AT91 USB Mass Storage Device Driver Implementation

AT91 ARM Thumb Microcontrollers

Application Note

1. Introduction

The **Mass Storage** (MS) class is an extension to the USB specification that defines how mass storage devices, such as a hard-disk, a disk-on-key or a USB floppy drive should operate on the USB.

Using the USB for a mass storage device has two advantages. First, **software development is much faster**, since most operating systems (if not all) include a fully-functional driver for the MSD class. Thus, only the device-side driver has to be developed to have a working product. In addition, relying on the Mass Storage standard guarantees that a device will be **useable on many operating systems** without any extra coding.

This Application Note describes how to implement a Mass Storage Device (MSD) driver with the **AT91 USB Framework** provided by Atmel® and developed for AT91 ARM® Thumb®-based microcontrollers. First, generic information about MSD-specific definitions and requirements is given. This document then details how to use the MSD class to create a hard disk operating on the internal Flash memory of a chip.

2. Related Documents

[1] Atmel Corp., AT91 USB Framework, lit. num. 6269

[2] T10, SCSI Block Commands - 3 (SBC-3), Revision 7, September 22, 2006.

[3] T10, SCSI Primary Commands - 4 (SPC-4), Revision 6, July 18, 2006.

[4] USB Mass Storage Class Bulk-Only Transport, Revision 1.0, September 31, 1999.

[5] USB Mass Storage Class Compliance Test Specification, Revision 0.9a, June 30, 2005.

[6] USB Mass Storage Class Specification Overview, Revision 1.2, June 23, 2003.

6283B–ATARM–17-Jul-09

3. Mass Storage Class Basics

This section gives generic details on the MSD class, including its purpose, architecture and how it is supported by various operating systems.

3.1 Purpose

The MSD class defines how devices such as a hard disk, a USB floppy disk drive or a disk-on-key shall operate on the USB. These devices are referred to as mass storage devices, since they usually offer a high storage capacity. When plugged to a PC, a device complying to the MSD specification is accessed like any other disk on the system.

Many devices use the MSD class in various ways. The simplest use is for disk-on-keys, which offer a portable storage with a high capacity compared to traditional floppy disks. External USB hard-drives are also common; they enable quick and easy connection to any system. Devices like MP3 & video players use MSD to simplify data transfer to and from the player. Finally, on some products the internal memory can be accessed through a MSD interface, allowing easy upgrade of the firmware.

In practice, the specification only defines a way to wrap existing data transfer protocols, such as SCSI or the Reduced Block Commands (RBC) set. A list of the supported protocols and their uses is given in the following section.

3.2 Data Transfer Protocols

The *Mass Storage Class Specification Overview 1.2* supports the following set of devices for an MSD device:

Table 3-1. Supported Protocols for MSD Devices

Subclass Code	Command Block Specification	Used by
01h	Reduced Block Commands (RBC)	Flash devices
02h	SFF-8020i, MMC-2	CD & DVD devices
03h	QIC-157	Tape devices
04h	UFI	Floppy disk drives
05h	SFF-8070i	Floppy disk drives
06h	SCSI transparent command set	Any

The SCSI transparent command set comprises all SCSI-related specifications, such as SCSI Primary Commands (SPC), SCSI Block Commands (SBC), and so on. A command will be issued by the host to determine exactly with which standard the device is compliant.

The protocol used by the device is specified in its Interface descriptor, in the *bInterfaceSubclass* field.

3.3 Transport Protocols

There are actually two different transport protocols for the MSD class:

- Control/Bulk/Interrupt (CBI) transport
- Bulk-Only Transport (BOT)

These two methods are described in two separate stand-alone documents. CBI can be considered obsolete and is being completely replaced by BOT. It was originally targeted at full-speed floppy disk drives. Therefore, the rest of this document will talk about Bulk-Only Transport exclusively.

3.4 Architecture

3.4.1 Interfaces

An MSD device only needs one single interface. It should display the MSD class code (08h) in the *bInterfaceClass* field, the corresponding data transfer protocol code (see Section 3.2 on page 2) in the *bInterfaceSubclass* field, and finally the transport protocol (see Section 3.3 on page 2) code in the *bInterfaceProtocol* field. The protocol code for Bulk-only transport is 50h.

Table 3-2. Transport Protocol Codes

bInterfaceProtocol	Protocol Implementation
00h	Control/Bulk/Interrupt protocol (with command completion interrupt)
01h	Control/Bulk/Interrupt protocol (without command completion interrupt)
50h	Bulk-only transport

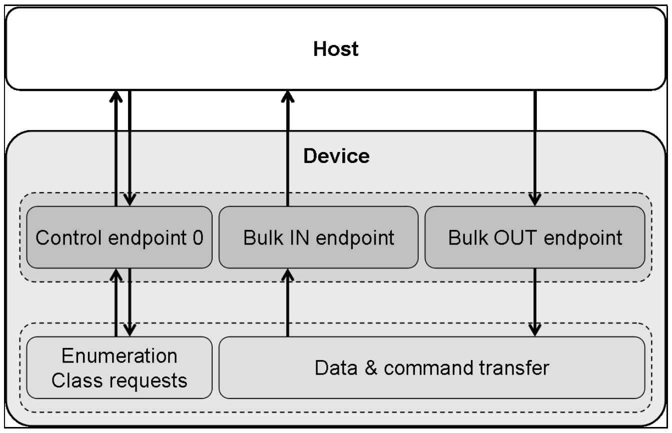
3.4.2 Endpoints

Exactly three endpoints (when using the Bulk-Only Transport protocol) are necessary for MSD devices.

The first one is the Control endpoint 0, and is used for class-specific requests and for clearing Halt conditions on the other two endpoints. Endpoints are halted in response to errors and host bad behavior during data transfers, and the CLEAR_FEATURE request is consequently used to return them to a functional state.

The other two endpoints, which are of type Bulk, are used for transferring commands and data over the bus. There must be one Bulk-IN and one Bulk-OUT endpoint.

Figure 3-1. Mass Storage Device Driver Architecture



3.4.3 Class-Specific Descriptors

There are no class-specific descriptors for an MSD device using the Bulk-only transport protocol.

3.4.4 Class-specific Requests

3.4.4.1 GetMaxLUN

A device can feature one or more Logical Unit (LU). Each of these units will be treated as a separate disk when plugged to a computer. A device can have up to 15 logical units.

The **GET_MAX_LUN** request is issued by the host to determine the maximum Logical Unit Number (LUN) supported by the device. This is not equivalent to the number of LU on the device; since units are numbered starting from 0, a device with 5 LUs should report a value of 4, which will be the index of the fifth unit.

Optionally, a device with only one LUN may STALL this request instead of returning a value of zero.

3.4.4.2 Bulk-Only Mass Storage Reset

This request is used to reset the state of the device and prepare it to receive commands and data. Note that the data toggle bits must not be altered by a RESET command; same for the Halt state of endpoints, i.e., halted endpoints must not be reset to a normal state.

3.4.5 Command/Data/Status

Each MSD transaction is divided into three steps:

- Command stage
- Data stage (optional)
- Status stage

During the command stage, a Command Block Wrapper (CBW) is transmitted by the host to the device. The CBW describes several parameters of the transaction (direction, length, LUN) and carries a variable-length command block. The command block contains data in the format defined by the transfer protocol used by the device (see Section 3.2 on page 2).

Table 3-3. Command Block Wrapper Data Format

Offset	Field Name	Length	Comment
0	dCBWSignature	4 bytes	Signature to identify CBW, must be 43425355h
4	dCBWTag	4 bytes	Tag sent by the host, echoed in the CSW
8	dCBWTransferLength	4 bytes	Length of transfer during the data stage
12	bmCBWFlags	1 byte	**Bits 0-6**: Reserved/obsolete **Bit 7**: Transfer direction (0 = OUT, 1 = IN)
13	bCBWLUN	1 byte	**Bits 0-3**: LUN to which the command is sent **Bits 4-7**: Reserved
14	bCBWCBLength	1 byte	**Bits 0-5**: Length of command block in bytes **Bits 6-7**: Reserved
15	CBWCB	0-16 bytes	Command block to be executed by the device

After the device has received and interpreted the command, an optional data stage may take place if the command requires it. During this step, data is transferred either to or from the device depending on the command, in several IN/OUT transfers.

Once the data stage is complete, the host issues a final IN request on the Bulk-IN endpoint of the device to request the Command Status Wrapper (CSW). The CSW is used to report correct or incorrect execution of the command, as well as indicating the length of remaining data that has not been transferred.

Table 3-4. Command Status Wrapper

Offset	Field Name	Length	Comment
0	dCSWSignature	4 bytes	Signature to identify CSW, must be 53425355h
4	dCSWTag	4 bytes	Copy of previous CBW tag
8	dCSWDataResidue	4 bytes	Difference between expected and real transfer length
12	bCSWStatus	1 byte	Indicates the success or failure of the command

These steps are all performed on the two Bulk endpoints, and do not involve Control endpoint 0 at all.

3.4.6 Reset Recovery

When severe errors occur during command or data transfers (as defined in the *Mass Storage Bulk-only Transport 1.0* document), the device must halt both Bulk endpoints and wait for a **Reset Recovery** procedure. The Reset Recovery sequence goes as follows:

- The host issues a Bulk-Only Mass Storage Reset request
- The host issues two **CLEAR_FEATURE** requests to unhalt each endpoint

A device waiting for a Reset Recovery must not carry out CLEAR_FEATURE requests trying to unhalt either Bulk endpoint until after a Reset request has been received. This enables the host to distinguish between severe and minor errors.

The only major error defined by the Bulk-Only Transport standard is when a CBW is not valid. This means one or more of the following:

- The CBW is not received after a CSW has been sent or a reset.
- The CBW is not exactly 31 bytes in length.
- The *dCBWSignature* field of the CBW is not equal to 43425355h.

3.5 Host Drivers

Almost all operating systems now provide a generic driver for the MSD class. However, the set of supported data transfer protocols (see Section 3.2 on page 2) may vary. For example, Microsoft® Windows® does not currently support the Reduced Block Command set.

4. Mass Storage SCSI Disk

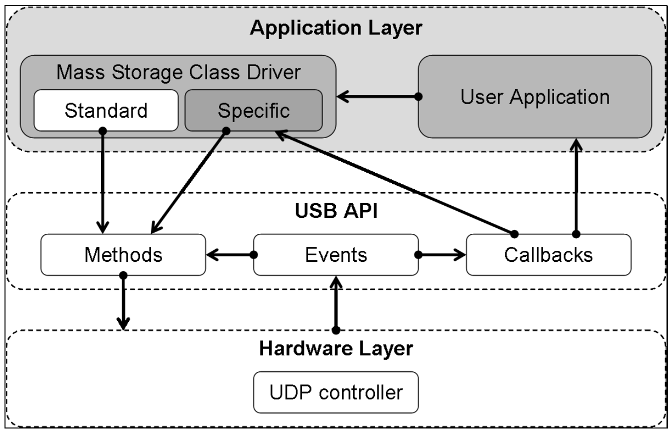
This section describes how to implement a USB disk by using the MSD class with the SCSI transparent command set and the AT91 USB Framework. For more information about the framework, please refer to the *AT91 USB Framework* Application Note; details about the USB and the Mass Storage class can be found in the *USB Specification 2.0* and the *MSC Bulk-Only Transport Specification 1.0* documents, respectively.

The software example provided with this document uses the internal flash of the chip as its storage medium, but has been designed in a modular way to allow easy modification for any medium, e.g. DataFlash®, SD card, external Flash chip.

4.1 Architecture

The AT91 USB Framework offered by ATMEL makes it easy to create USB class drivers. The example software described in the current chapter is based on this framework. Figure 4-1 shows the application architecture:

Figure 4-1. Application Architecture Using the AT91 USB Framework

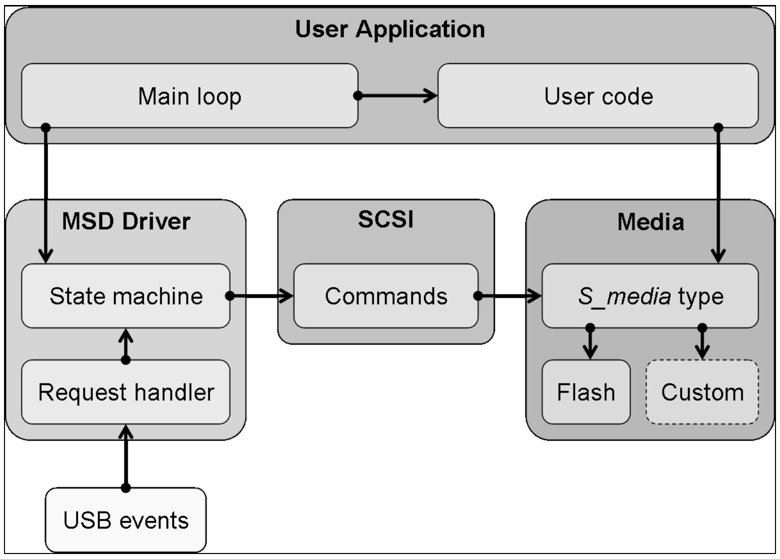


The internal architecture of the Application layers is slightly complex compared to other drivers like HID or CDC. This is so because of three factors:

- The Command/Data/Status flow described in Section 3.4.5 on page 4 requires the use of a **state machine** for non-blocking operation.
- The example software has been designed to be easily extended with support for other media.
- The example software has been designed to support multiple LUNs on one or more media.

Figure 4-2 shows the corresponding architecture:

Figure 4-2. Application Layer Architecture



These elements of the software will be described in the following sections.

4.2 Descriptors

There are no class-specific descriptors for a device using the MSD class with the Bulk-only transport protocol. This section thus only details the values which must be set in the standard descriptors.

4.2.1 Device Descriptor

Since the MSD class code is only specified at the Interface level, the **Device descriptor** is very basic:

Table 4-1. Device Descriptor for a Mass Storage SCSI Disk

Field	Value	Comment
bLength	sizeof(USBDeviceDescriptor)	Size of descriptor (18 bytes)
bDescriptorType	USBGenericDescriptor_DEVICE	Device descriptor type
bcdUSB	USBDeviceDescriptor_USB2_00	Supports USB 2.00
bDeviceClass	MSDeviceDescriptor_CLASS	Class, interface and protocol are specified at Interface level
bDeviceSubClass	MSDeviceDescriptor_SUBCLASS	No subclass at device level
bDeviceProtocol	MSDeviceDescriptor_PROTOCOL	No protocol at device level
bMaxPacketSize0	BOARD_USB_ENDPOINTS_MAXPACKETSIZE(0)	Maximum packet size for Control endpoint 0 (depends on USB controller)
idVendor	MSDDriverDescriptors_VENDORID	Vendor ID for ATMEL (03EBh)
idProduct	MSDDriverDescriptors_PRODUCTID	Product ID for the example (6202h)
bcdDevice	MSDDriverDescriptors_RELEASE	Device version number is 0.01
iManufacturer	1	Index of Manufacturer description string
iProduct	2	Index of Product description string
iSerialNumber	3	Index of Serial Number string
bNumConfigurations	1	This device supports only one configuration

Note that the Vendor ID is a special value attributed by the USB-IF organization. The product ID can be chosen freely by the vendor.

4.2.2 Configuration Descriptor

Since one interface is required by the MSD specification, this must be specified in the **Configuration descriptor**. There is no other value of interest to put here.

Table 4-2. Configuration Descriptor for a Mass Storage SCSI Disk

Field	Value	Comment
bLength	sizeof(USBConfigurationDescriptor)	Size of descriptor (9 bytes)
bDescriptorType	USBGenericDescriptor_CONFIGURATION	Configuration descriptor type
wTotalLength	sizeof(MSDConfigurationDescriptors)	Total length of all descriptors returned, including the configuration descriptor itself
bNumInterfaces	1	This configuration has only one interface
bConfigurationValue	1	This is configuration #1
iConfiguration	0	There is no string describing this configuration
bmAttributes	BOARD_USB_BMATTRIBUTES	Device is self-powered and does not support remote wakeup
bMaxPower	USBConfigurationDescriptor_POWER(100)	Maximum power consumption of device is 100 mA

When the Configuration descriptor is requested by the host (by using the GET_DESCRIPTOR command), the device must also send all the related descriptors, i.e., Interface, Endpoint and Class-Specific descriptors. It is convenient to create a single packed structure to hold all this data, for sending everything in one chunk. In the example software, a *MSDConfigurationDescriptors* structure has been declared to that end.

4.2.3 Interface Descriptor

As previously stated in Section 3.4.1 on page 3, the Interface descriptor must indicate several features:

- **Mass Storage Device** class code (08h) in the *bInterfaceClass* field
- **Data transport protocol** code in the *bInterfaceSubclass* field
- **Bulk-Only Transport** protocol code (50h) in the *bInterfaceProtocol* field

This example uses the SCSI transparent command set (code 06h). This is the most appropriate setting for a Flash device, given that the RBC command set is not supported by Microsoft Windows.

Here is the complete Interface descriptor for the example:

Table 4-3. Interface Descriptor for a Mass Storage SCSI Disk

Field	Value	Comment
bLength	sizeof(USBInterfaceDescriptor)	Size of descriptor (9 bytes)
bDescriptorType	USBGenericDescriptor_INTERFACE	Interface descriptor type
bInterfaceNumber	0	This is interface #0

Table 4-3. Interface Descriptor for a Mass Storage SCSI Disk

Field	Value	Comment
bAlternateSetting	0	No alternate setting for this interface
bNumEndpoints	2	Two endpoints are used by this interface
bInterfaceClass	MSInterfaceDescriptor_CLASS	Mass Storage Device class
bInterfaceSubClass	MSInterfaceDescriptor_SCSI	Data transfer protocol used is SCSI transparent command set
bInterfaceProtocol	MSInterfaceDescriptor_BULKONLY	Transport protocol used is Bulk-Only Transport
iInterface	0	No string describing this interface

4.2.4 Endpoint Descriptors

The two Bulk endpoints needed by the device (see Section 3.4.2 on page 3) must have the corresponding **Endpoint descriptors**. There is no special requirements on these apart from being Bulk-IN and Bulk-OUT:

Table 4-4. Bulk-OUT Endpoint Descriptor for a Mass Storage SCSI Disk

Field	Value	Comment
bLength	sizeof(USBEndpointDescriptor)	Size of descriptor (7 bytes)
bDescriptorType	USBGenericDescriptor_ENDPOINT	Endpoint descriptor type
bEndpointAddress	USBEndpointDescriptor_OUT | MSDDriverDescriptors_BULKOUT	This is an OUT endpoint with address 01h
bmAttributes	USBEndpointDescriptor_BULK	This is a Bulk endpoint
wMaxPacketSize	64	Endpoint max. packet size is 64 bytes
bInterval	0	Must be 0 for full-speed Bulk endpoints

Table 4-5. Bulk-IN Endpoint Descriptor for a Mass Storage SCSI Disk

Field	Value	Comment
bLength	sizeof(USBEndpointDescriptor)	Size of descriptor (7 bytes)
bDescriptorType	USBGenericDescriptor_ENDPOINT	Endpoint descriptor type
bEndpointAddress	USBEndpointDescriptor_IN | MSDDriverDescriptors_BULKIN	This is an IN endpoint with address 02h
bmAttributes	USBEndpointDescriptor_BULK	This is a Bulk endpoint
wMaxPacketSize	64	Endpoint max. packet size is 64 bytes
bInterval	0	Must be 0 for full-speed Bulk endpoints

4.2.5 String Descriptors

Several descriptors can be commented with a String descriptor. The latter are completely optional and do not influence the detection of the device by the operating system. Whether or not to include them is entirely up to the programmer.

There is one exception to this rule when using the MSD class. According to the specification, there must be a **Serial Number** string. It must contains at least 12 characters, and these characters must only be either letters (a-z, A-Z) or numbers (0-9). This cause no problem for the driver in practice, but this is a strict requirement for certification. Also remember that string descriptors use the **Unicode** format.

4.3 Class-Specific Requests

There are two Mass Storage-specific requests that the driver must handle (see Section 3.4.4 on page 4):

- GetMaxLUN
- Bulk-Only Mass Storage Reset

Standard requests can be forwarded to the *STD_RequestHandler* function, with one exception: the **CLEAR_FEATURE** request must be treated separately. This is necessary because when the device is waiting for a Reset Recovery (see Section 3.4.6 on page 5), a request to unhalt Bulk endpoints must not be executed until a Bulk-Only Mass Storage Reset is received.

4.3.1 ClearFeature

As previously stated, the **CLEAR_FEATURE** request must be handled in a particular way, depending on whether or not the device is waiting for a Reset Recovery sequence. If it is, then CLEAR_FEATURE requests to unhalt a Bulk endpoint must be discarded.

In the example software, this behavior is indicated by a boolean field in the driver structure, named *isWaitResetRecovery*. The handler only has to check this field value to decide whether to forward the request to the standard handler or to discard it.

```
//---------------------
case USBGenericRequest_CLEARFEATURE:
//---------------------
  switch (USBFeatureRequest_GetFeatureSelector(request)) {

    //---------------------
    case USBFeatureRequest_ENDPOINTHALT:
    //---------------------

      // Do not clear the endpoint halt status if the device is waiting
      // for a reset recovery sequence
      if (!msdDriver.waitResetRecovery) {

         // Forward the request to the standard handler
        USBDDriver_RequestHandler(&usbdDriver, request);
       }
      USBD_Write(0, 0, 0, 0, 0);
       break;

     //------
     default:
     //------
       // Forward the request to the standard handler
       USBDDriver_RequestHandler(&usbdDriver, request);
   }
```

```
        break;
```

4.3.2 GetMaxLUN

Usually, the first request issued by the host right after the enumeration phase will be a **GET_MAX_LUN** request. It enables it to discover how many different logical units the device has; each of these LUNs can then be queried in turn by the host when needed.

After the request is received by the device, it should return one byte of data indicating the maximum Logical Unit Number (LUN). It is equal to the number of LUNs used by the device minus one. For example, a device with three LUNs returns a **GET_MAX_LUN** value of two.

Sending this byte is done by calling the *USBD_Write* method on Control endpoint 0. Note that the data must be held in a permanent buffer (since the transfer is asynchronous); in the software provided with this application note, a dedicated field is used in the driver structure (*S_bot*) to store this value.

In addition, the *Mass Storage Bulk-Only Transport* specification defines strict requirements on the *wValue*, *wIndex* and *wLength* fields. They must have the following values:

Table 4-6. Valid Values for GetMaxLUN Request Fields

wValue	wIndex	wLength
0000h	Interface number (0)	0001h

In this case, there is only one interface, so the *wIndex* field shall have a value of zero. A request which does not comply to these requirements must be STALLed.

Here is the corresponding code:

```
    //-------------------
    case MSD_GET_MAX_LUN:
    //-------------------
      // Check request parameters
       if ((request->wValue == 0)
         && (request->wIndex == 0)
         && (request->wLength == 1)) {

         USBD_Write(0, &(msdDriver.maxLun), 1, 0, 0);
      }
       else {
        USBD_Stall(0);
      }
      break;
```

4.3.3 Bulk-Only Mass Storage Reset

The host issues **RESET** requests to return the MSD driver of the device to its initial state, i.e., ready to receive a new command. However, this request does not impact the USB controller state; in particular, endpoints must not be reset. This means the data toggle bit must not be altered, and Halted endpoint must not be returned to a normal state. After processing the reset, the device must return a Zero-Length Packet (ZLP) to acknowledge the SETUP transfer.

Like GET_MAX_LUN, this request must be issued with specific parameters. A request which does not have valid values in its field must be acknowledged with a STALL handshake from the device.

Table 4-7. Valid Values for Bulk-Only Mass Storage Reset Request Fields

wValue	wIndex	wLength
0000h	Interface number (0)	0000h

In the example, the handler for this request must return the state machine to its initial state. Also, if the device was waiting for a Reset Recovery, this is not the case anymore.

```
//-----------------------
case MSD_BULK_ONLY_RESET:
//-----------------------
  // Check parameters
   if ((request->wValue == 0)
     && (request->wIndex == 0)
     && (request->wLength == 0)) {

     // Reset the MSD driver
     MSDDriver_Reset();
     USBD_Write(0, 0, 0, 0, 0);
   }
   else {
    USBD_Stall(0);
   }
   break;
```

4.4 State Machine

4.4.1 Rationale

A state machine is necessary for **non-blocking operation** of the driver. As previously stated, there are three steps when processing a command:

- Reception of the CBW
- Processing of the command (with data transfers if required)
- Emission of the CSW

Without a state machine, the program execution would be stopped at each step to wait for transfers completion or command processing. For example, reception of a CBW does not always happen immediately (the host does not have to issue commands regularly) and can block the system for a long time.

Developing an asynchronous design based on a state machine is made easier when using Atmel AT91 USB framework, as most methods are asynchronous. For example, a write operation (using the *USBD_Write* function) returns immediately; a callback function can then be invoked when the transfer actually completes.

4.4.2 States

Apart from the three states corresponding to the command processing flow (CBW, command processing and CSW), two more can be identified. The reception/emission of CBW/CSW will be broken down into two different states: the first state is used to issue the read/write operation, while the second one waits for the transfer to finish. This can be done by monitoring a "transfer complete" flag which is set using a callback function.

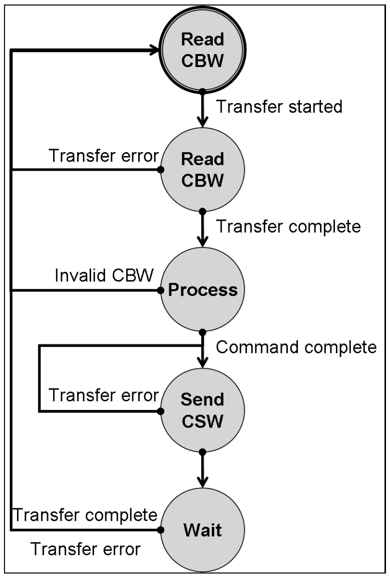
In addition, some commands can be quite complicated to process: they may require several consecutive data transfers mixed with media access. Each command thus has its own second-tier state machine. During execution of a command, the main state machine remains in the "processing" state, and proceeds to the next one (CSW emission) only when the command is complete.

Here is the states list:

- **MSDD_STATE_READ_CBW**: Start of CBW reception (initial state after reset)
- **MSDD_STATE_WAIT_CBW**: Waiting for CBW reception
- **MSDD_STATE_PROCESS_CBW**: Command processing
- **MSDD_STATE_SEND_CSW**: Start of CSW emission
- **MSDD_STATE_WAIT_CSW**: Waiting for CSW emission

A single function, named *BOT_StateMachine*, is provided by the driver. It must be called regularly during the program execution. The following subsections describe the actions that must be performed during each state.

Figure 4-3. MSD Driver State Machine



4.4.2.1 MSDD_STATE_READ_CBW

As said previously, this state is used to start the reception of a new Command Block Wrapper. This is done using the *USBD_Read* method of the USB framework. The result code of the function is checked for any error; the *USBD_STATUS_SUCCESS* code indicates that the transfer has been successfully started.

```
//---------------------
```

```
case MSDD_STATE_READ_CBW:
//----------------------
   // Start the CBW read operation
   transfer->semaphore = 0;
   status = MSDD_Read(cbw,
            MSD_CBW_SIZE,
            (TransferCallback) MSDDriver_Callback,
            (void *) transfer);
   // Check operation result code
   if (status == USBD_STATUS_SUCCESS) {
     // If the command was successful, wait for transfer
     pMsdDriver->state = MSDD_STATE_WAIT_CBW;
   }
   break;
```

A callback function to invoke when the transfer is complete is provided to the *USBD_Read* method, to update a **transfer status** structure. This structure indicates the transfer completion, the returned result code and the number of transferred and remaining bytes.

Table 4-8. Transfer Status Structure (*MSDTransfer*)

Field	Length	Comment
dBytesTransferred	4 bytes	Number of bytes received or emitted by the device
dBytesRemaining	4 bytes	Number of bytes remaining to send or receive
bSemaphore	1 byte	Increased by the callback function to indicate transfer completion.
bStatus	1 byte	Status code returned by the transfer function

The callback function is trivial and thus not listed here.

4.4.2.2 *MSDD_STATE_WAIT_CBW*

The first step here is to monitor the *bSemaphore* field of the transfer status structure (see above); this will enable detection of the transfer end. Please note that this field must be declared as **volatile** in C, or accesses to it might get optimized by the compiler; this can result in endless loops.

If the transfer is complete, then the result code must be checked to see if there was an error. If the operation is successful, the state machine can proceed to command processing. Otherwise, it returns to the *READ_CBW* state.

```
//----------------------
case MSDD_STATE_WAIT_CBW:
//----------------------
   // Check transfer semaphore
   if (transfer->semaphore > 0) {
     // Take semaphore and terminate transfer
     transfer->semaphore--;
     // Check if transfer was successful
     if (transfer->status == USBD_STATUS_SUCCESS) {
      // Process received command
       pMsdDriver->state = MSDD_STATE_PROCESS_CBW;
```

```
            }
            else if (transfer->status == USBD_STATUS_RESET) {
             pMsdDriver->state = MSDD_STATE_READ_CBW;
            }
            else {
             pMsdDriver->state = MSDD_STATE_READ_CBW;
            }
          }
          break;
```

4.4.2.3 *MSDD_STATE_PROCESS_CBW*

Once the CBW has been received, its validity must be checked. According to Section 3.4.6 on page 5, a CBW is not valid if:

- it has not been received right after a CSW was sent or a reset occurred or
- it is not exactly 31 bytes long or
- its signature field is not equal to 43425355h

The state machine prevents the first case from happening, so only the two other cases have to be verified.

The number of bytes transferred during a *USBD_Read* operation is passed as an argument to the callback function, if one has been specified. As stated previously, such a function is used to fill a transfer status structure (Section 4.4.2 on page 13). Therefore, it is trivial to check that the CBW is indeed 31 bytes by verifying that the number of bytes transferred is 31, and that there are no remaining bytes. The following table illustrates the three cases which may happen:

Table 4-9. CBW Length Cases

Number of bytes transferred	Number of bytes remaining	Meaning
dBytesTransferred < 31	dBytesRemaining == 0	CBW is too short
dBytesTransferred == 31	dBytesRemaining > 0	CBW is too long
dBytesTransferred == 31	dBytesRemaining == 0	CBW length is correct

Checking the signature is simply done by comparing the *dCBWSignature* field with the expected value (43425355h).

If the CBW is not valid, then the device must immediately halt both Bulk endpoints, to STALL further traffic from the host. In addition, it should stay in this state until a Reset Recovery is performed by the host. This is done by setting the *isWaitResetRecovery* flag in the driver structure. Finally, the CSW status is set to report an error, and the state machine is returned to *MSDD_STATE_READ_CBW.*

Otherwise, if the CBW is correct, then the command can be processed. Command processing is described in Section 4.6 on page 20. Remember the CBW tag must be copied regardless of the validity of the CBW.

Note that these steps are only necessary for a new command (remember commands are asynchronous and are carried out in several calls, see Section 4.4.2 on page 13), so a check can be performed to avoid useless processing. A value of zero for the internal command state indicates a new command.

```
    //------------------------
```

```
case MSDD_STATE_PROCESS_CBW:
//------------------------
  // Check if this is a new command
   if (commandState->state == 0) {
     // Copy the CBW tag
     csw->dCSWTag = cbw->dCBWTag;
     // Check that the CBW is 31 bytes long
     if ((transfer->transferred != MSD_CBW_SIZE) ||
       (transfer->remaining != 0)) {
      // Wait for a reset recovery
       pMsdDriver->waitResetRecovery = 1;
       // Halt the Bulk-IN and Bulk-OUT pipes
       MSDD_Halt(MSDD_CASE_STALL_OUT | MSDD_CASE_STALL_IN);
       csw->bCSWStatus = MSD_CSW_COMMAND_FAILED;
       pMsdDriver->state = MSDD_STATE_READ_CBW;
     }
     // Check the CBW Signature
     else if (cbw->dCBWSignature != MSD_CBW_SIGNATURE) {
      // Wait for a reset recovery
       pMsdDriver->waitResetRecovery = 1;
       // Halt the Bulk-IN and Bulk-OUT pipes
       MSDD_Halt(MSDD_CASE_STALL_OUT | MSDD_CASE_STALL_IN);
       csw->bCSWStatus = MSD_CSW_COMMAND_FAILED;
       pMsdDriver->state = MSDD_STATE_READ_CBW;
     }
     else {
       // Pre-process command
       MSDD_PreProcessCommand(pMsdDriver);
     }
   }
   // Process command
   if (csw->bCSWStatus == MSDD_STATUS_SUCCESS) {
     if (MSDD_ProcessCommand(pMsdDriver)) {
      // Post-process command if it is finished
       MSDD_PostProcessCommand(pMsdDriver);
       pMsdDriver->state = MSDD_STATE_SEND_CSW;
     }
  }
  break;
```

4.4.2.4 *MSDD_STATE_SEND_CSW*

This state is similar to *MSDD_STATE_READ_CBW*, except that a write operation is performed instead of a read and the CSW is sent, not the CBW. The same callback function is used to fill the transfer structure, which is checked in the next state:

```
//----------------------
case MSDD_STATE_SEND_CSW:
```

```
//----------------------
   // Set signature
   csw->dCSWSignature = MSD_CSW_SIGNATURE;
  // Start the CSW write operation
   status = MSDD_Write(csw,
                  MSD_CSW_SIZE,
                  (TransferCallback) MSDDriver_Callback,
                  (void *) transfer);
  // Check operation result code
   if (status == USBD_STATUS_SUCCESS) {
    // Wait for end of transfer
     pMsdDriver->state = MSDD_STATE_WAIT_CSW;
   }
   break;
```

4.4.2.5 *MSDD_STATE_WAIT_CSW*

Again, this state is very similar to *MSDD_STATE_WAIT_CBW*. The only difference is that the state machine is set to *MSDD_STATE_READ_CBW* regardless of the operation result code:

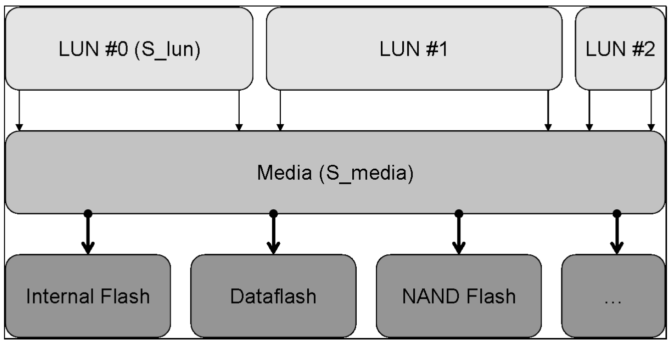
```
//----------------------
case MSDD_STATE_WAIT_CSW:
//----------------------
   // Check transfer semaphore
   if (transfer->semaphore > 0) {
    // Take semaphore and terminate transfer
     transfer->semaphore--;
    // Check if transfer was successful
     // Read new CBW
     pMsdDriver->state = MSDD_STATE_READ_CBW;
   }
   break;
```

4.5 Media

4.5.1 Architecture

Media access is done using a three-level abstraction, as shown in Figure 4-4:

Figure 4-4. Media Architecture



At the bottom level is the specific driver for each media type.

In the middle, a structure named *Media* is used to hide which specific driver a media instance is using. This enables transparent use of any media driver once it has been initialized. The structure has the following format:

Table 4-10. Media Data Format

Field	Comment
fWrite	Pointer to the media Write function
fRead	Pointer to the media Read function
fFlush	Pointer to the media Flush function
fHandler	Pointer to the media Interrupt Handler function
dBaseAddress	Base address of media
dSize	Size of media in bytes
sTransfer	Current transfer status
pInterface	Pointer to the physical interface used
bState	Status of the media (*MED_STATE_READY* or *MED_STATE_BUSY*)

Finally, a LUN abstraction is made over the media structure to allow multiple partitions over one media. This also makes it possible to place the LUN at any address and use any block size. When performing a write or read operation on a LUN, it forwards the operation to the underlying media while translating it to the correct address and length.

4.5.2 Drivers

4.5.2.1 Requirements

As suggested by the *Media* structure above, a media driver must provide several functions for:

- Reading data from the media
- Writing data on the media
- Handling interrupts on the media

The last function may be empty if the media does not require interrupts for asynchronous operation, or if synchronous operation produces an acceptable delay.

In addition, it should also define a function for initializing a *Media* structure with the correct values, as well as perform the necessary step for the media to be useable.

4.5.2.2 Internal Flash Driver

This section describes how the internal Flash driver used in the example software has been implemented. This is done by detailing the content of the five necessary functions, as defined previously**.**

- Initialization Function (FLA_Initialize)

  The first step of the initialization function is to setup the values of the *Media* structure. The 4 function pointers are set to the corresponding Flash methods; Flash parameters (base address, size, physical interface) are the ones defined for the chip in the Lib v3. Since most Atmel ARM® Thumb®-based AT91SAM chips have only one internal Flash bank, the peripheral interface value is hardcoded; it can however be easily passed as an argument of the initialization function. The transfer status structure is initialized with default values.

The next stage is to initialize the media so it is functional. In the case of the internal Flash, there is almost nothing to do; the only required operation is to configure the Mode register of the Embedded Flash Controller with the correct Microsecond Cycle Number. This value can be inferred given the frequency of the master clock:

```
// Configure Flash Mode register
SET(pFlash->FLA_FMR, (BOARD_MCK / 666666) << 16);
```

- Read Function (FLA_Read)

  Reading from the internal Flash is as simple as reading from the SRAM, given the address to access. The function should still verify that the target address and length are valid, and make sure that the media is not busy (i.e., another operation is already in progress). If everything is correct, then the specified number of bytes can be read from the Flash into the provided buffer. If a callback has been specified, it is invoked when the transfer finishes.

- Write Function (FLA_Write)

  Performing a write operation is a more complex than reading. First of all, since the internal Flash on most AT91SAM chips is single-plane, it cannot be read and written simultaneously. This means the software must actually run from the SRAM while writing the Flash. The simplest method is to copy the entire software to SRAM, set the remap bit in the memory controller and run from there.

  The actual Write function should first check the parameters. Since the Embedded Flash Controller only supports memory writes of exactly one double-word (4 bytes), any access with an address not aligned on a 4-bytes boundary or a length not multiple of 4 must be rejected. Of course, the data must not be bigger than the size of the media and the media must be in the ready state.

  If input parameters are correct, actual writing can begin. The internal Flash is divided into chunks of bytes called pages, which are generally between 64 and 256 bytes in length. A Write operation must be done one page at a time, and the page must be cleared before writing. This can lead to problems if only part of the page is altered; in this case, the remaining data can simply be copied before writing. A second function is used to write one page of data at a time; the Write method starts the whole transfer by writing the first page and initializing the transfer status.

  Each page write takes some time to complete, so the interrupt on the Flash Ready status bit is used to avoid blocking the system while waiting for the operation.

- Interrupt Handler Function (FLA_Handler)

  This interrupt handler is invoked when a Flash page has been written. It should check the transfer status to see if there is more data to write; if there is, then it calls the page write function again with new parameters. Otherwise, the transfer is finished and the callback function invoked if one has been defined. In any case, the Flash Ready interrupt must be disabled to avoid continuous interruption, and re-enabled when the page write operation starts.

4.5.2.3 *Virtual RAM Disk Driver*

This section describes how the Virtual RAM Disk driver of SDRAM used in the example software has been implemented. This is done by detailing the content of the three necessary functions, as defined previously**.**

- Initialization Function (FLA_Initialize)

  Usually the SDRAM is initialized when the binary starts to run. This function will check if the SDRAM is working, if not, it just invokes the standard initialize function *BOARD_ConfigureSdram* to initialize it. You can also refer to the AT91SAM datasheets for SDRAM initialize sequence.

- Read Function (MEDSdram_Read)

  Just reading from SDRAM. The media state and address is checked. If everything is OK the address is translated to a SDRAM address by adding to the base address of the ram disk and number of bytes specified is copied from that address to the assigned buffer.

- Write Function (MEDSdram_Write)

  Just writing to the SDRAM. If the media state and address check is passed, the actual address is calculated by the base address of the ram disk, and the number of bytes specified is copied from the input buffer to there.

4.6 SCSI Commands

The example software described in this application note uses SCSI commands with the MSD class, since this is the most appropriate setting for a Flash device. This section details how SCSI commands are processed.

4.6.1 Documents

There are several documents covering SCSI commands. In this application note, the reference document used is *SCSI Block Commands - 3 (SBC-3).* However, it makes many references to another SCSI document, *SCSI Primary Commands - 4 (SPC-4).* Both are needed for full details on required commands.

4.6.2 Endianness

SCSI commands use the big-endian format for storing word- and double word-sized data. This means the Most Significant Bit (MSB) is stored at the lowest address, and the Least Significant Bit (LSB) at the highest one.

On ARM Thumb microcontrollers, the endianness of the core is selectable. However, the little-endian mode is most often used. Therefore, SCSI command data must be converted before being usable. This is done by declaring word- and dword-sized fields as byte arrays, and then using a macro for loading or storing data. Several of them are available in the provided software:

- Load
  - WORDB: Converts a big-endian word value to little-endian
  - DWORDB: Converts a big-endian double-word value to little-endian
- Store
  - STORE_WORDB: Stores a little-endian word value in big-endian format
  - STORE_DWORDB: Stores a little-endian double-word value in big-endian format

4.6.3 Sense Data

When an error happens during the execution of a command, it is recorded by the device. The host may then issue a Request Sense command to retrieve **Sense Data**, i.e., information about previous errors.

While the sense data structure has many fields, only three are really important. The first one is the *Sense Key*. It indicates the result of the last command performed: success, media not ready, hardware error, etc. Two other fields can then be specified to give a more accurate description of the problem. They are named *Additional Sense Code* and *Additional Sense Code Qualifier*.

In the example application, each LUN has its own sense data. It is updated during command execution if there is any error.

4.6.4 Commands

The SBC-3 specification gives a list of mandatory and optional commands that are relevant for a block device (like a Flash drive). In practice, only a subset of the mandatory commands is effectively used by operating systems; conversely, several commands which are supposed to be optional are required. The software provided with this application note implements the following list of commands:

- SBC-3
  - Prevent/Allow Medium Removal
  - Read (10)
  - Read Capacity (10)
  - Verify (10)
  - Write (10)
- SPC-4
  - Inquiry
  - Mode Sense (6)
  - Request Sense
  - Test Unit Ready

These commands are described in the following paragraphs.

4.6.4.1 Internal State Machine

As previously stated, most commands have an internal state machine to prevent blocking the whole system during a data transfer (on the USB or when accessing a media). A result code is used to indicate that the corresponding function must be called again for the command to complete (*BOT_STATUS_INCOMPLETE*).

A command state structure is used by the driver to record several parameters during command processing:

Table 4-11. MSDCommandState Data Format

Field	Comment
sTransfer	Current transfer status
sCbw	CBW of the current command
sCsw	CSW to be returned when the current command is complete

Table 4-11. MSDCommandState Data Format

Field	Comment
bState	Current internal state of executing command
bPostprocess	Indicates post-processing actions to take (see Section 4.6.5 on page 31)
dLength	Remaining number of bytes that the command must process

Note that the *bState* field must be initialized when the command is first called. A value of 0 means that no command is currently being executed.

4.6.4.2 *Inquiry*

The host usually issues an **Inquiry** command right after the enumeration phase to get more information about the device. In particular, the Peripheral Device Type (PDT) and Version fields will determine which commands will be issued by the host.

Table 4-12. Inquiry Command Block

Offset	Field	Length	Comment
0	Operation Code	1 byte	Inquiry operation code (12h)
1	EPVD	1 bit	Enable Product Vital Data bit. When set, the device should return vital product data instead of standard values.
1.1	Obsolete	1 bit	Obsolete bit
1.2	Reserved	6 bits	Reserved bits
2	Page Code	1 byte	Page code of vital product data to return (when EPVD set)
3	Allocation Length	2 bytes	Maximum number of bytes to send
5	Control	1 byte	Unused, should be 00h

In practice, there is no need to check any of the fields except for *Allocation Length*, since this is the maximum transfer length. The device shall simply return the standard Inquiry data; please refer to the SPC-4 specification for the exact format to use, and to the software example for basic values.

The function for handling this command has a simple internal state machine, with only two states. It must be initialized to the first state (*SBC_STATE_WRITE*), and the value of the *Additional Length* field set correctly (depending on the *dLength* field of the command state structure). Note that there should be no data transfer when *dLength* is zero; the function can return immediately in this case. Otherwise, a zero-length packet will be emitted, which is considered an error. Here is the initialization code:

```
// Check if required length is 0
if (commandState->length == 0) {

  // Nothing to do
   result = MSDD_STATUS_SUCCESS;
}
// Initialize command state if needed
else if (commandState->state == 0) {

   commandState->state = SBC_STATE_WRITE;
```

```
      // Change additional length field of inquiry data
      lun->inquiryData->bAdditionalLength
        = (unsigned char) (commandState->length - 5);
   }
```

The function can then check the current command state. The first state, *SBC_STATE_WRITE*, is used to start the transfer using the *USBD_Write* method. Once again the *BOT_Callback* function is used with the driver transfer structure. See "MSDD_STATE_READ_CBW" on page 13.

```
   //-------------------
   case SBC_STATE_WRITE:
   //-------------------
   // Start write operation
      status = MSDD_Write((void *) lun->inquiryData,
                    commandState->length,
                    (TransferCallback) MSDDriver_Callback,
                    (void *) transfer);

      // Check operation result code
      if (status != USBD_STATUS_SUCCESS) {

       result = MSDD_STATUS_ERROR;
      }
      else {

        // Proceed to next state
        commandState->state = SBC_STATE_WAIT_WRITE;
     }
      break;
```

The second state (*SBC_STATE_WAIT_WRITE*) is used to check for the transfer completion and terminate the command execution. Since the dLength field of the command state structure is used to calculate the data residue, it must be decremented by the amount of data sent.

```
   //------------------------
   case SBC_STATE_WAIT_WRITE:
   //------------------------
   // Check the semaphore value
      if (transfer->semaphore > 0) {

      // Take semaphore and terminate command
      transfer->semaphore--;

      if (transfer->status != USBD_STATUS_SUCCESS) {

        result = MSDD_STATUS_ERROR;
      }
      else {
        result = MSDD_STATUS_SUCCESS;
      }

      // Update length field
      commandState->length -= transfer->transferred;
      }
      break;
```

4.6.4.3 *Read (10)*

As its name suggests, the **Read (10)** command is issued by the host to read data from one LUN. It can also retrieve protection information about the logical unit; read/write protection is not supported in the provided software example however. Here is the associated data format for the command:

Table 4-13. Read (10) Command Block

Offset	Field	Length	Comment
0	Operation Code	1 byte	Read (10) operation code (28h)
1	Obsolete	1 bit	Obsolete bit
1.1	FUA_NV	1 bit	Cache control bit
1.2	Reserved	1 bit	Reserved bit
1.3	FUA	1 bit	Cache control bit
1.4	DPO	1 bit	Cache control bit
1.5	RDProtect	3 bits	Protection information checking
2	Logical Block Address	4 bytes	Address of the first block to read
6	Group Number	5 bytes	Information group
6.5	Reserved	3 bits	Reserved bits
7	Transfer Length	2 bytes	Number of blocks to read
9	Control	1 byte	Unused, should be 00h

Several bits of the command can be set to specify where the data should be read from, i.e., directly from the media or from volatile/non-volatile cache. Since caching is out of the scope of this application note, these bits are not used in the example software.

The internal state machine for this command is made up of five different states. First, the requested data must be read from the media. This is done using two stages, one for starting the asynchronous read operation and one for waiting for the operation to complete. If an error occurs when trying to start the transfer, the operation is aborted and the LUN sense data updated. Note that only one block of data is read and sent at a time, so multiple iterations of the state machine may be needed to fulfill the request.

```
    //------------------
    case SBC_STATE_READ:
    //------------------
        // Read one block of data from the media
        status = LUN_Read(lun,
                      DWORDB(command->pLogicalBlockAddress),
                      lun->readWriteBuffer, 1,
                      (TransferCallback) MSDDriver_Callback, (void *) transfer);
        // Check operation result code
        if (status != LUN_STATUS_SUCCESS) {
          SBC_UpdateSenseData(&(lun->requestSenseData),
                              SBC_SENSE_KEY_NOT_READY,
                              SBC_ASC_LOGICAL_UNIT_NOT_READY, 0);
          result = MSDD_STATUS_ERROR;
        }
        else {
          // Move to next command state
          commandState->state = SBC_STATE_WAIT_READ;
        }
        break;

    //------------------------
    case SBC_STATE_WAIT_READ:
    //------------------------
        // Check semaphore value
        if (transfer->semaphore > 0) {
         // Take semaphore and move to next state
          transfer->semaphore--;
          commandState->state = SBC_STATE_WRITE;
        }
        break;
```

After the data has been read correctly, it can be sent to the host through the Bulk-IN endpoint. Again, this must be done using two states, the first one for starting the transfer and the other one for waiting for its completion. Result of the read operation if also verified prior to beginning the write transfer, adjusting the LUN sense data if an error occured.

```
   //--------------------
   case SBC_STATE_WRITE:
   //--------------------
      // Check the operation result code
      if (transfer->status != USBD_STATUS_SUCCESS) {
        SBC_UpdateSenseData(&(lun->requestSenseData),
                          SBC_SENSE_KEY_RECOVERED_ERROR,
                          SBC_ASC_LOGICAL_BLOCK_ADDRESS_OUT_OF_RANGE, 0);
        result = MSDD_STATUS_ERROR;
      }
      else {
        // Send the block to the host
        status = MSDD_Write((void*)lun->readWriteBuffer,
                          lun->blockSize,
                          (TransferCallback) MSDDriver_Callback, (void *)
   transfer);
       // Check operation result code
```

```
    if (status != USBD_STATUS_SUCCESS) {
      SBC_UpdateSenseData(&(lun->requestSenseData),
                       SBC_SENSE_KEY_HARDWARE_ERROR,
                       0, 0);
      result = MSDD_STATUS_ERROR;
    }
    else {
     // Move to next command state
      commandState->state = SBC_STATE_WAIT_WRITE;
    }
   }
   break;

//-----------------------
case SBC_STATE_WAIT_WRITE:
//-----------------------
   // Check semaphore value
   if (transfer->semaphore > 0) {
   // Take semaphore and move to next state
    transfer->semaphore--;
    commandState->state = SBC_STATE_NEXT_BLOCK;
   }
   break;
```

Finally, the last state checks whether the write operation was successful, and whether or not there are more blocks to send to the host. In any case, the *dLength* field of the command status is decremented by one, to account for the block which has just been sent. The state machine is either reset to the first state to send more blocks, or returns a success/error code.

```
//-----------------------------
case SBC_STATE_NEXT_BLOCK:
//-----------------------------
   // Check operation result code
   if (transfer->status != USBD_STATUS_SUCCESS) {
   SBC_UpdateSenseData(&(lun->requestSenseData),
               SBC_SENSE_KEY_HARDWARE_ERROR,
               0, 0);
    result = MSDD_STATUS_ERROR;
   }
   else {
   // Update transfer length and block address
    STORE_DWORDB(DWORDB(command->pLogicalBlockAddress) + 1,
                command->pLogicalBlockAddress);
    commandState->length--;
    // Check if transfer is finished
    if (commandState->length == 0) {
     result = MSDD_STATUS_SUCCESS;
    }
```

```
            else {
             commandState->state = SBC_STATE_READ;
            }
          }
          break;
```

4.6.4.4 *Read Capacity (10)*

The Read Capacity (10) command enables the host to retrieve the number of block present on a media, as well as their size.

Table 4-14. Read Capacity (10) Command Block

Offset	Field	Length	Comment
0	Operation Code	1 byte	Read capacity (10) operation code (25h)
1	Obsolete	1 bit	Obsolete bit
1.1	Reserved	7 bits	Reserved bits
2	Logical Block Address	4 bytes	Address of the block to examinate if PMI is set
6	Reserved	2 bytes	Reserved bytes
8	PMI	1 bit	Partial Medium Indicator bit
8.1	Reserved	7 bits	Reserved bits
9	Control	1 byte	Unused, should be 00h

After receiving this command, the device must respond by sending back the previously mentioned data using the following format:

Table 4-15. Read Capacity (10) Parameter Data Format

Offset	Field	Length	Comment
0	Returned Logical Block Address	4 bytes	Address of last logical block on LUN
4	Logical Block Length	4 bytes	Block size of last logical block on LUN

The function for handling this command uses a two-states internal machine. It is very similar to the Inquiry command, so it is not detailed here. See "Inquiry" on page 22. The PMI bit is not taken into account in the example software; refer to the SBC-3 specification for more information about its functionality.

4.6.4.5 *Request Sense*

After an error has occured during the execution of a command block, the host will issue a **Request Sense** to get more information about the problem. The command has the following format:

Table 4-16. Request Sense Command Block

Offset	Field	Length	Comment
0	Operation Code	1 byte	Request sense operation code (03h)
1	Desc	1 bit	Type of data returned

Table 4-16. Request Sense Command Block

Offset	Field	Length	Comment
1.1	Reserved	23 bits	Reserved bits
4	Allocation Length	1 byte	Maximum length of returned data
5	Control	1 byte	Unused, should be 00h

The *Desc* bit is used to specify whether the sense data must be in fixed format or in descriptor format. Since the former is easier to deal with, it is the only format supported by the example driver. If a Request Sense command is received with the *Desc* bit set (descriptor-format data requested), it is treated as an illegal request.

Table 4-17. Fixed Format Request Sense Data

Offset	Field	Length	Comment
0	Response Code	7 bits	Should be 70h or 71h
0.7	Valid	1 bit	Indicates if the value contained in the Information field is defined by SPC-4
1	Obsolete	1 byte	Obsolete byte
2	Sense Key	4 bits	Information about error
2.4	Reserved	1 bit	Reserved bit
2.5	ILI	1 bit	Indicates if requested logical block length does not match LUN block length
2.6	EOM	1 bit	Indicates if end-of-medium has been reached
2.7	Filemark	1 bit	Indicates that a filemark has been encountered
3	Information	4 bytes	Device type- and command-specific data
7	Additional Sense Length	1 byte	Size of returned data - 8
8	Command-Specific Information	4 bytes	Command-specific information
12	Additional Sense Code	1 byte	Information about error
13	Additional Sense Code Qualifier	1 byte	Information about error
14	Field Replaceable Unit Code	1 byte	Information about a component failure
15	Sense Key Specific	15 bits	Sense key specific data
17.7	SKSV	1 bit	Indicates if Sense Key Specific field is valid
18	Additional Sense Bytes	Any	Vendor-specific additional data

Most of the fields contained in the previous structure are not strictly necessary. In the example software, only the *Response Code*, *Sense Key, Additional Sense Length*, *Additional Sense Code* and *Additional Sense Code Qualifier* fields are set to non-zero values.

Handling this command is done in a similar way as the Inquiry command. See "Inquiry" on page 22.

4.6.4.6 *Test Unit Ready*

This command provides a way to check if a logical unit is ready. The device should perform a self-test and report to the USB host either with a "passed" status code (indicating that the unit is ready), or with an error. In this case, the request sense data shall indicate the current state of the LUN.

Table 4-18. Test Unit Ready Command Block

Offset	Field	Length	Comment
0	Operation Code	1 byte	Test unit ready operation code (00h)
1	Reserved	4 bytes	Reserved bytes
5	Control	1 byte	Unused, should be 00h

This command does not require an internal state machine, as it does not perform any data transfer. It only checks the status of the specified LUN, adjusting its request sense data if it is not ready.

4.6.4.7 Write (10)

The **Write (10)** command enables the host to write one or several blocks of data on a particular LUN of the device. Like *Read (10)*, this command may also provide protection information for each block but this is not supported in the software example.

Table 4-19. Write (10) Command Block

Offset	Field	Length	Comment
0	Operation Code	1 byte	Write (10) operation code (2Ah)
1	Obsolete	1 bit	Obsolete bit
1.1	FUA_NV	1 bit	Cache control bit
1.2	Reserved	1 bit	Reserved bit
1.3	FUA	1 bit	Cache control bit
1.4	DPO	1 bit	Cache control bit
1.5	WRProtect	3 bits	Protection information
2	Logical Block Address	4 bytes	Index of first block to write
6	Group number	5 bits	Grouping function
6.5	Reserved	3 bits	Reserved bits
7	Transfer Length	2 bytes	Number of blocks to write
9	Control	1 byte	Unused, should be 00h

This command is handled just like Read (10) (see "Read (10)" on page 24) except the read operation is done on the USB and the write operation on the LUN. They are performed in the same order however (read then write).

4.6.4.8 *Prevent/Allow Medium Removal*

This request notifies the device that it should prevent or allow removal of the media associated with the specified LUN. It has the following data format:

Table 4-20. Prevent/Allow Medium Removal Command Block

Offset	Field	Length	Comment
0	Operation Code	1 byte	Prevent/allow medium removal operation code (1Eh)
1	Reserved	3 bytes	Reserved bytes
4	Prevent	2 bits	Indicates if medium removal is allowed or prevented
4.2	Reserved	6 bits	Reserved bits
5	Control	1 byte	Unused, should be 00h

This command is not relevant for an internal Flash disk, as such a media cannot be removed. As such, the corresponding handler function simply returns with a *BOT_STATUS_SUCCESS* result code.

4.6.4.9 *Mode Sense (6)*

The **Mode Sense (6)** command tells the device to return its parameters for one or more *mode pages*. Each mode page covers a functional area of the device, e.g., exception handling, power management, and so on. The host can request either a particular page (or subpage), or all of them at once.

Table 4-21. Mode Sense (6) Command Block

Offset	Field	Length	Comment
0	Operation Code	1 byte	Mode sense (6) operation code (1Ah)
1	Reserved	3 bits	Reserved bits
1.3	DBD	1 bit	Disable block descriptors bit
1.4	Reserved	4 bits	Reserved bits
2	Page Code	6 bits	Requested page code
2.6	PC	2 bits	Values to return (Current, changeable, default, etc.)
3	Subpage Code	1 byte	Requested subpage code
4	Allocation Length	1 byte	Maximum number of bytes to return
5	Control	1 byte	Unused, should be 00h

After receiving this command, the device must answer with the corresponding mode pages, prefixed with a particular header. Optionally, information about blocks on the media can be returned by the mean of block descriptors. The mode parameter header (6) has the following format:

Table 4-22. Mode Parameter Header (6) Data Format

Offset	Field	Length	Comment
0	Mode Data Length	1 byte	Number of bytes returned by the device minus 1
1	Medium Type	1 byte	Should be 00h for block devices
2	Reserved	4 bits	Reserved bits
2.4	DPOFUA	1 bit	Indicates if DPO and FUA bits are supported

Table 4-22. Mode Parameter Header (6) Data Format

Offset	Field	Length	Comment
2.5	Reserved	2 bits	Reserved bits
2.7	WP	1 bit	Indicates if media is write protected
3	Block Descriptors Length	1 byte	Length of all block descriptors returned

In practice, it is not necessary to return any mode page or block descriptor. This is because this request is primarily issued by hosts to get the write-protection bit state. As such, it is possible to only return the mode parameter header information when the request is issued with the “return all” page code, which is what is done in the example software.

The function for handling the *Mode Sense (6)* command is similar as the one for the *Inquiry* command. Refer to “Inquiry” on page 22 for more information.

4.6.4.10 Verify (10)

The **Verify (10)** command instructs the device to verify the data written on one or more blocks of a LUN. This is done either by comparing it to data sent by the host, or by simply looking for corrupted blocks. Protection information can also be checked via this command. Finally, if the media uses a caching strategy, it should synchronize the data in cache and on the media.

Table 4-23. Verify (10) Command Block

Offset	Field	Length	Comment
0	Operation Code	1 byte	Verify (10) operation code (2Fh)
1	Obsolete	1 bti	Obsolete bit
1.1	BytChk	1 bit	Verification mode
1.2	Reserved	2 bits	Reserved bits
1.4	DPO	1 bit	Cache control bit
1.5	VRProtect	3 bits	Protection information
2	Logical Block Address	4 bytes	Index of first block to verify
6	Group Number	5 bits	Grouping function
6.5	Reserved	2 bits	Reserved bits
6.7	Restricted for MMC-4	1 bit	Restricted bit for MMC-4
7	Verification Length	2 bytes	Number of blocks to verify
9	Control	1 byte	Unused, should be 00h

This command is not implemented in the example provided with this document. This is not necessary because the *BytChk* flag is almost never set, meaning the check is done without comparing data sent by the host; since the internal Flash on AT91SAM chips cannot have corrupted blocks, the Verify operation always succeed.

4.6.5 Command Processing

4.6.5.1 Flow

Command processing is actually divided into three phases in the example software:

- Pre-processing

- Processing
- Post-processing

Since the command length and direction is replicated in both the CBW and the encapsulted command block, there may be differences between these values. The *Mass Storage Bulk-Only Transport 1.0* document list the 13 possible cases and the actions to perform when encountering them.

During the pre-processing stage, the program checks the command length and direction values contained in the CBW and the command block for any difference. This makes it possible to identify which case the transaction falls in, as well as set the correct length/direction.

The command is then processed normally. The number of bytes to process, if any, is updated each time the command is called. This enable the program to get the number of remaining bytes, in order to set the *dCSWDataResidue* field of the CSW.

Once the command is complete, the post-processing stage performs the necessary actions depending on the case which was identified during pre-processing.

4.6.5.2 The Thirteen Cases

There are basically three actions that should be performed depending on the case:

- STALL the Bulk-IN endpoint
- STALL the Bulk-OUT endpoint
- Report a Phase Error in the CSW

The table below lists all cases along with the actions which must be taken after the command, including the correct length/direction of the transfer. The following notation is used to characterize host and device expectations:

Table 4-24. Data Transfer Characterization

Notation	Meaning	Notation	Meaning
Hn	Host expects no data transfer	Dn	Device expects no data transfer
Hi	Host expects to **receive** data	Di	Device expects to **send** data
Ho	Host expects to **send** data	Do	Device expects to **receive** data
Lh	Length of data expected by the host	Ld	Length of data expected by the device
Hx = Dx	Host and device agree on transfer length and direction (x is either n, i or o)		
Hx > Dx	Host and device agree on transfer direction, host expects a larger transfer than device		
Hx < Dx	Host and device agree on transfer direction, device expects a larger transfer than host		
Hx <> Dy	Host and device disagree on transfer direction		

Table 4-25. The Thirteen Cases

#	Case	Length	Residue	Direction	STALL IN?	STALL OUT?	Phase Error?
1	Hn = Dn	0	0	Irrelevant			
2	Hn < Di	0	Ld - Lh	Irrelevant			X
3	Hn < Do	0	Ld - Lh	Irrelevant			X
4	Hi > Dn	0	Lh	Irrelevant	X		
5	Hi > Di	Ld	Lh - Ld	In	X		

Table 4-25. The Thirteen Cases

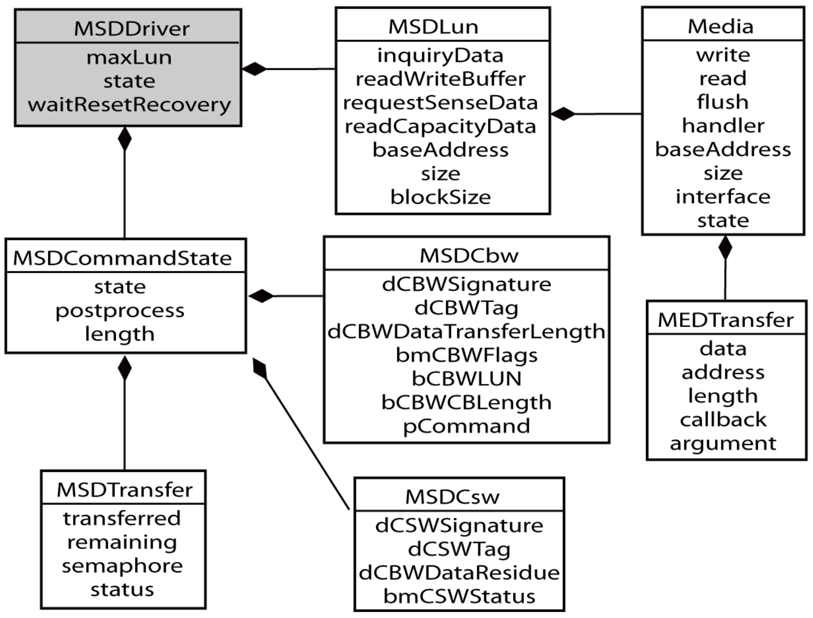
#	Case	Length	Residue	Direction	STALL IN?	STALL OUT?	Phase Error?
6	Hi = Di	Ld	0	In			
7	Hi < Di	Lh	Ld - Lh	In			X
8	Hi <> Do	0	0	Irrelevant	X		X
9	Ho > Dn	0	Lh	Irrelevant		X	
10	Ho <> Di	0	0	Irrelevant		X	X
11	Ho > Do	Ld	Lh - Ld	Out		X	
12	Ho = Do	Ld	0	Out			
13	Ho < Do	Lh	Lh - Ld	Out			X

4.7 Main Application

After the MSD driver and the media have been initialized using the corresponding functions, the only requirement for the main application is to regularly call the state machine function. This is necessary for processing received commands in a fully asynchronous way.

The application is otherwise free of doing any other task; for example, it could implement a file-system and a serial port interface to be accessed with a standard terminal. An MP3 player could also continue playing a song while its memory is accessed like an external hard disk.

Figure 4-5. Driver Class Diagram



4.8 Example Software Usage

4.8.1 File Architecture

The software example provided along with this application note is divided into several groups:

- **at91lib\usb\common\massstorage**: Folder with generic Mass Storage definitions.
- *MSDeviceDescriptor.h*: Definitions for Mass Storage Device Descriptor.

- *MSInterfaceDescriptor.h*: Definitions for Mass Storage Interface Descriptor.
- **at91lib\usb\device\massstorage**: Folder with definitions for Mass Storage Device Driver.
- *MSD.h*: header file with general definitions for MSD driver.
- *MSDDriver.h*: header file with definitions for the MSD Bulk-Only Transport driver.
- *MSDDriver.c*: source file for the MSD driver.
- *MSDDriverDescriptors.h*: header file with definitions for MSD driver descriptors.
- *MSDDriverDescriptots.c*: source code for the MSD driver descriptors.
- *MSDDStateMachine.h*: header file with definitions for MSD state machine to process.
- *MSDDstateMachine.c*: source file for MSD state machine to process.
- *MSDLun.h*: header file wih definitions for MSD LUN.
- *MSDLun.c*: source code for MSD LUN operations.
- *SBC.h*: header file generic SCSI definitions.
- *SBCMethods.h*: header file for SBC function definitions.
- *SBCMethods.c*: source file for SBC function implementation.
- **at91lib\memories**: folder with media interface definition and storage drivers.
- *Media.h*: header file with media definitions.
- *Media.c*: source file for media interrupt handling.
- *MEDFlash.h*: header file for the internal flash media driver.
- *MEDFlash.c*: source file for the internal flash media driver.
- *MEDSdram.h*: header file for the SDRAM virtual ram disk driver.
- *MEDSdram.c*: source file for the SDRAM virtual ram disk driver.
- **usb-device-massstorage-project**: Folder for application main functions.
- *main.c*: source file for MSD example application.

4.8.2 Compilation

The software is provided with a **Makefile** to build it. It requires the *GNU make* utility, which is available on www.GNU.org. Please refer to the *AT91 USB Device Framework* application note for more information on general options and parameters of the Makefile.

To build the USB to serial converter example just run “make” in directory **usb-device-massstorage-project**, and two parameters may be assigned in command line, the CHIP= and BOARD=, the default value of these parameters are “at91sam7s256” and “at91sam7s-ek”:

```
make CHIP=at91sam7se512 BOARD=at91sam7se-ek
```

In this case, the resulting binary will be named usb-device-massstorage-project-at91sam7se-ek-at91sam7se512-flash.*bin* and will be located in the usb-device-massstorage-project/*bin* directory.

4.9 Using a Generic Host Driver

Most operating systems have a generic driver for the Mass Storage class. They may however have different requirements in term of SCSI commands that must be supported by the device. This example driver should be fully functional under Microsoft Windows and Linux®.

5. Revision History

Table 5-1.

Document Ref.	Date	Comments	Change Request Ref.
6283A	08-Jan-07	First issue.	
6238B	02-Jul-09	Updated source file	

Headquarters

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

International

Atmel Asia
Unit 1-5 & 16, 19/F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
Hong Kong
Tel: (852) 2245-6100
Fax: (852) 2722-1369

Atmel Europe
Le Krebs
8, Rue Jean-Pierre Timbaud
BP 309
78054 Saint-Quentin-en-
Yvelines Cedex
France
Tel: (33) 1-30-60-70-00
Fax: (33) 1-30-60-71-11

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Product Contact

Web Site
www.atmel.com
www.atmel.com/AT91SAM

Technical Support
AT91SAM Support
Atmel techincal support

Sales Contacts
www.atmel.com/contacts/

Literature Requests
www.atmel.com/literature