

---

## Application Note: AN00125

# USB Mass Storage Device Class

This application note shows how to create a USB device compliant to the standard USB mass storage device class on an XMOS multicore microcontroller.

The code associated with this application note provides an example of using the XMOS Device Library and associated USB class descriptors to provide a framework for the creation of a USB mass storage device.

The mass storage framework uses XMOS libraries to provide a bidirectional mass storage device example over high speed USB.

Note: This application note provides a standard USB Mass Storage Device Class which addresses Bulk-Only Transport (BOT) or Bulk/Bulk/Bulk (BBB) specification and as a result does not require drivers to run on Windows, Linux or Mac.

The Peripheral Device Type (PDT) supported in this application note is SCSI (Small Computer System Interface) Block Command (SBC) Direct-access device (e.g., UHD (Ultra High Definition) Floppy disk). This example application uses the on-board serial flash M25P16 as its memory device.

---

## Required tools and libraries

- xTIMEcomposer Tools - Version 14.0.0
- XMOS USB Device Library - Version 3.1.0

## Required hardware

This application note is designed to run on an XMOS xCORE-USB series device.

The example code provided with the application has been implemented and tested on the xCORE-USB sliceKIT 1V2 (XP-SKC-U16) core board using USB AB sliceCARD 1V2 (XA-SK-USB-AB) but there is no dependency on this board and it can be modified to run on any development board which uses an xCORE-USB series device.

## Prerequisites

- This document assumes familiarity with the XMOS xCORE architecture, the Universal Serial Bus 2.0 Specification (and related specifications), the XMOS tool chain and the xC language. Documentation related to these aspects which are not specific to this application note are linked to in the references appendix.
- For descriptions of XMOS related terms found in this document please see the XMOS Glossary<sup>1</sup>.
- For the full API listing of the XMOS USB Device (XUD) Library please see the document XMOS USB Device (XUD) Library<sup>2</sup>.
- For information on designing USB devices using the XUD library please see the XMOS USB Device Design Guide for reference<sup>3</sup>.

---

<sup>1</sup><http://www.xmos.com/published/glossary>

<sup>2</sup><http://www.xmos.com/published/xuddg>

<sup>3</sup><http://www.xmos.com/published/xmos-usb-device-design-guide>

# 1 Overview

## 1.1 Introduction

The Universal Serial Bus (USB) is a communication architecture that gives a PC the ability to interconnect a variety of devices via a simple four-wire cable. One such device is the mass storage. Traditionally, mass storage device class provides adaptability for devices like

- USB flash drive
- memory card reader
- digital audio player
- digital camera
- external hard drive

The USB specification provides as standard device class for the implementation of USB mass storage.

([http://www.usb.org/developers/docs/devclass\\_docs/usbmassbulk\\_10.pdf](http://www.usb.org/developers/docs/devclass_docs/usbmassbulk_10.pdf))

## 1.2 Block diagram

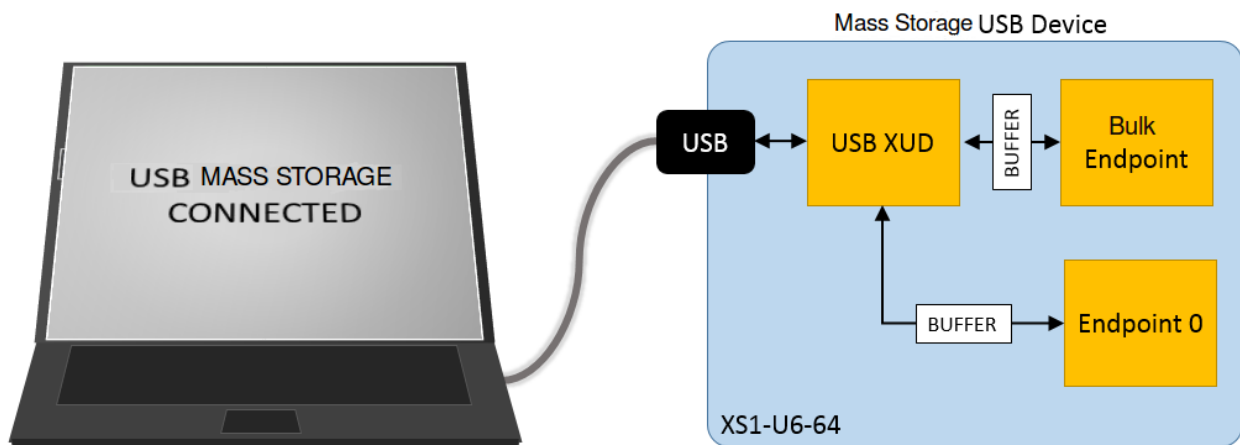


Figure 1: Block diagram of USB Mass storage application example

## 1.3 Features

This section describes the features that are supported by the demo application. The application uses an on-board flash with a user memory partition of 2 MB. The application does the following

- Enumerates as Mass Storage device
- On Windows host, the display appears as “ drive labelled as unformatted **Removable Disk** “
- On Mac, it displays as “ **XMOSLTD Flash Disk Media** “

As there is no file system supported by the application, formatting the drive, read and write operations on the drive is not featured.

## 2 USB Mass Storage Device Class application note

The demo in this note uses the XMOS USB device library and shows a simple program that receives data from and sends data to the host.

For the USB Mass storage device class application example, the system comprises three tasks running in separate logical cores of a xCORE-USB multicore microcontroller.

The tasks perform the following operations.

- A task containing the USB library functionality to communicate over USB
- A task implementing Endpoint0 responding both standard and mass storage class USB requests
- A task implementing the application code for receiving and sending mass storage data into the device

These tasks communicate via the use of xCONNECT channels which allow data to be passed between application code running on separate logical cores.

The following diagram shows the task and communication structure for this USB mass storage device class application example.

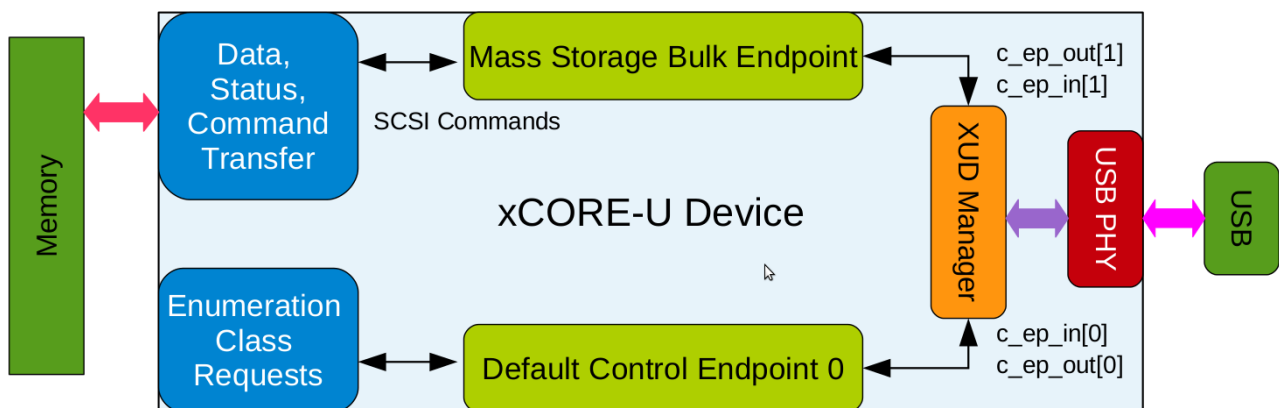


Figure 2: Task diagram of USB mass storage application example

### 2.1 Makefile additions for this application

To start using the USB library, you need to add **lib\_usb** to your makefile:

```
USED_MODULES = lib_usb
```

You can then access the USB functions on your source code via the **usb.h** header file:

```
#include <usb.h>
```

### 2.2 Declaring resource and setting up the USB components

**main.xc** contains the application implementation for a device based on the USB mass storage device class. There are some defines in it that are used to configure the XMOS USB device library. These are displayed below.

```

/* USB Endpoint Defines */
#define XUD_EP_COUNT_OUT 2 //Includes EP0 (1 out EP0 + Mass Storage data output EP)
#define XUD_EP_COUNT_IN 2 //Includes EP0 (1 in EP0 + Mass Storage data input EP)

```

These describe the endpoint configuration for this device. This example has bi-directional communication with the host machine via the standard endpoint0, an endpoint for receiving the bulk data from the host into our device and an endpoint for sending the bulk data to host from our device.

## 2.3 The application main() function

Below is the source code for the main function of this application, which is taken from the source file **main.xc**

```

int main()
{
    chan c_ep_out[XUD_EP_COUNT_OUT], c_ep_in[XUD_EP_COUNT_IN];

    par
    {
        on USB_TILE: xud(c_ep_out, XUD_EP_COUNT_OUT, c_ep_in, XUD_EP_COUNT_IN,
                        null, XUD_SPEED_HS, XUD_PWR_SELF);

        on USB_TILE: Endpoint0(c_ep_out[0], c_ep_in[0]);

        on USB_TILE: massStorageClass(c_ep_out[1], c_ep_in[1], 0);
    }
    return 0;
}

```

Looking at this in a more detail you can see the following:

- The *par* statement starts three separate tasks in parallel
- There is a task to configure and execute the USB library: *xud()*
- There is a task to startup and run the Endpoint0 code: *Endpoint0()*
- There is a task to deal with USB mass storage requests arriving from the host: *massStorageClass()*
- The define *USB\_TILE* describes the tile on which the individual tasks will run
- In this example all tasks run on the same tile as the USB PHY although this is only a requirement of *xud()*
- The xCONNECT communication channels used by the application are set up at the beginning of *main()*
- The USB defines discussed earlier are passed into the function *xud()*

## 2.4 Configuring the USB Device ID

The USB ID values used for vendor id, product id and device version number are defined in the file **endpoint0.xc**. These are used by the host machine to determine the vendor of the device (in this case XMOS) and the product plus the firmware version.

```

/* USB Device ID Defines */
#define BCD_DEVICE 0x0010
#define VENDOR_ID 0x20B1
#define PRODUCT_ID 0x10BA

```

## 2.5 USB Mass storage Class specific defines

The USB Mass storage Class is configured in the file **endpoint0.xc**. Below there are a set of standard defines which are used to configure the USB device descriptors to setup a USB mass storage device running on an xCORE-USB microcontroller.

```

/* USB Mass Storage Interface Subclass Definition */
#define USB_MASS_STORAGE_SUBCLASS 0x06 /* SCSI transparent command set */

/* USB Mass Storage interface protocol */
#define USB_MASS_STORAGE_PROTOCOL 0x50 /* USB Mass Storage Class Bulk-Only (BBB) Transport */

/* USB Mass Storage Request Code */
#define USB_MASS_STORAGE_RESET 0xFF /* Bulk-Only Mass Storage Reset */
#define USB_MASS_STORAGE_GML 0xFE /* Get Max LUN (GML) */
  
```

## 2.6 USB Device Descriptor

**endpoint0.xc** is where the standard USB device descriptor is declared for the mass storage device. Below is the structure which contains this descriptor. This will be requested by the host when the device is enumerated on the USB bus.

```

static unsigned char devDesc[] =
{
    0x12, /* 0 bLength */
    USB_DESCRIPTOR_DEVICE, /* 1 bdescriptorType */
    0x00, /* 2 bcdUSB version */
    0x02, /* 3 bcdUSB version */
    0x00, /* 4 bDeviceClass - Specified by interface */
    0x00, /* 5 bDeviceSubClass - Specified by interface */
    0x00, /* 6 bDeviceProtocol - Specified by interface */
    0x40, /* 7 bMaxPacketSize for EP0 - max = 64*/
    (VENDOR_ID & 0xFF), /* 8 idVendor */
    (VENDOR_ID >> 8), /* 9 idVendor */
    (PRODUCT_ID & 0xFF), /* 10 idProduct */
    (PRODUCT_ID >> 8), /* 11 idProduct */
    (BCD_DEVICE & 0xFF), /* 12 bcdDevice */
    (BCD_DEVICE >> 8), /* 13 bcdDevice */
    0x01, /* 14 iManufacturer - index of string */
    0x02, /* 15 iProduct - index of string */
    0x03, /* 16 iSerialNumber - index of string */
    0x01 /* 17 bNumConfigurations */
};
  
```

From this descriptor you can see that product, vendor and device firmware revision are all coded into this structure. This will allow the host machine to recognise the mass storage device when it is connected to the USB bus.

## 2.7 USB Configuration Descriptor

The USB configuration descriptor is used to configure the device in terms of the device class and the endpoint setup. For the USB mass storage device the configuration descriptor which is read by the host is as follows.

```

static unsigned char cfgDesc[] = {
    0x09,                /* 0  bLength */
    USB_DESCTYPE_CONFIGURATION, /* 1  bDescriptorType = configuration */
    0x20, 0x00,         /* 2  wTotalLength of all descriptors */
    0x01,                /* 4  bNumInterfaces */
    0x01,                /* 5  bConfigurationValue */
    0x00,                /* 6  iConfiguration - index of string */
    0x80,                /* 7  bmAttributes - Self powered */
    0x50,                /* 8  bMaxPower - 160mA */

    /* USB Bulk-Only Data Interface Descriptor */
    0x09,                /* 0  bLength */
    USB_DESCTYPE_INTERFACE, /* 1  bDescriptorType */
    0x00,                /* 2  bInterfaceNumber */
    0x00,                /* 3  bAlternateSetting */
    0x02,                /* 4: bNumEndpoints */
    USB_CLASS_MASS_STORAGE, /* 5: bInterfaceClass */
    USB_MASS_STORAGE_SUBCLASS, /* 6: bInterfaceSubClass */
    USB_MASS_STORAGE_PROTOCOL, /* 7: bInterfaceProtocol */
    0x00,                /* 8  iInterface */

    /* Bulk-In Endpoint Descriptor */
    0x07,                /* 0  bLength */
    USB_DESCTYPE_ENDPOINT, /* 1  bDescriptorType */
    0x81,                /* 2  bEndpointAddress - EP1, IN */
    XUD_EPTYPE_BUL,     /* 3  bmAttributes */
    0x00,                /* 4  wMaxPacketSize - Low */
    0x02,                /* 5  wMaxPacketSize - High */
    0x00,                /* 6  bInterval */

    /* Bulk-Out Endpoint Descriptor */
    0x07,                /* 0  bLength */
    USB_DESCTYPE_ENDPOINT, /* 1  bDescriptorType */
    0x01,                /* 2  bEndpointAddress - EP1, OUT */
    XUD_EPTYPE_BUL,     /* 3  bmAttributes */
    0x00,                /* 4  wMaxPacketSize - Low */
    0x02,                /* 5  wMaxPacketSize - High */
    0x00,                /* 6  bInterval */
};

```

From this you can see that the USB mass storage class defines described earlier are encoded into the configuration descriptor along with the bulk USB endpoint description for receiving storage data into the application code. These endpoint allows us to process the storage data request from the host and to the host inside the main mass storage application task.

## 2.8 USB string descriptor

There is one more descriptor within this file relating to the configuration of the USB Mass storage Class. This section should also be modified to match the capabilities of the mass storage device.

```

/* String table - unsafe as accessed via shared memory */
static char * unsafe stringDescriptors[]=
{
  "\x09\x04",          // Language ID string (US English)
  "XMOS",              // iManufacturer
  "xMASSstorage",     // iProduct
  "XD070101ho4I4KwM", // iSerial Number
};

```

## 2.9 USB Mass storage Class requests

Inside **endpoint0.xc** there is some code for handling the USB Mass storage device class specific requests. These are shown in the following code:

```

/* Mass Storage Class Requests */
int MassStorageEndpoint0Requests(XUD_ep ep0_out, XUD_ep ep0_in, USB_SetupPacket_t sp)
{
  unsigned char buffer[1] = {0};

  switch(sp.bRequest) {

    case USB_MASS_STORAGE_RESET:
      XUD_ResetEpStateByAddr(1);
      return XUD_RES_RST; // This request is used to reset the mass storage device
      break;

    case USB_MASS_STORAGE_GML:
      return XUD_DoGetRequest(ep0_out, ep0_in, buffer, 1, sp.wLength);
      break;

    default:
      debug_printf("MassStorageEndpoint0Requests @ default : 0x%x\n", sp.bRequest);
      break;
  }

  return XUD_RES_ERR;
}

```

These mass storage specific request are implemented by the application as they do not form part of the standard requests which have to be accepted by all device classes via endpoint0.

## 2.10 USB Mass storage Class Endpoint0

The function *Endpoint0()* contains the code for dealing with device requests made from the host to the standard endpoint0 which is present in all USB devices. In addition to requests required for all devices, the code handles the requests specific to the mass storage device class.

```

if(result == XUD_RES_OKAY)
{
    /* Set result to ERR, we expect it to get set to OKAY if a request is handled */
    result = XUD_RES_ERR;

    /* Stick bmRequest type back together for an easier parse... */
    bmRequestType = (sp.bmRequestType.Direction<<7) | (sp.bmRequestType.Type<<5) |
        (sp.bmRequestType.Recipient);

    if ((bmRequestType == USB_BMREQ_H2D_STANDARD_DEV) && (sp.bRequest == USB_CLEAR_FEATURE)) {
        // Host has set device address, value contained in sp.wValue
    }

    /* Handle specific requests first */
    switch(bmRequestType) {
        /* Direction: Device-to-host and Host-to-device
        * Type: Class
        * Recipient: Interface
        */
        case USB_BMREQ_H2D_CLASS_INT:
        case USB_BMREQ_D2H_CLASS_INT:

            /* Inspect for mass storage interface num */
            if(sp.wIndex == 0) {
                /* Returns XUD_RES_OKAY if handled,
                * XUD_RES_ERR if not handled,
                * XUD_RES_RST for bus reset */
                result = MassStorageEndpoint0Requests(ep0_out, ep0_in, sp);
            }
            break;
    }
}
}
    
```

## 2.11 Receiving storage data from the host

The application endpoint for Command Transport, Data-In, Data-Out and Status Transport are implemented in the file `mass_storage.xc` as per the flow shown in below figure. This is contained within the function `massStorageClass()` which is shown in next page.

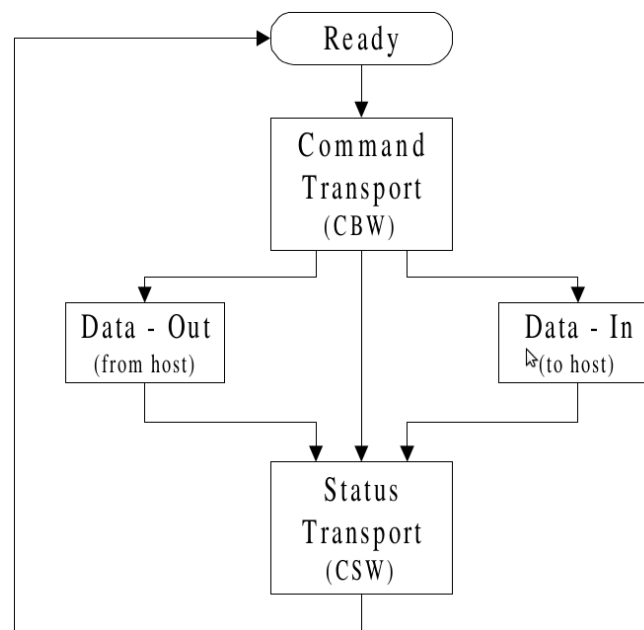


Figure 3: Command/Data/Status Flow



```

void massStorageClass(chanend chan_ep1_out,chanend chan_ep1_in, int writeProtect)
{
  unsigned char commandBlock[CBW_SHORT_PACKET_SIZE];
  unsigned char commandStatus[CSW_SHORT_PACKET_SIZE];
  unsigned host_transfer_length = 0;
  int readCapacity[8];
  int readLength, readAddress;
  int dCBWSignature = 0, bCBWDataTransferLength = 0;
  int bmCBWFlags = 0, bCBWLUN = 0, bCBWCBLength = 0;
  int Operation_Code = 0;
  XUD_Result_t result;
  int ready = 1;

  debug_printf("USB Mass Storage class demo started\n");

  /* Load some default CSW to reduce response time delay */
  memset(commandStatus,0,CSW_SHORT_PACKET_SIZE);
  /* Signature helps identify this data packet as a CSW */
  (commandStatus, int[])[0] = byterev(CSW_SIGNATURE);

  /* Initialise the XUD endpoints */
  XUD_ep ep1_out = XUD_InitEp(chan_ep1_out, XUD_EPTYPE_BUL);
  XUD_ep ep1_in = XUD_InitEp(chan_ep1_in, XUD_EPTYPE_BUL);

#ifdef !DETECT_AS_FLOPPY
  massStorageInit();
#endif

  while(1)
  {
    unsigned char bCSWStatus = CSW_STATUS_CMD_PASSED;
    // Get Command Block Wrapper (CBW)
    if(XUD_RES_OKAY == (result = XUD_GetBuffer(ep1_out, (commandBlock, char[CBW_SHORT_PACKET_SIZE]),
      ↪ host_transfer_length)))
    {
      /* The CBW shall start on a packet boundary and shall end as a short packet
       * with exactly 31 (0x1F) bytes transferred
       */
      assert(host_transfer_length == CBW_SHORT_PACKET_SIZE);
      /* verify Signature - that helps identify this packet as a CBW */
      dCBWSignature = commandBlock[0] | commandBlock[1] << 8 |
        commandBlock[2] << 16 | commandBlock[3] << 24;
      assert(dCBWSignature == CBW_SIGNATURE);

      bCBWDataTransferLength = commandBlock[8] | commandBlock[9]<<8 |
        commandBlock[10] << 16 | commandBlock[11] << 24;

      bmCBWFlags = commandBlock[12]; bCBWLUN = (commandBlock[13] & 0x0F);
      assert(bCBWCBLength = (commandBlock[14] & 0x1F) <= 16);
      Operation_Code = commandBlock[15];

      switch(Operation_Code)
      {
        case TEST_UNIT_READY_CMD: // Test unit ready:
          bCSWStatus = ready ? CSW_STATUS_CMD_PASSED : CSW_STATUS_CMD_FAILED;
          break;

        case REQUEST_SENSE_CMD: // Request sense
          requestSenseAnswer[2] = ready ? STATUS_GOOD : STATUS_CHECK_CONDITION;
          result = XUD_SetBuffer(ep1_in, requestSenseAnswer, sizeof(requestSenseAnswer));
          break;

        case INQUIRY_CMD: // Inquiry
          result = XUD_SetBuffer(ep1_in, inquiryAnswer, sizeof(inquiryAnswer));
          break;

        case START_STOP_CMD: // start/stop
          ready = ((commandBlock[19] >> 1) & 1) == 0;
          break;

        case MODE_SENSE_6_CMD: // Mode sense (6)
        case MODE_SENSE_10_CMD: // Mode sense (10) // For Mac OSX
          if (writeProtect) modeSenseAnswer[2] |= 0x80;

          result = XUD_SetBuffer(ep1_in, modeSenseAnswer, sizeof(modeSenseAnswer));
          break;
      }
    }
  }
}

```

```

case MEDIUM_REMOVAL_CMD: // Medium removal
    break;

case RECEIVE_DIAGNOSTIC_RESULT_CMD:
    memset(readCapacity,0x0000,sizeof(readCapacity));
    result = XUD_SetBuffer(ep1_in, (readCapacity, unsigned char[8]), 32);
    break;

case READ_FORMAT_CAPACITY_CMD: // Read Format capacity (UFI Command Spec)
    readCapacity[0] = bytereve(8);
    readCapacity[1] = bytereve(massStorageSize());
    readCapacity[2] = bytereve(MASS_STORAGE_BLOCKLENGTH) | (DETECT_AS_FLOPPY ? NO_CARTRIDGE_IN_DRIVE :
        ↪ FORMATTED_MEDIA);
    result = XUD_SetBuffer(ep1_in, (readCapacity, unsigned char[8]), 12);
    break;

case READ_CAPACITY_CMD: // Read capacity
    readCapacity[0] = bytereve(massStorageSize()-1);
    readCapacity[1] = bytereve(MASS_STORAGE_BLOCKLENGTH);
    result = XUD_SetBuffer(ep1_in, (readCapacity, unsigned char[8]), 8);
    break;

case READ_CAPACITY_16_CMD:
    memset(readCapacity,0x0000,sizeof(readCapacity));
    readCapacity[1] = bytereve(massStorageSize()-1);
    readCapacity[2] = bytereve(MASS_STORAGE_BLOCKLENGTH);
    result = XUD_SetBuffer(ep1_in, (readCapacity, unsigned char[8]), 32);
    break;

case READ_10_CMD: // Read (10)
    readLength = commandBlock[22] << 8 | commandBlock[23];
    readAddress = commandBlock[17] << 24 | commandBlock[18] << 16 |
        commandBlock[19] << 8 | commandBlock[20];
    for(int i = 0; i < readLength ; i++) {
        bCSWStatus |= massStorageRead(readAddress, blockBuffer);
        result = XUD_SetBuffer(ep1_in, blockBuffer, MASS_STORAGE_BLOCKLENGTH);
        readAddress++; }
    break;

case WRITE_10_CMD: // Write
    readLength = commandBlock[22] << 8 | commandBlock[23];
    readAddress = commandBlock[17] << 24 | commandBlock[18] << 16 |
        commandBlock[19] << 8 | commandBlock[20];
    for(int i = 0; i < readLength ; i++) {
        result = XUD_GetBuffer(ep1_out, (blockBuffer, char[128 * 4]),host_transfer_length);
        bCSWStatus |= massStorageWrite(readAddress, blockBuffer);
        readAddress++; }
    break;

default:
    debug_printf("Invalid Operation Code Received : 0x%x\n",Operation_Code);
    bCSWStatus = CSW_STATUS_CMD_FAILED;
    break;
}
}

/* Check for result, if it is found as XUD_RES_RST, then reset Endpoints */
if(result == XUD_RES_RST) {
    XUD_ResetEndpoint(ep1_out,ep1_in);
    break;
}

/* Setup Command Status Wrapper (CSW). The CSW shall start on a packet boundry
 * and shall end as a short packet with exactly 13 (0x0D) bytes transferred */
/* The device shall echo the contents of dCBWTag back to the host in the dCSWTag */
commandStatus[4] = commandBlock[4];
commandStatus[5] = commandBlock[5];
commandStatus[6] = commandBlock[6];
commandStatus[7] = commandBlock[7];
commandStatus[12] = bCSWStatus;

if(XUD_RES_RST == XUD_SetBuffer(ep1_in, commandStatus, CSW_SHORT_PACKET_SIZE))
    XUD_ResetEndpoint(ep1_out,ep1_in);

} //while(1)
} // END of massStorageClass

```

From this you can see the following.

- Two buffers are declared, one to receive the Command transport CBW (Command Block Wrapper) data which is streamed into the application from the host and the other to send Status transport CSW (Command Status Wrapper) to the host.
- This task operates inside a while (1) loop which waits for data to arrive and then processes it.
- It checks for the CBW Signature and packet size that get received from the host.
- Based on the Operation code available on the CBWCB (CBW Command Block) field, SCSI commands corresponding to flash drive are executed.
  - Operation code received may correspond to SCSI commands like *Inquiry, Test Unit Ready, Request Sense, Read Capacity, Mode Sense, Read/Write*, etc.
- Once the execution (Command transport, Data-In or Data-Out) is completed, the device shall echo the contents of CBWTag field sent by the host back along with the CSWStatus and CSW Signature as Status transport. This ensures that the *Status* send to host is associated with the *Command* received from host.

## 2.12 SCSI Command Implementation

Some of the SCSI command definitions and the response from the device are mentioned below.

- **INQUIRY:** command requests that information regarding parameters of the Device be sent to the Host.
  - The response to this command provides standard INQUIRY data of (36 bytes) PDT, RMB (Removable Medium Bit), Vendor Identification, Product Identification and Product Revision Level, etc.
- **TEST UNIT READY:** command provides a means to check if the Device is ready. This command is useful in that it allows a Host to poll a Device until it is ready without the need to allocate space for returned data.
  - The response to this command returns GOOD, if the device accepts an appropriate medium access command or CHECK CONDITION status with a sense key of NOT READY.
- **REQUEST SENSE:** command requests the device to transfer sense data to the host whenever an error is reported. The sense data describes what caused the error condition.
  - The response to this command is a sense key, Additional Sense Code (ASC) or ASC Qualifier (ASCQ) depending on which error occurred<sup>4</sup>.
- **READ CAPACITY:** command requests the device to transfer 8 bytes of parameter data describing the capacity of the installed medium of the device.
  - The response to this command returns Logical Block Address (LBA) and block length in bytes of the memory device.
- **MODE SENSE:** command requests the device to transfer parameter data describing the medium type, block descriptor length, read/write error recovery mode.
  - The response to this command returns PDT type as medium type, error recovery mode, descriptor length.
- **READ/WRITE:** command requests the device to read/write data onto the medium.
  - The response to *READ* transfers the most recent data value written in the addressed logical block of the medium to host.
  - The response to *WRITE* writes the data transferred by the host to the medium.

<sup>4</sup><http://www.t10.org/lists/asc-num.htm>

## 2.13 Serial Flash Functions

For accessing the on-board serial flash, you need to add the flash library **lflash** in *Makefile* as one of the *XCC\_FLAGS* and use **flashlib.h** to access the flash library functions. *Note: No separate core is required to handle the SPI read/write*

The below implementation does write/read operation onto the on-board serial flash M25P16.

```
#include "flashlib.h"

on tile[0]: fl_SPIPorts spiPort = {
    XS1_PORT_1A,
    XS1_PORT_1B,
    XS1_PORT_1C,
    XS1_PORT_1D,
    XS1_CLKBLK_1
};

fl_DeviceSpec flashSpec[1] = {FL_DEVICE_NUMONYX_M25P16}; // Equivalent of Micron M25P16

int pagesPerBlock_g = 0;
int bytesPerPage_g = 0;
unsigned char pageBuffer_g[MASS_STORAGE_BLOCKLENGTH];

void massStorageInit() {
    fl_connectToDevice(spiPort, flashSpec, 1);
    fl_setBootPartitionSize(FLASH_PARTITION_SIZE);
    bytesPerPage_g = fl_getPageSize();
    pagesPerBlock_g = (MASS_STORAGE_BLOCKLENGTH / bytesPerPage_g);
}

int massStorageWrite(unsigned int blockNr, unsigned char buffer[]) {
    for(int i = 0; i < pagesPerBlock_g; i++) {
        for(int j = 0; j < bytesPerPage_g; j++) {
            pageBuffer_g[j] = buffer[i * bytesPerPage_g + j];
        }
        fl_writeDataPage(blockNr * pagesPerBlock_g + i, buffer);
    }
    return 0;
}

int massStorageRead(unsigned int blockNr, unsigned char buffer[]) {
    for(int i = 0; i < pagesPerBlock_g; i++) {
        fl_readDataPage(blockNr * pagesPerBlock_g + i, pageBuffer_g);
        for(int j = 0; j < bytesPerPage_g; j++) {
            buffer[i * bytesPerPage_g + j] = pageBuffer_g[j];
        }
    }
    return 0;
}

int massStorageSize() {
#ifdef DETECT_AS_FLOPPY
    return FLOPPY_DISK_SIZE;
#else
    int x = fl_getNumDataPages();
    return x / pagesPerBlock_g;
#endif
}
```

## APPENDIX A - Demo Hardware Setup

To run the demo, connect the xCORE-USB sliceKIT USB-B and xTAG-2 USB-A connectors to separate USB connectors on your development PC.

On the xCORE-USB sliceKIT ensure that the xCONNECT LINK switch is set to ON, as per the image, to allow xSCOPE to function. The use of xSCOPE is required in this application so that the print message that are generated on the device as part of the demo do not interfere with the real-time behavior of the USB device.

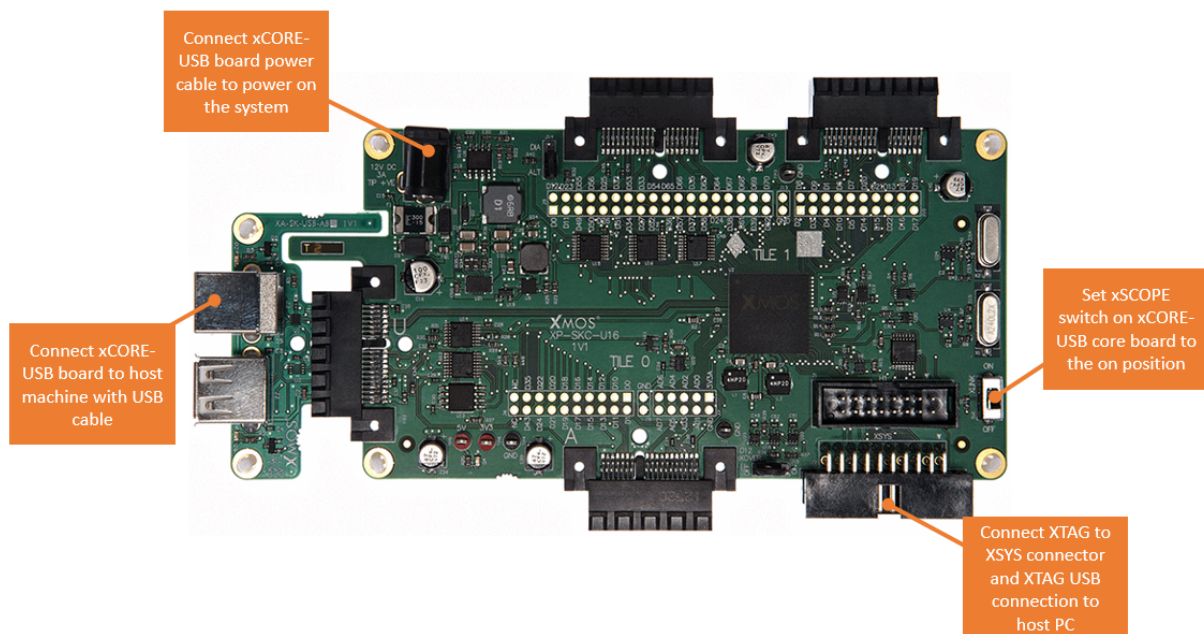


Figure 4: Xmos xCORE-USB sliceKIT

The hardware should be configured as displayed above for this demo:

- The XTAG debug adapter should be connected to the XSYS connector and the XTAG USB cable should be connected to the host machine
- The xCORE-USB core board should have a USB cable connecting the device to the host machine
- The xSCOPE switch on the board should be set to the ON position
- The xCORE-USB core board should have the power cable connected

## APPENDIX B - Launching the demo device

Once the demo example has been built either from the command line using `xmake` or via the build mechanism of `xTIMEcomposer studio` the application can be executed on the `xCORE-USB sliceKIT`.

Once built there will be a `bin` directory within the project which contains the binary for the `xCORE` device. The `xCORE` binary has a `XMOS` standard `.xe` extension.

### B.1 Launching from the command line

From the command line the `xrun` tool is used to download code to both the `xCORE` devices. Changing into the `bin` directory of the project we can execute the code on the `xCORE` microcontroller as follows:

```
> xrun --xscope AN00125.xe <-- Download and execute the xCORE code
```

Once this command has executed the mass storage USB device should have enumerated on your machine

### B.2 Launching from xTIMEcomposer Studio

From `xTIMEcomposer Studio` the `run` mechanism is used to download code to `xCORE` device. Select the `xCORE` binary from the `bin` directory, right click and then follow the instructions below.

- Select **Run As**.
- Select **Run Configurations**.
- Double click on `xCORE` application\*\*.
- Enable `xSCOPE` in Target I/O options:

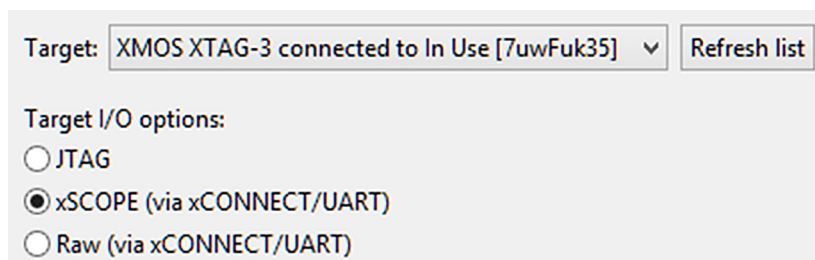


Figure 5: xTIMEcomposer xSCOPE configuration

- Click **Apply** and then **Run**.

Once this command has executed the mass storage USB device should have enumerated on your machine and the below message should get displayed on your `xTIMEcomposer` console window

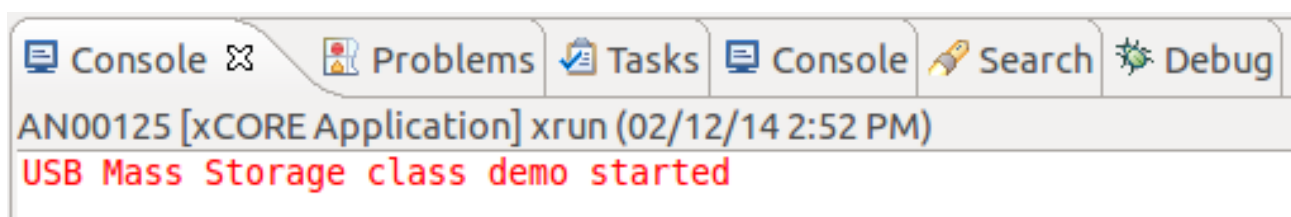


Figure 6: xTIMEcomposer Console Window

## APPENDIX C - Detecting the Mass Storage

### C.1 Windows Host

- There is no specific driver installation is required. Windows will automatically install a driver for the mass storage class. You should be able to get the below message screen once enumeration is complete.

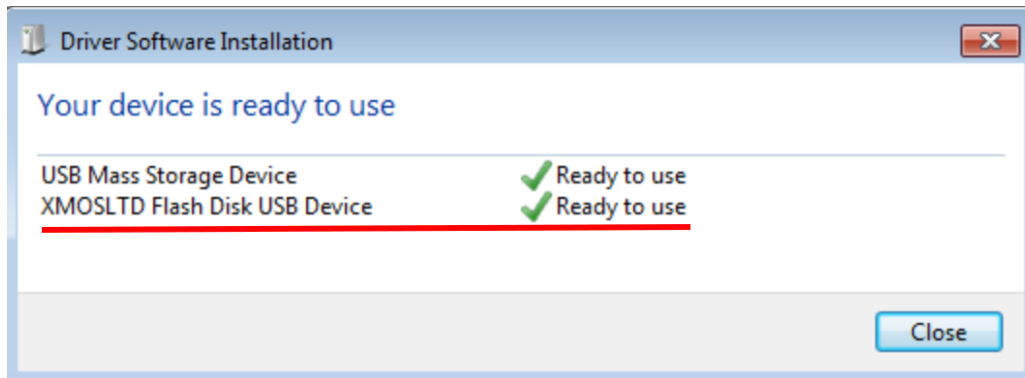


Figure 7: Driver Software Installation Screen (On Windows 7)

- You can also verify the device using **Device Manager** Menu
  - Click on **Start Menu** goto **My Computer** - *Right click* and navigate to **Manage** option.
  - On **Computer Management** screen *double click* **Device Manager** on left and navigate to **Universal Serial Bus controllers** option on right.
  - *Double click* on **Universal Serial Bus controllers** and you should be able to see **USB Mass Storage Device**

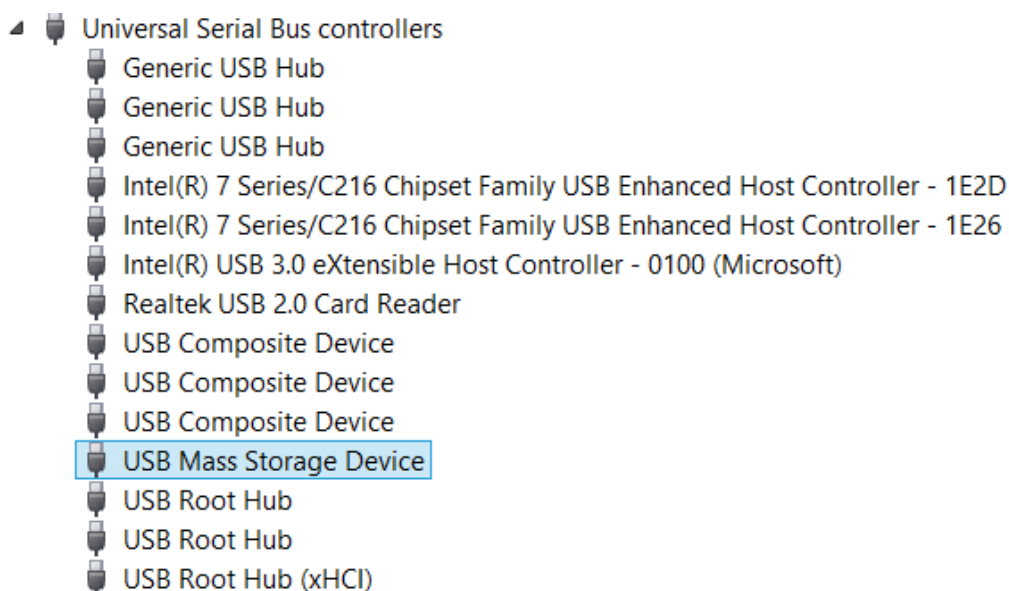


Figure 8: USB Mass Storage Device detection on Device Manager (On Windows 7)

- The Mass Storage Device should be seen as an **Removable Disk** (Drive ID may vary depending upon the number of drives available on your machine).

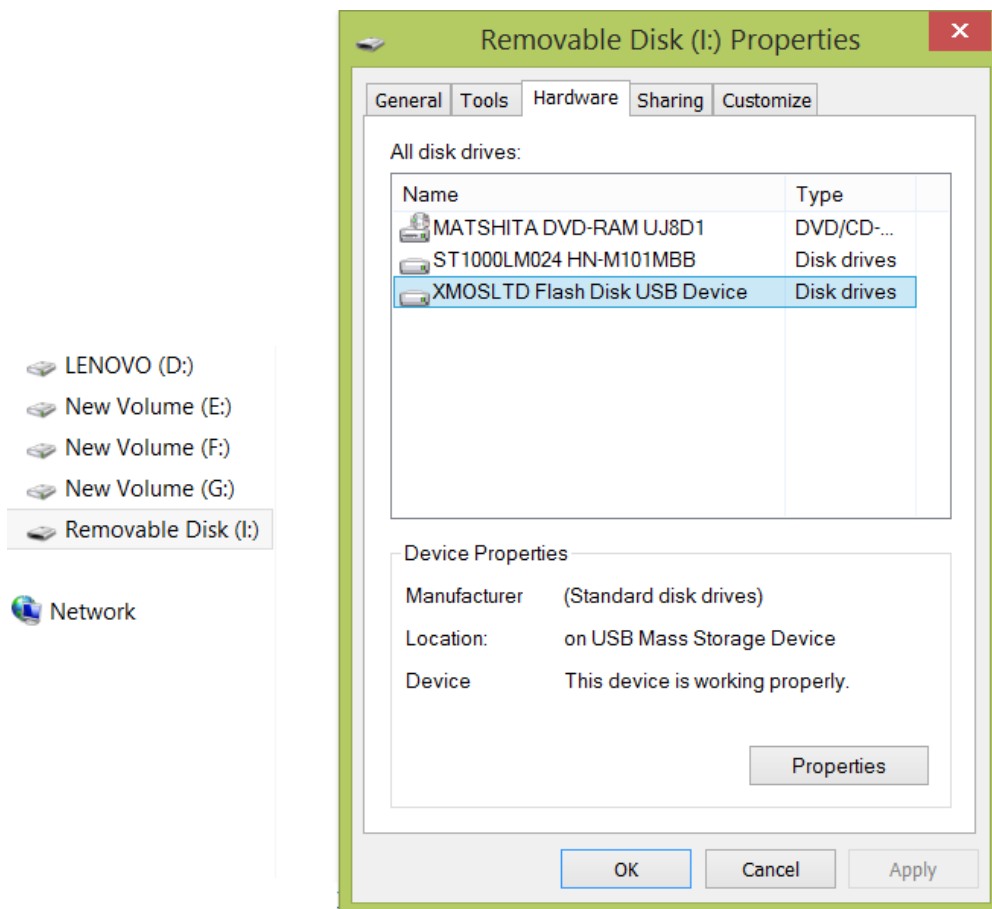


Figure 9: USB Mass Storage Device detecting as Removable Disk(I:) (On Windows 7)

- You could see the successful installation of driver after enumeration is done.

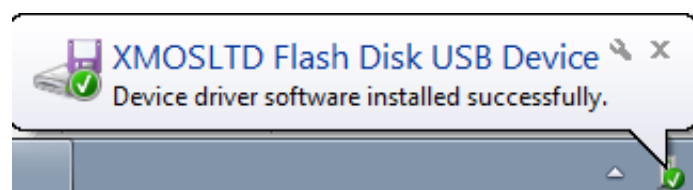


Figure 10: XMOS LTD Flash Disk - Device Driver software installed successfully



## C.2 OSX Host

- There is no specific driver installation is required. Mac will automatically detect and you should be able to get the below message screen once enumeration is complete.

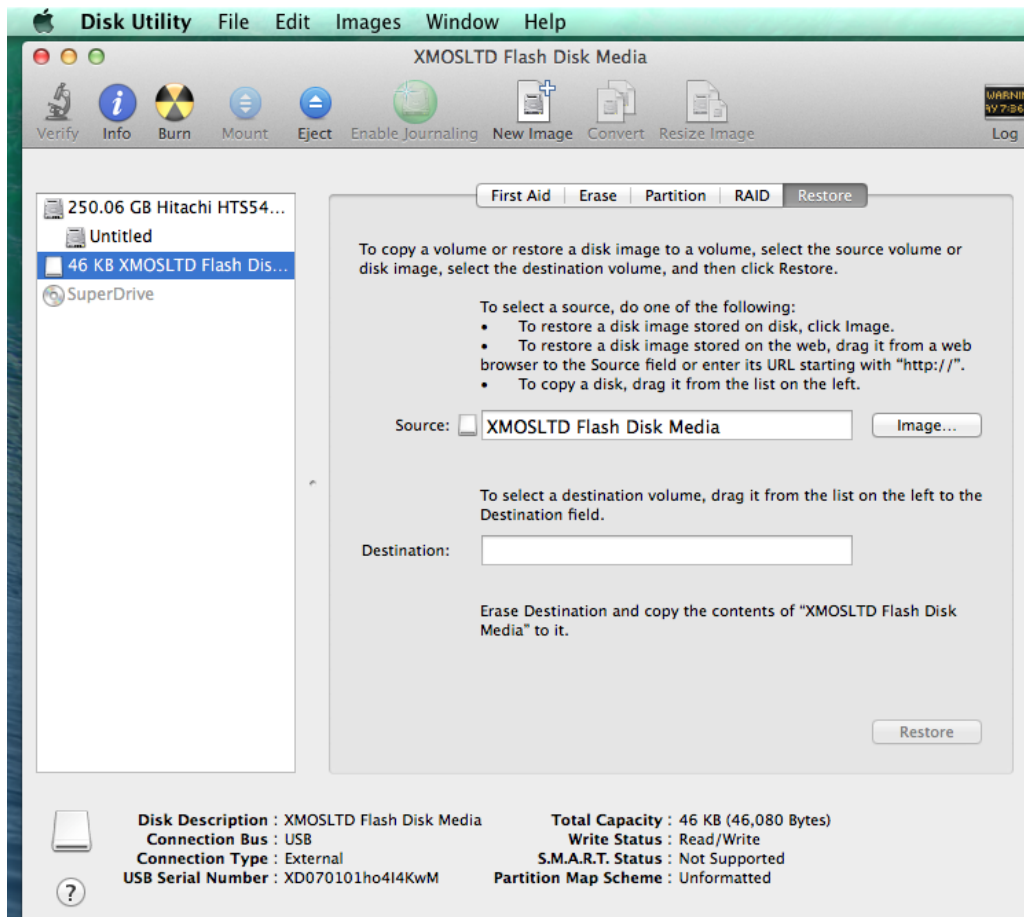


Figure 11: USB Mass Storage Device detection (On Mac)

## APPENDIX D - References

XMOS Tools User Guide

<http://www.xmos.com/published/xtimecomposer-user-guide>

XMOS xCORE Programming Guide

<http://www.xmos.com/published/xmos-programming-guide>

XMOS xCORE-USB Device Library:

<http://www.xmos.com/published/xuddg>

XMOS USB Device Design Guide:

<http://www.xmos.com/published/xmos-usb-device-design-guide>

USB Mass Storage Class Bulk-Only Transport

[http://www.usb.org/developers/docs/devclass\\_docs/usbmassbulk\\_10.pdf](http://www.usb.org/developers/docs/devclass_docs/usbmassbulk_10.pdf)

USB 2.0 Specification

[http://www.usb.org/developers/docs/usb20\\_docs/usb\\_20\\_081114.zip](http://www.usb.org/developers/docs/usb20_docs/usb_20_081114.zip)

USB Mass Storage Specification For Bootability

[http://www.usb.org/developers/docs/devclass\\_docs/usb\\_msc\\_boot\\_1.0.pdf](http://www.usb.org/developers/docs/devclass_docs/usb_msc_boot_1.0.pdf)

SCSI Command

[http://en.wikipedia.org/wiki/SCSI\\_command](http://en.wikipedia.org/wiki/SCSI_command)

SCSI Commands Reference Manual (Seagate)

<http://www.seagate.com/staticfiles/support/disc/manuals/Interface%20manuals/100293068c.pdf>

USB Mass Storage Device Specification Overview<sup>5</sup>.

---

<sup>5</sup>[http://www.usb.org/developers/docs/devclass\\_docs/Mass\\_Storage\\_Specification\\_Overview\\_v1.4\\_2-19-2010.pdf](http://www.usb.org/developers/docs/devclass_docs/Mass_Storage_Specification_Overview_v1.4_2-19-2010.pdf)

## APPENDIX E - Full Source Code listing

### E.1 Source Code for main.xc

```
// Copyright (c) 2016, XMOS Ltd, All rights reserved

#include <xscope.h>
#include "usb.h"
#include "debug_print.h"
#include "mass_storage.h"
#include "print.h"

/* USB Endpoint Defines */
#define XUD_EP_COUNT_OUT 2 //Includes EPO (1 out EPO + Mass Storage data output EP)
#define XUD_EP_COUNT_IN 2 //Includes EPO (1 in EPO + Mass Storage data input EP)

/* xSCOPE Setup Function */
#if (USE_XSCOPE == 1)
void xscope_user_init(void) {
    xscope_register(0, 0, "", 0, "");
    xscope_config_io(XSCOPE_IO_BASIC); /* Enable fast printing over links */
}
#endif

/* Prototype for Endpoint0 function in endpoint0.xc */
void Endpoint0(chanend c_ep0_out, chanend c_ep0_in);

/* The main function runs three cores: the XUD manager, Endpoint 0, and a mass storage endpoint. An array of
channels is used for both IN and OUT endpoints */
int main()
{
    chan c_ep_out[XUD_EP_COUNT_OUT], c_ep_in[XUD_EP_COUNT_IN];

    par
    {
        on USB_TILE: xud(c_ep_out, XUD_EP_COUNT_OUT, c_ep_in, XUD_EP_COUNT_IN,
            null, XUD_SPEED_HS, XUD_PWR_SELF);

        on USB_TILE: Endpoint0(c_ep_out[0], c_ep_in[0]);

        on USB_TILE: massStorageClass(c_ep_out[1], c_ep_in[1], 0);
    }
    return 0;
}
```

### E.2 Source Code for endpoint0.xc

```
// Copyright (c) 2016, XMOS Ltd, All rights reserved

/*
 * @brief Implements endpoint zero for an example Mass Storage class device.
 */

#include <xs1.h>
#include <string.h>
#include <xscope.h>

#include "usb.h"
#include "debug_print.h"
#include "print.h"

/* USB Device ID Defines */
#define BCD_DEVICE 0x0010
#define VENDOR_ID 0x20B1
#define PRODUCT_ID 0x10BA

/* USB Mass Storage Interface Subclass Definition */
#define USB_MASS_STORAGE_SUBCLASS 0x06 /* SCSI transparent command set */

/* USB Mass Storage interface protocol */
```

```

#define USB_MASS_STORAGE_PROTOCOL 0x50 /* USB Mass Storage Class Bulk-Only (BBB) Transport */

/* USB Mass Storage Request Code */
#define USB_MASS_STORAGE_RESET 0xFF /* Bulk-Only Mass Storage Reset */
#define USB_MASS_STORAGE_GML 0xFE /* Get Max LUN (GML) */

/* USB Device Descriptor */
static unsigned char devDesc[] =
{
    0x12, /* 0 bLength */
    USB_DESCRIPTOR_DEVICE, /* 1 bDescriptorType */
    0x00, /* 2 bcdUSB version */
    0x02, /* 3 bcdUSB version */
    0x00, /* 4 bDeviceClass - Specified by interface */
    0x00, /* 5 bDeviceSubClass - Specified by interface */
    0x00, /* 6 bDeviceProtocol - Specified by interface */
    0x40, /* 7 bMaxPacketSize for EPO - max = 64*/
    (VENDOR_ID & 0xFF), /* 8 idVendor */
    (VENDOR_ID >> 8), /* 9 idVendor */
    (PRODUCT_ID & 0xFF), /* 10 idProduct */
    (PRODUCT_ID >> 8), /* 11 idProduct */
    (BCD_DEVICE & 0xFF), /* 12 bcdDevice */
    (BCD_DEVICE >> 8), /* 13 bcdDevice */
    0x01, /* 14 iManufacturer - index of string */
    0x02, /* 15 iProduct - index of string */
    0x03, /* 16 iSerialNumber - index of string */
    0x01 /* 17 bNumConfigurations */
};

/* USB Configuration Descriptor */
static unsigned char cfgDesc[] = {
    0x09, /* 0 bLength */
    USB_DESCRIPTOR_CONFIGURATION, /* 1 bDescriptorType = configuration */
    0x20, 0x00, /* 2 wTotalLength of all descriptors */
    0x01, /* 4 bNumInterfaces */
    0x01, /* 5 bConfigurationValue */
    0x00, /* 6 iConfiguration - index of string */
    0x80, /* 7 bmAttributes - Self powered */
    0x50, /* 8 bMaxPower - 160mA */

    /* USB Bulk-Only Data Interface Descriptor */
    0x09, /* 0 bLength */
    USB_DESCRIPTOR_INTERFACE, /* 1 bDescriptorType */
    0x00, /* 2 bInterfaceNumber */
    0x00, /* 3 bAlternateSetting */
    0x02, /* 4: bNumEndpoints */
    USB_CLASS_MASS_STORAGE, /* 5: bInterfaceClass */
    USB_MASS_STORAGE_SUBCLASS, /* 6: bInterfaceSubClass */
    USB_MASS_STORAGE_PROTOCOL, /* 7: bInterfaceProtocol */
    0x00, /* 8 iInterface */

    /* Bulk-In Endpoint Descriptor */
    0x07, /* 0 bLength */
    USB_DESCRIPTOR_ENDPOINT, /* 1 bDescriptorType */
    0x81, /* 2 bEndpointAddress - EP1, IN */
    XUD_EPTYPE_BUL, /* 3 bmAttributes */
    0x00, /* 4 wMaxPacketSize - Low */
    0x02, /* 5 wMaxPacketSize - High */
    0x00, /* 6 bInterval */

    /* Bulk-Out Endpoint Descriptor */
    0x07, /* 0 bLength */
    USB_DESCRIPTOR_ENDPOINT, /* 1 bDescriptorType */
    0x01, /* 2 bEndpointAddress - EP1, OUT */
    XUD_EPTYPE_BUL, /* 3 bmAttributes */
    0x00, /* 4 wMaxPacketSize - Low */
    0x02, /* 5 wMaxPacketSize - High */
    0x00, /* 6 bInterval */
};

unsafe{
    /* String table - unsafe as accessed via shared memory */
    static char * unsafeStringDescriptors[] =
    {
        "\x09\x04", /* Language ID string (US English)

```

```

"XMOS",           // iManufacturer
"xMASSstorage",  // iProduct
"XD070101ho4I4KwM", // iSerial Number
};
}

/* Mass Storage Class Requests */
int MassStorageEndpoint0Requests(XUD_ep ep0_out, XUD_ep ep0_in, USB_SetupPacket_t sp)
{
  unsigned char buffer[1] = {0};

  switch(sp.bRequest) {

    case USB_MASS_STORAGE_RESET:
      XUD_ResetEpStateByAddr(1);
      return XUD_RES_RST; // This request is used to reset the mass storage device
      break;

    case USB_MASS_STORAGE_GML:
      return XUD_DoGetRequest(ep0_out, ep0_in, buffer, 1, sp.wLength);
      break;

    default:
      debug_printf("MassStorageEndpoint0Requests @ default : 0x%x\n", sp.bRequest);
      break;
  }

  return XUD_RES_ERR;
}

/* Endpoint 0 Task */
void Endpoint0(chanend chan_ep0_out, chanend chan_ep0_in)
{
  USB_SetupPacket_t sp;
  unsigned bmRequestType;
  XUD_BusSpeed_t usbBusSpeed;

  XUD_ep ep0_out = XUD_InitEp(chan_ep0_out, XUD_EPTYPE_CTL | XUD_STATUS_ENABLE);
  XUD_ep ep0_in = XUD_InitEp(chan_ep0_in, XUD_EPTYPE_CTL | XUD_STATUS_ENABLE);

  while(1)
  {
    /* Returns XUD_RES_OKAY on success */
    XUD_Result_t result = USB_GetSetupPacket(ep0_out, ep0_in, sp);

    if(result == XUD_RES_OKAY)
    {
      /* Set result to ERR, we expect it to get set to OKAY if a request is handled */
      result = XUD_RES_ERR;

      /* Stick bmRequest type back together for an easier parse... */
      bmRequestType = (sp.bmRequestType.Direction<<7) | (sp.bmRequestType.Type<<5) |
        (sp.bmRequestType.Recipient);

      if ((bmRequestType == USB_BMREQ_H2D_STANDARD_DEV) && (sp.bRequest == USB_CLEAR_FEATURE)) {
        // Host has set device address, value contained in sp.wValue
      }

      /* Handle specific requests first */
      switch(bmRequestType) {
        /* Direction: Device-to-host and Host-to-device
         * Type: Class
         * Recipient: Interface
         */
        case USB_BMREQ_H2D_CLASS_INT:
        case USB_BMREQ_D2H_CLASS_INT:

          /* Inspect for mass storage interface num */
          if(sp.wIndex == 0) {
            /* Returns XUD_RES_OKAY if handled,
             * XUD_RES_ERR if not handled,
             * XUD_RES_RST for bus reset */
            result = MassStorageEndpoint0Requests(ep0_out, ep0_in, sp);
          }
          break;
      }
    }
  }
}

```

```

    }
  }

  /* If we haven't handled the request above then do standard enumeration requests */
  if(result == XUD_RES_ERR) {
    /* Returns XUD_RES_OKAY if handled okay,
     * XUD_RES_ERR if request was not handled (i.e. STALLed),
     * XUD_RES_RST if USB Reset */
    result = USB_StandardRequests(ep0_out, ep0_in, devDesc,
      sizeof(devDesc), cfgDesc, sizeof(cfgDesc),
      null, 0, null, 0,
      stringDescriptors, sizeof(stringDescriptors)/sizeof(stringDescriptors[0]),
      sp, usbBusSpeed);
  }

  /* USB bus reset detected, reset EP and get new bus speed */
  if(result == XUD_RES_RST) {
    usbBusSpeed = XUD_ResetEndpoint(ep0_out, ep0_in);
  }
}
}
}

```

### E.3 Source Code for mass\_storage.xc

```

// Copyright (c) 2016, XMOS Ltd, All rights reserved

#include <xs1.h>
#include <platform.h>
#include <xclib.h>
#include <string.h>
#include <xassert.h>
#include "xud.h"
#include "mass_storage.h"
#include "debug_print.h"
//Flash_Functions_start
#include "flashlib.h"

on tile[0]: fl_SPIPorts spiPort = {
  XS1_PORT_1A,
  XS1_PORT_1B,
  XS1_PORT_1C,
  XS1_PORT_1D,
  XS1_CLKBLK_1
};

fl_DeviceSpec flashSpec[1] = {FL_DEVICE_NUMONYX_M25P16}; // Equivalent of Micron M25P16

int pagesPerBlock_g = 0;
int bytesPerPage_g = 0;
unsigned char pageBuffer_g[MASS_STORAGE_BLOCKLENGTH];

void massStorageInit() {
  fl_connectToDevice(spiPort, flashSpec, 1);
  fl_setBootPartitionSize(FLASH_PARTITION_SIZE);
  bytesPerPage_g = fl_getPageSize();
  pagesPerBlock_g = (MASS_STORAGE_BLOCKLENGTH / bytesPerPage_g);
}

int massStorageWrite(unsigned int blockNr, unsigned char buffer[]) {
  for(int i = 0; i < pagesPerBlock_g; i++) {
    for(int j = 0; j < bytesPerPage_g; j++) {
      pageBuffer_g[j] = buffer[i * bytesPerPage_g + j];
    }
    fl_writeDataPage(blockNr * pagesPerBlock_g + i, buffer);
  }
  return 0;
}

int massStorageRead(unsigned int blockNr, unsigned char buffer[]) {
  for(int i = 0; i < pagesPerBlock_g; i++) {
    fl_readDataPage(blockNr * pagesPerBlock_g + i, pageBuffer_g);
    for(int j = 0; j < bytesPerPage_g; j++) {
      buffer[i * bytesPerPage_g + j] = pageBuffer_g[j];
    }
  }
}

```

```

    }
  }
  return 0;
}

int massStorageSize() {
#ifdef DETECT_AS_FLOPPY
  return FLOPPY_DISK_SIZE;
#else
  int x = fl_getNumDataPages();
  return x / pagesPerBlock_g;
#endif
}
//Flash_Functions_end
static unsigned char inquiryAnswer[36] = {
0x00, // Peripheral Device Type (PDT) - SBC Direct-access device
0x80, // Removable Medium Bit is Set
0x02, // Version
0x02, // Obsolete[7:6],NORMACA[5],HISUP[4],Response Data Format[3:0]
0x1f, // Additional Length
0x73, // SCCS[7],ACC[6],TPGS[5:4],3PC[3],Reserved[2:1],PROTECT[0]
0x6d, // BQUE[7],ENC SERV[6],VS[5],MULTIP[4],MCHNGR[3],Obsolete[2:1],ADDR16[0]
0x69, // Obsolete[7:6],WBUS116[5],SYNC[4],LINKED[3],Obsolete[2],CMDQUE[1],VS[0]
'X', 'M', '0', 'S', 'L', 'T', 'D', 0, // Vendor Identification
'F', 'l', 'a', 's', 'h', ' ', 'D', 'i', 's', 'k', 0, ' ', ' ', ' ', ' ', ' ', // Product Identification
'0', '.', '1', '0' // Product Revision Level
};

static unsigned char modeSenseAnswer[4] = {
  0x04, 0x00, 0x10, 0x00
};

//Reference: http://www.usb.org/developers/docs/devclass_docs/usb_msc_boot_1.0.pdf
static unsigned char requestSenseAnswer[18] = {
  0x70, // Error Code
  0x00, // Segment Number (Reserved)
  0x02, // ILI, Sense Key
  0x00, 0x00, 0x00, 0x00, // Information
  0x0A, // Additional Sense Length (n-7), i.e. 17-7
  0x00, 0x00, 0x00, 0x00, // Command Specific Information
  0x3A, // Additional Sense Code
  0x00, // Additional Sense Qualifier (optional)
  0x00, 0x00, 0x00, 0x00 // Reserved
};

static unsigned char blockBuffer[MASS_STORAGE_BLOCKLENGTH];

/* This function receives the mass storage endpoint transfers from the host */
void massStorageClass(chanend chan_ep1_out,chanend chan_ep1_in, int writeProtect)
{
  unsigned char commandBlock[CBW_SHORT_PACKET_SIZE];
  unsigned char commandStatus[CSW_SHORT_PACKET_SIZE];
  unsigned host_transfer_length = 0;
  int readCapacity[8];
  int readLength, readAddress;
  int dCBWSignature = 0, bCBWDataTransferLength = 0;
  int bmCBWFlags = 0, bCBWLUN = 0, bCBWCBLength = 0;
  int Operation_Code = 0;
  XUD_Result_t result;
  int ready = 1;

  debug_printf("USB Mass Storage class demo started\n");

  /* Load some default CSW to reduce response time delay */
  memset(commandStatus,0,CSW_SHORT_PACKET_SIZE);
  /* Signature helps identify this data packet as a CSW */
  (commandStatus, int[])[0] = byterevert(CSW_SIGNATURE);

  /* Initialise the XUD endpoints */
  XUD_ep ep1_out = XUD_InitEp(chan_ep1_out, XUD_EPTYPE_BUL);
  XUD_ep ep1_in = XUD_InitEp(chan_ep1_in, XUD_EPTYPE_BUL);

#ifdef !DETECT_AS_FLOPPY
  massStorageInit();
#endif
}

```

```

while(1)
{
  unsigned char bCSWStatus = CSW_STATUS_CMD_PASSED;
  // Get Command Block Wrapper (CBW)
  if(XUD_RES_OKAY == (result = XUD_GetBuffer(ep1_out, (commandBlock, char[CBW_SHORT_PACKET_SIZE]),
  ↪ host_transfer_length)) )
  {
    /* The CBW shall start on a packet boundary and shall end as a short packet
    * with exactly 31 (0x1F) bytes transferred
    */
    assert(host_transfer_length == CBW_SHORT_PACKET_SIZE);
    /* verify Signature - that helps identify this packet as a CBW */
    dCBWSignature = commandBlock[0] | commandBlock[1] << 8 |
                    commandBlock[2] << 16 | commandBlock[3] << 24;
    assert(dCBWSignature == CBW_SIGNATURE);

    bCBWDataTransferLength = commandBlock[8] | commandBlock[9]<<8 |
                            commandBlock[10] << 16 | commandBlock[11] << 24;

    bmCBWFlags = commandBlock[12]; bCBWLUN = (commandBlock[13] & 0x0F);
    assert(bCBWCBLength = (commandBlock[14] & 0x1F) <= 16);
    Operation_Code = commandBlock[15];

    switch(Operation_Code)
    {
      case TEST_UNIT_READY_CMD: // Test unit ready:
        bCSWStatus = ready ? CSW_STATUS_CMD_PASSED : CSW_STATUS_CMD_FAILED;
        break;

      case REQUEST_SENSE_CMD: // Request sense
        requestSenseAnswer[2] = ready ? STATUS_GOOD : STATUS_CHECK_CONDITION;
        result = XUD_SetBuffer(ep1_in, requestSenseAnswer, sizeof(requestSenseAnswer));
        break;

      case INQUIRY_CMD: // Inquiry
        result = XUD_SetBuffer(ep1_in, inquiryAnswer, sizeof(inquiryAnswer));
        break;

      case START_STOP_CMD: // start/stop
        ready = ((commandBlock[19] >> 1) & 1) == 0;
        break;

      case MODE_SENSE_6_CMD: // Mode sense (6)
      case MODE_SENSE_10_CMD: // Mode sense (10) // For Mac OSX
        if (writeProtect) modeSenseAnswer[2] |= 0x80;

        result = XUD_SetBuffer(ep1_in, modeSenseAnswer, sizeof(modeSenseAnswer));
        break;

      case MEDIUM_REMOVAL_CMD: // Medium removal
        break;

      case RECEIVE_DIAGNOSTIC_RESULT_CMD:
        memset(readCapacity, 0x0000, sizeof(readCapacity));
        result = XUD_SetBuffer(ep1_in, (readCapacity, unsigned char[8]), 32);
        break;

      case READ_FORMAT_CAPACITY_CMD: // Read Format capacity (UFI Command Spec)
        readCapacity[0] = byterev(8);
        readCapacity[1] = byterev(massStorageSize());
        readCapacity[2] = byterev(MASS_STORAGE_BLOCKLENGTH) | (DETECT_AS_FLOPPY ? NO_CARTRIDGE_IN_DRIVE :
        ↪ FORMATTED_MEDIA);
        result = XUD_SetBuffer(ep1_in, (readCapacity, unsigned char[8]), 12);
        break;

      case READ_CAPACITY_CMD: // Read capacity
        readCapacity[0] = byterev(massStorageSize()-1);
        readCapacity[1] = byterev(MASS_STORAGE_BLOCKLENGTH);
        result = XUD_SetBuffer(ep1_in, (readCapacity, unsigned char[8]), 8);
        break;

      case READ_CAPACITY_16_CMD:
        memset(readCapacity, 0x0000, sizeof(readCapacity));

```



```

    readCapacity[1] = bytrev(massStorageSize()-1);
    readCapacity[2] = bytrev(MASS_STORAGE_BLOCKLENGTH);
    result = XUD_SetBuffer(ep1_in, (readCapacity, unsigned char[8]), 32);
    break;

case READ_10_CMD: // Read (10)
    readLength = commandBlock[22] << 8 | commandBlock[23];
    readAddress = commandBlock[17] << 24 | commandBlock[18] << 16 |
        commandBlock[19] << 8 | commandBlock[20];
    for(int i = 0; i < readLength ; i++) {
        bCSWStatus |= massStorageRead(readAddress, blockBuffer);
        result = XUD_SetBuffer(ep1_in, blockBuffer, MASS_STORAGE_BLOCKLENGTH);
        readAddress++; }
    break;

case WRITE_10_CMD: // Write
    readLength = commandBlock[22] << 8 | commandBlock[23];
    readAddress = commandBlock[17] << 24 | commandBlock[18] << 16 |
        commandBlock[19] << 8 | commandBlock[20];
    for(int i = 0; i < readLength ; i++) {
        result = XUD_GetBuffer(ep1_out, (blockBuffer, char[128 * 4]),host_transfer_length);
        bCSWStatus |= massStorageWrite(readAddress, blockBuffer);
        readAddress++; }
    break;

default:
    debug_printf("Invalid Operation Code Received : 0x%x\n",Operation_Code);
    bCSWStatus = CSW_STATUS_CMD_FAILED;
    break;
}
}

/* Check for result, if it is found as XUD_RES_RST, then reset Endpoints */
if(result == XUD_RES_RST) {
    XUD_ResetEndpoint(ep1_out,ep1_in);
    break;
}

/* Setup Command Status Wrapper (CSW). The CSW shall start on a packet boundry
 * and shall end as a short packet with exactly 13 (0x0D) bytes transferred */
/* The device shall echo the contents of dCBWTag back to the host in the dCSWTag */
commandStatus[4] = commandBlock[4];
commandStatus[5] = commandBlock[5];
commandStatus[6] = commandBlock[6];
commandStatus[7] = commandBlock[7];
commandStatus[12] = bCSWStatus;

if(XUD_RES_RST == XUD_SetBuffer(ep1_in, commandStatus, CSW_SHORT_PACKET_SIZE))
    XUD_ResetEndpoint(ep1_out,ep1_in);
} //while(1)
} // END of massStorageClass

```