# AN00125: USB Mass Storage Device Class

Publication Date: 2026/5/27
Document Number: XM-006293-AN v3.1.3

IN THIS DOCUMENT

1 Overview . . . 2
1.1 Features . . . 2
2 USB Mass Storage Device Class application note . . . 3
2.1 CMakeLists.txt additions for this application . . . 3
2.2 Declaring resource and setting up the USB components . . . 4
2.3 The application main() function . . . 4
2.4 Configuring the USB Device ID . . . 4
2.5 USB Mass storage Class specific defines . . . 5
2.6 USB Device Descriptor . . . 5
2.7 USB Configuration Descriptor . . . 5
2.8 USB string descriptor . . . 6
2.9 USB Mass storage Class requests . . . 6
2.10 USB Mass storage Class Endpoint0 . . . 7
2.11 Receiving storage data from the host . . . 7
2.12 SCSI Command Implementation . . . 10
2.13 Serial Flash Functions . . . 10
3 Demo Hardware Setup . . . 12
4 Example application . . . 13
4.1 Building the example . . . 13
4.2 Running the example . . . 13
5 References . . . 14

This application note shows how to create a USB device compliant to the standard USB mass storage device class on an *xmos xcore.ai* device.

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. flash-based storage). This example application uses the on-board serial flash as its memory device.

This application note is designed to run on an *XMOS xcore-200* or *xcore.ai* series device. The example code provided with the application has been implemented and tested on the *XK-EVK-XU316* board but there is no dependency on this board and it can be modified to run on any development board which uses an *xcore-200* or *xcore.ai* series device.

**Note**

- 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 the full API listing of the XMOS USB Device (XUD) Library please see the XMOS USB Device (XUD) Library.

# 1 Overview

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.

*Block diagram of USB Mass storage application* provides an overview of the system.

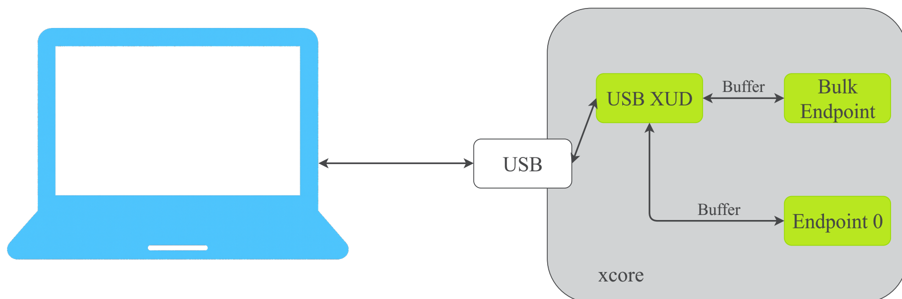


Fig. 1: Block diagram of USB Mass storage application

## 1.1 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 3 MB. The application does the following

- Enumerates as Mass Storage device
- Displays as " **XMOS DISK** "

The device will have a FAT12 filesystem and 3 MBytes of memory available.

> **Note**
>
> Read and write operations may be slow as this is just an example.

# 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* device.

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.

*Task diagram of USB mass storage application* shows the task and communication structure for this application note.

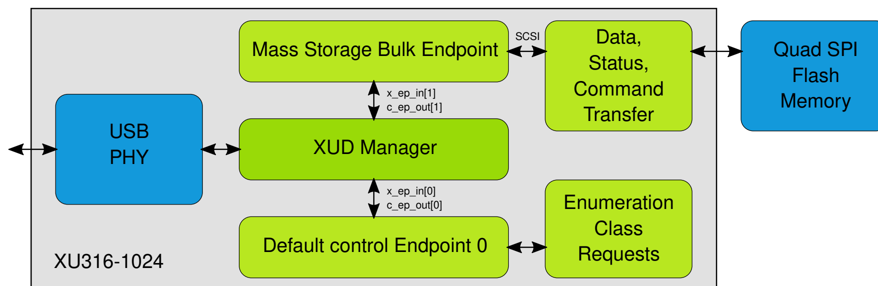


Fig. 2: Task diagram of USB mass storage application

## 2.1 CMakeLists.txt additions for this application

To start using the USB library, the user needs to add **lib_xud** to the *CMakeLists.txt*:

```
set (APP_DEPENDENT_MODULES "lib_xud")
```

After *CMakeLists.txt* has been modified, the USB APIs can be acessed via the **xud_device.h** header file:

```
#include <xud_device.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 and arrays 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)

XUD_EpType epTypeTableOut[XUD_EP_COUNT_OUT] = {XUD_EPTYPE_CTL | XUD_STATUS_ENABLE, XUD_EPTYPE_BUL};
XUD_EpType epTypeTableIn[XUD_EP_COUNT_IN] =   {XUD_EPTYPE_CTL | XUD_STATUS_ENABLE, XUD_EPTYPE_BUL};
```

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_Main(c_ep_out, XUD_EP_COUNT_OUT, c_ep_in, XUD_EP_COUNT_IN,
                              null, epTypeTableOut, epTypeTableIn,
                              XUD_SPEED_HS, XUD_PWR_BUS);

        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 a few things can be noted:

- The *par* statement starts three separate tasks in parallel
- There is a task to configure and execute the USB library: *XUD_Main()*
- 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
- 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_Main()

## 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* device.

```
/* 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_DESCTYPE_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 the user 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 */
```

(continues on next page)

(continued from previous page)

```
    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 code the user 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_OKAY; // 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);

            /* 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;

                default:
                    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 (*Command/Data/Status Flow*). This is contained within the function *massStorageClass()* which is shown in next page.

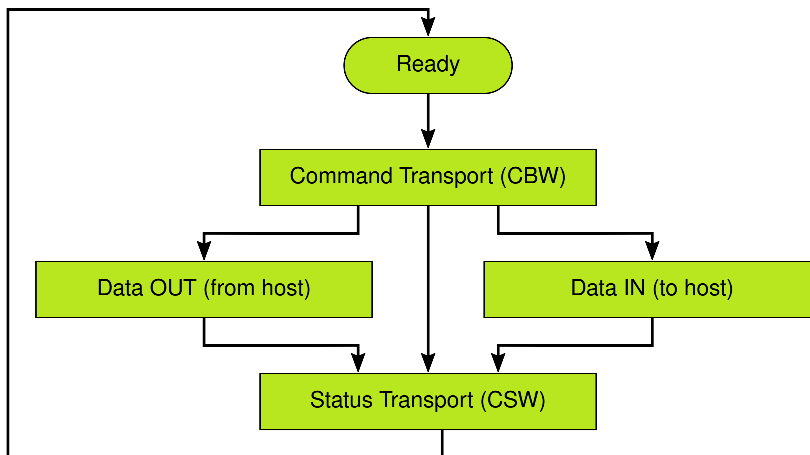


Fig. 3: Command/Data/Status Flow

```
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 writeLength, writeAddress;
    int dCBWSignature = 0, bCBWDataTransferLength = 0;
    int bmCBWFlags = 0, bCBWLUN = 0, bCBWCBLength = 0;
    int Operation_Code = 0;
    XUD_Result_t result;
    int ready = 1;
    unsigned char blockBuffer[MASS_STORAGE_BLOCKLENGTH];
    const int cswSignature = CSW_SIGNATURE;

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

    /* Init command status packet - first 4 bytes are the Command Status Wrapper signature */
    memset(commandStatus, 0, CSW_SHORT_PACKET_SIZE);
    memcpy(commandStatus, &cswSignature, sizeof(int));

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

#if !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);
            bCBWCBLength = (commandBlock[14] & 0x1F);
            assert(bCBWCBLength <= 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);
```

(continues on next page)

(continued from previous page)

```
                    break;

                case READ_CAPACITY_CMD: // Read capacity
                    /* -1 since report last block address */
                    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));
                    /* -1 since report last block address */
                    readCapacity[1] = byterev(massStorageSize()-1);
                    readCapacity[2] = byterev(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
                    writeLength = commandBlock[22] << 8 | commandBlock[23];
                    writeAddress = commandBlock[17] << 24 | commandBlock[18] << 16 | commandBlock[19] << 8 |
↪commandBlock[20];
                    for(int i = 0; i < writeLength ; i++)
                    {
                        result = XUD_GetBuffer(ep1_out, blockBuffer, host_transfer_length);
                        bCSWStatus |= massStorageWrite(writeAddress, blockBuffer);
                        writeAddress++;
                    }
                    break;

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

        XUD_SetBuffer(ep1_in, commandStatus, CSW_SHORT_PACKET_SIZE);

    }
} // END of massStorageClass
```

The following points can be observed from the code above:

- 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 reponse 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 Revison 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 occured.
- **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.

## 2.13 Serial Flash Functions

For accessing the on-board serial flash, the user will need to add the flash library **lflash** in *CMakeLists* as one of the *APP_COMPILER_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.

```
#include "quadflash.h"

on tile[0]: fl_QSPIPorts spiPort =
{
    PORT_SQI_CS,
    PORT_SQI_SCLK,
    PORT_SQI_SIO,
    XS1_CLKBLK_1
};

void massStorageInit()
{
```

(continues on next page)

(continued from previous page)

```
    /* Connect to the QSPI flash */
    fl_connect(spiPort);

    /* Run the SPI clock a faster than default */
    fl_dividerOverride(2);
}

int massStorageWrite(unsigned int blockNr, unsigned char buffer[])
{
    unsigned char sectorBuffer[4096];

    int fail = fl_writeData(blockNr * MASS_STORAGE_BLOCKLENGTH, MASS_STORAGE_BLOCKLENGTH, buffer, sectorBuffer);

    return fail != 0;
}

int massStorageRead(unsigned int blockNr, unsigned char buffer[])
{
    int fail = fl_readData(blockNr * MASS_STORAGE_BLOCKLENGTH, MASS_STORAGE_BLOCKLENGTH, buffer);

    return fail != 0;
}

int massStorageSize() {
#if DETECT_AS_FLOPPY
    return FLOPPY_DISK_SIZE;
#else
    int sizeInBytes = fl_getDataPartitionSize();

    /* Return number of blocks */
    return sizeInBytes / MASS_STORAGE_BLOCKLENGTH;
#endif
}
```

## 3 Demo Hardware Setup

To run the demo, connect the *XK-EVK-XU316* **DEBUG** and **USB** (*XMOS XK-EVK-XU316 Board*) recepticals to separate USB connectors on the development PC.

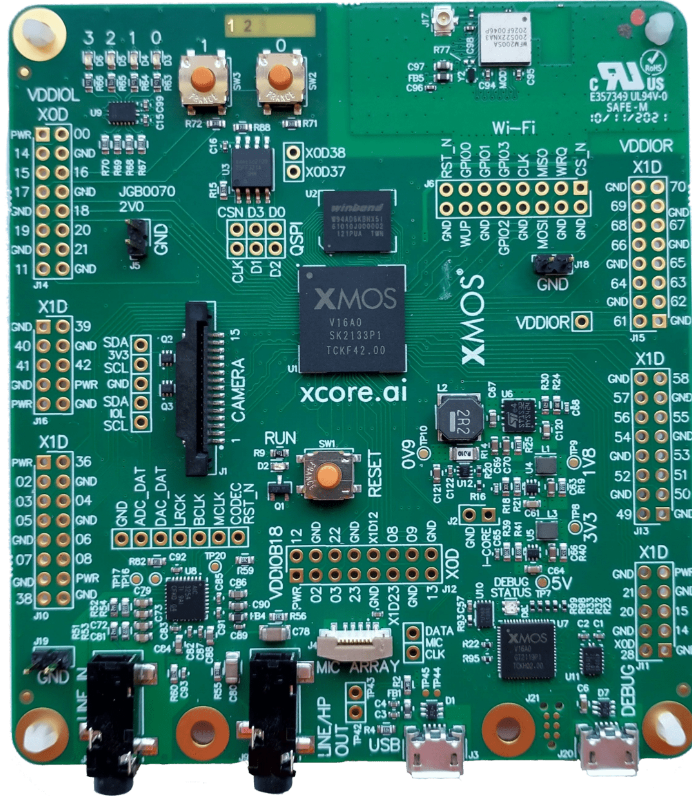


Fig. 4: XMOS XK-EVK-XU316 Board

# 4 Example application

## 4.1 Building the example

This section assumes that the XMOS XTC Tools have been downloaded and installed. The required version is specified in the accompanying `README`.

Installation instructions can be found here.

Special attention should be paid to the section on Installation of Required Third-Party Tools.

The application is built using the xcommon-cmake build system, which is provided with the XTC tools and is based on CMake.

The `an0xxxx` software ZIP package should be downloaded and extracted to a chosen working directory.

To configure the build, the following commands should be run from an XTC command prompt:

```
cd an00125
cd app_an00125
cmake -G "Unix Makefiles" -B build
```

All required dependencies are included in the software package. If any dependencies are missing, they will be retrieved automatically during this step.

The application binaries should then be built using `xmake`:

```
xmake -j -C build
```

Binary artifacts (.xe files) will be generated under the appropriate subdirectories of the `app_an0xxxx/bin` directory — one for each supported build configuration.

For subsequent builds, the `cmake` step may be omitted. If `CMakeLists.txt` or other build files are modified, `cmake` will be re-run automatically by `xmake` as needed.

## 4.2 Running the example

Once the application is built, the user will need to prepare the flash device by creating a data partition. From a XTC command prompt run the following command from the *an00125/app_an00125* directory.

```
xflash --boot-partition-size=0x20000 --data disk.img --target=XK-EVK-XU316 ./bin/app_mass_storage.xe
```

The disk.img file is a 3MB FAT12-formatted disk image created for use as a virtual mass storage device. It can be mounted and written to like a physical disk, allowing files such as hello.txt to be included for testing or emulation purposes.

After flashing the device, **XMOS DISK** device will appear on the host computer.

# 5 References


- XMOS XTC Tools Installation Guide
- XMOS XTC Tools User Guide
- XMOS USB Device Library
- USB Mass Storage Class Bulk-Only Transport
- USB 2.0 Specification
- USB Mass Storage Specification For Bootability
- SCSI Command
- SCSI Commands Reference Manual (Seagate)
- USB Mass Storage Device Specification Overview

Copyright © 2026, All Rights Reserved.

XMOS Ltd. is the owner or licensee of this design, code, or Information (collectively, the "Information") and is providing it to you "AS IS" with no warranty of any kind, express or implied and shall have no liability in relation to its use. XMOS Ltd makes no representation that the Information, or any particular implementation thereof, is or will be free from any claims of infringement and again, shall have no liability in relation to any such claims.

XMOS, XCORE, VocalFusion and the XMOS logo are registered trademarks of XMOS Ltd. in the United Kingdom and other countries and may not be used without written permission. Company and product names mentioned in this document are the trademarks or registered trademarks of their respective owners.