INTRODUCTION

With the introduction of Microchip's microcontrollers with the USB OTG peripheral, microcontroller applications can easily support USB Embedded Host functionality. One of the most common uses of this capability is to interface to mass storage devices, such as USB Flash Drives and memory card readers. These devices utilize the USB Mass Storage Class.

USB Mass Storage Class

Overview

Of the four transfer types supported by USB, the one most suitable for large data transfers is Bulk. Bulk transfers use the USB bandwidth efficiently, in that they utilize all of the remaining bandwidth in a frame after Control, Interrupt and Isochronous transfers are complete. They are not constrained to only a certain number of bytes per frame. They also incorporate error checking, so the data is ensured to be accurate. The exact amount of time available for a bulk transfer will depend on the amount of other traffic that is on the bus. If several other transfers must also be performed, there may be very little bandwidth available for Bulk transfers in a frame. Therefore, Bulk transfers should be used only for non-time critical operations.

The class, subclass and protocol designators for a Mass Storage Device are not contained in the bDeviceClass, bDeviceSubClass and bDeviceProtocol fields of the Device Descriptor. Instead, these fields are all set to 0x00, and the designators are specified in the bInterfaceClass, bInterfaceSubClass and bInterfaceProtocol fields of the Interface Descriptor. The most common configuration for USB Mass Storage devices is:

- bInterfaceClass - 0x08 (Mass Storage Class)
- bInterfaceSubClass - 0x06 (SCSI Primary Command-2 (SPC-2))
- bInterfaceProtocol - 0x50 (Bulk Only Transport)

A Mass Storage device may contain multiple logical units, each represented by a Logical Unit Number (LUN). All logical units on the device share the same device characteristics, but can be addressed independently via their LUN. LUNs are numbered from 0 to 15. If a device does not support multiple LUNs, then 0 is specified for the LUN.

This implementation of the Mass Storage Class provides support for the Bulk Only Transport. In this protocol, three endpoints are utilized:

- Endpoint 0 for Control transfers
- One Bulk IN endpoint
- One Bulk OUT endpoint

Bulk transfers consist of three stages:

- Command
- Data (optional)
- Status

The Command stage is sent from the Host to the Peripheral via the Bulk OUT endpoint. The Data stage, if present, utilizes the Bulk IN endpoint if the data is being transferred from the Peripheral to the Host, or the Bulk OUT endpoint if the data is being transferred from the Host to the Peripheral. The Status stage utilizes the Bulk IN endpoint for the Host to receive status information from the Peripheral about the transfer. The flow of the stages is shown in Figure 1.
FIGURE 1: COMMAND/DATA/STATUS FLOW

COMMAND BLOCK WRAPPER (CBW)

The Command Block Wrapper is sent to the Peripheral during the Command phase of the transfer. The CBW is a 31-byte packet that includes the following information:

- Tag to identify the transfer
- Number of bytes to transfer during the Data phase
- LUN to which the transfer applies
- Command block to be executed by the device

The format of the CBW is shown in Table 1.

TABLE 1: COMMAND BLOCK WRAPPER

<table>
<thead>
<tr>
<th>Byte</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td>dCBWSignature</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4-7</td>
<td>dCBWTag</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-11</td>
<td>dCBWDataTransferLength</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>bmCBWFlags</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Reserved (0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bCBWLUN</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Reserved (0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>BCBMCMCLength</td>
<td></td>
</tr>
<tr>
<td>15-30</td>
<td>CBWCB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The CBW is generated internally by the Mass Storage client driver.
COMMAND STATUS WRAPPER (CSW)

The Command Status Wrapper is sent to the Host from the Peripheral. The CSW is a 13-byte packet that includes the following information:

- Tag to identify the transfer (must match the tag in the CBW)
- The difference between the number of data bytes expected and the number actually transferred
- Success or Failure of the command

The format of the CSW is shown in Table 2.

<table>
<thead>
<tr>
<th>Byte</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dCBWSignature</td>
</tr>
<tr>
<td>4-7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dCSWTag</td>
</tr>
<tr>
<td>8-11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dCSWDataResidue</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bCSWStatus</td>
</tr>
</tbody>
</table>

The CSW is received and checked internally by the Mass Storage client driver.
USING THE MASS STORAGE CLIENT DRIVER

Installing the Mass Storage Client Driver

The Mass Storage Client Driver is installed as a part of complete USB Embedded Host support package, available from the Microchip web site (http://www.microchip.com/usb). Refer to “AN1140 USB Embedded Host Stack” for more information on installation.

Application Architecture

Most applications will not interface directly with the USB Host Mass Storage Client Driver. Instead, they will use a media interface layer, which will interface with the Client driver, which in turn will use the host stack driver. For example, the application described by “AN1145 Using a USB Flash Drive on an Embedded Host”, has five layers including the application layer, as shown in Figure 2.

FIGURE 2: APPLICATION ARCHITECTURE

<table>
<thead>
<tr>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>File System Support</td>
</tr>
<tr>
<td>SCSI Command Support</td>
</tr>
<tr>
<td>Mass Storage Client Driver</td>
</tr>
<tr>
<td>USB Embedded Host Driver</td>
</tr>
</tbody>
</table>

The “SCSI Command Support” layer is the media interface layer that converts file system commands to SCSI commands and sends them to the USB Peripheral using the USB Mass Storage Class.

Note: For detailed information about the USB Host Mass Storage Class Driver API, please refer to “AN1141 USB Embedded Host Stack Programmer's Guide” and the API documentation provided in the Help directory.
CONFIGURING THE CLASS

Using the USB Configuration Tool

Use the USB configuration tool, USBConfig.exe, to configure the Mass Storage client driver for an application. This tool is installed in the .\Microchip\USB directory of the installation.

In order to use the Mass Storage client driver for a USB Embedded Host, select the USB device type of the application on the **Main** tab.

**FIGURE 3: USB CONFIGURATION - MAIN**

![USB Configuration Tool](image-url)
Select the **Host** tab to configure basic Host operation as shown in Figure 4. The Mass Storage Client Driver requires support for Control and Bulk endpoints. If the application contains no classes that require Interrupt or Isochronous endpoints, then support for those endpoint types can be disabled.

Mass Storage devices can respond rather slowly in comparison to the USB’s 1 ms communication frame. Therefore, it is recommended to allow a large number of NAKs before terminating the communication attempt. Also, some devices require longer than the USB specification of 100 ms to initialize after power-up.

Therefore, it is recommended to increase the attach debounce interval. Then enter the name of the function in the main source file that serves as the application-level event handler.

The USB Mass Storage Client Driver can either poll the USB Host driver for transfer status or respond to USB Host driver transfer events. Refer to the section “**Event Generation**” below for more information about this selection.

**FIGURE 4: USB CONFIGURATION - HOST**

![USB Configuration - Host](image-url)
Select the **Mass Storage** tab, check the **Mass Storage Client is used in Host Mode** checkbox to enable support for a Mass Storage Embedded host as shown in Figure 5.

Many Mass Storage devices use a SCSI interface protocol. Support for this protocol is provided with the Mass Storage client driver. Since each function in this layer must complete before the next operation can begin, Mass Storage transfer events are not used.

**FIGURE 5: USB CONFIGURATION - MASS STORAGE**

![Microchip USB OTG Configuration Tool](image)

**DEFINING THE INTERFACE FUNCTIONS**

The client driver requires two interface functions in the media interface layer. The first is the initialization handler, which is called after the Peripheral has been enumerated and initialized by the Mass Storage client driver. The initialization handler should be of the type defined by the `typedef`:

```c
typedef BOOL (*USB_CLIENT_INIT)(BYTE address, DWORD flags);
```

This function performs initialization specific to the media interface. If initialization occurs with no error, this routine should return `TRUE`. If errors are encountered, this routine should return `FALSE`, and no transfers to the Peripheral will be allowed.

The second interface function is required to handle events that occur during normal operation. This event handler should be of the type defined by the `typedef`:

```c
typedef BOOL (*USB_CLIENT_EVENT_HANDLER)(BYTE address, USB_EVENT event, void *data, DWORD size);
```

Click on Generate to create the configuration files, `usb_config.c` and `usb_config.h`, and store them in the project directory.
For example, one of the events that can occur is EVENT_DETACH. This occurs when a device has detached from the bus. In this case, the media interface layer will need to update its status, by doing operations such as removing the device from its list of attached media.

See the API documentation provided in the Help directory for a complete list of events.

The client driver requires a list of the media interface's required Peripheral initialization and event handlers. This list is defined by the configuration tool USBConfig.exe, provided with the stack.

**EVENT GENERATION**

The client driver can be configured to utilize transfer events (EVENT_TRANSFER) from the USB Host layer. In addition, the client driver can be configured to generate transfer events (EVENT_MSD_TRANSFER) for the media interface layer. These two events can be configured independently of each other, giving four possible combinations as shown in Table 3 (below).

<table>
<thead>
<tr>
<th>EVENT CONFIGURATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>USB Host Driver</td>
</tr>
<tr>
<td>Poll for transfer status</td>
</tr>
<tr>
<td>Poll for transfer status</td>
</tr>
<tr>
<td>Generate transfer events</td>
</tr>
<tr>
<td>Generate transfer events</td>
</tr>
</tbody>
</table>

If USB Embedded Host transfer events are used, the application will require more program and data memory, but application processing will be performed more efficiently. The USB Embedded Host transfer event configuration is transparent to the media interface layer.

If USB Embedded Host MSD events are used, more program memory is required, and the media interface layer that handles these events must be structured properly. In general, the code architecture required to utilize transfer events is more sophisticated, and more difficult for beginning C programmers to design, develop, debug and maintain.

The choice of whether or not to utilize USB Embedded Host MSD transfer events can also depend on the implementation of the other layers in the application. For example, “AN1045 Implementing File I/O Functions using Microchip’s Memory Disk Drive File System Library” provides functions to open, close, read from and write to files in a format that PCs can use. Since a user should not be able to write to a file until that file is successfully opened, the implementation of the Memory Disk Drive File System blocks execution of other tasks until the requested operation is complete. Since the File System layer blocks execution, there is no benefit to structuring the media interface layer to utilize USB Embedded Host MSD transfer events. Therefore, the simpler polling mechanism is used.

**CLIENT DRIVER INITIALIZATION**

The Host Mass Storage Client Driver is initialized by a single function:

```c
BYTE USBCClientMSDInit(void);
```

This function initializes all internal variables for operation. It should only be called once during the application's execution.

The USB Configuration tool will provide a macro USBCClientInitialize() to call all of the initialization routines required by the USB Embedded Host driver, the supported client drivers and media interfaces.

**NORMAL CLIENT DRIVER OPERATION**

Normal background operation is performed by a single function:

```c
void USBCClientMSDDriverTasks(void);
```

This routine must be called on a regular basis to allow device operation. The polling rate is not critical, since most of the actual transfer of information is handled through the USB interrupt. Since an application may support multiple classes, this function does not call the USBCClientDriverTasks() function, which also must be called on a regular basis.

The USB Configuration tool will provide a macro USBCClientTasks() to call all of the background task routines required by the USB Host driver and the supported client drivers.
SUPPORTED LOGICAL UNIT NUMBERS

If the media interface initialization is successful, the Mass Storage Class driver will immediately inform the media interface layer of the maximum Logical Unit Number of the device via the EVENT_MSD_MAX_LUN event. All future transfer requests will be checked against this value to ensure that a valid LUN is being referenced.

Note: The media interface and file system layers may not be able to support multiple LUNs.

PERFORMING A TRANSFER

Communication with a Peripheral is initiated by two functions:

\[
\begin{align*}
\text{BYTE} & \quad \text{USBHostMSDRead}(
\begin{align*}
\text{BYTE} & \quad \text{deviceAddress,} \\
\text{BYTE} & \quad \text{deviceLUN,} \\
\text{BYTE} & \quad \text{*commandBlock,} \\
\text{BYTE} & \quad \text{commandBlockLength,} \\
\text{BYTE} & \quad \text{*data,} \\
\text{DWORD} & \quad \text{dataLength}
\end{align*}
\)
\end{align*}
\]

\[
\begin{align*}
\text{BYTE} & \quad \text{USBHostMSDWrite}(
\begin{align*}
\text{BYTE} & \quad \text{deviceAddress,} \\
\text{BYTE} & \quad \text{deviceLUN,} \\
\text{BYTE} & \quad \text{*commandBlock,} \\
\text{BYTE} & \quad \text{commandBlockLength,} \\
\text{BYTE} & \quad \text{*data,} \\
\text{DWORD} & \quad \text{dataLength}
\end{align*}
\)
\end{align*}
\]

The commandBlock is a block of up to 16 bytes that tells the Peripheral which operation to perform. When the SCSI media interface layer is used, this block contains the SCSI command to perform the requested operation.

A return code of \texttt{USB_SUCCESS} (0x00) indicates that the operation was started successfully.

After initiating communication, take care that \texttt{USBHostTasks()} and \texttt{USBHostMSDTasks()} are performed while waiting for the operation to complete. The status of the operation can be determined by calling the function:

\[
\begin{align*}
\text{BOOL} & \quad \text{USBHostMSDTransferIsComplete}(
\begin{align*}
\text{BYTE} & \quad \text{deviceAddress,} \\
\text{BYTE} & \quad \text{*errorCode,} \\
\text{DWORD} & \quad \text{*byteCount}
\end{align*}
\)
\end{align*}
\]

If the function returns \texttt{FALSE}, the transfer is not complete, and the returned error code and byte count are not valid. If the function returns \texttt{TRUE}, the returned error code indicates the status of the operation, and the returned byte count indicates that how many bytes were transferred.

A transfer of data from the Host to the Mass Storage Peripheral appears as Example 1.

EXAMPLE 1: MASS STORAGE DATA TRANSFER, PERIPHERAL TO HOST

\[
\begin{align*}
\text{error} & = \text{USBHostMSDRead}( \text{device, 0, command, 10, buffer, size }); \\
\text{if} \ {(!error)} \\
\{ \\
\quad \text{while} \ {(!\text{USBHostMSDTransferIsComplete}( \text{device, &error, &count )))} \\
\quad \{ \\
\quad \quad \text{USBHostTasks();} \\
\quad \quad \text{USBHostMSDTasks();} \\
\quad \} \\
\}
\end{align*}
\]
CONCLUSION

The USB Embedded Host Mass Storage class provides a simple interface to popular USB mass storage devices. Embedded applications now can easily take advantage of this flexible, widely available storage media.

RESOURCES

AN1045 “Implementing File I/O Functions using Microchip’s Memory Disk Drive File System Library”
- http://www.microchip.com
AN1140 “USB Embedded Host Stack”
- http://www.microchip.com
AN1141 “USB Embedded Host Stack Programmer’s Guide”
- http://www.microchip.com
Universal Serial Bus web site:
- http://www.usb.org
Microchip Technology Inc. web site:
- http://www.microchip.com
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 devices in life support and/or safety applications is entirely at the buyer’s risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. 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, KEELOQ logo, MPLAB, PIC, PICmicro, PICSTART, PRO MATE, rFIC and SmartShunt are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.
FilterLab, Linear Active Thermistor, MXDEV, MXLAB, 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, CodeGuard, dsPICDEM, dsPICDEM.net, dsPICworks, dsSPEAK, ECAN, ECONOMONITOR, FanSense, In-Circuit Serial Programming, ICSP, ICEPIC, Mindi, MiWi, MPASM, MPLAB Certified logo, MPLIB, MPLINK, mTouch, PICkit, PICDEM, PICDEM.net, PICtail, PIC<sup>®</sup> logo, PowerCal, PowerInfo, PowerMate, PowerTool, REAL ICE, rFLAB, Select Mode, Total Endurance, UNI/O, WiperLock and ZENA 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.
© 2008, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.
 Printed on recycled paper.

Microchip received ISO/TS-16949:2002 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company’s quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ<sup>®</sup> 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.