

USB Mass Storage Device Using a PIC[®] MCU

*Author: Gurinder Singh
Microchip Technology Inc.*

INTRODUCTION

In recent years, there has been immense growth in Universal Serial Bus (USB) based applications, primarily due to the Plug and Play nature of USB.

This application note describes the design and implementation of a USB Mass Storage Device (MSD) using a Secure Digital card, which should prove useful to developers of USB mass storage solutions. This application may be used as a stand-alone MSD or as a Secure Digital/Multimedia Card (SD/MMC) reader/writer interface.

This design consists of the following components:

- USB V2.0 compliant PIC18F4550 microcontroller
- PICDEM[™] FS USB Demonstration Board
- PICtail[™] Board for SD[™] and MMC Cards
- Windows[®] operating system compatible (Me, 2000, XP and Windows Server[™] 2003)

Figure 1 shows the hardware configuration of the MSD. The PICtail[™] Board for SD[™] and MMC Cards (check the Microchip web site for availability) is connected into a socket on the PICDEM[™] FS USB Demonstration Board. For additional details, refer to the PICDEM FS USB Demonstration Board User's Guide (see "References").

The MSD design has the following features:

- USB V2.0 full-speed compliant.
- No custom drivers required (Windows operating system built-in driver, `usbstor.sys`, is used).

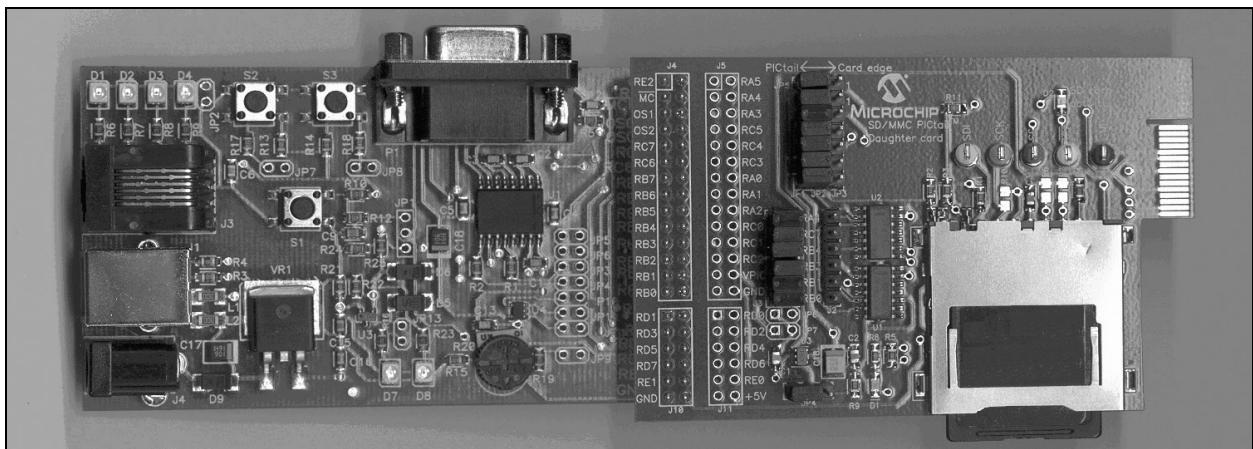
- Files created using FAT16, FAT32 or NTFS file format supported (see "References").
- SD and MMC cards supported (see "References").
- Uses the Windows operating system storage driver, `usbstor.sys`. The Windows Server 2003, Windows XP, Windows 2000 and Windows Me operating systems provide native support for USB Mass Storage Class devices. Therefore, MSD is compatible with these operating systems.

Version 1.0 of the MSD has the following limitations (later revisions will have additional features):

- Does not support the Windows 98 operating system (see **Appendix A: "Frequently Asked Questions"** for details).
- Does not support the FAT12 file system. In general, small capacity SD cards (i.e., ≤ 16 MB) can only be formatted in a FAT12 file system.
- If the SD card is removed, the USB cable must be disconnected and reconnected after the card has been reinserted.
- The SD card must be present at power-up.

Note: The implementation and use of the FAT file system, SD card specifications, MMC card specifications and other third party tools may require a license from various entities, including, but not limited to Microsoft[®] Corporation, SD Card Association and MMCA. It is your responsibility to obtain more information regarding any applicable licensing obligations. Some third party web sites have been listed in "References" for your convenience.

FIGURE 1: MSD HARDWARE CONFIGURATION



USB

A device endpoint is defined in the USB V2.0 specification as “a uniquely addressable portion of USB device that is the source or sink of information in a communication flow between the host and device” (see “**References**”). The unique address required for each endpoint consists of an endpoint number (which may range from 0 to 15) and direction (IN or OUT). The endpoint direction is from the host’s perspective; IN is towards the host and OUT is away from the host. An endpoint configured to do control transfers must transfer data in both directions, so a control endpoint actually consists of a pair of IN and OUT endpoints that share an endpoint number. All USB devices must have Endpoint 0 configured as a control endpoint.

USB V2.0 supports four types of data transfers: Control, Bulk, Interrupt and Isochronous.

Control transfer is used to configure a device at the time of plug-in and can be used for other device-specific purposes, including control of other pipes on the device.

Bulk data transfers are used when the data is generated or consumed in relatively large, bursty quantities.

Interrupt data transfers are used for timely, but reliable, delivery of data. For example, characters or coordinates with human perceptible echo or feedback response characteristics.

Isochronous data transfers occupy a pre-negotiated amount of USB bandwidth with pre-negotiated delivery latency (also called streaming real-time transfers).

For any given device configuration, an endpoint supports only one of the types of transfers described above. In this application, apart from Endpoint 0, we configure Endpoint 1 IN and OUT as bulk endpoints.

To meet the needs of various applications using USB, three speeds of operation have been designed in the USB V2.0 specification: Low-Speed (LS, 1.5 Mbps), Full-Speed (FS, 12 Mbps) and High-Speed (HS, 480 Mbps).

See “**References**” for detailed information on USB, including references to the USB specification and USB related publications.

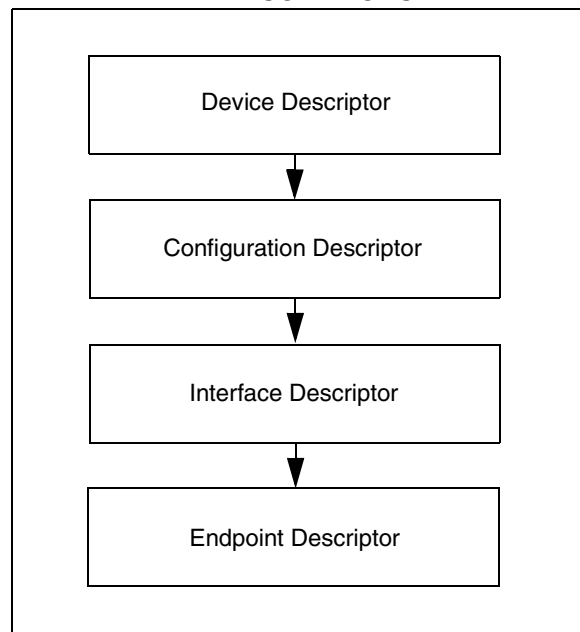
The PIC18F4550 used in this application is USB V2.0 compliant and can support LS and FS data transfers. For more information on the PIC18F4550, refer to the the device data sheet (see “**References**”).

Enumeration

Before the applications can communicate with the device, the host needs to learn about the device and assign a device driver. Enumeration is defined as the initial exchange of information that accomplishes this. During the enumeration process, the device moves through the following device states as defined by the USB V2.0 specification: Powered, Default, Address and Configured. Two other USB device states are: Attached and Suspend. Exact details of the enumeration process are beyond the scope of this document; however, the commands and structures used in the device configuration are briefly described.

Descriptors are data structures that enable the host to learn about a device. During enumeration, the host requests descriptors, starting from high-level device descriptors to low-level endpoint descriptors, in the sequence shown in Figure 2. The structure, description and values of device, configuration and interface descriptors are detailed in **Appendix C: “USB Descriptor Formats”**. A code example of descriptor structures is provided in **Appendix D: “USB Descriptor Structures”**. Details of bulk-only endpoint descriptors can be found in **Appendix F: “Bulk Endpoint Descriptors”**.

FIGURE 2: STANDARD USB DESCRIPTORS



ENUMERATION PROCESS

The following summarizes the steps involved in the enumeration of a USB device and explains how the device goes from Powered to Default, Address and the Configured state during the enumeration process.

1. User plugs a USB device into a USB port. The hub provides power to the port and the device is in the Powered state.
2. The hub detects the device.
3. The hub uses an interrupt pipe to report the event to the host.
4. Host sends `Get_Port_Status` request to obtain more information about the device.
5. Hub detects whether device is Low-Speed or Full-Speed operation and sends the information to the host in response to `Get_Port_Status`.
6. Host sends a `Set_Port_Feature` request, asking the hub to reset the port.
7. Hub resets the device.
8. Host learns if a Full-Speed device supports High-Speed operation (using Chirp K signal).
9. Host verifies if the device has exited the Reset state using `Get_Port_Status`.
10. At this point, the device is in the Default state (device is ready to respond to control transfers over the default pipe at Endpoint 0, default address is 00h and the device can draw up to 100 mA from the bus).
11. Host sends `Get_Descriptor` to learn the maximum packet size (Note: eighth byte of the device descriptor is `bMaxPacketSize`).
12. The host assigns an address by sending a `Set_Address` request. Device is now in the Address state.
13. Host sends `Get_Descriptor` to learn more about the device. The host responds by sending the descriptor followed by all other subordinate descriptors.
14. Host assigns and loads a device driver.
15. Host's device driver selects a configuration by sending a `Set_Configuration` request. The device is now in the Configured state.
16. Host assigns drivers for interfaces in composite devices.
17. If the hub detects an overcurrent, or if the host requests the hub to remove power, the device will be unpowered by the USB bus. In this case, the device and host cannot communicate and the device is in the Attached state.
18. If the device does not see any activity on the bus for 3 ms, it goes into the Suspend state. The device consumes minimal bus power in this state.

Control Transfer

Control transfer enables the host and the device to exchange information about device configuration and other control messages. Control transfers are ensured to have 10 percent of the bandwidth at Low-Speed and Full-Speed operation and 20 percent at High-Speed operation. A control transfer consists of a Setup stage, an optional Data stage and a Status stage.

Appendix E: "Standard USB Device Requests" summarizes the 11 USB standard control transfer requests, along with a description of each request. All USB devices must respond to these requests (even though the response may be just a STALL). Note that apart from the standard requests, a class may define requests for devices in its class. A class-specific request may be required or optional. For example, Mass Storage Devices may implement the `Get_Max_LUN` (Logical Unit Number) request that is used by the host to find out the number of logical units the device supports. The class-specific requests for the Mass Storage Devices are discussed in "**Mass Storage Class**".

Mass Storage Class

Bulk transfers are useful for transferring data when time is not a critical factor. Only High-Speed and Full-Speed devices can do bulk transfers. A bulk transfer can send large amounts of data without overloading the bus because it waits for the availability of the bus. The Mass Storage Class supports two transport protocols that determine which transfer type the device and host use to send command, data and status information. These two types of transport protocols are:

- Bulk-Only Transport (BOT)
- Control/Bulk/Interrupt (CBI) Transport

BOT is a data transport protocol that uses Bulk transport, whereas CBI transport uses Control transfer, Bulk transport and Interrupt transfer. CBI is further subdivided into a data transport protocol that uses Interrupt transfer and one that does not use Interrupt transfer. In this application, BOT is used as the data transport protocol.

AN1003

The Mass Storage Class specification (see “References”) defines two class-specific requests, `Get Max LUN` and `Mass Storage Reset`, that must be implemented by a Mass Storage Device. Bulk-Only Mass Storage Reset (`bmRequestType = 00100001b` and `bRequest = 11111111b`) is used to reset the Mass Storage Device and its associated interface. `Get Max LUN` (`bmRequestType = 10100001b` and `bRequest = 11111110b`) request is used to determine the number of logical units supported by the device. The value of Max LUN can vary between 0 and 15 (1-16 logical devices). Note that the LUN starts from 0. The device may share multiple logical units that share the common device characteristics. The host should not send the Command Block Wrapper (CBW) to a non-existing LUN.

The interface descriptor fields for configuring an interface as a Mass Storage Device implementing the BOT are shown in **Appendix C: “USB Descriptor Formats”**. Note that `bInterfaceClass = 08h` implies Mass Storage Class. Subclass code, `bInterfaceSubClass = 06h`, indicates that SCSI Primary Command-2 (SPC-2) definitions (see “References”) are supported by the device and the `bInterfaceProtocol = 50h` indicates the BOT implementation.

A device implementing BOT shall support at least three endpoints: Control, Bulk-In and Bulk-Out. The USB V2.0 specification defines a control endpoint (Endpoint 0) as the default endpoint that does not require a descriptor. The Bulk-In endpoint is used for transferring data and status from the device to the host, and the Bulk-Out endpoint is used for transferring commands and data from the host to the device. The endpoint descriptor values for configuring a Bulk-In and Bulk-Out endpoint are shown in **Appendix F: “Bulk Endpoint Descriptors”**.

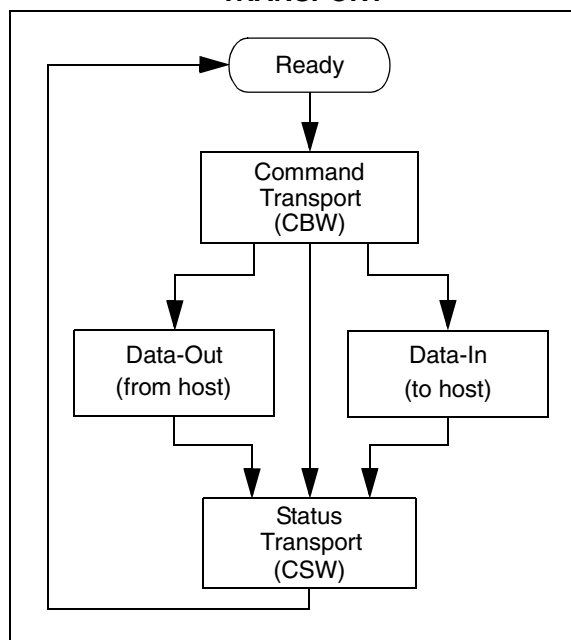
Bulk-Only Transport (BOT)

Like Control transfer, BOT also consists of a Command stage, an optional Data stage and a Status stage. The Data stage may or may not be present for all command requests. Figure 3 shows the flow of Command transport, Data-In, Data-Out and Status transport for BOT. The CBW is a short packet of exactly 31 bytes in length. The CBW and all subsequent data and Command Status Wrapper (CSW) start on a new packet boundary. It is important to note that all CBW transfers are ordered little-endian with LSB (byte 0) first.

Appendix G: “CBW and CSW” shows the format of a CBW packet. In the CBW, the `dCBWSignature` value, “43425355h” (little-endian), identifies a CBW packet. `dCBWTag` is the command block tag that is echoed back in CSW to associate the CSW with the corresponding CBW. `dCBWDataTransferLength` indicates the number of bytes the host expects to transfer on a Bulk-In or Bulk-Out endpoint (as indicated by the `Direction` bit). Only bit 7 of `bmCBWFlags` is used to indicate the direction of data flow, with a ‘1’ signifying Data-In (i.e., from device to host). The field, `bCBWLUN`, specifies the device LUN to which the command block is being sent. The field, `bCBWCB`, defines the valid length of the command block. The CBWCB is the command block to be executed by the device.

The size of a CSW is 13 bytes in length. A `dCSWSignature` value of “54425355h” (little-endian) identifies a CSW packet. The field, `dCSWTag`, echoes the `dCBWTag` value from the associated CBW. For Data-Out, `dCSWDataResidue` is the difference between the data expected and the actual amount of data processed by the device. For Data-In, it is the difference between the data expected and the actual amount of relevant data sent by the device. The value of `dCSWDataResidue` is always less than or equal to the value of `dCBWDataTransferLength`. The value of `bCSWStatus` indicates the success or failure of the command. The `bCSWStatus` value of 00h indicates command success, 01h indicates command failure, whereas 02h indicates phase error.

FIGURE 3: COMMAND/DATA/STATUS FLOW IN BULK-ONLY TRANSPORT



Secure Digital (SD) Card

A Secure Digital card is the most common storage media used in portable devices, such as PDAs, Digital Cameras and MP3 Players, among others. SD cards can be purchased with storage sizes ranging from 16 MB to 2 GB. Both SD cards and the MMC support the SPI™ transfer protocol and have an almost identical electrical interface. While the form factor and the shape of the SD card and the MMC are identical, SD cards can be operated up to four times faster, have a write-protect switch and may include cryptographic security for protection of copyrighted data. Due to these features, SD cards are more popular than MMC and are the focus of this design. However, MMC have been tested and found to be fully functional with the MSD design.

The SD card can be operated in SD Bus mode or SPI mode. In this application, the SD card is connected to the Serial Peripheral Interface (SPI) bus of the PIC18F4550 and operated in the SPI mode. In the SPI mode, only one data line is used for data transmission in each direction. The data transfer rate in this mode is therefore the same as the SD Bus mode with one data line (up to 25 Kbits per second).

Apart from the Power and Ground, the SPI bus consists of Chip Select (\overline{CS}), Serial Data Input (SDI), Serial Data Output (SDO) and Serial Clock (SCLK) signals. The SD card and MMC sample data input on the rising clock edge and set data output on the falling clock edge. On power-up, an SD card wakes up in the SD Bus mode. Therefore, an initialization routine is required to operate the SD card in SPI mode. This can be achieved by asserting the \overline{CS} signal (logic low) during the reception of the Reset command, CMD0. Unlike the SD Bus mode, in SPI mode, the selected card always responds to the command. In case of a data retrieval problem, the card responds with an error response instead of a time-out as in the SD Bus mode. See “**References**” for information on the SD card specification.

COMMUNICATION OVERVIEW

This section provides a general overview of the communication between the SD card and the Personal Computer (PC) application and system hardware.

Figure 4 shows the functional block diagram of the entire system. A device driver is defined as “*any code that handles communication details for a hardware device that interfaces to a CPU*”. In the layered driver model used in USB communications, each layer handles one part of the communication process. In this application, the MSD is enumerated as a Mass Storage Device implementing BOT. Therefore, the host uses the USB storage device driver (`usbstor.sys`) as the functional driver. The host loads `Disk.sys`, `PartMgr.sys` and `VolSnap.sys` as filter drivers to communicate between the end application and the device driver (`usbstor.sys`). The root hub driver (`usbhub.sys`) manages the port initialization and, in general, manages the communications between device drivers and the bus class driver. The bus class driver (`usbd.sys`) manages bus power, enumeration, USB transactions and communications between the root hub driver and the host controller driver.

On the MSD application side, the Serial Interface Engine (SIE) of the PIC18F4550 handles the low-level USB communications. USB data moves between the microcontroller core and the SIE through a memory space known as the USB RAM. The PIC18F4550 provides the capability to configure and control up to 16 bidirectional endpoints. In this application, two bidirectional endpoints are used. Endpoint 0 is required for all USB devices for control transfers and does not require configuration. The mass storage application configures Endpoint 1 IN and OUT as bulk endpoints for Bulk-Only Transport. It also communicates with the SD card's SPI bus to read the data from and write the data to the SD card.

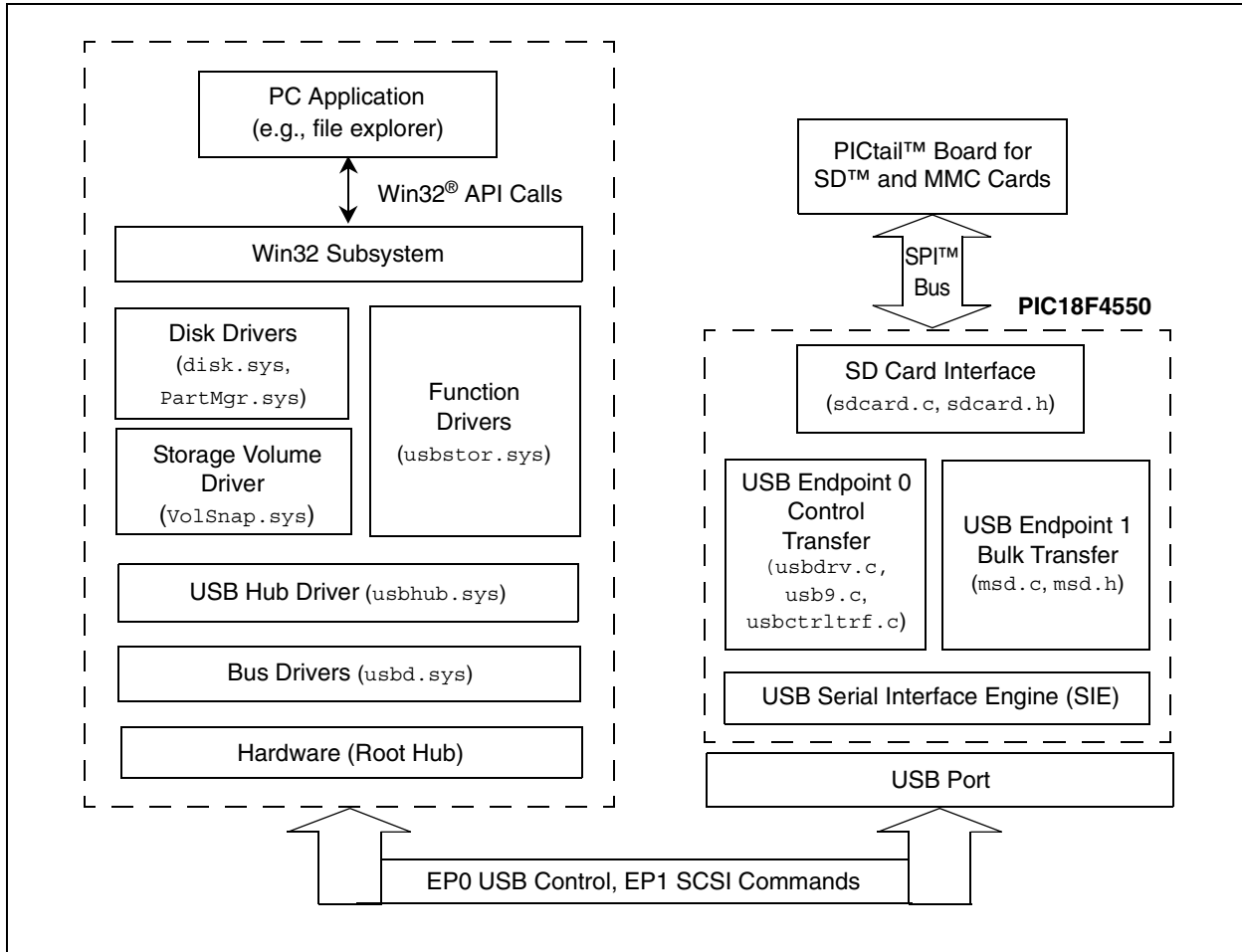
AN1003

Hardware

The PICDEM FS USB Demonstration Board is used as the platform for developing the USB MSD application. The PICtail™ Board for SD™ and MMC Cards is connected on the expansion headers, J6 and J7, on the PICDEM FS USB Demonstration Board. The USB V2.0 compliant PIC18F4550 microcontroller forms the heart of the PICDEM FS USB Demonstration Board. The PIC18F4550 has an on-chip USB voltage regulator, transceivers and pull-up resistors to minimize the

number of external components and enable a low-cost design. On a side note, the PICDEM FS USB Demonstration Board also features a potentiometer, simulating analog input for the controller and a digital temperature sensor. While not being used in the USB SD card mass storage application, these features may be useful in developing other USB applications. More details of the PICDEM FS USB Demonstration Board can be found in the PICDEM FS USB Demonstration Board User's Guide (see "References").

FIGURE 4: PC, MSD COMMUNICATION BLOCK DIAGRAM



The SD card operates in the 2.7-3.6V range, whereas the PIC18F4550 on the PICDEM FS USB Demonstration Board requires a 5V VDD. The voltage regulation from 5V to 3V is achieved using Microchip's TC1186-3.3VCT713.

Appendix J: "Schematic" shows the schematic for the PICtail™ Board for SD™ and MMC Cards (Part No. AC164122). Pin 7 (RA5) of the PIC18F4550 is connected to the SHDN (Shutdown input) pin of the TC1186 via jumper JP4 (position 2-3). This enables the user to turn off the SD card power using firmware (setting RA5 turns the power on). Alternately, by changing the jumper JP4 setting (position 1-2), the user may turn the SD card power on permanently. The PICtail™ Board for SD™ and MMC Cards also implements two MC74VHCT125A devices as signal level translators for translating 5V DC signals on the microcontroller side to 3.3V DC signals on the SD card side and vice versa. The MC74VHCT125A is a high-speed CMOS quad buffer fabricated with silicon gate CMOS technology. Since it has a full 5.0V CMOS level output swing, this device is ideal for signal level translation between 3.0V and 5.0V levels. For 5V to 3.3V signal translation, a supply voltage of 3.3V is applied to U1 and corresponding OE pins are enabled (active-low). Similarly, 3.3V to 5V translation is achieved by applying a 5V supply to U2 and enabling the corresponding OE pins. Further details on the MC74VHCT125A can be found in its respective data sheet (see "References").

<p>Note: The user application may not require the signal translation if the PIC microcontroller is operated at 3V.</p>

The PICtail™ Board for SD™ and MMC Cards also features a power LED (D1), an SD card activity LED (D2) and test points for monitoring the SPI bus. LED D1 indicates that power is available at the output of the TC1186. LED D2 indicates SD card activity (based on Chip Select signal, \overline{CS}). The Serial Data Input (SDI), Serial Clock (SCK), Serial Data Output (SDO), Chip Select (\overline{CS}) and Ground (GND) signals of the SPI bus can be monitored on the test points. The PICtail™ Board for SD™ and MMC Cards is designed to operate with a multitude of demonstration boards, including all demonstration boards having PICtail signals, Explorer 16 development board having card edge connectors and demonstration boards with non-standard PICtail signals. The following jumper settings must be used for operating the PICtail™ Board for SD™ and MMC Cards with different demonstration boards.

1. If the demonstration board has standard PICtail signals, connect JP1-JP2 and JP5 on the PICtail side (default setting).
2. If the demonstration board provides card edge signals, connect JP2-JP3 and JP5 on the card edge side.
3. If the demonstration board does not provide standard PICtail signals, open J3, jump signals from J5 and J11 to appropriate signals on J2.

SCSI Commands

After the successful enumeration of the target USB device, the host initiates commands according to the `bInterfaceSubClass` specified in the interface descriptor during the enumeration process.

The USB MSD application specifies `bInterfaceSubClass = 06h`, indicating that the device will support SCSI Primary Commands-2 (SPC-2) or later. A `bInterfaceProtocol` value of `0x50` in the interface descriptor indicates that the BOT protocol is being used. As shown in Figure 3, a BOT transfer begins with a CBW. The device indicates the successful transport of a CBW by accepting (ACKing) the CBW. If the host detects a STALL of the Bulk-Out endpoint during Command transport, the host shall respond with a Reset recovery. The host shall attempt to transfer an exact number of bytes to or from the device as specified by the `dCBWDataTransferLength` and the `Direction` bit. The device shall send each CSW to the host via the Bulk-In endpoint.

In this section, we briefly describe the SCSI commands that are supported in the MSD implementation. The reader may refer to SCSI Primary Commands-3 (SPC-3) and SCSI Block Commands-2 (SBC-2) specifications for further details (see "References"). The first byte of the command block, `CBWCB`, is always the operation code or opcode in short.

- **INQUIRY** (Opcode 12h)

The **INQUIRY** command requests that the information regarding the logical unit and SCSI target device be sent to the application client (host). The SPC-3 specification requires that the **INQUIRY** data should be returned even though the device server is not ready for other commands. Moreover, the standard **INQUIRY** data should be available without incurring any media access delays. The standard **INQUIRY** data is at least 36 bytes.

- **READ CAPACITY** (Opcode 25h)

The **READ CAPACITY** command requests that the device server transfer bytes of parameter data describing the capacity and medium format to the Data-In buffer. The response to the **READ CAPACITY** command is 4 bytes of returned `Logical Block Address` and 4 bytes of block length in bytes. `Returned Logical Block Address (LBA)` is the LBA of the last logical block on the direct access block device. If the number of logical blocks exceeds the maximum value that can be specified in the returned `Logical Block Address` field, the device shall set the returned `Logical Block Address` field to `FFFFFFFFh`.

- **READ (10)** (Opcode 28h)

The **READ (10)** command specifies that the device server read the specified logical block(s) and transfer them to the Data-In buffer. The **READ (10)** command is a 10-byte CBWCB with the eighth and ninth bytes specifying the **TRANSFER LENGTH** (see **Appendix I: “SCSI Command and Data Format”**). The **TRANSFER LENGTH** field specifies the number of contiguous logical blocks of data that shall be read and transferred to the Data-In buffer, starting with the logical block specified by the **Logical Block Address** field (bytes 3-6). A **TRANSFER LENGTH** field set to zero specifies that no logical blocks shall be read.

- **WRITE (10)** (Opcode 2Ah)

The **WRITE (10)** command requests that the device server transfer the specified logical blocks from the Data-Out buffer and write them. The CBWCB format for the **WRITE (10)** command is the same as the **READ (10)** command with **TRANSFER LENGTH** specifying the number of contiguous logical blocks of data that shall be transferred from the Data-Out buffer and written, starting with the logical block specified by the **Logical Block Address** field. A **TRANSFER LENGTH** field set to zero specifies that no logical blocks shall be written.

- **REQUEST SENSE (6)** (Opcode 03h)

The **REQUEST SENSE (6)** command requests that the device server transfer the sense data to the application client. **Appendix H: “SCSI Command Set”** shows the fixed format sense data response. The contents of the **RESPONSE CODE** field indicate the error type and format of the sense data. The **RESPONSE CODE 70h** signifies the current error, **RESPONSE CODE 71h** signifies the deferred error in the fixed format sense data and code values, **72h** and **73h**, indicate the current and deferred error code in the descriptor format sense data. The **SENSE KEY**, **ADDITIONAL SENSE CODE (ASC)** and **ADDITIONAL SENSE CODE QUALIFIER (ASCQ)** fields provide a hierarchy of information. The **SENSE KEY** field indicates the generic information describing an error or exception condition, **ASC** indicates further information related to the error reported in the **SENSE KEY** field, whereas the **ASCQ** field indicates the detailed information related to the **ADDITIONAL SENSE CODE**. Refer to Table 27 and Table 28 of the SPC-3 specification (see **“References”**) for a list of **SENSE KEY** error codes and **ASC** and **ASCQ** error code assignments. This application implements the fixed format, current error code sense data response (defined in `~\system\usb\class\msd\msd.h`).

- **MODE SENSE (6)** (Opcode 1Ah)

The **MODE SENSE (6)** command provides a means for a device server to report parameters to an application client. It is a complementary command to the **MODE SELECT (6)** command. The mode parameter header that is used by the **MODE SENSE (6)** and the **MODE SELECT (6)** command is shown in **Appendix H: “SCSI Command Set”**. The **MEDIUM TYPE** and **DEVICE SPECIFIC PARAMETER** fields are unique for each device type. In this application, the **MEDIUM TYPE** field is set to 00h, indicating a direct access block device. This value is the same as the value of the **PERIPHERAL DEVICE TYPE** field in the standard **INQUIRY** data.

- **PREVENT ALLOW MEDIUM REMOVAL** (Opcode 1Eh)

The **PREVENT ALLOW MEDIUM REMOVAL** command requests that the logical unit enable or disable the removal of the medium. The prevention of medium removal shall begin when an application client issues a **PREVENT ALLOW MEDIUM REMOVAL** command with a **PREVENT** field (fifth byte, bits 0-1 of CBWCB) of 01b or 11b (i.e., medium removal prevented). Since in an SD card, there is no way to prevent card removal, the firmware decodes the command and prepares to notify the host PC that the operation has been successfully completed. If the medium is inaccessible, the command is specified as Fail (**bCSWStatus** = 0x01) with the **SENSE KEY** set to Not Ready.

- **TEST UNIT READY** (Opcode 00h)

The **TEST UNIT READY** command provides a means to check if the logical unit is ready. This is not a request for self-check. If the logical unit is able to accept an appropriate medium access command, without returning a Check Condition status, this command shall return a Good status. Otherwise, the command is terminated with the Check Condition status and the **SENSE KEY** is set to reflect the error condition.

- **VERIFY (10)** (Opcode 2Fh)

The **VERIFY (10)** command requests that the device server verify the specified logical block(s) on the medium. If the **BYTCHK** bit is set to '0', the device shall perform medium verification with no data comparison and not transfer any data from the Data-Out buffer. If the **BYTCHK** bit is set to '1', the device server shall perform a byte-by-byte comparison of user data read from the medium and user data transferred from the Data-Out buffer. The firmware decodes the command and then prepares to notify the host PC that the command has been successfully completed. If the medium is inaccessible, the command is specified as Fail, with the **SENSE KEY** set to Not Ready.

- **START/STOP** (Opcode 1Bh)

The **START/STOP** command requests that the device server change the power condition of the logical unit, or load, or eject the medium. This includes specifying that the device server enable or disable the direct access block device for medium access operations by controlling power conditions and timers. The **POWER CONDITION** field (fifth byte, bit 7-4) is used to specify that the logical unit be placed into a power condition or to adjust a timer. If the value of this field is not equal to 0h, the **START** (fifth byte, bit 0) and **LOEJ** (Load Eject, fifth byte, bit 1) bits are ignored. If the **POWER CONDITION** field is Active (1h), Idle (2h) or Standby (3h), then the logical unit shall transition to the specified power. If **POWER CONDITION** = 0h (**START_VALID**), then the **START**, **LOEJ** = (0, 0) signifies that the logical unit shall transition to the stopped power condition; **START**, **LOEJ** = (0, 1) signifies the logical unit shall unload the medium; **START**, **LOEJ** = (1, 0) signifies that the logical unit shall transition to the active power condition. If **START**, **LOEJ** = (1, 1), then the logical unit shall load the medium.

UNSUPPORTED COMMANDS

If the command opcode field in the **CBWCB** is not supported, the **SENSE KEY** is set to Illegal Request, indicating that there was an illegal parameter in the **CDB** with **ASC** and **ACSQ** codes set corresponding to an invalid command opcode.

MASS STORAGE DEVICE (MSD) FIRMWARE

This firmware implements a USB-based Mass Storage Device using an SD card. When plugged into the USB port, the firmware enumerates the SD card as a removable disk drive and allows the user to exercise all standard features of a disk drive. The user can write, read, edit and delete files on the MSD just like any other removable disk media. This application also allows the user to format the SD card in any of the following FAT file formats: FAT16, FAT32 or NTFS (Windows drivers handle the format, firmware is only required to implement the SCSI commands). The firmware calculates the capacity of the SD card based on the Card Specific Data (CSD) register read from the SD card. This information is conveyed to the PC host in response to the **READ CAPACITY** command. The exact size of the disk can be seen in the disk properties on the PC. Further details on firmware and SCSI command implementation can be found in “**SCSI Commands**”.

The project framework is organized under a single root directory with each subdirectory containing files for each category or class of source code. If the SD card is not found, or not initialized, the 4 LEDs (D1, D2, D3 and D4) on the demonstration board are turned on permanently.

Using the PICtail™ Board for SD™ and MMC Cards

First, connect the demonstration board to the MPLAB® ICD 2 and program the device. The following steps are required to run the MSD application:

1. Connect the PICtail™ Board for SD™ and MMC Cards to the PICDEM FS USB Demonstration Board as shown in Figure 1.
2. Insert the SD card in the reader slot, making sure that write-protect is disabled on the SD card.
3. Apply power to the demonstration board.
4. Observe the LEDs (D1, D2, D3 and D4) on the demonstration board. If all of the LEDs are ON, this indicates an SD card initialization failure.
5. If no errors have occurred, connect a USB cable from the PC to the USB connector on the demonstration board.
6. Observe under **Control Panel > System > Hardware > Device Manager** (for Windows XP system) that **USB Mass Storage Device** gets enumerated under **Universal Serial Bus controllers**. Verify that the **Microchp Mass Storage USB Device** is enumerated under **Disk drive** and **Generic volume** is enumerated under **Storage volumes**, as shown in Figure 5.
7. Look for a removable drive under **My Computer**.
8. Use the removable drive icon to access the SD card as a MSD.
9. Before removing the USB cable from the demo board, click the “**Safely Remove Hardware**” icon in the Windows task bar.

The four LEDs on the PICDEM FS USB Demonstration Board have been implemented as follows:

1. LED D1 turns ON after successfully responding to the **INQUIRY** command.
2. LED D2 toggles ON after each successful **TEST UNIT READY** command execution.
3. LED D3 blinks during read operation (turns ON while a read from the SD card is taking place).
4. LED D4 blinks during write operation (turns ON while a write to the SD card is taking place).

FIGURE 5: DEVICE MANAGER



Directory Structure

The file structure consists of a collection of sub-directories containing specific files under a root project directory. The user may create the root project directory in any location with a valid directory name. The sub-directories structure should always be maintained. The basic directory structure is similar to the PICDEM FS USB demonstration code. This backward compatibility has been maintained to ensure that users already familiar with PICDEM FS USB demonstration code can easily integrate this mass storage application.

Figure 6 shows the directory structure for the MSD application.

Function Description

Table 1 and Table 2 provide brief descriptions of the functions contained in files `msd.c` and `sdcard.c`, respectively.

FIGURE 6: MSD DIRECTORY STRUCTURE

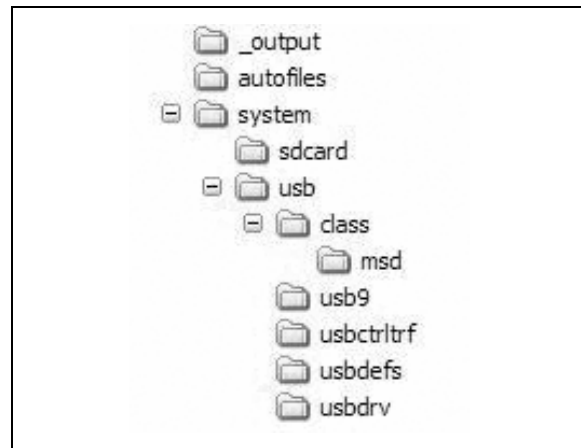


TABLE 1: msd.c FUNCTIONS

Function Name	Description
USBCheckMSDRequest	Handles the class-specific requests received on Endpoint 0.
ProcessIO()	Handles MSD requests on Endpoint 1.
MSDInitEP	Initializes Bulk-In and Bulk-Out endpoints (MSD_BD_IN, MSD_BD_OUT).
SDCardInit	Initializes the SD card in SPI™ mode.
MSDCommandHandler	Decodes and processes the received SCSI command.
MSDInquiryHandler	Executes the INQUIRY command.
MSDReadCapacityHandler	Executes the READ CAPACITY command.
MSDReadHandler	Executes the READ (10) command.
MSDWriteHandler	Executes the WRITE (10) command.
MSDModeSenseHandler	Executes the MODE SENSE (6) command.
MSDMediumRemovalHandler	Executes the PREVENT ALLOW MEDIUM REMOVAL command.
MSDRequestSenseHandler	Executes the REQUEST SENSE (6) command.
MSDTestUnitReadyHandler	Executes the TEST UNIT READY command.
MSDVerifyHandler	Executes the VERIFY (10) command.
MSDStopStartHandler	Executes the START/STOP command.
IsMeaningfulCBW	Checks if the received CBW is meaningful.
IsValidCBW	Checks if the received CBW is valid.
PrepareCSWData	Prepares CSW data (Tag and Signature).
SendData(byte* Y, byte X)	Sends X bytes of data starting at location Y.
SendCSW	Sends the CSW and sets MSD_State to MSD_WAIT.
ResetSenseData	Initializes the sense response data.
MSDDataIn	Sends data to the host.
MSDDataOut	Reads data from the host.

TABLE 2: sdcard.c FUNCTIONS

Function	Description
SDC_Error MediaInitialize(SDCSTATE*)	Initializes the SD card in SPI™ mode and reads its first sector.
SocketInitialize	Initializes card select, detect signals and socket interface.
SDC_Error SectorRead(dword SN, byte* buff)	Reads the specified sector SN into buffer (msd_buffer).
SDC_Error SectorWrite(dword SN, byte* buff)	Writes the data pointed to by buffer into sector SN.
SDC_Error CSDRead	Reads the CSD register from SD card.
MediaDetect	Returns True if SD card is detected.
SDC_Response SendSDCCmd(byte, dword)	Sends SDC command packet on SPI interface.
ReadMedia	Reads in one byte of data from SPI port while sending out 0xFF to SD card.

AN1003

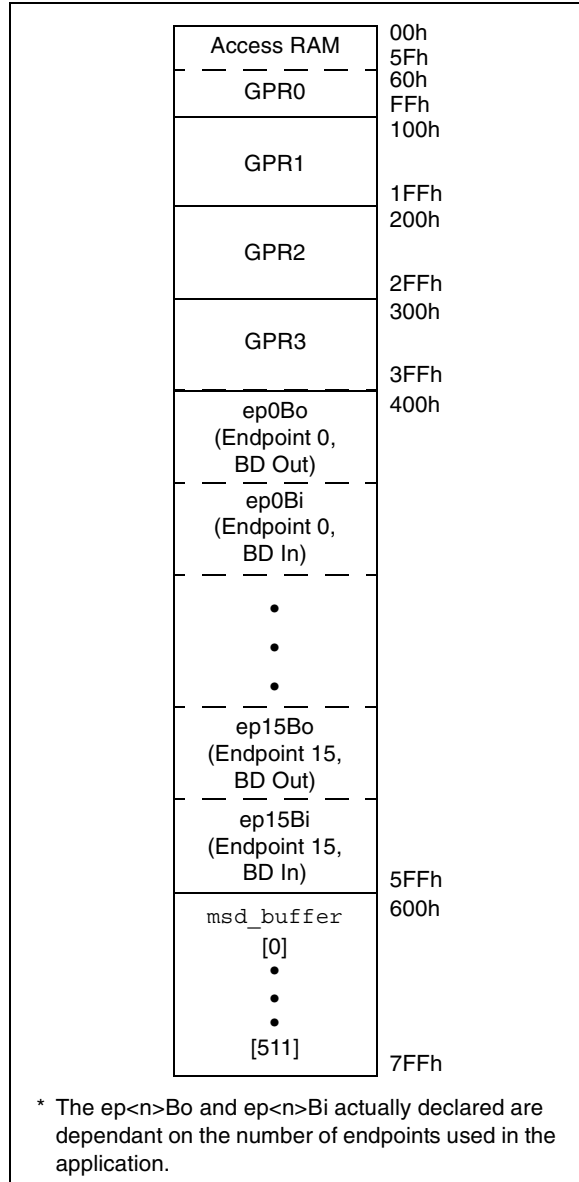
Memory Organization

Data banks 4 through 7 of the data memory are mapped to special dual port RAM (see Example 2). When the USB module is disabled, the General Purpose Registers (GPRs) in these banks are used like any other GPRs in the data memory space. When the USB module is enabled, the memory in these banks is allocated as buffer RAM for USB operation. This area is shared between the microcontroller core and the SIE and is used to transfer data directly between the two. Note that the linker script has been modified to define MSD as a single data bank of 512 bytes. The 512-byte `msd_buffer` has been defined in the MSD data bank (see Example 1). Figure 7 shows the entire memory map including the endpoint buffers.

EXAMPLE 1: BUFFERS FOR MSD (`usbmmmap.c`)

```
#if defined(USB_USE_MSD)
volatile far USB_MSD_CBW msd_cbw;
volatile far USB_MSD_CSW msd_csw;
#pragma udata myMSD=0x600
volatile far char msd_buffer[512];
#endif
```

FIGURE 7: COMPLETE MEMORY ORGANIZATION (INCLUDING ENDPOINT BUFFERS)



EXAMPLE 2: MODIFIED LINKER SCRIPT

```
ACCESSBANK NAME=accessram START=0x0 END=0x5F
DATABANK NAME=gpr0 START=0x60 END=0xFF
DATABANK NAME=gpr1 START=0x100 END=0x1FF
DATABANK NAME=gpr2 START=0x200 END=0x2FF
DATABANK NAME=gpr3 START=0x300 END=0x3FF
DATABANK NAME=usb4 START=0x400 END=0x4FF PROTECTED
DATABANK NAME=usb5 START=0x500 END=0x5FF PROTECTED

// Combine usb6 and usb7 banks to define a 512 byte msd bank .....
//DATABANK NAME=usb6 START=0x600 END=0x6FF PROTECTED
//DATABANK NAME=usb7 START=0x700 END=0x7FF PROTECTED

DATABANK NAME=msd START=0x600 END=0x7FF PROTECTED
```

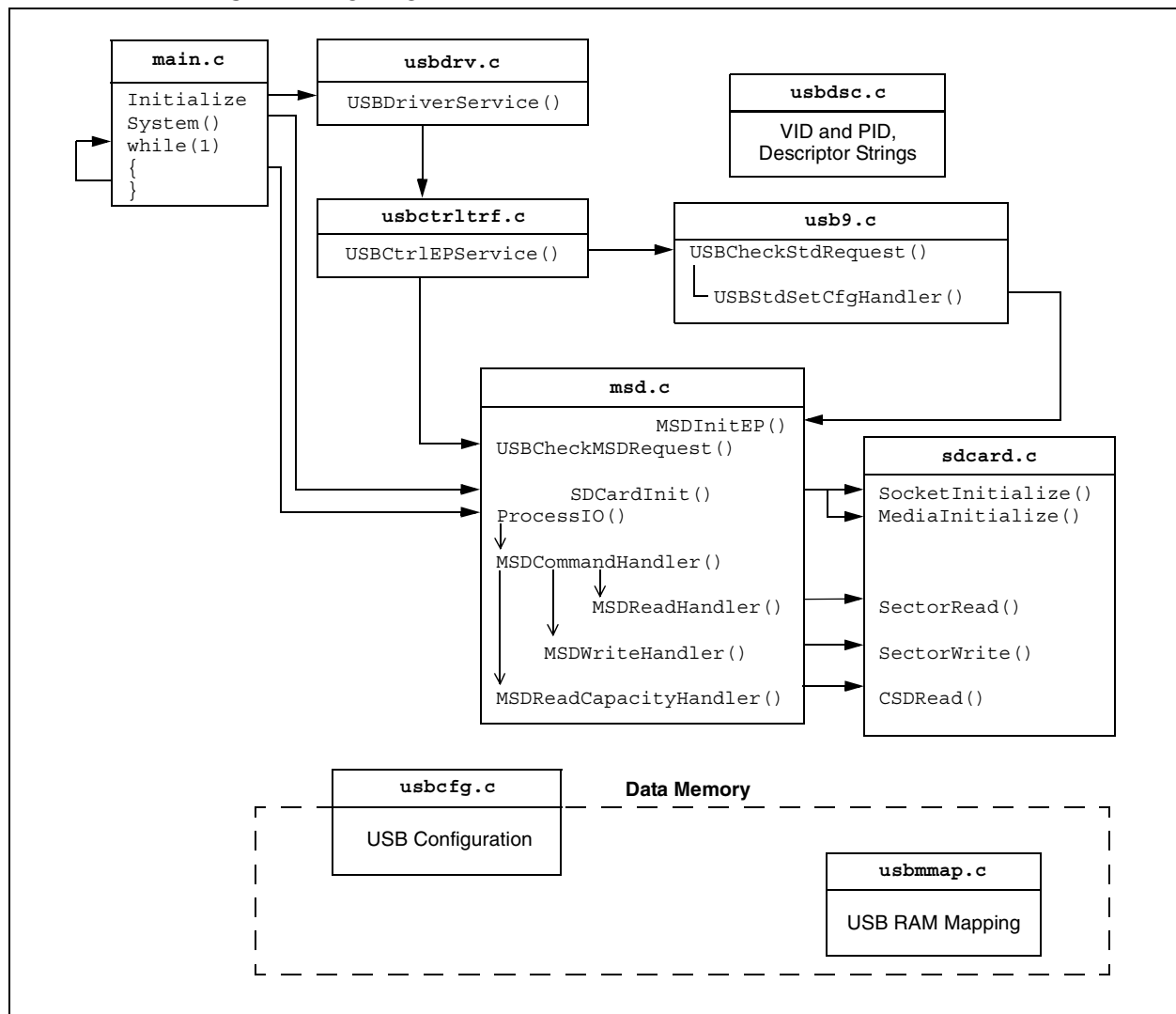
Firmware Description

Figure 8 shows the relationship between various files in this firmware. The details of the USB framework files can be found in the user's guide for the PICDEM FS USB Demonstration Board. This application note focuses on the USB mass storage application and communication with the SD card.

A USB request can be either standard or class-specific. A standard request is serviced by the `USBCheckStdRequest()` which handles the standard requests as specified in Chapter 9 of the USB V2.0 specification. A Mass Storage Class specific request is handled by the firmware in `msd.c`. If the

`USBCheckStdRequest()` cannot process the requests, it calls `USBCheckMSDRequest()`. `USBCheckMSDRequest()` checks if it is a class-specific request (`SetupPkt.RequestType == CLASS`) and `SetupPkt.bRequest == FFh` (Bulk-Only Mass Storage Reset) or `FEh` (`Get Max LUN`). If a Bulk-Only Mass Storage Reset request is received, the firmware disables Endpoint 1, clears the STALL and reinitializes Endpoint 1. The response to the `Get Max LUN` request is one byte that consists of the maximum LUN supported by the device. For example, if the device supports three LUNs, then the LUNs would be numbered from 0 to 2 and the return value would be '2'. In our case, the number of LUNs is 1, so the return value is '0'.

FIGURE 8: RELATIONSHIP BETWEEN USB FRAMEWORK DEMONSTRATION FILES AND THE MSD APPLICATION



The USB enumeration process is handled mainly in `usb9.c`. The `SET_CONFIGURATION` request is handled by `USBStdSetCfgHandler()`. This function calls the function, `MSDInitEP()`. The function, `MSDInitEP()`, configures and initializes a Bulk-In and a Bulk-Out endpoint.

The `main()` function in file `main.c` is an infinite loop that services different tasks – USB or mass storage application tasks. USB tasks are handled by `USBDriverService()` which handles all USB hardware interrupts. The mass storage application tasks are handled by `ProcessIO()`. `ProcessIO()` forms the core of the handling of mass storage communications on Endpoint 1. Figure 9 shows the flowchart of the `ProcessIO()`.

When Endpoint 1 is initialized, the `MSD_State` is set to `MSD_WAIT`. The firmware basically waits for a CBW to be received on Endpoint 1. Upon receiving a valid and meaningful CBW (as defined in the USB Mass Storage Class Bulk-Only Transport specification, see “References”), the CSW data is prepared. Basically, the `dCBWTag` is copied to `dCSWTag` in order to associate the CSW with the corresponding CBW and the `dCSWSignature` field is set to “53425355h” (little-endian). The `Direction` bit is read to find the direction of data transfer (i.e., from host to device or vice versa) and sets `MSD_State` to `MSD_DATA_OUT` or `MSD_DATA_IN`, respectively (see Figure 10). Further, the first byte of `CBWCB` is the operation code of the received command. This is used to decode the command and take the appropriate action (`MSDCommandHandler`). It may happen that the command does not require any data transfer. The `Direction` bit is ‘0’ in this case and the `MSD_State` is set to `MSD_DATA_OUT`. If there is no data transfer required for a given command, the command is executed and the status is sent using `sendCSW()`. The values of the `dDataResidue` and `bCSWStatus` fields are set based on the result of the command execution.

Figure 11 shows the flowchart of the `MSDDataIn()` function. This function is used to send the data prepared while processing the command in `MSDCommandHandler()`, from the device to the host, using `MSD_BD_IN`. After command execution, `dCSWDataResidue` reflects the number of bytes of data obtained as a result of the command execution that are to be sent to the host. In case of an error (`bCSWStatus! = 0x00`), zero padded data of the size expected by the host (`dCBWDataTransferLength`) is sent. If there is no error, the size of data to be sent (`dCSWDataResidue`), as a result of command execution, may not be the same as the size expected by the host (`dCBWDataTransferLength`). In this case, the `dCSWDataResidue` field in the CSW will reflect the difference. If the data to be sent is greater than `MSD_IN_EP_SIZE` (64 bytes, size of the Endpoint 1 IN

buffer), then `MSD_IN_EP_SIZE` bytes of data are sent; otherwise, the remaining `dCSWDataResidue` bytes of data are sent using the `MSD_BD_IN` buffer.

Note that the only command where data needs to be read from the host is the `WRITE (10)` command. In the `MSD_WAIT` state, the `MSD_BD_OUT` points to the `msd_cbw` structure in order to read the next command block. But when a `WRITE (10)` command is received, the device changes to `MSD_DATA_OUT`. In this state, the device must read more data from the host and write it to the SD card. This is done using the 512-byte `msd_buffer`. So, in the `MSD_DATA_OUT` state, the `MSD_BD_OUT` (Endpoint 1 OUT) buffer points to `msd_buffer`. In order to read the entire 512-byte data block, after every read, the `MSD_BD_OUT` points to a location in the `msd_buffer` incremented by `MSD_OUT_EP_SIZE` (size of the Endpoint 1 OUT buffer). Once the `msd_buffer` is filled (8 reads of 64 bytes), the block of data is written to a specific location in the SD card using the `SECTORwrite(...)` function (defined in `sdcard.c`). This process is repeated if multiple blocks of data are to be written to the SD card. The `LBA` field of the `WRITE (10)` `CBWCB` gives the information about the starting `LBA` and the `TRANSFER_LENGTH` field indicates the number of contiguous `LBAs` to be written.

EXAMPLE 3: STRUCTURE FOR COMMAND BLOCK WRAPPER

```
typedef struct _USB_MSD_CBW
{
    dword dCBWSignature;
    dword dCBWTag;
    dword dCBWDataTransferLength;
    byte bCBWFlags;
    byte bCBWLUN;
    byte bCBWCBLength;
    byte CBWCB[16];
} USB_MSD_CBW;
```

Out endpoint size is configured as 64 bytes. The `msd_buffer` is a 512-byte buffer declared in the USB dual port RAM area. The block size of the SD card is 512 bytes. The `msd_buffer` is used to read 512 bytes from the host using multiple 64-byte reads from `MSD_BD_OUT`. Once 512 bytes of data are read from the host (`msd_buffer` is filled), the entire block of data is written to the SD card using the function, `SECTORWrite()`. For `WRITE (10)` commands where the `TRANSFER_LENGTH > 1`, multiple blocks of 512 bytes of data are written to consecutive sectors, starting with the `Logical Block Address` field in the command block. The translation between `LBA` and the physical address is as follows: since each sector has $2^9 = 512$ bytes, the physical address is obtained by left shifting the `LBA` by 9 positions. Similarly, for `READ (10)`, a block of 512 bytes of data is read from the SD card using the function, `SECTORread()`, and then transmitted to the host in 64-byte packets using the `MSD_BD_IN`

buffer. Note that after each read or write from the MSD_BD_OUT or MSD_BD_IN, mUSBDriverService() is called to clear the TRNIF bit. The macro, mUSBBufferReady(MSD_BD_IN) or mUSBBufferReady(MSD_BD_OUT), is called to write or read the data from the corresponding Buffer Descriptor (BD) register. This call should be made after arming the corresponding BD registers. The mUSBBufferReady(...) macros toggle the Data Toggle Sync (DTS) bit and give the buffer ownership to SIE.

The firmware only supports the fixed format response to the INQUIRY command. According to the specifications, a media access delay must not be incurred in responding to the INQUIRY command. The INQUIRY data format and the values stored in ROM for this application are shown in **Appendix H: “SCSI Command Set”**. The standard INQUIRY data is at least 36 bytes, but can be up to 96 bytes, excluding the vendor-specific parameters as described in the SPC-3 specification.

An 8-byte response indicating the total number of LBAS and the block length in bytes is expected for the READ CAPACITY command. To obtain this information, we read the Card Specific Data (CSD) from the SD card by calling the CSDread(...) function (defined in sdcard.c). The CSDread function issues the SPI command, CSD_READ, to the SD card and reads the response in the global variable, gblCSDReg. The card capacity (not including the security protected area) can be computed from the C_SIZE, C_SIZE_MULT and READ_BL_LEN fields from the CSD register.

Memory capacity = BLOCKNR * BLOCK_LEN, where BLOCKNR = (C_SIZE + 1) * MULT and MULT = $2^{C_SIZE_MULT + 2}$.

The block length (BLOCK_LEN) can be computed using:

- READ_BL_LEN = WRITE_BL_LEN
- BLOCK_LEN = $2^{READ_BL_LEN}$

FIGURE 9: ProcessIO() FLOWCHART

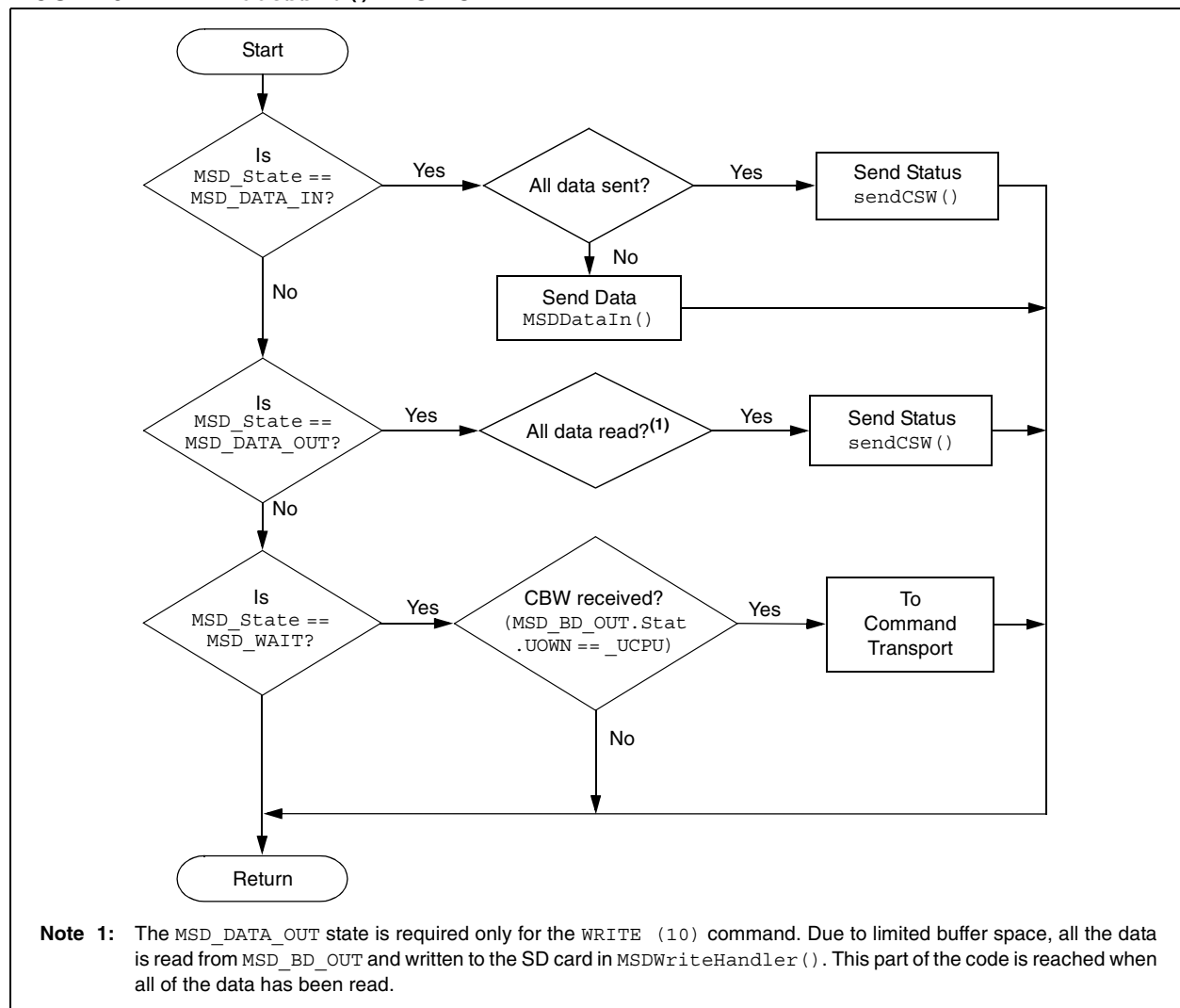


FIGURE 10: COMMAND TRANSPORT FLOWCHART

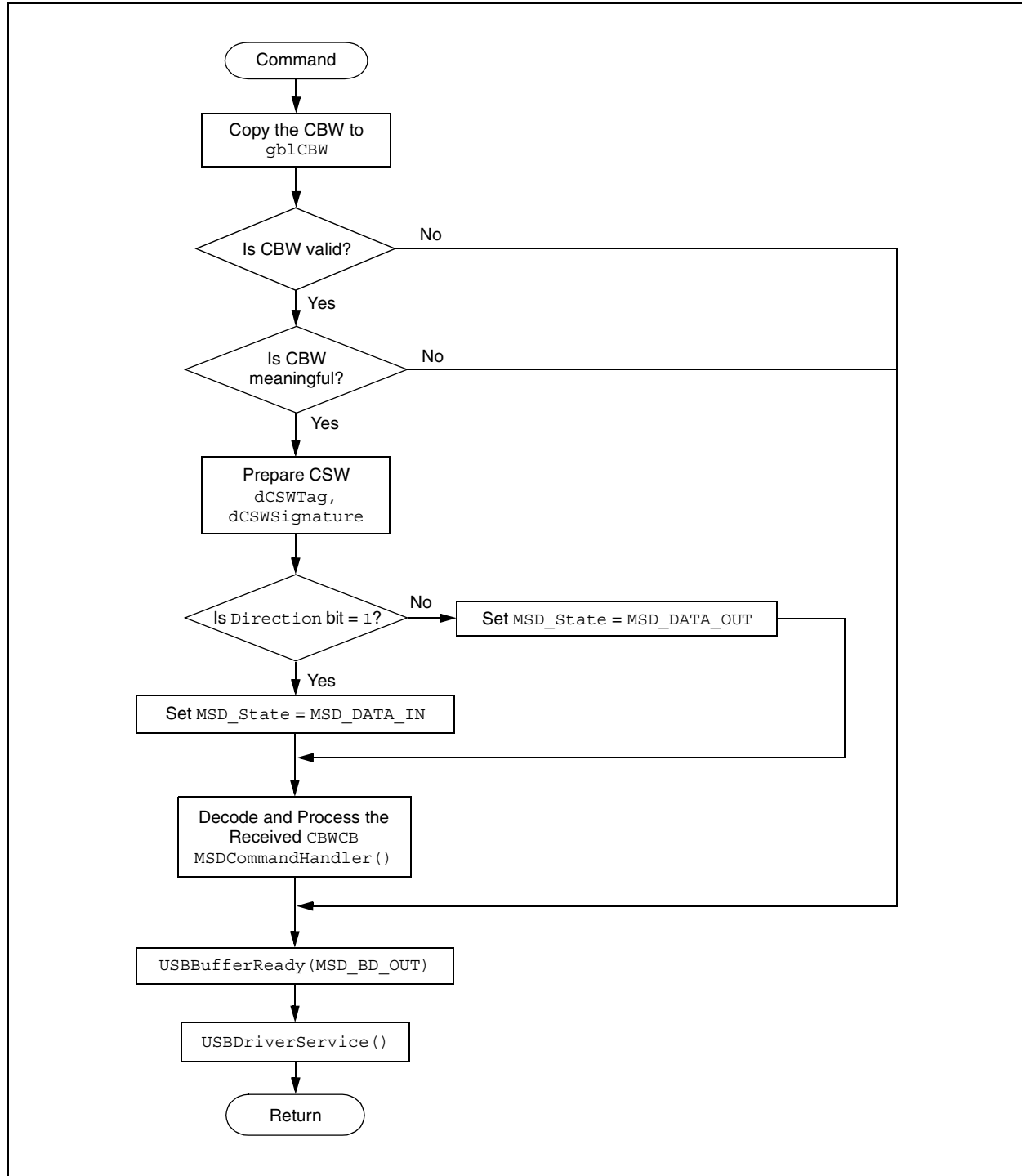
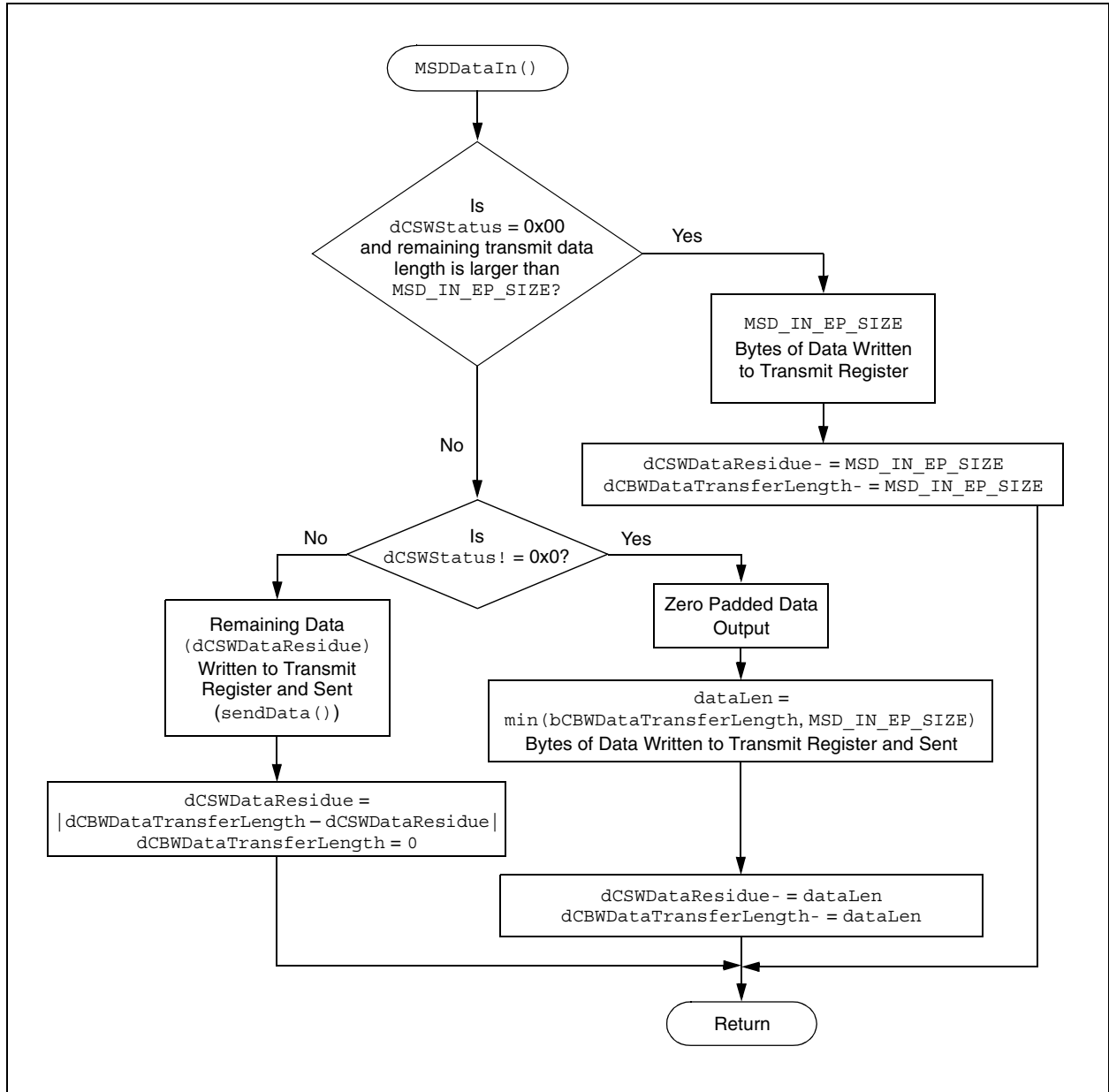


FIGURE 11: MSDDataIn() (BULK-IN TRANSPORT) FLOWCHART



TOOLS, TESTING AND CUSTOMIZATION

The firmware was developed using these Microchip development tools (see “References”):

- MPLAB® IDE, V7.11
- Microchip C18 C Compiler, V2.40

The following third party tools (see “References”) were used for USB packet level troubleshooting and analysis of sector level reading and writing on the SD card:

- SnoopyPro 0.22 (USB Sniffer)
- Directory Snoop™, V5.01 (Sector Level Media Analyzer)

The firmware has been tested with SD cards of varying sizes from different manufacturers. Table 3 summarizes the test results. The initialization, read, write, delete, format and rename operations were successfully tested for all SD cards listed. The firmware was also tested for 64 MB MMC. The 16 MB SD/MMC card is not supported as it utilizes the FAT12 file system that is not supported by the Windows drivers. The code has been tested on OHCI, UHCI and EHCI USB root hubs. Auto-triggering and accessing files from the SD card through PC applications has been tested. The enumerated disk drive can be formatted (using the Windows Explorer application) as a FAT16, FAT32 or NTFS volume. Like any removable disk drive application, the USB Mass Storage Device allows users to create, edit, save, delete, rename and read files or folders on the SD card.

TABLE 3: MSD SD CARD TEST RESULTS

Manufacturer	Card Capacity	Test Result
EP Memory	512 MB	Passed
Lexar	256 MB	Passed
Lexar	512 MB	Passed
Lexar	1.0 GB	Passed
SanDisk	128 MB	Passed
SanDisk	512 MB	Passed
SanDisk	1.0 GB	Passed
SimpleTech	256 MB	Passed
SimpleTech	1 GB	Passed
Viking	256 MB	Passed
Viking	512 MB	Passed

The firmware can be modified to interface with other portable media cards with an SPI interface. For example, compact Flash, mini-SD card, XD picture card, memory stick pro and so on. This enhanced application can be developed as a multi-card reader device. Many digital devices, such as Digital Cameras, MP3 Players and PDAs, have an SD card interface for bulk storage. This application is useful for developers of SD card media interfaces for other portable devices as well. Alternatively, this can be developed as a stand-alone data logger application by including FAT libraries. Another possible modification is to replace the SD card with Flash memory to develop a Flash thumb drive.

SUMMARY

This application note has demonstrated enumerating a PIC18F4550 as a Mass Storage Device using an SD card in Single-Bit mode on an SPI bus. In addition, it has demonstrated how to use bulk endpoints, MSD buffers, the Bulk-Only Transport protocol and SCSI commands for data transfer.

The application described is embedded FAT-free; however, code implementation is modular and allows for seamless integration of any embedded FAT file system for stand-alone applications.

REFERENCES

- Directory Snoop™, V5.01,
<http://www.briggsoft.com/dsnoop.htm>
- FAT File System Specification – available by license,
<http://www.microsoft.com/mscorp/ip/tech/fat.asp>
- “PIC18F2455/2550/4455/4550 Data Sheet”
(DS39632), <http://www.microchip.com>
- “PICDEM™ FS USB Demonstration Board User’s
Guide” (DS51526),
<http://www.microchip.com>
- MC74VHCT125A Data Sheet,
<http://www.onsemi.com>
- Microchip MPLAB® C18 C Compiler – student
edition is available by license free of charge from
the Microchip web site,
<http://www.microchip.com/C18>
- MMC Specifications – some are available by
license and others are available for purchase,
<http://www.mmca.org/compliance>
- Microchip MPLAB® IDE – available by license free
of charge from the Microchip web site,
<http://www.microchip.com/mplabide>
- SCSI Primary Commands-2 (SPC-2),
Revision i23, 18 July 2003,
<http://www.t10.org/ftp/t10/drafts/spc2/spc2i23.pdf>
- SCSI Primary Commands-3 (SPC-3),
Revision 21d, 14 February 2005,
<http://www.t10.org/ftp/t10/drafts/spc3/spc3r23.pdf>
- SCSI Block Commands-2 (SBC-2), Revision 16,
13 November 2004,
<http://www.t10.org/ftp/t10/drafts/sbc2/sbc2r16.pdf>
- SD Card Specification – available by license,
<http://www.sdcard.org>
- SnoopyPro 0.22,
<http://sourceforge.net/projects/usbsnoop/>
- “USB Complete: Everything You Need to Develop
Custom USB Peripherals” by Jan Axelson,
ISBN 0-9650819-5-8
- Universal Serial Bus Specification Revision 2.0,
<http://www.usb.org/developers/docs/>
- Universal Serial Bus Mass Storage Class
Bulk-Only Transport, Revision 1.0,
[http://www.usb.org/developers/devclass_docs/
usbmassbulk_10.pdf](http://www.usb.org/developers/devclass_docs/usbmassbulk_10.pdf)

APPENDIX A: FREQUENTLY ASKED QUESTIONS

In this appendix, answers are provided to some of the frequently asked questions about the Mass Storage Device as well as this implementation.

- Q:** When I plug in the SD card, all of the LEDs turn on. What does this mean?
- A:** This is an indication that there was an error in the SD card initialization. Try removing the external power to the board and USB cable, removing and reinserting the SD card, reconnecting the USB cable and external power (if applicable). If this does not solve the problem, verify whether the SD card you are using has been tested with MSD (see Table 3 for the list of cards tested).
- Q:** Which endpoints are used in the MSD application?
- A:** Endpoint 0 is the mandatory endpoint that must be implemented in all USB devices for control transfers. This application uses Endpoint 1 IN and OUT as Bulk transport endpoints for a bidirectional communication between the host and the device.
- Q:** What is the size of the `msd_buffer` and how does MSD handle large data transfers?
- A:** The size of the `msd_buffer` is 512 bytes. Data is written to and read from the SD card in blocks which are 512 bytes for the SD card. Large data transfers imply multiple block read/write operations. However, the multi-read and multi-write SD card commands are not implemented – instead, multiple single block read/write commands are issued to achieve large transfers.
- Q:** What is the size of the code in terms of single-word instructions?
- A:** The size of Chapter 9 code is approximately 3K and Mass Storage Device code is approximately another 1K.
- Q:** Why is LED D2 not blinking?
- A:** After initial enumeration and reading the directory from the SD card, the host repeatedly sends the `TEST UNIT READY` command to continue checking the status of the device. The D2 LED toggles when a `TEST UNIT READY` command returns a success. So if the D2 LED is not toggling, there was an error. Check whether the SD card was properly inserted in the card reader interface before connecting the USB cable.
- Q:** What is the communication protocol between the SD card and the PICDEM™ FS USB Demonstration Board?
- A:** The SD card is operated in Single-Bit mode using the SPI bus protocol.
- Q:** What is the data transfer speed?
- A:** The maximum data transfer speed observed during MSD testing was 944 kbps. The data transfer speed of the PICDEM™ FS USB bus is 12 Mbps. The system bottleneck is the SPI bus (with the SD card operated in Single-Bit SPI Bus mode). We believe that reading 64 bytes from the endpoints into the `msd_buffer` and writing 512 bytes into the SD card also slows the overall data transfer speed.
- Q:** Is it possible to integrate the FAT file system into this implementation?
- A:** Yes. The FAT file system is required if files need to be created on the SD card without using a PC (i.e., data logging applications). However, it is possible to reuse the functions associated with the SCSI command interface and the SD card communication interface in the firmware.
- Q:** Why does it show “**Microchp**” in **Device Manager > Disk drives**? Is this a typographical error?
- A:** **Microchp** is an 8-byte Vendor ID sent from the device to the host in response to the `INQUIRY` command. Since the Vendor ID field can be only 8 bytes, **Microchp** is a shortened version of Microchip.
- Q:** Why doesn't MSD work with Windows 98?
- A:** The MSD demonstration uses the native Windows driver, `usbstor.sys`. The Windows 98 operating system does not provide native support for `usbstor.sys`. Please refer to the Microsoft® web site for details: <http://www.microsoft.com/whdc/device/storage/usbfaq.mspx>
- Q:** Why does MSD implement the SCSI command set and not the RBC?
- A:** Currently, Windows 2000 and Windows XP do not provide support to handle devices that implement Reduced Block Commands (RBC, subclass 0x01) protocol. Please refer to the Microsoft web site for further details: <http://www.microsoft.com/whdc/device/storage/usbfaq.mspx>

APPENDIX B: SOURCE CODE

Software License Agreement

The software supplied herewith by Microchip Technology Incorporated (the "Company") is intended and supplied to you, the Company's customer, for use solely and exclusively with products manufactured by the Company.

The software is owned by the Company and/or its supplier, and is protected under applicable copyright laws. All rights are reserved. Any use in violation of the foregoing restrictions may subject the user to criminal sanctions under applicable laws, as well as to civil liability for the breach of the terms and conditions of this license.

THIS SOFTWARE IS PROVIDED IN AN "AS IS" CONDITION. NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. THE COMPANY SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.

The complete source code, including any demo applications and necessary support files, is available for download as a single archive file from the Microchip corporate web site, at:

www.microchip.com

AN1003

APPENDIX C: USB DESCRIPTOR FORMATS

TABLE C-1: DEVICE DESCRIPTOR

Device Descriptor		
Field	Value	Description
bDescriptorType	01h	Device descriptor type.
bDeviceClass	00h	Class specified in the interface descriptor.
bDeviceSubClass	00h	Subclass specified in the interface descriptor.
bDeviceProtocol	00h	Protocol specified in the interface descriptor.
bMaxPacketSize	10h	Endpoint 0 size.
idVendor	04D8h	Vendor ID assigned by USB-IF.
bNumConfiguration	01h	Number of possible configurations.

TABLE C-2: CONFIGURATION DESCRIPTOR

Field	Value	Description										
bDescriptorType	02h	Configuration descriptor type.										
bNumInterfaces	01h	Number of interfaces supported by this configuration.										
bConfigurationValue	01h	Index value of this configuration.										
bmAttributes	C0h	Configuration Characteristics: <table border="1"><thead><tr><th>Bit</th><th>Description</th></tr></thead><tbody><tr><td>7</td><td>Reserved (set to '1')</td></tr><tr><td>6</td><td>Self-powered</td></tr><tr><td>5</td><td>Remote wake-up</td></tr><tr><td>4-0</td><td>Reserved</td></tr></tbody></table>	Bit	Description	7	Reserved (set to '1')	6	Self-powered	5	Remote wake-up	4-0	Reserved
Bit	Description											
7	Reserved (set to '1')											
6	Self-powered											
5	Remote wake-up											
4-0	Reserved											
MaxPower	32h	Maximum power consumption of the USB device from the bus, expressed in 2 mA units (i.e., 50 = 100 mA).										

TABLE C-3: INTERFACE DESCRIPTOR

Field	Value	Description
bDescriptorType	04h	Interface descriptor type.
bInterfaceNumber	00h	Number of interface. Zero-based value identifying the index in the array of concurrent interfaces supported by this configuration.
bNumEndpoints	02h	Number of endpoints used by this interface (excluding Endpoint 0). This value shall be at least 2.
bInterfaceClass	08h	Mass Storage Class.
bInterfaceSubClass	06h	Subclass code (assigned by USB-IF). Indicates which industry standard command block definition to use.
bInterfaceProtocol	50h	Bulk-Only Transport.

APPENDIX D: USB DESCRIPTOR STRUCTURES

~\system\usb\usbdefs\usbdefs_std_dsc.h

DEVICE DESCRIPTOR

```
typedef struct _USB_DEV_DSC
{
    byte bLength;           byte bDscType;
    word bcdUSB;           byte bDevCls;
    byte bDevSubCls;      byte bDevProtocol;
    byte bMaxPktSize0;    word idVendor;
    word idProduct;       word bcdDevice;
    byte iMFR;            byte iProduct;
    byte iSerialNum;      byte bNumCfg;
} USB_DEV_DSC;
```

CONFIGURATION DESCRIPTOR

```
typedef struct _USB_CFG_DSC
{
    byte bLength;           byte bDscType;
    word wTotalLength;     byte bNumIntf;
    byte bCfgValue;        byte iCfg;
    byte bmAttributes;     byte bMaxPower;
} USB_CFG_DSC;
```

INTERFACE DESCRIPTOR

```
typedef struct _USB_INTF_DSC
{
    byte bLength;           byte bDscType;
    byte bIntfNum;         byte bAltSetting;
    byte bNumEPs;          byte bIntfCls;
    byte bIntfSubCls;      byte bIntfProtocol;
    byte iIntf;
} USB_INTF_DSC;
```

ENDPOINT DESCRIPTOR

```
typedef struct _USB_EP_DSC
{
    byte bLength;           byte bDscType;
    byte bEPAdr;           byte bmAttributes;
    word wMaxPktSize;      byte bInterval;
} USB_EP_DSC;
```

AN1003

APPENDIX E: STANDARD USB DEVICE REQUESTS

TABLE E-1: STANDARD REQUESTS FOR CONTROL TRANSFERS

bmRequestType	bRequest	Description
0000000b 0000001b 0000010b	CLEAR_FEATURE	The host requests to disable a feature on a device, interface or endpoint.
1000000b	GET_CONFIGURATION	The host requests the value of the current device configuration.
1000000b	GET_DESCRIPTOR	The host requests a specific descriptor.
1000001b	GET_INTERFACE	
1000000b 1000001b 1000010b	GET_STATUS	Index value of the configuration.
0000000b	SET_ADDRESS	The host specifies an address to use in future communications with the device.
0000000b	SET_CONFIGURATION	
0000000b	SET_DESCRIPTOR	The host adds a descriptor or updates an existing descriptor.
0000000b 0000001b 0000010b	SET_FEATURE	The host requests to enable a feature on a device, interface or endpoint.
0000001b	SET_INTERFACE	For devices with a configuration that supports multiple, mutually exclusive settings for the interface, the host requests the device to use specific settings.
1000010b	SYNC_FRAME	The device sets and reports an endpoint's synchronization frame.

APPENDIX F: BULK ENDPOINT DESCRIPTORS

TABLE F-1: BULK-IN ENDPOINT DESCRIPTOR

Bulk-In Endpoint Descriptor										
Field	Value	Description								
bDescriptorType	05h	Endpoint descriptor type.								
bEndpointAddress	81h	The address of this endpoint on the USB device. The address is encoded as follows: <table border="1" style="margin-left: 20px;"> <thead> <tr> <th>Bit</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>3-0</td> <td>The endpoint number</td> </tr> <tr> <td>6-4</td> <td>Reserved, set to '0'</td> </tr> <tr> <td>7</td> <td>1 = In</td> </tr> </tbody> </table>	Bit	Description	3-0	The endpoint number	6-4	Reserved, set to '0'	7	1 = In
Bit	Description									
3-0	The endpoint number									
6-4	Reserved, set to '0'									
7	1 = In									
bmAttributes	02h	This is a Bulk endpoint.								
bMaxPacketSize	40h	Maximum packet size. Shall be 8, 16, 32 or 64 bytes (64 bytes in our case).								
bInterval	00h	Does not apply to Bulk endpoints.								

TABLE F-2: BULK-OUT ENDPOINT DESCRIPTOR

Bulk-Out Endpoint Descriptor										
Field	Value	Description								
bDescriptorType	05h	Endpoint descriptor type.								
bEndpointAddress	01h	The address of this endpoint on the USB device. The address is encoded as follows: <table border="1" style="margin-left: 20px;"> <thead> <tr> <th>Bit</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>3-0</td> <td>The endpoint number</td> </tr> <tr> <td>6-4</td> <td>Reserved, set to '0'</td> </tr> <tr> <td>7</td> <td>0 = Out</td> </tr> </tbody> </table>	Bit	Description	3-0	The endpoint number	6-4	Reserved, set to '0'	7	0 = Out
Bit	Description									
3-0	The endpoint number									
6-4	Reserved, set to '0'									
7	0 = Out									
bmAttributes	02h	This is a Bulk endpoint.								
bMaxPacketSize	40h	Maximum packet size. Shall be 8, 16, 32 or 64 bytes (64 bytes in our case).								
bInterval	00h	Does not apply to Bulk endpoints.								

AN1003

APPENDIX G: CBW AND CSW

TABLE G-1: COMMAND BLOCK WRAPPER (CBW)

Command Block Wrapper (CBW)								
Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0-3	dCBWSignature							
4-7	dCBWTag							
8-11	dCBWDataTransferLength							
12	bmCBWFlags							
13	Reserved (0)				bCBWLUN			
14	Reserved (0)			BCBMCBLength				
15-30	CBWCB							

TABLE G-2: COMMAND STATUS WRAPPER (CSW)

Command Status Wrapper (CSW)								
Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0-3	dCBWSignature							
4-7	dCSWTag							
8-11	dCSWDataResidue							
12	bCSWStatus							

APPENDIX H: SCSI COMMAND SET

TABLE H-1: SCSI COMMAND SET

Command Name	Operation Code	Description
INQUIRY	12h	Gets device Information.
READ CAPACITY	25h	Requests for capacity and medium format parameters.
READ FORMATTED CAPACITY	23h	Reports current media capacity and formatting capacities supported by media.
READ (10)	28h	Transfers binary data from the media to the host.
WRITE (10)	2Ah	Transfers binary data from the host to the media.
MODE SENSE (6)	1Ah	Requests device to report parameters.
REQUEST SENSE (6)	03h	Transfers status sense data to the host.
PREVENT ALLOW MEDIUM REMOVAL	1Eh	Prevents or allows the removal of media from a removable media device.
TEST UNIT READY	00h	Requests to check if logical unit is ready.
VERIFY (10)	2Fh	Requests to verify a specified LBA on medium.
START/STOP	1Bh	Requests a removable media device to load or unload its media.

AN1003

APPENDIX I: SCSI COMMAND AND DATA FORMAT

TABLE I-1: MODE PARAMETER HEADER (6)

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	MODE DATA LENGTH							
1	MEDIUM TYPE							
2	DEVICE SPECIFIC PARAMETER							
3	BLOCK DESCRIPTOR LENGTH							

TABLE I-2: FIXED FORMAT SENSE DATA

Fixed Format Sense Data								
Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	RESPONSE CODE (70h or 71h)							
1	Obsolete							
2	FILEMARK	EOM	ILI	Reserved	SENSE KEY			
3-6	INFORMATION							
7	ADDITIONAL SENSE LENGTH (n-7)							
8-11	COMMAND SPECIFIC INFORMATION							
12	ADDITIONAL SENSE CODE							
13	ADDITIONAL SENSE CODE QUALIFIER							
14	FIELD REPLACEMENT UNIT CODE							
15-17	SENSE KEY SPECIFIC							
18-n	Additional Sense Bytes							

TABLE I-3: STANDARD INQUIRY DATA FORMAT

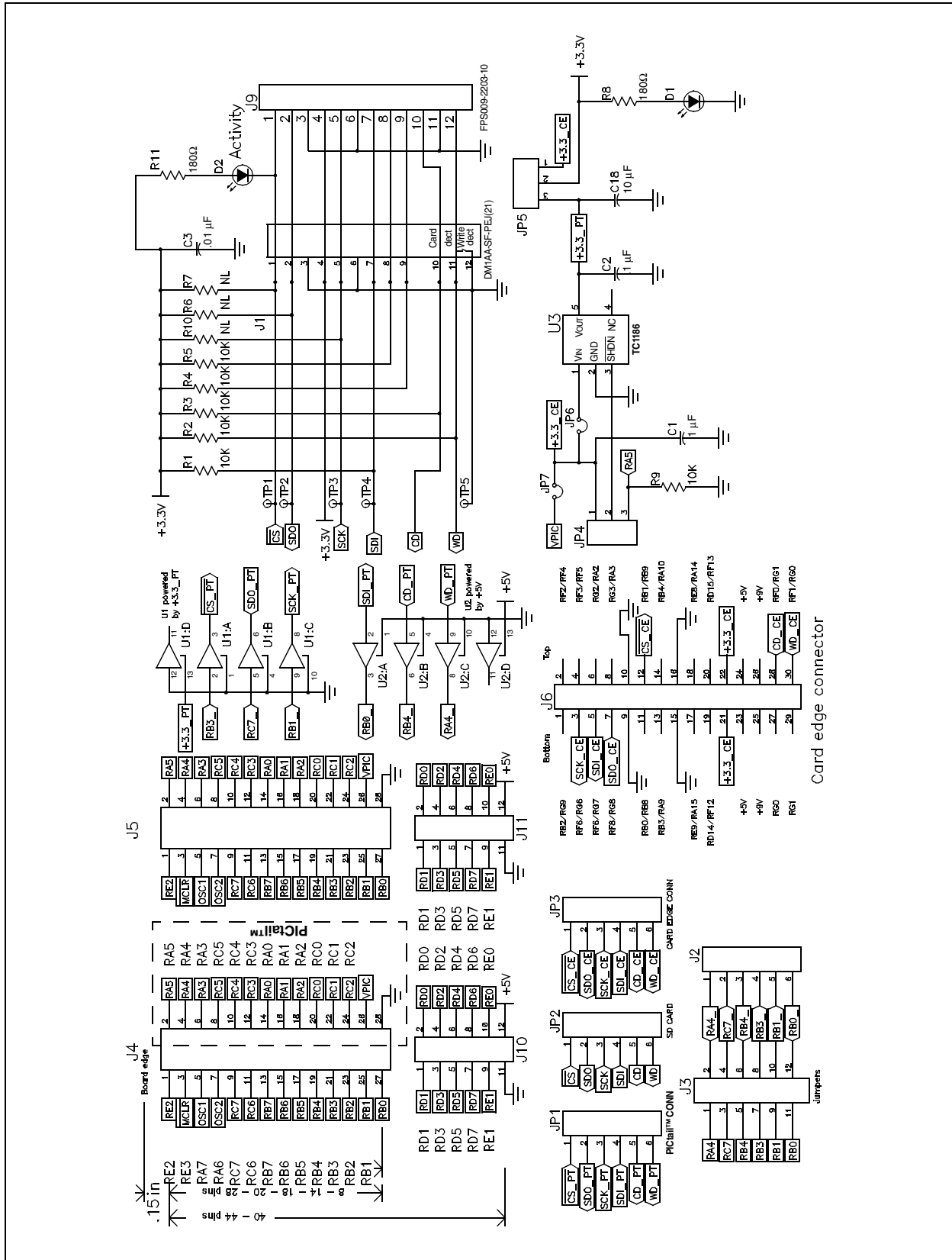
Standard Inquiry Data Format								
Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	PERIPHERAL QUALIFIER			PERIPHERAL DEVICE TYPE				
1	RMB	Reserved						
2	Version							
3	Obsolete		NORMACA	HISUP	RESPONSE DATA FORMAT			
4	ADDITIONAL LENGTH (n-4)							
5	SCCS	ACC	TPGS		3PC	Reserved		PRTOECT
6	BQUE	ENC SERV	VS	MULTIP	MCHNGR	Obsolete		ADDR16
7	Obsolete		WUSB16	SYNC	LINKED	Obsolete	CMDQUE	VS
8-15	(MSB)	T10 VENDOR IDENTIFICATION						(LSB)
16-31	(MSB)	PRODUCT IDENTIFICATION						(LSB)
32-35	(MSB)	PRODUCT REVISION LEVEL						(LSB)

TABLE I-4: READ (10) COMMAND

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Operation Code (28h)							
1	RDPROTECT		DPO		FUA	Reserved	FUA_NV	Obsolete
2	(MSB) Logical Block Address (LSB)							
5								
6	Reserved			Group Number				
7	(MSB) TRANSFER LENGTH (LSB)							
8								
9	CONTROL							

APPENDIX J: SCHEMATIC

FIGURE J-1: PICtail™ BOARD FOR SD™ AND MMC CARDS SCHEMATIC



Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION, INCLUDING BUT NOT LIMITED TO ITS CONDITION, QUALITY, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR PURPOSE. Microchip disclaims all liability arising from this information and its use. Use of Microchip's products as critical components in life support systems is not authorized except with express written approval by Microchip. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights.

Trademarks

The Microchip name and logo, the Microchip logo, Accuron, dsPIC, KEELOQ, microID, MPLAB, PIC, PICmicro, PICSTART, PRO MATE, PowerSmart, rPIC, and SmartShunt are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.


AmpLab, FilterLab, Migratable Memory, MXDEV, MXLAB, PICMASTER, SEEVAL, SmartSensor and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Analog-for-the-Digital Age, Application Maestro, dsPICDEM, dsPICDEM.net, dsPICworks, ECAN, ECONOMONITOR, FanSense, FlexROM, fuzzyLAB, In-Circuit Serial Programming, ICSP, ICEPIC, Linear Active Thermistor, MPASM, MPLIB, MPLINK, MPSIM, PICkit, PICDEM, PICDEM.net, PICLAB, PICtail, PowerCal, PowerInfo, PowerMate, PowerTool, rLAB, rPICDEM, Select Mode, Smart Serial, SmartTel, Total Endurance and WiperLock are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

All other trademarks mentioned herein are property of their respective companies.

© 2005, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

 Printed on recycled paper.

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
== ISO/TS 16949:2002 ==

Microchip received ISO/TS-16949:2002 quality system certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona and Mountain View, California in October 2003. The Company's quality system processes and procedures are for its PICmicro® 8-bit MCUs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.



WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office

2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
<http://support.microchip.com>
Web Address:
www.microchip.com

Atlanta

Alpharetta, GA
Tel: 770-640-0034
Fax: 770-640-0307

Boston

Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago

Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Dallas

Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit

Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

Kokomo

Kokomo, IN
Tel: 765-864-8360
Fax: 765-864-8387

Los Angeles

Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

San Jose

Mountain View, CA
Tel: 650-215-1444
Fax: 650-961-0286

Toronto

Mississauga, Ontario,
Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Australia - Sydney

Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing

Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

China - Chengdu

Tel: 86-28-8676-6200
Fax: 86-28-8676-6599

China - Fuzhou

Tel: 86-591-8750-3506
Fax: 86-591-8750-3521

China - Hong Kong SAR

Tel: 852-2401-1200
Fax: 852-2401-3431

China - Qingdao

Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai

Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang

Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen

Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

China - Shunde

Tel: 86-757-2839-5507
Fax: 86-757-2839-5571

China - Wuhan

Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xian

Tel: 86-29-8833-7250
Fax: 86-29-8833-7256

ASIA/PACIFIC

India - Bangalore

Tel: 91-80-2229-0061
Fax: 91-80-2229-0062

India - New Delhi

Tel: 91-11-5160-8631
Fax: 91-11-5160-8632

India - Pune

Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

Japan - Yokohama

Tel: 81-45-471-6166
Fax: 81-45-471-6122

Korea - Gumi

Tel: 82-54-473-4301
Fax: 82-54-473-4302

Korea - Seoul

Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Penang

Tel: 604-646-8870
Fax: 604-646-5086

Philippines - Manila

Tel: 632-634-9065
Fax: 632-634-9069

Singapore

Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu

Tel: 886-3-572-9526
Fax: 886-3-572-6459

Taiwan - Kaohsiung

Tel: 886-7-536-4818
Fax: 886-7-536-4803

Taiwan - Taipei

Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

Thailand - Bangkok

Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Weis

Tel: 43-7242-2244-399
Fax: 43-7242-2244-393

Denmark - Copenhagen

Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris

Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Munich

Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan

Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands - Drunen

Tel: 31-416-690399
Fax: 31-416-690340

Spain - Madrid

Tel: 34-91-352-30-52
Fax: 34-91-352-11-47

UK - Wokingham

Tel: 44-118-921-5869
Fax: 44-118-921-5820