AN2554

Creating a Multi-LUN USB Mass Storage Class Device Using the MPLAB® Harmony USB Device Stack

INTRODUCTION

The Universal Serial Bus (USB) protocol is widely used to interface storage devices to a USB Host computer. Such devices use a set of standards called the USB Mass Storage Class (MSC). Any Device that allows access to its internal storage using the Mass Storage Class protocol can be connected as a Mass Storage Device (MSD) to the Host computer over the USB interface.

This application note describes how to create an application that supports multiple logical units (multi-LUN), each appearing on the USB Host computer as a separate drive, using the MPLAB® Harmony USB Device Stack framework. A typical use case of a multiple LUN application can be found in multi-slot USB card reader applications.

This application note initially provides information about MSD-specific USB descriptors, requests, and transport protocol. This is followed by an overview of the MPLAB Harmony USB Device Architecture. The application note then details the step-wise creation of the multi-LUN application using the MPLAB Harmony Configurator (MHC) and describes the important data structures, functions, and the state-machines.

CONTROL TRANSFERS

The USB 2.0 Protocol features four data transfer types: Control, Bulk, Interrupt, and Isochronous. Each of these data transfer type possess characteristics that make these transfer types applicable to different types of applications. The MSD typically uses Control and Bulk Transfers. Refer to the section **"Bulk Transfers"** for bulk transfer type details.

In the USB Protocol, the USB Host typically uses Control Transfers to send standard and class specific requests to the device. Control transfers are typically sent on the default endpoint, Endpoint 0. This device endpoint is bidirectional and is always enabled. For Full-Speed devices, the size (the maximum number of bytes that can be transferred in a single transaction) of a control endpoint can be 8, 16, 32, or 64 bytes. For High-Speed devices, the size is fixed to 64 bytes. Unlike other USB transfers, a Control transfer consists of multiple stages: the Setup Stage, an optional Data Stage, and a Status Stage.

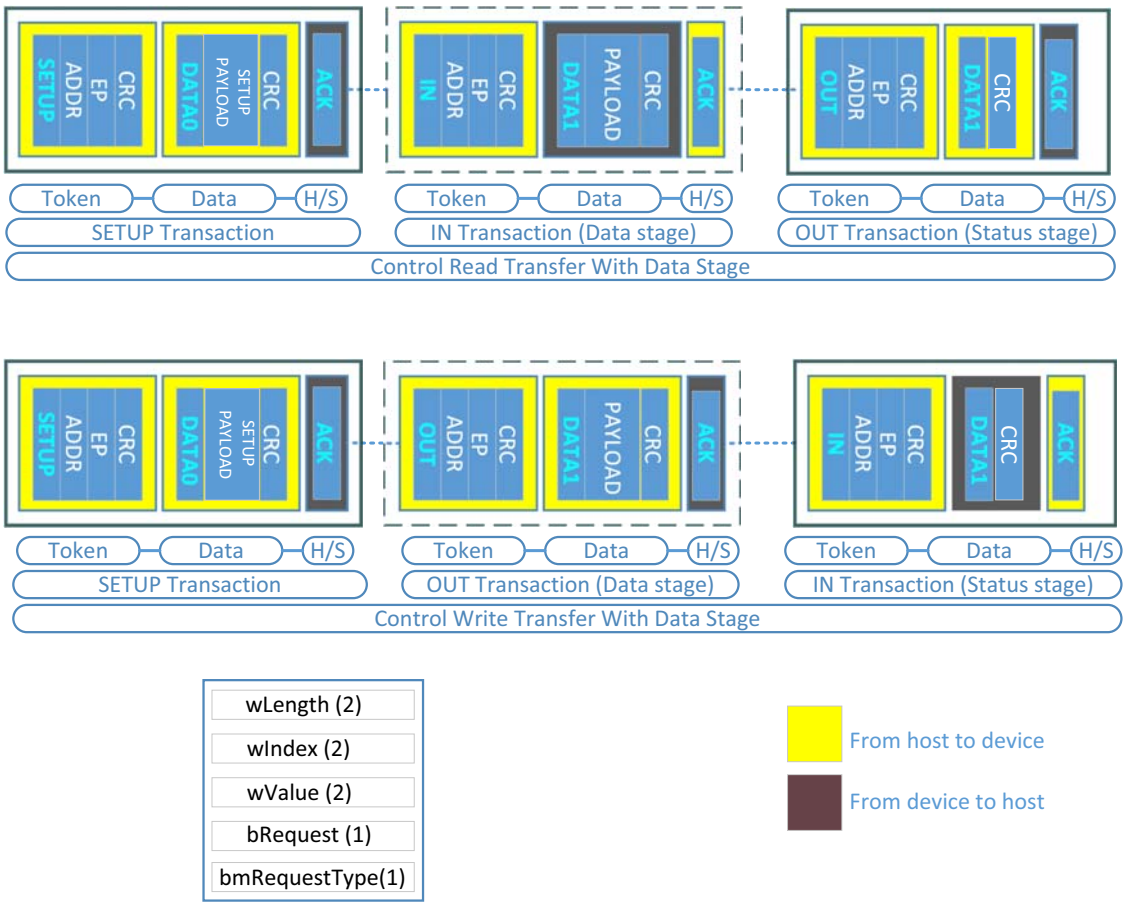
The Setup Stage consists of a Token packet followed by a Data packet and a Handshake packet. The Token packet contains a SETUP token. The Data packet contains a DATA0 token and an 8-byte command, the format of which is defined by the USB specification. This command contains details about the request type (standard, class-specific, or vendor), the recipient of the request (Device, interface, or endpoint), request code, interface or endpoint index, the direction of the Data Stage of the Control transfer, and the number of bytes to be transferred (this value could be 0), if there is a data stage. The data related to a Setup command is transferred in the Data Stage of the Control Transfer. In a Read Control transfer, data moves from Device to Host. In a Write Control Transfer, data moves from Host to Device.

The optional Data Stage of a Control Transfer contains three packets, a Token packet followed by a Data packet and a Handshake packet. The Token packet for a read transfer consists of an IN token. The Token packet for a write transfer consists of an OUT token. The Data packet starts with a DATA1 token and then alternates between DATA0 and DATA1 tokens for each transaction in the Data Stage. The Device can either send an ACK token to the Host indicating that it has successfully processed the transaction or send a NAK token indicating that the endpoint is busy, or it can send a STALL token indicating that an error has occurred.

The Status Stage of the Control Transfer is always in the opposite direction of a data transfer. It contains three packets, a Token packet followed by a Data packet and a Handshake packet. For a control read transfer, the Host provides the status by sending an OUT token packet, a Zero Length Data Packet (ZLP) to which the Device replies with an ACK token in the Handshake packet. For a control write transfer, Host requests status from the Device with an IN token in the Token packet. The Device responds by sending a DATA1 token ZLP in the Data packet, to which the Host replies with an ACK token in the Handshake packet.

Figure 1 shows the different stages of a Control read and write transfer. It also shows the fields of each packet involved in the transfer. For information on these fields, refer to the USB 2.0 Specification, which is available from www.usb.org.

FIGURE 1: CONTROL READ AND WRITER TRANSFER STAGES



Setup payload – fixed 8 byte format

BULK TRANSFERS

Bulk transfers are generally used for moving large amounts of data. This transfer type guarantees data integrity but does not guarantee latency. The USB protocol *does not* allocate bandwidth for bulk transfers. The USB Host schedules bulk transfer as and when bandwidth becomes available, which is typically after all other transfers (Control, Interrupt, and Isochronous) in a frame are complete. This makes the time of delivery of bulk transfers unpredictable.

Bulk transactions take place on endpoints that are configured for build transfers. Bulk endpoints are unidirectional. A USB device can receive data from the Host on a Bulk-OUT endpoint and transmit data to the Host on a Bulk-IN endpoint. For Full-Speed devices, the size of a Bulk endpoint can be 8, 16, 32 or 64 bytes. For High-Speed devices, the size is fixed to 512 bytes.

Figure 2 shows the packets involved in a Bulk IN (Read) and Bulk OUT (Write) transaction.

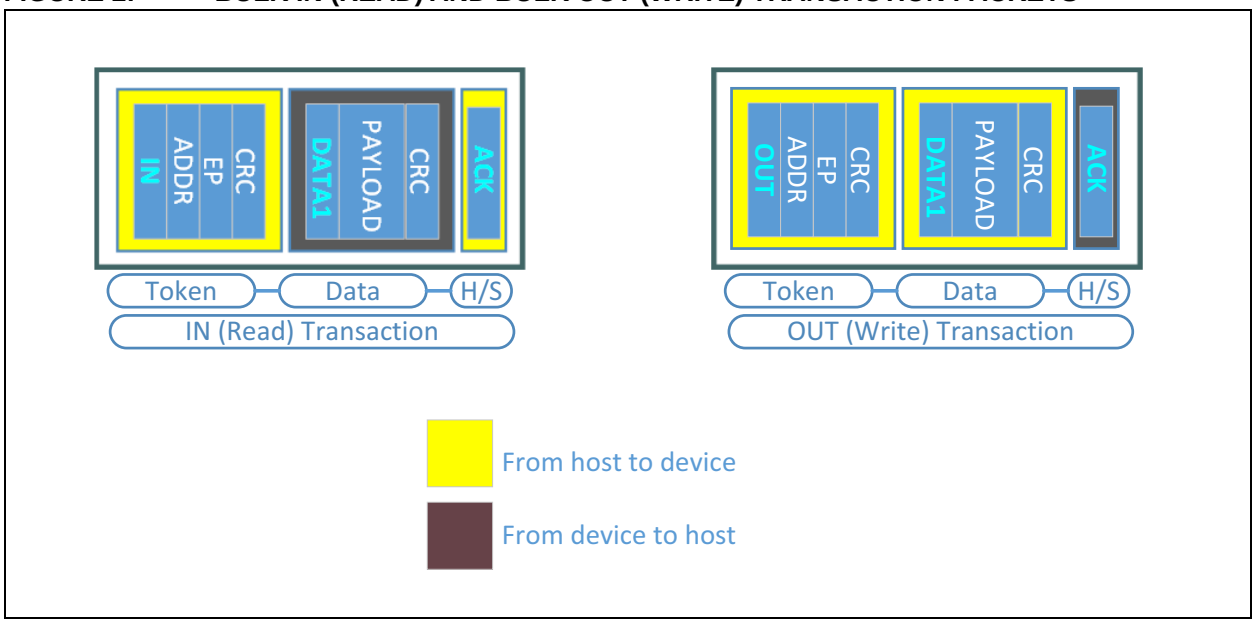
A Bulk Read transfer (Host reads from Device) can contain multiple IN transactions with each transaction containing the following interactions:

- The Host sends an IN request to device (Token packet)
- The device responds with bulk data to Host (Data packet) if it is ready to send data to the Host. The device sends NAK if it does not have data to send or is not yet ready. The device sends STALL if the endpoint has an error.
- The Host sends ACK to the device (Handshake packet).

A Bulk Write transfer (Host writes to Device) can contain multiple OUT transactions with each transaction containing the following interactions:

- Host sends OUT request to device (Token packet)
- The Host then sends bulk data to the device. (Data Packet)
- The device sends ACK to Host indicating success (Handshake packet). The device sends NAK if the endpoint buffer is not empty. A STALL is sent if the endpoint has an error.

FIGURE 2: BULK IN (READ) AND BULK OUT (WRITE) TRANSACTION PACKETS



USB MASS STORAGE DEVICE CLASS DESCRIPTORS

A USB device uses Standard USB Descriptors to advertise its functional characteristics to a USB Host. This section discusses the USB descriptors that are relevant to a USB MSD.

Figure 3 shows the logical blocks of a typical USB Mass Storage Class Device.

At a high level, a USB MSD device firmware must perform the following tasks:

1. Detect and respond to control requests on Endpoint 0.
2. Detect and respond to Mass Storage Class specific requests on control endpoint 0
3. Decode and respond to (Block device/SCSI) commands on bulk endpoints, and it must access media and respond to read/write (SCSI) requests on bulk endpoints.

Figure 4 shows the descriptor tree for a USB Mass storage device. A USB Mass storage class device requires a Device Descriptor, Configuration Descriptor, Interface Descriptor, and two Endpoint (for bulk-only transport protocol) Descriptors. The following tables provide details of each of these descriptors. Refer to Chapter 9 of the "USB 2.0 Specification" for additional information on standard USB descriptors and USB Device Descriptor Topology.

FIGURE 3: USB MASS STORAGE CLASS DEVICE BLOCK DIAGRAM

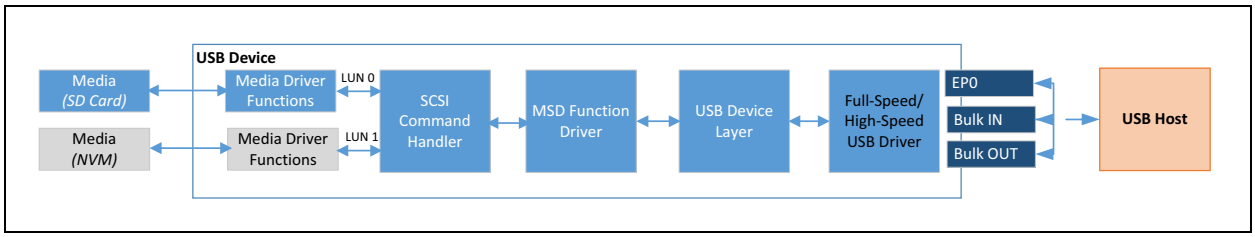


FIGURE 4: USB MASS STORAGE CLASS DEVICE DESCRIPTOR TREE

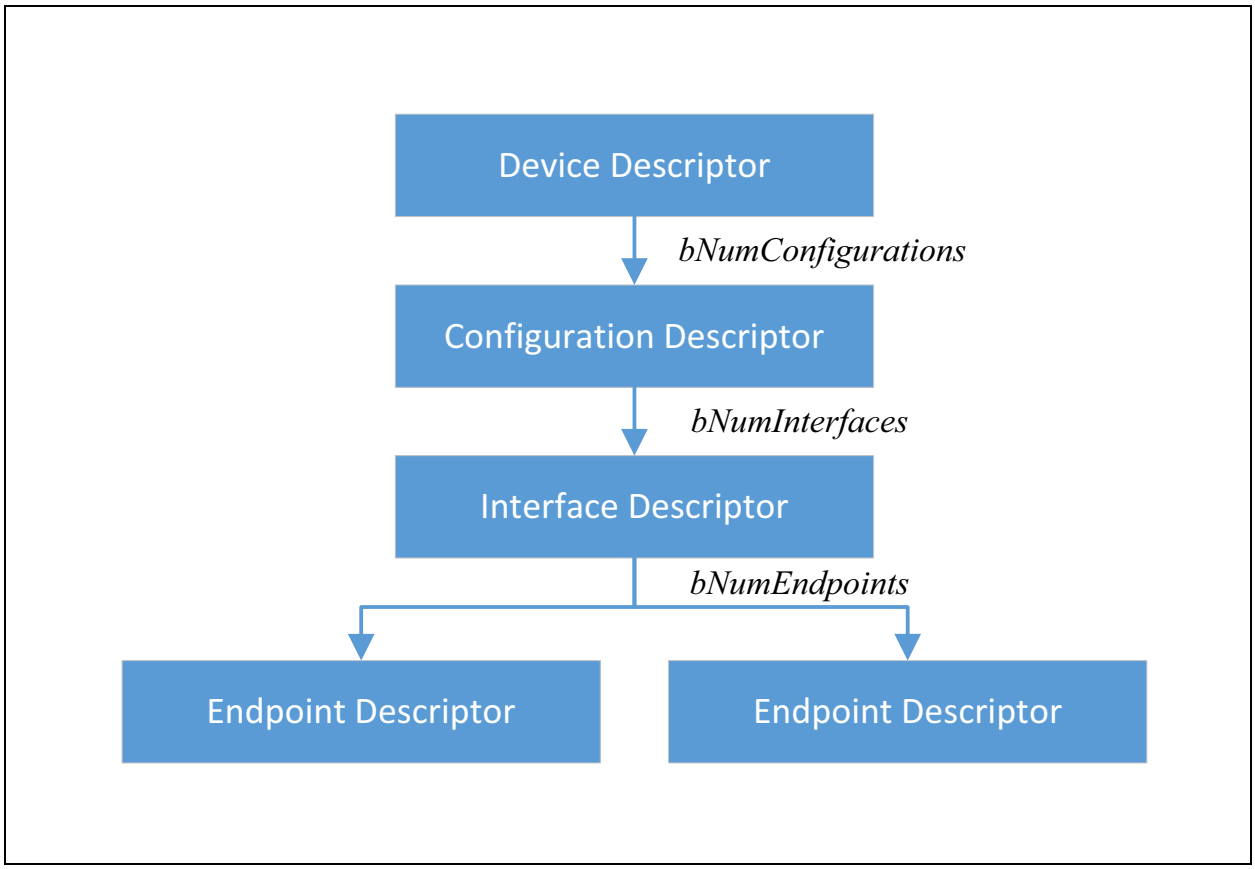


Table 1 provides details of the Device Descriptor for the MSD implemented in this application. Note that the *bDeviceClass*, *bDeviceSubClass* and *bDeviceProtocol* fields are set to 0x00, as these will be defined by the Interface Descriptor.

TABLE 1: DEVICE DESCRIPTOR

Offset	Field	Size (Bytes)	Value (Hex)	Description
0	*bLength*	1	0x12	Size of Device Descriptor in bytes.
1	*bDescriptorType*	1	0x01	DEVICE descriptor.
2	*bcdUSB*	2	0x0200	Supports USB 2.0 Specification.
4	*bDeviceClass*	1	0x00	Class is specified in the Interface Descriptor
5	*bDeviceSubClass*	1	0x00	Sub-Class is specified in the Interface Descriptor
6	*bDeviceProtocol*	1	0x00	Protocol is specified in the Interface Descriptor
7	*bMaxPacketSize0*	1	0x40	Maximum packet size for Endpoint 0 is 64 bytes.
8	*idVendor*	2	0x04D8	Vendor ID (for example, Microchip Vendor ID = 04D8h).
10	*idPacket*	2	0x009	Product ID.
12	*bcdDevice*	2	0x0100	Device version number (for example 01.00).
14	*iManufacturer*	1	0x01	Index of Manufacturer string in string descriptors.
15	*iProduct*	1	0x02	Index of Product string in string descriptors.
16	*iSerialNumber*	1	0x03	Index of Device serial number in string descriptors.
17	*bNumConfigurations*	1	0x01	Number of configurations is set to ‘1’.

Table 2 provides details of the Configuration Descriptor for the MSD implemented in this application. The *bNumInterfaces* field of this descriptor is set to 0x01. This indicates that the configuration has only one interface.

When the device receives a Get Configuration Descriptor request, it will return the configuration descriptor, Interface Descriptor and endpoint descriptors to the USB Host. During enumeration, the USB Host will issue Set Configuration request to the device, thereby asking the device to set this configuration as active.

TABLE 2: CONFIGURATION DESCRIPTOR

Offset	Field	Size (Bytes)	Value (Hex)	Description
0	*bLength*	1	0x09	Size of Configuration Descriptor in bytes.
1	*bDescriptorType*	1	0x02	CONFIGURATION descriptor.
2	*wTotalLength*	2	0x0020	Total length of data returned for this configuration is 32 bytes. Includes Configuration Descriptor (9) + Interface Descriptor (9) + Endpoint Descriptors (7+7).
4	*bNumInterfaces*	1	0x01	Number of interfaces in this configuration (supports Bulk-only data interface).
5	*bConfigurationValue*	1	0x01	Value to use as an argument to the `SetConfiguration` function request to select this configuration.
6	*iConfiguration*	1	0x00	Index of string descriptor describing this configuration.
7	*bmAttributes*	1	0xC0	Self-powered.
8	*MaxPower*	1	0x32	This device will consume maximum of 100 mA (2x *MaxPower*) from the bus when it is fully functional and this configuration is selected.

Table 3 provides details of the Interface Descriptor for the MSD implemented in this application. The value of *bNumEndpoints* is 0x02. This indicates that the device contains two endpoints.

The *bInterfaceClass* field is set to 0x08, which corresponds to Mass Storage class. This will direct the Host to associate a Mass Storage client driver with this device.

The *bInterfaceSubClass* field identifies the industry standard command sets that can be transported by the USB Mass Storage class.

TABLE 3: INTERFACE DESCRIPTOR

Offset	Field	Size (Bytes)	Value (Hex)	Description
0	*bLength*	1	0x09	Size of Configuration Descriptor in bytes.
1	*bDescriptorType*	1	0x04	INTERFACE descriptor.
2	*bInterfaceNumber*	1	0x00	Interface number. Zero-based value identifying the index in the array of concurrent interfaces supported by this configuration.
3	*bAlternateSetting*	1	0x00	No alternate settings for this interface.
4	*bNumEndpoints*	1	0x02	Number of endpoints used by this interface (Bulk IN and Bulk OUT).
5	*bInterfaceClass*	1	0x08	USB Mass storage class code.
6	*bInterfaceSubClass*	1	0x06	Data transfer protocol used is SCSI transparent commands set.
7	*bInterfaceProtocol*	1	0x50	Supports Bulk-only transport protocol.
8	*iInterface*	1	0x00	No string descriptors for this interface.

Table 4 shows some of the possible values for the *bInterfaceSubClass* field.

TABLE 4: *bInterfaceSubClass* POSSIBLE VALUES

Subclass Code	Command Block Specification	Used by:
01	Reduced Block Commands (RBC)	Flash devices
02	MMC-5 (ATAPI)	CD and DVD devices
03	QIC-157 (Obsolete)	Tape devices
04	UFI	Floppy disk drives
05	SFF-8070i (Obsolete)	Floppy disk drives
06	SCSI transparent command set	Any device

The Small Computer Systems Interface (SCSI) transparent command set is recommended for most devices. The SCSI transparent command set comprises all SCSI-related specifications, including the SCSI Primary Commands (SPC) and the SCSI Block Commands (SBC). The SCSI standards are defined outside of the USB specifications. During the enumeration process, the USB Host will identify the command protocol supported by the Device. The USB Host will issue SCSI commands in USB transfers, if the Device supports SCSI transparent command set. The SCSI commands allow the USB Host to read and write blocks of data in the storage media, request status information, and control the Device operation. The MPLAB Harmony USB Device Stack Mass Storage Class Function Driver only supports SCSI transparent command set.

The *bInterfaceProtocol* field in the Interface Descriptor defines the transport protocol used by the mass storage interface. It specifies the USB transfer types to be used on the bus for transporting mass storage commands, data and status information between the Host and the device. The Mass Storage Class defines two transport protocols:

- Control/Bulk/Interrupt (CBI)
- Bulk-only transport (BOT)

The USB MSC specification approves CBI transport protocol for use only with full-speed floppy disk drives. Moreover, CBI is completely replaced with bulk-only protocol and USB MSC specification discourages usage of CBI for new designs. The MPLAB Harmony USB Device Stack Mass Storage Function Drivers supports BOT protocol only. This protocol uses Bulk transfers for transferring command, data and status of Mass Storage Class operations. This application note addresses BOT protocol only.

During enumeration the *bInterfaceProtocol* field will indicate to Host that the device supports BOT protocol and hence, the Host will use bulk endpoints for transporting command, data and status stages of the BOT.

Bulk IN Endpoint Descriptor

The bulk IN endpoint is used to transfer data and status from device to Host.

Table 5 provides details of the Bulk IN Endpoint Descriptor for the MSD implemented in this application. The *wMaxPacketSize* field is set to 512 bytes.

TABLE 5: BULK IN ENDPOINT DESCRIPTOR

Offset	Field	Size (Bytes)	Value (Hex)	Description
0	*bLength*	1	0x07	Size of Configuration Descriptor in bytes.
1	*bDescriptorType*	1	0x05	ENDPOINT descriptor.
2	*bEndpointAddress*	1	0x81	Endpoint 1, direction input.
3	*bmAttributes*	1	0x02	This endpoint supports bulk transfers.
4	*wMaxPacketSize*	2	0x0200	Max size of this endpoint is 512 bytes for High-Speed devices and 64 bytes for Full-Speed devices.
6	*bInterval*	1	0x00	Does not apply to Bulk endpoints. Set it to ‘0’.

Bulk OUT Endpoint Descriptor

The bulk OUT endpoint is used to receive command and data transferred from Host to device.

Table 6 provides details of the Bulk OUT Endpoint Descriptor for the MSD implemented in this application. The *wMaxPacketSize* field is set to 512 bytes.

TABLE 6: BULK OUT ENDPOINT DESCRIPTOR

Offset	Field	Size (Bytes)	Value (Hex)	Description
0	*bLength*	1	0x07	Size of Configuration Descriptor in bytes.
1	*bDescriptorType*	1	0x05	ENDPOINT descriptor.
2	*bEndpointAddress*	1	0x01	Endpoint 1, direction output.
3	*bmAttributes*	1	0x02	This endpoint supports bulk transfers.
4	*wMaxPacketSize*	2	0x0200	Max size of this endpoint is 512 bytes for High-Speed devices and 64 bytes for Full-Speed devices.
6	*bInterval*	1	0x00	Does not apply to Bulk endpoints., set it to '0'.

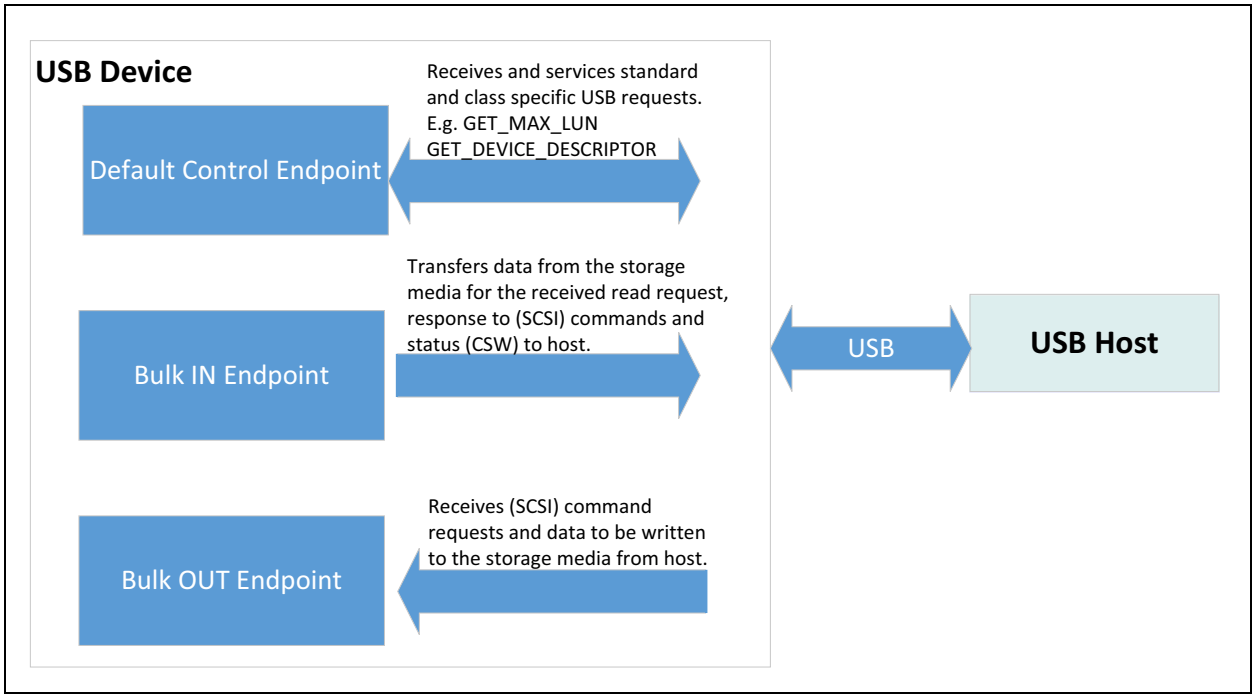
In summary, a USB Mass Storage Class Device (MSD) consists of 1 control endpoint and 2 bulk endpoints.

- Control endpoint (endpoint 0)
- Bulk IN endpoint
- Bulk OUT endpoint

As shown in Figure 5, the bidirectional control endpoint is used for enumerating the device and to respond to Standard and Class specific (Mass Storage Class in this case) USB requests. This endpoint is typically endpoint 0 and is always enabled, and it does not require a descriptor.

The Bulk OUT endpoint receives (SCSI) commands and the media data to be written to the device from the Host. The device sends (SCSI) command replies, (SCSI) commands status, and the requested media data to the Host through the Bulk IN endpoint.

FIGURE 5: USB MSD ENDPOINTS



USB MASS STORAGE CLASS SPECIFIC REQUESTS

This section describes the different Mass Storage Class specific Control Requests. The USB Mass Storage Class features only two class specific control requests, Bulk-only Mass Storage Reset (BOMSR), and Get Max LUN.

Bulk-only Mass Storage Reset (BOMSR)

This request is used to reset the mass storage device and its associated interfaces. After completion of this command, the device becomes ready to receive a new command. Despite the reset command, the bulk data toggle bits and the STALL condition on endpoints should not change. The BOMSR being a class specific control request is issued by the Host on Control Endpoint 0.

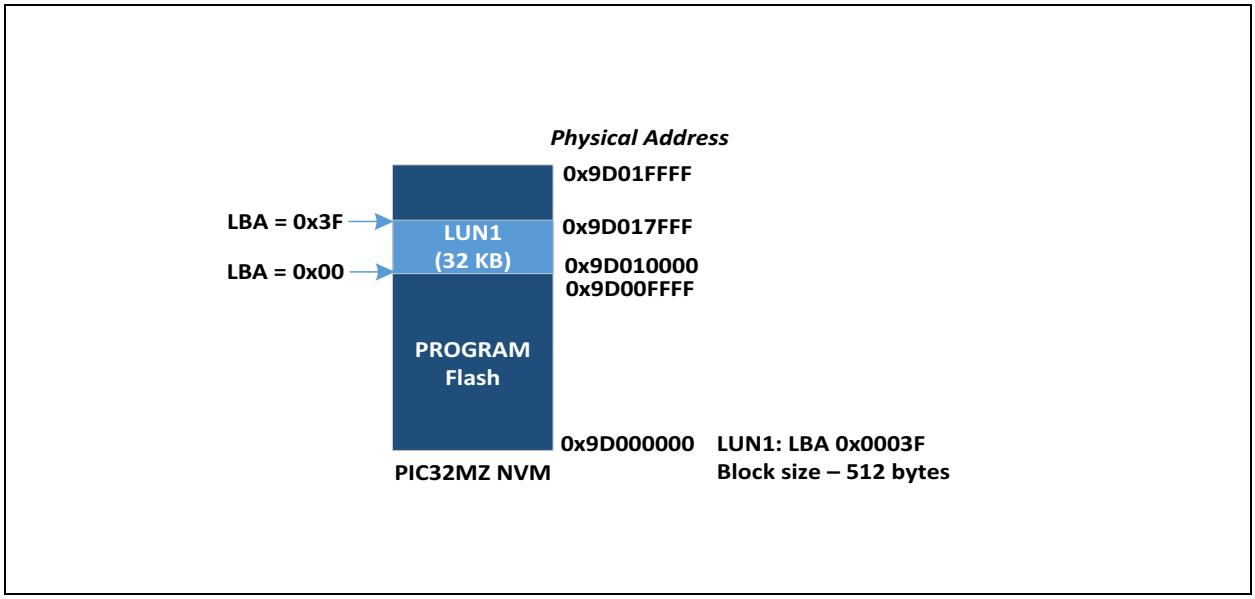
Get Max LUN

A Logical Unit is a contiguous and individually addressable unit presented by the USB Device to the USB Host. A logical unit number (LUN) identifies a logical unit, a device which can be addressed by the SCSI protocol. A device can implement multiple logical units which are numbered from 0 to a maximum of 15. The USB Host sends a Get Max LUN request to learn how many Logical Units are contained in the device. The device shall report one byte of data that contains maximum LUNs supported by the device. If the device supports 4 LUNs, a value of 3 shall be returned and the LUNs shall be numbered from 0 to 3. A device with no logical units shall report a value of 0. Devices that do not support multiple LUNs may stall this request. The Get Max LUN being a class specific request is issued by the Host on Control Endpoint 0.

A Logical Block Address (LBA) runs from 0 to the size of the logical unit. The starting LBA for each of the logical units is 0 and the end address depends on the size of each logical unit. For each LUN, the USB Host will query the capacity of the LUN to learn about the end address of the logical unit. With the combination of LUN and LBA, Host and device can identify the addressed memory location.

Figure 6 shows the 32 KB of internal Non-Volatile Memory (NVM) configured as the second logical unit in this application (SD card is configured as the first logical unit). Typical USB MSD Host access memory in units called sector. The size of a sector is generally 512 bytes. As a result, although the internal block size (row size) of PIC32MZ NVM is 2048 bytes it is presented to the Host as logical blocks (sectors) of 512 bytes (64 blocks x 512 bytes per block = 32KB). The Host addresses the memory by passing to the Device the LUN and LBA and the number of consecutive sectors (sector = 512 bytes) to read and write. The device must translate LUN and LBA to actual physical address.

FIGURE 6: NVM CONFIGURATION



USB MASS STORAGE BULK-ONLY TRANSPORT (BOT)

The MSC BOT protocol transports the command set supported by the device. In that, the BOT protocol wraps the commands as they are transferred over USB to the device. For external hard drives or thumb drives, these commands implement the SCSI protocol.

The BOT protocol divides a data transfer into three stages:

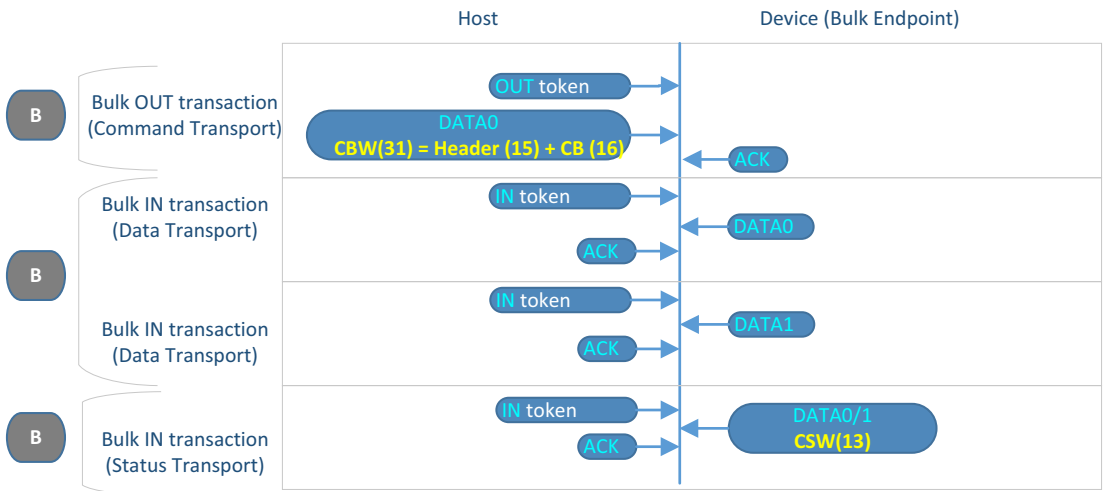
- Command Transport (OUT transaction)
- Data Transport (IN transaction or OUT transaction)
- Status Transport (IN transaction)

Figure 7 shows the transactions involved in a Bulk Read from the Host. The USB Host issues a read command in an OUT transaction followed by one to multiple IN transactions to read the data from the device. The data transfer ends with the USB Host requesting the status from the device in an IN transaction.

The wrapper used for transmitting (SCSI) commands over USB is Command Block Wrapper (CBW). The status of (SCSI) commands is transmitted in a Command Status Wrapper (CSW).

Each transfer begins with the Host sending a (SCSI) command wrapped in a CBW to the device on a Bulk-OUT endpoint. Depending on the command, there may or may not be a Data stage. If there is a Data stage, the Device receives data from the USB Host on the Bulk-OUT endpoint and transmits data to the USB Host on the Bulk-IN endpoint. After the Data stage is complete (or if there is no Data stage), the USB Host requests status from the Device. The Device returns the status wrapped in a CSW to the Host on the Bulk-IN endpoint, which marks the end of the transfer.

FIGURE 7: BULK READ TRANSACTIONS



Transactions involved in a Bulk read from host

bInterfaceClass = 0x50 - Command, Data and Status are through Bulk transfers (BBB).
bInterfaceSubClass = 0x06 – (SCSI) commands are transmitted under CBW and status in CSW.

Command Block Wrapper (CBW)

Table 7 provides the structure of the Command Block Wrapper (CBW). A CBW contains a 32-bit signature to identify it as a CBW packet, a tag to associate a CSW with its corresponding CBW, number of bytes to transfer in the Data Stage, direction of Data Stage, the LUN to which the command block is being sent, length of the Command Block and the Command Block (this is the SCSI command payload) itself.

TABLE 7: CBW STRUCTURE

Byte	Bit							
	7	6	5	4	3	2	1	0
0-3	*dCBWSignature*							
4-7	*dCBWTag*							
8-11 (0x08-0x0B)	*dCBWDataTransferLength*							
12 (0x0C)	*bmCBWFlags*							
13 (0x0D)	Reserved (0)				*bCBWLUN*			
14 (0x0E)	Reserved (0)			*bCBWCBLength*				
15-30 (0x0F-0x1E)	*CBWCB*							

dCBWSignature:

Contains the value 43425355. This identifies this data packet as a CBW.

dCBWTag:

A 32-bit command block tag sent by the Host to associate the CSW with its corresponding CBW. The Device shall echo the tag in the *dCSWTag* field of the CSW.

dCBWDataTransferLength:

Indicates the number of bytes of data Host expects to transmit or receive (based on the direction bit in *bmCBWFlags*). If this is set to zero, the device will ignore the direction bit in *bmCBWFlags* and there will be no data transfer between CBW and the associated CSW.

bmCBWFlags:

Bit 7:

1 = Data-In from device to Host (Host wants to read)
0 = Data-Out from Host to Device–(Host wants to write)

Note:	The device will ignore this bit if the *dCBWDataTranferLength* is set to zero.

Bit 6: Obsolete - Host will set it to zero.

Bit 5-0: Reserved - Host will set it to zero.

bCBWLUN:

The Logical Unit Number (LUN) of the device to which this Command Block is being sent. The USB Host will obtain the number of LUNs supported by the device by issuing the class specific request Get Max LUN. For devices that support multiple LUNs, the Host shall place into this field the LUN to which this command block is addressed. If the device has a single LUN, the Host will set it to zero.

bCBWCBLength:

This defines the valid length of Command Block (CBWCB). Valid values can be from 1 to 16.

CBWCB:

This contains the Command Block to be executed by the device. The device shall interpret the first *bCBWCBLength* bytes in this field as a Command Block defined by the command set identified by *bInterfaceSubClass* field of Interface Descriptor. For Mass Storage Device Class the Command Block will contain one of the SCSI commands.

Command Status Wrapper (CSW)

The MSD sends the status of the command to the Host in a Command Status Wrapper (CSW). Table 8 shows the structure of a CSW.

TABLE 8: CSW STRUCTURE

Byte	Bit							
	7	6	5	4	3	2	1	0
0-3	*dCBWSignature*							
4-7	*dCBWTag*							
8-11 (0x08-0x0B)	*dCBWDataResidue*							
12 (0x0C)	*bmCBWStatus*							

dCSWSignature:

Contains the value 53425355h that helps identify the data packet as a CSW.

dCSWTag:

The device will set this field to the value received in the dCBWTag field of the associated CBW.

dCSWDataResidue:

For Data-Out, the device will set this value to the difference of *dCBWDataTransferLength* received in CBW and the actual amount of data processed by the device. For Data-In, the device will set this value to the difference of *dCBWDataTransferLength* received in CBW and the actual data sent by the device.

bCSWStatus:

Indicates the status of the (SCSI) command. The possible values for *bCSWStatus* are shown in the following table.

TABLE 9: SCSI COMMAND VALUES

Value (Hex)	Description
0x00	Command passed.
0x01	Command failed.
0x02	Phase error. Host will perform a reset recovery.
0x03-0x04	Obsolete (Reserved).
0x05-0xFF	Reserved.

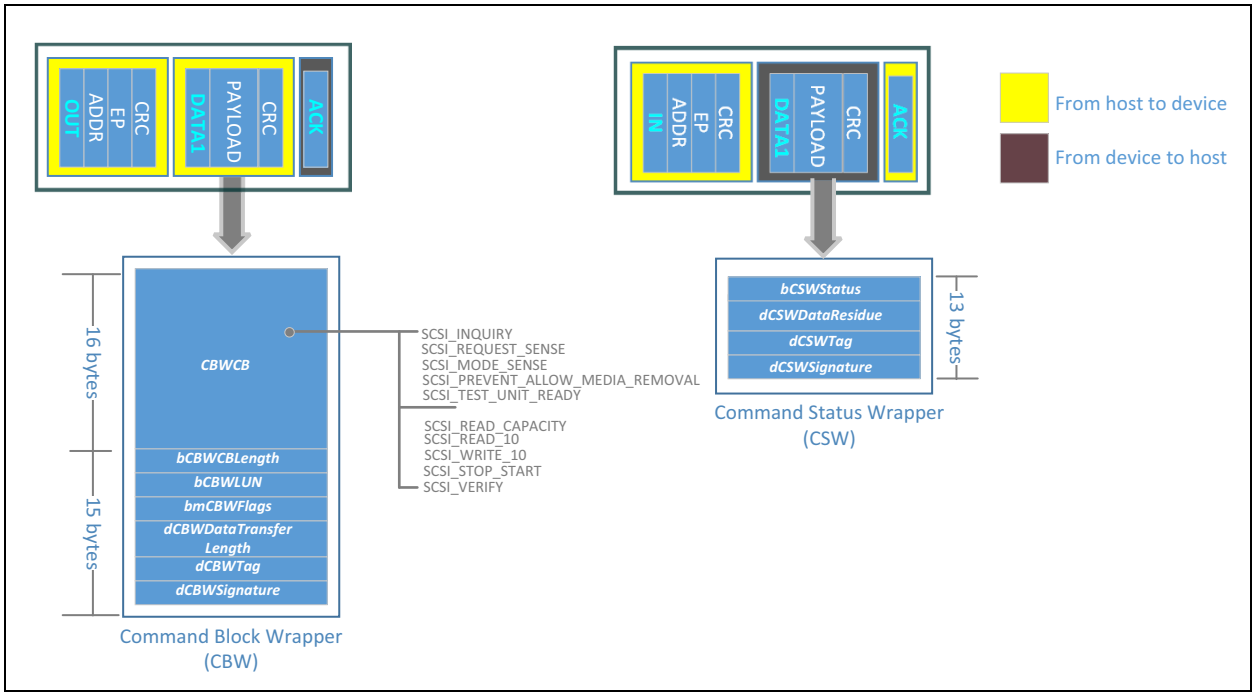
Phase Error indicates that the device has encountered an internal error, for which it has no reliable means of recovering, other than a reset.

Upon receiving a Phase Error, the USB Host should perform a reset recovery on the device by issuing the following commands in order on the Control endpoint.

1. Bulk-Only Mass Storage Reset:
   This is a Class specific Interface request. On completion of the request, the device is ready to receive a new CBW. The reset should not alter the data-toggle bits and the endpoint stall conditions.
2. Clear Feature HALT to the Bulk-In endpoint:
   This is a Standard Endpoint request. The device shall reset the endpoint's data toggle to DATA0. The endpoint should resume normal communication if possible.
3. Clear Feature HALT to the Bulk-Out endpoint:
   This is a Standard Endpoint request. The device shall reset the endpoint's data toggle to DATA0. The endpoint should resume normal communication if possible.

Figure 8 shows transmission of CBW and CSW in USB transactions.

FIGURE 8: CBW AND CSW TRANSMISSION DURING USB TRANSACTIONS



SMALL COMPUTERS SYSTEM INTERFACE (SCSI) PROTOCOL

Small Computers System Interface (SCSI) is a family of protocols that enables systems to communicate with storage devices. It provides applications with a standard architecture and a standardized command set for different class of devices. It thereby abstracts the underlying media and provides a standard mechanism to access different media types. SCSI provides the means for reading, writing and checking the status of the media, along with other operations.

A typical MSD uses the SCSI shared commands: SCSI Primary Commands (SPC) and the Device specific commands, SCSI Block Commands (SBC).

The SCSI Primary Commands are common to all device types. The SCSI Block Commands are specific to block devices. These are applicable to logical unit that declares itself to be a direct-access block device in the PERIPHERAL DEVICE TYPE field of the standard SCSI INQUIRY response.

Table 10 shows the SCSI commands supported by the MPLAB Harmony MSD SCSI implementation.

TABLE 10: SCSI COMMANDS SUPPORTED BY MPLAB HARMONY

SCSI Primary Commands (SPC)	SCSI Block Commands (SBC)
SCSI INQUIRY (Mandatory)	SCSI READ CAPACITY (10) (Mandatory)
SCSI REQUEST SENSE (Mandatory)	SCSI READ (10) (Mandatory)
SCSI MODE SENSE (Optional)	SCSI WRITE (10) (Mandatory for writable devices)
SCSI TEST UNIT READY (Mandatory)	—

SCSI Primary Commands

SCSI INQUIRY (OPCODE 0x12)

The USB Host issues the SCSI INQUIRY command to get information about the device, such as the Peripheral Device Type, Version of SPC supported, Vendor and Product identifications.

The MSD implemented in this application note sends 36 bytes in response to the SCSI INQUIRY command, as provided in the following table.

The Peripheral Device Type is set to '0', which indicates that a Direct Access Block Device is connected to its logical unit. The specifications for a direct access block Device are further specified in the SBC standard.

The RMB bit is set to '1' which indicates that the media is removable.

The VERSION is set to 0x04, which specifies that the device complies with SPC-2 version of the specification.

The VENDOR IDENTIFICATION specifies the vendor identification code specified by the T10 technical committee.

For additional information, refer to the **Section 7.3.2 "Standard INQUIRY Data"** of the SPC-2 Specification.

TABLE 11: SCSI INQUIRY COMMAND

Byte	Bit							
	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	PERIPHERAL QUALIFIER (0)			PERIPHERAL DEVICE TYPE (0)				
1	RMB (1)	Reserved						
2	VERSION (0x04)							
3	AERC (0)	Obsolete (0)	NormACA (0)	HiSup (0)	RESPONSE DATA FORMAT (0x02)			
4	ADDITIONAL LENGTH (n-4) (0x1F)							
5	SCCS (0)	Reserved						
6	BQUE (0)	ENCSERV (0)	VS (0)	MULTIP (0)	MCHNGR (0)	Obsolete (0)	Obsolete (0)	ADDR16 (0)
7	RELADR (0)	Obsolete (0)	WBUS16 (0)	SYNC (0)	LINKED (0)	Obsolete (0)	CMDQUE (0)	VS (0)
8 15	VENDOR IDENTIFICATION ("Microchp")							
16 31	PRODUCT IDENTIFICATION ("Mass Storage")							
32 35	PRODUCT REVISION LEVEL ("0001")							

SCSI MODE SENSE (OPCODE 0x1A)

The USB Host issues a MODE SENSE command to read device specific parameters. The device sends the response to a MODE SENSE command as shown in the following table.

TABLE 12: MODE SENSE COMMAND

Byte	Bit							
	7	6	5	4	3	2	1	0
0	MODE DATA LENGTH (0x03)							
1	MEDIUM TYPE (0x00)							
2	DEVICE-SPECIFIC PARAMETER (see **Note 1**)							
3	BLOCK DESCRIPTOR LENGTH (0x00)							

Note 1: The contents of these fields are described in the SBC Specification.

The MEDIUM TYPE is defined by the SBC Specification and is set to 0x00.

The DEVICE-SPECIFIC PARAMETER is defined by the SBC Specification as shown in the following table.

TABLE 13: DEVICE-SPECIFIC PARAMETER

Bit	7	6	5	4	3	2	1	0
Name	WP			DPOFUA	Reserved			

The WP bit indicates whether or not the medium is write-protected. The device checks the write protection status of the media before setting this bit.

The DPOFUA bit is set to '0' indicating that the device does not support caching.

The MPLAB Harmony USB device stack will fail this command by setting the *bCSWStatus* bit in CSW to 0x01 and updates the SENSE data if the media is not present. This is true for all other SCSI commands that depend on the media for a response.

Note:	For additional information, refer to the **Section 8.3.3** of the SPC-2 Specification and **Section 6.3.1** of the SBC-3 Specification.

SCSI REQUEST SENSE (OPCODE 0x03)

The Host can issue a REQUEST SENSE command to learn more about the failure of the previous command, after having received a command failed status for that command. The *bCSWStatus* field in the CSW indicates whether a command has failed or passed. At this point, the device will also record the error.

The device responds to the SCSI REQUEST SENSE command with a sense data structure. While the sense data structure has many fields, only the following three fields are important: The first field is the Sense Key, which indicates the result of the last command, such as Media not ready, illegal command request, media protected and so on. Two other fields, Additional Sense Code (ASC) and Additional Sense Code Qualifier (ASCQ), provide accurate descriptions of the problems, for example, Media not present, illegal command opcode, write-protected.

SCSI TEST UNIT READY (OPCODE 0x00H)

The USB Host issues the TEST UNIT READY command to check whether the media device is ready. The Host can also issue this command when the sense data indicates that the media is not ready. This command does not have a data stage. If the device is not ready, the *bCSWStatus* field in CSW is set to 0x01 (command failed). The Host can retry sending the TEST UNIT READY command until the device reports 00h in the *bCSWStatus* field in CSW, which indicates that media is ready.

For pluggable medias, such as the SD Card, the media is considered to be in a ready state if the media driver detects the media to be present and the media has been successfully initialized and is ready to be used.

For non-removable media, such as the NVM (internal Flash memory), the media is always considered to be in a ready state.

SCSI Block Commands

SCSI READ CAPACITY (10) (OPCODE 0x25)

The Host issues the READ CAPACITY command to learn how many bytes the device media can store. The device responds with a structure containing LBA of the last block on the media and the block size during the data stage, as shown in the following table.

TABLE 14: BLOCK SIZE AND BLOCK ADDRESS

Byte	Bit							
	7	6	5	4	3	2	1	0
0	LOGICAL BLOCK ADDRESS							
3								
4	BLOCK SIZE IN BYTES							
7								

This puts a theoretical limit of 2TB (0xFFFFFFFF x 512) for a block size of 512 bytes.

A logical unit of size 256 MB and a block size of 512 bytes shall report an LBA of 0x7FFFF and a block size of 512 bytes.

A logical unit of size 32768 bytes and an internal block size of 2048 bytes (NVM memory in case of this application) shall report LBA of 0x3F [(32768 / 512) - 1] and a (logical) block size of 512 bytes.

SCSI READ (10) (OPCODE 0x28)

The USB Host issues the SCSI READ command to read data from the device media. The Host will specify the starting address in the LOGICAL BLOCK ADDRESS and the number of contiguous logical blocks to read in the TRANSFER LENGTH field of the READ request. The LUN ID is specified in the *bCBWLUN* field of the CBW header. The device will read the requested logical blocks from the media and send it to the Host in IN data stage in chunks of 512 bytes.

Figure 9 shows the transactions involved in a USB Read operation.

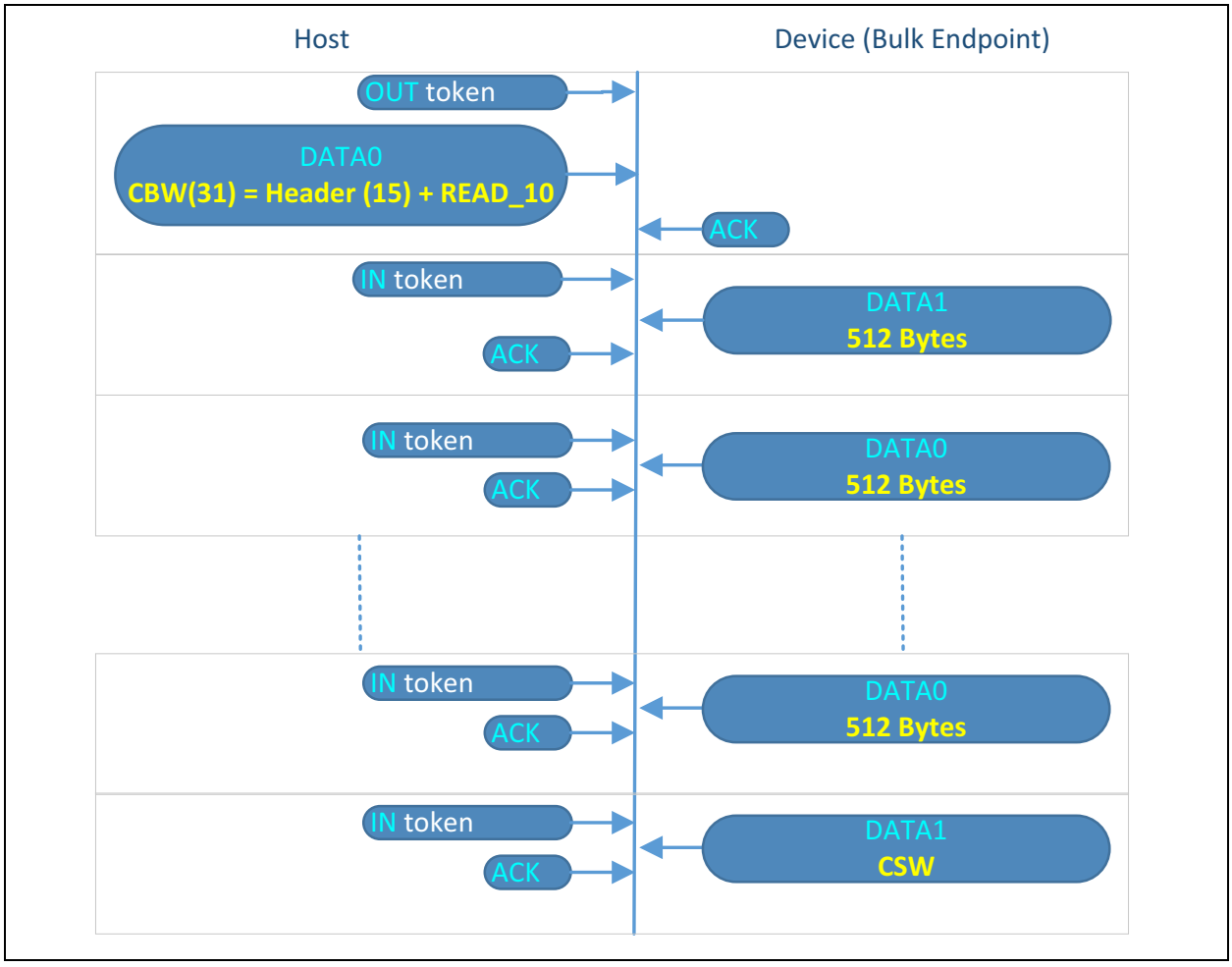
The MPLAB Harmony USB device stack allows buffering of data read from the media. For example, if the user has configured a buffer size of 8 sectors (size of one sector = 512 bytes), then a request to read 12 logical blocks of data (12 sectors or 6KB) will result in only two read requests to the media. During the first read request, 8 sectors (4KB) worth of data shall be read and saved in the buffer. The remaining 4 sectors (2KB) shall be read in the next read request to the media.

If the media data is not available, the device sends a NAK in response to the IN data request from the Host, while the device is reading from the media. The Host then retries the IN data transaction. With the buffering enabled, the media data is already available in the RAM buffer, and therefore, the IN data stage from the Host can be serviced without any media read delays. This results in lesser number of retries from the Host which in turn increases the overall throughput of the read operation, but, at the expense of increased RAM usage.

The Host shall receive all the 12 logical blocks of data over a period of 12 IN data transactions with each IN transaction transferring 512 bytes to Host.

The USB Bulk-only transport specification does not allow the USB Host to transfer a CBW to the device until the Host has received CSW for any outstanding CBW. This allows a common buffer to be shared by all the logical units supported by the USB Mass Storage Device Stack.

FIGURE 9: USB READ OPERATION TRANSACTIONS



SCSI WRITE (10) (OPCODE 0x2A)

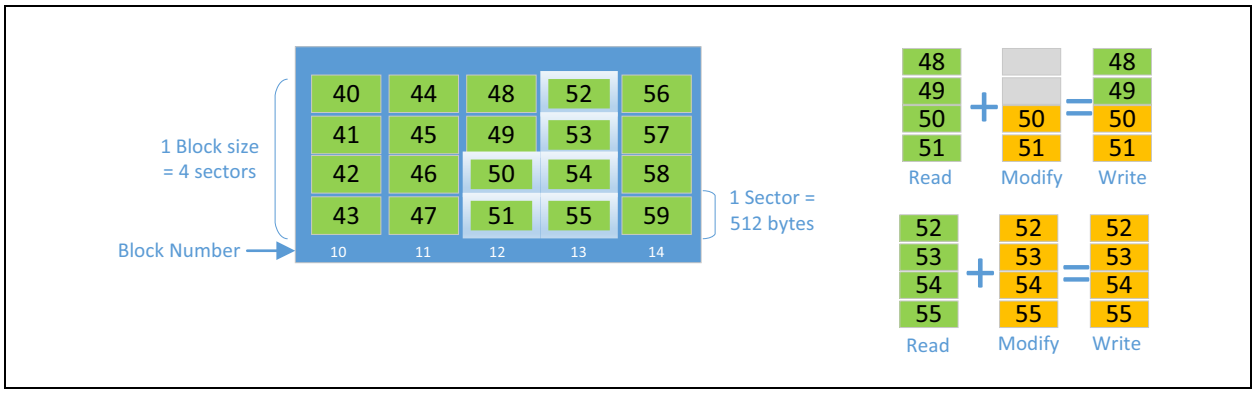
The USB Host issues the SCSI WRITE command to write data to the device media. The USB Host will specify the starting address in the LOGICAL BLOCK ADDRESS and the number of contiguous logical blocks to write in the TRANSFER LENGTH field of the WRITE request. The LUN ID is specified in the *bCBWLUN* field of the CBW header.

The MPLAB Harmony Device Stack performs a read-modify-erase-write operation for media whose block size is greater than 1 sector (or 512 bytes). For the MSD example implemented in this application, the write block (row) size of the NVM media is 4 sectors (or 2048 bytes). Figure 10 shows a WRITE request with LOGICAL BLOCK ADDRESS is set to 50, and TRANSFER LENGTH is set to 6, the device will first read the internal block number 12 (sectors 48-51) and modify the sectors 50 and 51 with the data received from the Host. The complete modified block (sectors 48-51) is written back to the media. After this, contents of block number 13 (sectors 52-55) are read and is modified with the data received from the Host. The complete modified block is written back to the media.

The Host will complete sending 6 logical blocks of data over a period of 6 OUT transactions, with each OUT transaction transmitting 512 bytes to the device. The MPLAB Harmony Device Stack reports a GOOD status to the Host only after the data has actually been written to the media.

For all the unsupported SCSI commands, the USB device sets the *bCSWStatus* field of the CSW to 01h (Command failed) and updates the sense data by setting the Sense Key to 0x05 (Illegal Request) and the Additional Sense Code to 0x20 (Invalid Command Opcode).

FIGURE 10: READ-MODIFY-ERASE-WRITE OPERATION

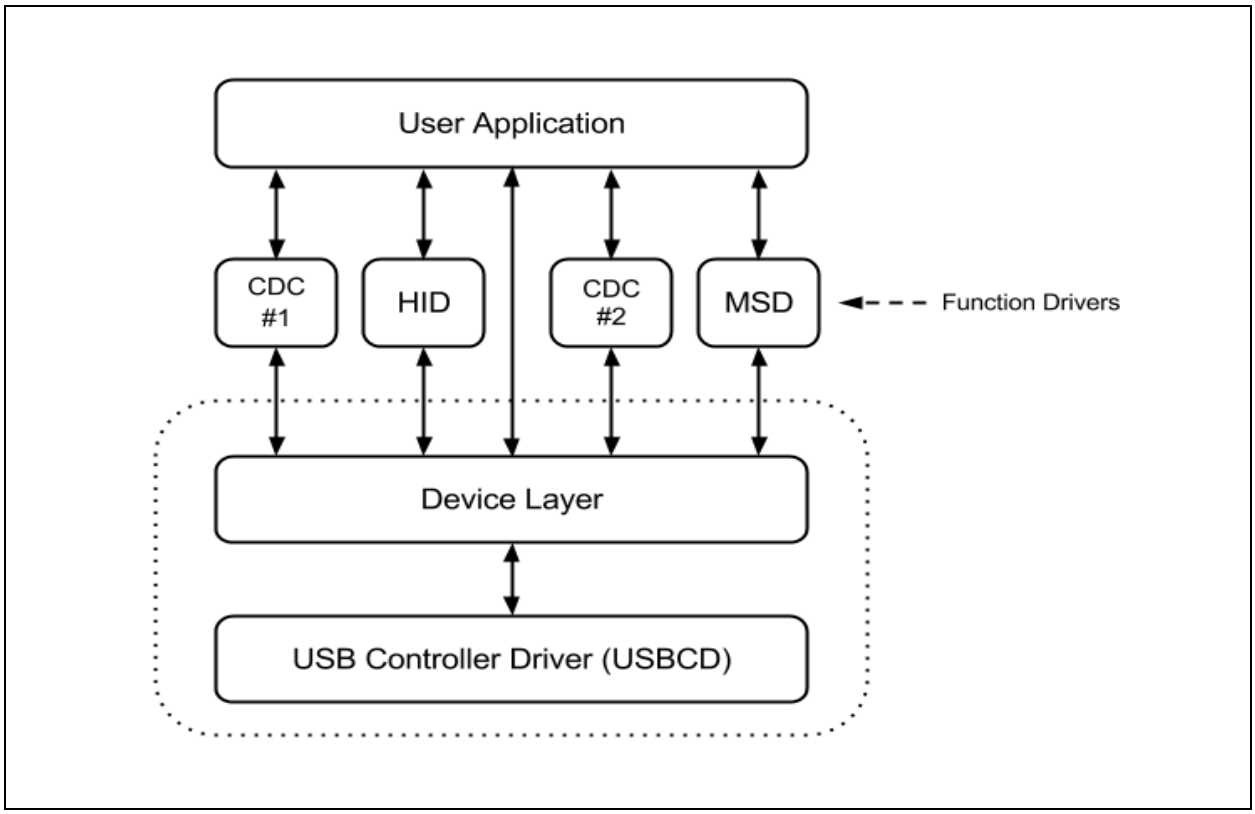


MPLAB HARMONY USB DEVICE LIBRARY ARCHITECTURE OVERVIEW

The MPLAB Harmony USB Device Library features a modular and layered framework that allows developers to design and develop a wide variety of USB devices. The USB Device Library facilitates the development of standard USB devices through function drivers that implement standard USB device class specification. The USB Device Library consists of the following three major components, as shown in Figure 11.

- USB Controller Driver (USBCD)
- Device Layer
- Function Drivers

FIGURE 11: USB DEVICE LIBRARY COMPONENTS



USBCD

The USB Controller Driver (USBCD) manages the state of the USB peripheral, and provides the Device Layer with structured data access methods to the USB. It also provides the Device Layer with USB events. It supports only one client, the Device Layer, per instance of USB peripheral. The USBCD is accessed exclusively by the Device Layer. It is recommended that the USBCD be configured for interrupt mode. This will reduce the impact of other application components on the operation of USB device stack.

The USBCD provides functions to perform these:

- Enable, disable, and stall endpoints
- Schedule USB transfers
- Attach or detach the device
- Control resume signaling

Device Layer

The Device Layer responds to the enumeration requests issued by the USB Host.

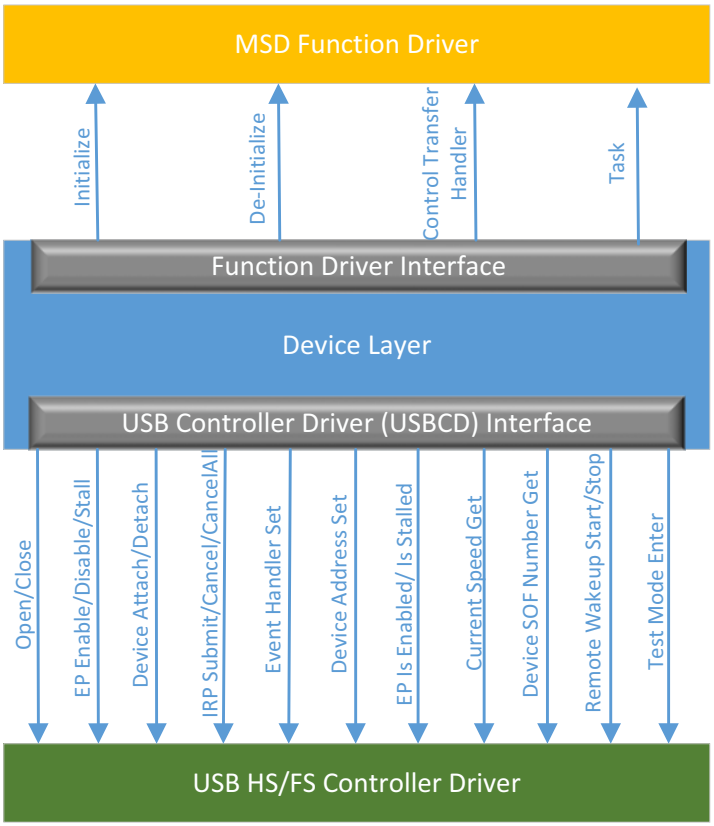
It has exclusive access to an instance of the USBCD and the control endpoint (Endpoint 0). When the Host issues a Class specific Control transfer request, the Device Layer will analyze the setup packet of the Control transfer and will route the Control transfer to the appropriate Function Driver. Figure 12 shows how the Device Layer interfaces with the USBCD and the Function Driver (MSD in this case).

The Device Layer initializes all function drivers that are registered with it when it receives a Set Configuration (for a supported configuration) request from the USB Host. It deinitializes the function drivers when a USB reset event occurs. It opens the USBCD and registers an event handler to receive USB events. The Device Layer can also be opened by the application (the application becomes a client to the Device Layer). The application can then receive bus and device events and respond to control transfer requests. The Device Layer provides events to the application such as Device configured or Device reset. Some of these events are notification-only events, while other events require the application to take action.

Function Drivers

The Function Drivers implements various USB device classes as per the class specification.

FIGURE 12: INTERFACING THE USBCD AND FUNCTION DRIVER WITH THE DEVICE LAYER



USB MASS STORAGE DEVICE FUNCTION DRIVER

Figure 13 illustrates the functional interaction between the application, the MSD Function Driver, the media drivers, and the USB Device Layer.

As shown in Figure 13, the application does not have to interact with MSD Function Driver. Also, the MSD Function Driver does not have application callable functions. The media drivers control the storage media. The application interacts with the media drivers to update or access the information on the storage media. The MSD Function Driver interacts with the media drivers to process data read and write requests that it receives from the USB Host. This data is always accessed in blocks.

The Device Layer initializes the MSD Function Driver when the Host sets the configuration (Set Configuration control request) that contains the Mass Storage Interfaces. The MSD Function Driver requires an initialization data structure to be defined for each instance of the Function Driver.

This initialization data structure should be of the type `USB_DEVICE_MSD_INIT_DATA`.

This initialization data structure contains the following:

- The number of Logical Unit Numbers (LUNs) in this MSD Function Driver instance
- A pointer to the `USB_MSD_CBW` type data structure
- A pointer to the `USB_MSD_CSW` type data structure.
- A pointer to the array of media driver initialization data structure

Figure 14 illustrates the MSD Function Driver initialization data structure.

FIGURE 13: FUNCTIONAL INTERACTION

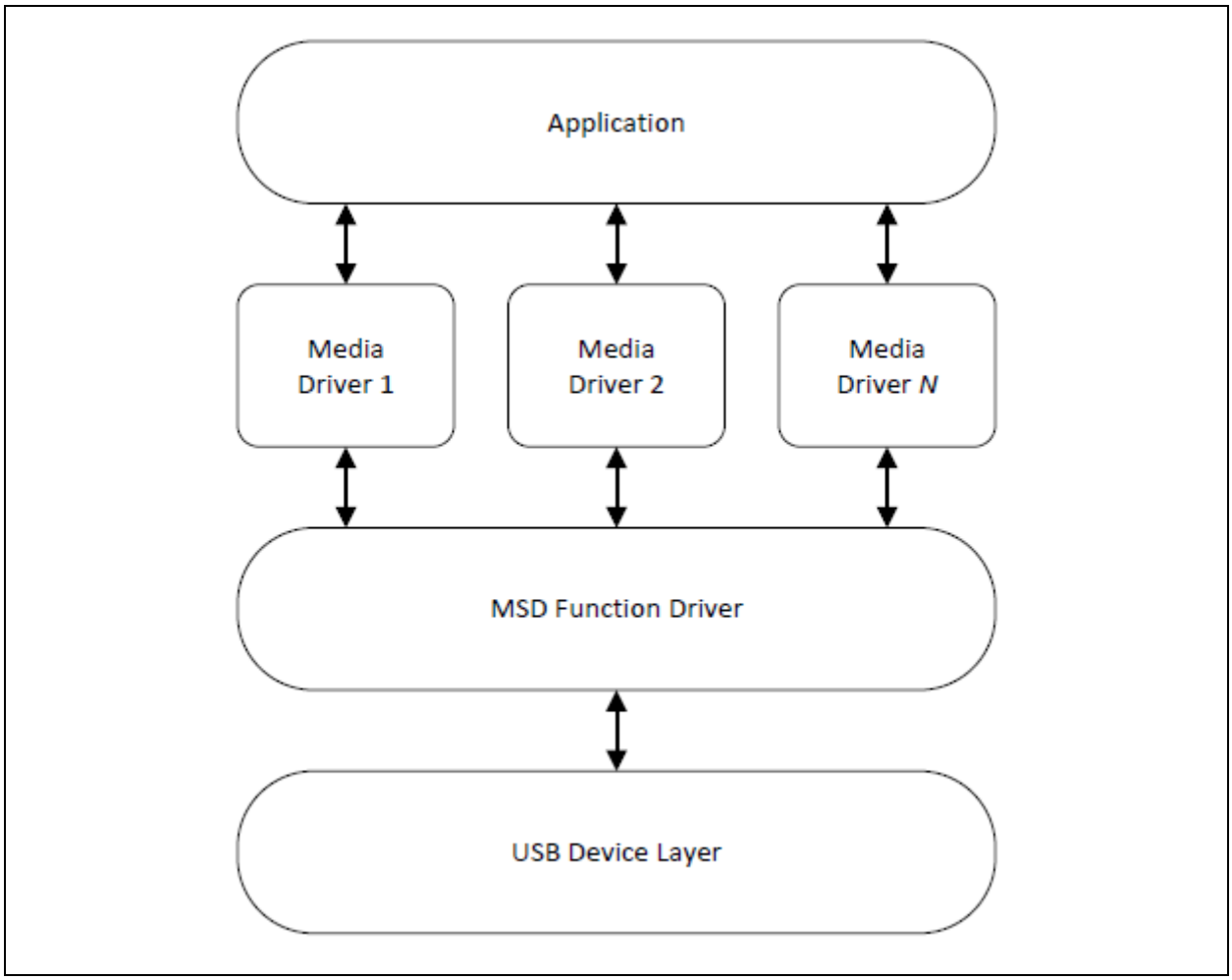
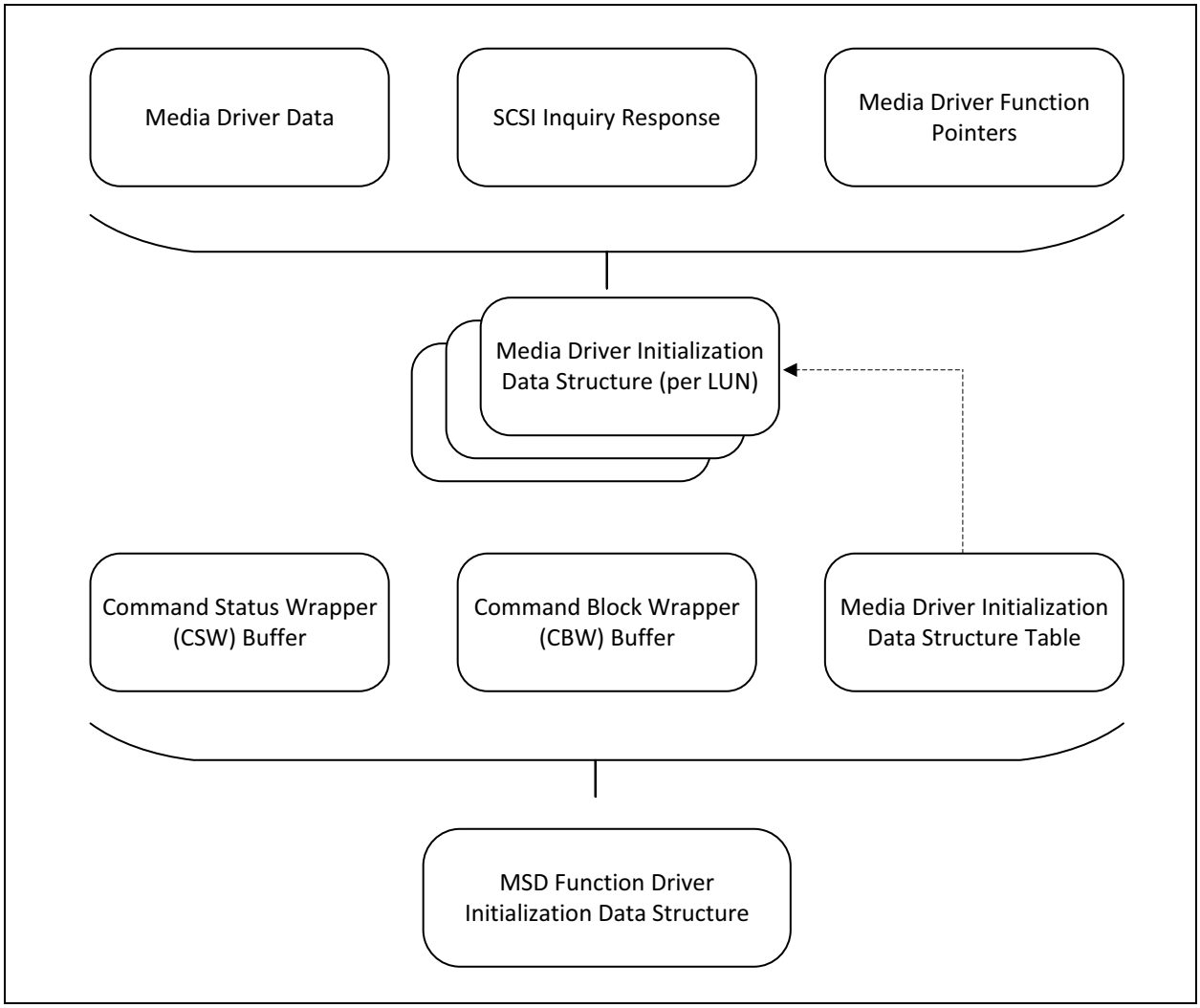


FIGURE 14: MSD FUNCTION DRIVER INITIALIZATION DATA STRUCTURE



During initialization, MSD Function Driver will initialize itself with the initialization data available in the `USB_DEVICE_MSD_INIT_DATA` data structure. The MSD Function Driver will also enable Bulk Endpoints and will have an exclusive access to the Bulk Endpoint.

A media driver is plugged into the MSD Function Driver by providing a media driver entry point in the MSD Function Driver initialization data structure. In the case of multi-LUN storage, multiple media drivers can be plugged into the MSD Function Driver, with each one being capable of accessing different storage media types.

The MSD Function Driver Tasks function is invoked in the context of the Device Layer Tasks function. The MSD Function Driver interfaces should be registered in the USB Device Layer Function Driver Registration Table.

The USB Control requests are first handled by the Device Layer and then forwarded to the MSD Function Driver to allow handling of standard interface requests GET/SET INTERFACE and class specific interface requests GET MAX LUN and USB MSD RESET in case of MSD. The standard control requests for Bulk Endpoints are handled by the Device Layer itself. The MSD Function Driver accesses the USB bus and Endpoint 0 through the Device Layer, as these are managed by the Device Layer.

Media Interface

The `USB_DEVICE_MSD_MEDIA_INIT_DATA` data structure allows a media driver to be plugged into the MSD Function Driver. Any media driver that needs to be plugged into the MSD Function Driver needs to implement the interface (function pointer signatures) specified by the `USB_DEVICE_MSD_MEDIA_FUNCTIONS` type.

For every LUN, a SCSI Inquiry Response data structure needs to be made available.

Following guidelines needs to be adhered to while developing a media driver:

- Read functions should be non-blocking
- Write functions should be non-blocking
- The media driver should provide an event to indicate when a block transfer has complete. It should allow the event handler to be registered.
- Where required, the write function should erase and write to the storage area in one operation. The MSD Function Driver does not explicitly call the erase operation.
- The media driver should provide a media geometry object when required. This media geometry object allows the MSD Function Driver to understand the media characteristics. This object is of the type `SYS_FS_MEDIA_GEOMETRY`.

Data Transfers

Once the device is configured, the Device Layer runs the MSD Function Driver tasks. The MSD Function Driver task's state machine waits for CBW packet from USB Host. Once the CBW packet is received, it is parsed and SCSI command requests are serviced. For SCSI command requests that require media access, based on the received LUN number, the corresponding media's Read and Write functions are called to read and write media sectors.

The media events are notified to the MSD Function Driver through an event handler registered by the MSD Function Driver. While the MSD Function Driver waits with the media driver for the media to complete the operation, the Function Driver will NAK the data stage of the MSD data transfer request. Once the media driver has completed the read/write operation, the MSD Function Driver will respond to the MSD data transfer request by submitting a I/O Request Packet (IRP) to the USB driver through the Device Layer. The USB driver maintains a IRP queue for each endpoint and allows queuing of multiple requests on the corresponding endpoint queue.

Error Handling

Valid CBW - A Device considers a CBW valid if:

- The CBW was received after the Device had sent a CSW or after a reset
- The CBW is 31 (0x1F) bytes in length
- The *dCBWSignature* is equal to 0x43425355

Meaningful CBW - A Device considers a CBW meaningful if:

- No reserved bits are set,
- The *bCBWLUN* contains a valid LUN supported by the Device, and
- Both *bCBWCBLength* and the content of the CBWCB are in accordance with *bInterfaceSubClass*

Not a valid CBW - If the size of CBW packet is greater than 31 bytes or if the *dCBWSignature* field in CBW is not equal to the valid signature (0x43425355), the device will send STALL condition on both IN and OUT endpoints and CSW is not sent. The device will remain in this state until the Host performs a reset recovery by sending BOMSR command followed by Clear Feature (Endpoint Halt) command for both the endpoints to the device. In addition to this, the following checks to be performed:

- Not a meaningful CBW - For a SCSI READ 10 or SCSI WRITE 10 request, if the *dCBWDataTransferLength* field in CBW is not equal to TRANSFER LENGTH field of CBWCB, the command is failed (*bCSWStatus* = 0x01)
- Not a meaningful CBW - For SCSI READ 10 or SCSI WRITE 10 request, if the direction of data transfer (indicated by *bmCBWFlags*) is incorrect, the command is failed.
- For a SCSI READ 10 or SCSI WRITE 10 request, if the media is not present, the command is failed by setting the *bCSWStatus* field in CSW to 0x01 (Command Failed). Please note: SCSI requests that depend on media for a response (like SCSI READ CAPACITY, SCSI MODE SENSE, SCSI TEST UNIT READY) will also be failed if the media is not present. If the media is not present, the SENSE KEY is set to SCSI_SENSE_NOT_READY (0x02) and the Additional Sense Code (ASC) is updated to SCSI_ASC_MEDIUM_NOT_PRESENT (0x3A).
- For SCSI WRITE 10 request, if the media is write protected, the command is failed (*bCSWStatus* = 0x01). At this point, the SENSE data is updated with SENSE KEY set to SCSI_SENSE_DATA_PROTECT (0x07), and the Additional Sense Code (ASC) to SCSI_ASC_WRITE_PROTECTED (0x27). The SENSE data will be sent to Host in response to SCSI REQUEST SENSE command.
- Media Driver Errors - For a valid and a meaningful CBW, the MSD Function Driver will respond with *bCSWStatus* = 0x01 (Command Failed) status in CSW, if any error is reported by the media driver.

If for a valid and a meaningful CBW, the number of bytes transmitted (during IN data transfers) or received (during OUT data transfers) by the MSD Function Driver is less than indicated by the Host in the *dCBWDataTransferLength* field of the CBW packet, the MSD Function Driver will set the *bCSWStatus* field of CSW to 0x00 (Command Passed) and the *dCSWDataResidue* field in CSW will be set to the difference of *dCBWDataTransferLength* and the actual number of bytes received or sent by the device. The device will then STALL the Bulk-IN endpoint.

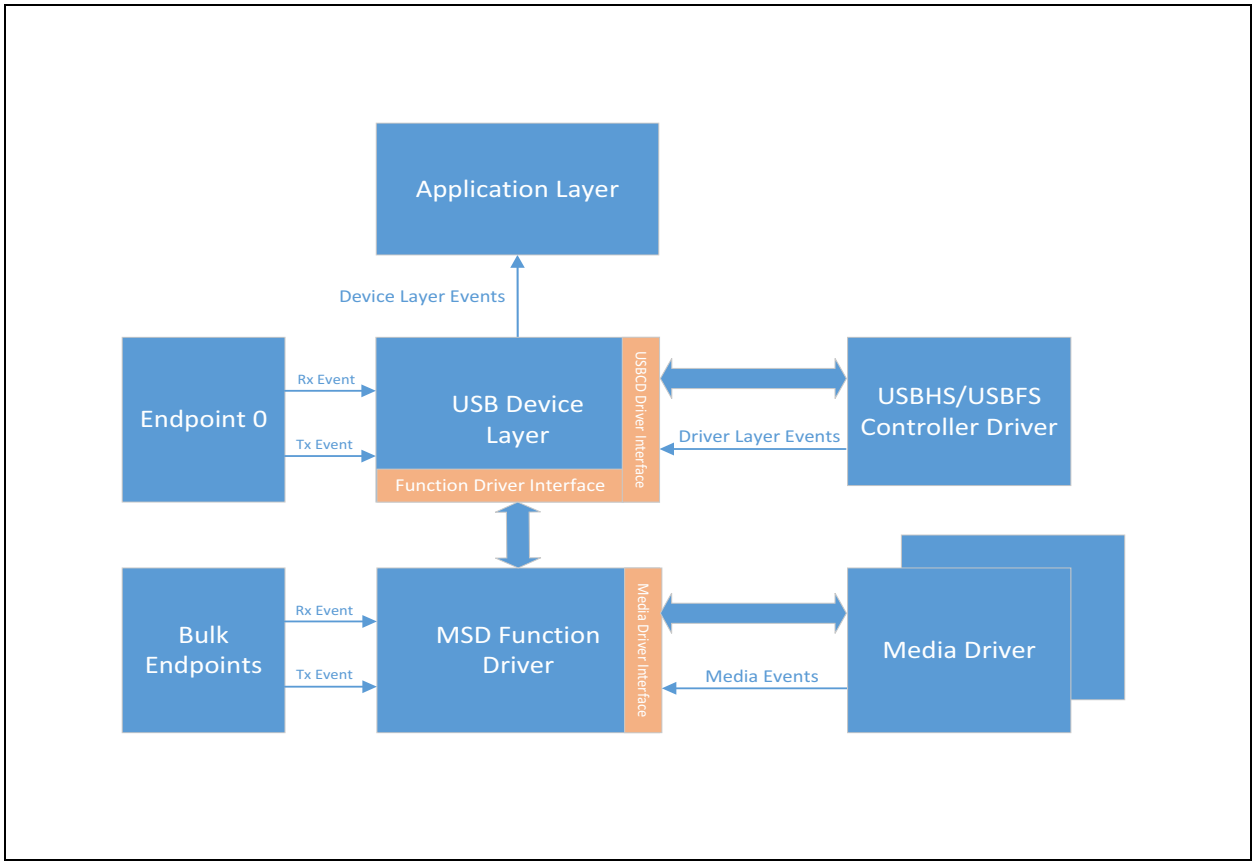
The Host upon attempting to read the CSW will receive a STALL on the IN endpoint from the device. After receiving a STALL, the Host will issue a Clear Feature (Endpoint Halt) control request (with the Bulk-IN endpoint's address) to clear the STALL condition on the Bulk-IN endpoint. Once the STALL condition is cleared, the device will send the CSW to the Host.

If for a valid and a meaningful CBW, the command execution is failed (*bCSWStatus* = 0x01, Command Failed), where the Host expects to receive or send data from/to the device (*dCBWDataTransferLength* > 0), the MSD Function Driver will stall the IN endpoint if the Host is reading (*bmCBWFlags* = 1) data from the device, and will stall the OUT endpoint if the Host is writing (*bmCBWFlags* = 0) data to the device (CSW is sent). When the IN endpoint is stalled, the device will wait for the Host to clear the STALL condition on the bulk IN endpoint by issuing a CLEAR FEATURE request (for Bulk IN endpoint) on the Control endpoint, before sending the CSW.

Events Handling

Figure 15 shows the handling of various events in MSD.

FIGURE 15: HANDLING OF VARIOUS EVENTS IN MSD



The application must register an event handler with the USB Device Layer to receive Device Layer events like `USB_DEVICE_EVENT_CONFIGURED`, `USB_DEVICE_EVENT_POWER_DETECTED`, `USB_DEVICE_EVENT_POWER_REMOVED` etc. The `USB_DEVICE_Attach` function can be called to attach the Device to USB when the USB_DEVICE_EVENT_POWER_DETECTED event occurs.

The Device Layer registers an event handler with the USB Controller Driver to receive events like `DRV_USB_EVENT_RESET_DETECT`, `DRV_USB_EVENT_RESUME_DETECT`, `DRV_USB_EVENT_IDLE_DETECT`, and so on from the controller driver. After handling the event, the Device Layer will call the application event handler to allow the application to handle these events.

The MSD Function Driver registers an event handler with the Media driver corresponding to each logical unit. The MSD Function Driver will receive media events: `SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_COMPLETE` and `SYS_FS_MEDIA_EVENT_BLOCK_COMMAND_ERROR`, after the completion of media operation.

The MSD Function Driver does not provide any events to the application. The application does not have to intervene in the functioning of the MSD Function Driver.

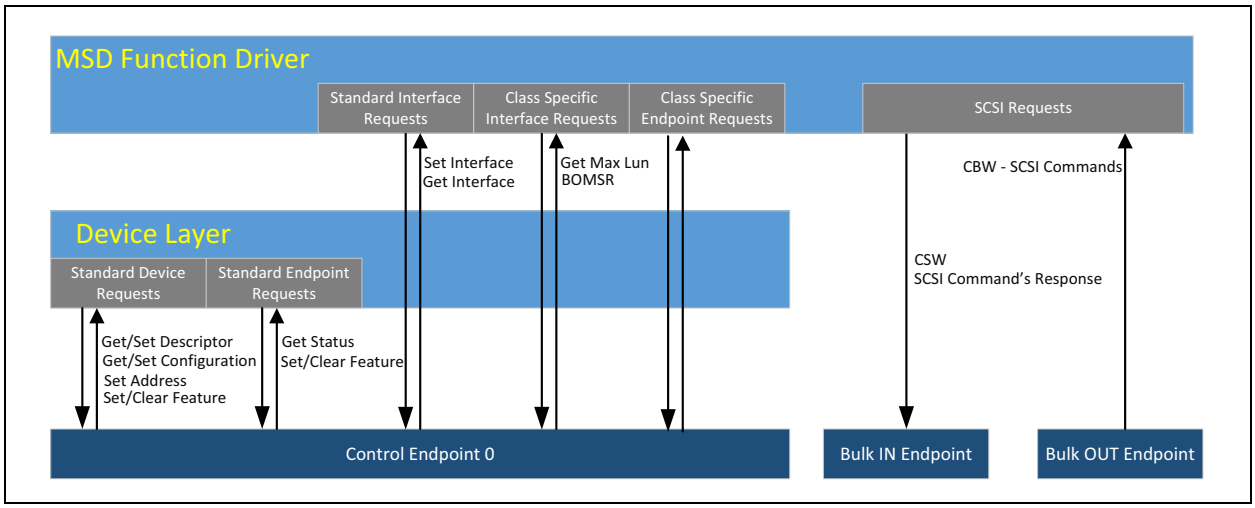
The Device Layer will also register event handlers to receive Endpoint 0 transmit and receive events. These callbacks are called after a Control packet is received or transmitted.

The MSD Function Driver can also register an event handler to receive events for Bulk endpoints that are managed by the MSD Function Driver.

It is possible that the application may also open the media driver while they are already opened by the MSD Function Driver. If the application and the MSD Function Driver try to write to the same media driver, the result could be unpredictable. It is recommended that the application restricts write access to the media driver while the USB Device is plugged into the Host.

Figure 16 shows firmware layers that handle different USB requests in the MPLAB Harmony Device Stack Mass Storage Class. The Device Layer handles the Standard requests for the device and the endpoint. Standard requests for Interface and Class specific requests for Interface and Endpoint are forwarded by the Device Layer to the Mass Storage Function Driver.

FIGURE 16: FIRMWARE LAYERS HANDLING USB REQUESTS IN THE MPLAB HARMONY USB DEVICE STACK MASS STORAGE CLASS

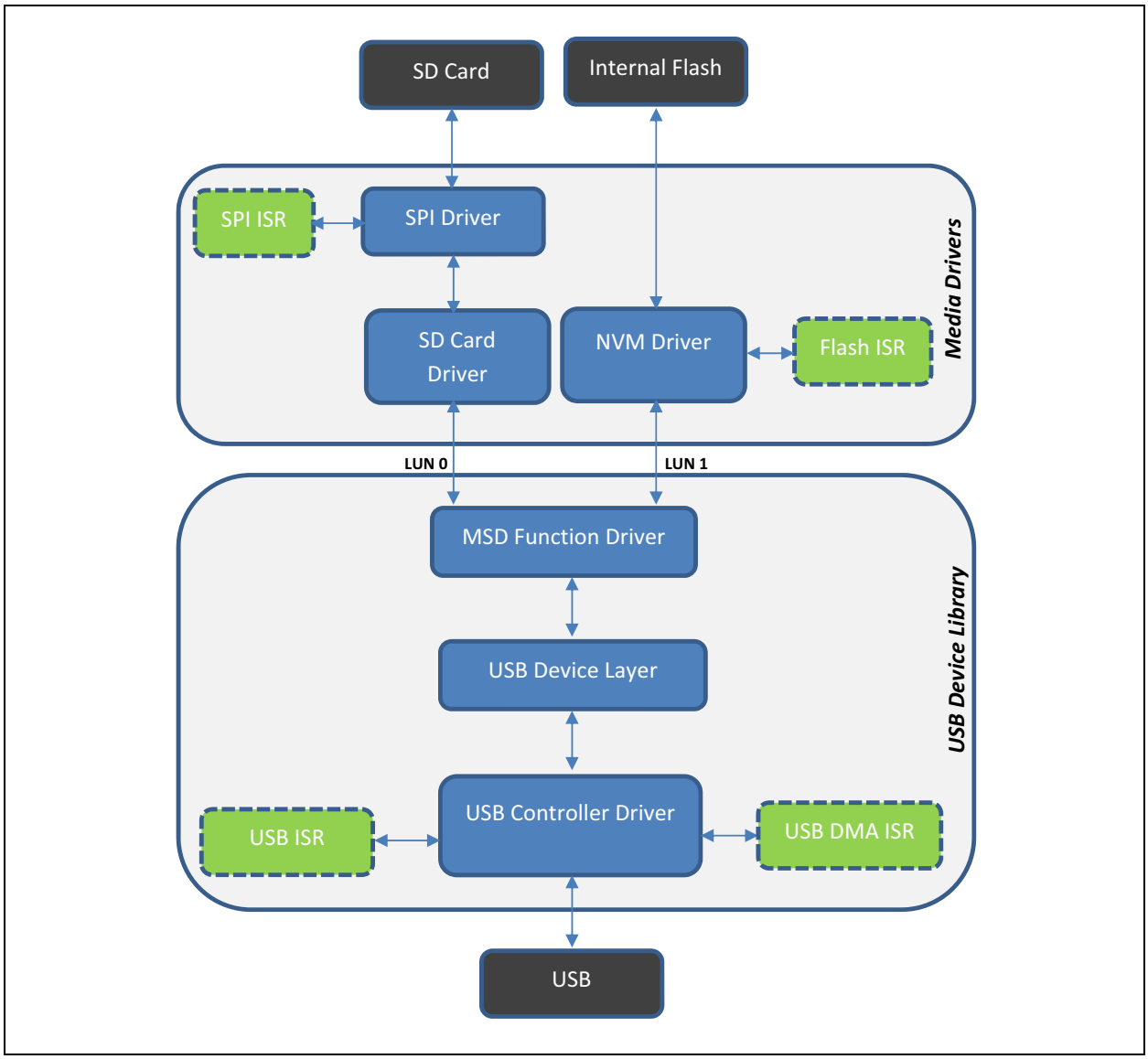


CONFIGURING THE USB MASS STORAGE DEVICE STACK WITH MPLAB HARMONY TO SUPPORT TWO LOGICAL UNITS

The section describes the configuration steps required to create a multi LUN MSD using MPLAB Harmony. The SD card will act as one logical unit and the PIC32MZ internal Flash memory as the second logical unit. These two will appear as two separate drives when connected to a USB Host PC.

Figure 17 shows the interaction between various drivers. For LUN 0 read/write requests, the MSD Function Driver reads/writes data from/to the SD Card using the SD Card driver. Similarly, LUN 1 read/write requests are serviced by the MSD Function Driver accessing the NVM media (Internal Flash) using the NVM driver. The SD Card Driver uses the SPI driver to interact with the SD card. The MHC generates the necessary code to bind the media drivers with the MSD Function Driver's media interface

FIGURE 17: INTERACTION BETWEEN VARIOUS DRIVERS



Hardware Requirements

- PIC32MZ EF Curiosity Development Board
- microSD Click board and SD card
- MPLAB REAL ICE Debugger (optional)
- USB Type-A to micro-B cables

Software Requirements

- MPLAB X IDE v3.40 or later
- MPLAB Harmony v2.02 or later

Using MPLAB Harmony Configurator (MHC)

Follow these steps to use MHC:

1. Create a new MPLAB Harmony project, and then configure the BSP and Clock.
2. Configure the USB Device Stack for Mass Storage functionality with two logical units, SD Card and NVM.
3. Configure the SD Card, SPI Driver, and SPI I/O pins.
4. Configure the NVM Driver.
5. Generate MHC code.
6. Add and modify application files.
7. Compile and run the code.

STEP1: CREATING THE MPLAB HARMONY PROJECT AND CONFIGURING THE BSP AND CLOCK.

1. In MPLAB X IDE, select *File > New Project* and then create a new 32-bit MPLAB Harmony Project. Specify the Project Location, Project Name, and then select the Target Device as PIC32MZ2048EFM100.
2. Click **Finish** to create the project (see Figure 18).
3. In MPLAB X IDE, select *Tools > Embedded* and then open **MPLAB Harmony Configurator**.
4. In the MHC tree view, expand the “BSP Configuration” and then select **USE BSP?**.
5. Select the PIC32MZ EF Curiosity Development Board. Selecting a BSP will automatically configure clock and board specific hardware (GPIOs, LEDs and Switches), as shown in Figure 19.

FIGURE 18: CREATE NEW PROJECT

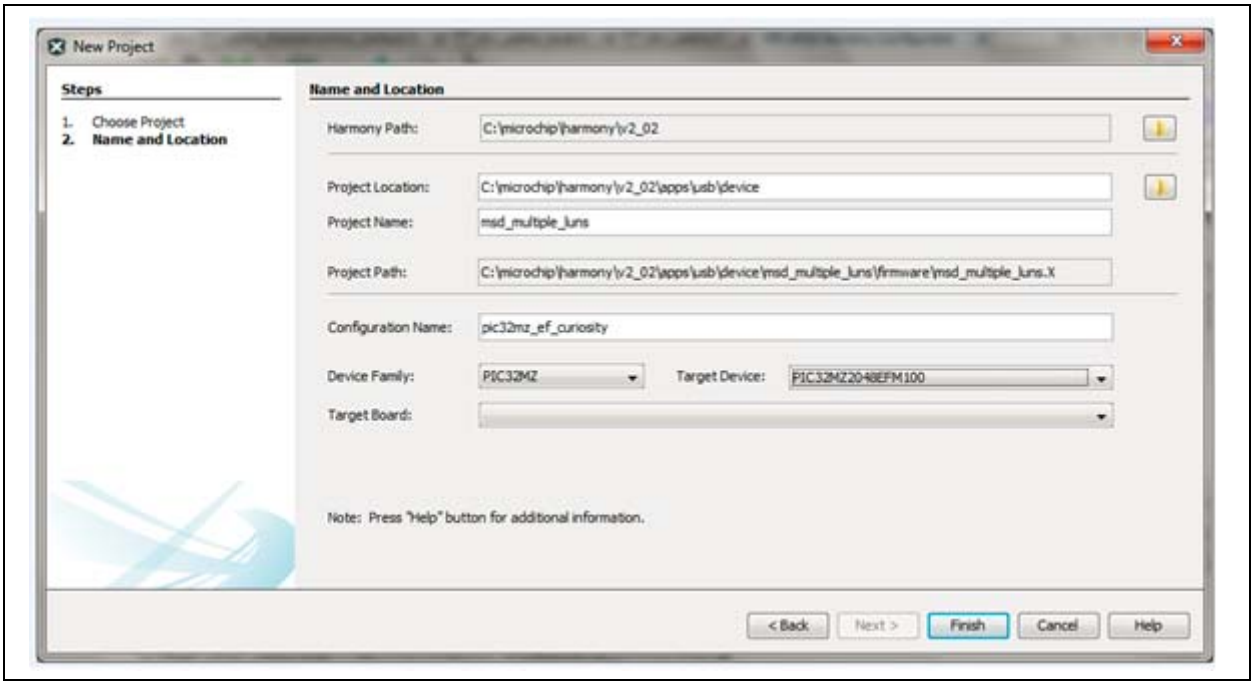
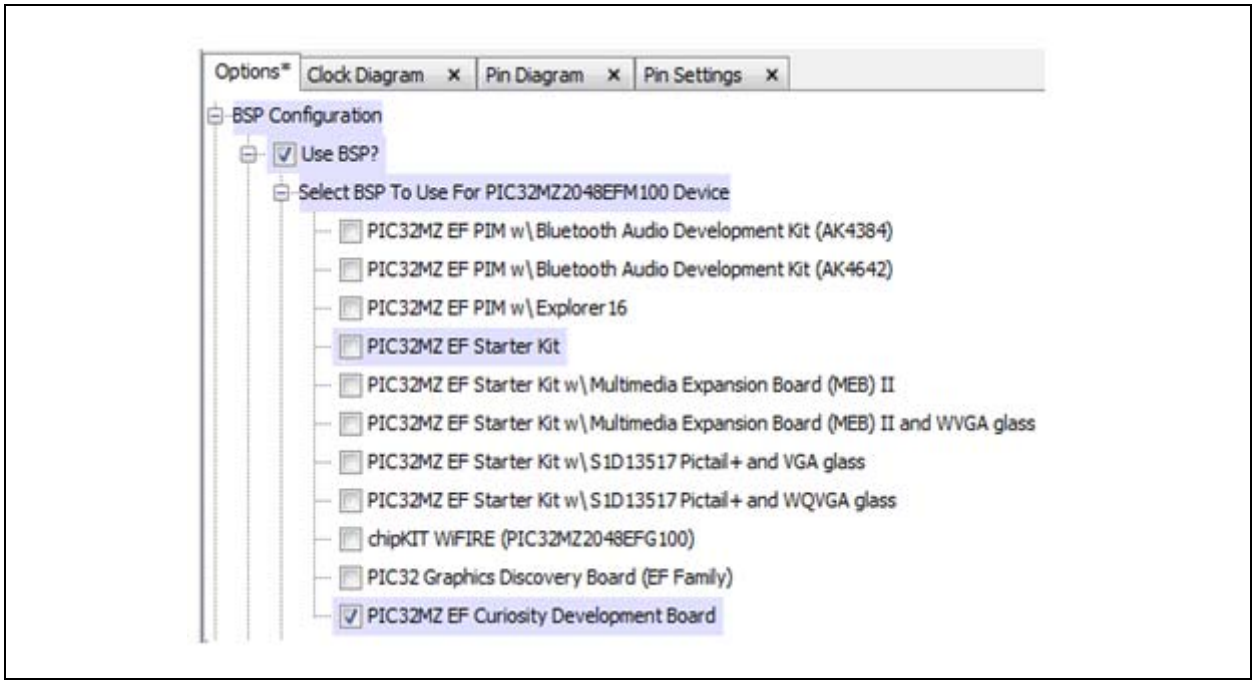
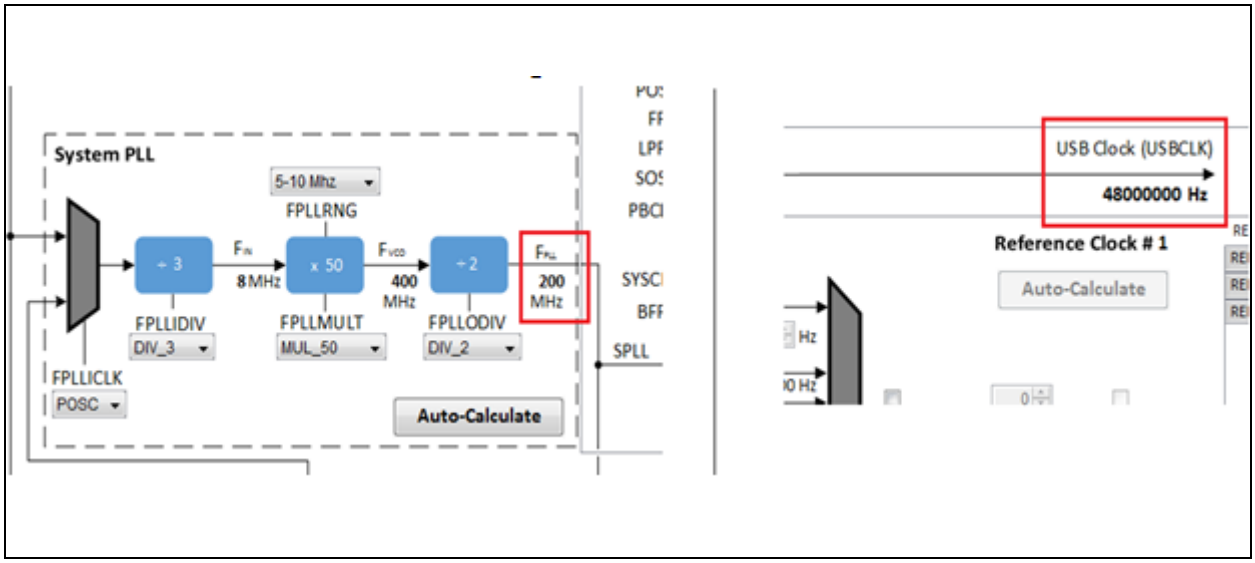


FIGURE 19: BSP SELECTION IN MHC



6. Click the Clock Diagram tab and verify that the System PLL output is set to 200 MHz and the USB Clock is set to 48 MHz, as shown in Figure 20.

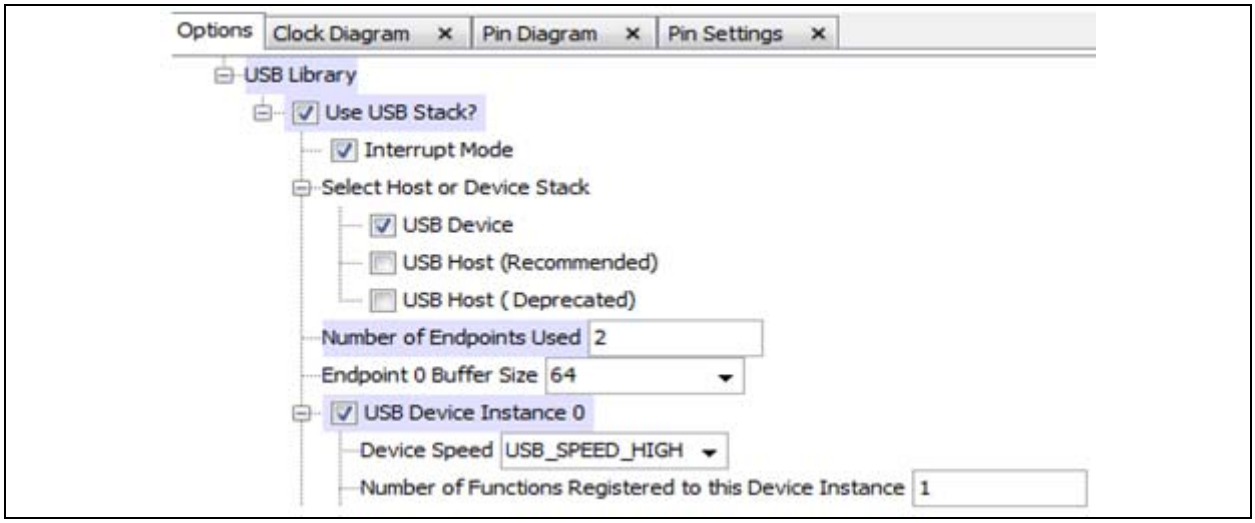
FIGURE 20: CLOCK CONFIGURATION IN MHC



STEP 2: CONFIGURING USB DEVICE STACK FOR MASS STORAGE FUNCTIONALITY WITH TWO LOGICAL UNITS -SD CARD AND NVM

1. Select *Harmony Framework Configuration > USB Library*.
2. Select **Use USB Stack?**. The Interrupt Mode is selected by default. This indicates that the USB driver state-machine will be run from interrupt context.
3. The Select Host or Device Stack is set to USB Device.
4. Change the number of Endpoints to 2. The USB MSD uses bulk-only transport (BOT) protocol. One endpoint is the control endpoint (EP0) used for control packet transfers. The second endpoint is the Bulk endpoint (Bulk IN, Bulk OUT) used for data transfers between the Host and device.
5. Retain the Endpoint 0 Buffer Size to 64. For High-Speed devices, the EP0 size is fixed to 64. For Full-Speed devices, the EP0 size can be 8, 16, 32, or 64 bytes.
6. The USB Device Instance 0 is selected as default. Each USB Device Instance represents a USB peripheral on the micro-controller. Since there is only one USB peripheral on PIC32MZ devices, there is only one USB Device Instance. user need to expand it.
7. Keep the device speed to default USB_SPEED_HIGH. PIC32MZ devices support both Full-Speed and High-Speed operation. Selecting High-Speed will allow the device to work at both Full-Speed and High-Speed.
8. The Number of Functions Registered to this Device Instance is set to 1. This indicates the number of USB Device class drivers registered with this Device instance. In this case, this will be the MSD Function Driver (see Figure 21).
9. Function1 is selected, expand it. Configure the Function Driver for USB MSD operation.
10. Select the Device Class to MSD.
11. Select the USB Configuration Value to which this Function Driver will be tied. The USB Device will set the active configuration to the configuration value received through the SET CONFIGURATION control command from the Host. The USB Device task will run through all the registered function drivers and will try to match the value of active configuration with the USB Configuration value for that Function Driver. When a match is found, the corresponding Function Driver's task is run. For this example, retain the default value of 1.
12. Retain the default value of 0 for Start Interface Number. This indicates that Interface number 0 is owned by this (MSD) Function Driver. This will result in standard and class specific control requests targeted to the interface, and class specific endpoint requests be forwarded to the Function Driver (MSD in this case) that manages the interface number 0.
13. The Speed member of the entry specifies the Device speeds for which this Function Driver should be initialized. This can be set to USB_SPEED_FULL, USB_SPEED_HIGH or a logical OR combination of both. The Device Layer will initialize the function if the devices' attach speed matches the speed mentioned in the Speed member of the entry. To allow for both High-Speed and Full-Speed operation, set it to USB_SPEED_HIGH|USB_SPEED_FULL.

FIGURE 21: USB STACK CONFIGURATION IN MHC



14. Set the Endpoint Number to 1. This indicates that Endpoint number 1 will be used for Bulk In and Bulk Out transfers. The value selected for Endpoint Number will be reflected in the Endpoint address field of the Endpoint descriptors. And based on the Endpoint address in the endpoint descriptors, MSD Function Driver will initialize the corresponding endpoints for Bulk-In and Bulk-Out transfers.
15. Set a value of 1 for Max number of sectors to buffer. This will set aside a buffer of size 512 x 1 bytes. This value may be changed to allow buffering of data read from the media as explained in the SCSI_READ_10 command. Buffering of media data minimizes the number of NAK sent to the USB Host in response to the IN data request and thereby increases the overall throughput, at the expense of increased RAM usage.
16. Set a value of 2 for Number of Logical Units, SD card and internal Flash (NVM).
17. Expand LUN 0 and select Media Type as SDCARD. Selecting SDCARD will cause MHC to automatically enable the SD card and SPI drivers.
18. Expand LUN 1 and select Media Type as NVM. Selecting NVM will cause MHC to automatically enable the NVM driver, see Figure 22.
19. Retain default value for Vendor ID, Product ID, Manufacturer String and Product String.
20. Retain the default Priority and Sub-priority values for USB Interrupt and USB DMA interrupts, see Figure 23.
21. Keep the Power State to SYS_MODULE_POWER_RUN_FULL, and retain the other values unchecked.

FIGURE 22: USB STACK CONFIGURATION IN MHC (CONTINUED)

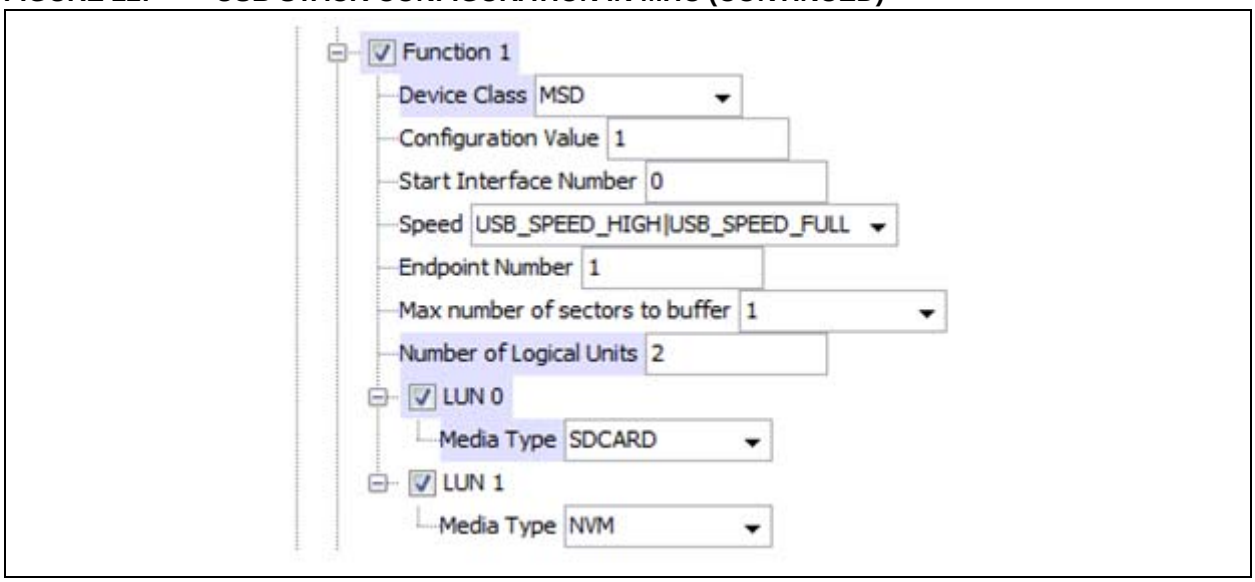
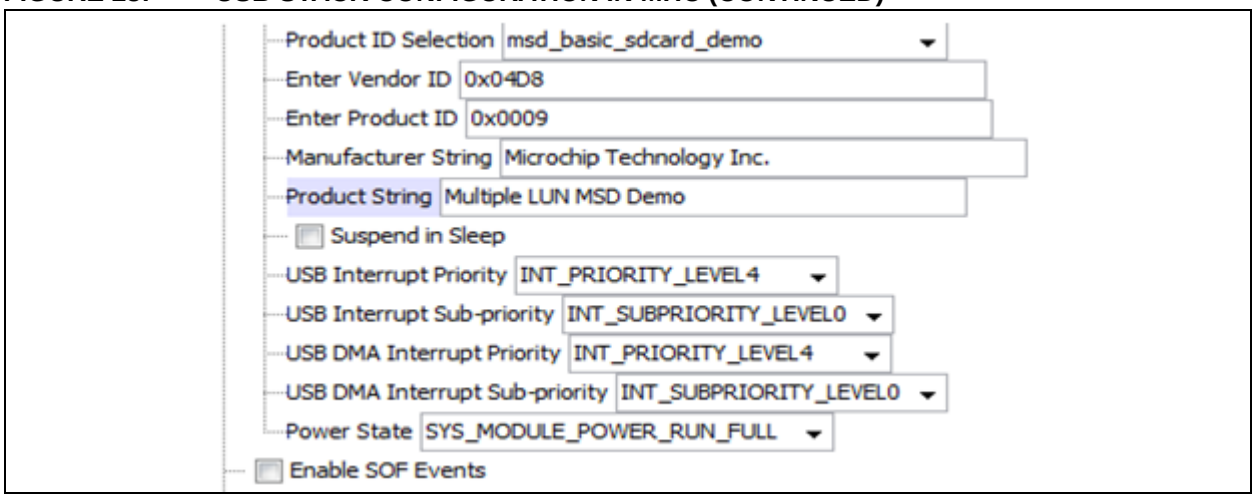


FIGURE 23: USB STACK CONFIGURATION IN MHC (CONTINUED)



STEP 3: CONFIGURING SD CARD, SPI DRIVER, AND SPI I/O PINS

1. Select *Harmony Framework Configuration > Drivers > SD Card*.
2. The "**Use SD Card Driver?**" option is already selected as the LUN0 media type is selected as SDCARD.
3. For "Clock To Use" choose CLK_BUS_PERIPHERAL_2, and then clear "Enable Write Protect Check?".
4. For "Chip Select Port" choose PORT_CHANNEL_D and for "Chip Select Port Bit" select PORTS_BIT_POS_5.
5. Clear "Register with File System?". The SD card memory will be accessed by the USB mass storage driver, see Figure 24.
6. Select *Harmony Framework Configuration > Drivers > SPI*.
7. The "Use SPI Driver?" option is already selected, as the SD card is configured to use SPI driver instance 0. The SPI driver is configured for 8-bit, Enhanced Buffer mode and Master mode, see Figure 25.
8. Expand SPI Driver Instance 0. Notice that driver is configured for Interrupt Mode operation. Change SPI Module ID to SPI_ID_2, since the SD Click board is connected to SPI peripheral 2 on the PIC32 MZ Curiosity Development Board, see Figure 26.
9. Change the Clock Mode to DRV_SPI_CLOCK_MODE_IDLE_LOW_EDGE_FALL and Clock Phase to SPI_INPUT_SAMPLING_PHASE_AT_END, see Figure 27.
10. To configure the SPI I/O lines, select *MPLAB Harmony Configurator > Pin Table*.
11. Map SCK2 to pin 10, SDI2 to pin 88, and SDO2 to pin 11, as shown Figure 28.

FIGURE 24: SD CARD DRIVER MHC CONFIGURATION

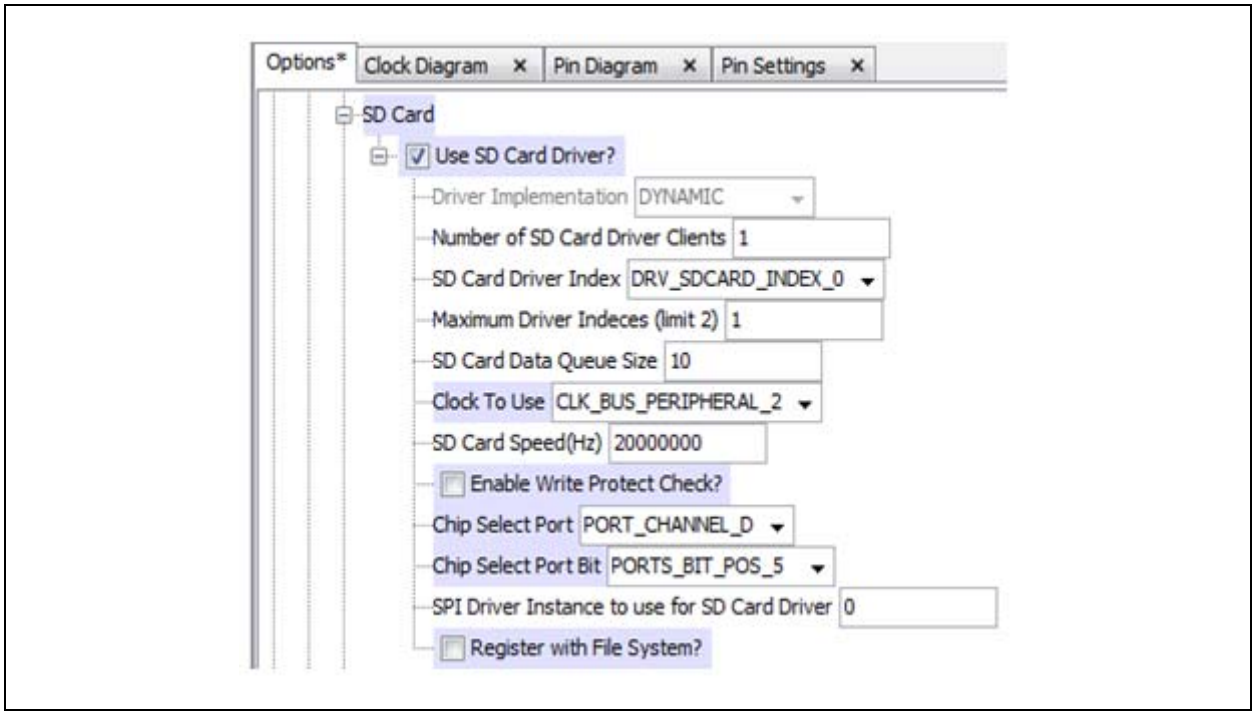


FIGURE 25: SPI DRIVER MHC CONFIGURATION

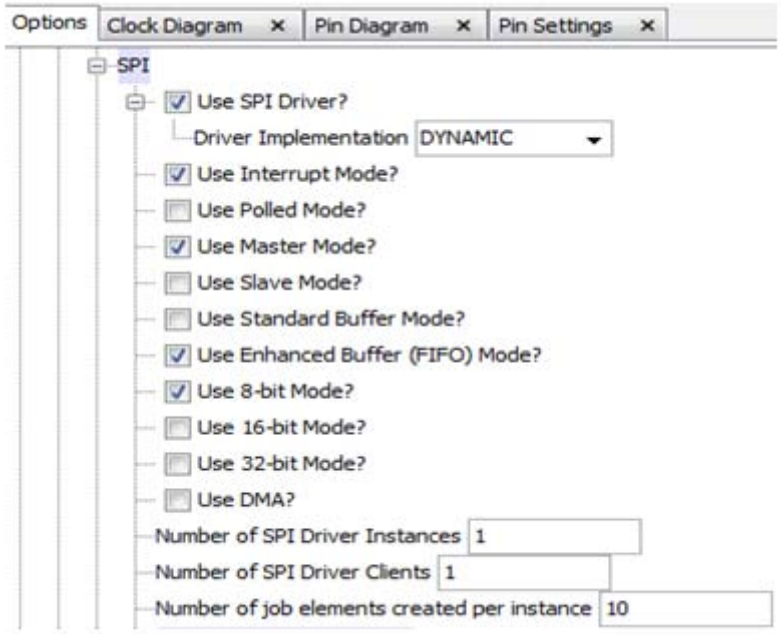


FIGURE 26: SPI DRIVER MHC CONFIGURATION (CONTINUED)

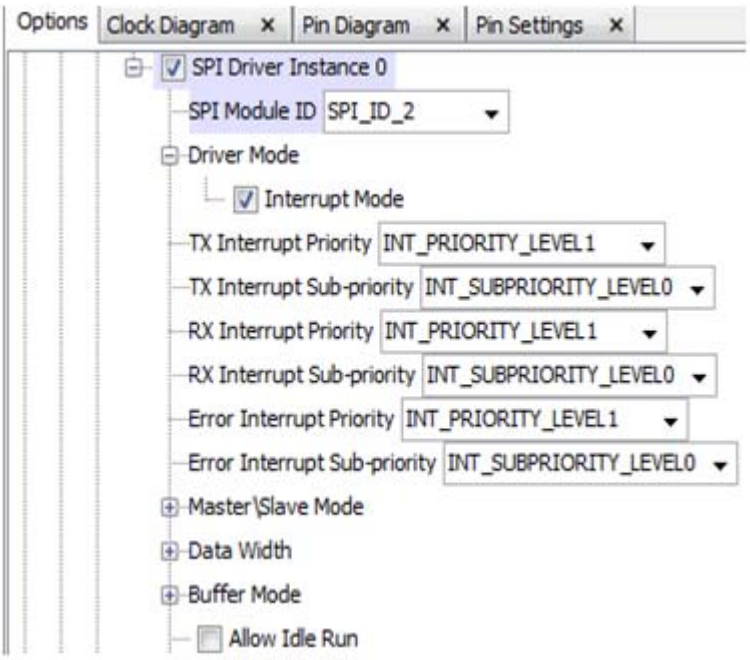


FIGURE 27: SPI DRIVER MHC CONFIGURATION (CONTINUED)

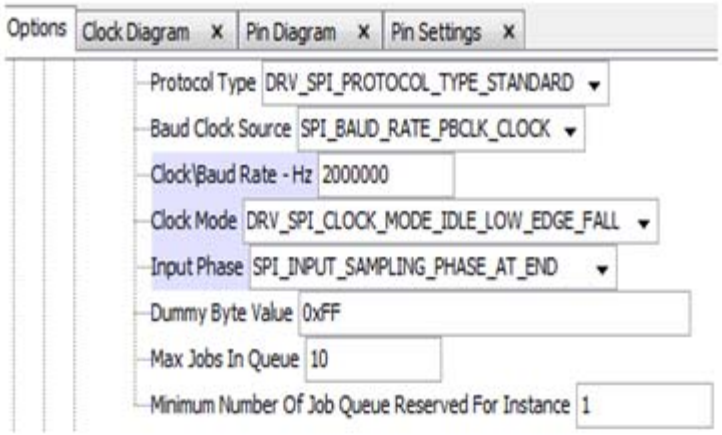
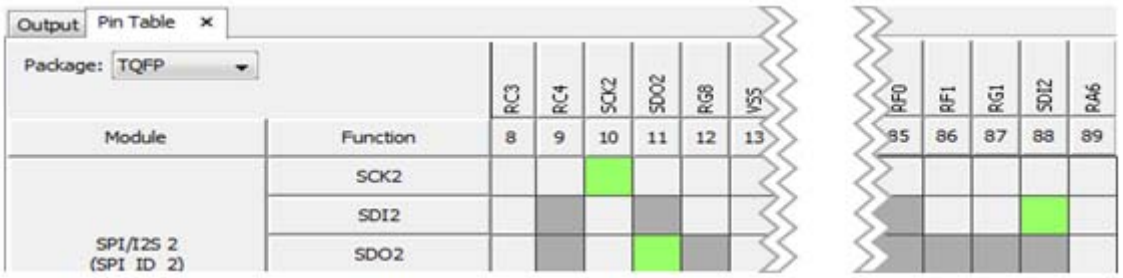


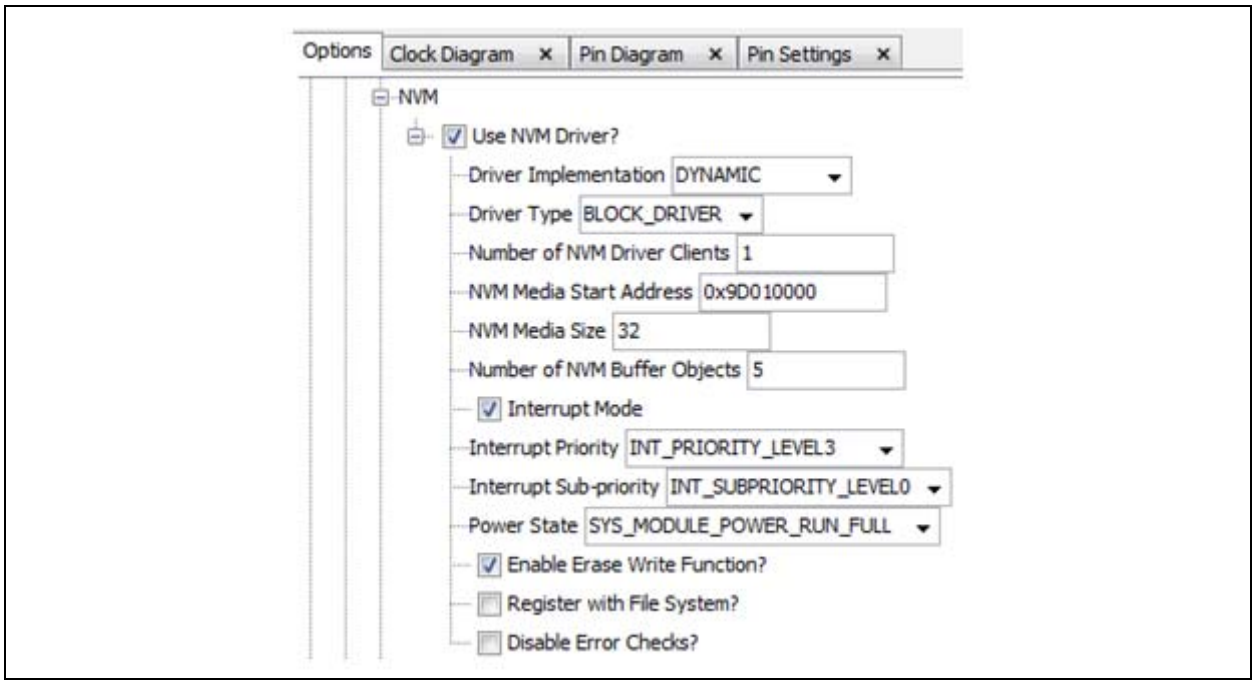
FIGURE 28: SPI I/O LINE CONFIGURATION



STEP 4: CONFIGURING NVM DRIVER

1. Select *Harmony Framework Configuration > Drivers > NVM*.
2. The “Use NVM Driver?” is already selected, as the media for LUN 1 is configured as NVM. The NVM driver is configured for dynamic and interrupt mode of operation.
3. For “NVM Media Start Address” enter 0x9D010000 and for “NVM Media Size” enter 32 (KB). The FAT12 file system on the NVM media will be mounted to the address pointed by NVM Media Start Address.
4. Select **Enable Erase Write Function?** to enable the row erase write feature, see Figure 29.

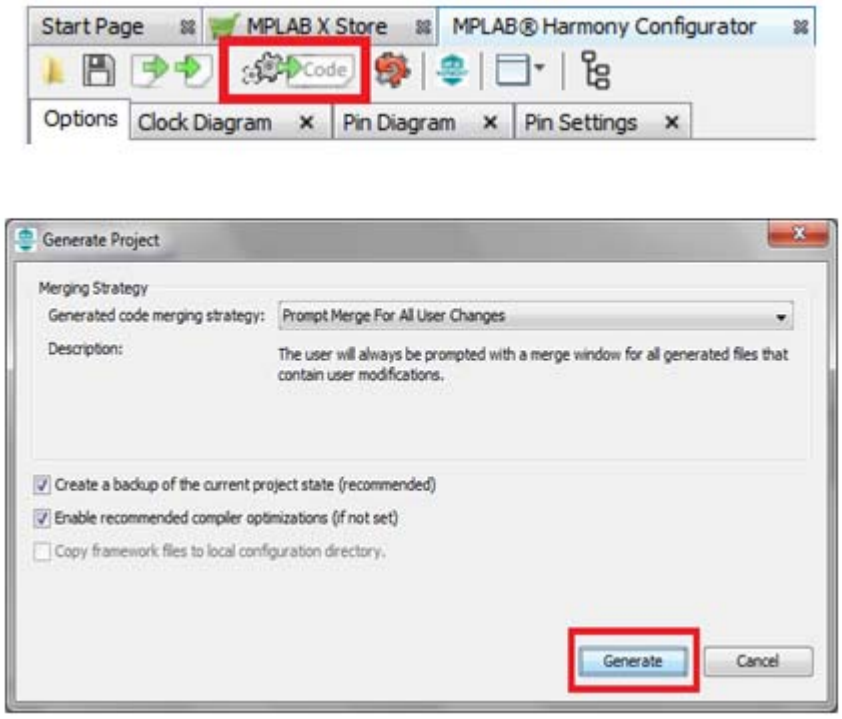
FIGURE 29: NVM DRIVER CONFIGURATION



STEP 5: GENERATING MHC CODE

Save the MHC configuration and then click **Generate** to generate the code, see Figure 30.

FIGURE 30: GENERATE CODE



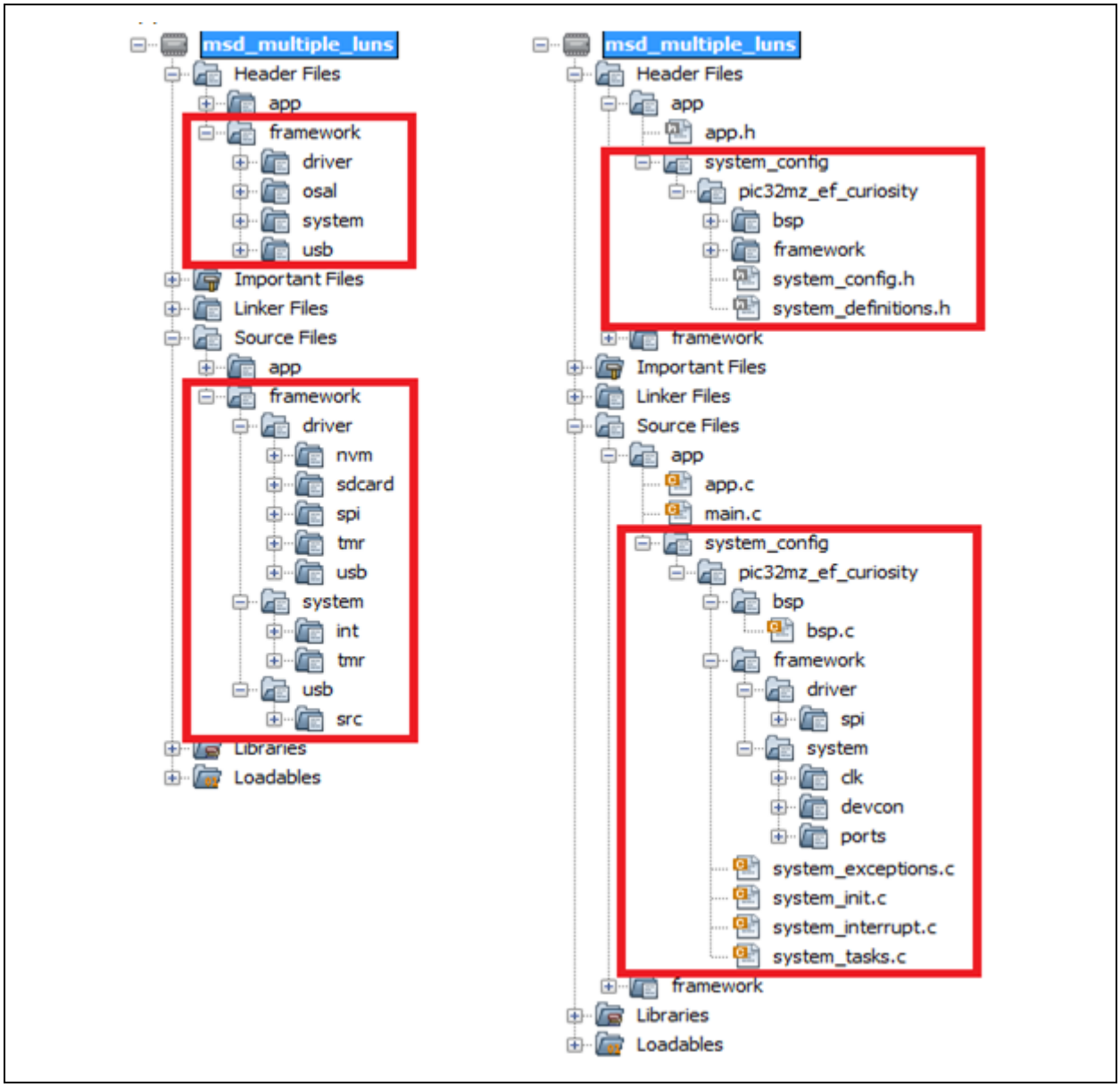
MHC will generate the code based on the MHC selections.

Make sure the Merging Strategy option is set to Prompt Merge For All User Changes. Since there are no user changes, there should not be any merge conflicts.

The framework folder under the Source Files folder contains the standard MPLAB Harmony Framework files that are added to the project by MHC (based on the MHC selections), see Figure 31. The `nvm`, `sdcard`, `spi`, `tmr` and `usb` driver folders are added to the `driver` folder. The system folder contains the drivers used by the system services. The `usb` folder contains the related USB Device Stack source files.

The framework folder under the `app` folder contains the customized MPLAB Harmony Framework files. These files are generated by MHC in response to the specific MHC selections. The `system_init.c` contains the initialization data for various drivers and the system initialization function, `SYS_Initialize`. The `SYS_Initialize` function performs system initialization by initializing the Clock system, BSP, I/Os, System services, Drivers, and Interrupts. The `APP_Initialize` function is then called to allow application initialization.

FIGURE 31: PROJECT FILES AND FOLDERS STRUCTURE



The system_interrupt.c contains the interrupt handlers for the configured drivers. The `system_tasks.c` contains the `SYS_Tasks` function that runs the state machines for various drivers and the USB stack. The `system_config.h` file contains the driver configuration definitions based on the MHC selections.

STEP 6: ADDING AND MODIFYING THE APPLICATION FILES

Registering Application Event Handler to Receive USB Device Layer Events

The `app.c` file is generated by the MHC which provides a template for initializing the application under `APP_Initialize` and running the application tasks under `APP_Tasks`. The application must register an event handler with the USB Device Layer to receive the Device Layer events. A call to `USB_DEVICE_Open` returns an handler to the specific instance of the USB Device Layer. This handler is then used to register the application event handler with the USB Device Layer by a call to the `USB_DEVICE_EventHandlerSet` function, as shown in Example 1.

The registered event handler will be called by the USB Device Layer to notify Device Layer events, such as USB Device reset, USB Device configured, USB Device power detected and so on. Under the USB Device power detected event, the application must call the `USB_DEVICE_Attach` function to attach the Device to the USB as shown in Example 2.

EXAMPLE 1: OPENING THE USB DEVICE LAYER AND REGISTERING AN EVENT HANDLER

```
void APP_Tasks ( void )
{

    /* Check the application's current state. */
    switch ( appData.state )
    {
        /* Application's initial state. */
        case APP_STATE_INIT:
        {
            appData.usbDevHandle = USB_DEVICE_Open(USB_DEVICE_INDEX_0, DRV_IO_INTENT_READWRITE);

            if(appData.usbDevHandle != USB_DEVICE_HANDLE_INVALID)
            {
                /* Set the Event Handler. We will start receiving events after
                 * the handler is set */
                USB_DEVICE_EventHandlerSet(appData.usbDevHandle, APP_USBDeviceEventHandler,
                                           (uintptr_t)&appData);

                /* Move the application to the next state */
                appData.state = APP_STATE_SERVICE_TASKS;
            }
```

EXAMPLE 2: HANDLING USB DEVICE LAYER EVENTS

```
void APP_USBDeviceEventHandler( USB_DEVICE_EVENT event, void * pEventData, uintptr_t context )
{
    /* This is an example of how the context parameter
       in the event handler can be used.*/

    APP_DATA* appData = (APP_DATA*)context;

    switch( event )
    {
        case USB_DEVICE_EVENT_RESET:
        case USB_DEVICE_EVENT_DECONFIGURED:

            /* Device was reset or deconfigured. Update LED status */
            BSP_LEDOn ( BSP_LED_1 );
            BSP_LEDOn ( BSP_LED_2 );
            BSP_LEDOn ( BSP_LED_3 );
            break;

-------------------------------------------------------------------------------

    case USB_DEVICE_EVENT_POWER_DETECTED:

            /* VBUS is detected. Attach the device. */
            USB_DEVICE_Attach(appData->usbDevHandle);
            break;

        case USB_DEVICE_EVENT_POWER_REMOVED:

            /* VBUS is not detected. Detach the device */
            USB_DEVICE_Detach(appData->usbDevHandle);
            break;

        /* These events are not used in this demo */
        case USB_DEVICE_EVENT_RESUMED:
        case USB_DEVICE_EVENT_ERROR:
        case USB_DEVICE_EVENT_SOF:
        default:
            break;
    }
}
```

Formating the NVM Memory Region to FAT12 File System

The NVM memory region must be formatted with a file system to enable the USB Host computer to understand the layout of files on the media. The FAT12 is a lightweight file system with very low overhead. In addition, the FAT12 file system can address volumes of size up to 12 MB which is sufficient for the NVM media partition of 32 Kbytes size. The `diskImage.c` file contains the FAT12 image for the NVM memory region. The FAT12 image formats the media to contain a `FILE.TXT` file containing the string “Data”. The `diskImage.c` file is not generated by the MHC, and it can be copied from the msd_multiple_luns demonstration, which is available in the `harmony-install-dir/apps/usb/device/msd_multiple_luns` folder.

Right click the *Source Files > app > Add Existing Item…* and then add the `diskImage.c` file to the project.

The complete project source code is available with the MPLAB Harmony version 2.02 (and later versions) installer. The project can be found in the MPLAB Harmony installer, by navigating to the `harmony-install-dir/apps/usb/device/msd_multiple_luns` folder.

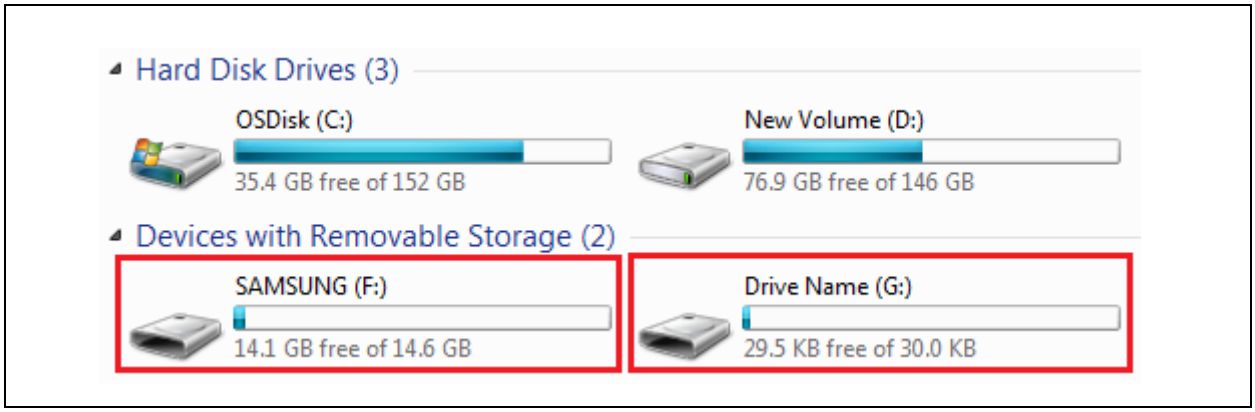
STEP 8: COMPILING AND RUNNING THE SOURCE CODE

After adding the necessary source code, build and program the PIC32MZ Curiosity Development Board.

Insert the microSD click board (with SD Card inserted) on the mikro bus header 2 (J10). Power the board using the USB Debug connector J3. Connect a USB micro cable from J12 on the USB device to a USB Host PC.

Wait for the PC to enumerate the USB device as a mass storage device. After enumeration is complete, both the SD Card and the NVM should appear as two separate logical drives on the Host PC, as shown in Figure 32.

FIGURE 32: LOGICAL DRIVES ON THE HOST PC



USB STACK INITIALIZATION DATA

The `system_init.c` file contains the initialization data for the USB stack.

USB Device Layer Initialization Data

Example 3 shows the USB Device Layer initialization data.

The `USB_DEVICE_INIT` structure contains the USB Device Layer initialization data. The `moduleInit` allows initialization of driver into the requested power mode. The `registeredFuncCount` indicates the number of function drivers registered to this instance of USB Device Layer. This application will have only one function drives, MSD Function Driver. The `registeredFunctions` points to the Function Driver Table for this instance of Device Layer. The Function Driver Table contains initialization data for the MSD Function Driver. The `usbMasterDescriptor` points to the USB Descriptor structure which in turn points to Device Descriptor, Configuration Descriptor, and string descriptors. The `usbDriverInterface` points to the USB driver functions. These functions provide the USB Device Layer with a structured access to the USB bus.

EXAMPLE 3: USB DEVICE LAYER INITIALIZATION DATA

```
const USB_DEVICE_INIT usbDevInitData =
{
    /* System module initialization */
    .moduleInit = {SYS_MODULE_POWER_RUN_FULL},

    /* Number of function drivers registered to this instance of the
       USB device layer */
    .registeredFuncCount = 1,

    /* Function driver table registered to this instance of the USB device layer*/
    .registeredFunctions = (USB_DEVICE_FUNCTION_REGISTRATION_TABLE*)funcRegistrationTable,

    /* Pointer to USB Descriptor structure */
    .usbMasterDescriptor = (USB_DEVICE_MASTER_DESCRIPTOR*)&usbMasterDescriptor,

    /* USB Device Speed */
    .deviceSpeed = USB_SPEED_HIGH,

    /* Index of the USB Driver to be used by this Device Layer Instance */
    .driverIndex = DRV_USBHS_INDEX_0,

    /* Pointer to the USB Driver Functions. */
    .usbDriverInterface = DRV_USBHS_DEVICE_INTERFACE,

};
```

USB Device Function Registration Table

Example 4 shows the instance of USB Device Function Registration Table.

The `funcRegistrationTable` contains a list of function drivers that allows registration of function drivers with the USB Device Layer. The USB Device Layer initialization data holds a pointer to the `funcRegistrationTable`.

The `configurationValue`, `interfaceNumber`, speed, `numberOfInterfaces` are all based on the MHC selections. The `funcDriverIndex` is used as an index into the MSD Function Driver instance data structure - `gUSBDeviceMSDInstance`. Since there is only one instance of USB MSD Function Driver, the `funcDriverIndex` is set to '0'.

Note: Although there are two media (SD card and NVM), only one instance of MSD Function Driver is required. The two media can be accessed based on the LUN number passed in the CBW packet by the USB Host.

The `driver` points to the interface exposed by the MSD Function Driver to the USB Device Layer. The USB Device Layer calls these interface functions at the time of an appropriate event. Example 4 shows the Function Driver interface as expected by the USB Device Layer.

Finally, the `funcDriverInit` points to the Function Driver initialization data, which can be used by the MSD Function Driver to initialize itself when the USB Device layer receives Set Configuration Control command from the USB Host.

EXAMPLE 4: USB DEVICE FUNCTION REGISTRATION TABLE

```
const USB_DEVICE_FUNCTION_REGISTRATION_TABLE funcRegistrationTable[1] =
{
    /* Function 1 */
    {
        .configurationValue = 1,    /* Configuration value */
        .interfaceNumber = 0,       /* First interfaceNumber of this function */
        .speed = USB_SPEED_HIGH|USB_SPEED_FULL,    /* Function Speed */
        .numberOfInterfaces = 1,    /* Number of interfaces */
        .funcDriverIndex = 0,  /* Index of MSD Function Driver */
        .driver = (void*)USB_DEVICE_MSD_FUNCTION_DRIVER,    /* USB MSD function data exposed
                                                               to device layer */
        .funcDriverInit = (void*)&msdInit0,   /* Function driver init data */
    },
};
```

USB Device Function Driver Interface

Example 5 shows the USB Function Driver Interface structure. The USB Device Layer calls the `intilizeByDescriptor` callback when the Set Configuration control request is received from the USB Host. The MSD Function Driver must initialize itself and enable the bulk endpoints. The `pDescriptor` argument will point to the configuration, interface, and Endpoint Descriptor corresponding to the configuration value received in the Set Configuration request. The Function Driver will initialize the bulk endpoints based on the interface and endpoint descriptors pointed by `pDescriptor`.

The `deInitialize` callback is called when the Device layer detects a Device detach, change in configuration, or for USB bus reset. The `controlTransferNotification` is called to allow the Function Driver to handle Standard and Class specific control requests targeted to MSD interface. The Standard endpoint requests targeted to Bulk Endpoints are handled by the Device Layer.

The USB Device Layer calls the MSD Function Driver task function pointed by the `tasks` if the current Device speed and current configuration matches with the speed and configuration values mentioned in the Function Driver Registration Table, and the Function Driver is in a configured state. Example 6 shows an instance of the MSD Function Driver.

EXAMPLE 5: USB FUNCTION DRIVER INTERFACE STRUCTURE

```
typedef struct
{
    /* Initialize gets called by the Device layer when it recieves set
       configuration event. The device layer will initialize a function driver
       for every descriptor. Based on the descriptor type the function driver
       has to initialize itself. */

    void (*initializeByDescriptor)
    (
        SYS_MODULE_INDEX funcDriverIndex,
        USB_DEVICE_HANDLE usbDeviceHandle,
        void* funcDriverInit,
        uint8_t interfaceNumber,
        uint8_t alternateSetting,
        uint8_t descriptorType,
        uint8_t * pDescriptor
    );

    /* Deinitialize gets called when the device layer detects a device dettach,
       change in configuration or ob USB bus reset.*/

    void (*deInitialize)(SYS_MODULE_INDEX funcDriverIndex);

    /* This function will be called by the device layer when there is a interface specific
       setup packet request */

    void (*controlTransferNotification)
    (
        SYS_MODULE_INDEX index,
        USB_DEVICE_EVENT controlEvent,
        USB_SETUP_PACKET * controlEventData
    );

    /* Function driver Tasks */
    void (*tasks)(SYS_MODULE_INDEX funcDriverIndex);

    /* Gloabl Initialize for function driver */
    void (*globalInitialize)(void);

} USB_DEVICE_FUNCTION_DRIVER;
```

EXAMPLE 6: MSD FUNCTION DRIVER INSTANCE

```
USB_DEVICE_FUNCTION_DRIVER msdFunctionDriver =
{
    /* MSD init function */
    .initializeByDescriptor = _USB_DEVICE_MSD_InitializeByDescriptorType ,

    /* MSD deinit function */
    .deInitialize = _USB_DEVICE_MSD_Deinitialization ,

    /* MSD set-up packet handler */
    .controlTransferNotification = _USB_DEVICE_MSD_ControlTransferHandler ,

    /* MSD tasks function */
    .tasks = _USB_DEVICE_MSD_Tasks
};
```

USB Device Function Driver Initialization Data

Example 7 shows the initialization data for the MSD Function Driver.

The USB_DEVICE_MSD_INIT contains the initialization data for the MSD Function Driver:

- `numberOfLogicalUnits` - This indicates the number of logical units in this instance of MSD Function Driver and is set to two.
- `msdCBW` - points to the CBW data structure. The pointer is used by the MSD Function Driver to receive CBW from the Host. For a PIC32MZ device with a Data Cache, such as the PIC32MZ used in this application, this array should be placed in coherent memory and should be aligned on a 16-byte boundary.
- `msdCSW` - points to the CSW data structure. The pointer is used by the MSD Function Driver to send CSW to the Host. For a PIC32MZ device, this array should be placed in coherent memory and should be aligned on a 16-byte boundary.
- `mediaInit` - points to the MSD media driver initialization data structure. There should be one structure for each LUN.

EXAMPLE 7: MSD FUNCTION DRIVER INITIALIZATION DATA

```
const USB_DEVICE_MSD_INIT msdInit0 =
{
    .numberOfLogicalUnits = 2,
    .msdCBW = &msdCBW0,
    .msdCSW = &msdCSW0,
    .mediaInit = &msdMediaInit0[0]
};
```

Media Driver Initialization Data

Example 8 shows the structure of the MSD media driver initialization data.

EXAMPLE 8: MSD MEDIA DRIVER INITIALIZATION DATA STRUCTURE

```
typedef struct
{
    /* Instance index of the media driver to open for this LUN */
    SYS_MODULE_INDEX instanceIndex;

    /* Sector size for this LUN. If 0, means that sector size will be available
       from media geometry. */
    uint32_t sectorSize;

    /* Pointer to a byte buffer whose size is the size of the sector on this
     * media. In case of a PIC32MZ device, this buffer should be coherent and
     * should be aligned on a 16 byte boundary */
    uint8_t * sectorBuffer;

    /* In a case where the sector size of this media is less than the size of
     * the write block, a byte buffer of write block size should be provided to
     * the function driver. For example, the PIC32MZ NVM flash driver has a
     * flash program memory row size of 4096 bytes which is more than the
     * standard 512 byte sector. In such a case the application should set this
     * pointer to 4096 byte buffer */
    uint8_t * blockBuffer;

    /* Block 0 Start Address on this media. If non zero, then this address will
       be passed to blockStartAddressSet function. This should be set to start
       of the storage address on the media. */
    void * block0StartAddress;

    /* Pointer to SCSI inquiry response for this LUN */
    SCSI_INQUIRY_RESPONSE inquiryResponse;

    /* Function pointers to the media driver functions */
    USB_DEVICE_MSD_MEDIA_FUNCTIONS mediaFunctions;

} USB_DEVICE_MSD_MEDIA_INIT_DATA;
```

The media initialization data consists the following:

- `instanceIndex` - points to the index of the media driver opened for this LUN. For LUN0 this is set to DRV_SDCARD_INDEX_0, which is an index into the SD card drivers and for LUN1 this is set to DRV_NVM_INDEX_0 which is an index into the NVM drivers
- `sectorSize` - This is the size of one sector on this media and is set to 512 for both the media. This is because from the Host point of view the media is organized in logical blocks of 512 bytes.
- `sectorBuffer` - points to a buffer whose size is 512 x `Max number of sectors` to buffer to allow buffering of data during a media read operation. In MHC, `Max number of sectors` to buffer was set to 1 in the USB Function Driver configuration.
- `blockBuffer` - This buffer is used by the MSD media driver to perform a read-modify-write operation for media whose block size is greater than the sector size (512 bytes). It is set to 0 (NULL) for LUN0. This is because the block size for SD card is 512 bytes. For LUN1, the NVM write block (Row) size is 2048 bytes and hence this is set to a buffer of size 2048 bytes. For NVM, a complete block of 2048 bytes will be read and modified in this buffer before the modified block is written back to the media.
- `inquiryResponse` - points to the SCSI inquiry response for the LUN.

`mediaFunctions` points to the media driver functions for this LUN. The MSD Function Driver expects the media driver to comply with the USB_DEVICE_MSD_MEDIA_FUNCTIONS interface. The USB_DEVICE_MSD_MEDIA_FUNCTIONS structure contains function pointers to media driver functions, such as `isAttached`, `open`, `close`, `geometryGet`, `blockRead`, `blockWrite`, `isWriteProtected`, and `blockEventHandlerSet`. For LUN0, the `mediaFunctions` point to SD card driver functions and for LUN1 it points to NVM driver functions:

- `isAttached` – The MSD Function Driver calls this function when it needs to know if the media is attached and ready to use.
- `open` – The MSD Function Driver calls this function to obtain a handle and gain access to the functionality of the specified instance of the media driver.
- `close` – The MSD Function Driver calls this function when the function driver gets deinitialized as a result of device detach or a change in configuration.
- `geometryGet` – The MSD Function Driver calls this function when it needs to know the storage capacity of the media.
- `blockRead` – The MSD Function Driver calls this function to read a block of data from the media.
- `blockWrite` – The MSD Function Driver calls this function to write a block of data to the media.
- `isWriteProtected` - The MSD Function Driver calls this function to find out if the media is write protected or not.
- `blockEventHandlerSet` – The MSD Function Driver calls this function to register a block event callback function with the media driver. This event callback will be called when a block related operation has completed.

Example 9 shows the media initialization data for both of the logical units.

EXAMPLE 9: MEDIA INITIALIZATION DATA

```
USB_DEVICE_MSD_MEDIA_INIT_DATA msdMediaInit0[2] =
{
        {
        DRV_SDCARD_INDEX_0,
        512,
        sectorBuffer,
        NULL,
        0,
        {
            0x00,// peripheral device is connected, direct access block device
            0x80,   // removable
            0x04,// version = 00=> does not conform to any standard, 4=> SPC-2
            0x02,// response is in format specified by SPC-2
            0x1F,// additional length
            0x00,// sccs etc.
            0x00,// bque=1 and cmdque=0,indicates simple queuing 00 is obsolete,
                    // but as in case of other device, we are just using 00
            0x00,// 00 obsolete, 0x80 for basic task queuing
            {
                'M','i','c','r','o','c','h','p'
            },
            {
                'M','a','s','s',' ','S','t','o','r','a','g','e',' ',' ',' ',' '
            },
            {
                '0','0','0','1'
            }
        },
        {
            DRV_SDCARD_IsAttached,
            DRV_SDCARD_Open,
            DRV_SDCARD_Close,
            DRV_SDCARD_GeometryGet,
            DRV_SDCARD_Read,
            DRV_SDCARD_Write,
            DRV_SDCARD_IsWriteProtected,
            DRV_SDCARD_EventHandlerSet,
            NULL
        }
    },
```

EXAMPLE 9: MEDIA INITIALIZATION DATA (CONTINUED)

```
    {
        DRV_NVM_INDEX_0,
        512,
        sectorBuffer,
        flashRowBackupBuffer,
        (void *)diskImage,
        {
            0x00,// peripheral device is connected, direct access block device
            0x80,   // removable
            0x04,// version = 00=> does not conform to any standard, 4=> SPC-2
            0x02,// response is in format specified by SPC-2
            0x1F,// additional length
            0x00,// sccs etc.
            0x00,// bque=1 and cmdque=0,indicates simple queueing 00 is obsolete,
                    // but as in case of other device, we are just using 00
            0x00,// 00 obsolete, 0x80 for basic task queueing
            {
                'M','i','c','r','o','c','h','p'
            },
            {
                'M','a','s','s',' ','S','t','o','r','a','g','e',' ',' ',' ',' '
            },
            {
                '0','0','0','1'
            }
        },
        {
            DRV_NVM_IsAttached,
            DRV_NVM_Open,
            DRV_NVM_Close,
            DRV_NVM_GeometryGet,
            DRV_NVM_Read,
            DRV_NVM_EraseWrite,
            DRV_NVM_IsWriteProtected,
            DRV_NVM_EventHandlerSet,
            NULL
        }
    },
};
```

USB Driver Initialization Data

Example 10 shows the initialization data structure for the USB Driver.

The `operationMode` is set to Device mode of operation. The `interruptSource` contains USB interrupt source number. The `interruptSourceUSBDma` contains the interrupt number corresponding to USB DMA. The driver will pass these as arguments to the peripheral library functions to control (enable/disable) the interrupt sources.

The PIC32MZ device has an integrated 8-channel DMA. If available, the DMA channel will be used to move the received data from the endpoint FIFO to the endpoint's data buffer. During a transmission, DMA will move the data from the endpoint's data buffer to the endpoint FIFO.

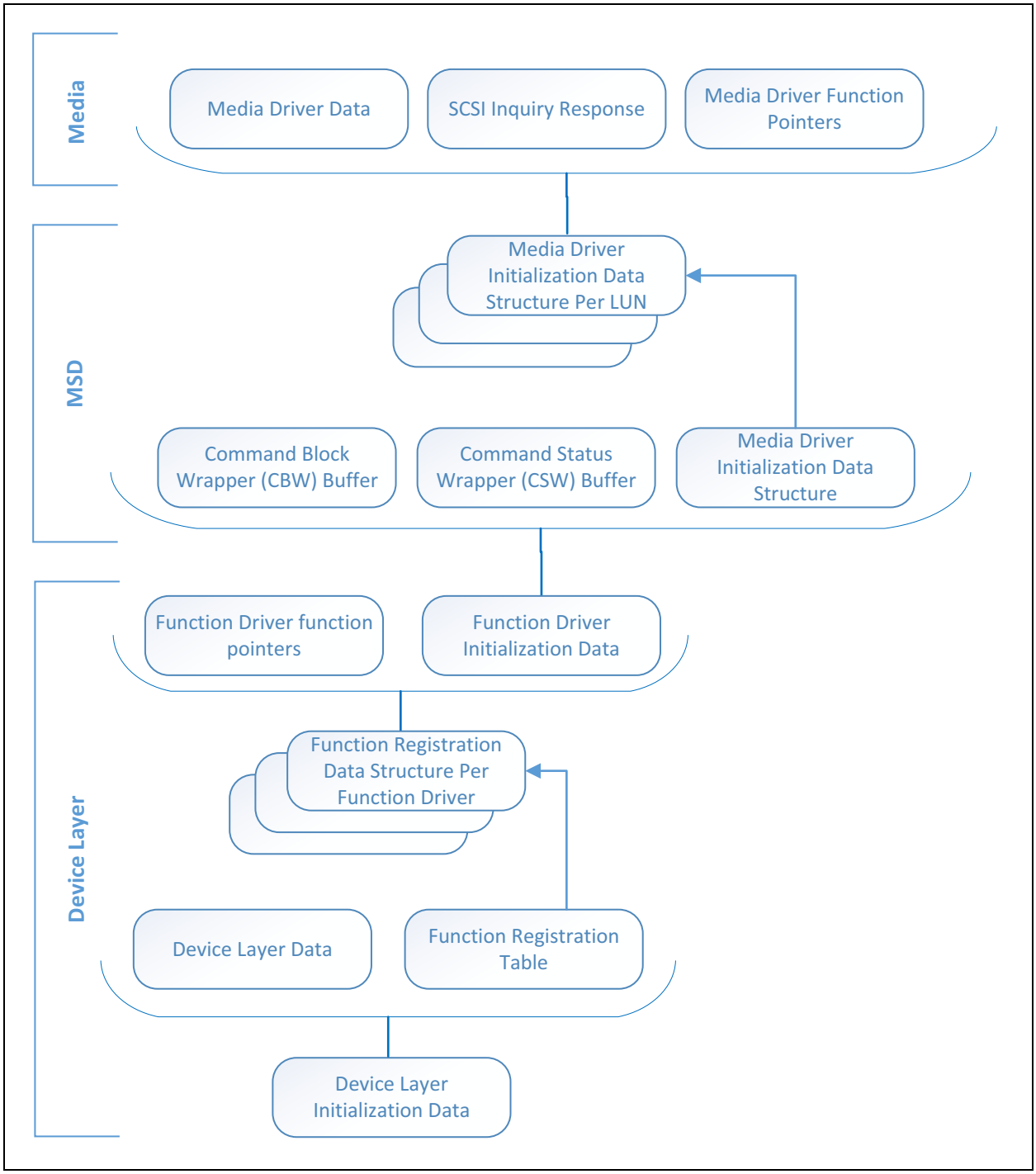
Figure 33 illustrates how the media drivers plug into the MSD Function Driver, and how the MSD Function Driver plugs into the Device Layer.

The `system_init.c` file also contains the USB descriptors for both High-Speed USB and Full-Speed USB. The string descriptors for Manufacturer String, Product String, and Serial Number String are also populated based on the MHC selections.

EXAMPLE 10: USB DRIVER INITIALIZATION DATA STRUCTURE

```
const DRV_USBHS_INIT drvUSBInit =
{
    /* Interrupt Source for USB module */
    .interruptSource = INT_SOURCE_USB_1,

    /* Interrupt Source for USB module */
    .interruptSourceUSBDma = INT_SOURCE_USB_1_DMA,

    /* System module initialization */
    .moduleInit = {SYS_MODULE_POWER_RUN_FULL},

    .operationMode = DRV_USBHS_OPMODE_DEVICE,

    .operationSpeed = USB_SPEED_HIGH,

    /* Stop in idle */
    .stopInIdle = false,

    /* Suspend in sleep */
    .suspendInSleep = false,

    /* Identifies peripheral (PLIB-level) ID */
    .usbID = USBHS_ID_0,
};
```

FIGURE 33: USB MSD APPLICATION INITIALIZATION DATA OVERVIEW



USB Mass Storage Device Task Function

Figure 34 shows the System Tasks function which runs the device driver tasks, system services, and the application task.

Figure 35 shows the USB MSD Task function. The USB MSD Task function (_USB_DEVICE_MSD_Tasks) is run in the context of the USB Device Task (USB_DEVICE_Tasks). The USB Device Task is, in turn, run from the System Tasks function (SYS_Tasks).

The USB MSD Task implements the MSD state machine which handles the CBW command, data stage, and CSW.

FIGURE 34: SYSTEM TASKS FUNCTION

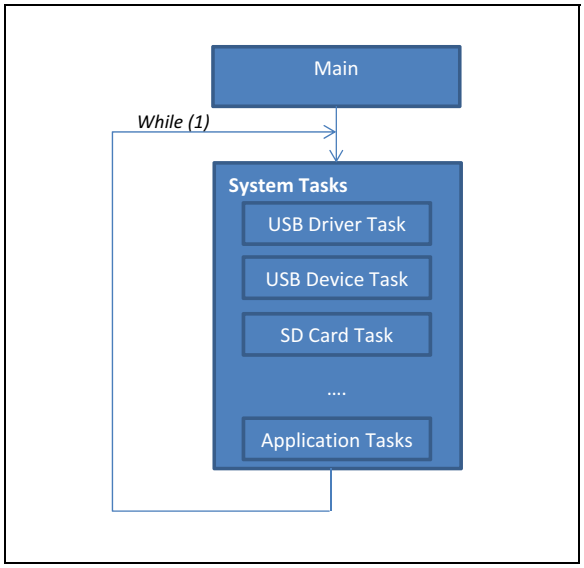


FIGURE 35: USB MSD TASK FUNCTION

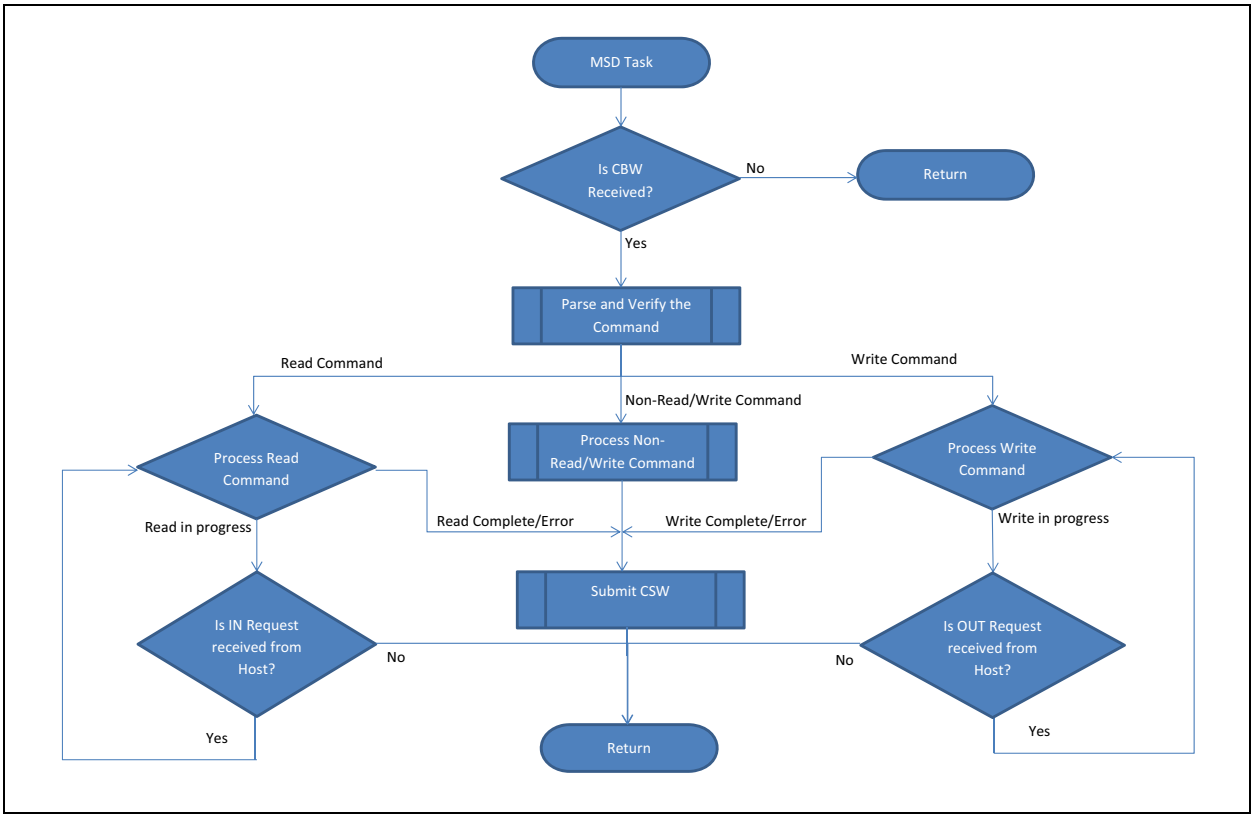


Table 15 provides important source and header file details in the MPLAB Harmony installer that are used by the application project. All of these files are automatically added into the MPLAB X IDE project by the MHC when the library or driver is selected.

TABLE 15: IMPORTANT SOURCE AND HEADER FILES REFERRED BY THE APPLICATION PROJECT

Source File Name	Description
framework/usb/src/dynamic/usb_device.c	This file implements the USB Device Layer interface.
framework/usb/src/dynamic/usb_device_msd.c	This file implements the MSD Function Driver interface.
framework/driver/usb/usbhs/src/dynamic/drv_usbhs.c	This file implements the functions accessed by the system module that allow it to initialize/deinitialize and maintain the driver. In addition it contains the client functions that allow opening, closing, and other general driver operations
framework/driver/usb/usbhs/src/dynamic/drv_usbhs_device.c	This fie implements the functions that allow the USB device stack to perform USB Device mode specific driver operations. It implements the USB driver interface expected by the USB Device Layer.

CONCLUSION

Developing a USB Mass Storage Device application requires an understanding of various standards, protocols, and middleware. Using the MPLAB Harmony USB Device Stack Framework, the accompanying MSD firmware, and the media drivers, users can design a solution without having to manage the underlying standards or protocols. The MPLAB Harmony provides users with a flexible, abstracted and fully integrated firmware development platform for PIC32 microcontrollers.

This application note introduces the basic concepts of the USB Mass Storage Class and provides an overview of the MPLAB Harmony USB Device Stack firmware architecture. It covers the terminologies, protocols, and standards that are involved in creating a USB Mass Storage Device application. It also describes how users can create and configure an MSD application with support for multiple logical units using the MPLAB Harmony Configurator (MHC) utility. Additional examples and demos for various MSD solutions are included with the MPLAB Harmony Integrated Software Framework, which is available for download from the Microchip website (see **"References"**).

REFERENCES

- AN1142 *"USB Mass Storage Class on an Embedded Host"* (http://www.microchip.com)
- USB Mass Storage - Designing and Programming Devices and Embedded Hosts - Jan Axelson (ISBN-10: 1931448043; ISBN-13: 978-1931448048)
- AT91 USB Mass Storage Device Driver Implementation (http://www.atmel.com)
- T10 Technical Committee Website (http://www.t10.org/drafts.htm)
- Universal Serial Bus Website (http://www.usb.org)
- help_harmony.pdf available with the MPLAB Harmony Software Framework Installer (http://www.microchip.com/mplab/mplab-harmony)
- USB MSD Demonstration with Multiple Drives - Stand-alone demonstration example (https://www.microchip.com/DevelopmentTools/ProductDetails.aspx?PartNO=DM320104)
- microSD Click board from MikroElektronika (https://shop.mikroe.com/click/storage/microsd)

NOTES:

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.

Microchip received ISO/TS-16949:2009 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® 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.

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
═ ISO/TS 16949 ═

Trademarks

ISBN: 978-1-5224-2327-0

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://www.microchip.com/support
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Austin, TX
Tel: 512-257-3370

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
Novi, MI
Tel: 248-848-4000

Houston, TX
Tel: 281-894-5983

Indianapolis
Noblesville, IN
Tel: 317-773-8323
Fax: 317-773-5453
Tel: 317-536-2380

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608
Tel: 951-273-7800

Raleigh, NC
Tel: 919-844-7510

New York, NY
Tel: 631-435-6000

San Jose, CA
Tel: 408-735-9110
Tel: 408-436-4270

Canada - Toronto
Tel: 905-695-1980
Fax: 905-695-2078

ASIA/PACIFIC

Australia - Sydney
Tel: 61-2-9868-6733

China - Beijing
Tel: 86-10-8569-7000

China - Chengdu
Tel: 86-28-8665-5511

China - Chongqing
Tel: 86-23-8980-9588

China - Dongguan
Tel: 86-769-8702-9880

China - Guangzhou
Tel: 86-20-8755-8029

China - Hangzhou
Tel: 86-571-8792-8115

China - Hong Kong SAR
Tel: 852-2943-5100

China - Nanjing
Tel: 86-25-8473-2460

China - Qingdao
Tel: 86-532-8502-7355

China - Shanghai
Tel: 86-21-3326-8000

China - Shenyang
Tel: 86-24-2334-2829

China - Shenzhen
Tel: 86-755-8864-2200

China - Suzhou
Tel: 86-186-6233-1526

China - Wuhan
Tel: 86-27-5980-5300

China - Xian
Tel: 86-29-8833-7252

China - Xiamen
Tel: 86-592-2388138

China - Zhuhai
Tel: 86-756-3210040

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-3090-4444

India - New Delhi
Tel: 91-11-4160-8631

India - Pune
Tel: 91-20-4121-0141

Japan - Osaka
Tel: 81-6-6152-7160

Japan - Tokyo
Tel: 81-3-6880- 3770

Korea - Daegu
Tel: 82-53-744-4301

Korea - Seoul
Tel: 82-2-554-7200

Malaysia - Kuala Lumpur
Tel: 60-3-7651-7906

Malaysia - Penang
Tel: 60-4-227-8870

Philippines - Manila
Tel: 63-2-634-9065

Singapore
Tel: 65-6334-8870

Taiwan - Hsin Chu
Tel: 886-3-577-8366

Taiwan - Kaohsiung
Tel: 886-7-213-7830

Taiwan - Taipei
Tel: 886-2-2508-8600

Thailand - Bangkok
Tel: 66-2-694-1351

Vietnam - Ho Chi Minh
Tel: 84-28-5448-2100

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

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

Finland - Espoo
Tel: 358-9-4520-820

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

Germany - Garching
Tel: 49-8931-9700

Germany - Haan
Tel: 49-2129-3766400

Germany - Heilbronn
Tel: 49-7131-67-3636

Germany - Karlsruhe
Tel: 49-721-625370

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

Germany - Rosenheim
Tel: 49-8031-354-560

Israel - Ra'anana
Tel: 972-9-744-7705

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

Italy - Padova
Tel: 39-049-7625286

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

Norway - Trondheim
Tel: 47-7289-7561

Poland - Warsaw
Tel: 48-22-3325737

Romania - Bucharest
Tel: 40-21-407-87-50

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

Sweden - Gothenberg
Tel: 46-31-704-60-40

Sweden - Stockholm
Tel: 46-8-5090-4654

UK - Wokingham
Tel: 44-118-921-5800
Fax: 44-118-921-5820