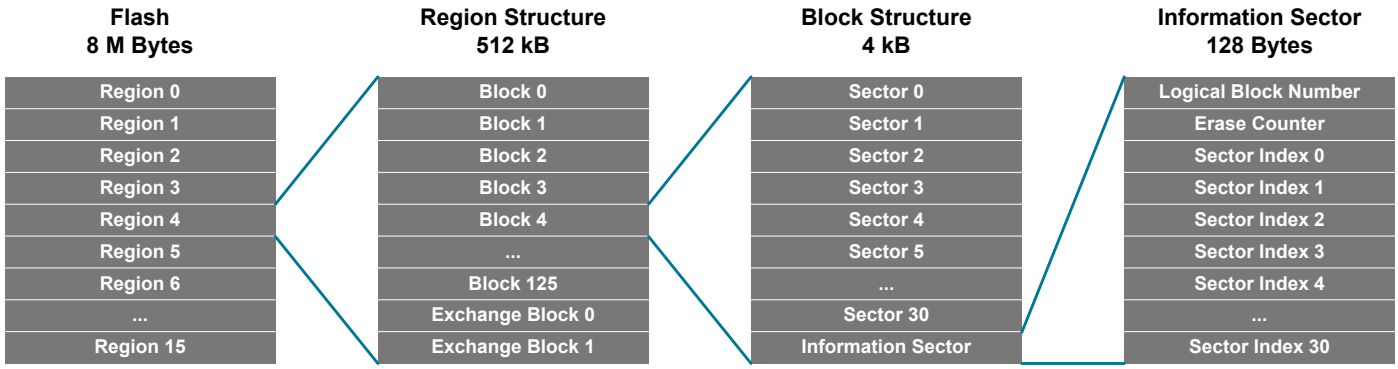


Figure 5.1. NOR-Flash Memory Layout

- There are 16 regions, 512 kB per region.
- There are 512 kB / 4 kB = 128 blocks per region, including 126 data blocks and 2 exchange blocks for data exchange buffer usage.
- There are 4096 / 128 = 32 sectors per block, including 31 data sector and 1 information sector.
- The information structure for each sector is:
  - Relative logical block number (0-125) within a region (2 bytes). The logical to physical block mapping array is used for a region. When trying to read/write another region, the firmware must rebuild the logical to physical block map for the new region.
  - Erase counter which records how many times the block has been erased (4 bytes). When writing data into a block, the find free block function will pick the block with the lowest number of times erased, which helps the average the write/erase times and implements basic wear leveling.
  - Relative sector index logical value (0-30) (1 byte for each sector for a total of 31 bytes). For example, when first writing logical sector 4, the system updates offset 0 to 4. Then, when writing logical sector 3, the system updates offset 1 to 3. Finally, when writing logical sector 4 again, the system updates offset 2 to 4. This offset value indicates that the latest and valid logical sector 4 data is physically located in sector 2 within the block.

Block Mapping

Logical Block Number	Physical Block Number
0	7
1	5
2	2
3	4
4	6
5	127
6	0
...	...
125	89

Sector Mapping

Physical Sector Index	Logical Sector Index
0	0
1	1
2	4
3	2
4	1
5	1
6	0
...	...
30	7

Figure 5.2. Block and Sector Mapping

5.3 Functions

This section discusses the detailed function declarations in the NOR-Flash logical to physical transition firmware example.

5.3.1 Write Sector

Description : Write a sector data into flash drive.

Prototype : `int8_t writeSector(uint32_t sec, uint8_t *buf)`

Parameters :
1. sec—Absolute sector number
2. buf—Pointer to sector data buffer

Return Value :
- 0—Success
- -1—Failure

Detailed Description : This function first checks the sector number within the boundary and then calculates the block number. It calls the logical block to physical block transition function to get the physical block number, and then finds a free sector within this physical block. Then, it calculates the physical address and writes sector data there.

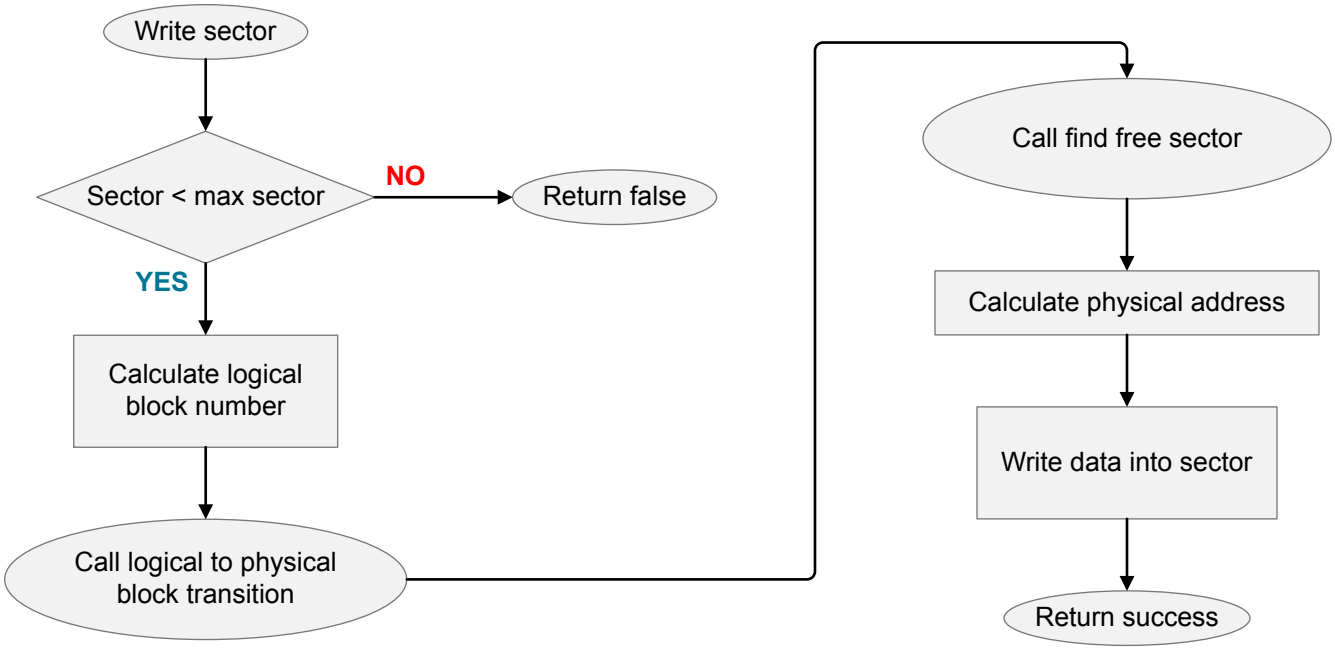


Figure 5.3. Write Sector

5.3.2 Read Sector

Description : Read a sector data into flash drive.

Prototype : `int8_t readSector(uint32_t sec, uint8_t *buf)`

Parameters :
1. sec—Absolute sector number
2. buf—Pointer to sector data buffer

Return Value :
- 0—Success
- -1—Failure

Detailed Description : This function first checks the sector number within the boundary and then calculates the block number. It calls the logical block to physical block transition function to get the physical block number. If the sector is already mapped, the function gets the physical sector number from the mapping array; otherwise, it finds a free sector within this physical block. Then, it calculates the physical address and reads data from the physical sector.

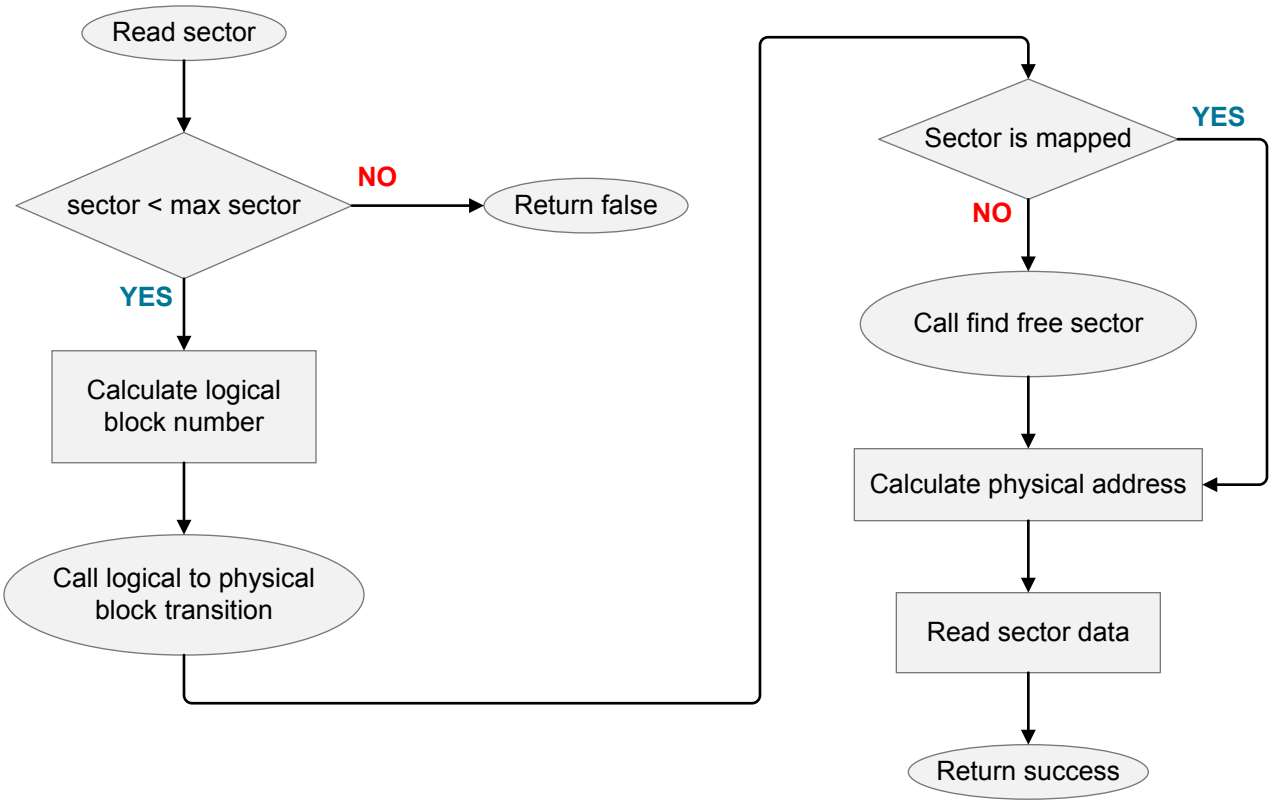


Figure 5.4. Read Sector

5.3.3 Logical to Physical Block (LBA2PBA) Transition

Description : Convert the logical block address (lba) to the physical block address (pba).

Prototype : `static void lba2pba(uint16_t logBlk, struct driverInfo *drv)`

Parameters :
1. logBlk—Logical block number
2. drv—Pointer to the driver information structure

Return Value : None.

Detailed Description : This function first checks the region number. If the region number is different, that means the current region mapping is not in the RAM. It calls the discover logical block to physical block function to build the mapping array. This function then gets the physical block number from the map. If there is no physical block corresponds to the logical block number, then it finds a new block. It builds up a sector mapping within the founded physical block and updates the region current block number with the physical block number.

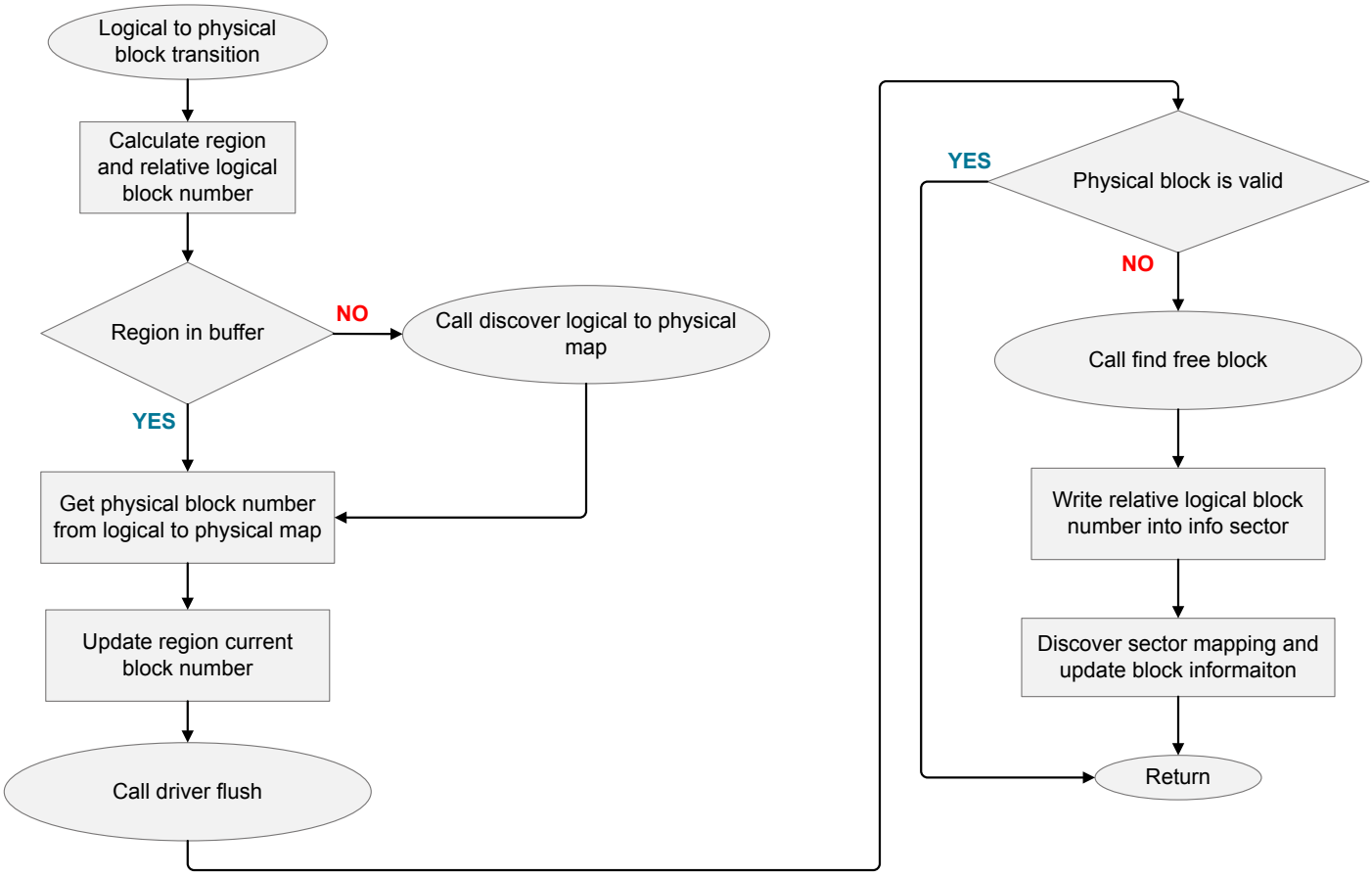


Figure 5.5. LBA2PBA Transition

5.3.4 Discover Logical to Physical Block (LBA2PBA) Map

Description : Discover logical to physical block map within region.

Prototype : `static void discoverLba2pbaTable(struct driverInfo *drv)`

Parameters : 1. drv—Pointer to the driver information structure

Return Value : None.

Detailed Description : This function reads every physical block's information sector to get the logical block number within a region and builds a logical to physical block mapping array. If there are two or more physical blocks mapped to the same logical block number, it combines them into new block.

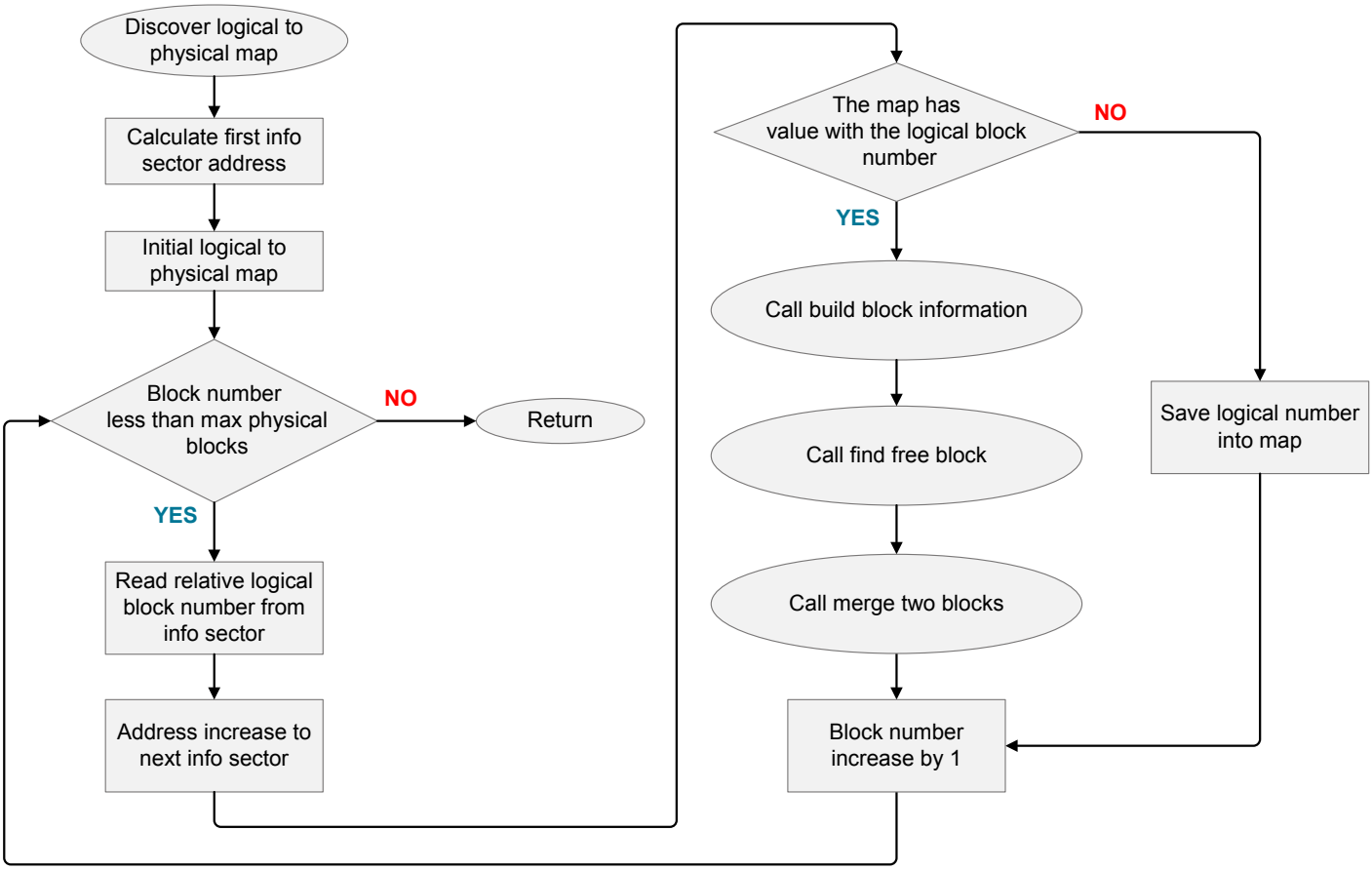


Figure 5.6. Discover LBA2PBA Table

5.3.5 Driver Flush

Description : Flush the last block info in Flash and build the sector map for current block.

Prototype : `static void driverFlush(struct driverInfo *drv)`

Parameters : 1. drv—Pointer to the driver information structure

Return Value : None.

Detailed Description : This function checks if the block number changed. If the previous block has only one block mapped to the logical block, then this function builds the sector mapping array in the current block. If the previous blocks has two physical blocks mapped to same logical block, then the function merges them into a new block. Finally, the function builds the sector mapping array in the current block.

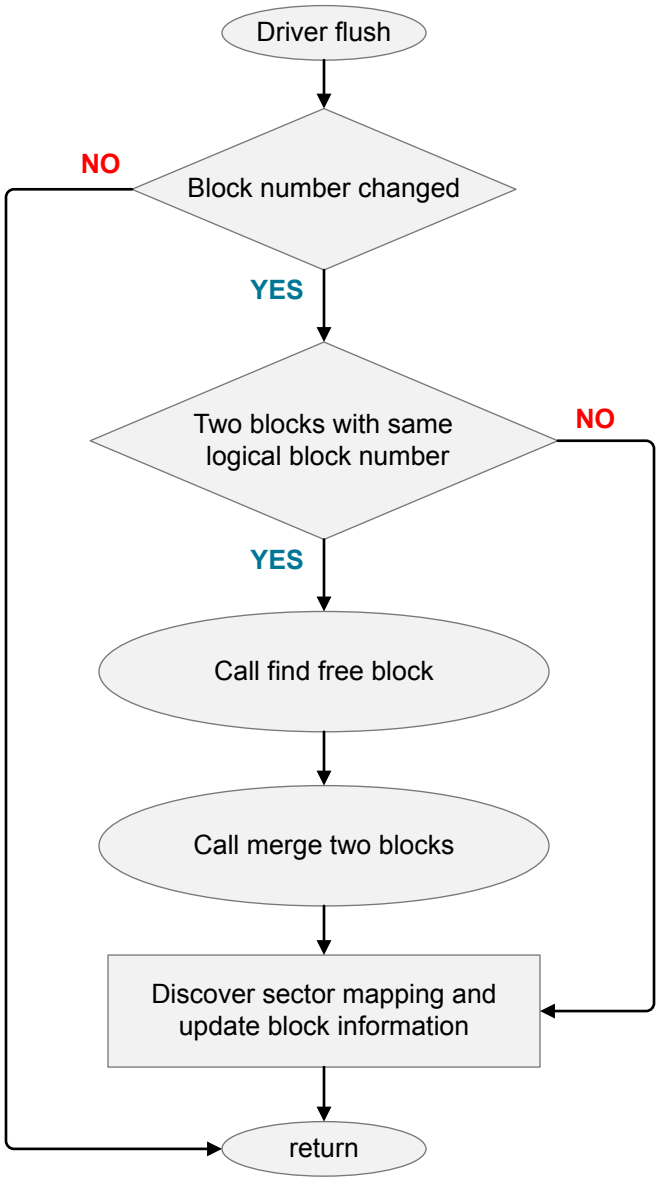


Figure 5.7. Driver Flush

5.3.6 Build Block Information

Description : Build the block information with two given blocks and update the block information with sector mapping information.

Prototype : `static void buildBlkInfo(uint16_t blk0, uint16_t blk1, struct blockInfo *blkInfo)`

Parameters :

1. blk0—Physical block number for first block
2. blk1—Physical block number for second block
3. blkInfo—Pointer to the block information structure

Return Value : None.

Detailed Description : This function determines which block is the current block by checking the last sector index value in the information sector. If the last sector index has a valid value, that means the block is full. This function will then set it as the previous block and update the next block set as current block. This function builds the previous block sector index mapping array and then builds the current block sector index mapping array. Finally, this function updates the blkInfo structure members.

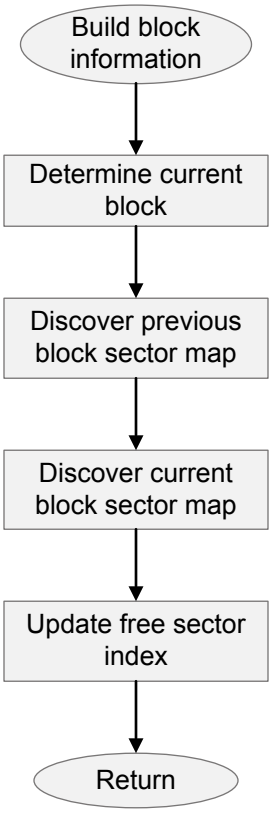


Figure 5.8. Build Block Information

5.3.7 Find Free Block

Description : Find a free block within a region.

Prototype : `static uint16_t findFreeBlock(struct driverInfo *drv)`

Parameters : 1. drv—Pointer to the driver information structure

Return Value : The new physical block number.

Detailed Description : This function searches every block information sector within a region and finds a free block with the minimum number of erases to implement wear leveling.

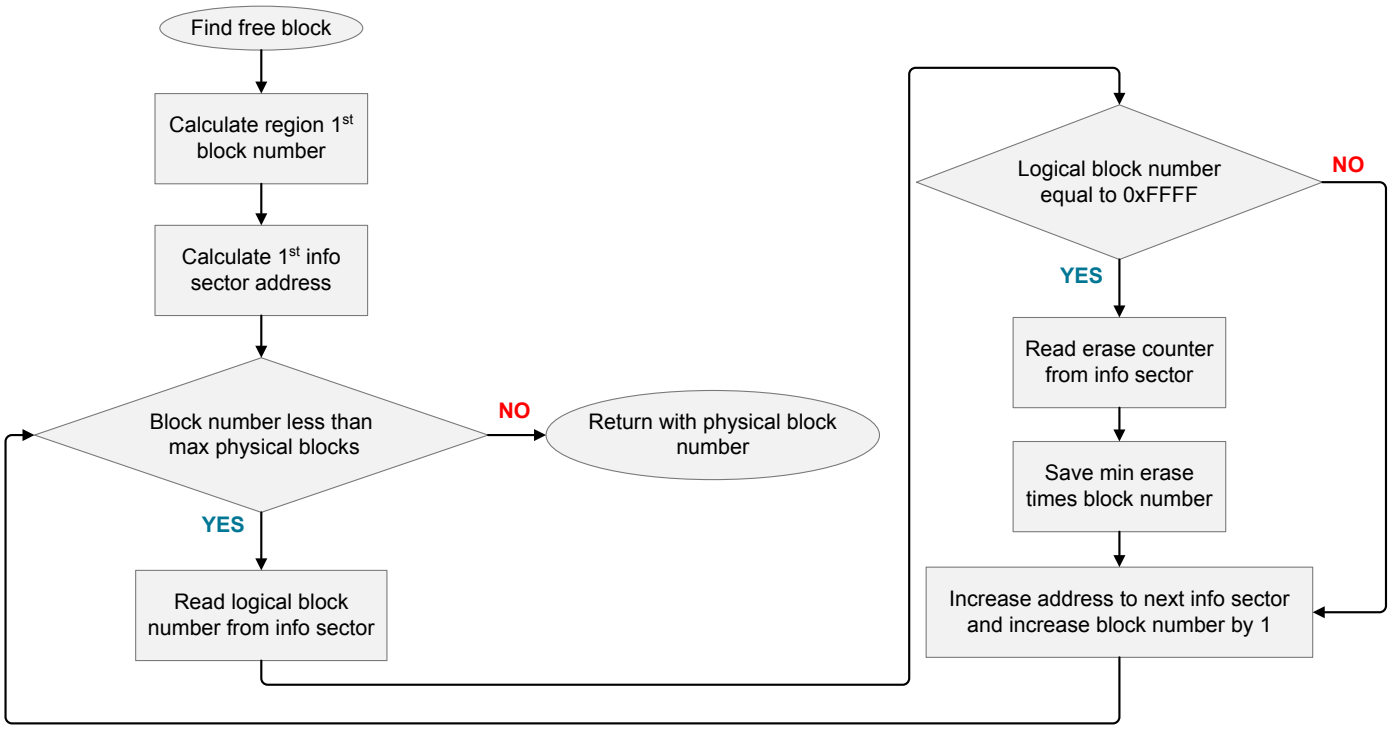


Figure 5.9. Find Free Block

5.3.8 Find Free Sector

Description : Find a free sector for a given relative logical sector.

Prototype : `static uint8_t findFreeSector(uint8_t rls, struct driverInfo *drv)`

Parameters :
1. rls—Relative logic sector number
2. drv—Pointer to the driver information structure

Return Value : The new physical block number.

Detailed Description : This function checks the current block. If the current block is full and if there is previous block, this function finds a new free block and combines two blocks into the new free block. If the new block is full after merging, it finds a new block and updates the block information structure. Then, it gets the relative physical sector number and updates the sector index in the information sector. Finally, this function updates block information structures (`blkMap` and `currSec`).

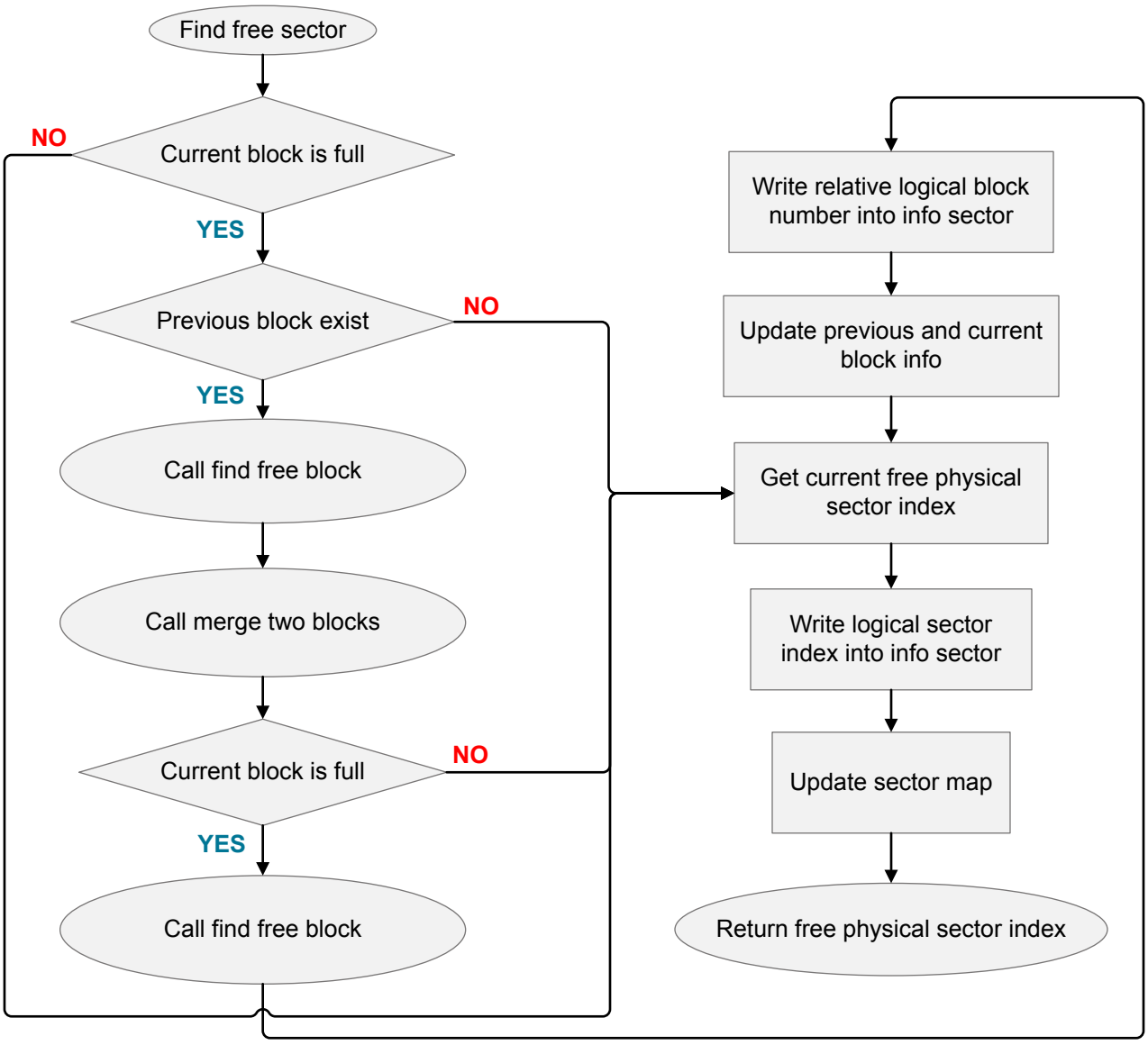


Figure 5.10. Find Free Sector

5.3.9 Merge Two Blocks

Description : Merge two blocks into a new block.

Prototype : `static void mergeTwoBlocks(struct driverInfo *drv, uint16_t blk)`

Parameters :
1. drv—Pointer to the driver information structure
2. blk—Destination block number

Return Value : None.

Detailed Description : This function gets valid sector data and copies it into the destination block. It checks if the logical sector number is mapped in the current block, and if not, then checks if the logical sector is in the previous block. The function copies the source data to the destination address and repeats these steps until all content has been copied into the destination block. It then erases the previous and current blocks. Finally, it builds the sector index map and updates the logical number in the information sector of the new block.

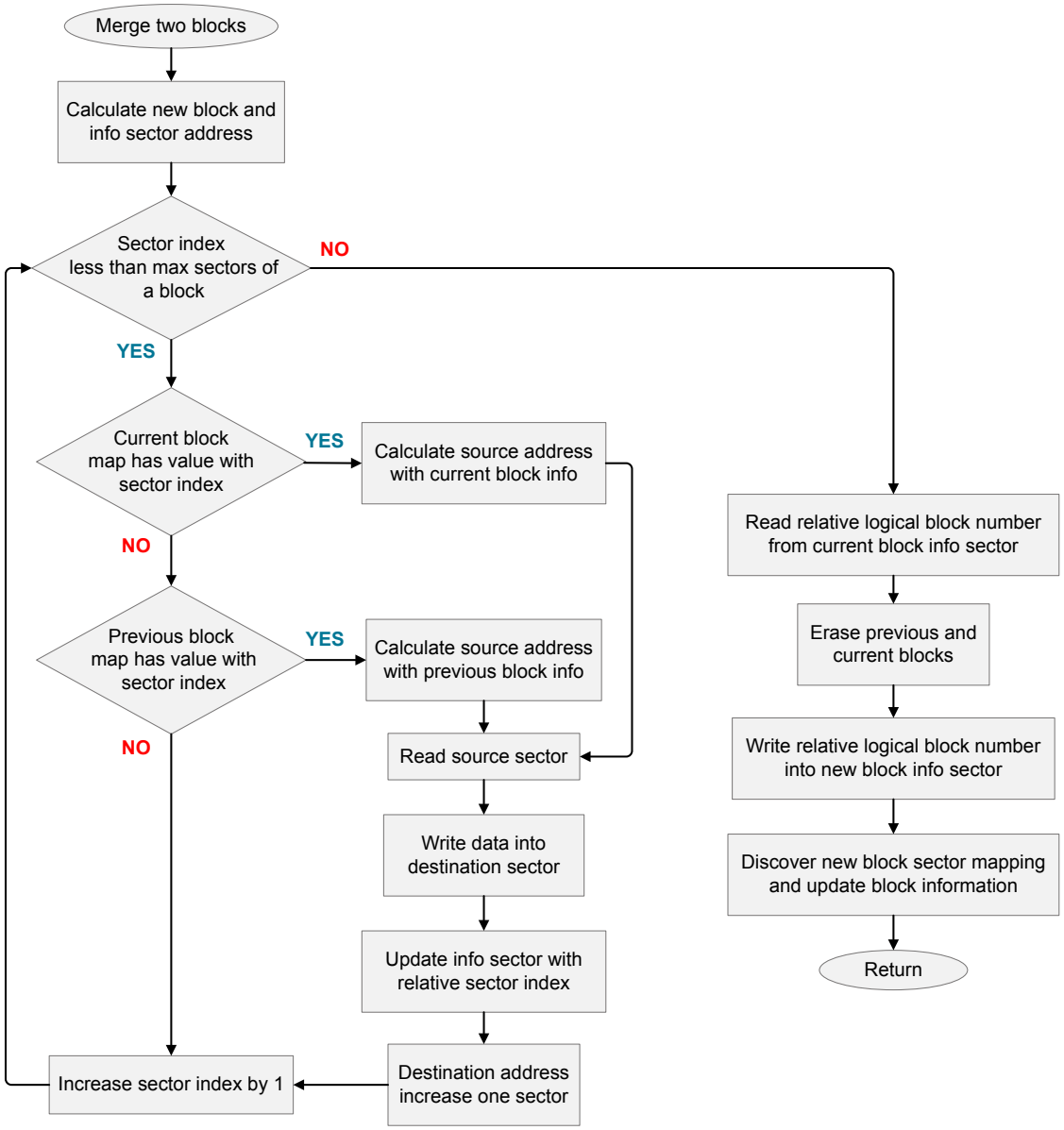


Figure 5.11. Merge Two Blocks

6. Mass Storage Device Mode Operation

When the device is in Mass Storage Device Mode, the following blocks are used. See Figure 3.3 USB MSD RD System Architecture on page 4 for the connections between the blocks.

- USB Low-level Interface
- Mass Storage Device Class Command Interpreter
- SCSI Command Interpreter
- Logical to Physical Transition

The interactions between the USB, MSD, SCSI, and Sector Server blocks are shown in the figures below.

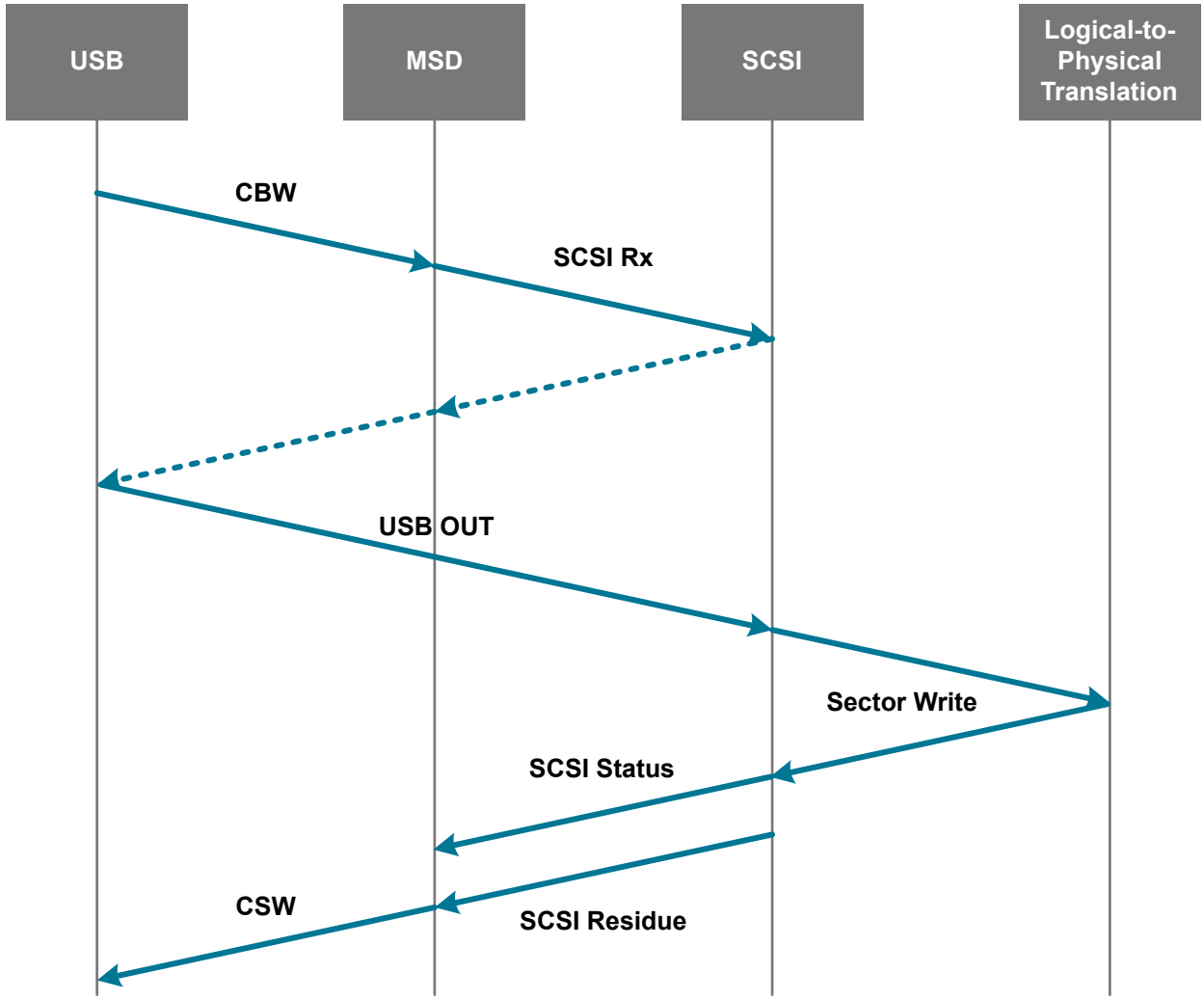


Figure 6.1. Mass Storage Device Operation (Host to Device)

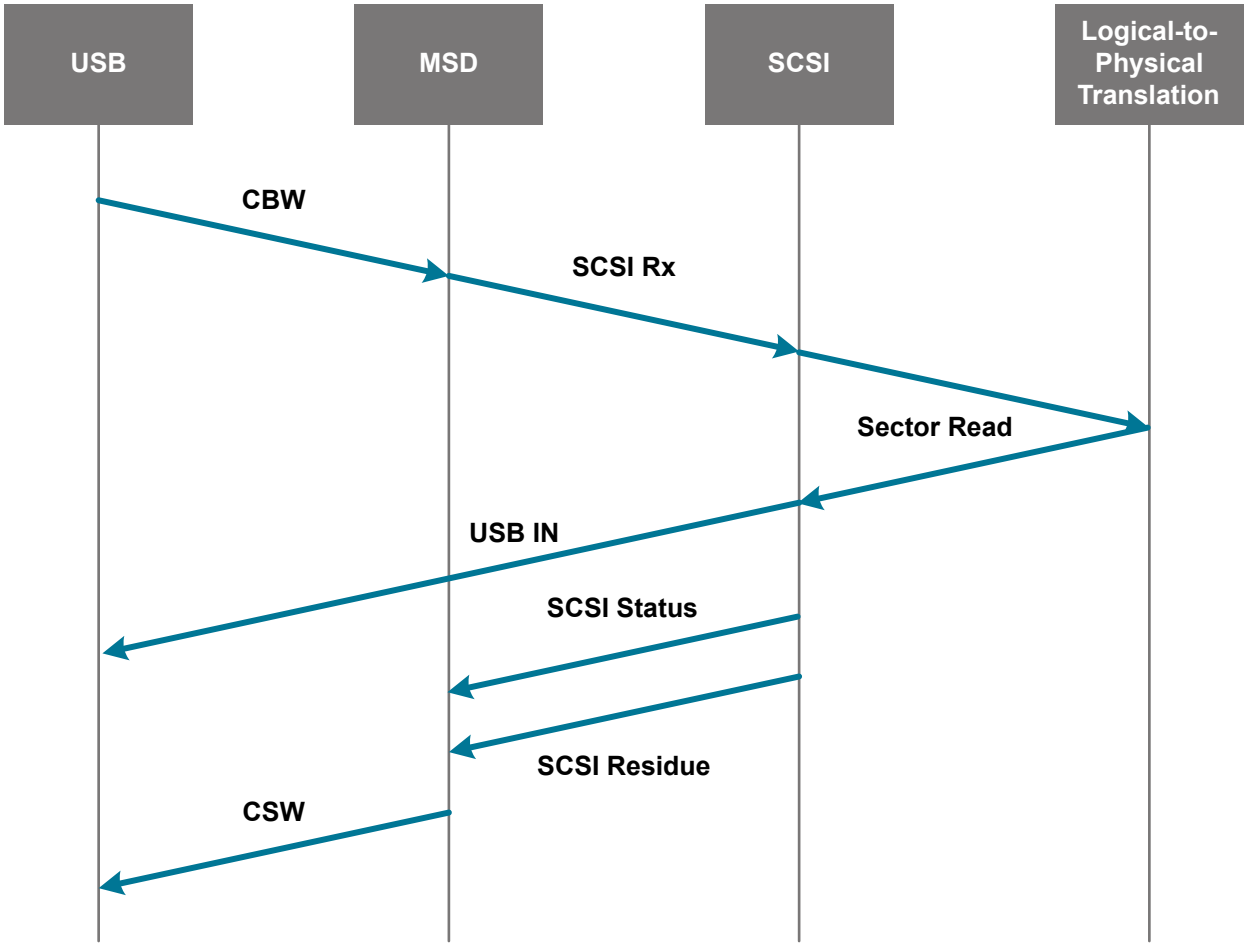


Figure 6.2. Mass Storage Device Operation (Device to Host)

7. Customizing USB MSD RD Firmware

The firmware components included in this reference design can be classified into three categories based on how they will be used in an end application. This classification shows which components should be modified when designing an application that is based on this reference design.

1. Firmware components that are typically used without any modification—USB, MSD, and SCSI are in this category.
2. Firmware components that need modification based on the end application's hardware design—External SPI NOR-Flash Memory is in this category.
3. Firmware components that can be customized with company and/or product information—the following fall in this category:
    a. The SCSI device name that is returned on a SCSI_INQUIRY command.
    b. The USB descriptor parameters that are returned on USB standard requests: VID, PID, and Serial Number.

8. Appendix A—MSD RD USB Descriptor Details

The USB Descriptor used by the USB MSD RD is available in the module `descriptor.c`. The descriptor has been written based on the information from the *USB MSD Bulk-Only Transport* specification. The salient points about this descriptor are listed here:

Device Descriptor

The device descriptor field must have a unique serial number that is at least 12 digits. A unique serial number on a USB device maintains the same device devnode as a user moves the device from one USB port to another. This unique devnode ensures that properties like icons, policies, and drive letters associated with the device are not reset when the device is moved to a different USB port or when a second device with the same VID/PID/REV is added to the system. This is set to "0078976543210" in this design and can be customized.

Interface Descriptor

- `bInterfaceClass` is set to 0x08. This indicates that the device belongs to the USB Mass Storage Device Class.
- `bInterfaceSubClass` is set to 0x06 (SCSI Transparent Mode). Microsoft supports 0x02 for ATAPI CD-ROM, 0x05 for ATAPI removable media, and 0x06 for Generic SCSI media.
- `bInterfaceProtocol` is set to 0x50 (Bulk-Only Transport).

9. Appendix B—USB Host Details

When the USB MSD RD is connected to the PC via a USB cable, it appears as a USB Mass Storage Device. There is no need to install any drivers because the operating system has built-in class drivers. In the case of the USB MSD RD, three Windows built-in drivers are automatically loaded. They are listed here:

Table 9.1. Drivers Loaded by Windows

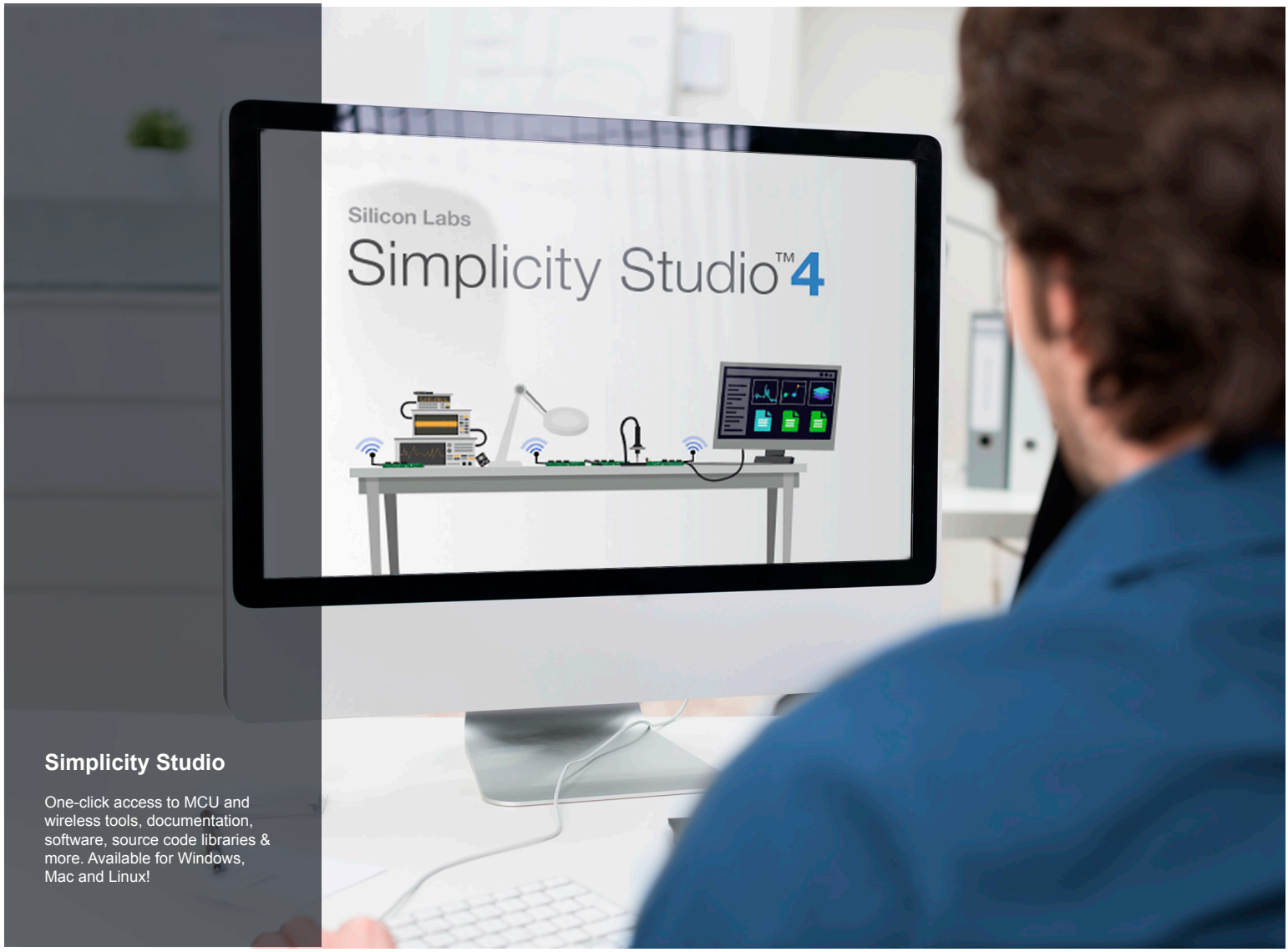
Device Driver Stack	Driver
Generic Volume	File System
Silicon Labs Mass Storage USB Device	`disk.sys`
USB Mass Storage Device	`usbstor.sys`

10. Revision History

10.1 Revision 0.1

November 28th, 2016

Initial release.



Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS®

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com