USB Mass Storage Device Implementation

References

- Universal Serial Bus Specification, revision 2.0
- Universal Serial Bus Class Definition for Communication Devices, version 1.1
- USB Mass Storage Overview, revision 1.2
- USB Mass Storage Bulk Only, revision 1.0

Abbreviations

- USB: Universal Serial Bus
- VID: Vendor Identifier
- PID: Product Identifier
- LUN: Logical Unit Number
- USB-IF: USB Information Forum
- SCSI: Small Computer System Interface

Supported Controllers

- AT89C5131A

USB Microcontrollers

Application Note

Rev. 7516A–USB–05/05

Introduction

Scope

The floppy disk is over! Too slow, too fragile, not enough capacity...

USB is now on every PC under Windows or Linux, on every Mac. And you can very easily find USB Mass Storage keys that are automatically recognized by the Operating System, with high capacities and excellent speed performances.

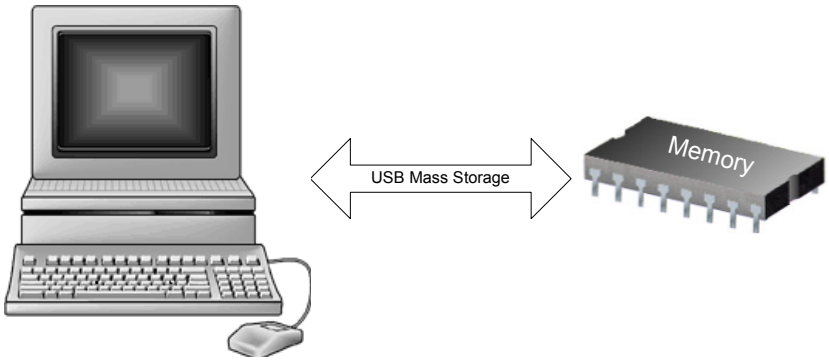
The USB Mass Storage device replaces Floppy, and can be used in many other applications:

- USB mouse for laptop with Mass Storage integrated
- External Hard Drive, data storage
- Access to internal memory of a mobile phone or MP3 player
- Firmware Upgrade of a system
- Small capacity memory to store private information or preferences (Smart Card Reader, Control Access, Electronic Wallet...)
- etc.

Overview

The aim of this document is to describe how to implement the USB Mass Storage class (with an example on the AT89C5131A product) and the firmware functions delivered by Atmel.

Figure 1. Application Overview



What is USB Mass Storage

Inside the computer, many Mass Storage peripherals (the hard drive for example) can be accessed using the SCSI command set.

Note: SCSI = “Small Computer System Interface”

The USB Mass Storage class defines the USB wrapper of the SCSI commands. USB Mass Storage is no more than the SCSI command set for the USB.

In order to access several memories at the same time, for a multiple-memory card reader for example, the Host selects the LUN (Logical Unit Number) whose SCSI command is addressed.

Operating Systems and USB Mass Storage

Currently, most Operating Systems support USB and have built-in drivers to support the USB Mass Storage:

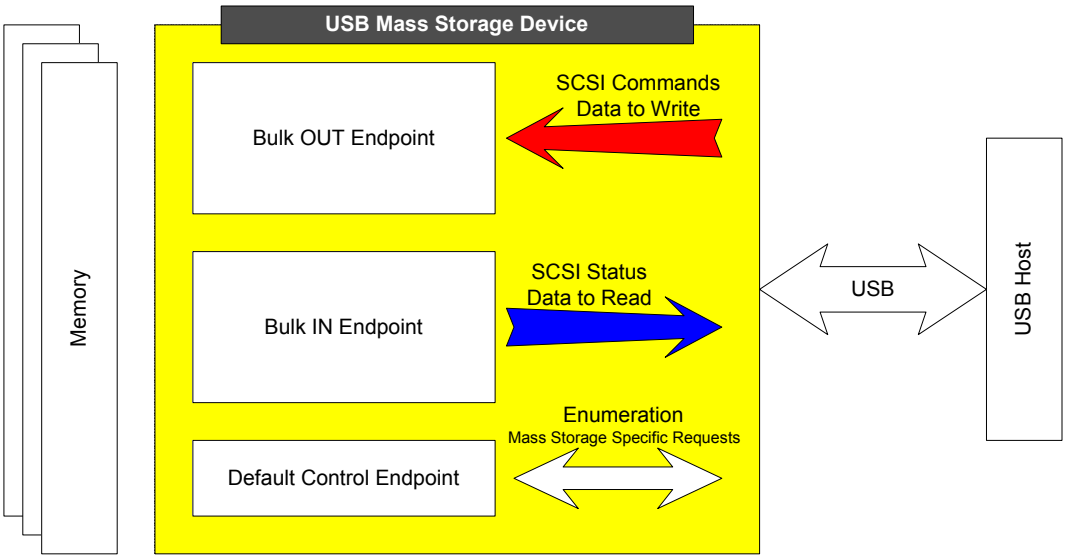
- Windows XP®: native driver
- Windows 2000®: native driver
- Windows Me®: native driver
- Windows 98SE®: Vendor specific driver required
- Linux®: USB mass storage is available in kernel 2.4 or later.
- USB mass storage is available in Mac OS 9/X® or later

This is very convenient for the final user who can now transfer any file from his own computer to any other.

For every Operating System, a USB Mass storage device is composed of:

- the Default Control Endpoint (0), to perform the enumeration process, to determine the number of LUN (Logical Unit Number), and to perform a reset recovery in case of mass storage error.
- a Bulk OUT Endpoint, used by the Host Controller to send the wrapped SCSI commands and to send the data to write inside the memory.
- a Bulk IN Endpoint, used by the Host Controller to read the wrapped SCSI status and to read data from the memory.

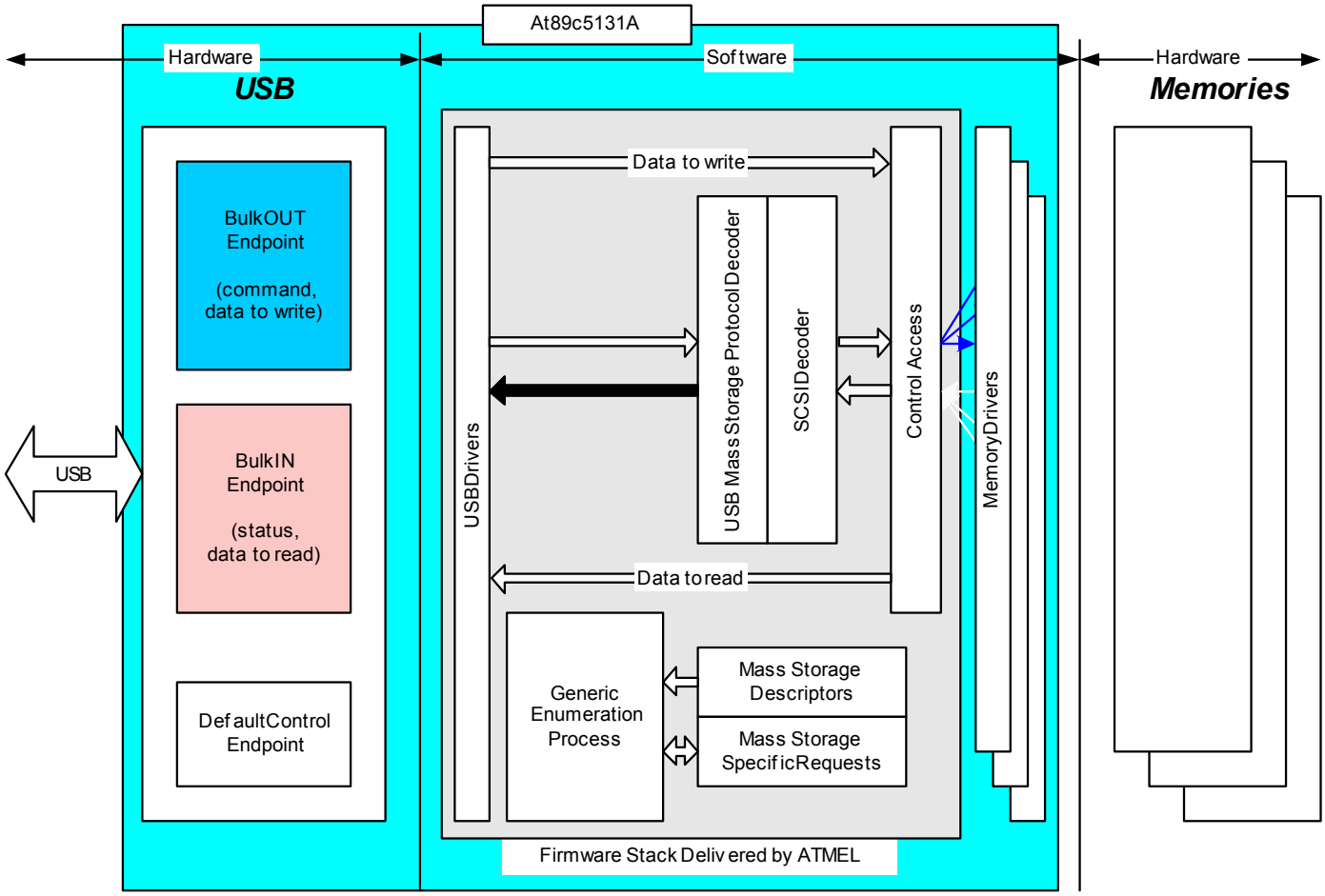
Figure 2. Mass Storage Device seen by a USB Host



USB Mass Storage Device Firmware Stack architecture

Figure 3. shows the architecture of the USB Mass Storage firmware stack.

Figure 3. USB Mass Storage Device Firmware architecture



The standard enumeration process (USB chapter 9 support) is performed by a generic function. The parameters and the descriptors are specific for each application. The USB Mass Storage class also requires specific request support.

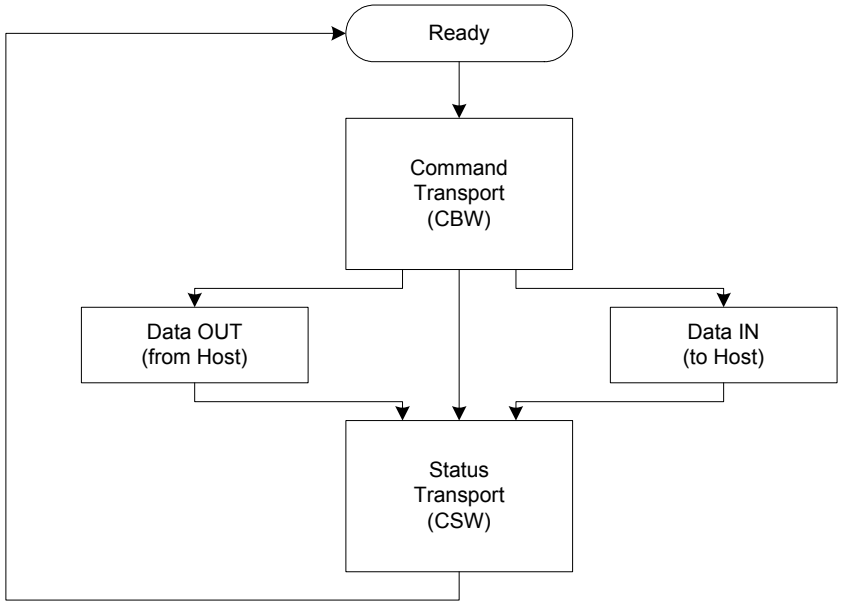
You can see that the commands coming from the USB Host controller are first un-wrapped by the USB Mass Storage Protocol Decoder function before entering into the SCSI decoder. Each SCSI command is then decoded and transmitted to the appropriate memory through a command set (Read, Write, is memory present, is memory write protected,...).

The memory answers are converted in SCSI status before being wrapped in USB CSW (Command Status Wrapper) and sent to the USB Host controller.

Remember that because the USB bus is a one master bus (the USB Host), each data transfer is initiated by the USB Host, following a specific Command-Data-Status flow (see Figure 4.).

This firmware is based on a scheduler with only the USB task running. Other application tasks can be very easily added into the scheduler without disturbing the USB management.

Figure 4. Command / Data / Status flow



The CBW (Command Block Wrapper) contains some USB information such as the LUN addressed, the length of the SCSI command, and of course, it also contains the SCSI command for the memory.

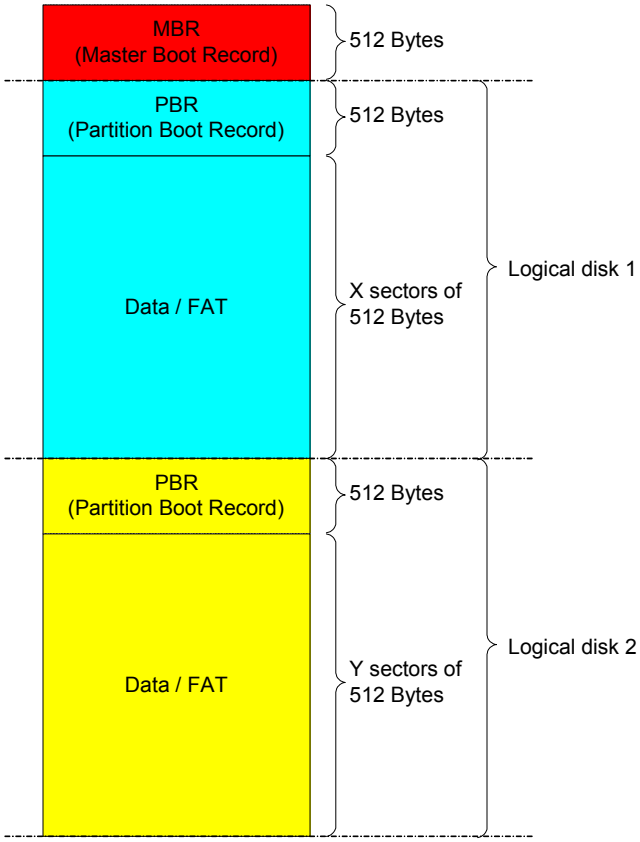
The data stage is not compulsory, depending on the SCSI command. For example, if the SCSI command is TEST UNIT READY, there is no data phase.

The CSW (Command Status Wrapper) contains the SCSI status. If the status is GOOD, the Host will send the next following command. If the status is different from GOOD (FAILED, PHASE ERROR,...), the Host will ask for more information regarding the error by sending a REQUEST SENSE command.

Memory Organization Overview

From a USB Host point of view, the USB memory is organized in logical blocks (sectors) of 512 Bytes.

Figure 5. Memory logical organization overview



The Host always addresses the memory with:

- the logical sector number
- the number of contiguous sectors to read / write.

The MBR (Master Boot Record) is always located at address 0. It contains information regarding the whole memory and the number of logical partitions inside. In case of a memory that has only 1 partition, the MBR is optional and can be replaced directly by the PBR (Partition Boot Record) that contains information regarding the logical partition and the entry point of the FAT (File Allocation Table).

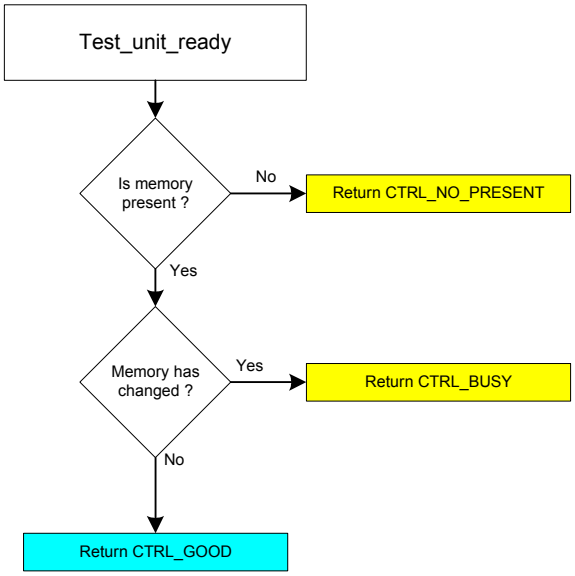
Memory Management

As seen in Figure 3., each memory is interfaced to the Atmel firmware by a specific memory driver, interfaced in the ctrl_access.c and ctrl_access.h files.

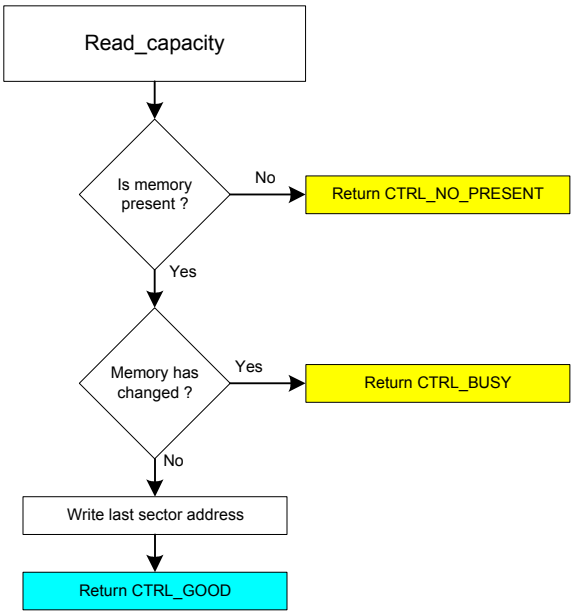
The following 8 functions have to be implemented in order to support a memory with the USB Mass Storage Device firmware. In order to support a new memory, the developer has to write the memory driver according to this memory interface. Some functions only return the status of the memory (present, write protected, total capacity and if the memory can be removed). The other functions are used to read or write into the memory. The functions read_10 and write_10 open the memory at a specific location. The functions usb_read and usb_write manage the data transfer between the USB Controller and the memory. Please refer to the Annex 1 and the Annex 2 for the write of these 2 functions.

Most of these functions returns a Ctrl_status byte that could be:

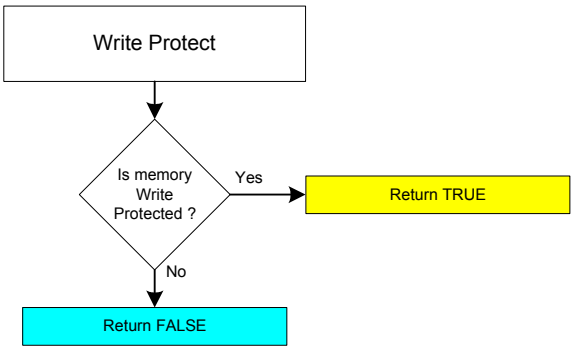
- CTRL_GOOD: function is PASS and another command can be sent
- CTRL_FAIL: there is a fail in the command execution
- CTRL_NO_PRESENT: the memory is not present
- CTRL_BUSY: the current memory is not initialized or its status has changed



Ctrl_status **<mem>_test_unit_ready**(void)
This function test the state of memory.

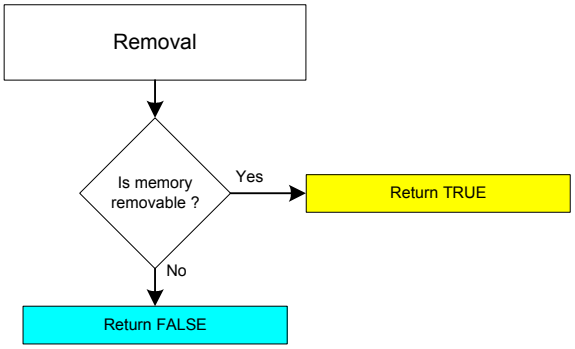


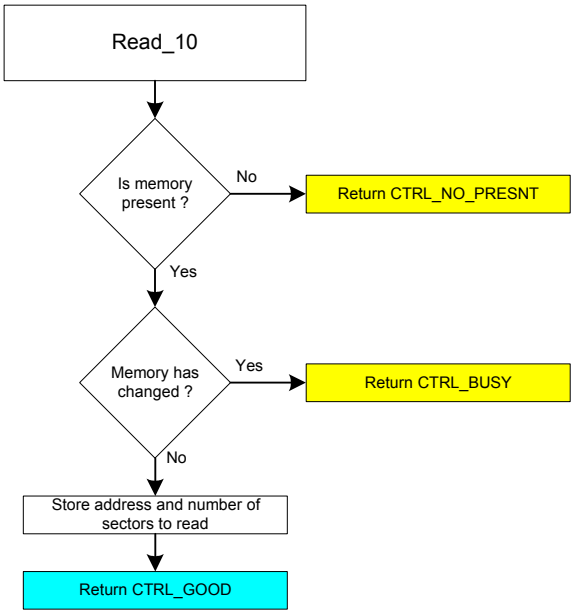
Ctrl_status **<mem>_read_capacity**(U32 *u32_nb_sector)
This function returns the address of the last valid sector, stored in u32_nb_sector. The sector size is fixed to 512 Bytes for OS compatibility.
For example, a memory of 16KBytes returns ((16 x 1024)/512) -1) = 31



Bool **<mem>_wr_protect**(void)
This function returns FALSE if the memory is not write protected.
This function returns TRUE if the memory is write protected.

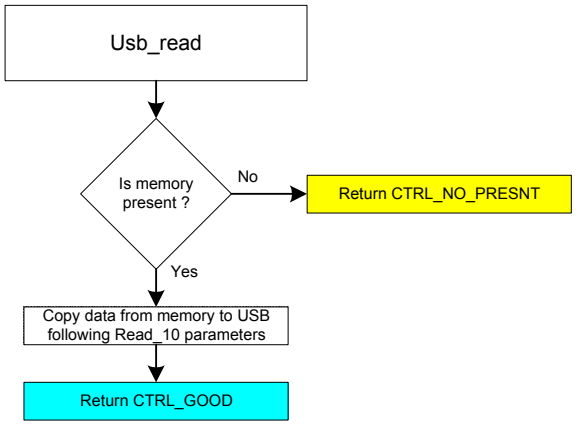
Bool **<mem>_removal**(void)
This function returns FALSE if the memory can't be removed.
This function returns TRUE if the memory can be removed.

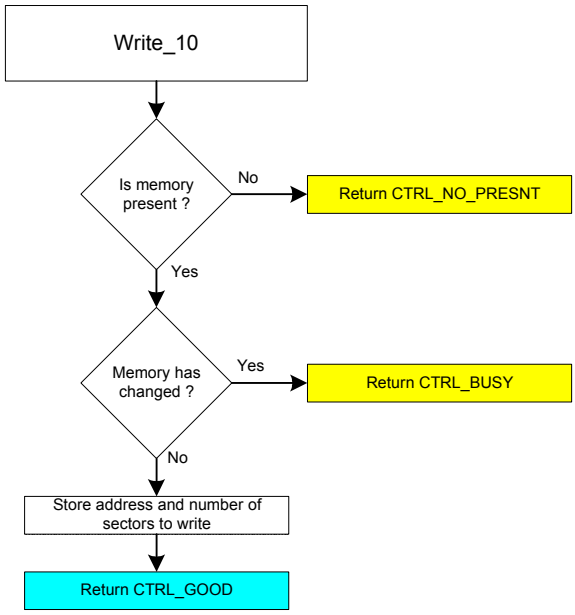




Ctrl_status **<mem>_read_10**(U32 addr , U16 nb_sector)
This function sets the sector address (addr) and the number of consecutive sector (512Bytes each) to read.

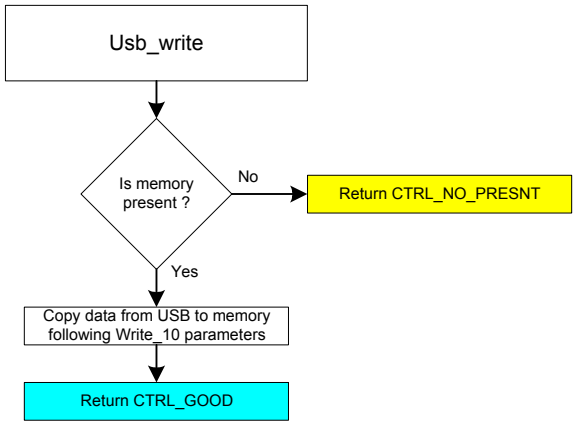
Ctrl_status **<mem>_usb_read**(void)
This function manages the data transfer from the memory to USB. The memory address and the number of sectors to read was previously defined by the <mem>_read_10(addr, nb_sector) function. Refer to the Annex 2 for copy data to USB.





Ctrl_status **<mem>_write_10**(U32 addr , U16 nb_sector)
This function sets the sector address (addr) and the number of consecutive sector (512Bytes each) to write.

Ctrl_status **<mem>_usb_write**(void)
This function manages the data transfer from USB to the memory. The memory address and the number of sectors to read was previously defined by the <mem>_write_10(addr, nb_sector) function. Refer to the Annex 2 for copy data from USB.



Integration of a Memory

The integration of a memory on the USB Mass Storage stack is performed in the conf_access.h.
The corresponding LUN has to be first set to ENABLE and the corresponding functions have to be defined.

The USB Mass Storage stack supports up to 8 different LUN.

Here is an example with the virtual memory sets as LUN_0:

```
// Active the Logical Unit
#define  LUN_0                ENABLE
#define  LUN_1                DISABLE
#define  LUN_2                DISABLE
#define  LUN_3                DISABLE
#define  LUN_4                DISABLE
#define  LUN_5                DISABLE
#define  LUN_6                DISABLE
#define  LUN_7                DISABLE

// LUN 0 DEFINE
#define  LUN_0_INCLUDE "lib_mem\virtual_mem\virtual_mem.h"
#define  Lun_0_test_unit_ready()                 virtual_test_unit_ready()
#define  Lun_0_read_capacity(nb_sect)virtual_read_capacity(nb_sect)
#define  Lun_0_wr_protect()                      virtual_wr_protect()
#define  Lun_0_removal()                         virtual_removal()
#define  Lun_0_read_10(ad, sec)                  virtual_read_10(ad, sec)
#define  Lun_0_usb_read()                        virtual_usb_read()
#define  Lun_0_write_10(ad, sec)                 virtual_write_10(ad, sec)
#define  Lun_0_usb_write()                       virtual_usb_write()
```

USB Configuration and Customizing

USB Clock

The USB clock is automatically configured using the FOSC value.

The FOSC value is the oscillator frequency, in KHz, and is defined in the config.h file. For example, the value of FOSC is set to 16000 for an oscillator frequency of 16 MHz.

Enumeration Parameters

Vendor ID / Product ID

For an application based on the USB Mass Storage stack, the Vendor ID and the Product ID have to be changed with the ID assigned to the manufacturer by USB-IF.

The Vendor ID and Product ID are defined in the conf_usb.h file.

For this demo, these parameters are:

```
#define VENDOR_ID             0xEB03 // Atmel vendor ID = 03EBh
#define PRODUCT_ID            0x1320 // Product ID: 2013h = Mass Storage
```

Strings

The manufacturer can change the USB strings in the conf_usb.h file.

These strings are:

- Manufacturer Name
- Product Name
- Serial Number (at least 12 characters)

Do not forget to adapt the string length definitions according to the modifications:

```
#define USB_MN_LENGTH         5
#define USB_PN_LENGTH         28
#define USB_SN_LENGTH         13
```

Endpoint

If an application uses different Endpoints than those used in the demo, this can be done in the conf_usb.h file by changing these define:

```
#define EP_MS_IN              4
#define EP_MS_OUT             5
```

In such a case, do not forget to specify in the conf_usb.h file if the endpoints to use support the Ping-Pong mode or not:

```
#define NO_SUPPORT_USB_PING_PONG
```

or no define.

Annex 1: Reading Data from USB

The figures below show how to manage a Bulk OUT endpoint. These algorithms can be used to build your own <mem>_usb_write function required by the USB Mass Storage Device stack.

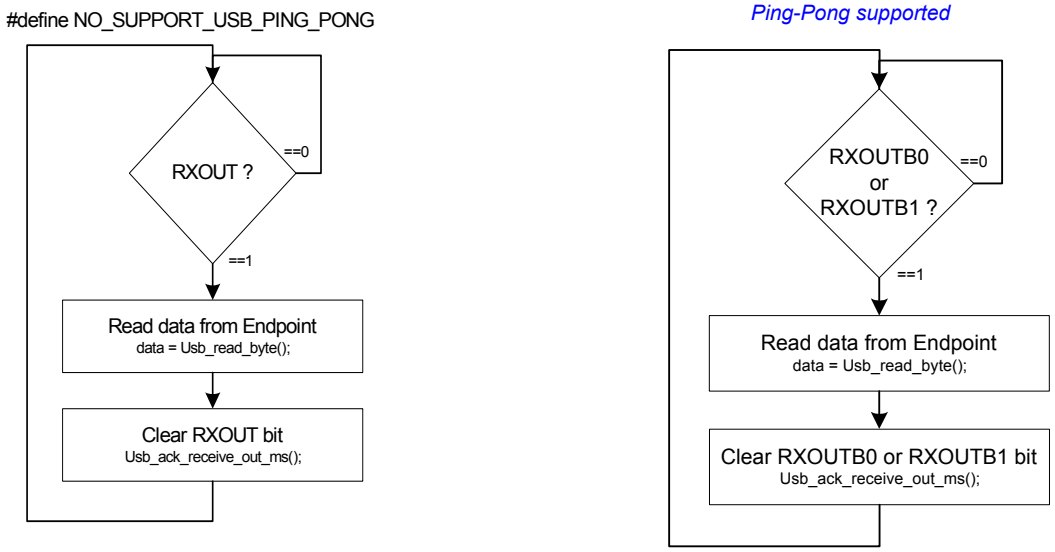
In case the OUT endpoint declared for the Mass Storage class **does not** support the Ping-Pong mode, the #define NO_SUPPORT_USB_PING_PONG has to be set in the conf_usb.h file. This disable the automatic management of the Ping-Pong for the mass storage endpoint.

In case the OUT endpoint declared for the Mass Storage class **does** support the Ping-Pong mode, the #define NO_SUPPORT_USB_PING_PONG has to be removed in the conf_usb.h file.

In every case, the following functions of the USB driver can be used:

- ***Is_usb_receive_out()***: this function check if a new OUT message has been received, for a ping-pong endpoint or not.
- ***Usb_read_byte()***: this function return the current byte store in the endpoint FIFO.
- ***Usb_ack_receive_out_ms()***: this function frees the last bank in the endpoint FIFO to allow a new OUT message to be stored. The behavior of this function is different if the endpoint support the ping-pong mode or not, that's why the user has to check if the #define NO_SUPPORT_USB_PING_PONG is correctly set or removed.

Figure 6. Reading data from USB



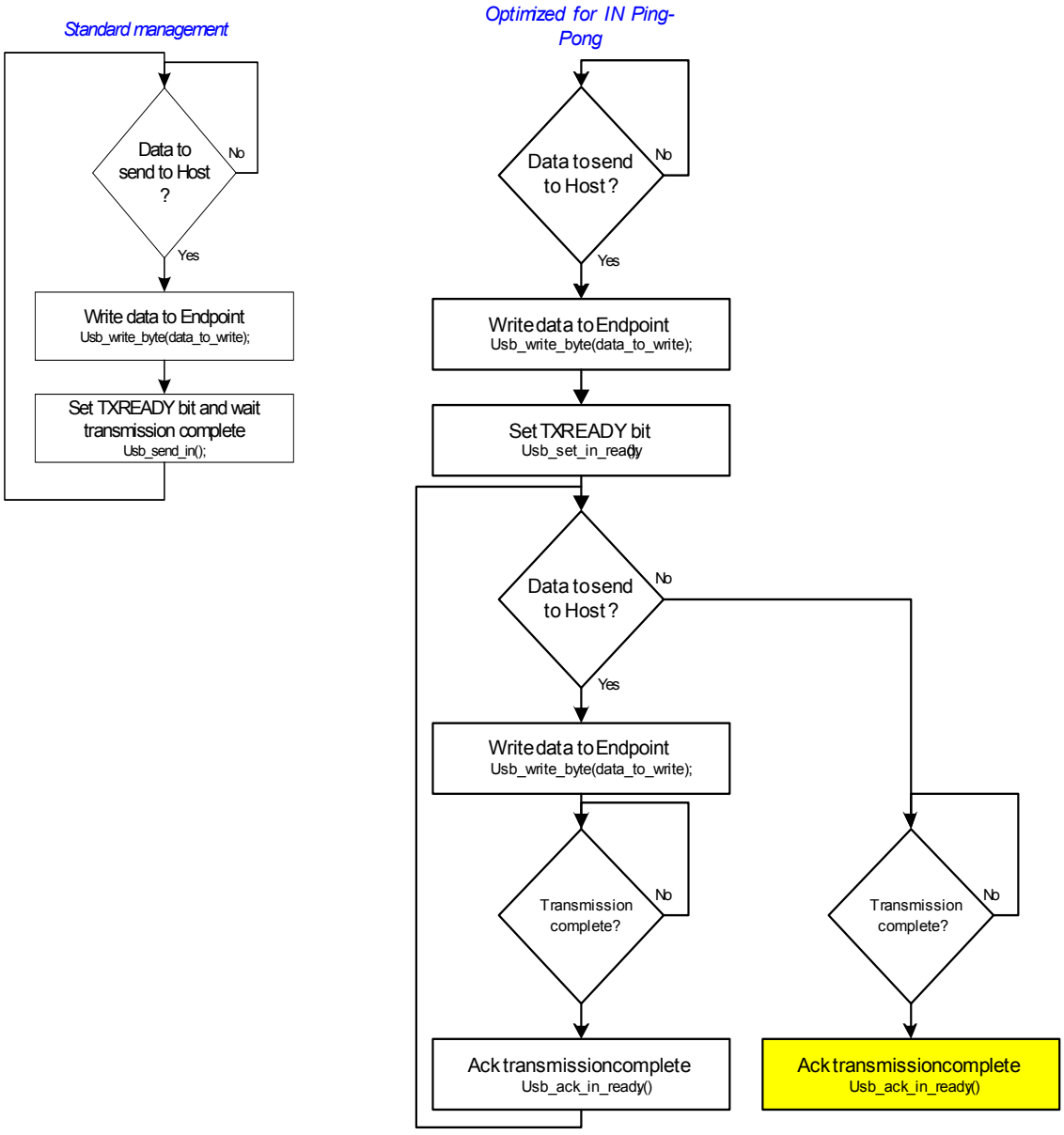
Annex 2: Writing Data to USB

The figures below show how to manage a Bulk IN endpoint. These algorithms can be used to build its own <mem>_usb_read function required by the USB Mass Storage Device stack.

The standard management can be used for a standard endpoint or a Ping-Pong endpoint. It consists of writing into the endpoint FIFO, advise the USB controller that a new IN message is ready and wait for the data to be sent on the USB before writing new data.

In order to increase Mass Storage Read performances, the firmware can be optimized if the IN endpoint declared for the Mass Storage Device class support the Ping-Pong mode.

Figure 7. Writing data to USB



Annex 3: Implementation Example with 1 Nand Flash

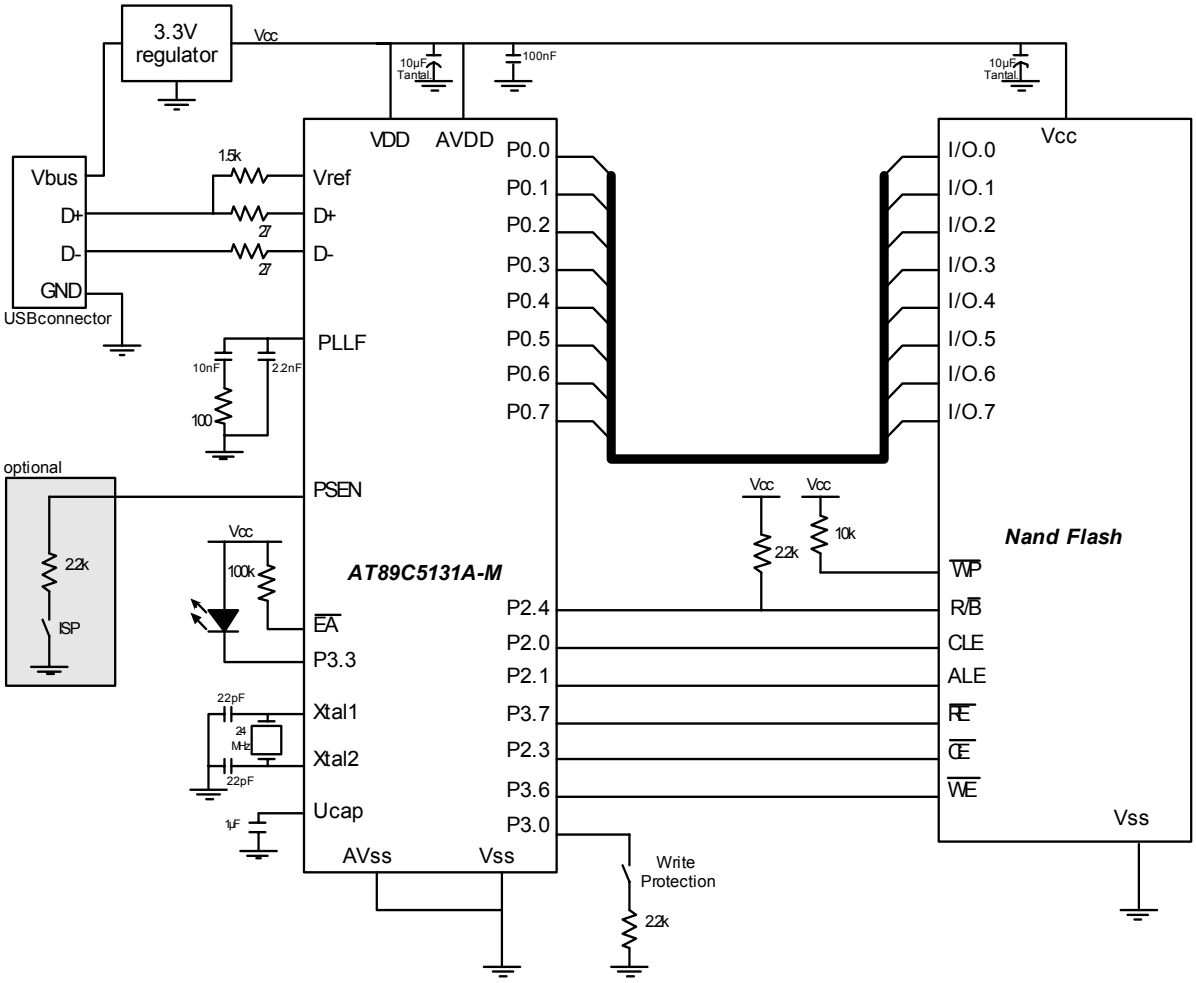
The schematic below is an example of the USB Mass Storage key.

This example is based on the AT89C5131 microcontroller. This microcontroller integrates a USB macro which supports the Ping-Pong mode on several endpoints.

The writing speed is between 300 and 550 kBytes/s, depending on the FAT implementation.
The reading speed is about 700-800 kBytes/s.

P0 and P2 ports are mandatory in order to access the Nand Flash using MOVX instructions.

Figure 8. Implementation example with the AT89C5131A and 1 Nand Flash



Atmel Corporation

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

Regional Headquarters

Europe
Atmel Sarl
Route des Arsenaux 41
Case Postale 80
CH-1705 Fribourg
Switzerland
Tel: (41) 26-426-5555
Fax: (41) 26-426-5500

Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimshatsui
East Kowloon
Hong Kong
Tel: (852) 2721-9778
Fax: (852) 2722-1369

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

Atmel Operations

Memory
2325 Orchard Parkway
San Jose, CA 95131
Tel: 1(408) 441-0311
Fax: 1(408) 436-4314

Microcontrollers
2325 Orchard Parkway
San Jose, CA 95131
Tel: 1(408) 441-0311
Fax: 1(408) 436-4314

La Chantrerie
BP 70602
44306 Nantes Cedex 3, France
Tel: (33) 2-40-18-18-18
Fax: (33) 2-40-18-19-60

ASIC/ASSP/Smart Cards
Zone Industrielle
13106 Rousset Cedex, France
Tel: (33) 4-42-53-60-00
Fax: (33) 4-42-53-60-01

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906
Tel: 1(719) 576-3300
Fax: 1(719) 540-1759

Scottish Enterprise Technology Park
Maxwell Building
East Kilbride G75 0QR, Scotland
Tel: (44) 1355-803-000
Fax: (44) 1355-242-743

RF/Automotive
Theresienstrasse 2
Postfach 3535
74025 Heilbronn, Germany
Tel: (49) 71-31-67-0
Fax: (49) 71-31-67-2340

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906
Tel: 1(719) 576-3300
Fax: 1(719) 540-1759

Biometrics/Imaging/Hi-Rel MPU/
High Speed Converters/RF Datacom
Avenue de Rochepleine
BP 123
38521 Saint-Egreve Cedex, France
Tel: (33) 4-76-58-30-00
Fax: (33) 4-76-58-34-80

Literature Requests
www.atmel.com/literature

7516A–USB–05/05 /xM