Use SPI Flash as USB MSC Device

Example Code Introduction for 32-bit NuMicro® Family

Information

Application	The microcontroller plays a role of USB mass storage device and uses SPI Flash as a storage medium.
BSP Version	NUC123Series_BSP_CMSIS_v3.01.001
Hardware	NuTiny-EVB-123-LQFP64 V1.0

The information described in this document is the exclusive intellectual property of Nuvoton Technology Corporation and shall not be reproduced without permission from Nuvoton.

Nuvoton is providing this document only for reference purposes of NuMicro microcontroller based system design. Nuvoton assumes no responsibility for errors or omissions.

All data and specifications are subject to change without notice.

For additional information or questions, please contact: Nuvoton Technology Corporation.

www.nuvoton.com

1 Function Description

1.1 Introduction

The microcontroller NUC123 connects to the computer via a USB port, plays a role of a mass storage class device, and connects an external SPI Flash Memory as a medium. The user can use the microcontroller as a general external USB flash drive to write and read files on a personal computer.

1.2 Principle

In this experiment, the USB Mass Storage Device Class (MSC) Bulk-Only Transfer protocol is adopted. According to the protocol specification, the flow of data exchange between Host and Device can be divided into three stages, which are Command Transport (CBW), Data-Out / Data-In, and Status Transport (CSW). The state switching process is shown in Figure 1-1.

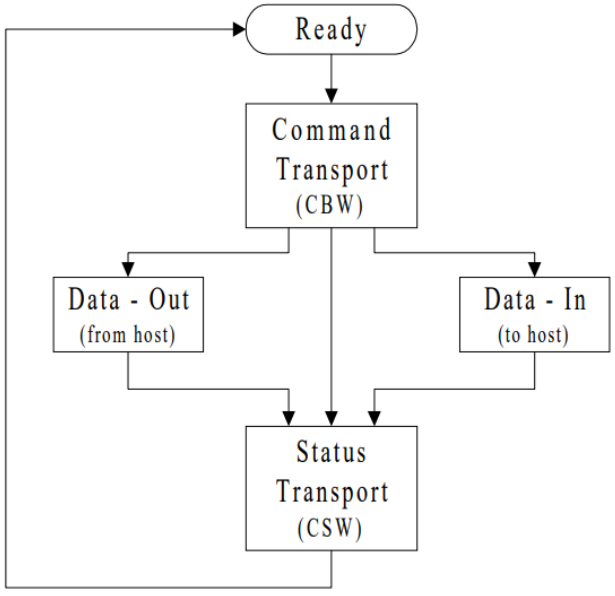


Figure 1-1 Mass Storage Class Command/Data/Status Protocol

The MSC also defines two exclusive category requests, namely Bulk-Only Mass Storage Reset and Get Max LUN commands. The Bulk-Only Mass Storage Reset command is used to reset the Device state to Ready, and to inform the device to receive the CBW command; the Get Max LUN is used to obtain the number of logical units (storage slots) supported by the Device. See the USB Mass Storage Class Bulk-Only Transport Specification for the detail information.

1.2.1 Command Transport (CBW)

The Command Transport contains a Transaction with the data direction OUT. The Host sends the data packet with specified command to the Device through the Bulk-Out Endpoint and encapsulates it in the format of Command Block Wrapper (CBW). The packet format is shown in Figure 1-2. In the CBW packet, the field CBWCB stores the Host's command to access or control the Device's storage medium. The type of the command determines whether the next state is Data-In or Data-Out, and the content of the transmitted data. The detailed description of the instructions can be found in the UFI Command Specification document.

Table 5.1 - Command Block Wrapper

bit / Byte	7	6	5	4	3	2	1	0
0-3	*dCBWSignature*							
4-7	*dCBWTag*							
8-11 (08h-0Bh)	*dCBWDataTransferLength*							
12 (0Ch)	*bmCBWFlags*							
13 (0Dh)	Reserved (0)				*bCBWLUN*			
14 (0Eh)	Reserved (0)				*bCBWCBLength*			
15-30 (0Fh-1Eh)	*CBWCB*							

Figure 1-2 Command Block Wrapper (CBW) packet format

1.2.2 Data-In / Data-Out

The behavior in the Data-In / Data-Out state is affected by the commands in the CBW. For example, if the UFI instruction in the CBWCB is WRITE, it means that the Host wants to store the file to the Device storage medium, and the state is switched to Data-Out. On the other hand, if the UFI command is READ, indicating that the Host reads the file from the storage of the Device, then the Device is required to send out the data packet to the Host. At this time, the state is switched to Data-In. The Data-In / Data-Out stage allows multiple Transactions, and it is affected by the dCBWDataTransferLength field in CBWCB. By the way, the storage medium of this experiment uses an external SPI Flash Memory. After receiving the READ or the WRTIE command, the microcontroller must drive the peripheral SPI to read and program the data to the specified sector of the SPI Flash.

1.2.3 Status Transport (CSW)

The purpose of the Status Transport is that the Device responds to the Host with the status of the CBW command execution. It contains a Transaction and the data direction is IN. The data packet at this stage is encapsulated in the format of Command Status Wrapper (CSW). The packet format is shown in Figure 1-3. Host can read the bCSWStatus field in CSW to determine whether Device has an error during the CBW command. The value of bCSWStatus is shown in Figure 1-4.

Table 5.2 - Command Status Wrapper

bit / Byte	7	6	5	4	3	2	1	0
0-3	*dCSWSignature*							
4-7	*dCSWTag*							
8-11 (8-Bh)	*dCSWDataResidue*							
12 (Ch)	*bCSWStatus*							

Figure 1-3 Command Status Wrapper (CSW) Packet Format

Table 5.3 - Command Block Status Values

Value	Description
00h	Command Passed ("good status")
01h	Command Failed
02h	Phase Error
03h and 04h	Reserved (Obsolete)
05h to FFh	Reserved

Figure 1-4 Values for bCSWStatus field

1.2.4 USB Endpoint Configuration

This example requires 4 USB endpoints to be configured as follows:

- Endpoint 0: Control-In
- Endpoint 1: Control-Out
- Endpoint 2: Bulk-In
- Endpoint 3: Bulk-Out

1.2.5 Serial Flash Memory

This experiment uses Winbond 25X16A SPI Flash as the medium of Mass Storage Device with a storage capacity of 2MB. The minimum program size is a Page (256 B), each 16 Pages can be group into a Sector (4K Byte), and each 256 Pages constitutes a Block (4 KB), where the Sector is the minimum erase unit. The control of SPI Flash is very simple. The microcontroller first issues instructions to the SPI Flash through the following command set, including Read, Program or Erase. Then both of the Master and the Slave will decide the content and the length of the subsequent data according to the type of the instruction.

INSTRUCTION NAME	BYTE 1 CODE	BYTE 2	BYTE 3	BYTE 4	BYTE 5	BYTE 6	N-BYTES
Write Enable	06h						
Write Disable	04h						
Read Status Register	05h	(S7–S0)(1)					(2)
Write Status Register	01h	S7–S0					
Read Data	03h	A23–A16	A15–A8	A7–A0	(D7–D0)	(Next byte)	continuous
Fast Read	0Bh	A23–A16	A15–A8	A7–A0	dummy	(D7–D0)	(Next Byte) continuous
Fast Read Dual Output	3Bh	A23–A16	A15–A8	A7–A0	dummy	I/O = (D6,D4,D2,D0) O = (D7,D5,D3,D1)	(one byte per 4 clocks, continuous)
Page Program	02h	A23–A16	A15–A8	A7–A0	(D7–D0)	(Next byte)	Up to 256 bytes
Block Erase (64KB)	D8h	A23–A16	A15–A8	A7–A0			
Sector Erase (4KB)	20h	A23–A16	A15–A8	A7–A0			
Chip Erase	C7h						
Power-down	B9h						
Release Power-down / Device ID	ABh	dummy	dummy	dummy	(ID7-ID0)(4)		
Manufacturer/ Device ID (3)	90h	dummy	dummy	00h	(M7-M0)	(ID7-ID0)	
JEDEC ID	9Fh	(M7-M0) Manufacturer	(ID15-ID8) Memory Type	(ID7-ID0) Capacity			

Figure 1-5 Winbond SPI Flash Command List

Figure 1-5 shows the Winbond 25X16A command list. Take the Page Program (02h) and Read Status Register (05h) instructions as an example, refer to the timing diagrams in Figure 1-6 and Figure 1-7. The Master first sends out a 1-byte instruction 02h, and continues the 24-bit data write bit. After the address and 256 Bytes of data, the command transfer of the Page Program is completed. The microcontroller can then use the Read Status Register (05h) command to ask for the status of the SPI Flash to check if the data has been written. Users can refer to the Winbond W25X16A Datasheet for detailed descriptions for each instruction.

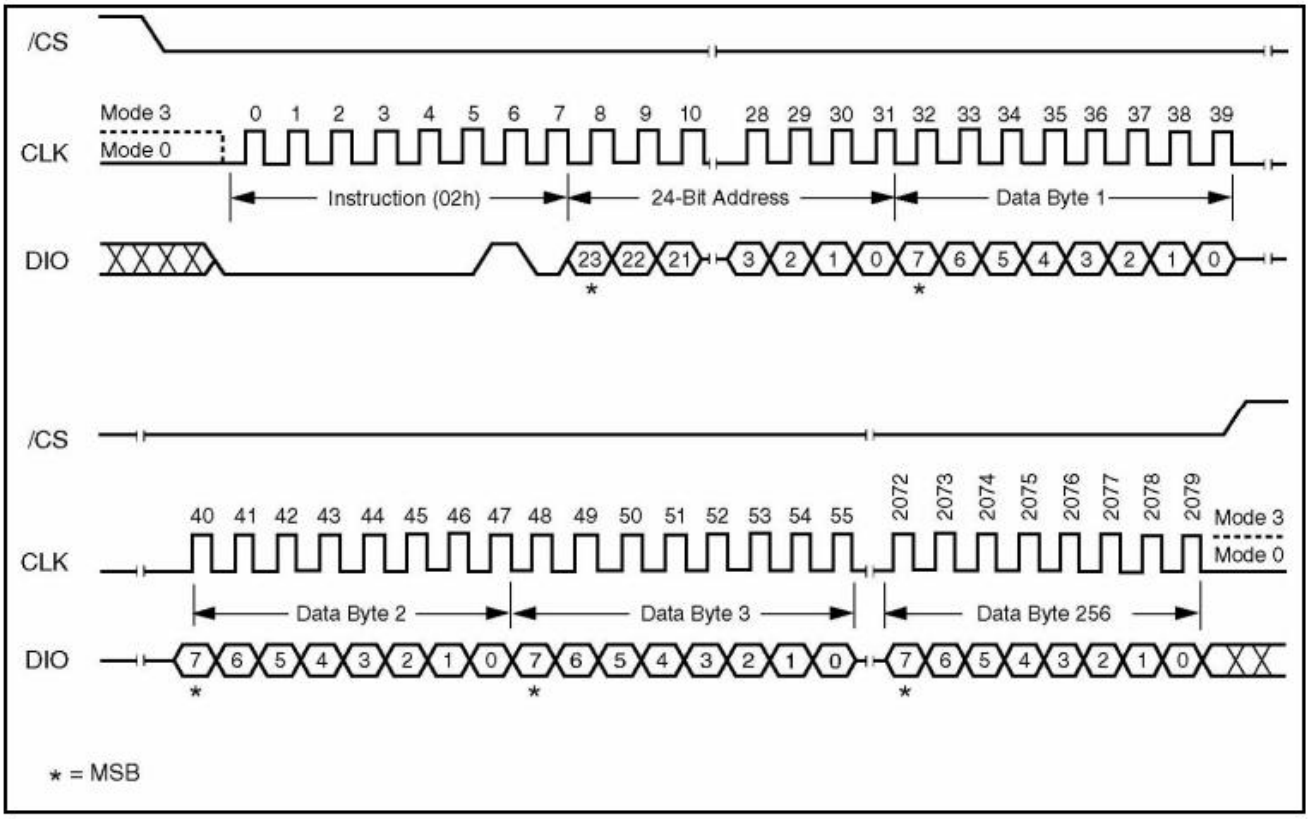


Figure 1-6 Page Program Command Sequence and Timing.

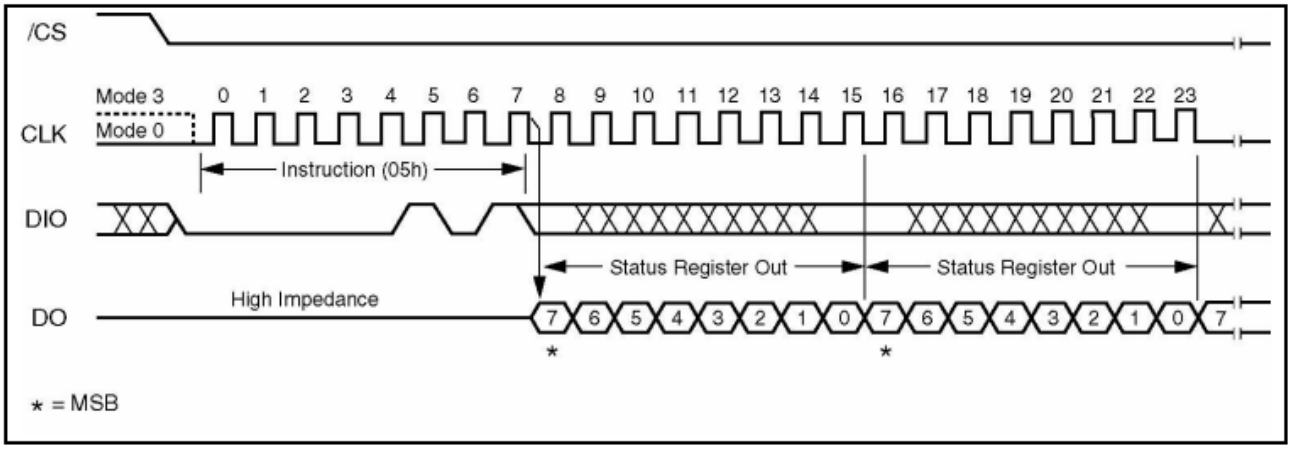


Figure 1-7 Read Status Register Command Sequence and Timing

1.3 Demo Result

Connect the NUC123 USB to the computer, open the [device manager] console, you can see the Nuvoton USB Mass Storage USB Device in the Disk drives area, as shown in Figure 1-8. Then go to [This PC] to open the disk drive of the external device. After formatting, the files can be created, modified, and stored. For example, our experiment creates a Helloworld.txt file and saves it after entering the string "USB Mass Storage SPI Flash Example." to the file. Close the file, remove the USB, and then plug it in again. Check the contents of Helloworld.txt is still correct, indicating that the data has been correctly written to the external SPI Flash, the result is shown in Figure 1-9.

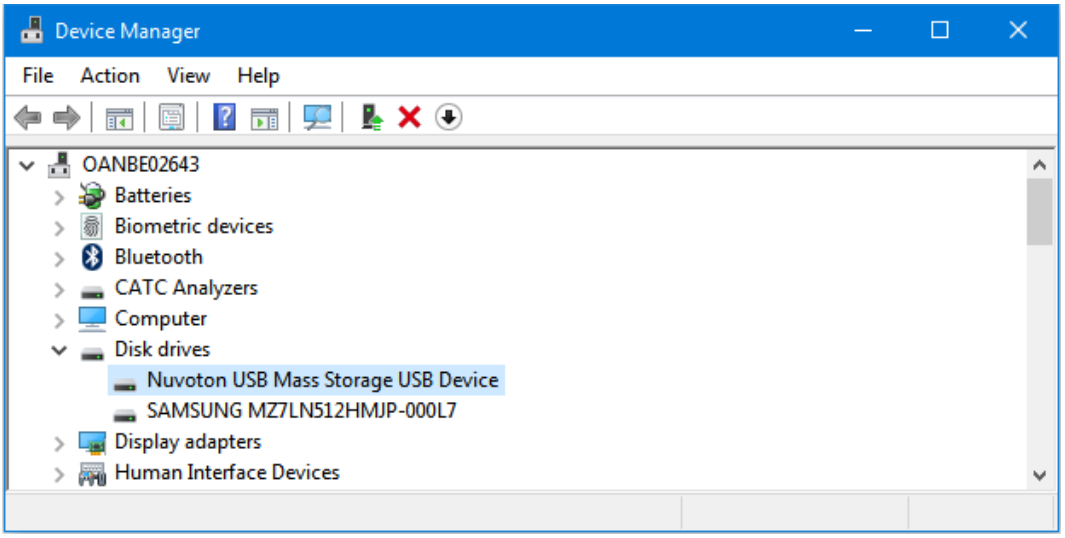


Figure 1-8 Device Manager screenshot

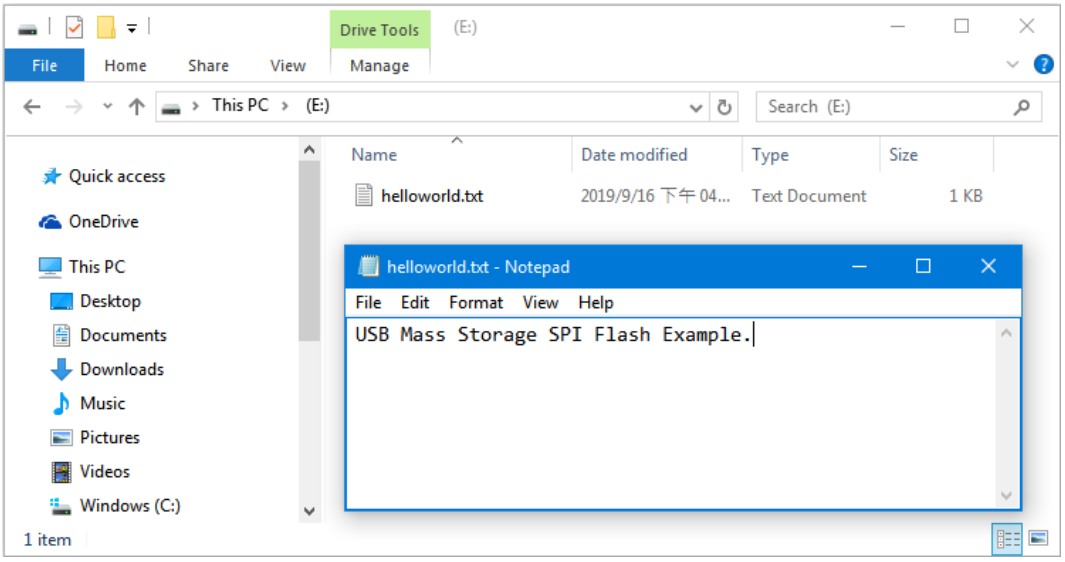


Figure 1-9 Screenshot of performing read and write to the text file.

2 Code Description

2.1 Class Request - Get Max LUN

When Device receives the Get Max LUN class request, it must reply to the number of logical units supported by itself. This example only supports a single logical slot (number 0), so fill the buffer of EP0 (Control IN) with 0, data length. It is 1 Byte and is replied to Host in the next Transaction.

```
void MSC_ClassRequest(void)
{
    uint8_t buf[8];

    USBD_GetSetupPacket(buf);

    if(buf[0] & EP_INPUT)    /* request data transfer direction */
    {
        // Device to host
        switch(buf[1])
        {
            case GET_MAX_LUN:
            {
                if((buf[4] == gsInfo.gu8ConfigDesc[LEN_CONFIG + 2]) && (buf[2] + buf[3] +
                    buf[6] + buf[7] == 1))
                {
                    M8(USBD_BUF_BASE + USBD_GET_EP_BUF_ADDR(EP0)) = 0;
                    /* Data stage */
                    USBD_SET_DATA1(EP0);
                    USBD_SET_PAYLOAD_LEN(EP0, 1);
                    /* Status stage */
                    USBD_PrepareCtrlOut(0, 0);
                }
                else
                    USBD_SET_EP_STALL(EP1); // Stall when wrong parameter
                break;
            }
            default:
            {
```

```
                /* Setup error, stall the device */
                USBD_SetStall(0);
                DBG_PRINTF("Unknow MSC req(0x%x). stall ctrl pipe\n", buf[1]);
                break;
            }
        }
    }
```

2.2 Class Request – Bulk Only Mass Storage Reset

When Device receives the Bulk-Only Mass Storage Reset command, the hardware must be configured to a state, which is ready to receive a CBW. Refer to Figure 1-1 Mass Storage Class Command/Data/Status Protocol. The CBW packet length is fixed at 31 Bytes, thus the Endpoint 3 (Bulk-Out) payload length is set to 31.

```
else
    {
        // Host to device
        switch(buf[1])
        {
            case BULK_ONLY_MASS_STORAGE_RESET:
            {
                /* Check interface number with cfg descriptor and check wValue = 0,
                   wLength = 0 */
                if((buf[4] == gsInfo.gu8ConfigDesc[LEN_CONFIG + 2]) && (buf[2] + buf[3] +
                    buf[6] + buf[7] == 0))
                {

                    g_u32Length = 0; // Reset all read/write data transfer
                    USBD_LockEpStall(0);

                    /* Clear ready */
                    USBD->EP[EP2].CFGP |= USBD_CFGP_CLRRDY_Msk;
                    USBD->EP[EP3].CFGP |= USBD_CFGP_CLRRDY_Msk;

                    /* Prepare to receive the CBW */

                    g_u8EP3Ready = 0;
                    g_u8BulkState = BULK_CBW;
```

```
                    USBD_SET_DATA1(EP3);
                    USBD_SET_EP_BUF_ADDR(EP3, g_u32BulkBuf0);
                    USBD_SET_PAYLOAD_LEN(EP3, 31);

                }
                else
                {
                    /* Stall when wrong parameter */
                    USBD_SET_EP_STALL(EP1);
                }
                /* Status stage */
                USBD_SET_DATA1(EP0);
                USBD_SET_PAYLOAD_LEN(EP0, 0);

                break;
            }
            default:
            {
                // Stall
                /* Setup error, stall the device */
                USBD_SetStall(0);
                DBG_PRINTF("Unknow MSC req (0x%x). stall ctrl pipe\n", buf[1]);
                break;
            }
        }
    }
}
```

2.3 Mass Storage Class Command Process

The CBW contains the Host access, read, write and other commands for the Device. The function MSC_ProcessCmd() in the MassStorage.c file shows how the Device parses the CBW packet and handles each UFI instruction. Here are a few more basic and commonly used instructions as an example.

```
void MSC_ProcessCmd(void)
{
    uint8_t u8Len;
    int32_t i;

    if(g_u8EP3Ready)
    {
        g_u8EP3Ready = 0;
        if(g_u8BulkState == BULK_CBW)
        {
            u8Len = USBD_GET_PAYLOAD_LEN(EP3);
            if(u8Len > 31) u8Len = 31;

            /* Check Signature & length of CBW */
            /* Bulk Out buffer */
            if((*(uint32_t *)((uint32_t)USBD_BUF_BASE + g_u32BulkBuf0) != CBW_SIGNATURE)
            /* || (u8Len != 31)*/)
            {
                USBD_SET_EP_STALL(EP2);
                USBD_SET_EP_STALL(EP3);
                USBD_LockEpStall(1 << EP3);
                g_u8BulkState = BULK_CBW;
                DBG_PRINTF("CBW signature fail. stall bulk out pipe\n");
                return;
            }

            /* Get the CBW */
            for(i = 0; i < u8Len; i++)
                *((uint8_t *)(&g_sCBW.dCBWSignature) + i) = \
                      *(uint8_t *)((uint32_t)USBD_BUF_BASE + g_u32BulkBuf0 + i);

            /* Prepare to echo the tag from CBW to CSW */
            g_sCSW.dCSWTag = g_sCBW.dCBWTag;
```

2.3.1 INQUIRY

Host asks for basic information about the device, such as the manufacturer, product name, or version. After the device receives the INQUIRY instruction, the state switches to Data-In and the product message is written to the Endpoint 2 (Bulk-In) buffer.

```
case UFI_INQUIRY:
{
    uint32_t u32Len;
    uint8_t u8PageCode;

    u32Len = g_sCBW.dCBWDataTransferLength;
    /* Limit length */
    if(u32Len > 36) u32Len = 36;
        u8PageCode = g_sCBW.au8Data[0];

    g_u8BulkState = BULK_IN;
    if(u32Len)
    {
        /* u8PageCode should be zero */
        if(u8PageCode)
        {
         /* Expecting a STALL after data phase completes with a zero-length or short
            packet */
            //USBD_SET_EP_STALL(EP0);
            USBD_SET_EP_STALL(EP3);
            USBD_LockEpStall(1 << EP3);
            DBG_PRINTF("INQUIRY page code = %d", u8PageCode);
        }

        /* Bulk IN buffer */
        USBD_MemCopy((uint8_t *)((uint32_t)USBD_BUF_BASE + g_u32BulkBuf1), \
                     (uint8_t *)g_au8InquiryID, u32Len);
        USBD_SET_PAYLOAD_LEN(EP2, u32Len);
        DBG_PRINTF("Inquiry, len %d\n", u32Len);
    }
    Else
    {
        /* Next is status phase if zero data length in data phase */
        MSC_AckCmd();
     }
       return;
}
```

2.3.2 READ_FORMAT_CAPACITIES

READ_FORAMT_CAPACITIES is the instruction for the Host to read the storage capacity and formattable capacity of the Device. After receiving this command, the Device switches the state to Data-In and calls MSC_ReadFormatCapacity (). Then, it converts the number of logical sectors according to the capacity of the SPI Flash, writes the information to the Endpoint 2 (Bulk-In) buffer, and sends it in the next Transaction.

```
case UFI_READ_FORMAT_CAPACITY:
{
    g_u32Length = g_sCBW.dCBWDataTransferLength;

    /* format capacity descriptor length is fixed to be 12 bytes */
    if(g_u32Length > 20) g_u32Length = 20;
       g_u32Address = MassCMD_BUF;

    MSC_ReadFormatCapacity();
    g_u8BulkState = BULK_IN;
    if(g_u32Length > 0)
    {
        if(g_u32Length > EP2_MAX_PKT_SIZE)
            g_u8Size = EP2_MAX_PKT_SIZE;
        else
            g_u8Size = g_u32Length;

        /* Bulk IN buffer */
        USBD_MemCopy((uint8_t *)(USBD_BUF_BASE + g_u32BulkBuf1),
                     (uint8_t *)g_u32Address, g_u8Size);
        g_u32BytesInStorageBuf = g_u8Size;

        g_u32Address += g_u8Size;
        USBD_SET_EP_BUF_ADDR(EP2, g_u32BulkBuf0);
        MSC_Read();
    }
//  else
//      MSC_AckCmd();
    return;
}
```

2.3.3 READ_10 / READ_12

READ_10 / READ_12 is the instruction used by Host to read the file in the Device storage. After receiving the instruction, Device will switch the status to Data-In. At this moment, MSC_ReadMedia() is called to access the SPI Flash. Device will then read the specified disk address and data length according to the Logic Block Address and dCBWDataTransferLength field in CBW. If the data length exceeds the maximum single packet size of Endpoint 2 (Bulk-In), it can be divided into multiple Bulk-In Transactions.

```
case UFI_READ_10:
case UFI_READ_12:
{
    /* Check if it is a new transfer */
    if(g_u32Length == 0)
    {
        /* Prepare the data for Bulk IN transfer */

        /* Get LBA address */
        g_u32Address = get_be32(&g_sCBW.au8Data[0]);
        g_u32LbaAddress = g_u32Address * UDC_SECTOR_SIZE;
        g_u32Length = g_sCBW.dCBWDataTransferLength;
        g_u32BytesInStorageBuf = g_u32Length;
        DBG_PRINTF("Read addr=0x%x, len=0x%x\n", g_u32LbaAddress, g_u32Length);
        /* Error check  */
        if((g_u32LbaAddress > DRVSPIFLASH_TOTAL_SIZE) || (g_u32LbaAddress + g_u32Length
            > DRVSPIFLASH_TOTAL_SIZE))
        {
            USBD_SET_EP_STALL(EP2);
            USBD_SET_EP_STALL(EP3);
            USBD_LockEpStall((1 << EP2) | (1 << EP3));
            DBG_PRINTF("Stall ep2, ep3. addr=0x%x, len=0x%x\n", g_u32LbaAddress,
                        g_u32Length);

            return;
        }

        i = g_u32Length;
        if(i > STORAGE_BUFFER_SIZE)
            i = STORAGE_BUFFER_SIZE;

        MSC_ReadMedia(g_u32Address * UDC_SECTOR_SIZE, i, (uint8_t *)STORAGE_DATA_BUF);
```

```
        g_u32BytesInStorageBuf = i;
        g_u32LbaAddress += i;
    }
    g_u32Address = STORAGE_DATA_BUF;

    /* Indicate the next packet should be Bulk IN Data packet */
    g_u8BulkState = BULK_IN;

    if(g_u32BytesInStorageBuf > 0)
    {
        /* Set the packet size */
        if(g_u32BytesInStorageBuf > EP2_MAX_PKT_SIZE)
            g_u8Size = EP2_MAX_PKT_SIZE;
        else
            g_u8Size = g_u32BytesInStorageBuf;

        /* Prepare the first data packet (DATA1) */
        /* Bulk IN buffer */
        USBD_MemCopy((uint8_t *)((uint32_t)USBD_BUF_BASE + g_u32BulkBuf1),
                     (uint8_t *)g_u32Address, g_u8Size);
        g_u32Address += g_u8Size;

        /* kick - start */
        USBD_SET_EP_BUF_ADDR(EP2, g_u32BulkBuf1);
        /* Trigger to send out the data packet */
        USBD_SET_PAYLOAD_LEN(EP2, g_u8Size);
        g_u32Length -= g_u8Size;
        g_u32BytesInStorageBuf -= g_u8Size;
    }
//   else
//         MSC_AckCmd();
      return;
    }
```

2.3.4 WRITE_10 / WRITE_12

WRITE_10 / WRITE_12 is the instruction that Host writes the file to the Device storage device. After receiving the command, Device switches the state to Data-Out, and prepares Endpoint 3 (Bulk-Out) to receive the data from Host. The file content will arrive in the further Transactions. Similarly, if the file being written exceeds the maximum single packet size of Endpoint 3, the data is split into several Transactions.

```
case UFI_WRITE_10:
case UFI_WRITE_12:
{
    if(g_u32Length == 0)
    {
        g_u32Length = g_sCBW.dCBWDataTransferLength;
        g_u32Address = STORAGE_DATA_BUF;
        g_u32DataFlashStartAddr = get_be32(&g_sCBW.au8Data[0]) * UDC_SECTOR_SIZE;
    }
    DBG_PRINTF("Write 0x%x  0x%x\n", g_u32Address, g_u32Length);

    if((g_u32Length > 0))
    {
        USBD_SET_PAYLOAD_LEN(EP3, EP3_MAX_PKT_SIZE);
        g_u8BulkState = BULK_OUT;
    }
    return;
}
```

When the packet of the file is received in the Data-Out state, the Device will call MSC_Write(), which will read the data in Endpoint 3 buffer, and drive the peripheral SPI to store the file to the specified region of the SPI Flash.

```
else if(g_u8BulkState == BULK_OUT)
{
    switch(g_sCBW.u8OPCode)
    {
        case UFI_WRITE_10:
        case UFI_WRITE_12:
        case UFI_MODE_SELECT_10:
        {
            MSC_Write();
            return;
        }
```

MSC_Write() will further call the low-level function SpiWrite() to perform the write operation. Note that if the address written is not Sector Alignment, special handling is required. Because the minimum erase size of SPI Flash is a Sector (4K Bytes), if the address does not have Alignment, the microcontroller will first read the original data in the entire Sector, modify the contents of the specified part, then wipe out the entire Sector and re-program the data again.

```
void SpiWrite(uint32_t u32Addr, uint32_t u32Size, uint32_t u32BufAddr)
{
    /* This is low level write function of USB Mass Storage */
    int32_t i32Len, i, i32Offset;
    uint32_t *pu32;
    uint32_t u32AlignAddr;

    i32Len = (int32_t)u32Size;

    if (i32Len == DRVSPIFLASH_SECTOR_SIZE && ((u32Addr & (DRVSPIFLASH_SECTOR_SIZE-1)) ==
        0))
    {
        /* one-sector length & the start address is sector-alignment */
        WINBOND25X16A_EraseSector(u32Addr);

        while (i32Len >= DRVSPIFLASH_PAGE_SIZE)
        {
            WINBOND25X16A_ProgramPage( u32Addr, (uint8_t *)u32BufAddr);
            i32Len     -= DRVSPIFLASH_PAGE_SIZE;
            u32BufAddr += DRVSPIFLASH_PAGE_SIZE;
            u32Addr    += DRVSPIFLASH_PAGE_SIZE;
        }
    }
    else
    {
        do
        {
            /* u32AlignAddr: sector address */
            u32AlignAddr = u32Addr & 0x1FFFF000;

             /* Get the sector i32Offset*/
            i32Offset = (u32Addr & (DRVSPIFLASH_SECTOR_SIZE-1));

           if (i32Offset || (u32Size < DRVSPIFLASH_SECTOR_SIZE))
           {
```

```
        /* if the start address is not sector-alignment or the u32Size is less
                    than one sector, read back the data of the destination sector to
                    g_au32SectorBuf[]. */
                SpiRead(u32AlignAddr, DRVSPIFLASH_SECTOR_SIZE,
                        (uint32_t)&g_au32SectorBuf[0]);
            }

            /* Update the data */
            pu32 = (uint32_t *)u32BufAddr;

            /* i32Len: the byte count between the start address and the end of a sector.*/
            i32Len = DRVSPIFLASH_SECTOR_SIZE - i32Offset;

            /* check if the range of data arrive at the end of a sector. */
            if (u32Size < i32Len)
            {
                /* Not arrive at the end of a sector. "i32Len" indicate the actual byte
                    count of data. */
                i32Len = u32Size;
            }
            for (i=0; i<i32Len/4; i++)
            {
                g_au32SectorBuf[i32Offset/4 + i] = pu32[i];
            }

            WINBOND25X16A_EraseSector(u32AlignAddr);

            for (i=0;i<16;i++) /* one sector (16 pages) */
            {
                WINBOND25X16A_ProgramPage(u32AlignAddr + (i << 8),
                                          (uint8_t *)g_au32SectorBuf + (i << 8));
            }

            u32Size -= i32Len;
            u32Addr += i32Len;
            u32BufAddr += i32Len;

        }while (u32Size > 0);
    }
}
```

2.3.5 TEST_UNIT_READY

If there is no other command to execute, Host will periodically send a TEST_UNIT_READY command to Device to check whether Device is in Ready state. In general, the Device only needs to reply to the CSW packet of Good Status.

```
case UFI_TEST_UNIT_READY:
{
    DBG_PRINTF("Test Unit\n");
    g_u8BulkState = BULK_IN;
    MSC_AckCmd();
    return;
}
```

In MSC_AckCmd (), set the bCSWStatus field to 0, indicating a Good Status.

```
case UFI_TEST_UNIT_READY:
{
                g_sCSW.dCSWDataResidue = 0;
                g_sCSW.bCSWStatus = 0;
                break;
}
```

3 Software and Hardware Environment

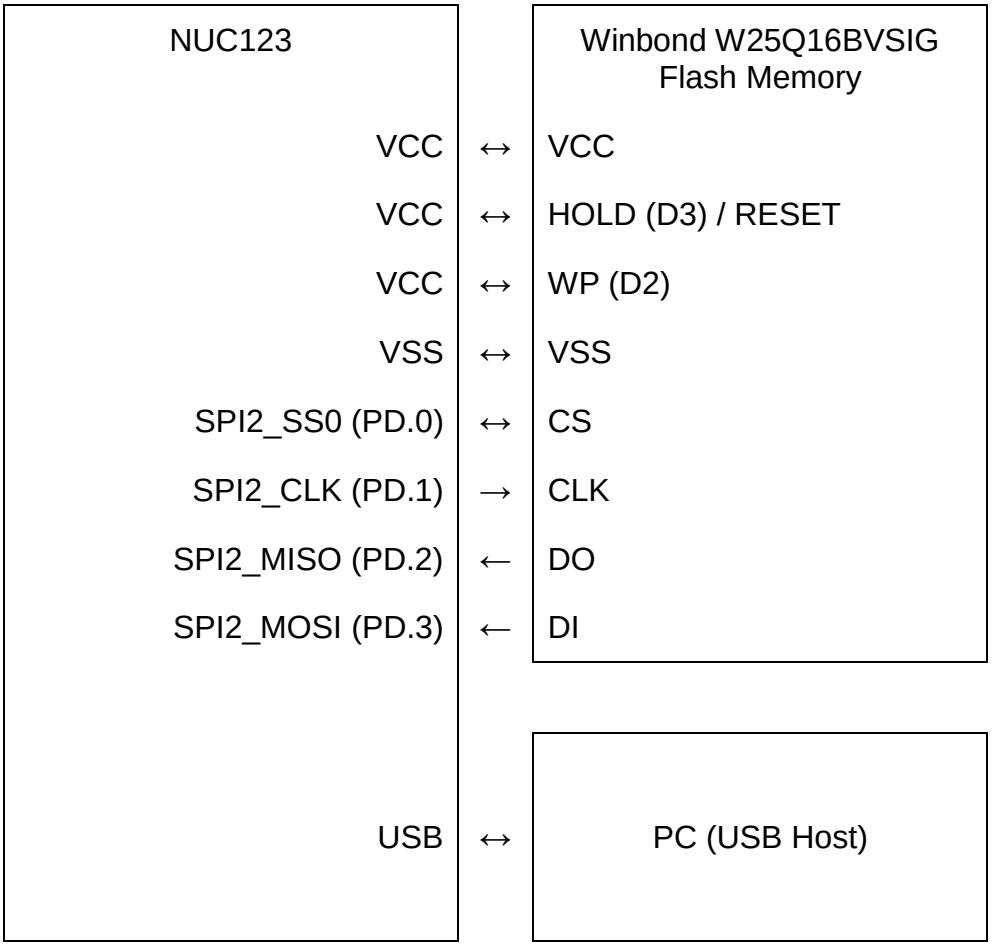
Software Environment

- BSP version
  - NUC123Series_BSP_CMSIS_V3.01.0001
- IDE version
  - Keil uVersion 5.26

Hardware Environment

- Circuit components
  - NuTiny-EVB-123-LQFP64 V1.0 Developer Board
  - Winbond W25X16A Serial Flash Memory
  - USB Mini USB Cable

- Diagram


NUC123
Winbond W25Q16BVSIG
Flash Memory
VCC ↔ VCC
VCC ↔ HOLD (D3) / RESET
VCC ↔ WP (D2)
VSS ↔ VSS
SPI2_SS0 (PD.0) ↔ CS
SPI2_CLK (PD.1) → CLK
SPI2_MISO (PD.2) ← DO
SPI2_MOSI (PD.3) ← DI
USB ↔ PC (USB Host)

4 Directory Information

- EC_NUC123_USBD_MSC_SPI_Flash_V1.00
  - Library — Sample code header and source files
    - CMSIS — Cortex® Microcontroller Software Interface Standard (CMSIS) by Arm® Corp.
    - Device — CMSIS compliant device header file
    - StdDriver — All peripheral driver header and source files
  - SampleCode
    - ExampleCode — Source file of example code

5 How to Execute Example Code

1. Browsing into sample code folder by Directory Information (section 4) and double click NUC123_USBD_MSC_SPI_Flash.uvprojx.
2. Enter Keil compile mode
   a. Build
   b. Download
   c. Start/Stop debug session
3. Enter debug mode
   a. Run

6 Revision History

Date	Revision	Description
Sep. 30, 2019	1.00	1. Initially issued.

Important Notice

Nuvoton Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, "Insecure Usage".

Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, the control or operation of dynamic, brake or safety systems designed for vehicular use, traffic signal instruments, all types of safety devices, and other applications intended to support or sustain life.

All Insecure Usage shall be made at customer's risk, and in the event that third parties lay claims to Nuvoton as a result of customer's Insecure Usage, customer shall indemnify the damages and liabilities thus incurred by Nuvoton.

Please note that all data and specifications are subject to change without notice.
All the trademarks of products and companies mentioned in this datasheet belong to their respective owners.