

## Introduction

STM32Cube™ initiative was originated by STMicroelectronics to ease developers life by reducing development efforts, time and cost. STM32Cube covers STM32 portfolio.

STM32Cube Version 1.x includes:

- The STM32CubeMX, a graphical software configuration tool that allows to generate C initialization code using graphical wizards.
- A comprehensive embedded software platform, delivered per series (such as STM32CubeF4 for STM32F4 series)
  - The STM32Cube HAL, an STM32 abstraction layer embedded software, ensuring maximized portability across STM32 portfolio
  - A consistent set of middleware components such as RTOS, USB, TCP/IP, Graphics.
  - All embedded software utilities coming with a full set of examples.

The Universal Serial Bus (USB) is the most successful interconnect in the history of personal computing which is used to connect devices like mouse, game-pads and joysticks, scanners, digital cameras, and printers. USB has also migrated into consumer electronics and mobile products.

This user manual describes the STM32Cube USB device library which is part of the STM32Cube firmware package that can be downloaded free from ST website (<http://www.st.com/stm32cube>). It is intended for developers who use STM32Cube firmware on STM32 microcontrollers. It describes how to start and implement a USB device applications for most common USB device classes (HID, MSC, Audio, CDC...) based on the USB device stack that supports all STM32 microcontroller series.

**Note:** *This document is applicable to all STM32 Series that feature an USB peripheral. However for simplicity reason, the STM32F4xx devices and STM32CubeF4 are used as reference platform. To know more about the examples implementation on your STM32 device, refer to the readme file provided within the associated STM32Cube firmware package.*



# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Overview</b>                             | <b>6</b>  |
| 1.1      | Acronyms and abbreviations                  | 6         |
| 1.2      | Additional Information                      | 6         |
| 1.3      | References                                  | 6         |
| <b>2</b> | <b>USB device library description</b>       | <b>7</b>  |
| 2.1      | Overview                                    | 7         |
| 2.2      | Features                                    | 8         |
| <b>3</b> | <b>USB device library architecture</b>      | <b>9</b>  |
| 3.1      | Architecture overview                       | 9         |
| <b>4</b> | <b>USB OTG Hardware Abstraction Layer</b>   | <b>11</b> |
| 4.1      | Driver architecture                         | 11        |
| 4.2      | USB driver programming manual               | 12        |
| 4.2.1    | Configuring USB driver structure            | 12        |
| <b>5</b> | <b>USB device library overview</b>          | <b>15</b> |
| 5.1      | USB device library description              | 15        |
| 5.1.1    | USB device library flow                     | 15        |
| 5.1.2    | USB device data flow                        | 19        |
| 5.1.3    | Core interface with low level driver        | 20        |
| 5.1.4    | USB device library interfacing model        | 21        |
| 5.1.5    | Configuring the USB device firmware library | 22        |
| 5.1.6    | USB control functions                       | 23        |
| 5.2      | USB device library functions                | 24        |
| 5.3      | USB device class interface                  | 27        |
| <b>6</b> | <b>USB device library class module</b>      | <b>30</b> |
| 6.1      | HID class                                   | 31        |
| 6.1.1    | HID class implementation                    | 31        |
| 6.1.2    | HID user interface                          | 31        |
| 6.1.3    | HID Class Driver APIs                       | 32        |

---

|          |  |           |
|----------|--|-----------|
| 6.2      | Mass storage class . . . . .                                 | 32        |
| 6.2.1    | Mass storage class implementation . . . . .                  | 32        |
| 6.2.2    | Get Max MUN (class-specific request) . . . . .               | 34        |
| 6.2.3    | MSC Core files . . . . .                                     | 35        |
| 6.2.4    | Disk operation structure definition . . . . .                | 37        |
| 6.2.5    | Disk operation functions . . . . .                           | 38        |
| 6.3      | Device firmware upgrade (DFU) class . . . . .                | 38        |
| 6.3.1    | Device firmware upgrade (DFU) class implementation . . . . . | 39        |
| 6.3.2    | Device firmware upgrade (DFU) Class core files . . . . .     | 41        |
| 6.4      | Audio class . . . . .  | 43        |
| 6.4.1    | Audio class implementation . . . . .                         | 43        |
| 6.4.2    | Audio core files . . . . .                                   | 44        |
| 6.4.3    | How to use this driver . . . . .                             | 46        |
| 6.4.4    | Audio known limitations . . . . .                            | 47        |
| 6.5      | Communication device class (CDC) . . . . .                   | 47        |
| 6.5.1    | Communication . . . . .                                      | 48        |
| 6.5.2    | Data IN transfer management (from device to host) . . . . .  | 48        |
| 6.5.3    | Data OUT transfer management (from host to device) . . . . . | 48        |
| 6.5.4    | Command request management . . . . .                         | 48        |
| 6.5.5    | Command device class (CDC) core files . . . . .              | 48        |
| 6.5.6    | How to use . . . . .   | 51        |
| 6.5.7    | CDC known limitations . . . . .                              | 52        |
| 6.6      | Adding a custom class . . . . .                              | 52        |
| 6.7      | Library footprint optimization . . . . .                     | 53        |
| <b>7</b> | <b>Frequently-asked questions . . . . .</b>                  | <b>56</b> |
| <b>8</b> | <b>Revision history . . . . .</b>                            | <b>58</b> |

## List of tables

|           |  |    |
|-----------|--|----|
| Table 1.  | List of terms . . . . .                          | 6  |
| Table 2.  | USB device status . . . . .                      | 14 |
| Table 3.  | Standard requests . . . . .                      | 16 |
| Table 4.  | API description . . . . .                        | 20 |
| Table 5.  | Low level Event Callback functions . . . . .     | 22 |
| Table 6.  | USB library configuration . . . . .              | 22 |
| Table 7.  | USB device core files . . . . .                  | 24 |
| Table 8.  | Class drivers files . . . . .                    | 24 |
| Table 9.  | usbd_core (.c,.h) files . . . . .                | 24 |
| Table 10. | usbd_ireq (.c,.h) files functions . . . . .      | 25 |
| Table 11. | usbd_ctrlq (.c,.h) files functions . . . . .     | 26 |
| Table 12. | USB device class files . . . . .                 | 30 |
| Table 13. | usbd_hid.c,h files . . . . .                     | 32 |
| Table 14. | SCSI commands . . . . .                          | 34 |
| Table 15. | usbd_msc (.c,.h) files . . . . .                 | 35 |
| Table 16. | usbd_msc_bot (.c,.h) files . . . . .             | 35 |
| Table 17. | usbd_msc_scsi (.c,.h) . . . . .                  | 36 |
| Table 18. | Disk operation functions . . . . .               | 38 |
| Table 19. | DFU states . . . . .                             | 39 |
| Table 20. | Supported requests . . . . .                     | 41 |
| Table 21. | usbd_dfu (.c,.h) files . . . . .                 | 41 |
| Table 22. | Audio control requests . . . . .                 | 44 |
| Table 23. | usbd_audio_core (.c,.h) files . . . . .          | 44 |
| Table 24. | usbd_audio_if (.c,.h) files . . . . .            | 46 |
| Table 25. | Audio player states . . . . .                    | 46 |
| Table 26. | usbd_cdc (.c,.h) files . . . . .                 | 49 |
| Table 27. | Configurable CDC parameters . . . . .            | 50 |
| Table 28. | usbd_cdc_interface (.c,.h) files . . . . .       | 50 |
| Table 29. | Variables used by usbd_cdc_xxx_if.c/.h . . . . . | 51 |
| Table 30. | Document revision history . . . . .              | 58 |

## List of figures

|            |  |    |
|------------|--|----|
| Figure 1.  | STM32Cube USB device library . . . . .                             | 8  |
| Figure 2.  | USB device library architecture . . . . .                          | 10 |
| Figure 3.  | Driver architecture overview . . . . .                             | 11 |
| Figure 4.  | USBD_HandleTypeDef . . . . .                                       | 13 |
| Figure 5.  | USB device library directory structure . . . . .                   | 15 |
| Figure 6.  | Data structure for SETUP packet . . . . .                          | 16 |
| Figure 7.  | USB device library process flowchart . . . . .                     | 18 |
| Figure 8.  | USB device data flow . . . . .                                     | 20 |
| Figure 9.  | USB device library interfacing model . . . . .                     | 21 |
| Figure 10. | USB Class callback structure . . . . .                             | 27 |
| Figure 11. | USB device descriptors structure . . . . .                         | 28 |
| Figure 12. | Example of USBD_HID_SendReport function . . . . .                  | 31 |
| Figure 13. | BOT Protocol architecture . . . . .                                | 33 |
| Figure 14. | Disk operation structure description . . . . .                     | 37 |
| Figure 15. | Example of standard inquiry definition . . . . .                   | 37 |
| Figure 16. | DFU Interface state transitions diagram . . . . .                  | 40 |
| Figure 17. | Audio core structures . . . . .                                    | 45 |
| Figure 18. | CDC core structures . . . . .                                      | 49 |
| Figure 19. | CDC interface callback structure . . . . .                         | 50 |
| Figure 20. | Example of USB descriptors declared as constants . . . . .         | 54 |
| Figure 21. | Example of dynamic memory allocation for class structure . . . . . | 55 |

# 1 Overview

## 1.1 Acronyms and abbreviations

*Table 1* gives a brief definition of acronyms and abbreviations used in this document.

**Table 1. List of terms**

| Term | Meaning   |
|------|---|
| API  | Application Programming Interface                                   |
| CDC  | Communication Device Class  |
| DFU  | Device Firmware Upgrade   |
| FS   | Full Speed (12 Mbps)  |
| HID  | Human Interface Device  |
| Mbps | Megabit per second  |
| MSC  | Mass Storage Class  |
| OTG  | On-The-Go: An OTG peripheral can switch HOST/DEVICE role on the fly |
| PID  | USB Product Identifier  |
| SCSI | Small Computer System Interface                                     |
| SOF  | Start Of Frame  |
| VID  | USB Vendor Identifier   |
| USB  | Universal Serial Bus  |

## 1.2 Additional Information

In addition to this document STMicroelectronics provides several other resources related to the USB:

- USB Host library user manual (UM1720)
- Description of STM32F4xx HAL drivers (UM1725) where you can find the two USB generic drivers description (HCD for Host and PCD for Device)

## 1.3 References

- Universal Serial Bus Specification, Revision 2.0, <http://www.usb.org>
- USB device class specifications (Audio, HID, MSC, etc.): <http://www.usb.org>

## 2 USB device library description

### 2.1 Overview

STMicroelectronics offers to its customers new USB stacks (device stack and host stack) that support all STM32 MCUs together with many development tools such as *Atollic®* *TrueSTUDIO*, *IAR Embedded Workbench for ARM®*, and *Keil uVision®*.

This document focuses on USB device stack. For the host stack, please refer to the related users manual.

The USB device library is generic for all STM32 microcontrollers, only the HAL layer is adapted to each STM32 device.

The USB device library comes on top of the STM32Cube USB device HAL driver and offers all the APIs required to develop a USB device application. The present document describes the STM32Cube USB device library middleware module and illustrates how the user can develop easily his own USB device application by using this library.

The USB device library is a part of STM32Cube package for each STM32 series.

It contains:

- The USB low level driver
- Commonly used USB class drivers
- A set of applications for the most common USB device classes supporting USB Full speed and High speed transfer types (control, interrupt, bulk and isochronous).

The USB device library aim is to provide at least one firmware demonstration per USB transfer type:

- Human Interface Device HID  
HID Joystick demonstration based on the embedded joystick on the EVAL boards and Custom HID examples
- Audio streaming  
Audio device example for streaming audio data
- Communication Device (CDC)  
VCP USB-to-RS232 bridge to realize a virtual COM port.
- Mass storage  
Mass storage demonstration based on the microSD card available on the EVAL boards.
- Device Firmware Upgrade  
DFU for firmware downloads and uploads
- Dual Core devices demonstration  
Based on mass storage with Human interface and mass storage with CDC device examples

Among the topics covered are:

- USB device library architecture
- USB device library description
- USB device library state machine overview
- USB device classes overview.

## 2.2 Features

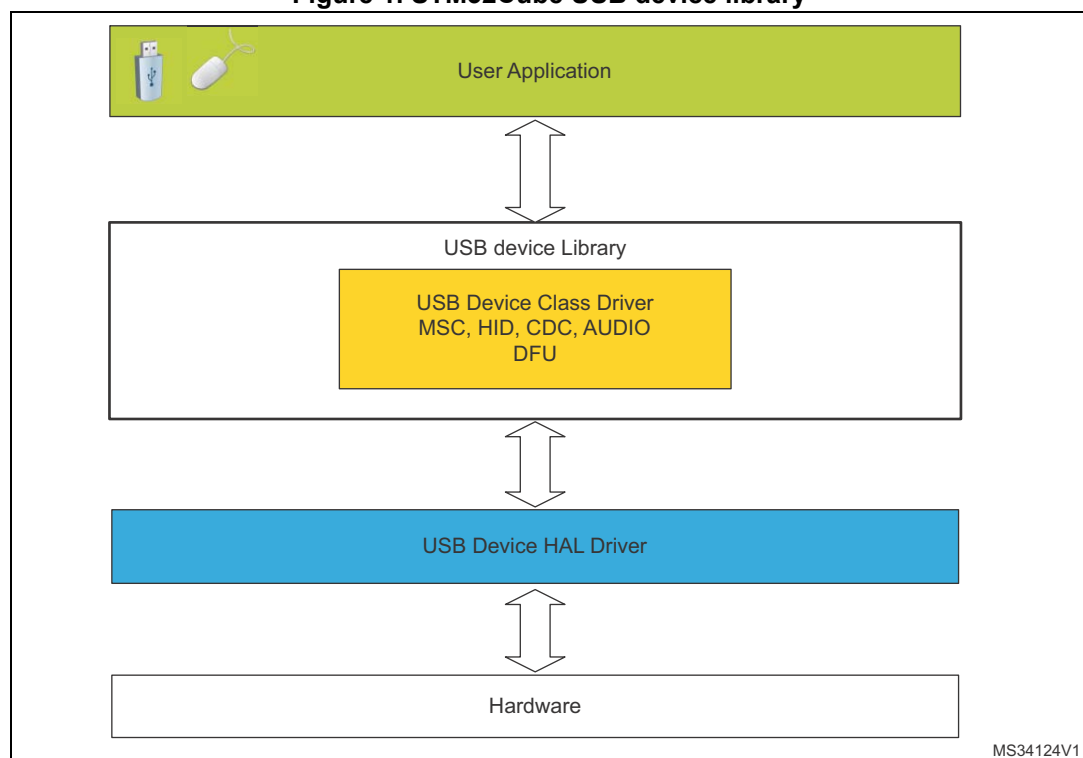
The main USB device library features are:

- Support of multi packet transfer features allowing sending big amount of data without splitting it into max packet size transfers.
- Support of up to 3 back to back transfers on control endpoints (compatible with OHCI controllers).
- Configuration files to change the core and the library configuration without changing the library code (Read Only).
- 32-bits aligned data structures to handle DMA based transfer in High speed modes.
- Support of multi USB OTG core instances from user level (configuration file).

*Note: The USB device library can be used with or without RTOS; the CMSIS RTOS wrapper is used to make abstraction with OS kernel.*

*USB device examples do not display log messages.*

**Figure 1. STM32Cube USB device library**



1. The user application is shown in green, the USB library core blocks in orange and the USB Device HAL driver in blue.



## 3 USB device library architecture

### 3.1 Architecture overview

The USB device library is mainly divided into three layers. The applications is developed on top of them as shown in [Figure 2: USB device library architecture](#).

The first Layer is composed of the **core** and the **class** drivers.

- Core drivers

The library core is composed of four main blocks:

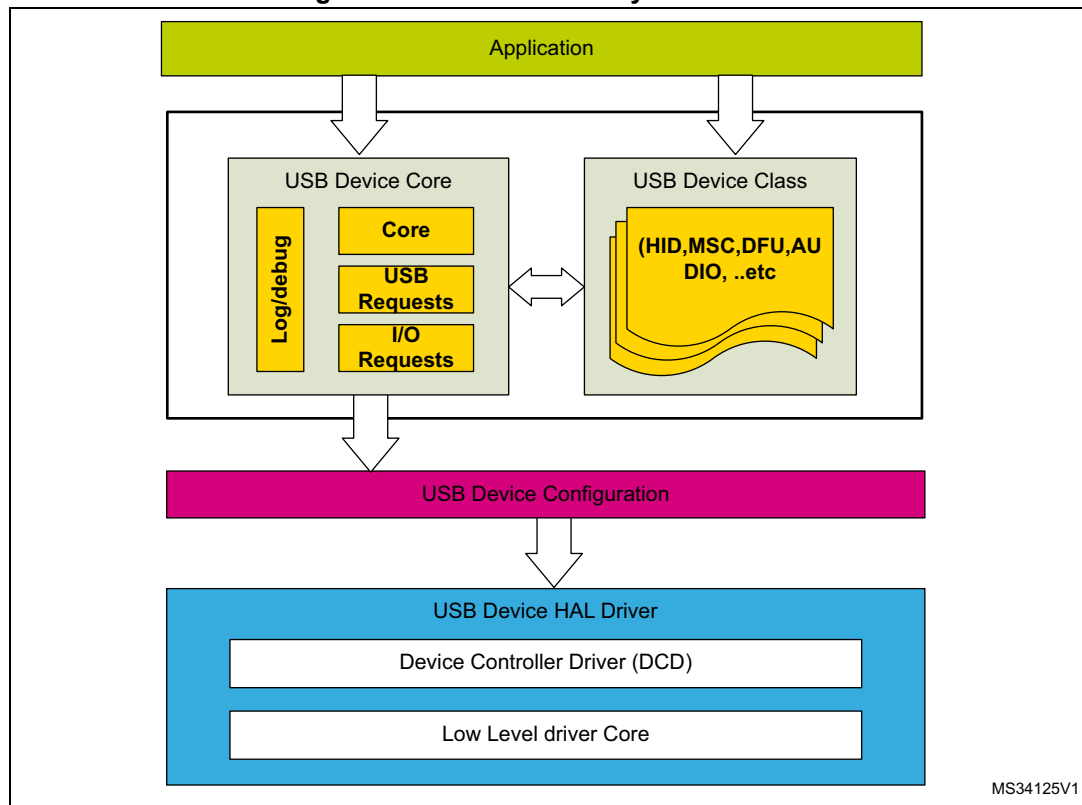
- USB core module that offers a full set of APIs to manage the internal USB device library state machine and call back processes from USB Interrupts
- USB Requests module that handles chapter 9 requests
- USB I/O requests module: handles low level I/O requests
- USB Log and debug module that follows debug level `USB_DEBUG_LEVEL`, outputs user, log, error and debug messages.

- Class drivers

The USB Device classes is composed of a set predefined class drivers ready to be linked to the USB core through the `USBD_RegisterClass ()` routine.

The USB device library is a USB 2.0 compatible generic USB device stack, compliant with all the STM32 USB cores. It can be easily linked to any USB HAL driver thanks to the configuration wrapper file which avoids any dependency between the USB library and the low level drivers.

Figure 2. USB device library architecture



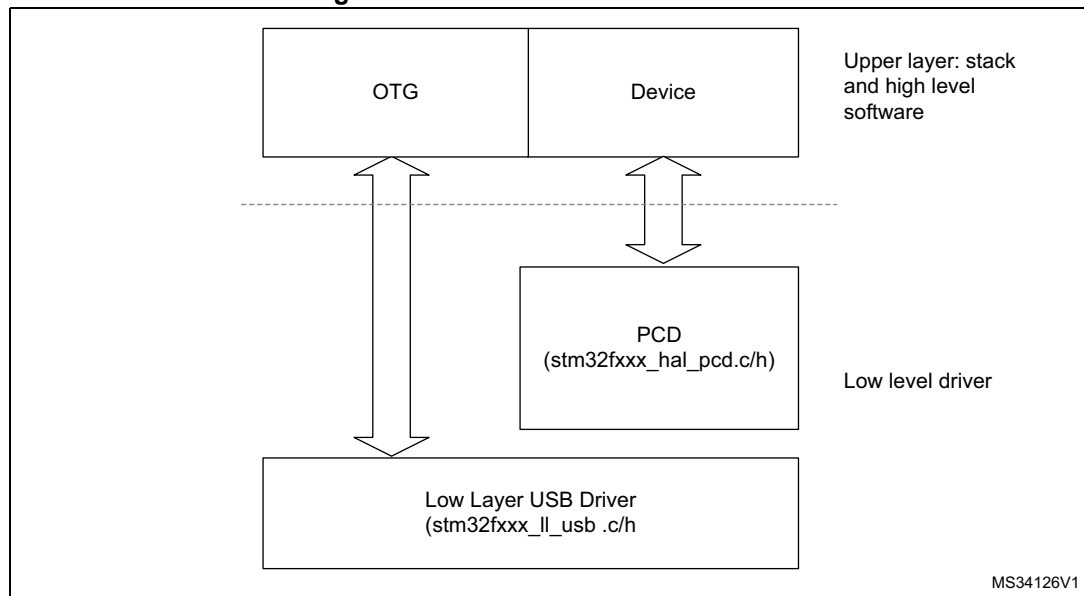
1. The USB library core blocks are shown in orange, the USB Device Configuration in magenta and the USB HAL driver in blue.

## 4 USB OTG Hardware Abstraction Layer

The low level driver can be used to connect the USB OTG core with the high level stack.

### 4.1 Driver architecture

Figure 3. Driver architecture overview



- The bottom layer (Low Layer USB driver) provides common APIs for device and OTG modes. It performs the core initialization in each mode and controls of the transfer flow.
- The Peripheral controller driver (PCD) layer provides an API for device mode access and the main interrupt routine for this mode.
- The OTG controller driver (OTG) layer provides an API for OTG mode access and the main interrupt routine for this mode.

**Note:** For more details on how to use the PCD driver, please refer to UM1725 that describes all PCD driver APIs.

## 4.2 USB driver programming manual

### 4.2.1 Configuring USB driver structure

#### Device initialization

The device is initialized using the following function contained in `stm32fxxx_hal_pcd.c` file:

```
HAL_StatusTypeDef HAL_PCD_Init(PCD_HandleTypeDef *hpcd)
```

#### Endpoint configuration

Once the USB core is initialized, the upper layer can call the low level driver to open or close the active endpoint and start transferring data. The following two APIs can be used:

```
HAL_StatusTypeDef HAL_PCD_EP_Open(PCD_HandleTypeDef *hpcd, uint8_t ep_addr,
uint16_t ep_mps, uint8_t ep_type)
```

```
HAL_StatusTypeDef HAL_PCD_EP_Close(PCD_HandleTypeDef *hpcd, uint8_t ep_addr)
```

`ep_addr`, `ep_mps` and `ep_type` are respectively the endpoint address, the maximum data transfer and transfer type.

#### Device core structure

The main structure used in the device library is the device handle. Its type is “`USBD_HandleTypeDef`” (see [Figure 4 on page 13](#)).

The USB Global device structure contains all the variables and structures used to keep all the information related to devices in real-time, as well as store the control transfer state machine and the endpoint information and status.

In this structure, `dev_config` holds the current USB device configuration and `ep0_state` controls the state machine with the following states:

```
/* EP0 State */
#define USBD_EP0_IDLE                0
#define USBD_EP0_SETUP               1
#define USBD_EP0_DATA_IN             2
#define USBD_EP0_DATA_OUT            3
#define USBD_EP0_STATUS_IN           4
#define USBD_EP0_STATUS_OUT          5
#define USBD_EP0_STALL                6
```

In this structure, `dev_state` defines the connection, configuration and power status:

```
/* Device Status */
#define USBD_DEFAULT                  1
#define USBD_ADDRESSED                2
#define USBD_CONFIGURED              3
#define USBD_SUSPENDED               4
```

**Note:** The USB specification (see Chapter 9) defines six USB device states:

**Attached:** the device is attached to the USB but is not powered by the USB.

**Powered:** the device is attached to the USB and has been powered but has not yet received any reset request.

**Default:** the device is attached to the USB. It is powered and reset, but no unique address has been assigned to it.

**Address:** the device is attached to the USB, it is powered and reset and has had a unique address assigned to it.

**Configured:** the device is already in address state and configured. It is not in suspend state.

**Suspended:** the device is attached and configured, but has not detected any activity on the bus for at least 3 ms.

**Figure 4. USBD\_HandleTypeDef**

```
typedef struct _USBHandleTypeDef
{
    uint8_t            id;
    uint32_t           dev_config;
    uint32_t           dev_default_config;
    uint32_t           dev_config_status;
    USBSpeedTypeDef    dev_speed;
    USBEndpointTypeDef ep_in[15];
    USBEndpointTypeDef ep_out[15];
    uint32_t           ep0_state;
    uint32_t           ep0_data_len;
    uint8_t            dev_state;
    uint8_t            dev_old_state;
    uint8_t            dev_address;
    uint8_t            dev_connection_status;
    uint8_t            dev_test_mode;
    uint32_t           dev_remote_wakeup;
    USBSetupReqTypeDef request;
    USBDescriptorsTypeDef *pDesc;
    USBClassTypeDef    *pClass;
    void                *pClassData;
    void                *pUserData;
    void                *pData;
} USBHandleTypeDef;
```

## USB data transfer flow

The PCD layer provides all the APIs required to start and control a transfer flow. This is done through the following set of functions:

```
HAL_StatusTypeDef HAL_PCD_EP_Transmit(PCD_HandleTypeDef *hpcd, uint8_t
ep_addr, uint8_t *pBuf, uint32_t len)
HAL_StatusTypeDef HAL_PCD_EP_Receive(PCD_HandleTypeDef *hpcd, uint8_t
ep_addr, uint8_t *pBuf, uint32_t len)
HAL_StatusTypeDef HAL_PCD_EP_SetStall(PCD_HandleTypeDef *hpcd, uint8_t
ep_addr)
HAL_StatusTypeDef HAL_PCD_EP_ClrStall(PCD_HandleTypeDef *hpcd, uint8_t
ep_addr)
HAL_StatusTypeDef HAL_PCD_EP_Flush(PCD_HandleTypeDef *hpcd, uint8_t
ep_addr)
```

The PCD layer contains one function that must be called by the USB interrupt:

```
void HAL_PCD_IRQHandler(PCD_HandleTypeDef *hpcd)
```

The `stm32fxxx_hal_pcd.h` file contains the function prototypes called from the library core layer to handle the USB events.

## Important enumerated typedefs

- **USBD\_StatusTypeDef**

Almost all library functions return a status of type `USBD_StatusTypeDef`. The user application should always check the returned status.

```
typedef enum
{
    USBH_OK      = 0,
    USBH_BUSY,
    USBH_FAIL,
}USBH_StatusTypeDef;
```

[Table 2](#) describes the possible returned status:

**Table 2. USB device status**

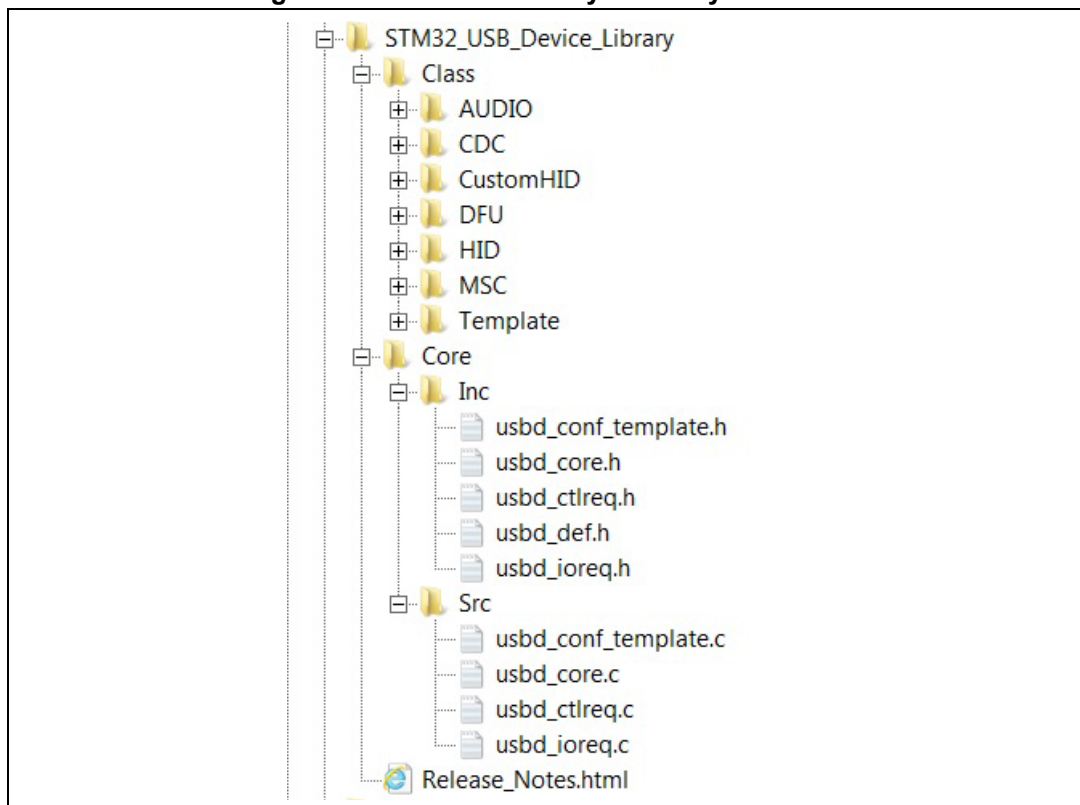
| Status    | Description   |
|-----------|---|
| USBH_OK   | Returned when operation is completed successfully.                            |
| USBH_BUSY | Returned when operation is still not completed (busy).                        |
| USBH_FAIL | Returned when operation has failed due to a low level error or protocol fail. |

## 5 USB device library overview

The USB device library is based on the generic USB low level driver. It has been developed to work in Full speed and High speed mode.

It implements the USB device library machines as defined by *Universal Serial Bus Specification* revision 2.0. The library functions are covered by the files located in the **Core** folder within the USB device library firmware package (see [Figure 5](#)). The USB class module is the class layer built in compliance with the protocol specification.

**Figure 5. USB device library directory structure**



### 5.1 USB device library description

#### 5.1.1 USB device library flow

##### Handling control endpoint

The USB specification defines four transfer types: control, interrupt, bulk and isochronous. The USB host sends requests to the device through the control endpoint (in this case, control endpoint is endpoint 0). The requests are sent to the device as SETUP packets. These requests can be classified into three categories: standard, class-specific and vendor-specific. Since the standard requests are generic and common to all USB devices, the library receives and handles all the standard requests on the control endpoint 0.

The format and the meaning of the class-specific requests and the vendor specific requests are not common for all USB devices.

All SETUP requests are processed with a state machine implemented in an interrupt model. An interrupt is generated at the end of the correct USB transfer. The library code receives this interrupt. In the interrupt process routine, the trigger endpoint is identified. If the event is a setup on endpoint 0, the payload of the received setup is saved and the state machine starts.

### Transactions on non-control endpoint

The class-specific core uses non-control endpoints by calling a set of functions to send or receive data through the data IN and OUT stage callbacks.

### Data structure for the SETUP packet

When a new SETUP packet arrives, all the eight bytes of the SETUP packet are copied to an internal structure *USB\_SETUP\_REQ req*, so that the next SETUP packet cannot overwrite the previous one during processing. This internal structure is defined as:

**Figure 6. Data structure for SETUP packet**

```
typedef struct usb_setup_req
{
    uint8_t    bmRequest;
    uint8_t    bRequest;
    uint16_t   wValue;
    uint16_t   wIndex;
    uint16_t   wLength;
}USBD_SetupReqTypeDef;
```

### Standard requests

Most of the requests specified in [Table 3](#) of the USB specification are handled as standard requests in the library. [Table 3](#) lists all the standard requests and their valid parameters in the library. Requests that are not in [Table 3](#) are considered as non-standard requests.

**Table 3. Standard requests**

|            | State | bmRequestType | Low byte of wValue | High byte of wValue | Low byte of wIndex | High byte of wIndex | wLength | Comments   |
|------------|-------|---------------|--------------------|---------------------|--------------------|---------------------|---------|--|
| GET_STATUS | A, C  | 80            | 00                 | 00                  | 00                 | 00                  | 2       | Gets the status of the Device.                                       |
|            | C     | 81            | 00                 | 00                  | N                  | 00                  | 2       | Gets the status of Interface, where N is the valid interface number. |
|            | A, C  | 82            | 00                 | 00                  | 00                 | 00                  | 2       | Gets the status of endpoint 0 OUT direction.                         |
|            | A, C  | 82            | 00                 | 00                  | 80                 | 00                  | 2       | Gets the status of endpoint 0 IN direction.                          |
|            | C     | 82            | 00                 | 00                  | EP                 | 00                  | 2       | Gets the status of endpoint EP.                                      |



Table 3. Standard requests (continued)

|                   | State | bmRequestType | Low byte of wValue | High byte of wValue | Low byte of wIndex | High byte of wIndex | wLength | Comments  |
|-------------------|-------|---------------|--------------------|---------------------|--------------------|---------------------|---------|---|
| CLEAR_FEATURE     | A, C  | 00            | 01                 | 00                  | 00                 | 00                  | 00      | Clears the device remote wakeup feature.  |
|                   | C     | 02            | 00                 | 00                  | EP                 | 00                  | 00      | Clears the STALL condition of endpoint EP. EP does not refer to endpoint 0.   |
| SET_FEATURE       | A, C  | 00            | 01                 | 00                  | 00                 | 00                  | 00      | Sets the device remote wakeup feature.  |
|                   | C     | 02            | 00                 | 00                  | EP                 | 00                  | 00      | Sets the STALL condition of endpoint EP. EP does not refer to endpoint 0.   |
| SET_ADDRESS       | D, A  | 00            | N                  | 00                  | 00                 | 00                  | 00      | Sets the device address, N is the valid device address.   |
| GET_DESCRIPTOR    | All   | 80            | 00                 | 01                  | 00                 | 00                  | Non-0   | Gets the device descriptor.   |
|                   | All   | 80            | N                  | 02                  | 00                 | 00                  | Non-0   | Gets the configuration descriptor; where N is the valid configuration index.  |
|                   | All   | 80            | N                  | 03                  | LangID             |                     | Non-0   | Gets the string descriptor; where N is the valid string index. This request is valid only when the string descriptor is supported.          |
| GET_CONFIGURATION | A, C  | 80            | 00                 | 00                  | 00                 | 00                  | 1       | Gets the device configuration.  |
| SET_CONFIGURATION | A, C  | 80            | N                  | 00                  | 00                 | 00                  | 00      | Sets the device configuration; where N is the valid configuration number.   |
| GET_INTERFACE     | C     | 81            | 00                 | 00                  | N                  | 00                  | 1       | Gets the alternate setting of the interface N; where N is the valid interface number.   |
| SET_INTERFACE     | C     | 01            | M                  | 00                  | N                  | 00                  | 00      | Sets alternate setting M of the interface N; where N is the valid interface number and M is the valid alternate setting of the interface N. |

**Note:** In column State: D = Default state; A = Address state; C = Configured state; All = All states.  
 EP: D0-D3 = endpoint address; D4-D6 = Reserved as zero; D7= 0: OUT endpoint, 1: IN endpoint.

## Non-standard requests

All the non-standard requests are passed to the class specific code through callback functions.

- **SETUP stage**

The library passes all the non-standard requests to the class-specific code with the callback `pdev->pClass->Setup(pdev, req)` function.

The non-standard requests include the user-interpreted requests and the invalid requests. User-interpreted requests are class-specific requests, vendor-specific requests or the requests that the library considers as invalid requests that the application wants to interpret as valid requests

Invalid requests are the requests that are not standard requests and are not user-interpreted requests. Since `pdev->pClass->Setup(pdev, req)` is called after the SETUP stage and before the data stage, user code is responsible, in the **`pdev->pClass->Setup(pdev, req)`** to parse the content of the SETUP packet (`req`). If a request is invalid, the user code has to call **`USBD_CtlError(pdev, req)`** and return to the caller of **`pdev->pClass->Setup(pdev, req)`**

For a user-interpreted request, the user code prepares the data buffer for the following data stage if the request has a data stage; otherwise the user code executes the request and returns to the caller of `pdev->pClass->Setup(pdev, req)`.

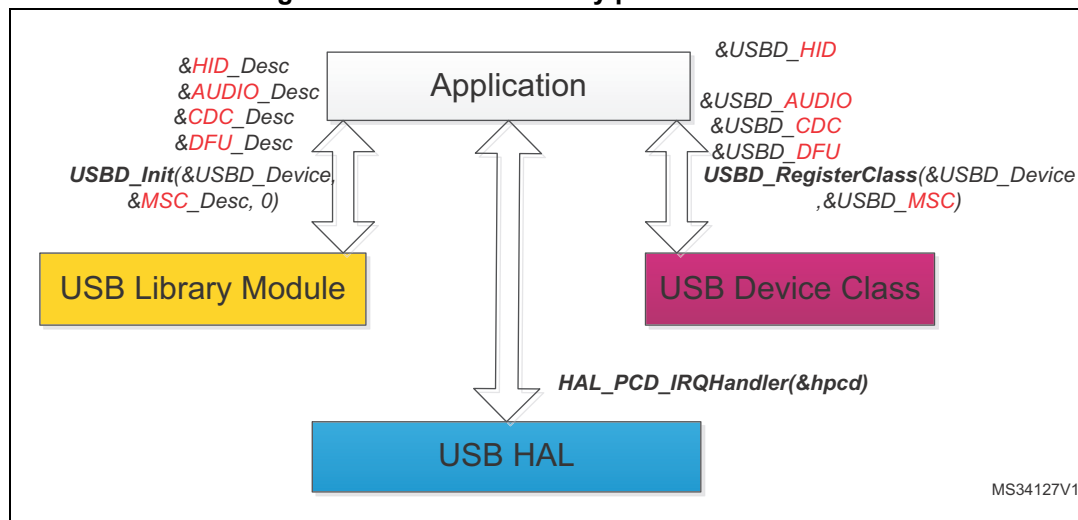
- **DATA stage**

The class layer uses the standard `USBD_CtlSendData` and `USBD_CtlPrepareRx` to send or receive data. The data transfer flow is handled internally by the library and the user does not need to split the data in `ep_size` packet.

- **Status stage**

The status stage is handled by the library after returning from the `pdev->pClass->Setup(pdev, req)` callback.

**Figure 7. USB device library process flowchart**



1. The red text identifies the USB device configuration.

As shown in the [Figure 7: USB device library process flowchart](#), only the following modules are necessary for USB programming: USB library, USB Device class and main application.

The main application executes the user defined program. **main.c**, **stm32fxx\_it.c**, **usbd\_conf.c** and **usbd\_desc.c** together with their header files are the main files (mandatory for the application) that the user needs to develop his own application. The user can modify them according to his application requirements (class driver).

Only simple APIs are called. They allow interfacing between the application layer and the USB library module which handles the USB initialization and getting the current status of the USB.

To initialize the USB HAL driver, the USB device library and the hardware on the used board (BSP) and to start the library, the user application must call these three APIs:

- **USBD\_Init ()**: This function initializes the device stack and loads the class driver.  
The device descriptor is stored in the *usbd\_desc.c* and *usbd\_desc.h* (used for the configuration descriptor type) files:
- **USBD\_RegisterClass()**: This function links the class driver to the device core.
- **USBD\_Start()**: This function allows user to start the USB device core

For example the user can add additional endpoints in the **usbd\_conf** file, depending on the class requirement. This is done by calling **USBD\_LL\_Init()** function. The *dev\_endpoints* should contain the number of required endpoints following the USB class specifications.

The USB device library provides several configurations thanks to the *usbd\_conf.h* file (for more details refer to [Section 5.1.5: Configuring the USB device firmware library on page 22](#)).

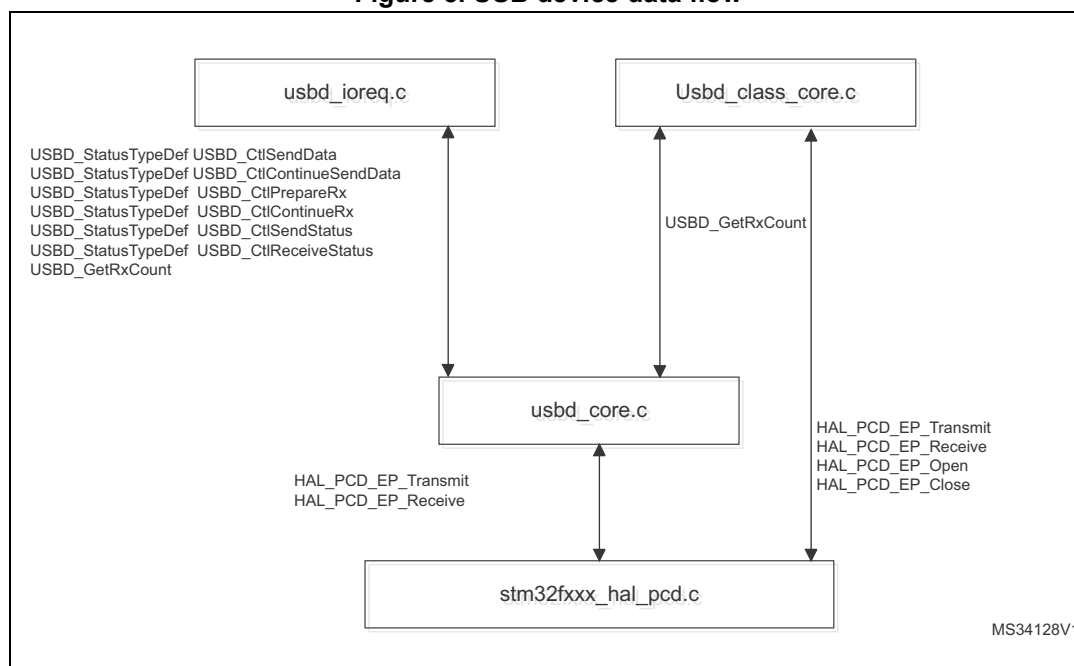
*Note: The HAL library initialization is done through the HAL\_Init() API in the stm32fxxx\_hal.c. This function performs the following operation:*

- Reset of all peripherals
- Configuration of Flash prefetch, Instruction cache, Data cache
- Enabling of SysTick and configuration of 1 ms tick (default clock after Reset is HSI)

### 5.1.2 USB device data flow

The USB library (USB core and USB class layer) handles the data processing on endpoint 0 (EP0) through the I/O request layer when a wrapping is needed to manage the multi-packet feature on the control endpoint, or directly from the *stm32fxxx\_hal\_pcd* layer when the other endpoints are used since the USB OTG core supports the multi-packet feature. [Figure 8](#) illustrates this data flow scheme.

Figure 8. USB device data flow



### 5.1.3 Core interface with low level driver

As mentioned before, the USB device library interfaces with the STM32Cube HAL low layer drivers using a low level interface layer which acts as a link layer with the STM32Cube HAL.

The low level interface implements low level API functions and calls some library core callback functions following some USB events.

In the STM32Cube package, the implementation of the low level interface is provided as part of the USB device examples since some parts of the low level interface are board and system dependent.

[Table 4](#) lists the low level API functions:

**Note:** *These APIs are provided by the USB Device Configuration file (usbd\_conf.c). They should be implemented in the user files and adapted to the USB Device Controller Driver.*

*The user can start from the usbd\_conf.c file provided within STM32Cube package. This file can also be copied to the application folder and modified depending on the application needs.*

Table 4. API description

| API             | Description                                  |
|-----------------|--|
| USBD_LL_Init    | Low level initialization.                    |
| USBD_LL_DeInit  | Low level de-initialization.                 |
| USBD_LL_Start   | Low level start.                             |
| USBD_LL_Stop    | Low level stop.                              |
| USBD_LL_OpenEP  | Initializes an endpoint.                     |
| USBD_LL_CloseEP | Closes and de-initializes an endpoint state. |

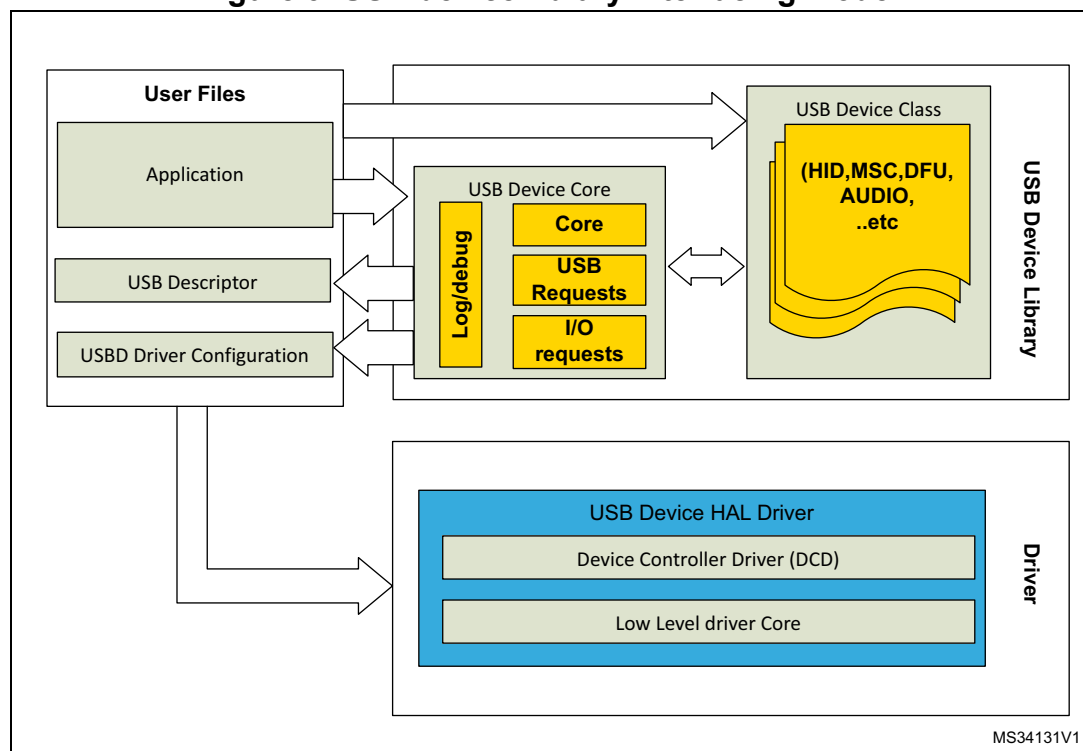
Table 4. API description (continued)

| API                    | Description  |
|------------------------|--|
| USBD_LL_FlushEP        | Flushes an endpoint of the Low Level Driver.                     |
| USBD_LL_StallEP        | Sets a Stall condition on an endpoint of the Low Level Driver.   |
| USBD_LL_ClearStallEP   | Clears a Stall condition on an endpoint of the Low Level Driver. |
| USBD_LL_IsStallEP      | Returns Stall condition.   |
| USBD_LL_SetUSBAddress  | Assigns an USB address to the device.                            |
| USBD_LL_Transmit       | Transmits data over an endpoint.                                 |
| USBD_LL_PrepareReceive | Prepares an endpoint for reception.                              |
| USBD_LL_GetRxDataSize  | Returns the last transferred packet size.                        |

#### 5.1.4 USB device library interfacing model

The USB device library is built around central generic and portable class modules.

Figure 9. USB device library interfacing model



[Table 5](#) shows all the device library callback functions which are called from the low level interface following some USB events.

**Table 5. Low level Event Callback functions**

| Callback functions               | Description                   |
|----------------------------------|-------------------------------|
| HAL_PCD_ConnectCallback          | Device connection Callback.   |
| HAL_PCD_DataInStageCallback      | Data IN stage Callback.       |
| HAL_PCD_DataOutStageCallback     | Data OUT stage Callback.      |
| HAL_PCD_DisconnectCallback       | Disconnection Callback.       |
| HAL_PCD_ISOINIncompleteCallback  | ISO IN transaction Callback.  |
| HAL_PCD_ISOOUTIncompleteCallback | ISO OUT transaction Callback. |
| HAL_PCD_ResetCallback            | USB Reset Callback.           |
| HAL_PCD_ResumeCallback           | USB Resume Callback.          |
| HAL_PCD_SetupStageCallback       | Setup stage Callback.         |
| HAL_PCD_SOFCallback              | Start Of Frame callback.      |
| HAL_PCD_SuspendCallback          | Suspend Callback.             |

### 5.1.5 Configuring the USB device firmware library

The USB device library can be configured using the *usbd\_conf.h* file.

The **usbd\_conf.h** is a specific configuration file used to define some global parameters and specific configurations. This file is used to link the upper library with the HAL drivers and the BSP drivers.

**Table 6. USB library configuration**

| item                              | Parameter                  | Description   |
|-----------------------------------|----------------------------|---|
| <b>Common configuration</b>       | USBD_MAX_NUM_CONFIGURATION | Maximum number of supported configurations [1 to 255].          |
|                                   | USBD_MAX_NUM_INTERFACES    | Maximum number of supported Interfaces [1 to 255].              |
|                                   | USBD_MAX_STR_DESC_SIZ      | Maximum size of string descriptors [uint16].                    |
|                                   | USBD_SELF_POWERED          | Enables self power feature [0/1].                               |
|                                   | USBD_DEBUG_LEVEL           | Debug and log level.  |
|                                   | USBD_SUPPORT_USER_STRING   | Enables user string support[0/1].                               |
| <b>Mass Storage configuration</b> | MSC_MEDIA_PACKET           | Media I/O buffer size multiple of 512 bytes [512 to 32 Kbytes]. |
| <b>HID Configuration</b>          | NA                         | NA.   |

Table 6. USB library configuration (continued)

| item                | Parameter                | Description   |
|---------------------|--------------------------|---|
| DFU Configuration   | USBD_DFU_MAX_ITF_NUM     | Maximum media interface number [1 to 255].                      |
|                     | USBD_DFU_XFER_SIZE       | Media I/O buffer size multiple of 512 bytes [512 to 32 Kbytes]. |
|                     | USBD_DFU_APP_DEFAULT_ADD | Application address (0x0800C000).                               |
| CDC Configuration   | NA                       | NA.   |
| Audio Configuration | USBD_AUDIO_FREQ          | 8 to 48 KHz.  |

**Note:** The user can start from the `usbd_conf.c` file provided within STM32Cube package. This file could be also copied to the application folder and modified depending on the application needs.

By default for USB device examples, library and user messages **are not displayed** on the LCD.

However the user can implement his own messages. To redirect the library messages on the LCD screen, `lcd_log.c` driver must be added to the application sources. He can choose to display them or not by modifying define values in the `usbd_conf.h` configuration file available under the project includes directory. For example:

0: No Log/Debug messages

1: log messages enabled

2: log and debug messages enabled

## 5.1.6 USB control functions

### Device reset

When the device receives a reset signal from the USB, the library resets and initializes both application software and hardware. This function is part of the interrupt routine.

### Device suspend

When the device detects a suspend condition on the USB, the library stops all the ongoing operations and puts the system in suspend state (if low power mode management is enabled in the `usbd_conf.c` file).

### Device resume

When the device detects a resume signal on the USB, the library restores the USB core clock and puts the system in idle state (if low power mode management is enabled in the `usbd_conf.c` file).

## 5.2 USB device library functions

The **Core** folder contains the USB device library machines as defined by the Universal Serial Bus Specification, revision 2.0.

**Table 7. USB device core files**

| File                      | Description  |
|---------------------------|--|
| usbd_core (.c, .h)        | This file contains the functions for handling all USB communication and state machine.                           |
| usbd_req(.c,.h)           | This file includes the requests implementation listed in Chapter 9 of the specification.                         |
| usbd_ctlreq(.c,.h)        | This file handles the results of the USB transactions.   |
| usbd_conf_template(.c,.h) | Template file for the low layer interface file, should be customized by user and included with application file. |
| usbd_def(.c, .h)          | Common library defines.  |

The **Class** folder contains all the files relative to the class implementation and complies with the specification of the protocol built in these classes.

**Table 8. Class drivers files**

| USB class          | file                  | Description                          |
|--------------------|-----------------------|--------------------------------------|
| Mass-Storage       | usbh_msc (.c,.h)      | Mass-storage class handler.          |
|                    | usbh_msc_bot(.c,.h)   | Bulk-only transfer protocol handler. |
|                    | usbh_msc_scsi(.c,.h)  | SCSI commands.                       |
|                    | usbd_msc_data (.c,.h) | Vital inquiry pages and sense data.  |
| HID Joystick mouse | usbh_hid(.c,.h)       | HID class state handler.             |
| Audio speaker      | usbh_audio(.c,.h)     | Audio class handler.                 |
| Audio speaker      | usbh_cdc(.c,.h)       | CDC virtual comport handler.         |
| Custom HID         | usbd_customhid(.c,.h) | Custom HID Class Handler.            |
| DFU Class          | usbd_dfu(.c,.h)       | DFU class handler.                   |

**Table 9. usbd\_core (.c,.h) files**

| Functions  | Description  |
|--|--|
| USBD_StatusTypeDef<br>USBD_Init(USBD_HandleTypeDef *pdev,<br>USBD_DescriptorsTypeDef *pdesc, uint8_t id) | Initializes the device library and loads the class driver and the user call backs. |
| USBD_StatusTypeDef<br>USBD_DeInit(USBD_HandleTypeDef *pdev)  | De-Initializes the device library.   |
| USBD_StatusTypeDef<br>USBD_RegisterClass(USBD_HandleTypeDef *pdev,<br>USBD_ClassTypeDef *pclass)         | Loads the class driver.  |



**Table 9. usbd\_core (.c,.h) files (continued)**

| Functions   | Description   |
|---|---|
| USBD_StatusTypeDef USBD_Start<br>(USBD_HandleTypeDef *pdev)   | Starts the device library process.                            |
| USBD_StatusTypeDef USBD_Stop<br>(USBD_HandleTypeDef *pdev)  | Stops the device library process and frees related resources. |
| USBD_StatusTypeDef<br>USBD_LL_SetupStage(USBD_HandleTypeDef *pdev,<br>uint8_t *psetup)                  | Handles setup stage from ISR.                                 |
| USBD_StatusTypeDef<br>USBD_LL_DataOutStage(USBD_HandleTypeDef *pdev ,<br>uint8_t epnum, uint8_t *pdata) | Handles data out stage from ISR.                              |
| USBD_StatusTypeDef<br>USBD_LL_DataInStage(USBD_HandleTypeDef *pdev<br>,uint8_t epnum, uint8_t *pdata)   | Handles data IN stage.  |
| USBD_StatusTypeDef<br>USBD_LL_Reset(USBD_HandleTypeDef *pdev)   | Handles USB Reset from ISR.                                   |
| USBD_StatusTypeDef<br>USBD_LL_SetSpeed(USBD_HandleTypeDef *pdev,<br>USBD_SpeedTypeDef speed)            | Sets USB Core Speed   |
| USBD_StatusTypeDef<br>USBD_LL_Suspend(USBD_HandleTypeDef *pdev)   | Handles Suspend Event.  |
| USBD_StatusTypeDef<br>USBD_LL_Resume(USBD_HandleTypeDef *pdev)  | Handles Resume event.   |
| USBD_StatusTypeDef<br>USBD_LL_SOFTWARE(USBD_HandleTypeDef *pdev);                                       | Handles Start Of Frame Event.                                 |
| USBD_StatusTypeDef<br>USBD_LL_IsoINIncomplete(USBD_HandleTypeDef<br>*pdev, uint8_t epnum)               | Handles Incomplete ISO IN transaction Event.                  |
| USBD_StatusTypeDef<br>USBD_LL_IsoOUTIncomplete(USBD_HandleTypeDef<br>*pdev, uint8_t epnum)              | Handles Incomplete ISO OUT transaction Event.                 |
| USBD_StatusTypeDef<br>USBD_LL_DevConnected(USBD_HandleTypeDef *pdev)                                    | Notifies about device connection from ISR.                    |
| USBD_StatusTypeDef<br>USBD_LL_DevDisconnected(USBD_HandleTypeDef<br>*pdev)                              | Notifies about device disconnection from ISR.                 |

**Table 10. usbd\_ioreq (.c,.h) files functions**

| Functions  | Description                         |
|--|-------------------------------------|
| USBD_StatusTypeDef<br>USBD_CtlSendData (USBD_HandleTypeDef *pdev,<br>uint8_t *pbuf,uint16_t len) | Sends the data on the control pipe. |

**Table 10. usbd\_ioreq (.c,.h) files functions (continued)**

| Functions  | Description  |
|--|--|
| USBD_StatusTypeDef<br>USBD_CtlContinueSendData (USBD_HandleTypeDef *pdev, uint8_t *pbuf, uint16_t len) | Continues sending data on the control pipe.            |
| USBD_StatusTypeDef<br>USBD_CtlPrepareRx (USBD_HandleTypeDef *pdev, uint8_t *pbuf, uint16_t len)        | Prepares the core to receive data on the control pipe. |
| USBD_StatusTypeDef USBD_CtlContinueRx<br>(USBD_HandleTypeDef *pdev, uint8_t *pbuf, uint16_t len)       | Continues receiving data on the control pipe.          |
| USBD_StatusTypeDef<br>USBD_CtlSendStatus (USBD_HandleTypeDef *pdev)                                    | Sends a zero length packet on the control pipe.        |
| USBD_StatusTypeDef USBD_CtlReceiveStatus<br>(USBD_HandleTypeDef *pdev)                                 | Receives a zero length packet on the control pipe.     |
| uint16_t USBD_GetRxCount (USBD_HandleTypeDef *pdev, uint8_t ep_addr)                                   | Returns the received data length.                      |

**Table 11. usbd\_ctrlq (.c,.h) files functions**

| Functions  | Description                               |
|--|---|
| USBD_StatusTypeDef<br>USBD_StdDevReq (USBD_HandleTypeDef *pdev, USBD_SetupReqTypeDef *req) | Handles standard USB device requests.     |
| USBD_StatusTypeDef<br>USBD_StdItfReq (USBD_HandleTypeDef *pdev, USBD_SetupReqTypeDef *req) | Handles standard USB interface requests.  |
| USBD_StatusTypeDef<br>USBD_StdEPReq (USBD_HandleTypeDef *pdev, USBD_SetupReqTypeDef *req)  | Handles standard USB endpoint requests.   |
| static void USBD_GetDescriptor(USBD_HandleTypeDef *pdev, USBD_SetupReqTypeDef *req)        | Handles Get Descriptor requests.          |
| static void USBD_SetAddress(USBD_HandleTypeDef *pdev, USBD_SetupReqTypeDef *req)           | Sets new USB device address.              |
| static void USBD_SetConfig(USBD_HandleTypeDef *pdev, USBD_SetupReqTypeDef *req)            | Handles Set device configuration request. |
| static void USBD_GetConfig(USBD_HandleTypeDef *pdev, USBD_SetupReqTypeDef *req)            | Handles Get device configuration request. |
| static void USBD_GetStatus(USBD_HandleTypeDef *pdev, USBD_SetupReqTypeDef *req)            | Handles Get Status request.               |
| static void USBD_SetFeature(USBD_HandleTypeDef *pdev, USBD_SetupReqTypeDef *req)           | Handles Set device feature request.       |
| static void USBD_ClrFeature(USBD_HandleTypeDef *pdev, USBD_SetupReqTypeDef *req)           | Handles Clear device feature request.     |
| void USBD_CtlError(USBD_HandleTypeDef *pdev, USBD_SetupReqTypeDef *req)                    | Handles USB Errors on the control pipe.   |

Table 11. usbd\_ctrlq (.c,.h) files functions (continued)

| Functions  | Description                                 |
|--|---|
| void USBD_GetString(uint8_t *desc, uint8_t *unicode, uint16_t *len)        | Converts ASCII string into unicode one.     |
| static uint8_t USBD_GetLen(uint8_t *buf)                                   | Returns the string length.                  |
| void USBD_ParseSetupRequest<br>(USBD_SetupReqTypeDef *req, uint8_t *pdata) | Copies request buffer into setup structure. |

## 5.3 USB device class interface

### USB Class callback structure

The USB class is chosen during the USB device library initialization by selecting the corresponding class callback structure. The class structure is defined as follows:

Figure 10. USB Class callback structure

```
typedef struct _Device_cb
{
    uint8_t  (*Init)                (struct _USBD_HandleTypeDef *pdev ,
    uint8_t cfgidx);
    uint8_t  (*DeInit)              (struct _USBD_HandleTypeDef *pdev ,
    uint8_t cfgidx);
    /* Control Endpoints*/
    uint8_t  (*Setup)                (struct _USBD_HandleTypeDef *pdev ,
    USBD_SetupReqTypeDef *req);
    uint8_t  (*EP0_TxSent)           (struct _USBD_HandleTypeDef *pdev );
    uint8_t  (*EP0_RxReady)          (struct _USBD_HandleTypeDef *pdev );
    /* Class Specific Endpoints*/
    uint8_t  (*DataIn)               (struct _USBD_HandleTypeDef *pdev ,
    uint8_t epnum);
    uint8_t  (*DataOut)              (struct _USBD_HandleTypeDef *pdev ,
    uint8_t epnum);
    uint8_t  (*SOF)                  (struct _USBD_HandleTypeDef *pdev);
    uint8_t  (*IsoINIncomplete)      (struct _USBD_HandleTypeDef *pdev ,
    uint8_t epnum);
    uint8_t  (*IsoOUTIncomplete)     (struct _USBD_HandleTypeDef *pdev ,
    uint8_t epnum);
    uint8_t  (*GetHSConfigDescriptor)(uint16_t *length);
    uint8_t  (*GetFSConfigDescriptor)(uint16_t *length);
    uint8_t  (*GetOtherSpeedConfigDescriptor)(uint16_t *length);
    uint8_t  (*GetDeviceQualifierDescriptor)(uint16_t *length);
} USBD_ClassTypeDef;
```

- **Init:** this callback is called when the device receives the set configuration request; in this function the endpoints used by the class interface are open.
- **DeInit:** This callback is called when the clear configuration request has been received; this function closes the endpoints used by the class interface.

- **Setup**: This callback is called to handle the specific class setup requests.
- **EP0\_TxSent**: This callback is called when the send status is finished.
- **EP0\_RxSent**: This callback is called when the receive status is finished.
- **DataIn**: This callback is called to perform the data in stage relative to the non-control endpoints.
- **DataOut**: This callback is called to perform the data out stage relative to the non-control endpoints.
- **SOF**: This callback is called when a SOF interrupt is received; this callback can be used to synchronize some processes with the SOF.
- **IsolInIncomplete**: This callback is called when the last isochronous IN transfer is incomplete.
- **IsoOUTIncomplete**: This callback is called when the last isochronous OUT transfer is incomplete.
- **GetHSConfigDescriptor**: This callback returns the HS USB Configuration descriptor.
- **GetFSConfigDescriptor**: This callback returns the FS USB Configuration descriptor.
- **GetOtherSpeedConfigDescriptor**: This callback returns the other configuration descriptor of the used class in High Speed mode.
- **GetDeviceQualifierDescriptor**: This callback returns the Device Qualifier Descriptor.

### USB device descriptors structure

The library provides also descriptor callback structures that allow user to manage the device and string descriptors at application run time. This descriptors structure is defined as follows:

**Figure 11. USB device descriptors structure**

```
typedef struct
{
    uint8_t  (*GetDeviceDescriptor)( USBD_SpeedTypeDef speed ,
    uint16_t *length);
    uint8_t  (*GetLangIDStrDescriptor)( USBD_SpeedTypeDef speed ,
    uint16_t *length);
    uint8_t  (*GetManufacturerStrDescriptor)( USBD_SpeedTypeDef speed
    , uint16_t *length);
    uint8_t  (*GetProductStrDescriptor)( USBD_SpeedTypeDef speed ,
    uint16_t *length);
    uint8_t  (*GetSerialStrDescriptor)( USBD_SpeedTypeDef speed ,
    uint16_t *length);
    uint8_t  (*GetConfigurationStrDescriptor)( USBD_SpeedTypeDef speed
    , uint16_t *length);
    uint8_t  (*GetInterfaceStrDescriptor)( USBD_SpeedTypeDef speed ,
    uint16_t *length);
} USBD_DescriptorsTypeDef;
```

- **GetDeviceDescriptor**: This callback returns the device descriptor.
- **GetLangIDStrDescriptor**: This callback returns the Language ID string descriptor.

- ***GetManufacturerStrDescriptor***: This callback returns the manufacturer string descriptor.
- ***GetProductStrDescriptor***: This callback returns the product string descriptor.
- ***GetSerialStrDescriptor***: This callback returns the serial number string descriptor.
- ***GetConfigurationStrDescriptor***: This callback returns the configuration string descriptor.
- ***GetInterfaceStrDescriptor***: This callback returns the interface string descriptor.

*Note:* The `usbd_desc.c` file provided within USB Device examples implements these callback bodies.

## 6 USB device library class module

The class module contains all the files related to the class implementation. It complies with the specification of the protocol built in these classes. [Table 12](#) shows the USB device class files for the MSC, HID, DFU, Audio, CDC classes.

**Table 12. USB device class files**

| Class             | Files                              | Description  |
|-------------------|------------------------------------|--|
| <b>HID</b>        | usbd_hid (.c, .h)                  | This file contains the HID class callbacks (driver) and the configuration descriptors relative to this class.        |
| <b>MSC</b>        | usbd_msc( .c, .h)                  | This file contains the MSC class callbacks (driver) and the configuration descriptors relative to this class.        |
|                   | usbd_bot (.c, .h)                  | This file handles the bulk only transfer protocol.   |
|                   | usbd_scsi (.c, .h)                 | This file handles the SCSI commands.   |
|                   | usbd_msc_data (.c,.h)              | This file contains the vital inquiry pages and the sense data of the mass storage devices.                           |
|                   | usbd_msc_storage_template (.c,.h)  | This file provides a template driver which allows you to implement additional functions for MSC.                     |
| <b>DFU</b>        | usbd_dfu (.c,.h)                   | This file contains the DFU class callbacks (driver) and the configuration descriptors relative to this class.        |
|                   | usbd_dfu_media_template_if (.c,.h) | This file provides a template driver which allows you to implement additional memory interfaces.                     |
| <b>Audio</b>      | usbd_audio (.c,.h)                 | This file contains the AUDIO class callbacks (driver) and the configuration descriptors relative to this class.      |
|                   | usbd_audio_if_template (.c,.h)     | This file provides a template driver which allows you to implement additional functions for Audio.                   |
| <b>CDC</b>        | usbd_cdc (.c,.h)                   | This file contains the CDC class callbacks (driver) and the configuration descriptors relative to this class.        |
|                   | usbd_cdc_if_template (.c,.h)       | This file provides a template driver which allows you to implement low layer functions for a CDC terminal.           |
| <b>Custom HID</b> | usbd_customhid (.c,.h)             | This file contains the Custom HID class callbacks (driver) and the configuration descriptors relative to this class. |

## 6.1 HID class

### 6.1.1 HID class implementation

This module manages the HID class V1.11 following the “Device Class Definition for Human Interface Devices (HID) Version 1.11 June 27, 2001”.

The HID specification can be found searching for “hidpage” on [www.st.com](http://www.st.com).

This driver implements the following aspects of the specification:

- The boot interface subclass
- The mouse protocol
- Usage page: generic desktop
- Usage: joystick
- Collection: application

### 6.1.2 HID user interface

**Input** reports are sent only via the **Interrupt In** pipe (HID mouse example).

Feature and Output reports must be initiated by the host via Control pipe or an Interrupt Out pipe (Custom HID example)

The `USBD_HID_SendReport` can be used by the HID mouse application to send HID reports, the HID driver, in this release, handles only IN traffic. An example of use of this function is shown below:

**Figure 12. Example of USBD\_HID\_SendReport function**

```
static __IO uint32_t counter=0;
HAL_IncTick();
/* check Joystick state every 10ms */
if (counter++ == 10)
{
    GetPointerData(HID_Buffer);
    /* send data though IN endpoint*/
    if((HID_Buffer[1] != 0) || (HID_Buffer[2] != 0))
    {
        USBD_HID_SendReport(&USBD_Device, HID_Buffer, 4);
    }
    counter =0;
}
Toggle_Leds();
}
```

### 6.1.3 HID Class Driver APIs

All HID class driver APIs are defined in `usbd_hid.c` and summarized in the table below.

**Table 13. `usbd_hid.c,h` files**

| Functions   | Description  |
|---|--|
| <code>static uint8_t USBD_HID_Init</code><br>( <code>USBD_HandleTypeDef *pdev</code> , <code>uint8_t cfgidx</code> )                                | Initializes the HID interface and open the used endpoints. |
| <code>static uint8_t USBD_HID_DeInit</code><br>( <code>USBD_HandleTypeDef *pdev</code> , <code>uint8_t cfgidx</code> )                              | Un-Initializes the HID layer and close the used endpoints. |
| <code>static uint8_t USBD_HID_Setup</code><br>( <code>USBD_HandleTypeDef *pdev</code> ,<br><code>USBD_SetupReqTypeDef *req</code> )                 | Handles the HID specific requests.                         |
| <code>uint8_t USBD_HID_SendReport</code><br>( <code>USBD_HandleTypeDef *pdev</code> , <code>uint8_t *report</code> ,<br><code>uint16_t len</code> ) | Sends HID reports.   |

The HID stack is initialized by calling the `USBD_HID_Init()`, Then the application has to call the `USBD_HID_SendReport()` function to send the HID reports.

The Following HID specific requests are implemented through the endpoint 0 (Control):

```
#define HID_REQ_SET_PROTOCOL      0x0B
#define HID_REQ_GET_PROTOCOL      0x03
#define HID_REQ_SET_IDLE         0x0A
#define HID_REQ_GET_IDLE         0x02
#define HID_REQ_SET_REPORT       0x09
#define HID_REQ_GET_REPORT       0x01
```

The IN endpoint address and the maximum number of bytes that can be sent are given by these defines:

```
#define HID_EPIN_ADDR            0x81
#define HID_EPIN_SIZE            0x04
```

## 6.2 Mass storage class

### 6.2.1 Mass storage class implementation

This module manages the MSC class V1.0 following the "Universal Serial Bus Mass Storage Class (MSC) Bulk-Only Transport (BOT) Version 1.0 Sep. 31, 1999".

This driver implements the following aspects of the specification:

- Bulk-only transport protocol
- Subclass: SCSI transparent command set (ref. SCSI Primary Commands - 3)

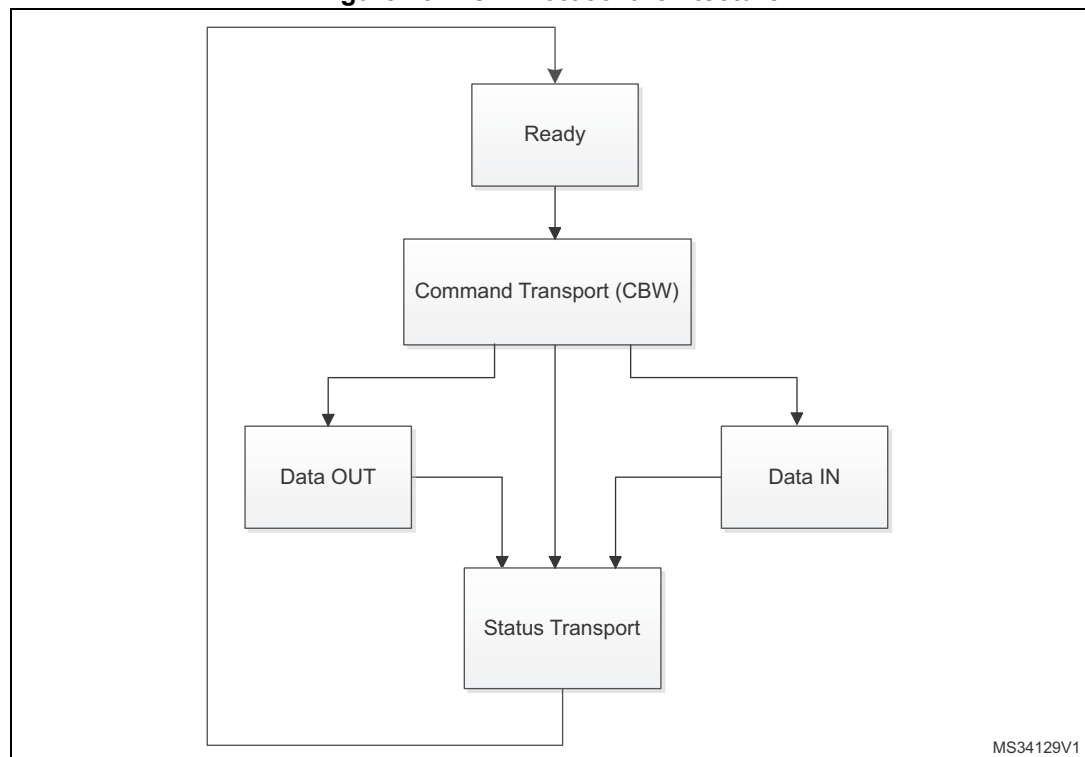
The USB mass storage class is built around the Bulk Only Transfer (BOT). It uses the SCSI transparent command set.



A general BOT transaction is based on a simple basic state machine. It begins in ready state (idle state). If a CBW is received from the host, three cases can be managed:

- **DATA-OUT-STAGE:** when the direction flag is set to “0”, the device must be prepared to receive an amount of data indicated in `cbw.dDataLength` in the CBW block. At the end of data transfer, a CSW is returned with the remaining data length and the STATUS field.
- **DATA-IN-STAGE:** when direction flag is set to “1”, the device must be prepared to send an amount of data indicated in `cbw.dDataLength` in the CBW block. At the end of data transfer, a CSW is returned with the remaining data length and the STATUS field.
- **ZERO DATA:** in this case, no data stage is required and the CSW block is sent immediately after the CBW one.

**Figure 13. BOT Protocol architecture**



The following table shows the supported SCSI commands.

**Table 14. SCSI commands**

| Command specification | Command   | Remark  |
|-----------------------|---|---|
| SCSI                  | SCSI_PREVENT_REMOVAL,<br>SCSI_START_STOP_UNIT,<br>SCSI_TEST_UNIT_READY,<br>SCSI_INQUIRY,<br>SCSI_READ_CAPACITY10,<br>SCSI_READ_FORMAT_CAPACITY,<br>SCSI_MODE_SENSE6,<br>SCSI_MODE_SENSE10<br>SCSI_READ10,<br>SCSI_WRITE10,<br>SCSI_VERIFY10 | READ_FORMAT_CAPACITY<br>(0x23) is an UFI command. |

As required by the BOT specification, the **Bulk-only mass storage reset** request (class-specific request) is implemented. This request is used to reset the mass storage device and its associated interface. This class-specific request should prepare the device for the next CBW from the host.

To generate the BOT Mass Storage Reset, the host must send a device request on the default pipe of:

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

### 6.2.2 Get Max MUN (class-specific request)

The device can implement several logical units that share common device characteristics. The host uses `bCBWLUN` to indicate 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 generate a Get Max LUN device request, the host sends a device request on the default pipe of:

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

### 6.2.3 MSC Core files

**Table 15. usbd\_msc (.c,.h) files**

| Functions  | Description   |
|--|---|
| uint8_t USBD_MSC_Init (USB_D_HandleTypeDef *pdev, uint8_t cfgidx)            | Initializes the MSC interface and opens the used endpoints. |
| uint8_t USBD_MSC_DeInit (USB_D_HandleTypeDef *pdev, uint8_t cfgidx)          | De-initializes the MSC layer and close the used endpoints.  |
| uint8_t USBD_MSC_Setup (USB_D_HandleTypeDef *pdev, USB_SetupReqTypeDef *req) | Handles the MSC specific requests.                          |
| uint8_t USBD_MSC_DataIn (USB_D_HandleTypeDef *pdev, uint8_t epnum)           | Handles the MSC Data In stage.                              |
| uint8_t USBD_MSC_DataOut (USB_D_HandleTypeDef *pdev, uint8_t epnum)          | Handles the MSC Data Out stage.                             |

**Table 16. usbd\_msc\_bot (.c,.h) files**

| Functions  | Description   |
|--|---|
| void MSC_BOT_Init (USB_D_HandleTypeDef *pdev)  | Initializes the BOT process and physical media.                     |
| void MSC_BOT_Reset (USB_D_HandleTypeDef *pdev)                                       | Resets the BOT Machine.   |
| void MSC_BOT_DeInit (USB_D_HandleTypeDef *pdev)                                      | De-Initializes the BOT process.                                     |
| void MSC_BOT_DataIn (USB_D_HandleTypeDef *pdev, uint8_t epnum)                       | Handles the BOT data IN Stage.                                      |
| void MSC_BOT_DataOut (USB_D_HandleTypeDef *pdev, uint8_t epnum)                      | Handles the BOT data OUT Stage.                                     |
| static void MSC_BOT_CBW_Decode (USB_D_HandleTypeDef *pdev)                           | Decodes the CBW command and sets the BOT state machine accordingly. |
| static void MSC_BOT_SendData (USB_D_HandleTypeDef *pdev, uint8_t* buf, uint16_t len) | Sends the requested data.   |
| void MSC_BOT_SendCSW (USB_D_HandleTypeDef *pdev, uint8_t CSW_Status)                 | Sends the Command Status Wrapper.                                   |
| static void MSC_BOT_Abort (USB_D_HandleTypeDef *pdev)                                | Aborts the current transfer.  |
| void MSC_BOT_CplClrFeature (USB_D_HandleTypeDef *pdev, uint8_t epnum)                | Completes the Clear Feature request.                                |

Table 17. usbd\_msc\_scsi (.c,.h)

| Functions  | Description                                    |
|--|--|
| int8_t SCSI_ProcessCmd(USBD_HandleTypeDef *pdev, uint8_t lun, uint8_t *params)                                     | Processes the SCSI commands.                   |
| static int8_t SCSI_TestUnitReady(USBD_HandleTypeDef *pdev, uint8_t lun, uint8_t *params)                           | Processes the SCSI Test Unit Ready command.    |
| static int8_t SCSI_Inquiry(USBD_HandleTypeDef *pdev, uint8_t lun, uint8_t *params)                                 | Processes the Inquiry command.                 |
| static int8_t SCSI_ReadCapacity10(USBD_HandleTypeDef *pdev, uint8_t lun, uint8_t *params)                          | Processes the Read Capacity 10 command.        |
| static int8_t SCSI_ReadFormatCapacity(USBD_HandleTypeDef *pdev, uint8_t lun, uint8_t *params)                      | Processes the Read Format Capacity command.    |
| static int8_t SCSI_ModeSense6(USBD_HandleTypeDef *pdev, uint8_t lun, uint8_t *params)                              | Processes the Mode Sense 6 command.            |
| static int8_t SCSI_ModeSense10(USBD_HandleTypeDef *pdev, uint8_t lun, uint8_t *params)                             | Processes the Mode Sense 10 command.           |
| static int8_t SCSI_RequestSense(USBD_HandleTypeDef *pdev, uint8_t lun, uint8_t *params)                            | Processes the Request Sense command.           |
| void SCSI_SenseCode(USBD_HandleTypeDef *pdev, uint8_t lun, uint8_t sKey, uint8_t ASC)                              | Loads the last error code in the error list.   |
| static int8_t SCSI_StartStopUnit(USBD_HandleTypeDef *pdev, uint8_t lun, uint8_t *params)                           | Processes the Start Stop Unit command.         |
| static int8_t SCSI_Read10(USBD_HandleTypeDef *pdev, uint8_t lun, uint8_t *params)                                  | Processes the Read10 command.                  |
| static int8_t SCSI_Write10(USBD_HandleTypeDef *pdev, uint8_t lun, uint8_t *params)                                 | Processes the Write10 command.                 |
| static int8_t SCSI_Verify10(USBD_HandleTypeDef *pdev, uint8_t lun, uint8_t *params)                                | Processes the Verify10 command.                |
| static int8_t SCSI_CheckAddressRange(USBD_HandleTypeDef *pdev, uint8_t lun, uint32_t blk_offset, uint16_t blk_nbr) | Checks if the LBA is inside the address range. |
| static int8_t SCSI_ProcessRead(USBD_HandleTypeDef *pdev, uint8_t lun)  | Handles the Burst Read process.                |
| static int8_t SCSI_ProcessWrite(USBD_HandleTypeDef *pdev, uint8_t lun)   | Handles the Burst Write process.               |

## 6.2.4 Disk operation structure definition

**Figure 14. Disk operation structure description**

```

USB_DiskTypeDef USBD_DISK_fops = {
    STORAGE_Init,
    STORAGE_GetCapacity,
    STORAGE_IsReady,
    STORAGE_IsWriteProtected,
    STORAGE_Read,
    STORAGE_Write,
    STORAGE_GetMaxLun,
    STORAGE_Inquirydata,
};

```

**Note:** *MicroSD is the default media interface provided by the library. However you can add other media (Flash memory....) using the template file provided in `usbd_msc_storage_template.c`*

The storage callback for MSC class is added in the user application by calling `USB_Disk_RegisterStorage(&USB_Disk_Device, &USB_Disk_fops)`.

The standard inquiry data are given by the user inside the `STORAGE_Inquirydata` array. They should be defined as shown in [Figure 15](#).

**Figure 15. Example of standard inquiry definition**

```

int8_t STORAGE_Inquirydata[] = { /* 36 */
    /* LUN 0 */
    0x00,
    0x80,
    0x02,
    0x02,
    (STANDARD_INQUIRY_DATA_LEN - 5),
    0x00,
    0x00,
    0x00,
    'S', 'T', 'M', ' ', ' ', ' ', ' ', ' ', /* Manufacturer: 8 bytes */
    'P', 'r', 'o', 'd', 'u', 'c', 't', ' ', /* Product : 16 Bytes */
    ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
    '0', '.', '0', '1', /* Version : 4 Bytes */
};

```

## 6.2.5 Disk operation functions

**Table 18. Disk operation functions**

| Functions   | Description  |
|---|--|
| int8_t STORAGE_Init (uint8_t lun)   | Initializes the storage medium.  |
| int8_t STORAGE_GetCapacity (uint8_t lun, uint32_t *block_num, uint16_t *block_size)   | Returns the medium capacity and block size.  |
| int8_t STORAGE_IsReady (uint8_t lun)  | Checks whether the medium is ready.  |
| int8_t STORAGE_IsWriteProtected (uint8_t lun)   | Checks whether the medium is write-protected.  |
| int8_t STORAGE_Read (uint8_t lun, uint8_t *buf, uint32_t blk_addr, uint16_t blk_len)  | Reads data from the medium:<br>blk_address is given in sector unit<br>blk_len is the number of the sector to be processed. |
| int8_t STORAGE_Write (uint8_t lun, uint8_t *buf, uint32_t blk_addr, uint16_t blk_len) | Writes data to the medium:<br>blk_address is given in sector unit<br>blk_len is the number of the sector to be processed.  |
| int8_t STORAGE_GetMaxLun (void)   | Returns the number of supported logical units.   |

## 6.3 Device firmware upgrade (DFU) class

The DFU core manages the DFU class V1.1 following the "Device Class Specification for Device Firmware Upgrade Version 1.1 Aug 5, 2004".

This core implements the following aspects of the specification:

- Device descriptor management
- Configuration descriptor management
- Enumeration as DFU device (in DFU mode only)
- Request management (supporting ST DFU sub-protocol)
- Memory request management (Download / Upload / Erase / Detach / GetState / GetStatus)
- DFU state machine implementation.

*Note:* **ST DFU sub-protocol is compliant with DFU protocol. It uses sub-requests to manage memory addressing, command processing, specific memory operations (that is, memory erase, etc.)**

As required by the DFU specification, only endpoint 0 is used in this application.

Other endpoints and functions may be added to the application (that is, HID, etc.).

These aspects may be enriched or modified for a specific user application.

The USB driver does not implement the Manifestation Tolerant mode defined in the specification. However it is possible to manage this feature by modifying the driver.

### 6.3.1 Device firmware upgrade (DFU) class implementation

The DFU transactions are based on endpoint 0 (control endpoint) transfer. All requests and status control are sent / received through this endpoint.

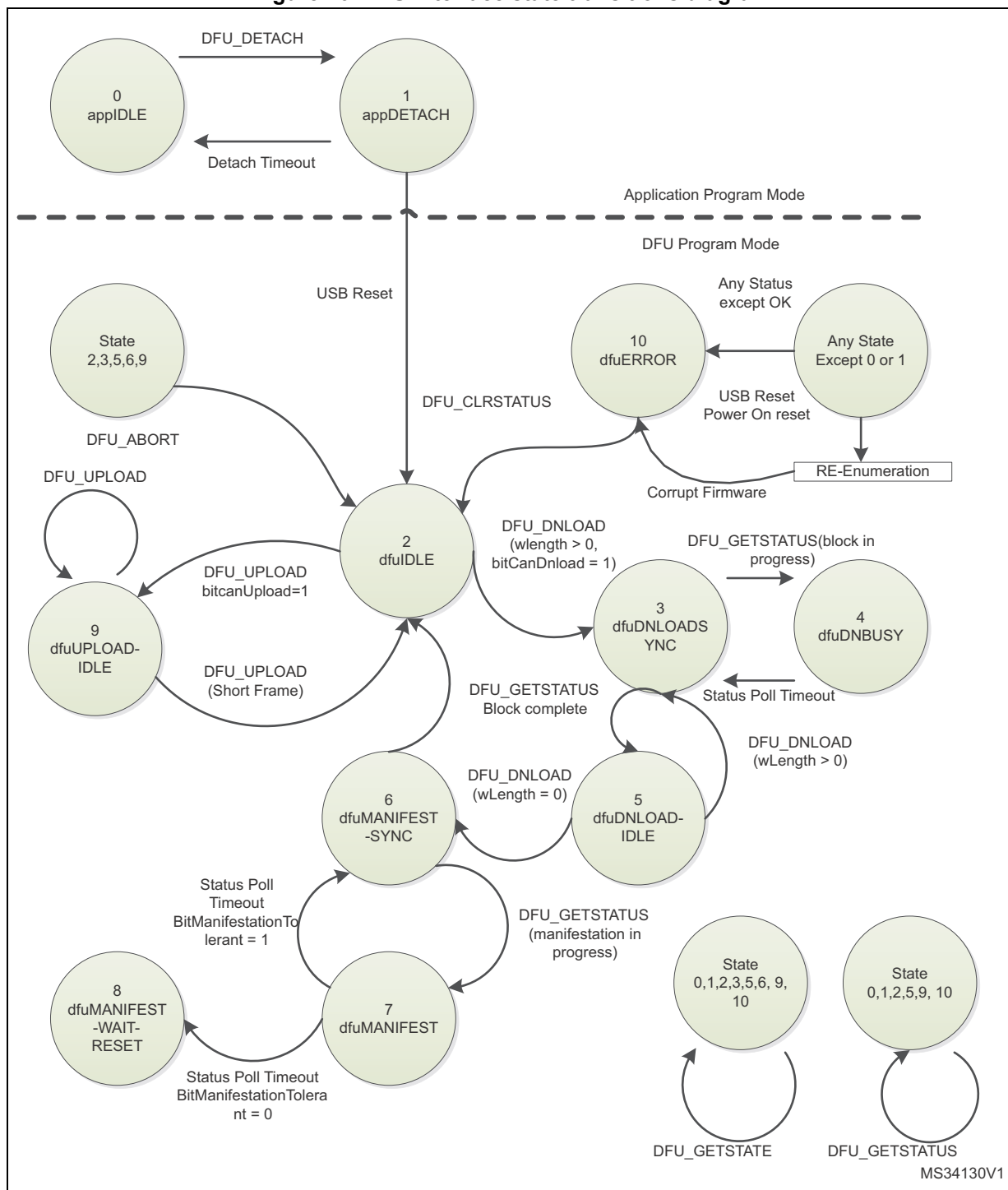
The DFU state machine is based on the following states:

**Table 19. DFU states**

| State                  | State code |
|------------------------|------------|
| appIDLE                | 0x00       |
| appDETACH              | 0x01       |
| dfuIDLE                | 0x02       |
| dfuDNLOAD-SYNC         | 0x03       |
| dfuDNBUSY              | 0x04       |
| dfuDNLOAD-IDLE         | 0x05       |
| dfuMANIFEST-SYNC       | 0x06       |
| dfuMANIFEST            | 0x07       |
| dfuMANIFEST-WAIT-RESET | 0x08       |
| dfuUPLOAD-IDLE         | 0x09       |
| dfuERROR               | 0x0A       |

The allowed state transitions are described in the specification document.

Figure 16. DFU Interface state transitions diagram



To protect the application from spurious accesses before initialization, the initial state of the DFU core (after startup) is dfuERROR. The host must then clear this state by sending a DFU\_CLRSTATE request before generating another request.

The DFU core manages all supported requests (see [Table 20](#)).



Table 20. Supported requests

| Request       | Code | Details  |
|---------------|------|--|
| DFU_DETACH    | 0x00 | When bit 3 in <i>bmAttributes</i> (bit <i>WillDetach</i> ) is set, the device generates a detach-attach sequence on the bus when it receives this request. |
| DFU_DNLOAD    | 0x01 | The firmware image is downloaded via the control-write transfers initiated by the <i>DFU_DNLOAD</i> class specific request.                                |
| DFU_UPLOAD    | 0x02 | The purpose of the upload is to provide the capability of retrieving and archiving a device firmware.  |
| DFU_GETSTATUS | 0x03 | The host employs the <i>DFU_GETSTATUS</i> request to facilitate synchronization with the device.   |
| DFU_CLRSTATUS | 0x04 | Upon receipt of <i>DFU_CLRSTATUS</i> , the device sets a status of OK and transitions to the <i>dfuIDLE</i> state.   |
| DFU_GETSTATE  | 0x05 | This request solicits a report about the state of the device.  |
| DFU_ABORT     | 0x06 | The <i>DFU_ABORT</i> request enables the host to exit from certain states and to return to the <i>DFU_IDLE</i> state.                                      |

Each transfer to the control endpoint belong to one of the two main categories:

- Data transfers: These transfers are used to
  - Get some data from the device (DFU\_GETSTATUS, DFU\_GETSTATE and DFU\_UPLOAD).
  - Or, to send data to the device (DFU\_DNLOAD).
- No-Data transfers: These transfers are used to send control requests from host to device (DFU\_CLRSTATUS, DFU\_ABORT and DFU\_DETACH).

### 6.3.2 Device firmware upgrade (DFU) Class core files

#### usbd\_dfu (.c, .h)

This driver is the main DFU core. It allows the management of all DFU requests and state machine. It does not directly deal with memory media (managed by lower layer drivers).

Table 21. usbd\_dfu (.c,.h) files

| Functions  | Description                                      |
|--|--|
| static uint8_t USBDFU_Init<br>(USBD_HandleTypeDef *pdev, uint8_t cfgidx);                | Initializes the DFU interface.                   |
| static uint8_t USBDFU_DeInit<br>(USBD_HandleTypeDef *pdev, uint8_t cfgidx);              | De-initializes the DFU layer.                    |
| static uint8_t USBDFU_Setup<br>(USBD_HandleTypeDef *pdev,<br>USBD_SetupReqTypeDef *req); | Handles the DFU request parsing.                 |
| static uint8_t USBDFU_EP0_TxReady<br>(USBD_HandleTypeDef *pdev);                         | Handles the DFU control endpoint data IN stage.  |
| static uint8_t USBDFU_EP0_RxReady<br>(USBD_HandleTypeDef *pdev);                         | Handles the DFU control endpoint data OUT stage. |

Table 21. usbd\_dfu (.c,.h) files (continued)

| Functions  | Description  |
|--|--|
| static uint8_t* USBD_DFU_GetUsrStringDesc ( USBD_HandleTypeDef *pdev, uint8_t index , uint16_t *length); | Manages the transfer of memory interfaces string descriptors.  |
| static void DFU_Detach (USB_D_HandleTypeDef *pdev, USB_D_SetupReqTypeDef *req);                          | Handles the DFU DETACH request.  |
| static void DFU_Download (USB_D_HandleTypeDef *pdev, USB_D_SetupReqTypeDef *req);                        | Handles the DFU DNLOAD request.  |
| static void DFU_Upload (USB_D_HandleTypeDef *pdev, USB_D_SetupReqTypeDef *req);                          | Handles the DFU UPLOAD request.  |
| static void DFU_GetStatus (USB_D_HandleTypeDef *pdev);   | Handles the DFU GETSTATUS request.   |
| static void DFU_ClearStatus (USB_D_HandleTypeDef *pdev);   | Handles the DFU CLRSTATUS request.   |
| static void DFU_GetState (USB_D_HandleTypeDef *pdev);  | Handles the DFU GETSTATE request.  |
| static void DFU_Abort (USB_D_HandleTypeDef *pdev);   | Handles the DFU ABORT request.   |
| static void DFU_Leave (USB_D_HandleTypeDef *pdev);   | Handles the sub-protocol DFU leave DFU mode request (leaves DFU mode and resets device to jump to user loaded code). |

**Note:** *Internal Flash memory is the default memory provided by the library. However you can add other memories using the `usbd_dfu_media_template.c` template file.*

To use the driver:

1. Use the file `usbd_conf.h`, to configure:
  - The number of media (memories) to be supported (define `USB_D_DFU_MAX_ITF_NUM`).
  - The application default address (where the image code should be loaded): define `USB_D_DFU_APP_DEFAULT_ADD`.
2. Call `USB_D_DFU_Init()` function to initialize all memory interfaces and DFU state machine.
3. All control/request operations are performed through control endpoint 0, through the functions: `USB_D_DFU_Setup()` and `USB_D_DFU_EP0_TxReady()`. These functions can be used to call each memory interface callback (read/write/erase/get state...) depending on the generated DFU requests. No user action is required for these operations.
4. To close the communication, call the `USB_D_DFU_DeInit()` function.

**Note:** *When the DFU application starts, the default DFU state is `DFU_STATE_ERROR`. This state is set to protect the application from spurious operations before having a correct configuration*

## 6.4 Audio class

The USB driver manages the Audio Class 1.0 following the “USB Device Class Definition for Audio Devices V1.0 Mar 18, 98”.

The driver implements the following aspects of the specification:

- Device descriptor management
- Configuration descriptor management
- Standard AC Interface Descriptor management
- 1 Audio Streaming Interface (with single channel, PCM, Stereo mode)
- 1 Audio Streaming endpoint
- 1 Audio Terminal Input (1 channel)
- Audio Class-Specific AC Interfaces
- Audio Class-Specific AS Interfaces
- Audio Control Requests: only `SET_CUR` and `GET_CUR` requests are supported (for Mute)
- Audio Feature Unit (limited to Mute control)
- Audio Synchronization type: Asynchronous
- Single fixed audio sampling rate (configurable in `usbd_conf.h` file)

*Note: The Audio Class 1.0 is based on USB Specification 1.0 and thus supports only Low and Full speed modes and does not allow High Speed transfers. Please refer to “USB Device Class Definition for Audio Devices V1.0 Mar 18, 98” for more details.*

These aspects can be enriched or modified for a specific user application.

This driver does not implement the following aspects of the specification (but it is possible to manage these features with some modifications on this driver):

- Audio Control endpoint management
- Audio Control requests other than `SET_CUR` and `GET_CUR`
- Abstraction layer for Audio Control requests (only mute functionality is managed)
- Audio Synchronization type: Adaptive
- Audio Compression modules and interfaces
- MIDI interfaces and modules
- Mixer/Selector/Processing/Extension Units (featured unit is limited to Mute control)
- Any other application-specific modules
- Multiple and Variable audio sampling rates
- Audio Out Streaming Endpoint/Interface (microphone)

### 6.4.1 Audio class implementation

The Audio transfers are based on isochronous endpoint transactions. Audio control requests are also managed through control endpoint (endpoint 0).

In each frame, an audio data packet is transferred and must be consumed during this frame (before the next frame). The audio quality depends on the synchronization between data transfer and data consumption. This driver implements simple mechanism of synchronization relying on accuracy of the delivered I2S clock. At each start of frame, the

driver checks if the consumption of the previous packet has been correctly performed and aborts it if it is still ongoing. To prevent any data overwrite, two main protections are used:

- Using DMA for data transfer between USB buffer and output device registers (I2S).
- Using multi-buffers to store data received from USB.

Based on this mechanism, if the clock accuracy or the consumption rates are not high enough, it will result in a bad audio quality.

This mechanism may be enhanced by implementing more flexible audio flow controls like USB feedback mode, dynamic audio clock correction or audio clock generation/control using SOF event.

The driver also supports basic Audio Control requests. To keep the driver simple, only two requests have been implemented. However, other requests can be supported by slightly modifying the audio core driver.

**Table 22. Audio control requests**

| Request | Supported | Meaning  |
|---------|-----------|--|
| SET_CUR | Yes       | Sets Mute mode On or Off (can also be updated to set volume level...). |
| SET_MIN | No        | NA.  |
| SET_MAX | No        | NA.  |
| SET_RES | No        | NA.  |
| SET_MEM | No        | NA.  |
| GET_CUR | Yes       | Gets Mute mode state (can also be updated to get volume level...).     |
| GET_MIN | No        | NA.  |
| GET_MAX | No        | NA.  |
| GET_RES | No        | NA.  |
| GET_MEM | No        | NA.  |

## 6.4.2 Audio core files

### *usbd\_audio (.c, .h)*

This driver is the audio core. It manages audio data transfers and control requests. It does not directly deal with audio hardware (which is managed by lower layer drivers).

**Table 23. usbd\_audio\_core (.c,.h) files**

| Functions  | Description                                |
|--|--|
| static uint8_t USBD_AUDIO_Init<br>(USBD_HandleTypeDef *pdev, uint8_t cfgidx);                | Initializes the Audio interface.           |
| static uint8_t USBD_AUDIO_DeInit<br>(USBD_HandleTypeDef *pdev, uint8_t cfgidx);              | De-initializes the Audio interface.        |
| static uint8_t USBD_AUDIO_Setup<br>(USBD_HandleTypeDef *pdev,<br>USBD_SetupReqTypeDef *req); | Handles the Audio control request parsing. |

**Table 23. usbd\_audio\_core (.c,.h) files (continued)**

| Functions  | Description   |
|--|---|
| static uint8_t USBD_AUDIO_EP0_RxReady (USBD_HandleTypeDef *pdev);                      | Handles audio control requests data.                            |
| static uint8_t USBD_AUDIO_DataIn (USBD_HandleTypeDef *pdev, uint8_t epnum);            | Handles the Audio In data stage.                                |
| static uint8_t USBD_AUDIO_DataOut (USBD_HandleTypeDef *pdev, uint8_t epnum);           | Handles the Audio Out data stage.                               |
| static uint8_t USBD_AUDIO_SOF (USBD_HandleTypeDef *pdev);                              | Handles the SOF event (data buffer update and synchronization). |
| static void AUDIO_REQ_GetCurrent(USBD_HandleTypeDef *pdev, USBD_SetupReqTypeDef *req); | Handles the GET_CUR Audio control request.                      |
| static void AUDIO_REQ_SetCurrent(USBD_HandleTypeDef *pdev, USBD_SetupReqTypeDef *req); | Handles the SET_CUR Audio control request.                      |

The low layer hardware interfaces are managed through their respective driver structure:

**Figure 17. Audio core structures**

|  |
|--|
| <pre>typedef struct {     int8_t (*Init)          (uint32_t AudioFreq, uint32_t Volume, uint32_t options);     int8_t (*DeInit)        (uint32_t options);     int8_t (*AudioCmd)      (uint8_t* pbuf, uint32_t size, uint8_t cmd);     int8_t (*VolumeCtl)     (uint8_t vol);     int8_t (*MuteCtl)       (uint8_t cmd);     int8_t (*PeriodicTC)    (uint8_t cmd);     int8_t (*GetState)      (void); }USBD_AUDIO_ItfTypeDef;</pre> |
|--|

Each audio hardware interface driver should provide a structure pointer of type USBD\_AUDIO\_ItfTypeDef. The functions and constants pointed by this structure are listed in the following sections. If a functionality is not supported by a given memory interface, the relative field is set as NULL value.

### ***usbd\_audio\_if (.c, .h)***

This driver manages the low layer audio hardware. *usbd\_audio\_if.c/h* driver manages the Audio Out interface (from USB to audio speaker/headphone). user can call lower layer Codec driver (i.e. *stm324xg\_eval\_audio.c/h*) for basic audio operations (play/pause/volume control...).

This driver provides the structure pointer:

```
extern USBD_AUDIO_ItfTypeDef USBD_AUDIO_fops;
```

**Table 24. usbd\_audio\_if (.c,.h) files**

| Functions  | Description  |
|--|--|
| static int8_t Audio_Init(uint32_t AudioFreq, uint32_t Volume, uint32_t options); | Initializes the audio interface.   |
| static int8_t Audio_DeInit(uint32_t options);                                    | De-initializes the audio interface and free used resources.                                    |
| static int8_t Audio_PlaybackCmd(uint8_t* pbuf, uint32_t size, uint8_t cmd);      | Handles audio player commands (play, pause...).  |
| static int8_t Audio_VolumeCtl(uint8_t vol);                                      | Handles audio player volume control.   |
| static int8_t Audio_MuteCtl(uint8_t cmd);  | Handles audio player mute state.   |
| static int8_t Audio_PeriodicTC(uint8_t cmd);                                     | Handles the end of current packet transfer (not needed for the current version of the driver). |
| static int8_t Audio_GetState(void);  | Returns the current state of the driver audio player (Playing/Paused/Error ...).               |

**Note:** *The usbd\_audio\_if\_template (.c,.h) file provides a template driver which allows you to implement additional functions for your Audio application*

The Audio player state is managed through the following states:

**Table 25. Audio player states**

| State           | Code | Description                            |
|-----------------|------|--|
| AUDIO_CMD_START | 0x01 | Audio player is initialized and ready. |
| AUDIO_CMD_PLAY  | 0x02 | Audio player is currently playing.     |
| AUDIO_CMD_STOP  | 0x04 | Audio player is stopped.               |

### 6.4.3 How to use this driver

This driver uses an abstraction layer for hardware driver (i.e. HW Codec, I2S interface, I2C control interface...). This abstraction is performed through a lower layer (i.e. *usbd\_audio\_if.c*) which you can modify depending on the hardware available for your application.

To use this driver:

1. Configure the audio sampling rate (define USBD\_AUDIO\_FREQ) through the file *usbd\_conf.h*,
2. Call the USBD\_AUDIO\_Init() function at startup to configure all necessary firmware and hardware components (application-specific hardware configuration functions are also called by this function). The hardware components are managed by a lower layer interface (i.e. *usbd\_audio\_if.c*) and can be modified by user depending on the application needs.
3. The entire transfer is managed by the following functions (no need for user to call any function for out transfers):
  - usbd\_audio\_DataIn() and usbd\_audio\_DataOut() which update the audio buffers with the received or transmitted data. For Out transfers, when data are received,

they are directly copied into the audiobuffer and the write buffer (wr\_ptr) is incremented.

4. The Audio Control requests are managed by the functions USBD\_AUDIO\_Setup() and USBD\_AUDIO\_EP0\_RxReady(). These functions route the Audio Control requests to the lower layer (i.e. *usbd\_audio\_if.c*). In the current version, only SET\_CUR and GET\_CUR requests are managed and are used for mute control only.

#### 6.4.4 Audio known limitations

- If a low audio sampling rate is configured (define USBD\_AUDIO\_FREQ below 24 kHz) it may result in noise issue at pause/resume/stop operations. This is due to software timing tuning between stopping I2S clock and sending mute command to the external Codec.
- Supported audio sampling rates range from 96 kHz to 24 kHz (non-multiple of 1 kHz values like 11.025 kHz, 22.05 kHz or 44.1 kHz are not supported by this driver). For frequencies multiple of 1000 Hz, the Host will send integer number of bytes each frame (1 ms). When the frequency is not multiple of 1000Hz, the Host should send non integer number of bytes per frame. This is in fact managed by sending frames with different sizes (i.e. for 22.05 kHz, the Host will send 19 frames of 22 bytes and one frame of 23 bytes). This difference of sizes is not managed by the Audio core and the extra byte will always be ignored. It is advised to set a high and standard sampling rate in order to get best audio quality (i.e. 96 kHz or 48 kHz). Note that maximum allowed audio frequency is 96 kHz (this limitation is due to the Codec used on the Evaluation board. The STM32 I2S cell enables reaching 192 kHz).

### 6.5 Communication device class (CDC)

The USB driver manages the "Universal Serial Bus Class Definitions for Communications Devices Revision 1.2 November 16, 2007" and the sub-protocol specification of "Universal Serial Bus Communications Class Subclass Specification for PSTN Devices Revision 1.2 February 9, 2007".

This driver implements the following aspects of the specification:

- Device descriptor management
- Configuration descriptor management
- Enumeration as CDC device with 2 data endpoints (IN and OUT) and 1 command endpoint (IN)
- Request management (as described in section 6.2 in specification)
- Abstract Control Model compliant
- Union Functional collection (using 1 IN endpoint for control)
- Data interface class

**Note:** *For the Abstract Control Model, this core can only transmit the requests to the lower layer dispatcher (i.e. *usbd\_cdc\_vcp.c/.h*) which should manage each request and perform relative actions.*

These aspects can be enriched or modified for a specific user application.

This driver does not implement the following aspects of the specification (but it is possible to manage these features with some modifications on this driver):

- Any class-specific aspect relative to communication classes should be managed by user application.
- All communication classes other than PSTN are not managed.

### 6.5.1 Communication

The CDC core uses two endpoint/transfer types:

- Bulk endpoints for data transfers (1 OUT endpoint and 1 IN endpoint)
- Interrupt endpoints for communication control (CDC requests; 1 IN endpoint)

Data transfers are managed differently for IN and OUT transfers:

### 6.5.2 Data IN transfer management (from device to host)

The data transfer is managed periodically depending on host request (the device specifies the interval between packet requests). For this reason, a circular static buffer is used for storing data sent by the device terminal (i.e. USART in the case of Virtual COM Port terminal).

### 6.5.3 Data OUT transfer management (from host to device)

In general, the USB is much faster than the output terminal (i.e. the USART maximum bitrate is 115.2 Kbps while USB bitrate is 12 Mbps for Full speed mode and 480 Mbps in High speed mode). Consequently, before sending new packets, the host has to wait until the device has finished to process the data sent by host. Thus, there is no need for circular data buffer when a packet is received from host: the driver calls the lower layer OUT transfer function and waits until this function is completed before allowing new transfers on the OUT endpoint (meanwhile, OUT packets will be NACKed).

### 6.5.4 Command request management

In this driver, control endpoint (endpoint 0) is used to manage control requests. But a data interrupt endpoint may be used also for command management. If the request data size does not exceed 64 bytes, the endpoint 0 is sufficient to manage these requests.

The CDC driver does not manage command requests parsing. Instead, it calls the lower layer driver control management function with the request code, length and data buffer. Then this function should parse the requests and perform the required actions.

### 6.5.5 Command device class (CDC) core files

#### *usbd\_cdc (.c, .h)*

This driver is the CDC core. It manages CDC data transfers and control requests. It does not directly deal with CDC hardware (which is managed by lower layer drivers).



Table 26. usbd\_cdc (.c,.h) files

| Functions  | Description                            |
|--|--|
| static uint8_t USBD_CDC_Init<br>(USBD_HandleTypeDef *pdev, uint8_t cfgidx);                    | Initializes the CDC interface.         |
| static uint8_t USBD_CDC_DeInit<br>(USBD_HandleTypeDef *pdev, uint8_t cfgidx);                  | De-initializes the CDC interface.      |
| static uint8_t USBD_CDC_Setup<br>(USBD_HandleTypeDef *pdev,<br>USBD_SetupReqTypeDef *req);     | Handles the CDC control requests.      |
| static uint8_t USBD_CDC_EP0_RxReady<br>(USBD_HandleTypeDef *pdev);                             | Handles CDC control request data.      |
| static uint8_t USBD_CDC_DataIn<br>(USBD_HandleTypeDef *pdev, uint8_t epnum);                   | Handles the CDC IN data stage.         |
| static uint8_t USBD_CDC_DataOut<br>(USBD_HandleTypeDef *pdev, uint8_t epnum);                  | Handles the CDC Out data stage.        |
| uint8_t USBD_CDC_RegisterInterface<br>(USBD_HandleTypeDef *pdev,<br>USBD_CDC_ItfTypeDef *fops) | Adds CDC Interface Class.              |
| uint8_t USBD_CDC_SetTxBuffer<br>(USBD_HandleTypeDef *pdev, uint8_t *pbuff,<br>uint16_t length) | Sets application TX Buffer.            |
| uint8_t USBD_CDC_SetRxBuffer<br>(USBD_HandleTypeDef *pdev, uint8_t *pbuff)                     | Sets application RX Buffer.            |
| uint8_t<br>USBD_CDC_TransmitPacket(USBD_HandleTypeDef *pdev)                                   | Transmits Transfer completed callback. |
| uint8_t<br>USBD_CDC_ReceivePacket(USBD_HandleTypeDef *pdev)                                    | Receives Transfer completed callback.  |

The low layer hardware interfaces are managed through their respective driver structure

Figure 18. CDC core structures

```
typedef struct _USBD_CDC_Itf
{
    int8_t (* Init)          (void);
    int8_t (* DeInit)       (void);
    int8_t (* Control)      (uint8_t, uint8_t *, uint16_t);
    int8_t (* Receive)      (uint8_t *, uint32_t *);
}USBD_CDC_ItfTypeDef;
```

Each hardware interface driver should provide a structure pointer of type USBD\_CDC\_ItfTypeDef. The functions pointed by this structure are listed in the following sections.

If a function is not supported by a given memory interface, the relative field is set as NULL value.

**Note:** *In order to get the best performance, it is advised to calculate the values needed for the following parameters (all of them are configurable through defines in the `usbd_cdc.h` and `usbd_cdc_interface.h` files):*

**Table 27. Configurable CDC parameters**

| Define  | Parameter  | Typical value |            |
|---|--|---------------|------------|
|   |  | Full Speed    | High Speed |
| CDC_DATA_HS_IN_PACKET_SIZE / CDC_DATA_FS_IN_PACKET_SIZE   | Size of each IN data packet.                                   | 64            | 512        |
| CDC_DATA_HS_OUT_PACKET_SIZE / CDC_DATA_FS_OUT_PACKET_SIZE | Size of each OUT data packet.                                  | 64            | 512        |
| APP_TX_DATA_SIZE  | Total size of circular temporary buffer for OUT data transfer. | 2048          | 2048       |
| APP_RX_DATA_SIZE  | Total size of circular temporary buffer for IN data transfer.  | 2048          | 2048       |

### usbd\_cdc\_interface (.c, .h)

This driver can be part of the user application. It is not provided in the library, but a template **usbd\_cdc\_if\_template (.c, .h)** can be used to build it and an example is provided for the USART interface. It manages the low layer CDC hardware. The `usbd_cdc_interface.c/.h` driver manages the terminal interface configuration and communication (i.e. USART interface configuration and data send/receive).

This driver provides the structure pointer:

**Figure 19. CDC interface callback structure**

```

USBD_CDC_ItfTypeDef USBD_CDC_fops =
{
    CDC_Itf_Init,
    CDC_Itf_DeInit,
    CDC_Itf_Control,
    CDC_Itf_Receive
};

```

**Table 28. usbd\_cdc\_interface (.c,.h) files**

| Functions                            | Description                                 |
|--------------------------------------|---|
| static int8_t CDC_Itf_Init (void);   | Initializes the low layer CDC interface.    |
| static int8_t CDC_Itf_DeInit (void); | De-initializes the low layer CDC interface. |

**Table 28. usbd\_cdc\_interface (.c,.h) files**

| Functions   | Description   |
|---|---|
| static int8_t CDC_Itf_Control<br>(uint8_t cmd, uint8_t* pbuf, uint16_t length); | Handles CDC control request parsing and execution.                              |
| static int8_t CDC_Itf_Receive<br>(uint8_t* pbuf, uint32_t *Len);                | Handles CDC data reception from USB host to low layer terminal (OUT transfers). |

In order to accelerate data management for IN/OUT transfers, the low layer driver (*usbd\_cdc\_interface.c/.h*) use these global variables:

**Table 29. Variables used by usbd\_cdc\_xxx\_if.c/.h**

| Variable                                  | Usage   |
|---|---|
| uint8_t<br>UserRxBuffer[APP_RX_DATA_SIZE] | Writes CDC received data in this buffer from USART. These data will be sent over USB IN endpoint in the CDC core functions. |
| uint32_t UserTxBufPtrOut                  | Increments this pointer or rolls it back to start the address when writing received data in the buffer UserRxBuffer.        |
| uint8_t<br>UserTxBuffer[APP_TX_DATA_SIZE] | Writes CDC received data in this buffer. These data are received from USB OUT endpoint in the CDC core functions.           |
| UserTxBufPtrIn                            | Increments this pointer or rolls back to start address when data are received over USART.                                   |

### 6.5.6 How to use

The USB driver uses an abstraction layer for hardware driver (i.e. USART control interface...). This abstraction is performed through a lower layer (i.e. *stm32fxxx\_hal\_msp.c*) which you can modify depending on the hardware available for your application.

To use this driver:

- Through the file *usbd\_cdc.h* and *usbd\_cdc\_interface.h*, configure:
  - The Data IN and OUT and command packet sizes (defines CDC\_DATA\_XX\_IN\_PACKET\_SIZE, CDC\_DATA\_XX\_OUT\_PACKET\_SIZE)
  - The size of the temporary circular buffer for IN/OUT data transfer (define APP\_RX\_DATA\_SIZE and APP\_TX\_DATA\_SIZE).
  - The device string descriptors.
- Call the function `USBD_CDC_Init()` at startup to configure all necessary firmware and hardware components (application-specific hardware configuration functions are called by this function as well). The hardware components are managed by a lower layer

interface (i.e. *usbd\_cdc\_interface.c*) and can be modified by user depending on the application needs.

3. CDC IN and OUT data transfers are managed by two functions:
  - USBD\_CDC\_SetTxBuffer should be called by user application each time a data (or a certain number of data) is available to be sent to the USB Host from the hardware terminal.
  - USBD\_CDC\_SetRxBuffer is called by the CDC core each time a buffer is sent from the USB Host and should be transmitted to the hardware terminal. This function should exit only when all data in the buffer are sent (the CDC core then blocks all coming OUT packets until this function finishes processing the previous packet).
4. CDC control requests should be handled by the function *Controllability()*. This function is called each time a request is received from Host and all its relative data are available if any. This function should parse the request and perform the needed actions.
5. To close the communication, call the function *USBD\_CDC\_DeInit()*. This closes the used endpoints and calls lower layer de-initialization functions.

### 6.5.7 CDC known limitations

When using this driver with the OTG HS core, enabling DMA mode (define *USB\_OTG\_HS\_INTERNAL\_DMA\_ENABLED* in *usb\_conf.h* file) results in data being sent only by multiple of 4 bytes. This is due to the fact that USB DMA does not allow sending data from non word-aligned addresses. For this specific application, it is advised not to enable this option unless required.

## 6.6 Adding a custom class

This section explains how to create a new custom class based on an existing USB class.

To create a new custom Class, follow the steps below:

1. Add *USBD\_CustomClass\_cb* (In order to receive various USB bus Events) as described in [Section 5.3](#), in the *usbd\_template.c/.h* available under **Class/Template**

**directory.** This template contains all the functions that should be adapted to the application's needs and may be also used to implement any type of USB Device class.

2. Customizing the descriptors:

The descriptors retrieved by the host must be configured to describe a device depending on the specifications for the application class devices. The following list is not complete but gives an overview about the various descriptors that may be required:

- Standard device descriptor
- Standard configuration descriptor
- Standard interface descriptor for the Class that is implemented
- Standard endpoint descriptors for IN and OUT endpoints

3. The firmware must configure the STM32 to enable USB transfer (isochronous, Bulk, Interrupt or Control) depending on the user application:

- In the `DataIn` and `DataOut` functions, the user can implement the internal protocol or state machine
- In the *Setup*; the class specific requests are to be implemented. The configuration descriptor is to be added as an array and passed to the USB device library.
- Through the `GetConfigDescriptor` function which should return a pointer to the USB configuration descriptor and its length.
- Additional functions could be added as the `IsoINIncomplete` and `IsoOUTIncomplete` could be eventually used to handle incomplete isochronous transfers (for more information, refer to the *USB audio device example*).
- `EP0_TxSent` and `EP0_RxReady` could be eventually used when the application needs to handle events occurring before the Zero Length Packets (see the *DFU example*).

4. Memory allocation process: Memory is allocated to the applications using the `malloc` (`USBD_malloc`):

- `USBD_malloc(sizeof (USBD_CUSTOM_CLASS_HandleTypeDef))`: this is dynamically allocates memory for a Class structure

## 6.7 Library footprint optimization

In this section we review some basic tips about how to optimize the footprint of an application developed on top of the USB device library.

Reducing the USB examples footprint is important objective especially for STM32 products with small Flash/RAM memory size, such as STM32 L0 and F0 series.

### Reduce the heap and stack size settings (in the Linker file)

The stack is the memory area where the program stores:

- Local variables
- Return addresses
- Function arguments
- Compiler temporaries
- Interrupt contexts

If your linker configuration reserves more amounts of heap and stack than necessary for your application, you can determine accurately the appropriate sizes.

### Whenever possible use local instead if global variables

If a variable is used only in a function, then it should be declared inside the function as a local variable.

### Constant should be allocated in the flash

It is recommended to allocate all constant global variables, which never change, to a read-only section. As example, the USB descriptors are declared as constant using the C keyword "const".

**Figure 20. Example of USB descriptors declared as constants**

```
/* USB Standard Device Descriptor */
const uint8_t USBD_DeviceDesc[USB_LEN_DEV_DESC]= {
    0x12,                /* bLength */
    USB_DESC_TYPE_DEVICE, /* bDescriptorType */
    0x00,                /* bcdUSB */
    0x02,
    0x00,                /* bDeviceClass */
    0x00,                /* bDeviceSubClass */
    0x00,                /* bDeviceProtocol */
    USB_MAX_EP0_SIZE,    /* bMaxPacketSize */
    LOBYTE(USBD_VID),    /* idVendor */
    HIBYTE(USBD_VID),    /* idVendor */
    LOBYTE(USBD_PID),    /* idVendor */
    HIBYTE(USBD_PID),    /* idVendor */
    0x00,                /* bcdDevice rel. 2.00 */
    0x02,
    USBD_IDX_MFC_STR,     /* Index of manufacturer string */
    USBD_IDX_PRODUCT_STR, /* Index of product string */
    USBD_IDX_SERIAL_STR, /* Index of serial number string */
    USBD_MAX_NUM_CONFIGURATION /* bNumConfigurations */
}; /* USB_DeviceDescriptor */
```

### Use static memory allocation rather than malloc

The USB device library uses dynamic memory allocation for a class handle structure to allow multi-instance support (in case of the dual core operation), this means for example we can have same USB class used for the two instances of the USB (HS and FS).

The secondary reason for using dynamic allocation is to allow freeing memory when USB is no more used.

However dynamic memory allocation adds some footprint overhead, mainly for the ROM memory. For this it's advised to use static allocation for the low memory STM32 devices or when multi-instance support is not needed. In that case it's necessary to declare a static buffer having the size of the class handle structure.

Below an example of implementation:

1. In usbd\_conf.h file, define the memory static allocation and routines;

```

USBStatic_malloc() and USBStatic_free()

#define MAX_STATIC_ALLOC_SIZE 4 /* HID Class structure size */
#define USBD_malloc      (uint32_t *)USBStatic_malloc
#define USBD_free        USBStatic_free

```

2. The implementation is done in usbd\_conf.c file as below:

**Figure 21. Example of dynamic memory allocation for class structure**

```

/**
 * @brief static single allocation.
 * @param size: size of allocated memory
 * @retval None
 */
void *USBStatic_malloc(uint32_t size)
{
    static uint32_t mem[MAX_STATIC_ALLOC_SIZE];
    return mem;
}

/**
 * @brief Dummy memory free
 * @param *p pointer to allocated memory address
 * @retval None
 */
void USBStatic_free(void *p)
{
}

```

## 7 Frequently-asked questions

### 1. How can the Device and string descriptors be modified on-the-fly?

In the *usbd\_desc.c* file, the descriptor relative to the device and the strings can be modified using the Get Descriptor callbacks. The application can return the correct descriptor buffer relative to the application index using a switch case statement.

### 2. How can the mass storage class driver support more than one logical unit (LUN)?

In the *usbd\_msc\_storage\_template.c* file, all the APIs needed to use physical media are defined. Each function comes with the “LUN” parameter to select the addressed media.

The number of supported LUNs can be changed using the define `STORAGE_LUN_NBR` in the *usbd\_msc\_storage\_xxx.c* file (where, xxx is the medium to be used).

For the inquiry data, the `STORAGE_Inquirydata` buffer contains the standard inquiry data for each LUN.

Example: 2 LUNs are used

```
const int8_t STORAGE_Inquirydata[] = {

    /* LUN 0 */
    0x00,
    0x80,
    0x02,
    0x02,
    (USBD_STD_INQUIRY_LENGTH - 5),
    0x00,
    0x00,
    0x00,
    'S', 'T', 'M', ' ', ' ', ' ', ' ', ' ', /* Manufacturer:
      8 bytes */
    'm', 'i', 'c', 'r', 'o', 'S', 'D', ' ', /* Product:
      16 Bytes */
    'F', 'l', 'a', 's', 'h', ' ', ' ', ' ', ' ',
    '1', '.', '0', '0', /* Version: 4 Bytes */

    /* LUN 1 */
    0x00,
    0x80,
    0x02,
    0x02,
    (USBD_STD_INQUIRY_LENGTH - 5),
    0x00,
    0x00,
    0x00,
    'S', 'T', 'M', ' ', ' ', ' ', ' ', ' ', /* Manufacturer:
      8 bytes */
```



```
'N', 'a', 'n', 'd', ' ', ' ', ' ', ' ', ' ', /* Product:
    16 Bytes */
'F', 'l', 'a', 's', 'h', ' ', ' ', ' ', ' ',
'1', '.', '0', '0', /* Version: 4 Bytes */
};
```

### 3. Where endpoints address are defined?

Endpoints address are defined in the header file of the class driver. In the case of the MSC demo case for example, the IN/OUT endpoints address are defined in the `usbd_msc.h` file as below:

```
#define MSC_EPIN_ADDR      0x81 For Endpoint 1 IN
#define MSC_EPOUT_ADDR     0x01 For Endpoint 1 OUT
```

### 4. Can the USB device library be configured to run in either High Speed or Full Speed mode?

Yes, the library can handle the USB OTG HS and USB OTG FS core, if the USB OTG FS core can only work in Full Speed mode, the USB OTG HS can work in High or Full Speed mode.

To select the appropriate USB Core to work with, user must add the following macro defines within the compiler preprocessor (already done in the preconfigured projects provided with the examples):

- "USE\_USB\_HS" when using USB High Speed (HS) Core
- "USE\_USB\_FS" when using USB Full Speed (FS) Core
- "USE\_USB\_HS" and "USE\_USB\_HS\_IN\_FS" when using USB High Speed (HS) Core in FS mode

### 5. How can the used endpoints be changed in the USB device class driver?

To change the endpoints or to add a new endpoint:

- a) Perform the endpoint initialization using `USBD_LL_OpenEP()`.
- b) Configure the TX or the Rx FIFO size of the new defined endpoints in the `usb_conf.c` file using these APIs in the `USBD_LL_Init()` function
  - For STM32F2 and STM32F4 series (FS and HS cores):
 

```
HAL_PCD_SetRxFiFo()
HAL_PCD_SetTxFiFo()
```

The total size of the Rx and Tx FIFOs should be lower than the total FIFO size of the used core, that is 320 x 32 bits (1.25 Kbytes) for USB OTG FS core and 1024 x 32 bits (4 Kbytes) for the USB OTG HS core.

- For STM32F0, STM32L0, STM32F1 and STM32F3 series (FS core only):
 

```
HAL_PCD_PMA_Config()
```

### 6. Is the USB device library compatible with Real Time operating system (RTOS)?

Yes, The USB device library could be used with RTOS, the CMSIS RTOS wrapper is used to make abstraction with OS kernel.

## 8 Revision history

**Table 30. Document revision history**

| Date        | Revision | Changes  |
|-------------|----------|--|
| 27-May-2014 | 1        | Initial release.   |
| 28-Nov-2014 | 2        | Updated <a href="#">Section : Introduction</a> , <a href="#">Figure 1: STM32Cube block diagram</a> and <a href="#">Section 2.1: Overview</a><br>All figures: added missing titles, updated figure style and clarified color codes.<br>updated sequence to use the driver in <a href="#">Section 6.3.2: Device firmware upgrade (DFU) Class core files</a> , <a href="#">Section 6.4.3: How to use this driver</a> , <a href="#">Section 6.5.6: How to use</a> and <a href="#">Section 6.6: Adding a custom class</a> . |
| 27-May-2015 | 3        | <a href="#">Section : Introduction</a> updated and merged with section <i>STM32Cube overview</i> .   |

**IMPORTANT NOTICE – PLEASE READ CAREFULLY**

STMicroelectronics NV and its subsidiaries ("ST") reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST's terms and conditions of sale in place at the time of order acknowledgement.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of Purchasers' products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2015 STMicroelectronics – All rights reserved