TOSHIBA

Application Note

USBD RAM Disk

TMPM365

Ver 1

Sep, 2017

TOSHIBA ELECTRONIC DEVICES & STORAGE CORPORATION

CMDR-M365UDM-01E

RESTRICTIONS ON PRODUCT USE

- DO NOT USE THIS SOFTWARE WITHOUT THE SOFTWARE LISENCE AGREEMENT.

Index

1. Introduction 1
2. MSC RAMdisk Demo 2
3. Enumeration 3
4. Descriptor 4
    4.1 Device Descriptor 4
    4.2 Steps to Get Descriptor 6
    4.3 MSC Request 7
5. Bulk-Only Transport 8
6. Software File Structure 9
7. Key Code and Flowchart 10
    7.1 Key code 10
    7.2 Flowchar 14
8. Call back function 18

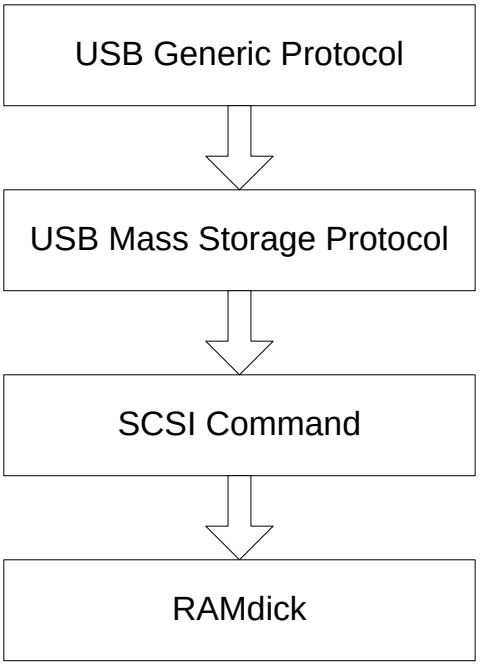
1. Introduction

The USBD MSC driver can be easily re-used by user to realize the Mass Storage Class.

This document will introduce the key points in creating a simple RAMdisk demonstration program, which is based on Toshiba TMPM365 Evaluation board and USBD driver.

It is assumed that the reader has had some basic knowledge, such as “descriptor”, “endpoint” and “report” for the USB.

The following picture shows the basic structure of the USB Mass Storage Framework.



2. MSC RAMdisk Demo

The USB mass storage device class, otherwise known as **USB MSC** or **UMS**, is a protocol that allows a Universal Serial Bus (USB) device to become accessible to a host computing device, to enable file transfers between the two.

The USB mass storage device class comprises a set of computing communications protocols defined by the USB Implementers Forum that run on the Universal Serial Bus. The standard provides an interface to a variety of storage devices.

Some of the devices that are connected to computers via this standard include:

- external magnetic hard drives
- external optical drives, including CD and DVD reader and writer drives
- portable flash memory devices
- …..

Devices which support this standard are referred to as **MSC** (Mass Storage Class) devices. While MSC is the official abbreviation, **UMS** (Universal Mass Storage) has become common in online jargon.

On TMPM365 evaluation board, a simple MSC demonstration program was designed to realize a “24k” RAM dick.

The prerequisite development environment must include:

- IAR Embedded Workbench for ARM 6.401 which should support M365
- M365 evaluation board (Not for sale)
  (Hereafter, we will call the evaluation board as “EVB” for short.)
- IAR J-Link-ARM v8.0
- PC (OS: Windows XP)

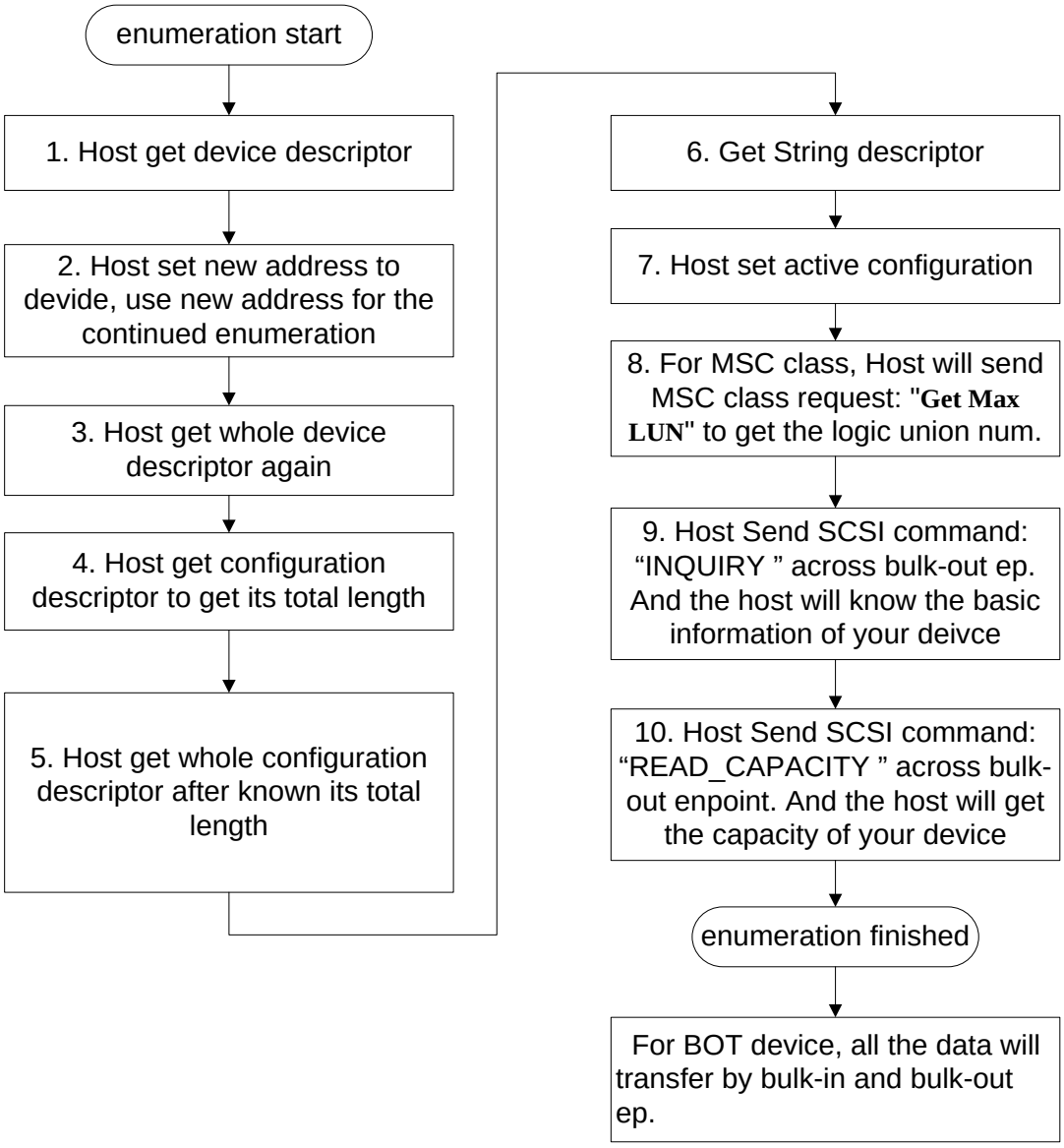
Once the EVB is connected via USB interface to PC and then powered on, it will be recognized to 24k RAMDisk. You can format it by the computer and save file into it.

3. Enumeration

Each time a device is plugged into the USB port of a host PC, it must be properly enumerated before entering its normal operating mode so as to transfer its data to the host PC.

The enumeration process contains some steps that host requests a number of descriptors to describe all attributes of the device.

Below are the simplified enumeration steps for MSC Class device:



4. Descriptor

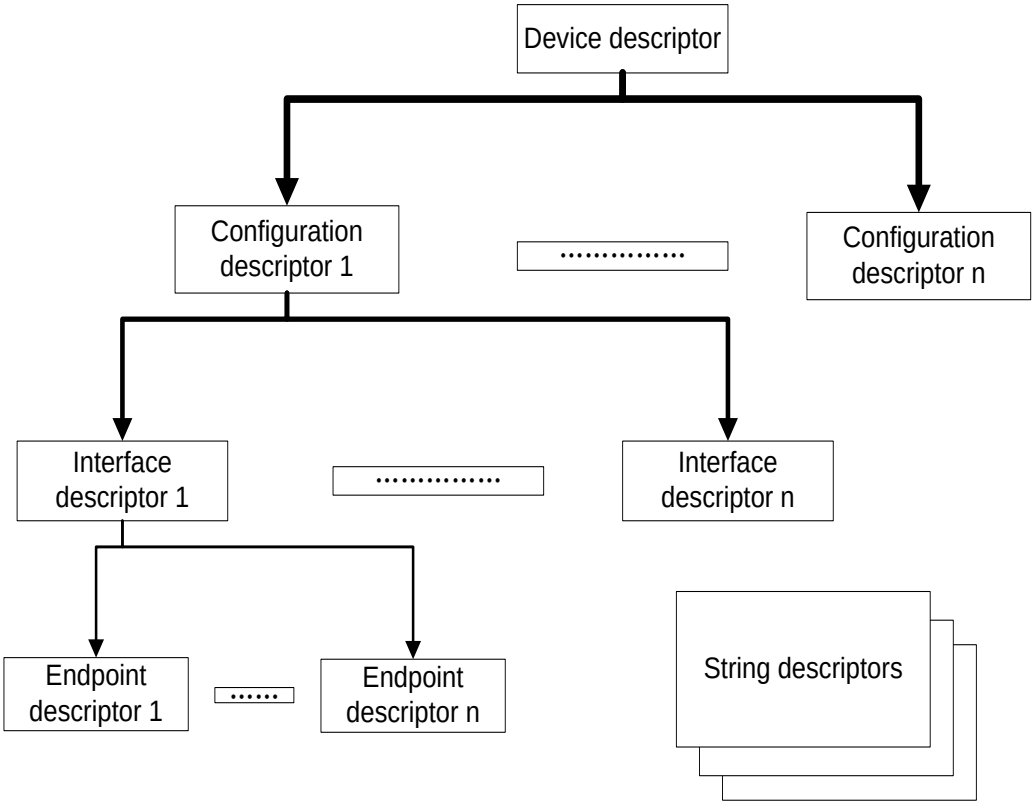
4.1 Device Descriptor

USB devices report their information to host by using descriptors.

A descriptor is a data structure with a specified format.
The first byte of descriptor indicates the total number of bytes in this descriptor.
The second byte identifies the descriptor type (Device, Configuration, Interface, Endpoint, Class, and so on…).
The remaining fields (bytes or words) are the content for this descriptor depending on its type.
(refer to ***usbd_descriptor_msc.c*** for more details)

A USB device has only one device descriptor and one or more configuration descriptor. Each configuration has one or more interface descriptor. Each interface has one or more endpoint descriptor.

The layer structure of descriptor is as shown in the figure below:



Interface Descriptors

The device shall support at least one interface, known herein as the Bulk-Only Data Interface.

In the RAMDick demo, we can get the array "**gInterfaceDescriptor1_0[]**" in the file ***usbd_descriptor_msc.c***

```
const InterfaceDescriptor_t gInterfaceDescriptor1_0 = {
    CbLength_Interface,              /* bLength                */
    CbDescripterType_Interface,      /* bDescriptorType        */
    0x00U,                           /* bInterfaceNumber       */
    0x00U,                           /* bAlternateSetting      */
    0x02U,                           /* bNumEndpoints          */
    0x08U,                           /* bInterfaceClass        */
    0x06U,                           /* bInterfaceSubClass     */
    0x50U,                           /* bInterfaceProtocol     */
    0x00U                            /* iInterface             */
};
```

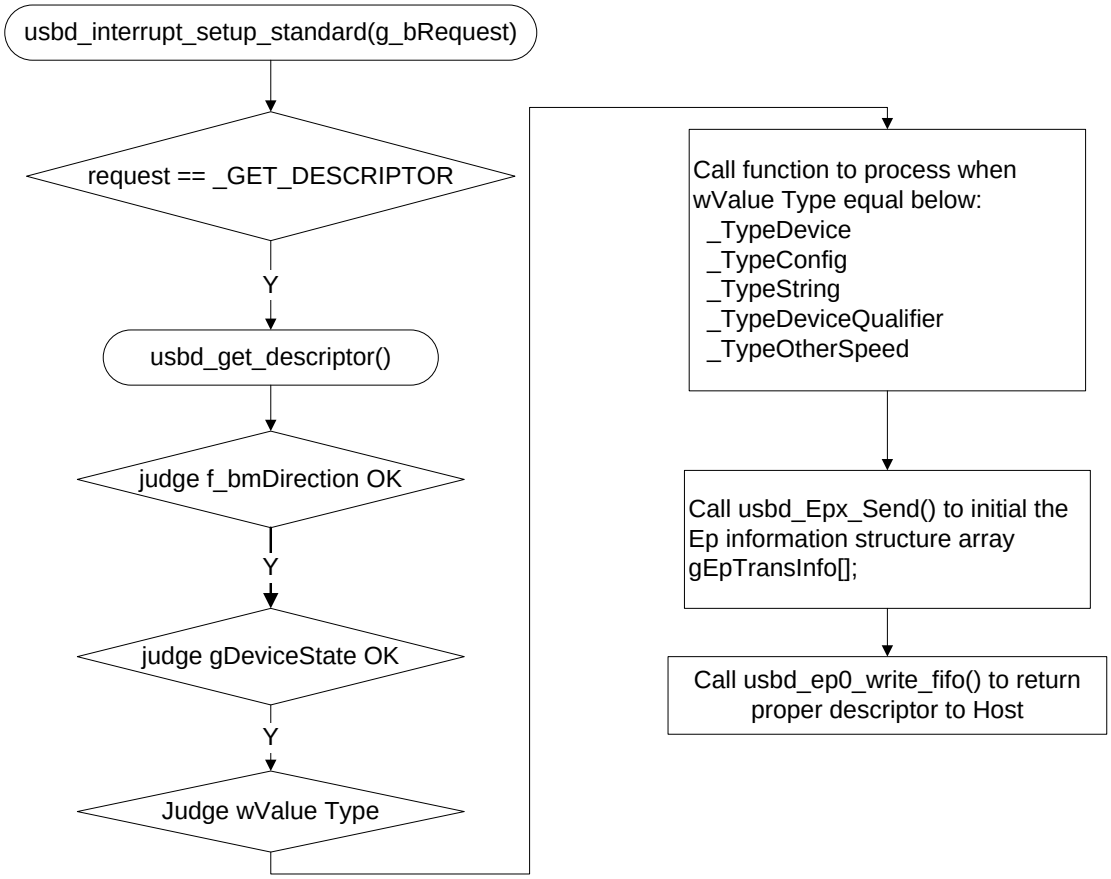
Let's focus on the following item. By these three items, the host will treat the device as Mass Storage device, and know how to communicate with it.

bInterfaceClass = 0x08 = Mass Storage
bInterfaceSubClass = 0x06 = SCSI Transparent
bInterfaceProtocol = 0x50 = Bulk Only Transport

4.2 Steps to Get Descriptor

As mentioned in enumeration part above, the most important work during enumeration is to return the required descriptor to host.

Below is the simplified flow for getting descriptors:



4.3 MSC Request

Bulk-Only Mass Storage Reset *(class-specific request)*

This request is used to reset the mass storage device and its associated interface.

This class-specific request shall ready the device for the next CBW from the host.

The host shall send this request via the default pipe to the device. The device shall preserve the value of its bulk data toggle bits and endpoint STALL conditions despite the Bulk-Only Mass Storage Reset.

The device shall NAK the status stage of the device request until the Bulk-Only Mass Storage Reset is complete.

To issue the Bulk-Only Mass Storage Reset the host shall issue a device request on the default pipe of:

- **bmRequestType:** Class, Interface, host to device
- **bRequest:** 255 (FFh)
- **wValue:** 0
- **wIndex:** the interface number
- **wLength:** 0

Get Max LUN *(class-specific request)*

The Get Max LUN device request is used to determine the number of logical units supported by the device. Logical Unit Numbers on the device shall be numbered contiguously starting from LUN0 to a maximum LUN of 15 (Fh).

To issue a Get Max LUN device request, the host shall issue a device request on the default pipe:

- **bmRequestType:** Class, Interface, device to host
- **bRequest:** 254 (FEh)
- **wValue:** 0
- **wIndex:** the interface number
- **wLength: 1**

5. Bulk-Only Transport

The RAMDick demo use bulk–only transport protocol. After enumeration, all the data transfer by the bulk-in and bulk-out endpoints.

The command block of **CBW** contain the **SCSI** command which to be executed by the device. The device shall interpret the command block, and perform as the command. At last, the device returns a CSW to host.

Figure 1 - Command/Data/Status Flow shows the flow for Command Transport, Data-In, Data-Out and Status Transport.

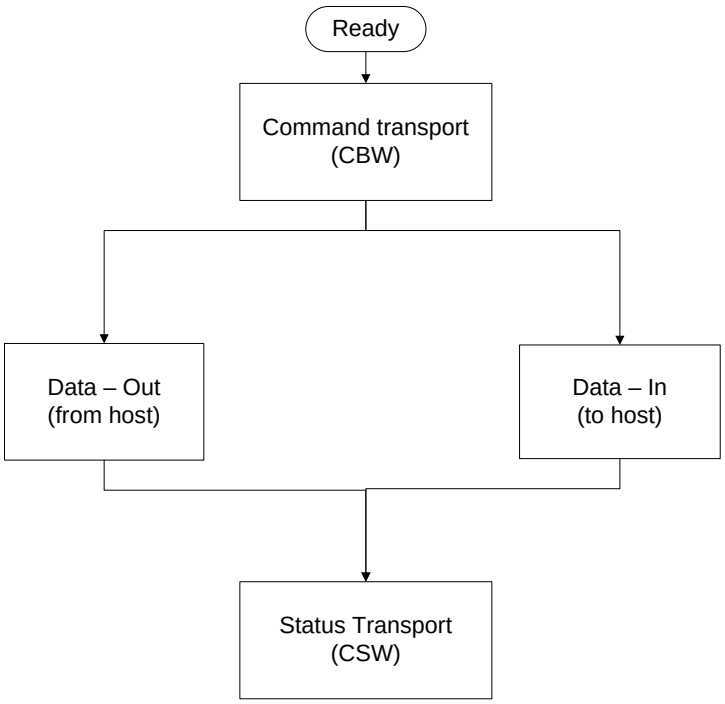


Figure 1 - Command/Data/Status

For more detail, please read the function:
_USBD_MSC_SCSI_Command_Analysis(uint8_t *pucData) in usbd_scsi.c and flowchart in section 7.2(4).

This function will interpret the CBW, and branch to the relevant command responding function.

6. Software File Structure

The main file structure of this demo is shown as below:

```
/---MSC_RAMDick
|     +---APP
|     |     |     msc_ramdisk.c
|     |     |
|     |     +---msc_inc
|     |     |           usbd_descriptor_msc.h
|     |     |           usbd_msc.h
|     |     |           usbd_scsi.h
|     |     |
|     |     \---msc_src
|     |                 usbd_descriptor_msc.c
|     |                 usbd_msc.c
|     |                 usbd_scsi.c
|     +---IAR
|     |           msc_ramdisk.ewd
|     |           msc_ramdisk.ewp
|     |           msc_ramdisk.eww
|     |
|     \---KEIL
|                 MSC_RAMDisk.uvopt
|                 MSC_RAMDisk.uvproj
\---USBD_Common
       +---inc
       |           usbd_descriptor.h
       |           usbd_device_request.h
       |           usbd_hw_com.h
       |           usbd_hw_interrupt.h
       |           usbd_trans.h
       |           usbd_typedefs.h
       |           usbd_var.h
       \---src
                   usbd_device_request.c
                   usbd_hw_com.c
                   usbd_hw_interrupt.c
                   usbd_trans.c
                   usbd_var.c
```

7. Key Code and Flowchart

The key codes and flowchart for this demo software are listed below:

7.1 Key code

1. Initialize the internal variable and buffer for receive and send data.

```
/* Initialization of an internal variable */
l_ucMediaStat          = false;
l_ulDataTransferLen = 0U;
l_ulLBA                = 0U;
l_usBlockNum           = 0U;
l_ulDataLenCount       = 0U;
l_usBlockCount         = 0U;
l_bRRState             = true;

/* Initialize buffers */
memset(l_aucSendBuff, 0x00U, BUFFER_SIZE);
memset(l_aucRecvBuff, 0x00U, BUFFER_SIZE);
```

2. Initialize MSC(SCSI) driver. The structure **“gSample_MSC_CB”** contain the address of call back function.

```
/* Set MSC(SCSI) initialize parameters */
gSample_MSC_CB.pfnConnStat             = _Sample_ConnStat;
gSample_MSC_CB.pfnMassStorageReset = _Sample_MassStorageReset;
gSample_MSC_CB.pfnGetMaxLun            = _Sample_GetMaxLun;
gSample_MSC_CB.pfnMediaRead            = _Sample_MediaRead;
gSample_MSC_CB.pfnMediaWrite           = _Sample_MediaWrite;
gSample_MSC_CB.pfnSendData             = _Sample_SendData;
gSample_MSC_CB.pfnRecvData             = _Sample_RecvData;

/* Initialize MSC(SCSI) driver */
USBD_MSC_SCSI_Init(&gSample_MSC_CB);
```

3. Initialize the RAM Disk.

```
static void _Sample_RAM_Disk_Init(void)
{
    /* Clear sector 0 */
    memset(g_aucRamDisk[0], 0x00U, BLOCK_SIZE);
}
```

4. Initialize work data

```
void usbd_initialize_standard_class_work_data(void)
{
	const ConfigDescriptor_t *config;
	ConfigDescriptor_bmAttributes_t *attribute;

	gUDC2Addr.All = CgUD2ADDR_INIT;
	gUDC2AddrBuf.All = CgUD2ADDR_INIT;

	fEP0StallFeature = CEPxStallFeature_OFF;
	fEP1StallFeature = CEPxStallFeature_OFF;
	fEP2StallFeature = CEPxStallFeature_OFF;
	fEP3StallFeature = CEPxStallFeature_OFF;

	gEP0Payload_Size = CwMaxPacketSize_EP0;
	gEP1Payload_Size = CwMaxPacketSize_EP1_INIT;
	gEP2Payload_Size = CwMaxPacketSize_EP2_INIT;
	gEP3Payload_Size = CwMaxPacketSize_EP3_INIT;

	gEPxConfigArray[USBD_EP1] = 0x88U;	/* EP1 as BULK IN for CDC,MSC */
	gEPxConfigArray[USBD_EP2] = 0x08U;	/* EP2 as BULK OUT for CDC,MSC */
	gEPxConfigArray[USBD_EP3] = 0x8CU;	/* EP3 as INTERRUPT IN for CDC,HID
*/

	s_Buf_Current_Config = CbConfigurationValue_Init;
	s_Buf_Current_Interface = CbInterfaceNumber_Init;
	s_Buf_Current_Alternate = CbAlternateSetting_Init;

	s_Current_Config = s_Buf_Current_Config;
	s_Current_Interface = s_Buf_Current_Interface;
	s_Current_Alternate = s_Buf_Current_Alternate;

	g_USB_Stage = IDLE_STAGE;

	memset(gEpTransInfo, 0x00U, sizeof(gEpTransInfo));

	/* make status device result	*/
	sGetStatusDevice = CsGetStatusDevice_INIT;
	config = gStructConfigDesc[CbConfigurationValue1 - 1].p_config;
	attribute = (ConfigDescriptor_bmAttributes_t *) config->bmAttributes;
	fSelfPowered = attribute->SelfPowered;
	fRemoteWakeup = attribute->RemoteWakeup;
```

```
    return;
}
```

5. Configure USBD module:
    - Enable USB clock supply
    - Enable the pull up resistor in D+ pin
    - Power on reset for USBD module
    - Set USB interrupt masks
    - Enable USBD interrupt
    - Reset endpoint 0

```
void Config_USB(void)
{
    USBD_PowerCtrl pwr = { 0U };

    TSB_CG->USBCTL = 0x100U;       /* enable USB Clock */

    /*USBON pin config*/
    GPIO_SetInput(GPIO_PG, GPIO_BIT_5);
    GPIO_SetInputEnableReg(GPIO_PG, GPIO_BIT_5, ENABLE);

    /* pin PA0 control the pull up of D+, must be set output '1' */
    GPIO_SetOutput(GPIO_PA, GPIO_BIT_0);
    GPIO_WriteDataBit(GPIO_PA, GPIO_BIT_0, GPIO_BIT_VALUE_1);

    USBD_SetINTMask(USBD_INT_USB_RESET_END, ENABLE);
    USBD_SetINTMask(USBD_INT_USB_RESET, ENABLE);

    /*   UDPWCTL Power Reset and                  */
    /*          PHY Reset & Suspend: 1ms          */
    pwr = USBD_GetPowerCtrlStatus();
    pwr.Bit.PW_Resetb = 0U;
    pwr.Bit.PHY_Resetb = 0U;
    pwr.Bit.PHY_Suspend = 1U;
    USBD_SetPowerCtrl(pwr);
    usbd_wait_1ms();

    /* PHY Reset off : 1ms                       */
    pwr = USBD_GetPowerCtrlStatus();
    pwr.Bit.PHY_Resetb = 1U;
    pwr.Bit.PHY_Suspend = 1U;
    USBD_SetPowerCtrl(pwr);
    usbd_wait_1ms();
```

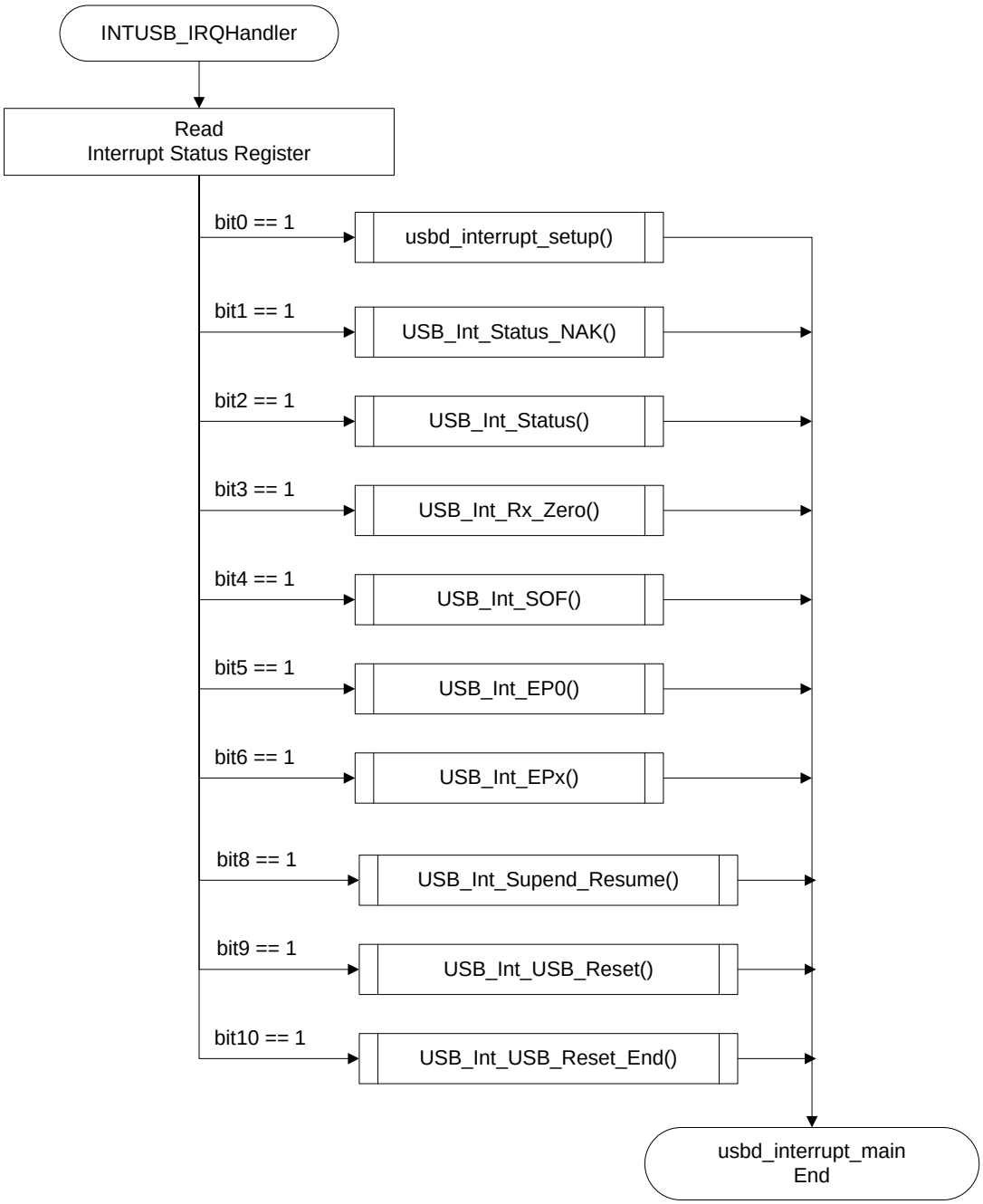
```
    /*   PHY Suspend off : 1ms                    */
    pwr = USBD_GetPowerCtrlStatus();
    pwr.Bit.PHY_Suspend = 0U;
    USBD_SetPowerCtrl(pwr);
    usbd_wait_1ms();
    usbd_wait_1ms();

    /*   UDPWCTL Power Reset Off: 1ms             */
    pwr = USBD_GetPowerCtrlStatus();
    pwr.Bit.PW_Resetb = 1U;
    USBD_SetPowerCtrl(pwr);
    usbd_wait_1ms();
    usbd_wait_1ms();
    pwr = USBD_GetPowerCtrlStatus();

    /* clear pending UDC2 interrupt and disable INT for SOF and NAK */
    USBD_WriteUDC2Reg(UDC2_INT, 0x90FFU);

    USBD_SetEPCMD(USBD_EP0, USBD_CMD_ALL_EP_INVALID);
    USBD_SetEPCMD(USBD_EP0, USBD_CMD_USB_READY);

    NVIC_EnableIRQ(INTUSB_IRQn);

// VBUS
    NVIC_EnableIRQ(INTUSBPON_IRQn);

    return;
}
```

6. After proper configuration, the firmware will wait for USB interrupt and then finish the standard enumeration steps so as to be treated as an RAMDisk in interrupt service routine. You can format it with the pc. After the formation, you can save file into it, or read file from it.

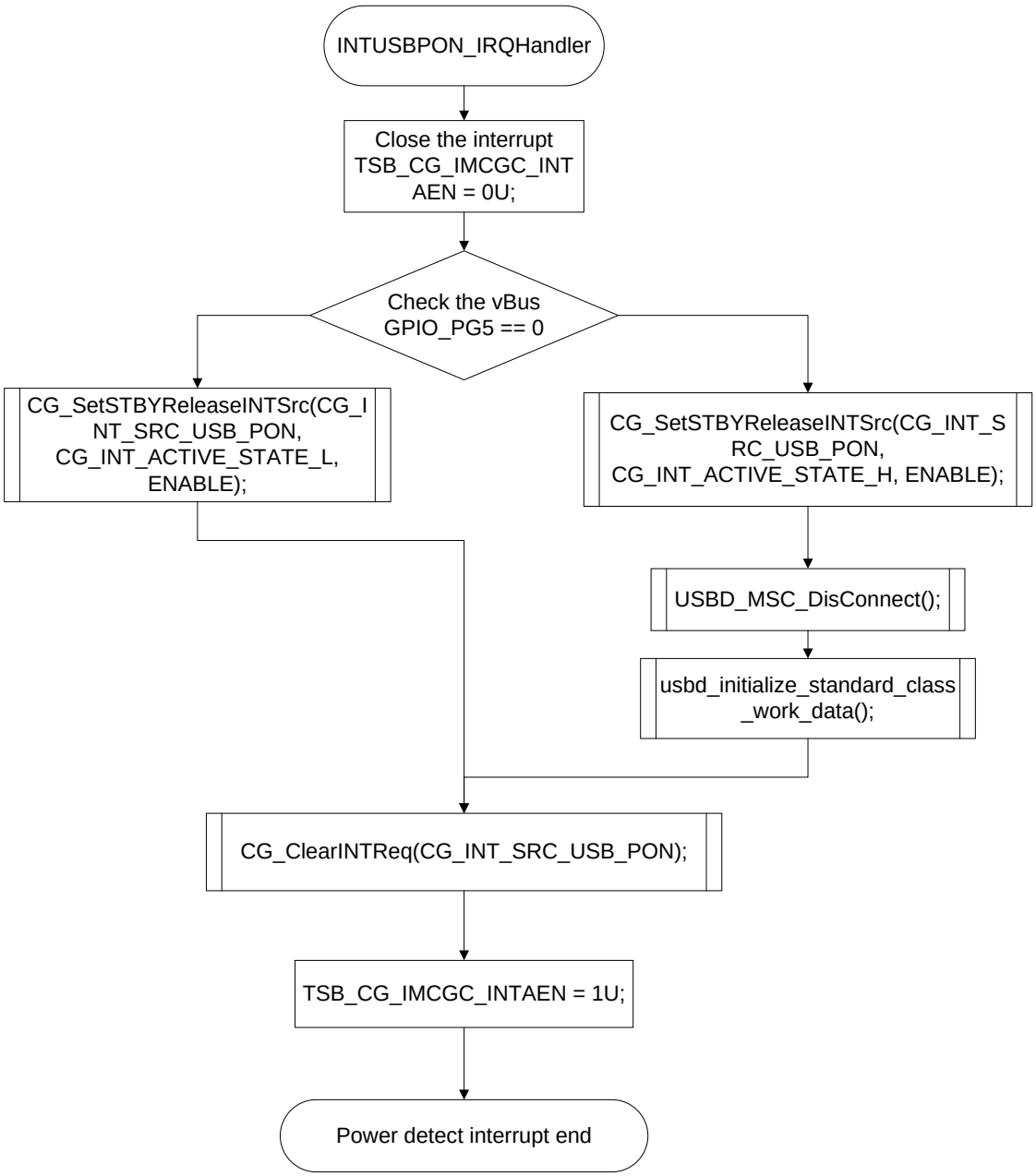
   For more details, please refer to the function:
   - INTUSB_IRQHandler() and INTUSBPON_IRQHandle() in **usbd_hw_interrupt.c**.
   - usbd_interrupt_setup_standard() in **usbd_device_request.c**
   - usbd_interrupt_MSC_req() in **usbd_msc.c**.
   - _USBD_MSC_SCSI_Command_Analysis(uint8_t *pucData) in **usbd_scsi.c**.

7.2 Flowchar

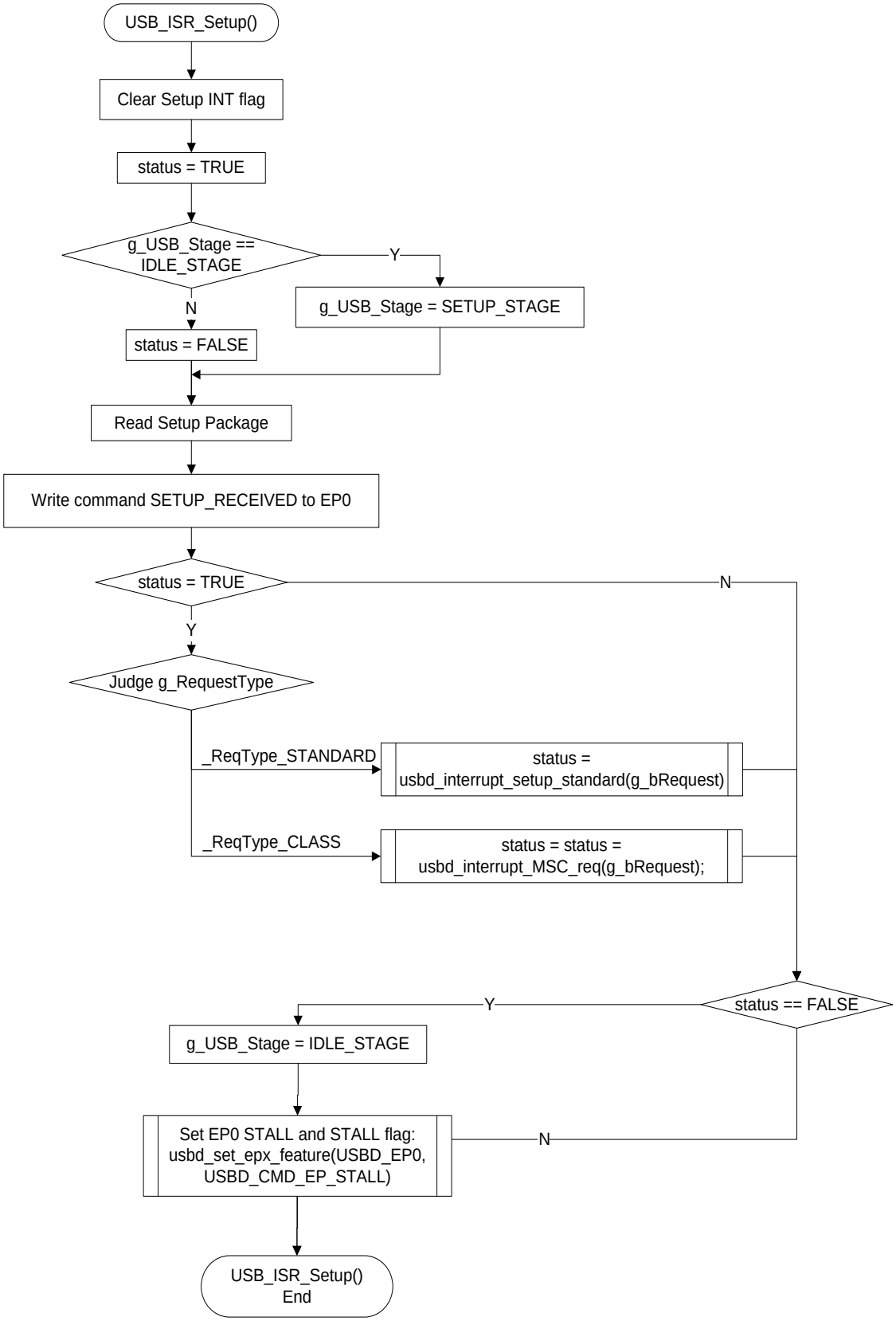
1) The following flowchart is for the main USB interrupts routine



2) The following flowchart is for VBUS detect Interrupt Service Routine.



3) The following flowchart is to get setup package and process it.



4) The following flowchart is to analysis the SCSI command carried by the CBW.
After receive the CBW, it will cause epx interrupt, and then the system will call function **_USBD_MSC_SCSI_Command_Analysis(uint8_t *pucData)** to analysis the **CBW.**

8. Call back function

The following structure is used to record the callback function and help to initial the system.

```
/* Callback Information */
typedef struct {
      MSC_CB_ConnStat                pfnConnStat;
      MSC_CB_MassStorageReset        pfnMassStorageReset;
      MSC_CB_GetMaxLun               pfnGetMaxLun;
      SCSI_CB_Media                  pfnMediaRead;
      SCSI_CB_Media                  pfnMediaWrite;
      SCSI_CB_TrnsData               pfnSendData;
      SCSI_CB_TrnsData               pfnRecvData;
} SCSI_CB_t;
```

They can be divided into three kind of function:
1. Hardware response:
pfnConnStat; Callback for connect/disconnect

2. MSC Class requests: they will be call when the host sent out the request to the device.
pfnMassStorageReset; pfnGetMaxLun;

3. SCSI Data transfer:
pfnMediaRead: Callback for read media.

pfnMediaWrite: Callback for write media.

pfnSendData: Callback for Send data (Read).Each time finish a data transfer to the host, this call back function will be called to judge whether it finish all data transferring.

pfnRecvData: Callback for received data (Write). Each time finish a data transfer from the host to the device, this call back function will be called to judge whether the device receive all the data.