USB Mass Storage Device Reference Design Programmer's Guide

Relevant Devices

This application note applies to the following devices:
C8051F340, C8051F341, C8051F342, C8051F343, C8051F344, C8051F345, C8051F346, C8051F347

1. Introduction

Among the USB device classes natively supported by popular operating systems, the USB Mass Storage Device (MSD) class is one of the most widely supported device classes. 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 Laboratories USB microcontrollers that complies with the MSD class specification. This USB MSD Reference Design Programmer's Guide describes in detail the various components of the device firmware. The USB-MSD-RD Reference Design Kit User's Guide includes demonstration instructions and information about the kit contents.

2. USB MSD Reference Design Hardware Overview

The MSD reference design hardware consists of two boards - C8051F340-TB Target Board and CF, SD, MMC Memory Expansion Board (AB5). These are contained in the kits C8051F340DK and USB-MSD-RD, respectively. See the USB-MSD-RD User's Guide for details on the contents of these kits. Figure 1 shows these two boards connected via the expansion connector. Figure 2 shows a block diagram with the connections between the hardware components. Refer to the USB-MSD-RD User's Guide for detailed pin connections, schematic, and bill of materials of the evaluation board.

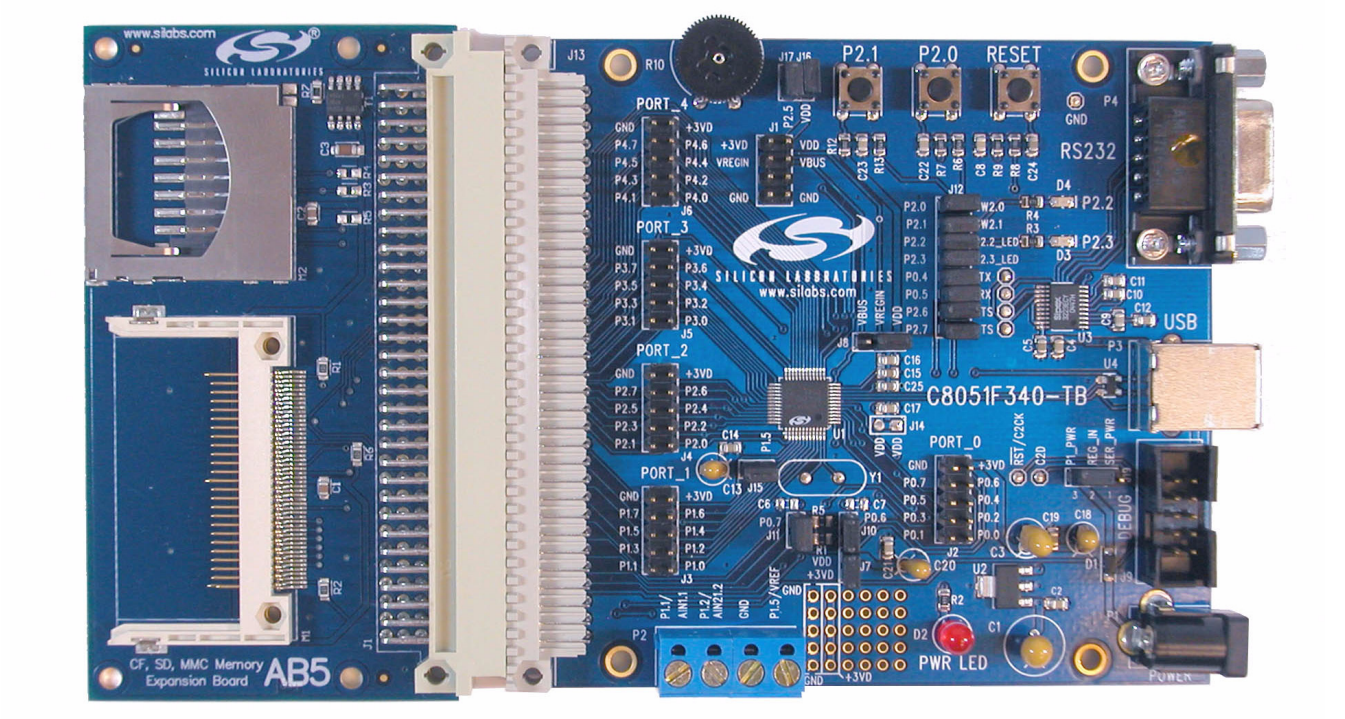

Figure 1. C8051F340-TB Target Board connected to AB5 Expansion Board

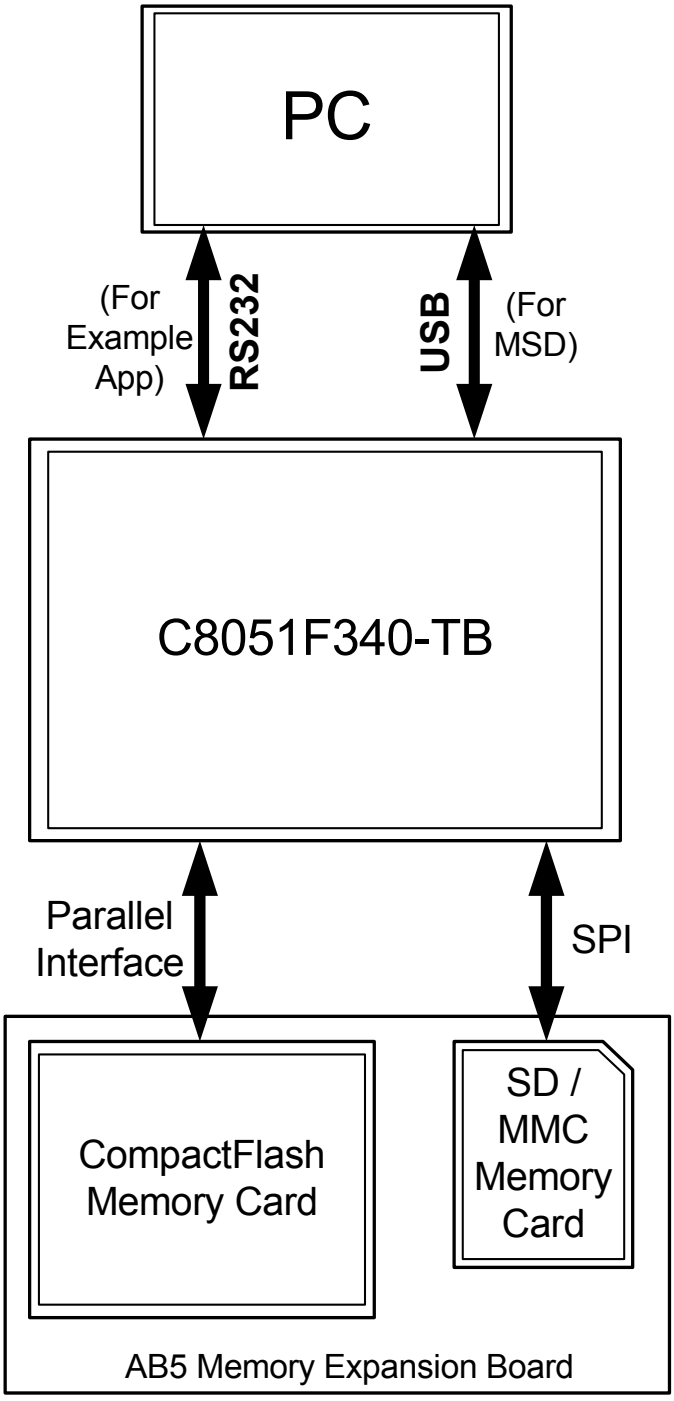

Figure 2. USB MSD RD Hardware Block Diagram

3. USB MSD Reference Design Firmware Overview

The USB MSD RD firmware consists of many distinct blocks that work together. The overall system architecture is shown in Figure 3. The code space usage ratio of the different blocks is shown in Figure 4.

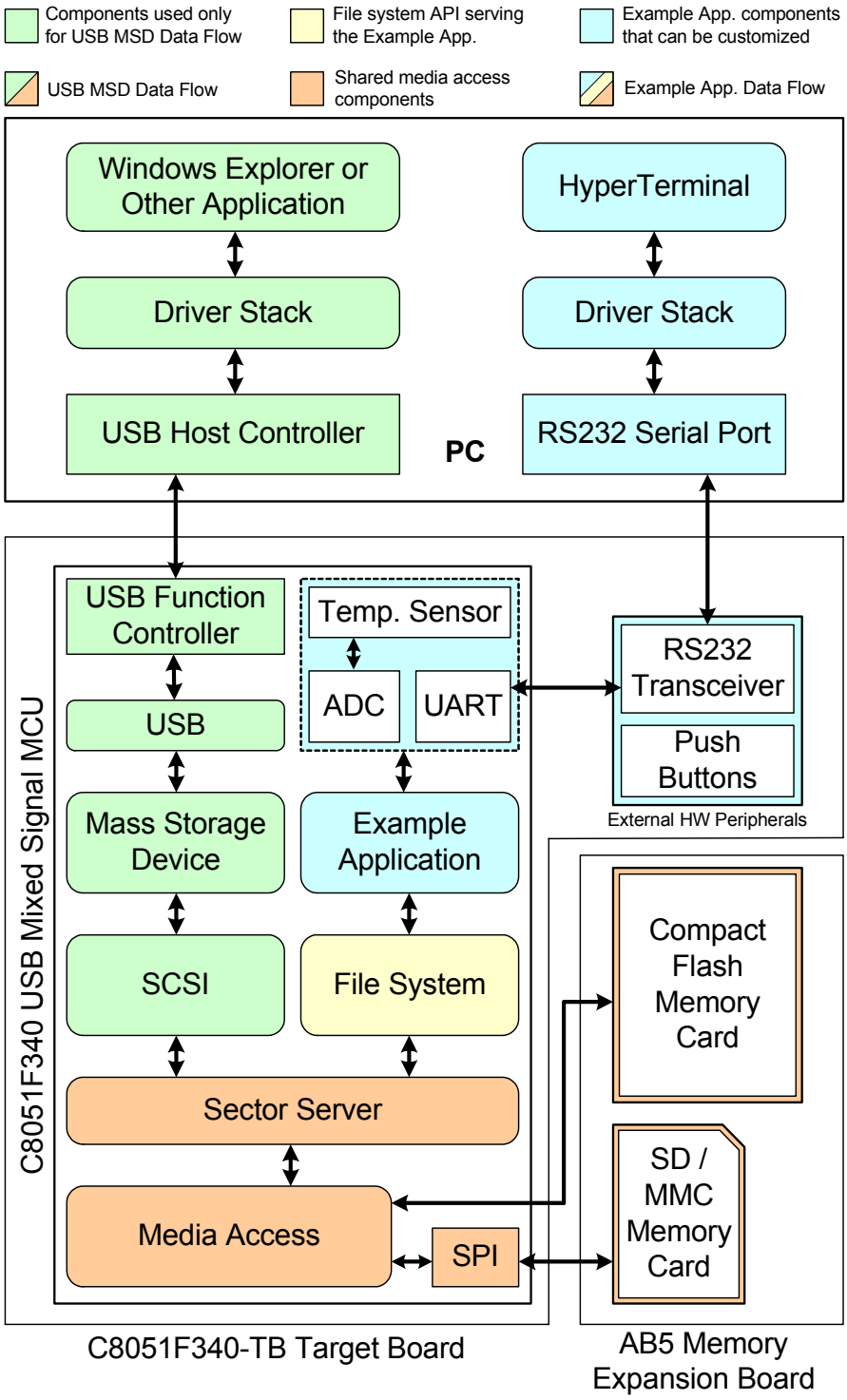

Figure 3. USB MSD RD System Architecture

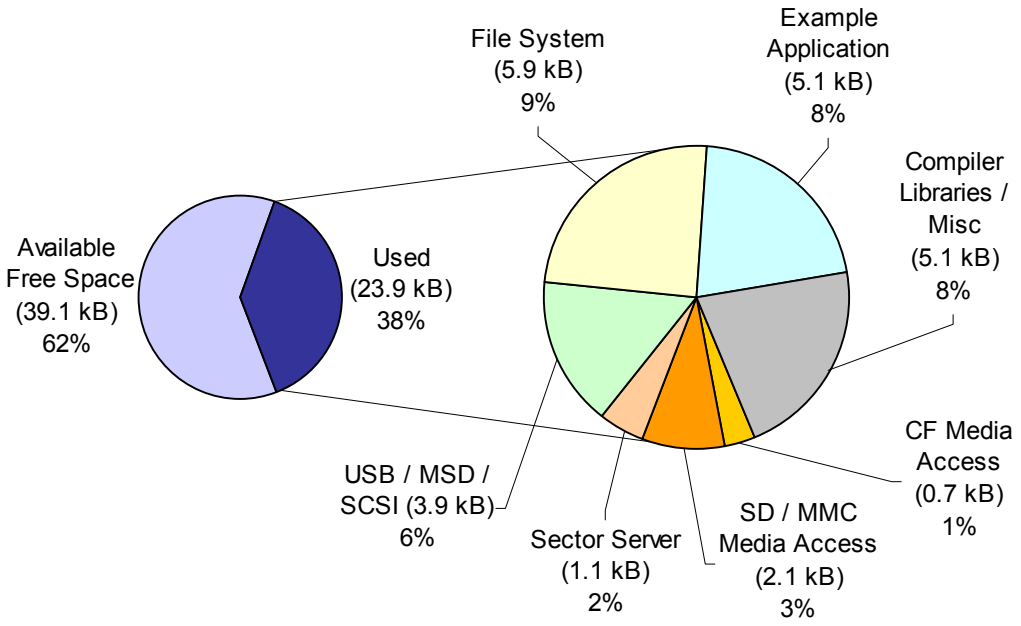

Figure 4. USB MSD Firmware Code Space Usage on the C8051F340

The firmware is designed such that the reference design can be used in one of two ways:

- Mass Storage Device Mode (connected to the PC via USB)
- Independent Embedded System Mode (connected to the PC via RS232 serial interface).

In MSD Mode, the device appears as a USB Mass Storage Device on the PC. This mode uses the firmware components 'USB', 'Mass Storage Device', 'SCSI', 'Sector Server', and 'Media Access'. In Embedded System Mode, the example application can accept commands via a UART command interpreter shell, and perform appropriate functions. In this mode, a RS232 serial connection is made between the 'F340 target board and the PC. This mode uses an external RS232 transceiver, some on-chip hardware peripherals (Temp. sensor, ADC, UART), and the firmware components 'Example Application', 'File System', 'Sector Server', and 'Media Access'. Each component is explained in detail in the following sections. The USB-MSD-RD User's Guide contains step-by-step instructions that demonstrate both the modes of operation.

4. USB MSD RD Firmware Components

4.1. USB MSD RD Media Access Firmware

The USB MSD RD includes firmware to access SD/MMC and CompactFlash cards. In a typical application where only one type of interface is needed, the unnecessary media access firmware can be removed to save code space. Also, some parts of this media access firmware might need to be modified to fit the end application. For example, the pins that are assigned to the SPI peripheral might need to be changed to accomodate other system requirements.

4.1.1. CompactFlash Interface Firmware

The CompactFlash (CF) Interface Firmware is the low-level interface that allows the rest of the system to access a CF memory card. The CF card is accessed by the firmware using a parallel interface that consists of an 8-bit data bus, a 3-bit address bus, and 6 control signals. A CF card is connected to the 'F340 device port pins as described in Section “2. USB MSD Reference Design Hardware Overview”. After a device reset, the Init_CF function is called by the Sector Server. If this detects a card, then a call to Identify_Drive is made to get the size information of the card. The Read_Sector and Write_Sector functions are then used for data transfers.

The CompactFlash media access functions are described below. There is typically no need to call any of these functions directly from the Application-level firmware because the Sector Server layer encapsulates them. When designing new hardware, these functions may need to be modified to suit the new hardware connections.

4.1.1.1. Init_CF

Description: Initializes a CompactFlash memory card.

Supported Devices: C8051F340/1/2/3/4/5/6/7
Module: F34x_MSD_CF_Basic_Functions.c
Prototype: `char Init_CF (void)`
Parameters: None.

Return Value: 0, if initialization was successful.
CF_NO_CARD, if no card was detected.

4.1.1.2. Identify_Drive

Description: Reads the card identifier from a CompactFlash card.

Supported Devices: C8051F340/1/2/3/4/5/6/7
Module: F34x_MSD_CF_Basic_Functions.c
Prototype: `char Identify_Drive (char* buffer)`
Parameters: 1. buffer-pointer to a memory location for returning the card identifier information.

Return Value: None.

4.1.1.3. Read_Sector

Description: Reads a 512-byte block from a CompactFlash card starting at the location specified by *address*. The block is copied to the memory location pointed to by *buffer*.

Supported Devices: C8051F340/1/2/3/4/5/6/7
Module: F34x_MSD_CF_Basic_Functions.c
Prototype: `char Read_Sector (unsigned long address, char* buffer)`
Parameters: 1. address-Starting address of the 512-byte block in the CF card.
2. buffer-Pointer to a memory location where the data will be copied to.

Return Value: 0, if the read operation was successful.
CF_NO_CARD, if no CF card was detected.
Errorcode, otherwise.

4.1.1.4. Write_Sector

Description: Write a 512-byte block to a CompactFlash card starting at the location specified by *address*. The block is copied from the memory location pointed to by *buffer* to the CF card.

Supported Devices: C8051F340/1/2/3/4/5/6/7
Module: F34x_MSD_CF_Basic_Functions.c
Prototype: `char Write_Sector (unsigned long address, char* buffer)`
Parameters: 1. address-Starting address of the 512-byte block in the CF card.
2. buffer-Pointer to a memory location where the data will be copied from.

Return Value: 0, if the write operation was successful.
CF_NO_CARD, if no CF card was detected.
Errorcode, otherwise.

4.1.2. SD/MMC Interface Firmware

The SD/MMC Interface Firmware is the low-level interface that allows the rest of the system to access Secure Digital (SD) and MultiMediaCard (MMC) memory cards. SD/MMC cards use SPI for communication in 4-wire mode. A SD/MMC card is connected to the 'F340 device port pins as described in Section “2. USB MSD Reference Design Hardware Overview”. After a device reset, the Sector Server checks if a CF card is present as described in Section 4.1.1. If no CF card is detected, then MMC_FLASH_Init is called to check if a SD/MMC card is present.

The SD/MMC media access functions are described below. There is typically no need to call any of these functions directly from the Application-level firmware because the Sector Server layer encapsulates them. When designing new hardware, these functions may need to be modified to suit the new hardware connections.

4.1.2.1. MMC_FLASH_Init

Description: Initializes a SD/MMC memory card. This calls the internal function SPI_Init to initialize the SPI hardware, examines the Operating Conditions Register (OCR) to ensure that the device has been initialized correctly, and also determines the size of the card by reading the Card Specific Data Register (CSD).

Supported Devices: C8051F340/1/2/3/4/5/6/7

Module: F34x_MSD_MMC.c

Prototype:
```
void MMC_FLASH_Init (void)
```

Parameters: None.

Return Value: None.

4.1.2.2. MMC_FLASH_Block_Read

Description: Reads a 512-byte block from a SD/MMC card starting at the location specified by *address*. The block is copied to the memory location pointed to by *pchar*.

Supported Devices: C8051F340/1/2/3/4/5/6/7

Module: F34x_MSD_MMC.c

Prototype:
```
unsigned int MMC_FLASH_Block_Read (unsigned long address, unsigned char* pchar)
```

Parameters:
1. address-Starting address of the 512-byte block in the SD/MMC card.
2. pchar-Pointer to a memory location where the data will be copied to.

Return Value: 0, if the read operation was successful.
Card Response Code, otherwise.

4.1.2.3. MMC_FLASH_Block_Write

Description: Write a 512-byte block to a SD/MMC card starting at the location specified by *address*. The block is copied from the memory location pointed to by *wdata* to the CF card.

Supported Devices: C8051F340/1/2/3/4/5/6/7

Module: F34x_MSD_MMC.c

Prototype:
```
unsigned char MMC_FLASH_Block_Write (unsigned long address, unsigned char *wdata)
```

Parameters:
1. address-Starting address of the 512-byte block in the CF card.
2. wdata-Pointer to a memory location where the data will be copied from.

Return Value: 0, if the write operation was successful.
Card Response Code, otherwise.

4.2. Sector Server

The tasks and capabilities of the Sector Server are listed below:

- Encapsulates lower-level CF and SD/MMC media access functions, so that those functions do not need to be called from the Application-level firmware.
- Detects and initializes the memory card on startup.
- Reads and validates the boot sector of the memory card. Checks for a block size of 512 bytes, FAT16 filesystem, and a boot sector signature of 0xAA55 (little endian).
- Reads and stores a global copy of details about the memory card such as the number of FAT copies, the number of root directory entries, the number of sectors, and the size of each FAT.
- Maintains one 512-byte scratch buffer that is used by the entire system to read or write blocks of data.
- Handles multiple FAT copies by keeping them in sync with each other. Transaction commits and rollbacks are not supported because of this approach, i.e. transactions are always committed, and cannot be rolled back.
- Supports memory cards with capacity up to 4 GB. This restriction is imposed by the FAT16 filesystem.
- Maintains a global disk map. This is an overview of where each FAT copy begins and ends, where the root directory is located, and where the file data area begins. See Table 1 for an example Sector Server View of a 32 MB MMC card.

Table 1. Sector Server View of a 32 MB MMC Card (FAT16 filesystem)

MMC block Numbers	SCSI Block Numbers	Used for:	Number of Blocks	Size
0	Non-existent	Partition table	1	512 B
1..31	Non-existent	Not used	31	15.5 K
32	0	Boot record	1	512 B
33..275	1..243	1st FAT copy	243	122 KB
276..518	244..486	2nd FAT copy	243	122 KB
519..550	487..518	Root directory	32	16 KB
551..end	519..end	File data	62000+	30+ MB

The Embedded File System API and the SCSI Command Interpreter encapsulate the Sector Server functions. So, there is typically no need to call any of the Sector Server functions directly from the Application-level firmware. Because of this, these functions are not explained here in detail. Instead, a list of Sector Server functions with brief explanations is presented in Table 2.

Table 2. Sector Server Functions

Function Name	Description
Module: F34x_MSD_Sect_Serv.c	
Sect_Init	Initializes and validates the memory card.
Sect_Validate	Checks the validity of the memory card boot record.
Sect_Sectors	Returns the number of sectors of the current memory card.
Sect_Print	Prints memory card type and size.
Sect_Read	Reads one sector into the scratch buffer.
Sect_Write	Writes one sector from the scratch buffer.
Sect_Write_Multi_Fat	Writes changes to the 1st FAT copy in the 2nd FAT copy as well, thus keeping both the FAT copies in sync.
Sect_Root_Dir	Returns the first sector of the root directory.
Sect_Root_Dir_Last	Returns the last sector of the root directory.
Sect_File_Data	Returns the first sector of file data.
Sect_Fat1	Returns the first sector of the 1st FAT
Sect_Fat2	Returns the first sector of the 2nd FAT

4.3. USB Low-level Interface

The USB descriptor is an important component of the USB low-level interface. When a USB device is plugged into a USB host, USB Enumeration is initiated, 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 "Appendix A—MSD RD USB Descriptor Details" on page 20 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 bus conditions USB 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. Please refer to the module "F34x_MSD_USB_ISR.c" for the functions and their descriptions.

4.4. 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. They are described below:

4.4.1. Command Block Wrapper (CBW)

CBW is defined as a packet containing a command block and associated information. See Figure 5 for the format of this structure.

Table 3. 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)							

4.4.2. Command Status Wrapper (CSW)

CSW is defined as a packet containing the status of a command block. See Figure 5 for the format of this structure.

Table 4. 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)							

4.4.3. Command Interpreter State Machine

The command interpreter implements a simple state machine that is very similar to the Command/Data/Status Flow shown in Figure 5. In the state machine implementation, the 'Data - In' and 'Data - Out' stages in this figure have been combined into one state.

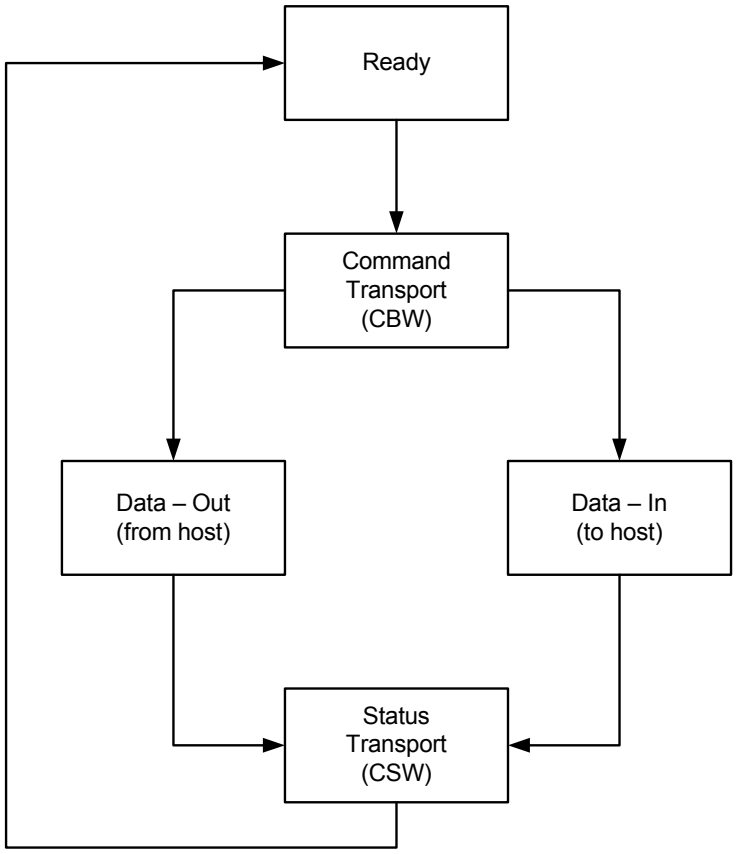

Figure 5. MSD Class - Command/Data/Status Flow

- 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 moves to the next state, which is MSD_DATA.
- MSD_DATA: In this state, the valid CBW is sent to the SCSI block for processing. Depending on the contents of the CBW (direction bit in bmCBWFlags), either a Data-In or a Data-Out transfer happens. Then, the state machine transitions to the next state.
- MSD_STATUS_TRANSPORT: In this state, the Good/Failed/PhaseError status and the amount of unprocessed bytes is returned in a CSW. After this state, the state machine reverts to the MSD_READY state.

The state machine is implemented as one function, which is described below.

4.4.4. Msd_Step

Description:	The MSD Class Command Interpreter is implemented as a state machine in this function. This function should be called periodically from the main loop. For the USB MSD RD firmware, the interval should be less than 3 ms to keep the data transfer at the highest possible level. Increasing the calling interval over 3 ms will lower the data transfer speed. The determination of the 3 ms interval involves the timing of the low-level media access functions. So, the best interval should be reevaluated if any modifications are made to those functions.
Supported Devices:	C8051F340/1/2/3/4/5/6/7
Module:	F34x_MSD_MSD.c
Prototype:	`void Msd_Step (void)`
Parameters:	None.
Return Value:	None.

Refer to Section “5. Mass Storage Device Mode Operation” for an illustration of how the MSD block communicates with the other firmware components.

4.5. 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 Table 5. Unknown commands are also properly handled.

Table 5. SCSI Commands, Codes, and Responses

SCSI command	SCSI code	Response
SCSI_TEST_UNIT_READY	0x00	“Passed”

SCSI_INQUIRY	0x12	code const BYTE
		Scsi_Standard_Inquiry_Data[36]= {
		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	code const BYTE Scsi_Mode_Sense_6[4]= {
		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	BYTE Scsi_Read_Capacity_10[8]={
		0x00,0x00,0xF4,0x5F, // Last block address
		0x00,0x00,0x02,0x00 // Block length
		};

SCSI_READ_10	0x28	Read a number of sectors from the memory card 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 memory card.
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. After processing the command, the SCSI Command Interpreter is responsible for setting Scsi_Status and Status_Residue to appropriate values.

- Scsi_Status can be set to one of three values: Passed (0), Failed (1), or Phase Error (2).
- Scsi_Residue indicates how many bytes of data have not been processed.

4.6. Embedded File System Interface

This firmware component is unique compared to the other blocks because this is the only one that is solely used by the Application Firmware when the device is in 'Embedded System Mode', and not at all when it is in 'Mass Storage Device Mode'. This component provides the Application Firmware with an API to the FAT16 file system. This interface is commonly referred to as a "Stream I/O interface" because of the use of functions like fopen, fread, and fclose. Note that Long File Names (LFNs) are not supported by this API functions. Abbreviated LFNs (abcdef~1.txt etc) can be used in place of LFNs. The API functions are described in the section below.

4.6.1. FileSys_Init

Description: Initializes variables that are used for navigation over directories.

Supported Devices: C8051F340/1/2/3/4/5/6/7
Module: F34x_MSD_File_System.c
Prototype: `void FileSys_Init (void)`
Parameters: None.

Return Value: None.

4.6.2. write_current_dir

Description: Sends the current working directory path via the UART.

Supported Devices: C8051F340/1/2/3/4/5/6/7
Module: F34x_MSD_File_System.c
Prototype: `void write_current_dir (void)`
Parameters: None.

Return Value: None.

4.6.3. chngdir

Description: Changes the current working directory.

Supported Devices: C8051F340/1/2/3/4/5/6/7
Module: F34x_MSD_File_System.c
Prototype: `BYTE chngdir (char* dirname)`
Parameters: 1. dirname-Pointer to a memory location that contains the directory name.

Return Value: 0, if the directory does not exist.
1, if the directory change was successful.

4.6.4. mkdir

Description:	Creates a new directory.
Supported Devices:	C8051F340/1/2/3/4/5/6/7
Module:	F34x_MSD_File_System.c
Prototype:	`BYTE mkdir (char* dirname)`
Parameters:	1. dirname-Pointer to a memory location that contains the directory name.
Return Value:	0, if the directory was created successfully. DIRECTORY_EXISTS, if a directory by the specified name already exists. NO_PLACE_FOR_DIRECTORY, if there is no space to create a directory.

4.6.5. rmdir

Description:	Removes the specified directory. The contents of the directory (files, subdirectories) are also deleted.
Supported Devices:	C8051F340/1/2/3/4/5/6/7
Module:	F34x_MSD_File_System.c
Prototype:	`BYTE rmdir (char* dirname)`
Parameters:	1. dirname-Pointer to a memory location that contains the directory name.
Return Value:	1, if the directory was successfully removed. 0, otherwise.

4.6.6. fcreate

Description:	Creates a new file with the specified name.
Supported Devices:	C8051F340/1/2/3/4/5/6/7
Module:	F34x_MSD_File_System.c
Prototype:	`static BYTE fcreate (find_info* findinfo, char* filename)`
Parameters:	1. findinfo-Pointer to a structure with details about the file. 2. filename-Pointer to a memory location that contains the filename.
Return Value:	1, if the file was created successfully. 0, otherwise.

4.6.7. fopen

Description:	Opens a file in one of three modes - Read (r), Write (w), Append (a). In Write/Append modes, if the specified filename does not exist, it is created. In Write mode, if the specified file exists, it is overwritten.
Supported Devices:	C8051F340/1/2/3/4/5/6/7
Module:	F34x_MSD_File_System.c
Prototype:	`int fopen (FILE* f, char* filename, char* mode)`
Parameters:	1. f-Pointer to file structure 2. filename-Pointer to a memory location that contains the filename. 3. mode-Pointer to a memory location that contains the file open mode.
Return Value:	1, if file was opened successfully. 0, otherwise.

4.6.8. fread

Description:	Reads the specified number of bytes from a file.
Supported Devices:	C8051F340/1/2/3/4/5/6/7
Module:	F34x_MSD_File_System.c
Prototype:	`unsigned fread (FILE* f, BYTE* buffer, unsigned count)`
Parameters:	1. f-Pointer to file structure 2. buffer-Pointer to a memory location where the data will be copied to. 3. count-The maximum number of bytes to read from the file.
Return Value:	Number of bytes read from the file.

4.6.9. fwrite

Description:	Writes the specified number of bytes to a file.
Supported Devices:	C8051F340/1/2/3/4/5/6/7
Module:	F34x_MSD_File_System.c
Prototype:	`unsigned fwrite (FILE* f, BYTE* buffer, unsigned count)`
Parameters:	1. f-Pointer to file structure 2. buffer-Pointer to a memory location where the data will be copied from. 3. count-The number of bytes to write to the file.
Return Value:	Number of bytes written to the file.

4.6.10. feof

Description:	Checks whether the specified file's current position pointer has reached the end of the file.
Supported Devices:	C8051F340/1/2/3/4/5/6/7
Module:	F34x_MSD_File_System.c
Prototype:	`int feof (FILE* f)`
Parameters:	1. f-Pointer to file structure
Return Value:	0, if the file's current position pointer has not reached the end of the file. 1, if the file's current position pointer has reached the end of the file.

4.6.11. fclose

Description:	Closes a file.
Supported Devices:	C8051F340/1/2/3/4/5/6/7
Module:	F34x_MSD_File_System.c
Prototype:	`void fclose (FILE* f)`
Parameters:	1. f-Pointer to file structure
Return Value:	None.

4.6.12. fdelete

Description:	Deletes a file in the current working directory.
Supported Devices:	C8051F340/1/2/3/4/5/6/7
Module:	F34x_MSD_File_System.c
Prototype:	`int fdelete (char* name)`
Parameters:	1. name-Pointer to a memory location that contains the file name.
Return Value:	1, if the file was deleted successfully. 0, otherwise.

4.6.13. Format_Disk

Description:	Formats a disk to the FAT16 file system. Note that the disk should have an existing partition for this to work. WARNING: Calling this function will erase all the data on the disk.
Supported Devices:	C8051F340/1/2/3/4/5/6/7
Module:	F34x_MSD_Format_Disk.c
Prototype:	`void Format_Disk (void)`
Parameters:	None.
Return Value:	None.

5. Mass Storage Device Mode Operation

When the device is in Mass Storage Device Mode, the following blocks are used. See Figure 3 for the connections between the blocks.

- USB Low-level Interface
- Mass Storage Device Class Command Interpreter
- SCSI Command Interpreter
- Sector Server
- Media Access Firmware

The interactions between the USB, MSD, SCSI, and Sector Server blocks are shown in Figure 6 and Figure 7.

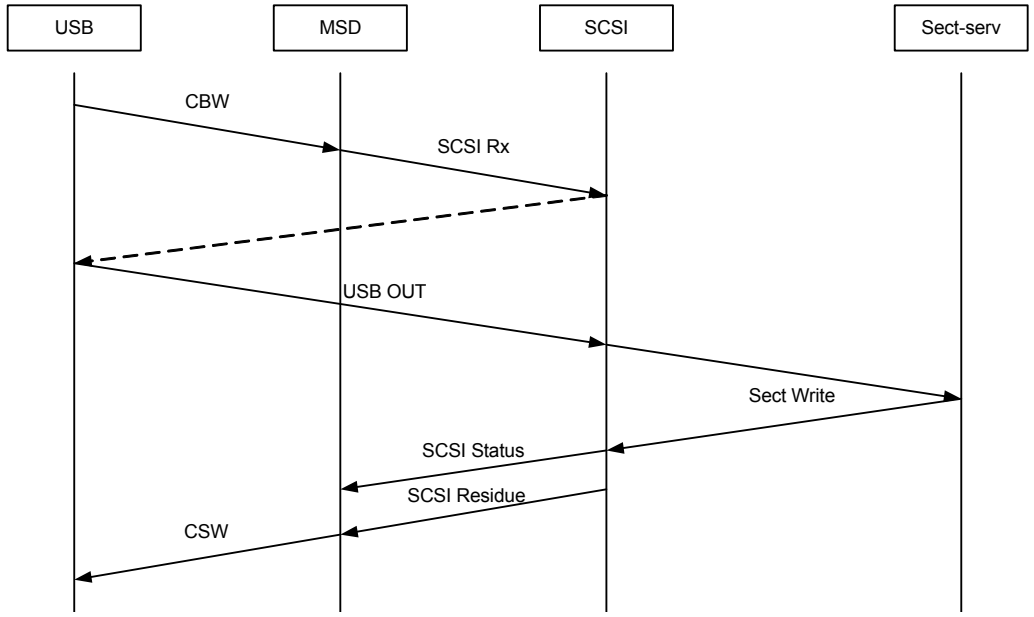

Figure 6. Mass Storage Device Mode Operation (Host to Device)

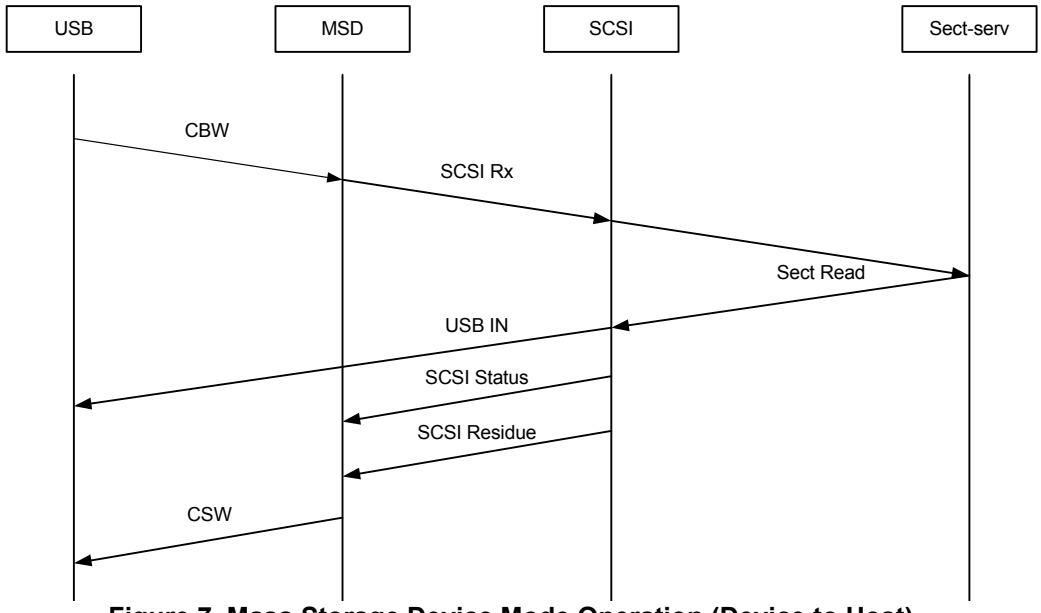

Figure 7. Mass Storage Device Mode Operation (Device to Host)

6. Embedded System Mode Operation

When the device is in the Embedded System Mode, the following blocks are used. See Figure 3 for the connections between the blocks.

- App. Specific Hardware (RS232 Transceiver)
- On-chip hardware peripherals (Temp. sensor, ADC, and UART)
- Application Firmware
- File System API
- Sector Server
- Media Access Firmware

See the USB MSD User's Guide for demonstration instructions and a list of commands supported by the Example Application's UART-based Command Interpreter Shell. For detailed information about the implementation of the example application, refer to the source files in the software package that accompanies this application note.

7. Customizing USB MSD RD Firmware

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

1. Firmware components that are typically used without any modification - USB, MSD, SCSI, and Sector Server fall in this category.
2. Firmware components that need modification based on the end application's hardware design - Media Access Firmware falls in this category.
3. Firmware components that can be customized with company and/or product information - the following fall in this category:
   a. SCSI device name that is returned on a SCSI_INQUIRY command
   b. USB descriptor parameters that are returned on USB standard requests: VID, PID, and Serial Number.
4. Firmware components that are fully customizable - the Example Application and all its related functions fall in this category. You can either modify the provided example code to fit your application needs, or you can create an application from scratch using the File System API functions listed in Section 4.6.

8. References

The following specifications/standards were used as references for this design.

[1] Universal Serial Bus Specification, Revision 2.0, December 21, 2000.

[2] Universal Serial Bus Mass Storage Device Class Bulk-Only Transport, Revision 1.0, September 31, 1999

[3] SCSI Architecture Model - 3 (SAM-3), Revision 9, September 12, 2003

[4] SCSI Block Commands - 2 (SBC-2), Revision 10, September 13, 2003

[5] SCSI Primary Commands - 3 (SPC-3), Revision 17, January 28, 2004

[6] Multimedia Commands - 4 (MMC-4), Revision 2d, September 2, 2003

[7] CF+ and CompactFlash, Revision 3.0

[8] MMC System Specification, Revision 3.31

APPENDIX A—MSD RD USB DESCRIPTOR DETAILS

The USB Descriptor used by the USB MSD RD is available in the module 'F34x_MSD_USB_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).

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 6. Drivers Loaded by Windows

Device Driver Stack	Driver
Generic Volume	File System
SiLabs Mass Storage USB Device	disk.sys
USB Mass Storage Device	usbstor.sys

CONTACT INFORMATION

Silicon Laboratories Inc.
4635 Boston Lane
Austin, TX 78735
Email: MCUinfo@silabs.com
Internet: www.silabs.com