6 Mass storage demo

This demo runs on the following STMicroelectronics evaluation boards, and can be easily tailored to any other hardware:

- STM3210B-EVAL
- STM3210E-EVAL
- STM32L152-EVAL
- STM32373C-EVAL
- STM32303C-EVAL
- STM32L152D-EVAL

To select the STMicroelectronics evaluation board used to run the demo, uncomment the corresponding line in the *platform_config.h* file.

6.1 General description

The mass storage demo gives a typical example of how to use the STM32 USB-FS_Device peripheral to communicate with the PC host using bulk transfer.

This demo supports the BOT (bulk only transfer) protocol and all needed SCSI (small computer system interface) commands, and is compatible with Windows XP (SP1, SP2, SPI3), Windows 2000 (SP4), Windows Vista and Windows 7.

6.2 Mass storage demo overview

The mass storage demo complies with USB 2.0 and USB mass storage class (bulk-only transfer subclass) specifications. After running the application, the user just has to plug the USB cable into a PC Host and the device is automatically detected without any additional drive (with Win 2000, XP, Vista and Windows 7). A new removable drive appears in the system window and write/read/format operations can be performed as with any other removable drive (see *Figure 7*).

Figure 7. New removable disk in Windows

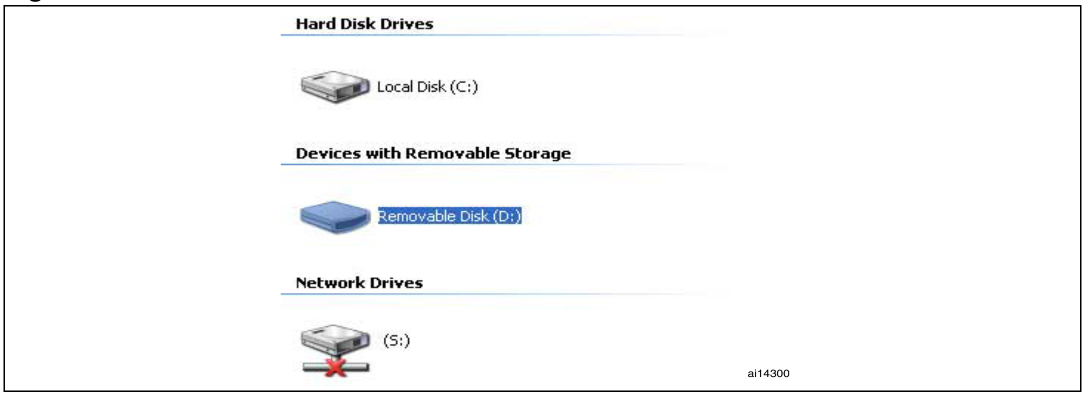


Table 11 gives details of the memory support used for each eval board.

Table 11. Eval board memory support

Eval board	Memory support	IP interface
STM3210E-EVAL	MicroSD and NAND Flash	SDIO and FSMC
STM3210B-EVAL	MicroSD	SPI
STM32L152-EVAL	MicroSD	SPI
STM32L152D-EVAL	MicroSD	SDIO
STM32373C-EVAL	MicroSD	SPI
STM32303C-EVAL	MicroSD	SPI

Note: All related firmware used to initialize, read from and write to the media are available in the stm32xxx_eval_sdio_sd.c.c/.h, stm32xxx_eval_spi_sd.c/.h and fsmc_nand.c/.h files.

Note: For mass storage class, the device firmware does not need to know or take into account the file system the host is using. The firmware just stores and sends blocks of data as requested by the host.

6.3 Mass storage protocol

6.3.1 Bulk-only transfer (BOT)

The BOT protocol uses only bulk pipes to transfer commands, status and data (no interrupt or control pipes). The default pipe (pipe 0, or in other words, Endpoint 0) is only used to clear the bulk pipe status (clear STALL status) and to issue the two class-specific requests: Mass Storage reset and Get Max LUN.

Command transfer

To send a command, the host uses a specific format called command block wrapper (CBW). The CBW is a 31-byte length packet. *Table 12* shows the different fields of a CBW.

Table 12. CBW packet fields

	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)			bCBWCBLength				
15-30	CBWCB							

- ***dCBWSignature***: 43425355 : *USBC* (in little Endian)
- ***dCBWTag***: The host specifies this field for each command. The device should return the same *dCBWTag* in the associated status.
- ***dCBWDataTransferLength***: total number of bytes to transfer (expected by the host).
- ***bmCBWFlags***: This field is used to specify the direction of the data transfer (if any). The bits of this field are defined as follows:
  - **Bit 7**: Direction bit:

    0: Data Out transfer (host to device).

    1: Data In transfer (device to host).

    Note: The device ignores this bit if the `dCBWDataTransferLength` *field is cleared to zero.*
  - **Bits 6:0**: reserved (cleared to zero).
- ***bCBWLUN***: concerned Logical Unit number.
- ***bCBWCBLength****:* this field specify the length (in bytes) of the command CBWCB.
- ***CBWCB***: the command block to be executed by the device.

Status transfer

To inform the host about the status of each received command, the device uses the command status wrapper (CSW). *Table 13* shows the different fields of a CSW.

Table 13. CSW packet fields

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

- ***dCSWSignature***: 53425355 *USBS* (little Endian).
- ***dCSWTag***: the device sets this field to the received value of *dCBWTag* in the concerned CBW.
- ***dCSWDataResidue***: the difference between the expected data (the value of the *dCBWDataTransferLength* field of the concerned CBW) and the real value of the data received or sent by the device.
- ***bCSWStatus***: the status of the concerned command. This field can assume one of the three non-reserved values shown in *Table 14*.

Table 14. Command block status values

Value	Description
0x00	Command passed
0x01	Command failed
0x02	Phase error
0x03=>0xFF	Reserved

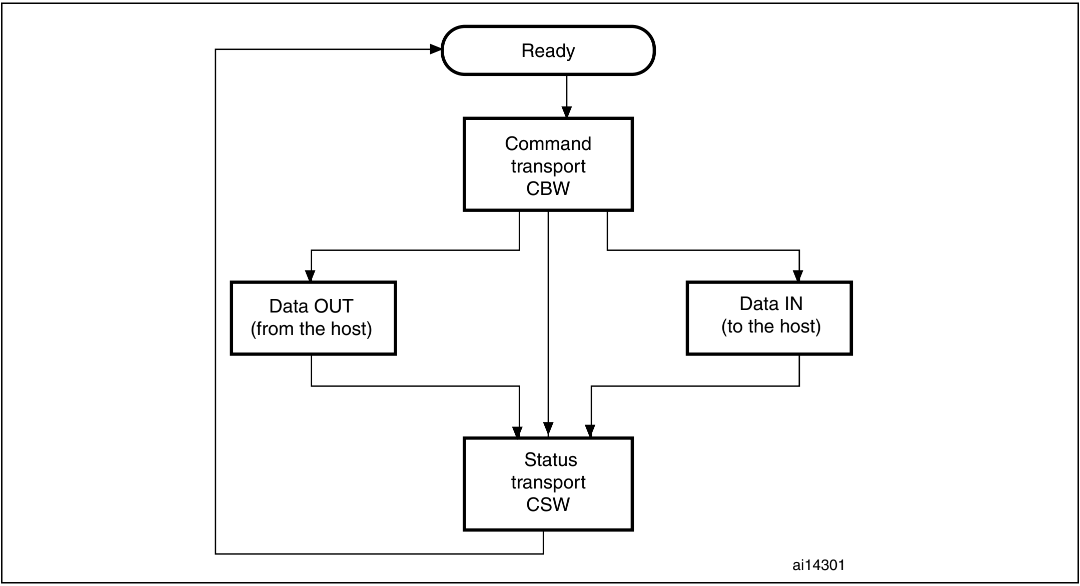
Data transfer

The data transfer phase is specified by the *dCBWDataTransferLength* and *bmCBWFlags* of the correspondent CBW. The host attempts to transfer the exact number of bytes to or from the device.

The diagram shown in *Figure 8* shows the state machine of a BOT transfer.

Note: For more information about the BOT protocol, please refer to the “Universal Serial Bus Mass Storage Class Bulk-Only Transport” specification.

Figure 8. BOT state machine



6.3.2 Small computer system interface (SCSI)

The SCSI command set is designed to provide efficient peer-to-peer operation of SCSI devices like, for example, hard desks, tapes and mass storage devices. In other words these are used to ensure the communication between an SCSI device and an operating system in a PC host.

Table 15 shows SCSI commands for removable devices. Not all commands are shown. For more information, please refer to the SPC and RBC specifications.

Table 15. SCSI command set

Command name	OpCode	Command support[1]	Description	Reference
Inquiry	0x12	M	Get device information	SPC-2
Read Format Capacities	0x23	M	Report current media capacity and formattable capacities supported by medium	SPC-2
Mode Sense (6)	0x1A	M	Report parameters to the host	SPC-2
Mode Sense (10)		M	Report parameters to the host	SPC-2
Prevent\ Allow Medium Removal	0x1E	M	Prevent or allow the removal of media from a removable media device	SPC-2
Read (10)	0x28	M	Transfer binary data from the medium to the host	RBC
Read Capacity (10)	0x25	M	Report current medium capacity	RBC
Request Sense	0x03	O	Transfer status sense data to the host	SPC-2
Start Stop Unit	0x1B	M	Enable or disable the Logical Unit for medium access operations and controls certain power conditions	RBC
Test Unit Ready	0x00	M	Request the device to report if it is ready	SPC-2
Verify (10)	0x2F	M	Verify data on the medium	RBC
Write (10)	0x2A	M	Transfer binary data from the host to the medium	RBC

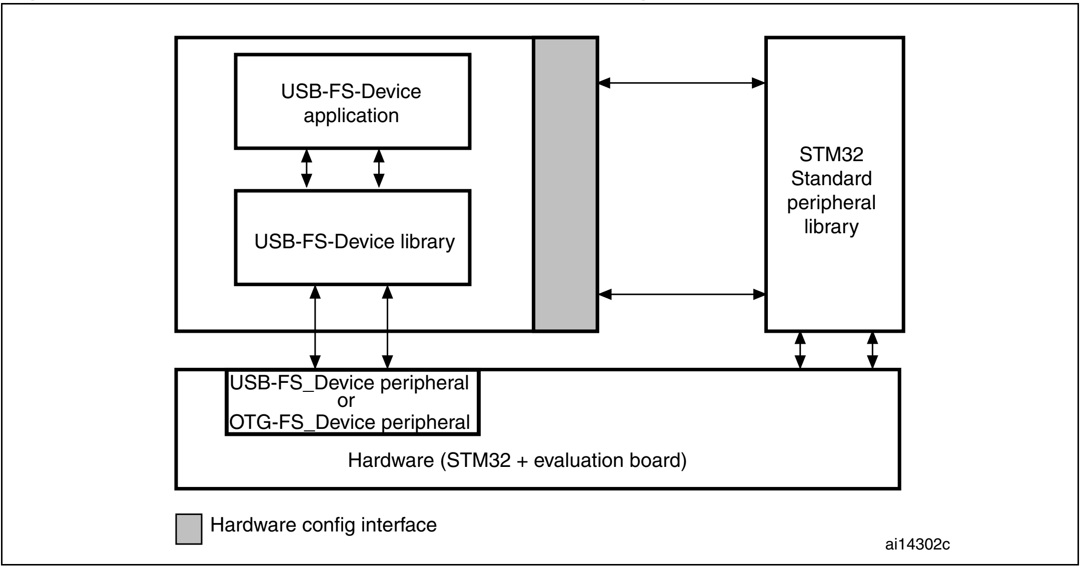
1. Command Support key: M = support is mandatory, O = support is optional.

6.4 Mass storage demo implementations

6.4.1 Hardware configuration interface

The hardware configuration interface is a layer between the USB application (in our case the Mass Storage demo) and the internal/external hardware of the STM32 microcontroller. This internal and external hardware is managed by the STM32 standard peripheral library, so from the firmware point of view, the hardware configuration interface is the firmware layer between the USB application and the standard peripheral library. *Figure 9* shows the interaction between the different firmware components and the hardware environment.

Figure 9. Hardware and firmware interaction diagram


USB-FS-Device application
USB-FS-Device library
STM32 Standard peripheral library
USB-FS_Device peripheral or OTG-FS_Device peripheral
Hardware (STM32 + evaluation board)
Hardware config interface
ai14302c

The hardware configuration layer is represented by the two files *HW_config.c* and *HW_config.h*. For the Mass Storage demo, the hardware management layer manages the following hardware requirements:

- System and USB-FS_Device peripheral clock configuration
- Read and write LED configuration
- LED command
- Initialize the memory medium
- Get the characteristics of the memory medium (the block size and the memory capacity)

6.4.2 Endpoint configurations and data management

This section provides a description of the configuration and the data flow according to the transfer mode.

Endpoint configurations

The endpoint configurations should be done after each USB reset event, so this part of code is implemented in the `MASS_Reset` function (file *usp_prop.c*).

For all STM32 except Connectivity line devices:

To configure endpoint 0 it is necessary to:

- Configure endpoint 0 as the default control endpoint
- Configure the endpoint 0 Rx and Tx count and buffer addresses in the BTABLE (*usb_conf.h* file)
- Configure the endpoint Rx status as VALID and the Tx status as NAK.

The bulk pipes (endpoints 1 and 2) are configured as follows:

1. Configure endpoint 1 as bulk IN
2. Configure the endpoint 1 Tx count and data buffer address in the BTABLE (*usb_conf.h* file)
3. Disable the endpoint 1 Rx
4. Configure the endpoint 1 Tx status as NAK
5. Configure the endpoint 2 as bulk OUT
6. Configure the endpoint 2 Rx count and data buffer address in the BTABLE (*usb_conf.h* file)
7. Disable the endpoint 2 Tx
8. Configure the endpoint 2 Rx status as VALID.

Data management

Data management consists of the transfer of the needed data directly from the specified data buffer address in the USB memory, according to the related endpoint (IN: ENDP1TXADDR; OUT: ENDP2RXADDR). For these transfers, the following two functions are used (*usb_sil.c* file):

- **USB_SIL_Read ()**: this function transfers the received bytes from the USB memory to the internal RAM. This function is used to copy the data sent by the host to the device. The number of received data bytes is determined into the function (not passed as parameter) and this value is returned by the function at the end of the operation.
- **USB_SIL_Write ()**: this function transfers the specified number of bytes from the internal RAM to the USB memory. This function is used to send the data from the device to the host.

6.4.3 Class-specific requests

The Mass Storage Class specification describes two class-specific requests:

Bulk-only mass storage reset

This request is used to reset the Mass Storage device and its associated interface. This class-specific request makes the device ready for the next CBW sent by the PC host.

To issue the bulk-only mass storage reset, the host issues a device request on the default pipe (endpoint 0) of:

- *bmRequestType*: Class, Interface, Host to device
- *bRequest* field set to 0xFF
- *wValue* field set to 0
- *wIndex* field set to the interface number (0 for this implementation)
- *wLength* field set to 0

This request is implemented as a no-data class-specific request in the `MASS_NoData_Setup()` function (*usb_prop.c* file).

After receiving this request, the device clears the data toggle of the two bulk endpoints, initializes the CBW signature to the default value and sets the BOT state machine to the BOT_IDLE state to be ready to receive the next CBW.

GET MAX LUN request

A Mass Storage Device may implement several logical units that share common device characteristics. The host uses bCBWLUN to designate which logical unit of the device is the destination of the CBW.

The Get Max LUN device request is used to determine the number of logical units supported by the device.

To issue a Get Max LUN request the host must issue a device request on the default pipe (endpoint 0) of:

- *bmRequestType*: Class, Interface, Host to device
- *bRequest* field set to 0xFE
- *wValue* field set to 0
- *wIndex* field set to the interface number (0 for this implementation)
- *wLength* field set to 1

This request is implemented as a data class-specific request in the `MASS_Data_Setup()` function (*usb_prop.c* file). Note that in case of the STM3210E-EVAL board two LUNs are supported

6.4.4 Standard request requirements

To be compliant with the BOT specification the device must respond to the two following requirements after receiving the same standard requests:

- When the device switches from the unconfigured to the configured state, the data toggle of all endpoints must be cleared. This requirement is served by the `Mass_Storage_SetConfiguration()` function in the *usb_prop.c* file.
- When the host sends a CBW command with an invalid signature or length, the device must keep endpoints 1 and 2 both as STALL until it receives the Mass Storage Reset class-specific request. This functionality is managed by the `Mass_Storage_ClearFeature()` function in the *usb_prop.c* file.

6.4.5 BOT state machine

To provide the BOT protocol, a specific state machine with five states is implemented. The states are described below:

- **BOT_IDLE**: this is the default state after a USB reset, Bulk-Only Mass Storage Reset or after sending a CSW. In this state the device is ready to receive a new CBW from the host
- **BOT_DATA_OUT**: the device enters this state after receiving a CBW with data flow from the host to the device
- **BOT_DATA_IN**: the device enters this state after receiving a CBW with data flow from the device to the host
- **BOT_DATA_IN_LAST**: the device enters this state when sending the last of the data asked for by the host
- **BOT_CSW_SEND**: the device moves to this state when sending the CSW. When the device is in this state and a correct IN transfer occurs, the device moves to the BOT_IDLE state to be able to receive the next CBW
- **BOT_ERROR:** Error state

The BOT state machine is managed using the functions described below (*usb_bot.c* and *usb_bot.h* firmware files):

- **Mass_Storage_In ()**; **Mass_Storage_Out ()**: these two functions are called when a correct transfer (IN or OUT) occurs. The aim of these two functions is to provide the next step after receiving/sending a CBW, data or CSW
- **CBW_Decode ()**: this function is used to decode the CBW and to dispatch the firmware to the corresponding SCSI command
- **DataInTransfer ()**: this function is used to transfer the characteristic device data to the host
- **Set_CSW ()**: this function is used to set the CSW fields with the needed parameters according to the command execution
- **Bot_Abort ()**: this function is used to STALL the endpoints 1 or 2 (or both) according to the Error occurring in the BOT flow

6.4.6 SCSI protocol implementation

The aim of the SCSI Protocol is to provide a correct response to all SCSI commands needed by the operating system on the PC host. This section details the method of management for all implemented SCSI commands.

- **INQUIRY** command (OpCode = 0x12):

  Send the needed inquiry page data (in this demo only page 0 and the standard page are supported) with the needed data length according to the *ALLOCATION LENGTH* field of the command.
- **SCSI READ FORMAT CAPACITIES** command (OpCode = 0x23):

  Send the Read Format Capacity data response (`ReadFormatCapacity_Data[ ]` from the *SCSI_data.c* file) after checking the presence of the medium. If no medium has been detected a MEDIUM_NOT_PRESENT error is returned to force the host to update its internal parameters.
- **SCSI READ CAPACITY (10)** command (OpCode = 0x25):

  Send the Read Capacity (10) data response (`ReadCapacity10_Data[ ]` from the *SCSI_data.c* file) after checking the presence of the medium. If no medium has been detected a MEDIUM_NOT_PRESENT error is returned to force the host to update its internal parameters.
- **SCSI MODE SENSE (6)** command (OpCode = 0x1A):

  Send the Mode Sense (6) data response (`Mode_Sense6_data[ ]` from the *SCSI_data.c* file).
- **SCSI MODE SENSE (10)** command (OpCode = 0x5A):

  Send the Mode Sense (10) data response (`Mode_Sense10_data[ ]` from the *SCSI_data.c* file).
- **SCSI REQUEST SENSE** command (OpCode = 0x03):

  Send the Request Sense data response. Note that the `Resquest_Sense_Data [ ]` array (*SCSI_data.c* file) is updated using the `Set_Scsi_Sense_Data()` function in order to set the *Sense key* and the *ASC* fields according to any error occurring during the transfer.
- **SCSI TEST UNIT READY** command (OpCode = 0x00):

  Check the presence of the medium. If no medium has been detected a MEDIUM_NOT_PRESENT error is returned to force the host to update its internal parameters.

- **SCSI PREVENT/ALLOW MEDIUM REMOVAL** command (OpCode = 0x1E):
  Always return a CSW with COMMAND PASSED status.
- **SCSI START STOP UNIT** command (OpCode = 0x1B):
  This command is sent by the PC host when a user right-clicks on the device (in Windows) and selects the Eject operation. In this case the firmware programs the data in the internal Flash memory using the `Stor_Data_In_Flash()` function.
- **SCSI READ 10** command (OpCode = 0x28) and **SCSI WRITE 10** command (OpCode = 0x2A):
  The host issues these two commands to perform a read or a write operation. In these cases the device has to verify the address compatibility with the memory range and the direction bit in the bmFlag of the command. If the command is validated the firmware launches the read or write operation from the microSD card.
- **SCSI VERIFY 10** command (OpCode =0x2F):
  The SCSI VERIFY 10 command requests the device to verify the data written on the medium. In this case no Flash-like memory support is used, so when the SCSI VERIFY 10 command is received, the device tests the BLKVFY bit. If the BLKVFY bit is set to one, a Command Passed status is returned in the CSW.

6.4.7 Memory management

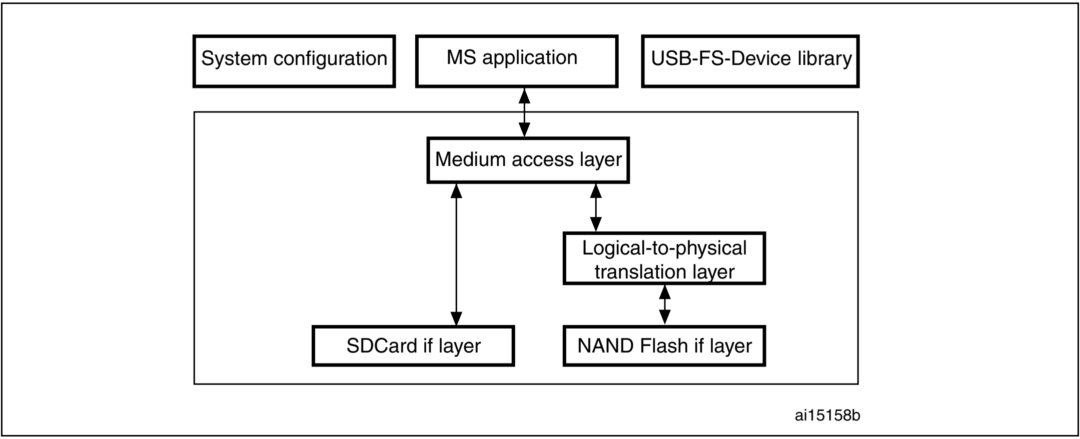
All the memory management functions are grouped in the two files: *memory.c* and *memory.h.* Memory management consists of two basic processes:

- Management and validation of the address range for the SCSI READ (10) and SCSI WRITE (10) commands: this process is done by the `Address_Management_Test()` function. The role of this function is to extract the real address and memory offset in the medium memory and test if the current transfer (Read or Write) is in the memory range. If this is not the case, the function STALLs endpoint 1 or 2 or both endpoints (according to the transfer Read or Write) and returns a bad status to disable the transfer.
- Management of the Read and Write processes: this process is done by the two functions `Read_Memory()` and `Write_Memory()`. These two functions manage the medium access based on the two functions “MAL_WriteBlock” and “MAL_ReadBlock” from the *mass_mal.c* file. After each access, the current memory offset and the next Access Address are updated using the length of the previous transfer.

6.4.8 Medium access management

Logical access to the addressed medium takes place in a separate layer called the medium access layer (mass_mal.c and mass_mal.h) through the logical unit number (LUN). This layer makes the medium access independent of the upper layer and dispatches write and read operations to the addressed medium.

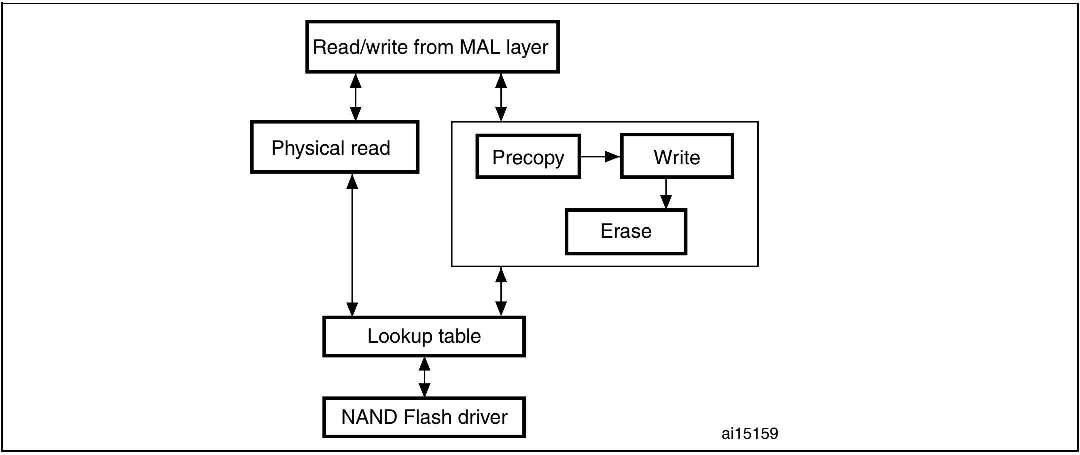
Figure 10. Medium access layer



Physical access to the NAND and physical access to the micro SD are not similar. In the case of the micro SD, write, read and erase operations can be made by page units known as logical sectors. This means that access to the medium is linear and the logical address is the same as the physical one. In the case of the NAND, write and read operations can be made by page unit but erase operations are carried out by block unit. This means that a write operation in a used block is performed in five steps as follows:

1. Allocate a free physical block.
2. Precopy old pages.
3. Write new pages.
4. Erase the old block.
5. Assign the current logical address to the new block.

Figure 11. NAND write operation



The logical-to-physical layer is used to keep a compatibility between the NAND and the microSD access methods by using the same input parameters for the two media. In the case of the NAND, the physical address is calculated internally and write and read operations are carried out in this layer.

Caution: The build look-up table (LUT) process used to translate logical addresses to physical ones and keep the block status is patented by STMicroelectronics. It is not allowed to use outside the STM32 firmware, and it should not be reproduced without STMicroelectronics's agreement.

6.5 How to customize the mass storage demo

The implemented firmware is a simple example used to demonstrate the STM32 USB peripheral capability in bulk transfer. However it can be customized according to user requirements. This customizing can be done in the three layers of the implemented mass storage protocol:

- **Customizing the BOT layer:** the user can implement their own BOT state machine or modify the implemented one just by modifying the two files *usb_BOT.c* and *usb_BOT.h* and by keeping the same data transfer method.
- **Customizing the SCSI layer:** the implemented SCSI protocol presents, more than the supported command listed in *Section 6.4.6: SCSI protocol implementation*, a list of unsupported commands. When the host sends one of these commands, a corresponding function is called by the `CBW_Decode()` function like a common command. However, all the functions related to unsupported commands are defined by the `SCSI_Invalid_Cmd()` function, (see *usb_scsi.c* file). The `SCSI_Invalid_Cmd()` function STALLs the two endpoints (1 and 2), sets the Sense data to *invalid command key* and sends a CSW with a *Command Failed* status.
  To support one of the invalid commands, the user has to comment out the concerned line and implement their own process. For example, for the need to support the `SCSI_FormatUnit` command, comment the line:
  ```
  // #define SCSI_FormatUnit_Cmd SCSI_Invalid_Cmd
  ```
  And implement a process in a function with the same name in the *usb_scsi.c* file:
  ```
  void SCSI_Invalid_Cmd (void)
  {
  // your implementation
  }
  ```
  In this way the custom function is called automatically by the `CBW_Decode()` function (*usb_BOT.c* file).

  However if you need to implement a command not listed in the previous list you have to modify the `CBW_Decode()` and implement the protocol of the new command.

Mass storage descriptors

Table 16. Device descriptor

Field	Value	Description
bLength	0x12	Size of this descriptor in bytes
bDescriptortype	0x01	Descriptor type (device descriptor)
bcdUSB	0x0200	USB specification release number: 2.0
bDeviceClass	0x00	Device Class
bDeviceSubClass	0x00	Device subclass
bDeviceProtocol	0x00	Device protocol
bMaxPacketSize0	0x40	Max Packet Size of Endpoint 0: 64 bytes
idVendor	0x0483	Vendor identifier (STMicroelectronics)
idProduct	0x5720	Product identifier
bcdDevice	0x0100	Device release number: 1.00

Table 16. Device descriptor (continued)

Field	Value	Description
iManufacturer	4	Index of the manufacturer String descriptor: 4
iProduct	42	Index of the product String descriptor: 42
iSerialNumber	96	Index of the serial number String descriptor
bNumConfigurations	0x01	Number of possible configurations: 1

Table 17. Configuration descriptor

Field	Value	Description
bLength	0x09	Size of this descriptor in bytes
bDescriptortype	0x02	Descriptor type (configuration descriptor)
wTotalLength	32	Total length (in bytes) of the returned data by this descriptor (including interface endpoint descriptors)
bNumInterfaces	0x0001	Number of interfaces supported by this configuration (only one interface)
bConfigurationValue	0x01	Configuration value
iConfiguration	0x00	Index of the Configuration String descriptor
bmAttributes	0x80	Configuration characteristics: Bus powered
Maxpower	0x32	Maximum power consumption through USB bus: 100 mA

Table 18. Interface descriptors

Field	Value	Description
bLength	0x09	Size of this descriptor in bytes
bDescriptortype	0x04	Descriptor type (Interface descriptor)
bInterfaceNumber	0x00	Interface number
bAlternateSetting	0x00	Alternate Setting number
bNumEndpoints	0x02	Number of used Endpoints: 2
bInterfaceClass	0x08	Interface class: Mass Storage class
bInterfaceSubClass	0x06	Interface subclass: SCSI transparent
bInterfaceProtocl	0x50	Interface protocol: 0x50
iInterface	106	Index of the interface String descriptor

Table 19. **Endpoint descriptors**

Field	Value	Description
IN endpoint		
bLength	0x07	Size of this descriptor in bytes
bDescriptortype	0x05	Descriptor type (endpoint descriptor)
bEndpointAddress	0x81	IN endpoint address 1.
bmAttributes	0x02	Bulk endpoint
wMaxPacketSize	0x40	64 bytes
bInterval	0x00	Does not apply for bulk endpoints
OUT endpoint		
bLength	0x07	Size of this descriptor in bytes
bDescriptortype	0x05	Descriptor type (endpoint descriptor)
bEndpointAddress	0x02	Out endpoint address 2
bmAttributes	0x02	Bulk endpoint
wMaxPacketSize	0x40	64 bytes
bInterval	0x00	Does not apply for bulk endpoints