使用SPI Flash 作为 USB 储存装置

NuMicro® 32 位系列微控制器范例代码介绍

文件信息

代码简述	微控制器扮演 USB 大容量储存装置，并使用 SPI Flash 作为储存媒介。
BSP 版本	NUC123Series_BSP_CMSIS_v3.01.001
开发平台	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 功能介绍

1.1 简介

微控制器 NUC123 透过 USB 通讯端口连接计算机，扮演大容量储存装置类别，并连接外部的 SPI Flash Memory 作为储存装置的媒介，用户即可将微控制器作为一般的外接U盘使用，可在个人计算机上进行档案的写入与读取。

1.2 原理

本次实验采用 USB Mass Storage Device Class (MSC) Bulk-Only Transfer 协议，根据该协议规范，Host 与 Device 交换资料的过程可分为三个状态阶段，分别为Command Transport (CBW)，Data-Out / Data-In，以及 Status Transport (CSW)，其状态切换流程如图1-1所示。

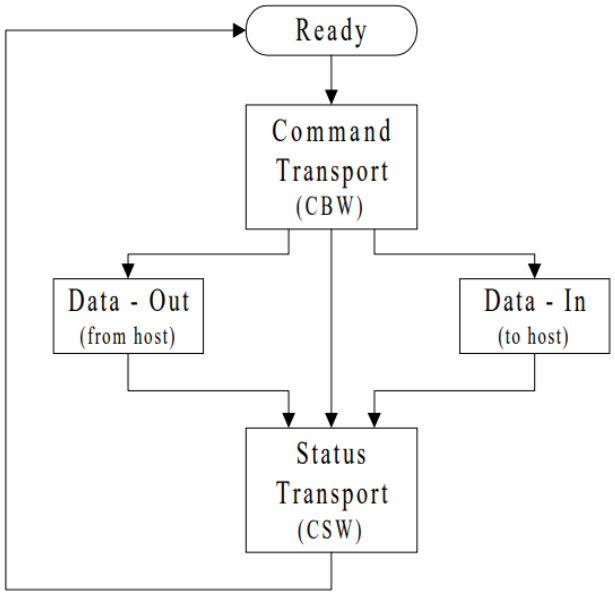


图1-1 Mass Storage Class Command/Data/Status Protocol

MSC 同时也定义了两个专属的类别请求，分别为 Bulk-Only Mass Storage Reset，以及 Get Max LUN 命令，Bulk-Only Mass Storage Reset 命令用以重置 Device 回到 Ready 状态，通知装置准备接收 CBW 命令；Get Max LUN 则用以取得 Device 所支持的逻辑单元 (储存槽) 数量，其命令的详细内容与格式可参阅 USB Mass Storage Class Bulk-Only Transport Specification 文件。

1.2.1 Command Transport (CBW)

Command Transport 內含有一个 Transaction ，数据方向为 OUT，由 Host 透过 Bulk-Out Endpoint 对 Device 送出指定命令的数据封包，并以 Command Block Wrapper (CBW) 之格式封装，封包格式如图1-2所示。在 CBW 封包当中，字段 CBWCB 存放 Host 对于 Device 的储存媒体进行存取或控制的指令，该指令类型会决定下一个状态为 Data-In 或是 Data-Out，以及双方该传递的内容，详细的指令描述可参阅 UFI Command Specification 文件规范。

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*							

图1-2 Command Block Wrapper (CBW) 封包格式

1.2.2 Data-In / Data-Out

Data-In / Data-Out 状态下的行为受 CBW 内的命令所影响，举例来说，若 CBWCB 内的 UFI 指令为 WRITE，则代表由 Host 希望向 Device 的储存媒介写入档案数据，同时双方将状态切换为 Data-Out；反之若 UFI指令为 READ，则表示 Host 向 Device 读取储存装置内的档案，由 Device 向 Host 送出数据封包，此时状态切换至 Data-In。Data-In / Data-Out 阶段允许包含多个 Transaction，由 CBWCB 内的 dCBWDataTransferLength 字段所影响。顺带一提，本次实验的储存媒介是采用外接的 SPI Flash Memory，微控制器接收到读取或写入的指令后，必须驱动周边 SPI 对外部 Flash Memory 的指定区块进行数据的读写。

1.2.3 Status Transport (CSW)

Status Transport 的用途是 Device 向 Host 回复本次 CBW 命令执行的状态，含有一个 Transaction，数据方向为 IN，此阶段的数据封包以 Command Status Wrapper (CSW) 之格式封装，封包格式如图1-3所示。Host 可透过 CSW 内的 bCSWStatus 字段判断 Device 在执行 CBW 命令的过程是否出现错误，bCSWStatus 数值意义如图1-4所示。

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*							

图1-3 Command Status Wrapper (CSW) 封包格式

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

图1-4 bCSWStatus 字段内容

1.2.4 USB 端点配置

本次实验需要配置四个端点给予 USB 接口使用，如下所示：

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

1.2.5 Serial Flash Memory

本次实验使用 Winbond 25X16A SPI Flash 作为 Mass Storage Device 的储存媒介，储存容量为 2 MB，最小的写入单位为一个 Page (256 B)，每 16 Page 可组成一个 Sector (4 KB)，每 256 Page 则组成一个 Block (64K Byte)，其中 Sector 为最小的抹除单位。SPI Flash 的控制方式非常简便，微控制器首先透过以下的命令集合对 SPI Flash 下达指令，包含 Read、Program 或是 Erase 等行为，接着双方会依据指令的类别，决定后续传递的数据的内容与长度。

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			

图1-5 Winbond SPI Flash 命令表

图1-5为 Winbond 25X16A 指令表，以 Page Program (02h) 与 Read Status Register (05h) 指令为例，可参阅下图1-6与图1-7之时序，Master 首先送出1 Byte 的指令 02h，接续 24-Bit 的数据写入地址与 256 Byte 的数据后，便完成 Page Program 的指令传输。之后，微控制器可再使用 Read Status Register (05h) 指令来询问 SPI Flash 的状态，来检查数据是否已经写入完毕。使用者可参阅 Winbond W25X16A Datasheet 文件，内有各个指令详细的使用方法与说明。

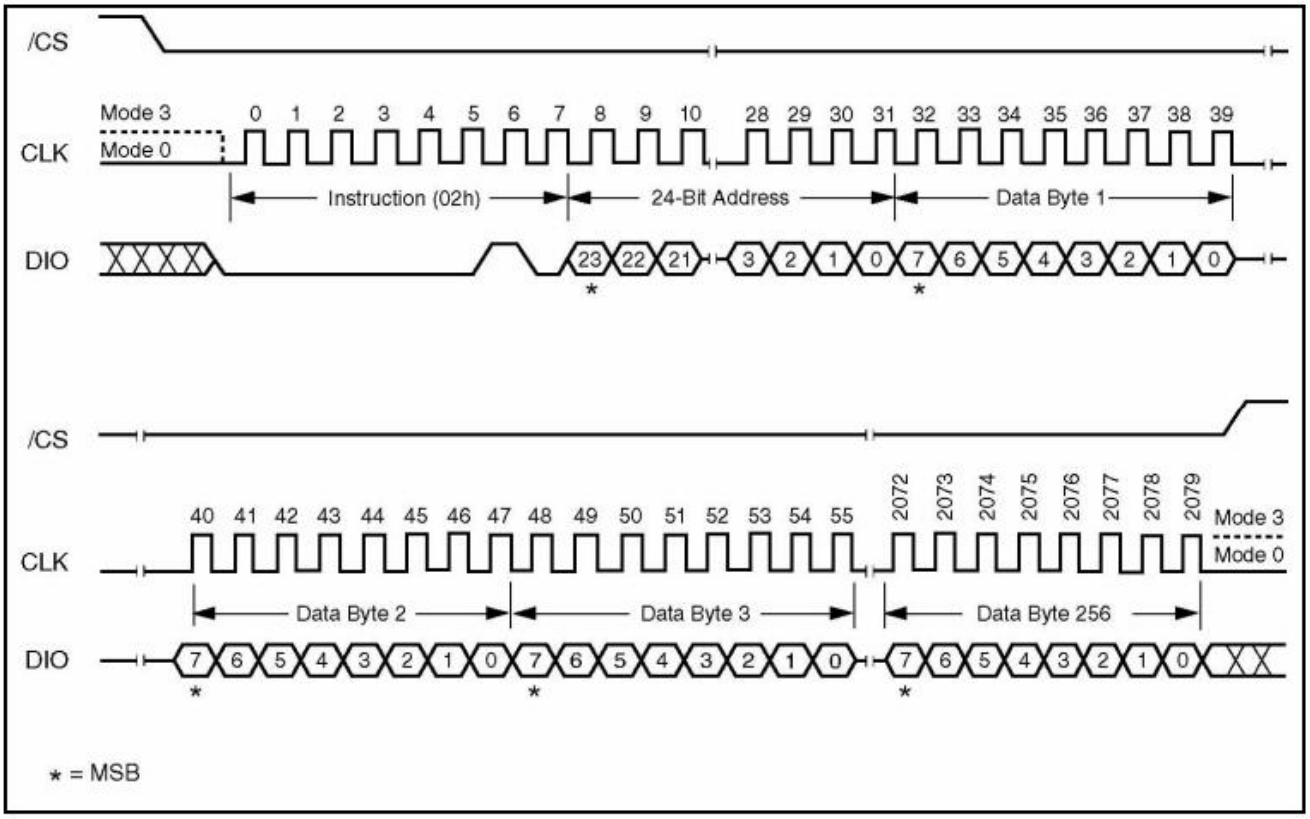


图1-6 Page Program 命令时序图

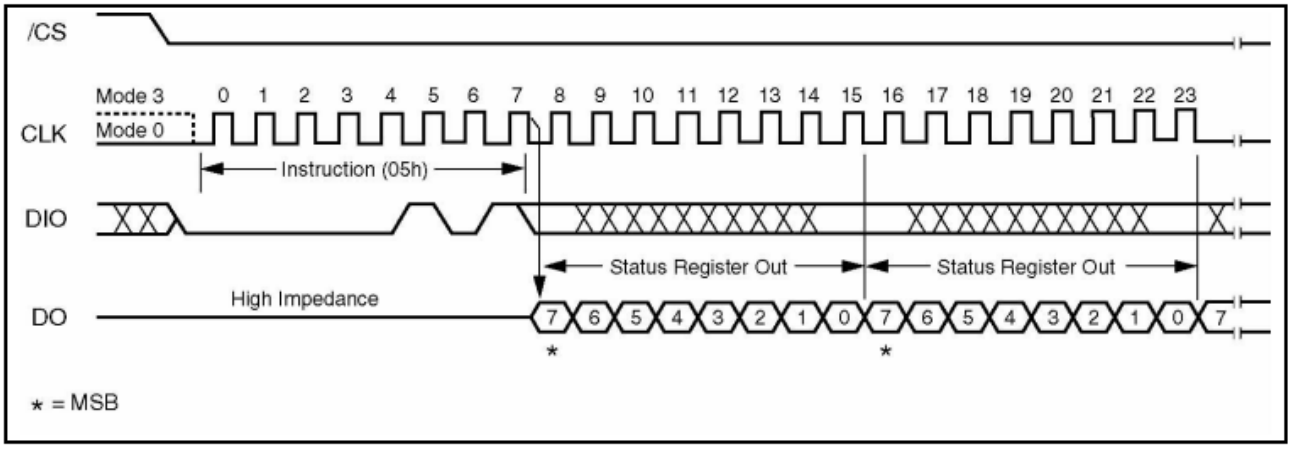


图1-7 Read Status Register 命令时序图

1.3 执行结果

将 NUC123 USB 接入计算机，开启【设备管理器】，可看到 Disk drives 区域出现 Nuvoton USB Mass Storage USB Device 装置，如图1-8所示。此时至【我的计算机】开启外接装置的储存槽，格式化后即可进行档案的创建、修改、与储存等动作。举例来说，本次实验创建一个Helloworld.txt档案，并在档案输入 “USB Mass Storage SPI Flash Example.” 字符串后储存，关闭档案并移除USB 后再重新插入，检查 Helloworld.txt 的内容可正常读取，表示数据已经正确写入至外部 SPI Flash 当中，执行结果如图1-9所示。

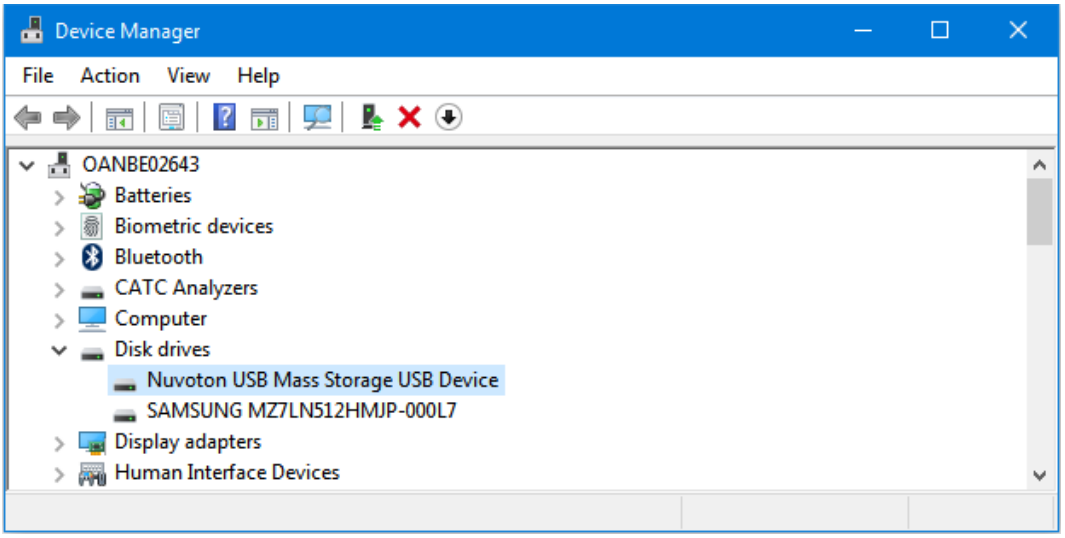


图 1-8 Device Manager screenshot

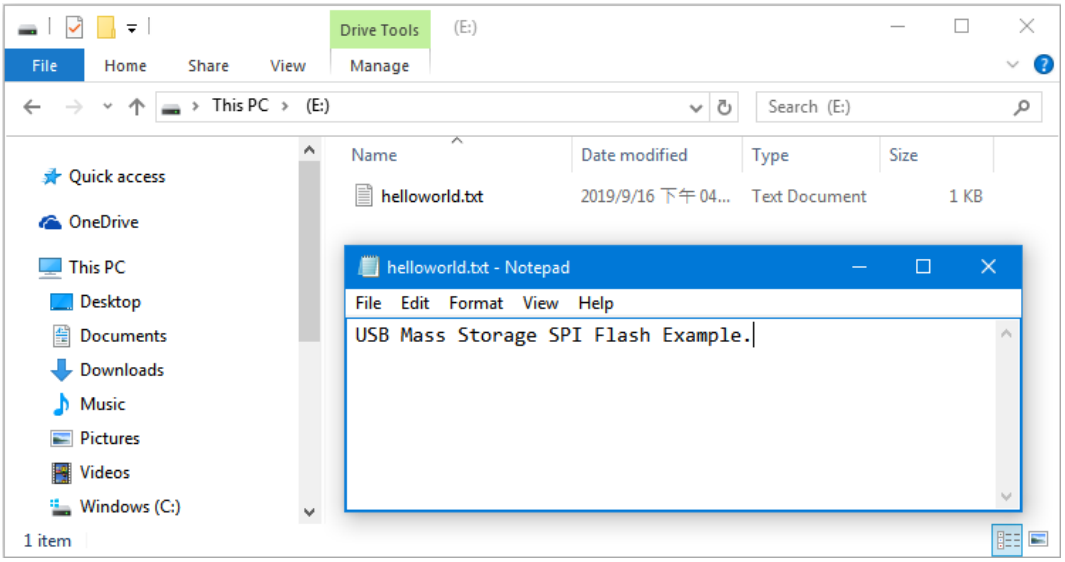


图 1-9 文本文件读写画面

2 代码介绍

2.1 类别请求 - Get Max LUN

当 Device 接收到 Get Max LUN 类别请求，必须回复该装置所支持的逻辑单元数量，本次范例仅支持单一逻辑槽 (编号0)，因此将 EP0 (Control IN) 的缓冲区填入0，数据长度为 1 Byte，并在下一次的Transaction 当中回复给 Host。

```
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 类别请求 - Bulk-Only Mass Storage Reset

当 Device 接收到 Bulk-Only Mass Storage Reset 指令，必须将硬件配置至准备接收 CBW 的状态，可参阅图 1-1 Mass Storage Class Command/Data/Status Protocol 流程。CBW 封包长度为固定 31 Byte，因此将 Endpoint 3 (Bulk-Out) payload 长度设为 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 命令处理

CBW 内夹带了 Host 对于 Device 进行访问、读取、写入等命令， MassStorage.c 档案内的 MSC_ProcessCmd() 函式当中实作了 Device 如何解析 CBW 封包以及应对各个 UFI 指令的行为，以下以几条较为基本与常用的指令做为范例。

```
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

主机询问装置的基本信息，例如制造商、产品名称或版本等信息。装置接收到 INQUIRY 指令后，状态切换至 Data-In，将产品讯息写入 Endpoint 2 (Bulk-In) 缓冲区。

```
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 为 Host 读取 Device 的储存容量与可格式化容量的指令，Device 接收到此指令后，将状态切换至 Data-In，并呼叫 MSC_ReadFormatCapacity ()。此时根据 SPI Flash 的容量大小换算逻辑扇区的数量，将信息写入至 Endpoint 2 (Bulk-In) 缓冲区，并在下一个 Transaction 送出。

```
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 为 Host 用来读取 Device 储存装置内档案的指令，Device 接收到该指令后将状态切换为 Data-In，此处呼叫 MSC_ReadMedia() 对 SPI Flash 进行读取操作。Device 根据 CBW 内的 Logic Block Address 以及 dCBWDataTransferLength 字段来读取指定的扇区地址以及数据长度，若数据长度超过 Endpoint 2 (Bulk-In) 的最大单一封包大小，则可以分割为多个 Bulk-In Transaction 送出。

```
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 为 Host 将档案写入 Device 储存装置的指令，Device 接收到该指令后将状态切换为 Data-Out，准备 Endpoint 3 (Bulk-Out) 接收来自 Host 的数据，实际的档案内容将会在下一个 Transaction 当中到来。同样地，若写入的档案超过 Endpoint 3 的最大单一封包大小，则数据会被分割至数个 Transaction 当中。

```
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;
}
```

在 Data-Out 状态下接收到档案封包，呼叫 MSC_Write() 将 Endpoint 3 缓冲区数据读出，并驱动周边 SPI 将档案写入 SPI Flash 的指定扇区。

```
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() 会进一步呼叫底层 SpiWrite() 执行写入操作，此部分须注意若写入的地址并非 Sector Alignment 则需要采取特别处理，因 SPI Flash 的最小擦除单位为一个 Sector (4K Bytes)，地址没有 Alignment 的情况下，微控制器会先将整个 Sector 内原有的数据读出，修改指定地址的内容，擦出整个 Sector 后再将数据重新写入。

```
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

Host 在没有其他命令要执行的情况下，会定期向 Device 送出 TEST UNIT READY 指令，检查 Device 是否 Ready，正常状态下 Device 仅需回复 Good Status 的 CSW 封包即可

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

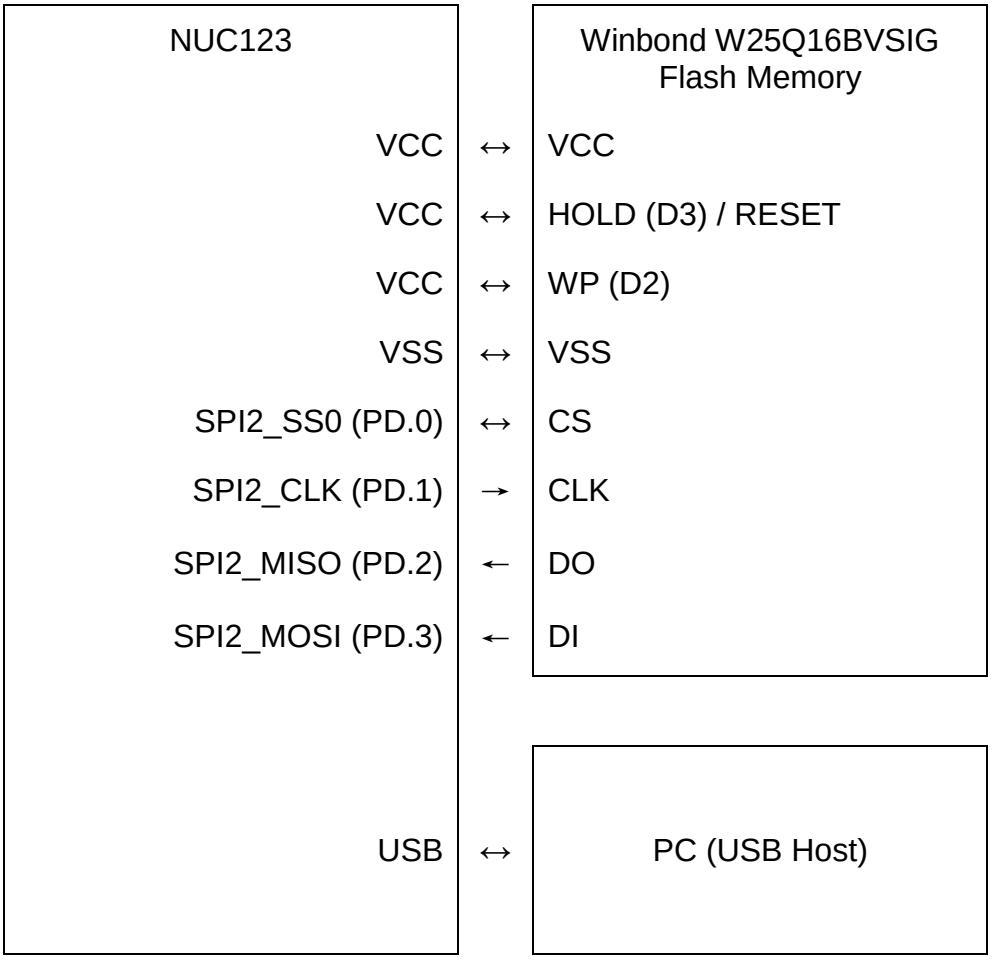
MSC_AckCmd () 内，将 bCSWStatus 字段设为 0，表示Good Status。

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

3 软件与硬件环境

- 软件环境
  - BSP 版本
    - NUC123Series_BSP_CMSIS_v3.01.001
  - IDE 版本
    - Keil uVersion 5.26
- 硬件环境
  - 电路组件
    - NuTiny-EVB-123-LQFP64 V1.0 Developer Board
    - Winbond W25X16A Serial Flash Memory
    - USB Mini USB Cable

■ 示意图


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 目录信息

- 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 如何执行范例程序

1. 根据目录信息章节进入 ExampleCode 路径中的 KEIL 文件夹，双击 NUC123_USBD_MSC_SPI_Flash.uvprojx。
2. 进入编译模式接口
   a. 编译
   b. 下载代码至内存
   c. 进入 / 离开除错模式
3. 进入除错模式接口
   a. 执行代码

6 修订纪录

Date	Revision	Description
Sep. 30, 2019	1.00	1. 初始发布.

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.