Using SPI Flash as USB MSC Device

Example Code Introduction for 32-bit NuMicro® Family

Document Information

Application	This example code supports to program files to SPI Flash through mass storage interface.
BSP Version	M032_Series_BSP_CMSIS_V3.05.000
Hardware	NuMaker-M032KI V1.0

1. Overview

This example code uses the M032 series microcontroller as a SPI Flash programmer to update SPI Flash through the USB interface. The M032 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. By using mass storage interface, it is not necessary to provide any tool or driver at PC to copy a bin file to SPI Flash. When the user plugs in the device by USB cable, PC will recognize it as a mass storage disk and user just needs to copy a "*Update.bin*" file to the mass storage disk to do "program file to SPI Flash". The FAT information is saved to SRAM, and the FAT data will be lost after reset or power-off. So, in other words, the disk will also be reset to empty after the example code restarts.

1.1 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.

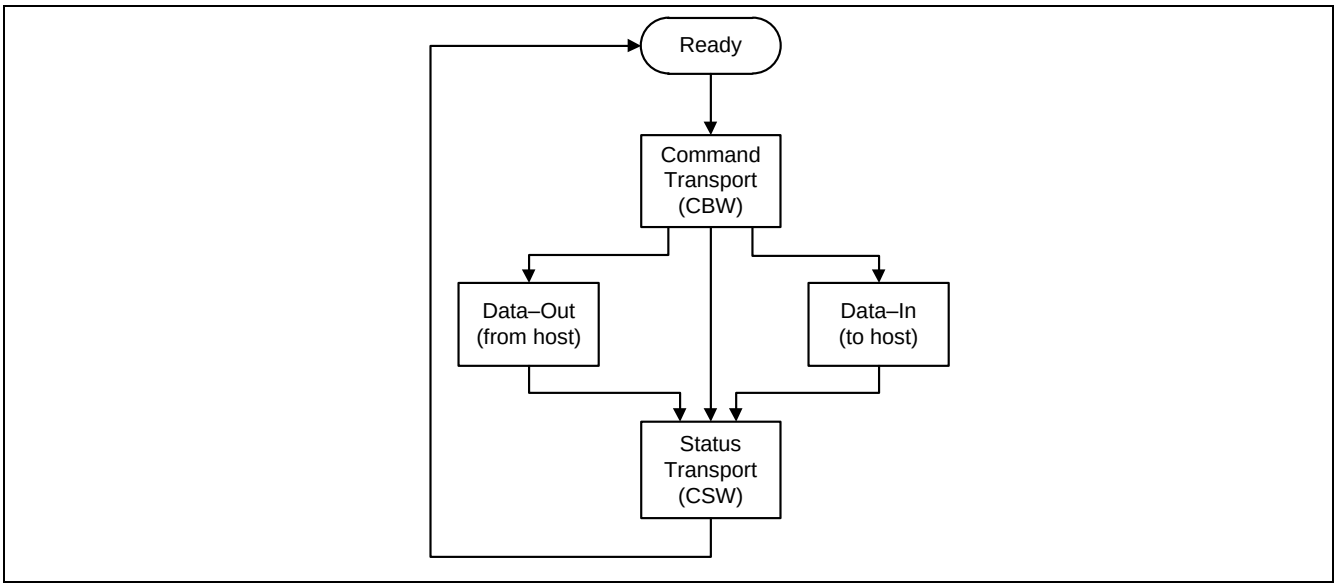

Figure 1-1 Mass Storage Class Command/Data/Status Protocol

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 detailed information (https://www.usb.org/sites/default/files/usbmassbulk_10.pdf).

1.1.1 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 Table 1-1. 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 (https://www.usb.org/sites/default/files/usbmass-ufi10.pdf).

Byte \ bit	7	6	5	4	3	2	1	0
0-3	*dCBWSignature*							
4-7	*dCBWTag*							
8-11	*dCBWDataTransferLength*							
12	*bmCBWFlags*							
13	*Reserved(0)*				*bCBWLUN*			
14	*Reserved(0)*				*bCBWCBLengh*			
15-30	*CBWCB*							

Table 1-1 Command Block Wrapper (CBW) Packet Format

1.1.2 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.

1.1.3 Status Transport (CSW)

The purpose of the Status Transport is that the Device responds to the Host with the execution status of the CBW command. 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 Table 1-2. Host can read the bCSWStatus field in CSW to determine whether the Device has an error during the CBW command. The value of bCSWStatus is shown in Table 1-3.

Byte \ bit	7	6	5	4	3	2	1	0
0-3	*dCSWSignature*							
4-7	*dCSWTag*							
8-11	*dCSWDataResidue*							
12	*bCSWStatus*							

Table 1-2 Command Status Wrapper (CSW) Packet Format

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

Table 1-3 Values for bCSWStatus Field

1.1.4 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

1.1.5 Serial Flash Memory

The example code uses Winbond W25Q32JVSIG SPI Flash as the medium of Mass Storage Device with a storage capacity of 4 MB. The minimum program size is a Page (256 B), each 16 Pages can be grouped into a Sector (4 KB), and each 256 Pages constitutes a Block (64 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.

Data Input Output	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Number of Clock(1-1-1)	**8**	**8**	**8**	**8**	**8**	**8**	**8**
Write Enable	**06h**						
Volatile SR Write Enable	**50h**						
Write Disable	**04h**						
Release Power-down / ID	**ABh**	Dummy	Dummy	Dummy	(ID7-ID0)(2)		
Manufacturer/Device ID	**90h**	Dummy	Dummy	00h	(MF7-MF0)	(ID7-ID0)	
JEDEC ID	**9Fh**	(MF7-MF0)	(ID15-ID8)	(ID7-ID0)			
Read Unique ID	**4Bh**	Dummy	Dummy	Dummy	Dummy	(UID63-0)	
Read Data	**03h**	A23-A16	A15-A8	A7-A0	(D7-D0)		
Fast Read	**0Bh**	A23-A16	A15-A8	A7-A0	Dummy	(D7-D0)	
Page Program	**02h**	A23-A16	A15-A8	A7-A0	D7-D0	D7-D0(3)	
Sector Erase (4KB)	**20h**	A23-A16	A15-A8	A7-A0			
Block Erase (32KB)	**52h**	A23-A16	A15-A8	A7-A0			
Block Erase (64KB)	**D8h**	A23-A16	A15-A8	A7-A0			
Chip Erase	**C7h/60h**						
Read Status Register-1	**05h**	(S7-S0)(2)					
Write Status Register-1(4)	**01h**	(S7-S0)(4)					
Read Status Register-2	**35h**	(S15-S8)(2)					
Write Status Register-2	**31h**	(S15-S8)					
Read Status Register-3	**15h**	(S23-S16)(2)					
Write Status Register-3	**11h**	(S23-S16)					
Read SFDP Register	**5Ah**	A23-A16	A15-A8	A7–A0	dummy	(D7-0)	
Erase Security Register(5)	**44h**	A23-A16	A15-A8	A7-A0			
Program Security Register(5)	**42h**	A23-A16	A15-A8	A7-A0	D7-D0	D7-D0(3)	
Read Security Register(5)	**48h**	A23-A16	A15-A8	A7-A0	Dummy	(D7-D0)	
Global Block Lock	**7Eh**						
Global Block Unlock	**98h**						
Read Block Lock	**3Dh**	A23-A16	A15-A8	A7-A0	(L7-L0)		
Individual Block Lock	**36h**	A23-A16	A15-A8	A7-A0			
Individual Block Unlock	**39h**	A23-A16	A15-A8	A7-A0			
Erase / Program Suspend	**75h**						
Erase / Program Resume	**7Ah**						
Power-down	**B9h**						
Enable Reset	**66h**						
Reset Device	**99h**						

Table 1-4 Winbond SPI Flash Command List

Table 1-4 shows the Winbond W25Q32JVSIG command list. Take the Page Program (02h) and Read Status Register (05h) instructions as an example; refer to the timing diagrams in Figure 1-2 and Figure 1-3. 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 W25Q32JVSIG Datasheet* for detailed description for each instruction.

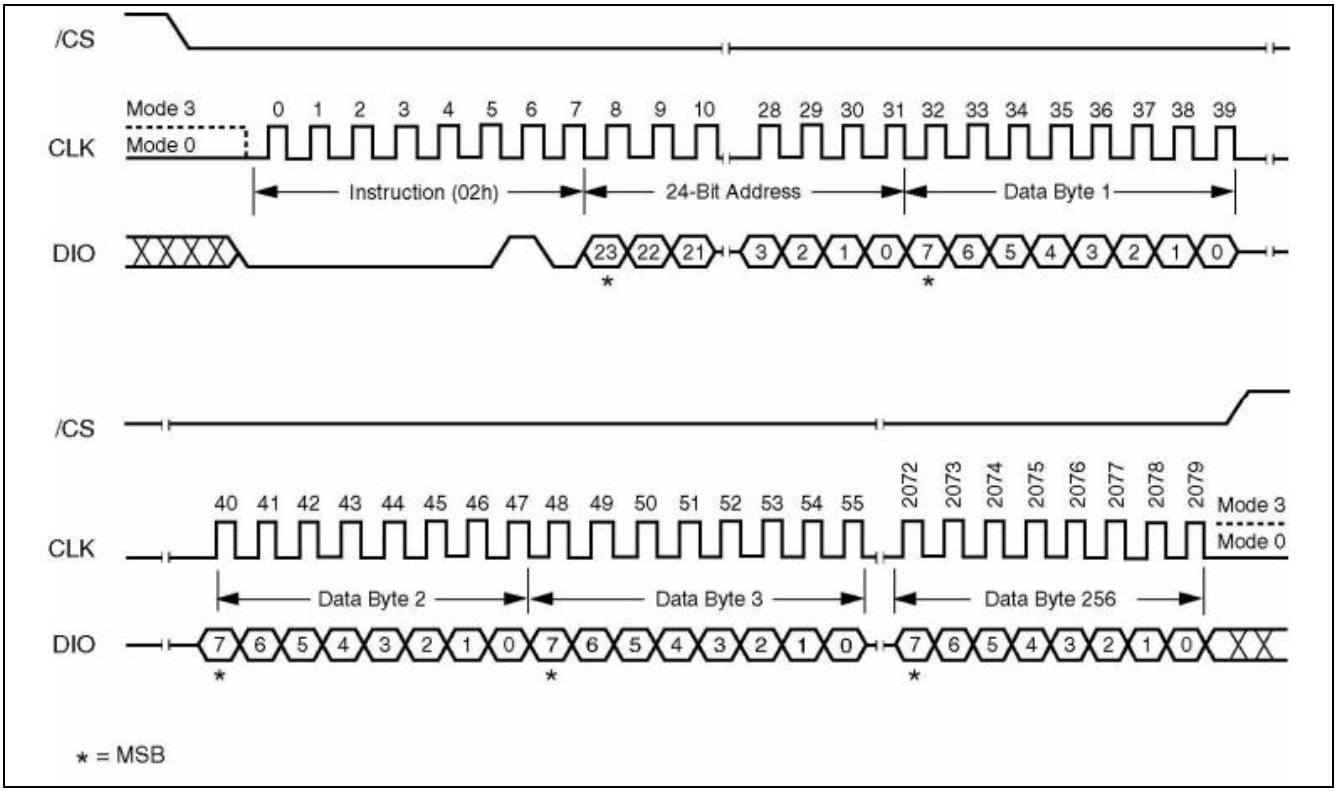

Figure 1-2 Page Program Command Sequence and Timing

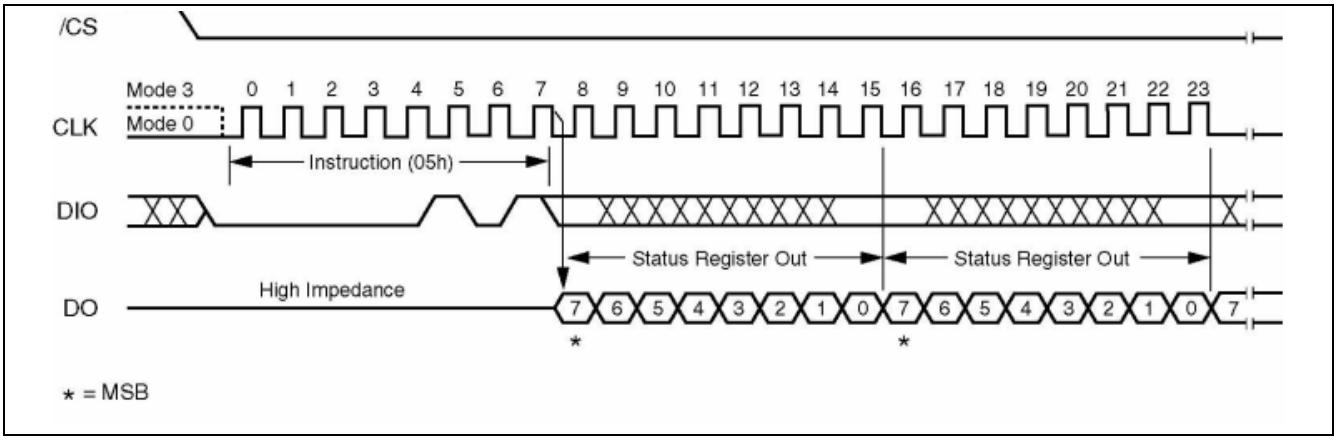

Figure 1-3 Read Status Register Command Sequence and Timing

1.2 Demo Result

1.2.1 Virtual Disk

PC can detect a mass storage class device and Data Flash can export a 3.98 MB disk when the M032 is plugged into PC's USB port. The user needs to name the file as *Update.bin* and drag or copy-and-paste the file to the disk.

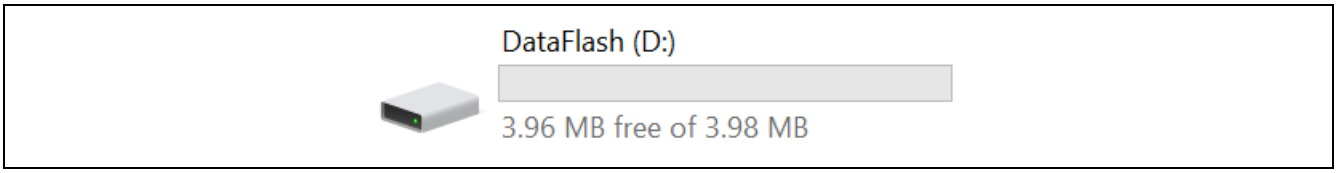

Figure 1-4 M032 Virtual Disk

1.2.2 Read Data

Copy *Update.bin* from the disk without buffering and it is the same as the data written to M032.

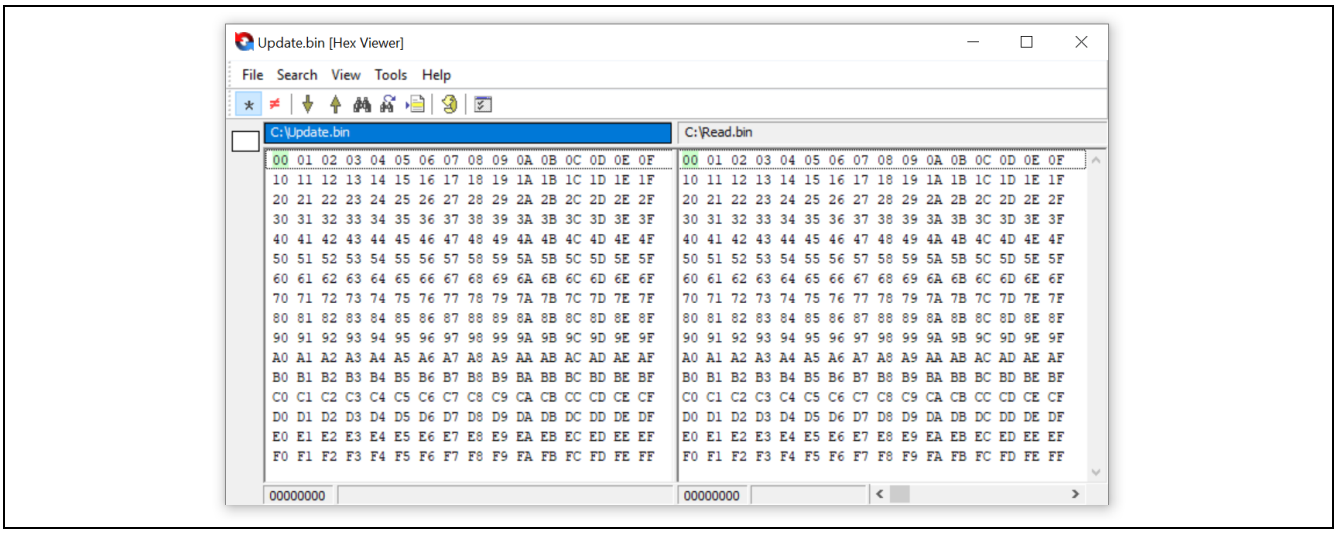

Figure 1-5 Comparison Result

1.2.3 Copy File without Buffering

1.2.3.1 Windows

Windows command - Xcopy

Xcopy Command	
Item	**Description**
/j	This option copies files without buffering, a feature useful for very big files. This option was first available in Windows 7.

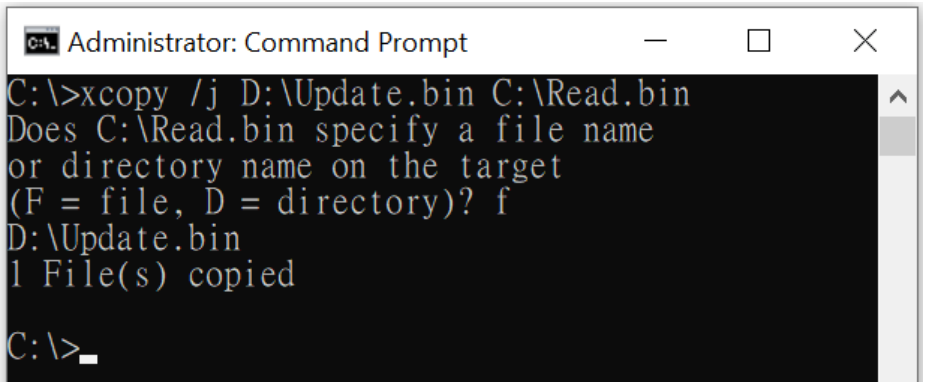

Figure 1-6 Copy File without Buffering by xcopy

1.2.3.2 Linux

Linux command - dd

dd Command	
Item	**Description**
iflg=direct	direct use direct I/O for data.

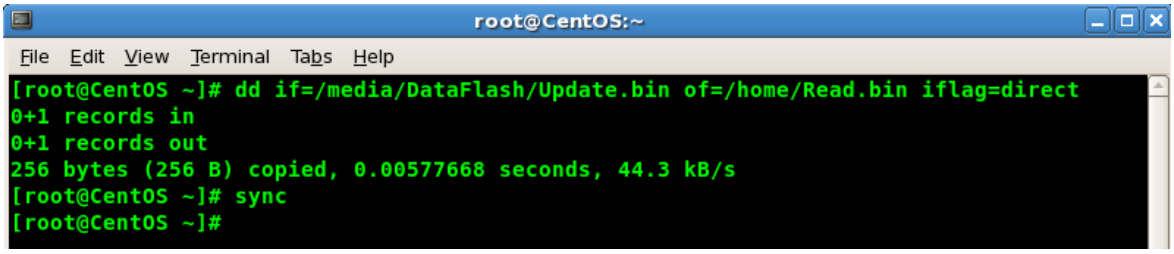

Figure 1-7 Copy File without Buffering by dd Command

2. Code Description

2.1 Virtual Disk Properties

The code for the virtual disk properties is in DataFlashProg.c as follows. It's the boot sector data for the virtual disk.

```
uint8_t u8BootSectorData[512] =
{
    /* Instructions to jump to boot code */
    0xEB, 0x3C, 0x90,
    /* Name string (MSDOS5.0) */
    0x4D, 0x53, 0x44, 0x4F, 0x53, 0x35, 0x2E, 0x30,
    /* Bytes/sector (0x0200 = 512) */
    0x00, 0x02,
    /* Sectors/cluster */
    0x08,
    /* Size of reserved area */
    (RSVD_SEC_CNT & 0xFF), (RSVD_SEC_CNT >> 8),
    /* Number of FATs */
    NUM_FAT,
    /* Byte 17 & 18 (Max. number of root directory entries) */
    (ROOT_ENT_CNT & 0xFF) , (ROOT_ENT_CNT >> 8),
    /* Byte 19 & 20 (Total number of sectors) */
    (SECTOR_CNT & 0xFF), (SECTOR_CNT >> 8),
    /* Media type (removable) */
    0xF8,
    /* Byte 22 & 23 (FAT size) */
    (FAT_SZ & 0xFF), (FAT_SZ >> 8),
    /* Sectors/track */
    0x01, 0x00,
    /* Number of heads */
    0x01, 0x00,
    /* Number of sector before partition */
    0x00, 0x00, 0x00, 0x00,
    /* Total number of sectors */
    0x00, 0x00, 0x00, 0x00,
    /* Drive number */
    0x80,
    /* Unused */
    0x00,
    /* Extended boot signature */
    0x29,
    /* Volume serial number */
    0x2F, 0x44, 0x83, 0x7A,
    /* Volume label - "NO NAME " */
    0x4E, 0x4F, 0x20, 0x4E, 0x41, 0x4D, 0x45, 0x20, 0x20, 0x20, 0x20,
    /* File system type label ("FAT12 ") */
    0x46, 0x41, 0x54, 0x31, 0x32, 0x20, 0x20, 0x20,
    ...
    /* Signature value (0xaa55) */
    0x55, 0xAA
};
```

The definition of the virtual disk properties is in DataFlashProg.h and user can modify the definition of “MB_SIZE” to change the virtual disk size.

```
#define MB_SIZE                 4
#define DISK_SIZE               (MB_SIZE*1024 * 1024)
...
#define BYTE_PER_SEC            512
#define ROOT_ENT_CNT            512
#define ROOT_ENT_SEC_CNT        ((32 * ROOT_ENT_CNT) / BYTE_PER_SEC)
#define NUM_FAT                 2
#define FAT_SEC                 RSVD_SEC_CNT
#define FAT_SEC_ADDR            (RSVD_SEC_CNT * BYTE_PER_SEC)
#define ROOT_SEC_ADDR           (FAT_SEC_ADDR + FAT_SZ * NUM_FAT * BYTE_PER_SEC)
#define DATA_SEC_ADDR           (ROOT_SEC_ADDR + ROOT_ENT_SEC_CNT * BYTE_PER_SEC)
#define SECTOR_CNT              (DISK_SIZE / BYTE_PER_SEC)
```

2.2 FAT and Flash Program Function

The DataFlashRead() function is used to respond to the FAT table on PC and read the specified disk address and data length according to the Logic Block Address and dCBWDataTransferLength field in CBW.

```
void DataFlashRead(uint32_t addr, uint32_t size, uint32_t buffer)
{
    /* This is low level read function of USB Mass Storage */
    uint32_t * pu32Buf = (uint32_t *)buffer;
    memset((uint8_t *)pu32Buf, 0, STORAGE_BUFFER_SIZE);

    if (addr == 0x00000000)
    {
#ifdef __FAT_INFO__
        if(g_ShowFat && GET_FAT_SZ != 0)
        {
            printf("\n");
            printf("Default FAT_SEC        0x%08X ", FAT_SEC);
            printf("Current FAT_SEC       0x%08X\n", GET_FAT_SEC);
            printf("Default FAT_SZ         0x%08X ",FAT_SZ);
            printf("Current FAT_SZ        0x%08X\n", GET_FAT_SZ);
            printf("Default ROOT_SEC_ADDR  0x%08X ",ROOT_SEC_ADDR);
            printf("Current ROOT_SEC_ADDR 0x%08X\n", GET_ROOT_SEC_ADDR);
            printf("Default DATA_SEC_ADDR  0x%08X ",DATA_SEC_ADDR);
            printf("Current DATA_SEC_ADDR 0x%08X\n", DATA_SEC_ADDR);
            g_ShowFat = 0;
        }
#endif
        USBD_MemCopy((uint8_t *)buffer, u8BootSectorData, sizeof(u8BootSectorData));
    }
    else
    {
        if (addr >= ROOT_SEC_ADDR && addr < DATA_SEC_ADDR)     /* Root Directory */
            USBD_MemCopy( (uint8_t *)buffer, u8DirData + (addr - ROOT_SEC_ADDR), 512);
        else if (addr >= FAT_SEC_ADDR && addr < ROOT_SEC_ADDR) /* File Allocation Table */
            USBD_MemCopy((uint8_t *)buffer, u8FAT + ((addr - FAT_SEC_ADDR)/* % (FAT_SZ *
                        BYTE_PER_SEC)*/), 512);
        else if(addr > DATA_SEC_ADDR) /* Data */
        {
```

```
            SpiRead(u32SPI_RAddress, size, (uint32_t)pu32Buf);
            if(u32SPI_RAddress % 0x40000 == 0)
                DbgPrintf("Read from SPI 0x%08X - 0x%08X\n",u32SPI_RAddress, *(uint32_t
                          *)pu32Buf);
            u32SPI_RAddress+=size;
        }
    }
}
```

The DataFlashWrite() calls the low-level function SpiWrite() to perform the write operation to store the file to the specified region of the SPI Flash. After the write operation is completed, it calls SpiRead() to verify the data stored in SPI Flash.

```
void DataFlashWrite(uint32_t addr, uint32_t size, uint32_t buffer)
{
    int k,l;
    /* This is low level write function of USB Mass Storage */
    if(addr >= DATA_SEC_ADDR)  /* Data */
    {
        if(addr == DATA_SEC_ADDR)
        {
            if(g_UpdateEnable == 0 && g_u8Windows == 0)
            {
                DbgPrintf("OS : Windows\n");
                g_u8Windows = 1;
            }
        }
        if(g_UpdateEnable || (g_u8Windows == 0 && g_u8MACOS == 0))
        {
            if(u32SPI_WAddress == 0)
            {
                if(g_u8Windows == 0 && g_u8MACOS == 0)
                    DbgPrintf("OS : Linux\n");
            }

            addr -= DATA_SEC_ADDR;

            SpiWrite(u32SPI_WAddress, STORAGE_BUFFER_SIZE, (uint32_t)buffer);

            SpiRead(u32SPI_WAddress, STORAGE_BUFFER_SIZE, (uint32_t)Storage_Block_Verify);

            if(memcmp((void *)buffer, (void *)Storage_Block_Verify,
               STORAGE_BUFFER_SIZE) !=0)
                DbgPrintf("Verify fail 0x%08X\n",u32SPI_WAddress);

            u32SPI_WAddress += STORAGE_BUFFER_SIZE;

            if(u32SPI_WAddress % 0x40000 == 0)
            {
                DbgPrintf("\rData Transferred %d KB",u32SPI_WAddress >> 10);
                if(g_file_size != 0)
                    DbgPrintf(" / %d KB",*g_file_size >>10);
            }
            if(g_file_size != 0 && u32SPI_WAddress >= *g_file_size)
            {
                DbgPrintf("\rData Transferred %d KB",u32SPI_WAddress >> 10);
```

```
                if(g_file_size != 0)
                    DbgPrintf(" / %d KB",*g_file_size >>10);

                g_TransferDone = 1;
                u32SPI_RAddress = 0;
                g_UpdateEnable = 0;
                DbgPrintf("\nTransferred Done\n");
            }
        }
        else
        {
            uint8_t *pu8Data = (uint8_t *)buffer;

            if(pu8Data[8] == 'M' && pu8Data[9] == 'a' && pu8Data[10] == 'c')
            {
                if(g_u8MACOS == 0)
                {
#if (ENABLE_DEBUG_MSG)
                    char *puchar = (char *)(&pu8Data[8]);
#endif
                    DbgPrintf("OS : %s\n",puchar);
                    g_u8MACOS = 1;
                }
                /* Finder progresses a copy of a file - HFS type code 'brok' & HFS creator
                    code 'MACS' */
                if(pu8Data[50] == 'b' && pu8Data[51] == 'r' && pu8Data[52] == 'o' &&
                    pu8Data[53] == 'k' && pu8Data[54] == 'M' && pu8Data[55] == 'A' &&
                    pu8Data[56] == 'C' && pu8Data[57] == 'S')
                {
                    g_u8MACOS_Update = 1;
                }
            }
        }
    }
    else if (addr == 0x00000000)
    {
        USBD_MemCopy(u8BootSectorData,(uint8_t *) buffer, 512);
#ifdef __FAT_INFO__
        g_ShowFat = 1;
#endif
    }
    else if (addr >= ROOT_SEC_ADDR && addr < DATA_SEC_ADDR) /* Root Directory */
    {
        USBD_MemCopy(u8DirData + (addr - ROOT_SEC_ADDR), (uint8_t *)buffer, 512);

        if((g_UpdateEnable == 0 && g_TransferDone == 0) || ((g_u8MACOS == 1 || g_u8Windows
           == 0)&& g_u8ShowFile != 1))           /* Check File name (When Update Function
           is not Enabled or MAC OS) */
        {
            for(k =0; k<32; k++)                 /* Check Root Directory */
            {
                for(l=0; l<FILE_NAME_LENGTH; l++)
                {
                    if(u8FileName[l] != 0)       /* Need to Check File Name - Character */
                        if(u8DirData[k*16+i8FileIndex[l]] != u8FileName[l])      /* Check
                           File Name */
```

```
                            break;
                }

                if(l == FILE_NAME_LENGTH)                               /* Match */
                {
                    g_file_size = (uint32_t *)&u8DirData[k *16 + 60];  /* Get File Size */

                    if(*g_file_size == 0)
                    {
                        g_file_size = 0;
                        continue;
                    }
                    g_UpdateEnable = 1;

                    DbgPrintf("\nFile Name:");
                    for(l=0; l<FILE_NAME_LENGTH; l++)   /* Display the Update File Name */
                    {
                        if((u8DirData[k *16 + i8FileIndex[l]] != 0) && (u8DirData[k *16 +
                           i8FileIndex[l]+1] == 0) )
                            DbgPrintf("%c",u8DirData[k *16 + i8FileIndex[l]]);
                        else
                            break;
                    }

                    if(g_u8MACOS == 1 || g_u8Windows == 0)
                    {
                        g_u8ShowFile = 1;
                    }
                    DbgPrintf("\nFile Size: %dB\n",*g_file_size);
                    if(u32SPI_WAddress >= *g_file_size)
                    {
                        u32SPI_RAddress = 0;
                        g_TransferDone = 1;
                        DbgPrintf("\nTransferred Done\n");
                    }
                    break;
                }
            }
        }
    }
    else if (addr >= FAT_SEC_ADDR && addr < ROOT_SEC_ADDR)  /* File Allocation Table */
        USBD_MemCopy(u8FAT + ((addr - FAT_SEC_ADDR) /*% (FAT_SZ * BYTE_PER_SEC)*/),
                     (uint8_t *)buffer, 512);
}
```

3. Software and Hardware Requirements

3.1 Software Requirements

- BSP version
  - M032_Series_BSP_CMSIS_V3.05.000
- IDE version
  - Keil uVersion 5.28

3.2 Hardware Requirements

- Circuit components
  - NuMaker-M032KI V1.0
  - Level1-Training
  - Micro USB cable
- Pin Connect

NuMaker-M032KI		Winbond W25Q32JVSIG Flash Memory
VCC	↔	VCC
VCC	↔	HOLD (D3) / RESET
VCC	↔	WP (D2)
VSS	↔	VSS
SPI0_SS0 (PA.3)	↔	CS
SPI0_CLK (PA.2)	→	CLK
SPI0_MISO (PA.1)	←	DO
SPI0_MOSI (PA.0)	←	DI
USB	↔	PC (USB Host)

Figure 3-1 Pin Connect

4. Directory Information

The directory structure is shown below.

EC_M032_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

Figure 4-1 Directory Structure

5. Example Code Execution

1. Browse the sample code folder as described in the Directory Information section and double-click *M032_USBD_MSC_SPI_Flash.uvproj*.
2. Enter Keil compile mode.
   - Build
   - Download
   - Start/Stop debug session
3. Enter debug mode.
   - Run

6. Revision History

Date	Revision	Description
2023.06.07	1.00	Initial version.

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.