Application Note

32-bit Cortex™-M0 MCU
NuMicro® Family

USB MassStorage Application Node & Sample Code

Table of Contents-

1 INTRODUCTION

This document explains the sample code, “Smpl_UDC” which is included in the ***AN_1020_EN.ZIP*** file and demonstrates how to create an USB MassStorage device through the related libraries.

1.1 Scope

This article is provided for programmers applying the USB IP to USB applications. It is assumed that the reader is familiar with the Universal Serial Bus (USB) Specification 1.1 and the Device Class Definition for MassStorage.

1.2 Features

- Support customization of VID (Vendor ID) / PID (Product ID).
- Support customization the storage device.

2 CODE SECTION

2.1 Main Function (in Smpl_UDC.c)

Refer chapter of Calling Sequence for its calling sequence.

In the main function, the PLL clock should be set to 48MHz for USB PHY clock.

After the hardware initialization and clock setting is finished, udcInit() will be called to reset and initialize USB endpoints state. It will check and set g_u8UsbState according to the USB connection status.

After USB initialization, udcFlashInit() will be called to initialize the storage device. User can add the customized initialization code for their new storage type.

The last step is to call udcMassBulk(), which is the polling loop internally and will check and response the various events about USB.

```
int32_t main(void)
{
      ...

      /* Initialize USB Device function */
      udcInit();
      /* Initialize mass storage device */
      udcFlashInit();
      /* Start USB Mass Storage */
      udcMassBulk();

      return 0;
}
```

2.2 The polling flow (in MassStorage.c)

As you see, function udcMassBuld() is only a loop that polling USB by function UsbIsr(). UsbIsr() is not a real interrupt handler, and it's just called in main process to demonstrate the USB function simply and easily. (In more complex application, the similar procedure like UsbIsr() can be moved into USB interrupt handler.)

```
void udcMassBulk(void)
{
    /* Handler the USB ISR by polling */
      while(1)
      {
```

```
            UsbIsr();
      }
}



void UsbIsr(void)
{
      uint32_t u32EVF = _DRVUSB_GET_EVF();
      if (u32EVF & EVF_FLD)
      {
          /* Handle the USB attached/detached event */
             UsbFdt();
      }
      else if(u32EVF & EVF_BUS)
      {
          /* Handle the USB bus event: Reset, Suspend, and Resume */
             UsbBus();
      }
      else if(u32EVF & EVF_USB)
      {
          /* Handle the USB Protocol/Clase event */
             UsbUsb(u32EVF);
    }
}
```

As the code above, UsbIsr() checks the USB event flag, including attach/detach event, USB bus event and USB protocol/class event, and calls the corresponding response function if any event flag was set.

In all of the response functions, the most two important functions are UsbBulkOutAck and UsbBulkInAck called by UsbUsb(), which complete most of the communication works to the USB host. UsbBulkOutAck/UsbBulkInAck implements the SCSI command needed by a MassStorage device, which is represented as struct CBW::u8OPCode in the sample code.

(Further more, user can try to check “u8OPCode” and support customized SCSI command on this USB device. For windows users, please seek keywords DeviceIOControl and SCSI_IOCTL_DATA_OUT about how send the customized SCSI command by a Windows program.)

Other two functions SpiRead and SpiWrite, called by the response functions, provided for user to customize the storage I/O operation. User can rewrite SpiRead/SpiWrite to support new storage device. The prototype functions are:

```
void SpiRead(uint32_t addr, uint32_t size, uint32_t buffer)
{
    /* This is low level read function of USB Mass Storage */
}

void SpiWrite(uint32_t addr, uint32_t size, uint32_t buffer)
{
    /* This is low level write function of USB Mass Storage */
}
```

2.3 How to apply new storage type

To apply new storage type, user needs to rewrite the 3 functions:

- uint8_t udcFlashInit(void)

  Initialize the storage device
- void SpiRead(uint32_t addr, uint32_t size, uint32_t buffer)

  Read data from storage device
- void SpiWrite(uint32_t addr, uint32_t size, uint32_t buffer)

  Write data to storage device

3 CALLING SEQUENCE

3.1 Run as an USB Device

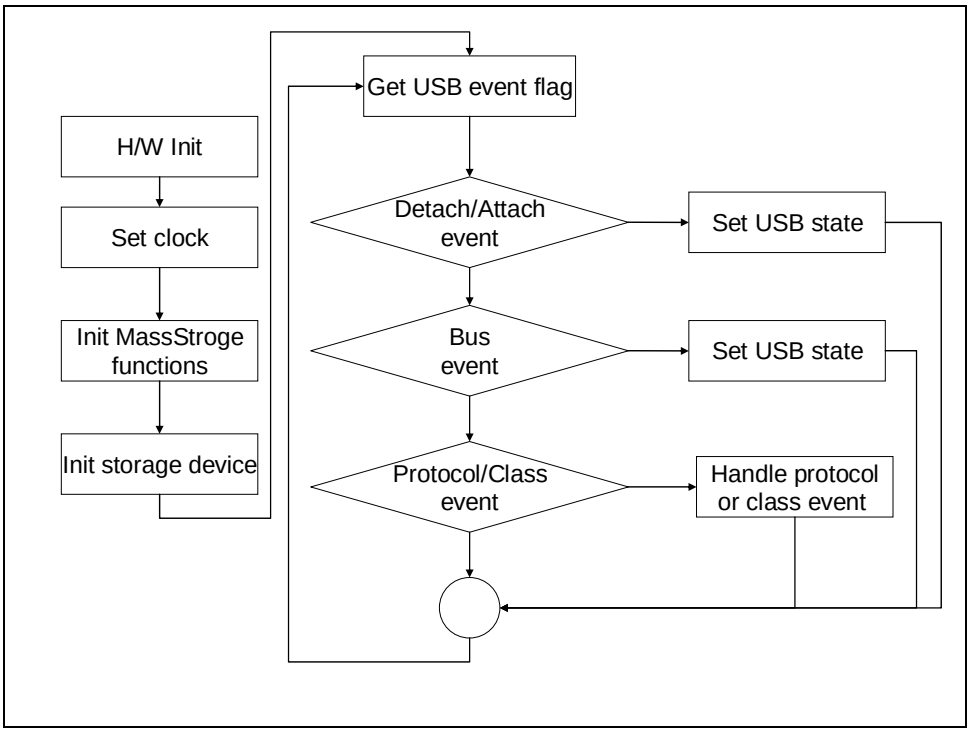

Figure 3-1 Control Flow of Running as an USB Device

1. Initialize hardware
2. Set clock for USB.
3. Initialize USB MassStorage functions.
4. Initialize the storage device.
5. Polling USB status
    1) Check and handle USB attached/detached event
    2) Check and handle USB bus event: Reset, Suspend and Resume
    3) Check and handle USB protocol/Class event
6. Go to step 5 to polling USB status again

3.2 Customize a storage Device

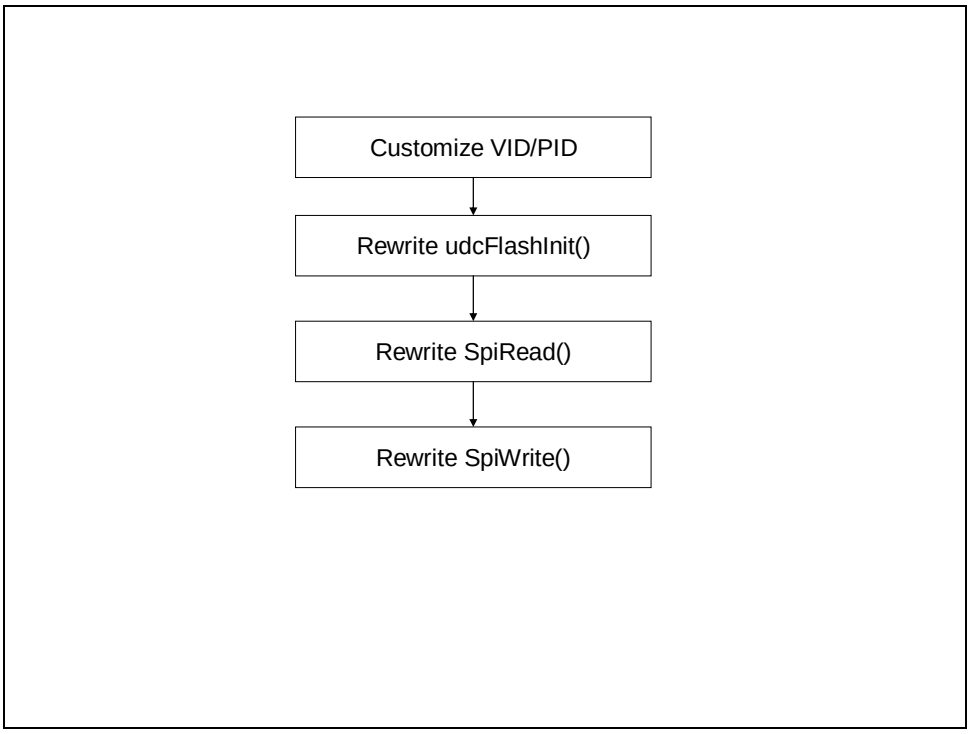

Figure 3-2 Control Flow of Customize Vendor USB Device as MassStorage

1. Customize VID / PID if needed.
2. Rewrite function udcFlashInit(), SpiRead() and SpiWrite() for customized storage device.
3. Please refer section 2.3 for the description of the 3 functions above.

3.3 API Usage Reference

- USB Driver Reference Guide.doc

4 EXECUTION ENVIRONMENT SETUP AND RESULT

4.1 Test Smpl_UDC

The USB MassStorage sample code, Smpl_UDC, could be built by Keil MDK tool and download to NUC1xx series DEV Board through ICE. Then user is able to execute the code in ICE environment or reset the DEV board to execute the code which had been programmed in on-chip Program Flash to verify the Smpl_UDC code.

4.2 Result

When the DEV board executes this sample code, the DEV board performs the behavior as USB disk. Once the USB port is connected to the PC with Windows OS, there will be new disk volume found in the windows explorer.

REVISION HISTORY

REV.	DATE	DESCRIPTION
0.01	March 11, 2010	1. Initially issued.

Important Notice

Nuvoton products are not designed, intended, authorized or warranted for use as components in systems or equipment intended for surgical implantation, atomic energy control instruments, airplane or spaceship instruments, transportation instruments, traffic signal instruments, combustion control instruments, or for other applications intended to support or sustain life. Further more, Nuvoton products are not intended for applications wherein failure of Nuvoton products could result or lead to a situation wherein personal injury, death or severe property or environmental damage could occur.

Nuvoton customers using or selling these products for use in such applications do so at their own risk and agree to fully indemnify Nuvoton for any damages resulting from such improper use or sales.

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.