# UM1717
# User manual

## STM32F0x2xx USB Full Speed Device Library

## Introduction

The STM32F0x2xx USB Full Speed Device Library (STSW-STM32092) is a firmware and application software package that includes examples based on a set of six classes (Audio, CCID, CDC, HID, MSC and DFU), for easy development of applications using USB full speed transfer types (control, interrupt, bulk and isochronous).

The STM32F072xx and STM32F042xx devices embed the following new features:

- The LPM (Link Power Management) to introduce a new power-save state, L1(Sleep), with fast entry and exit times compared to traditional L2 state (Suspend)
- Analog USB Phy Transceiver with BCD (Battery Charging Device)
- Integrated CRS (Clock Recovery System) to get precise-enough clock for USB without any external resonator component (Crystal) just using the USB traffic as timing reference.

This new USB FS Device Library is a STM32F0x2xx devices-dedicated and is the result of merging the current USB FS Device Library (V4.0.0) and the USB OTG Host and Device Library (V2.1.0) ensuring a full API compatibility.

It is built with a reduced footprint to provide optimum solution for low memory STM32 products.

This document describes all the components, including examples for the following types of devices:

Human Interface Device HID:

– HID mouse and Custom HID examples

Audio:

– Audio device Example for streaming audio data

Communication Device (CDC):

– VCP USB-to-RS232 bridge to realize a virtual COM port.

BULK:

– Mass Storage Demo based on the micro SD

Device Firmware Upgrade:

– DFU for firmware downloads and uploads

CCID: Integrated Circuits Cards Interface devices (New development)

– USB CCID device

Composite examples:

– CDC-HID and HID-MSC.

All the examples are developed and validated on the STM32072B-EVAL evaluation board and can be easily tailored to any other hardware.

# Contents

# List of tables

# List of figures

# 1 Reference information

## 1.1 Glossary

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

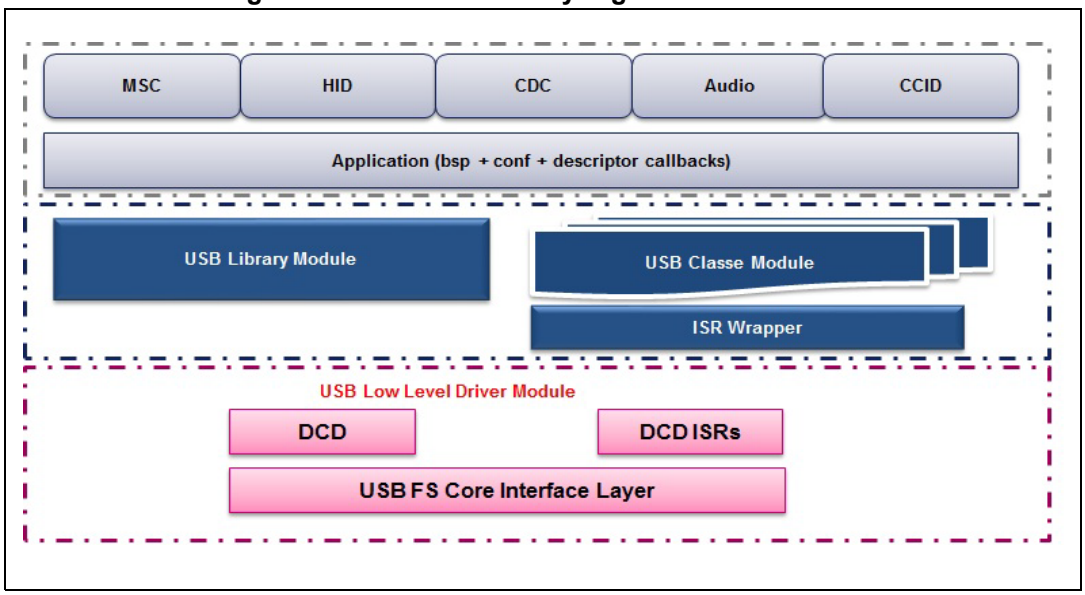
**Table 1. List of terms**

| Term | Meaning |
|---|---|
| API | Application Programming Interface |
| ADC | Analog-to-Digital Conversion |
| CCID | Integrated Circuit(s) Cards Interface Devices |
| CDC | Communication Device Class |
| DAC | Digital-to-Analog Conversion |
| DCD | Device Core Driver |
| DFU | Device Firmware Upgrade |
| FS | Full Speed (12 Mbps) |
| GUI | Graphical User Interface |
| HID | Human Interface Device |
| Mbps | Megabit per second |
| MSC | Mass Storage Class |
| PID | USB Product Identifier |
| SOF | Start Of Frame |
| VID | USB Vendor Identifier |
| USB | Universal Serial Bus |

# 2 USB device library overview

The following figure gives an overview of the USB device library.

**Figure 1. USB device library organization overview**




The USB device library is mainly divided into many layers with the applications being developed on top of theme.

## 2.1 Main features

The USB device library is:

- Compatible with the FS USB modes
- Fully compliant with the Universal Serial Bus Revision 2.0 Specification.
- Built with a reduced footprint in order to provide optimum solution for low memory STM32 products.

Built following a generic and easy-to-use architecture

- Able to add further specific vendor classes.
- Supports multi-interface applications (composite devices)

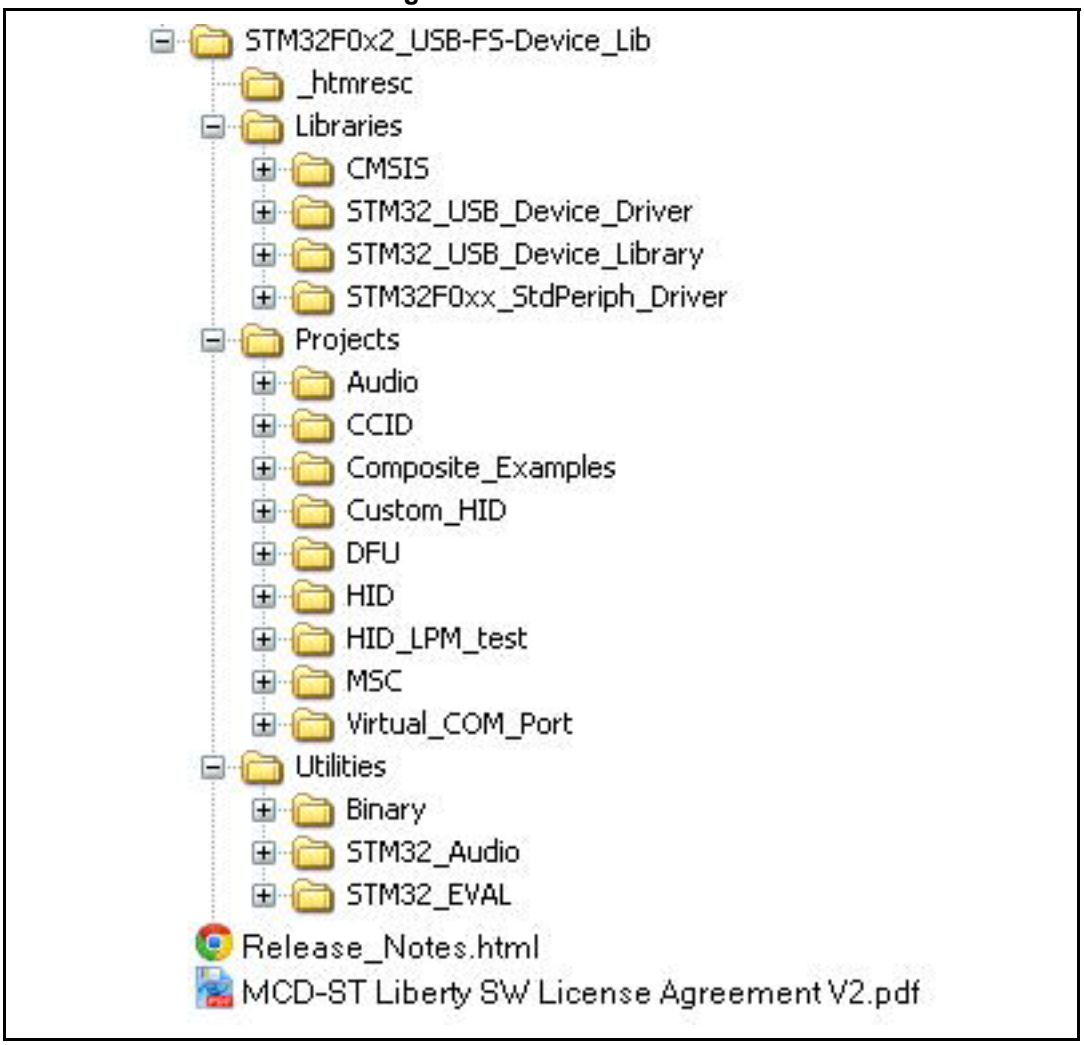
# 3 USB device library folder structure

*Figure 2* illustrates the tree structure of the USB device library folder.

The project is composed of three main directories, organized as follows:

1. ***Libraries***: contains the STM32 USB low-level driver, the standard peripherals libraries, the device libraries.
2. ***Projects***: contains the workspaces and the sources files for the examples given with the package.
3. ***Utilities***: contains the STM32 drivers relative to the used boards (SD card, buttons, joystick, etc). This folder contains also the related directory which includes a set of sources files that implement the Audio Low Layer Drivers.

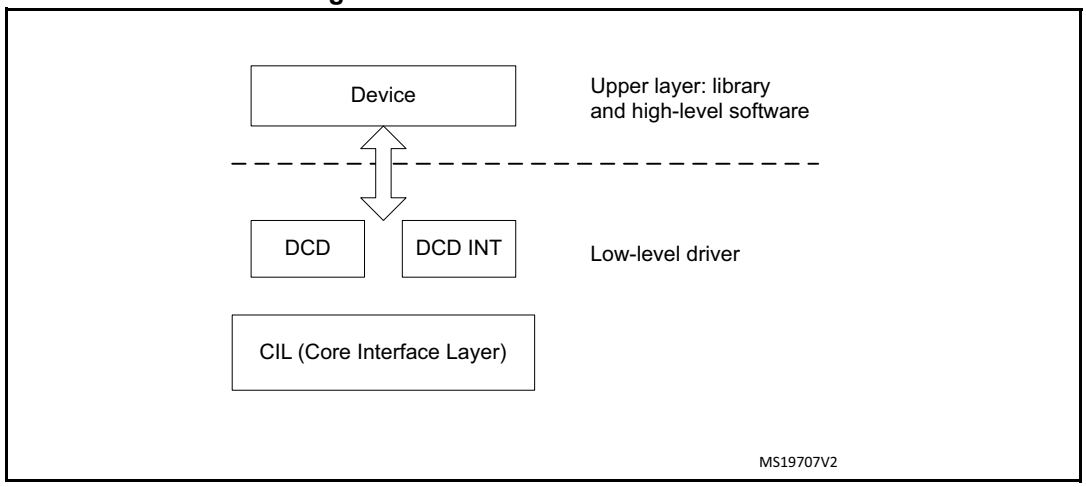
**Figure 2. Folder structure**



# 4 USB low level driver

## 4.1 USB low level driver architecture

**Figure 3. Driver architecture overview**




The low level driver can be used to connect the USB core with the high level stack. The user may develop an interface layer above the Low level driver to provide the adequate APIs needed by the used stack.

## 4.2 USB low level driver files

**Figure 4. Driver files**

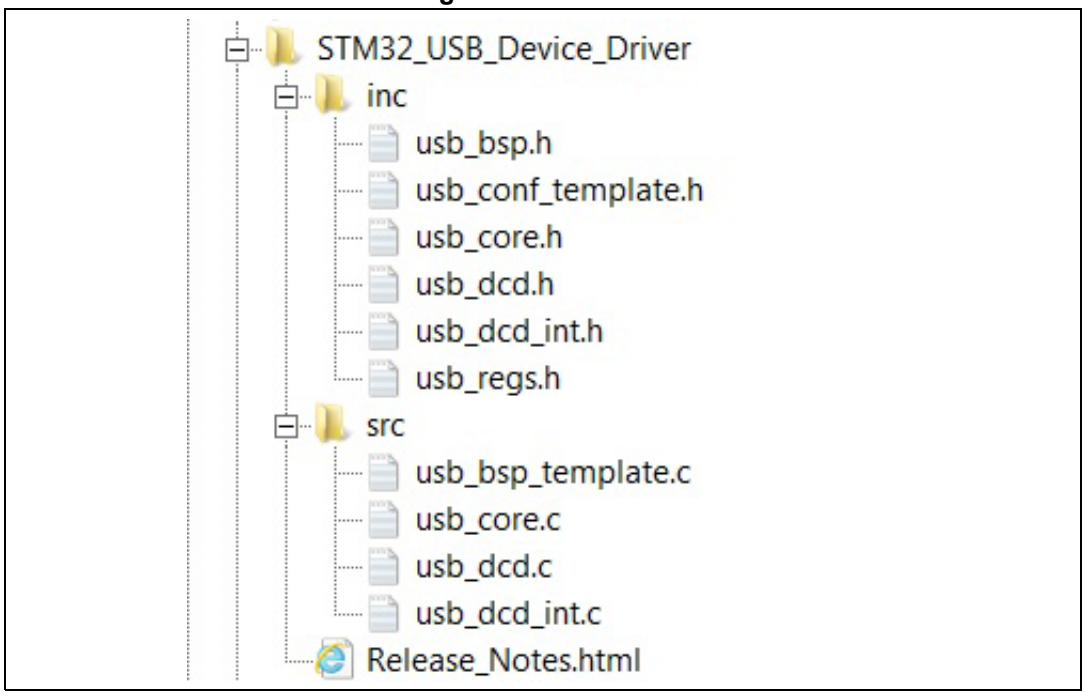


Table 2. USB low level file descriptions

| Mode | Files | Description |
|---|---|---|
| Common | *usb_core.c/h* | This file provides the interface functions to USB cell registers |
| | *usb_reg.h* | This file implements the hardware abstraction layer, it offers a set of basic functions for accessing the USB-FS_Device peripheral registers |
| | *usb_bsp_template.c* | This file contains the low level core configuration (interrupts, GPIO).<br>This file should be copied to the application folder and modified depending on the application needs. |
| Device | *usb_dcd.c/h* | This file contains the device interface layer used by the library to access the core. |
| | *usb_dcd_int.c/h* | This file contains the interrupt subroutines for the Device mode. |

## 4.3 USB driver programming manual

### 4.3.1 Low level driver structures

The low level driver does not have any exportable variables. A global structure `(USB_CORE_HANDLE)` which keeps all the variables, state and buffers used by the core to handle its internal state and transfer flow, should be used to allocate in the application layer the handle instance for the core to be used.

The global USB core structure is defined as follows:

```
typedef struct USB_Device_handle
{
  DCD_DEV     dev;
}
USB_DEVICE_HANDLE, USB_CORE_HANDLE;
```

### 4.3.2 Programming device drivers

#### Device initialization

The device is initialized using the following function:

```
DCD_Init (USB_CORE_HANDLE *pdev)
```

#### Endpoint configuration

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

```
uint32_t DCD_EP_Open (USB_CORE_HANDLE *pdev ,
uint8_t ep_addr,
uint16_t ep_mps,
uint8_t ep_type)
uint32_t DCD_EP_Close (USB_CORE_HANDLE *pdev,
uint8_t ep_addr)
```

### Device core structure

The DCD_DEV structures contain all the variables and structures used to keep in real-time all the information related to devices, the control transfer state machine and also the endpoint information and status.

```
typedef struct _DCD
{
  uint8_t        device_config;
  uint8_t        device_state;
  uint8_t        device_status;
  uint8_t        device_old_status;
  uint8_t        device_address;
  uint32_t       DevRemoteWakeup;
  uint32_t       speed;
  uint8_t        setup_packet [8];
  USB_EP     in_ep   [EP_NUM];
  USB_EP     out_ep  [EP_NUM];
  USBD_Class_cb_TypeDef         *class_cb;
  USBD_Usr_cb_TypeDef           *usr_cb;
  USBD_DEVICE                   *usr_device;
  uint8_t        *pConfig_descriptor;
 }
DCD_DEV , *DCD_PDEV;
```

*Note: The USB_Device_dev struct size depends on the Endpoint number specified in the `EP_NUM` define in the `usb_conf.h` file. The minimum size (when EPU_NUM=1) is 128 bytes. This size is increased by 80 bytes when you add a new endpoint, the table below summarizes the USB_Device_dev size in term of EP_NUM.*

**Table 3. USB_Device_dev struct size**

| Size of *USB_Device_dev (Bytes)* | EP_NUM (n#0) |
|---|---|
| 128 | EPU_NUM = 1 (n=0) |
| 208 | EPU_NUM = 2 (n=1) |
| 288 | EPU_NUM = 3 (n=2) |
| 368 | EPU_NUM = 4 (n=3) |
| ... | ..... |
| 128 + n*80 | EPU_NUM = n |

In this structure, `device_config` holds the current USB device configuration and *device_state* controls the state machine with the following states:

```
/* EP0 State */
#define USB_EP0_IDLE              0
#define USB_EP0_SETUP             1
#define USB_EP0_DATA_IN           2
#define USB_EP0_DATA_OUT          3
#define USB_EP0_STATUS_IN         4
#define USB_EP0_STATUS_OUT        5
#define USB_EP0_STALL             6
```

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

```
/* Device Status */
#define USB_UNCONNECTED 0
#define USB_DEFAULT                          1
#define USB_ADDRESSED                        2
#define USB_CONFIGURED                       3
#define USB_SUSPENDED                        4
```

### USB data transfer flow

The DCD layer offers the user all APIs needed to start and control a transfer flow using the following set of functions:

```
uint32_t   DCD_EP_PrepareRx ( USB_CORE_HANDLE *pdev,
                       uint8_t   ep_addr,
                       uint8_t *pbuf,
                       uint16_t  buf_len);
uint32_t    DCD_EP_Tx (USB_CORE_HANDLE *pdev,
                              uint8_t  ep_addr,
                              uint8_t  *pbuf,
                              uint32_t   buf_len);
uint32_t    DCD_EP_Stall (USB_CORE_HANDLE *pdev,
                             uint8_t   epnum);
uint32_t    DCD_EP_ClrStall (USB_CORE_HANDLE *pdev,
                                 uint8_t epnum);
```

The DCD layer of the USB Low Level Driver has one function that must be called by the USB interrupt :

```
uint32_t    DCD_Handle_ISR (USB_CORE_HANDLE *pdev)
```

The *usb_dcd_int.h* file contains the function prototypes of the functions called from the library core layer to handle the USB events.

### USB driver structure definition

```
typedef struct _USBD_DCD_INT
{
   uint8_t (* DataOutStage) (USBCORE_HANDLE *pdev , uint8_t epnum);
   uint8_t (* DataInStage)  (USB_CORE_HANDLE *pdev , uint8_t epnum);
   uint8_t (* SetupStage) (USB_CORE_HANDLE *pdev);
```

```
  uint8_t (* SOF) (USB_CORE_HANDLE *pdev);
  uint8_t (* Reset) (USB_CORE_HANDLE *pdev);
  uint8_t (* Suspend) (USB_CORE_HANDLE *pdev);
  uint8_t (* Resume) (USB_CORE_HANDLE *pdev);
}USBD_DCD_INT_cb_TypeDef;
```

In the library layer, once the `USBD_DCD_INT_cb_TypeDef` structure is defined, it should be assigned to the `USBD_DCD_INT_fops` pointer.

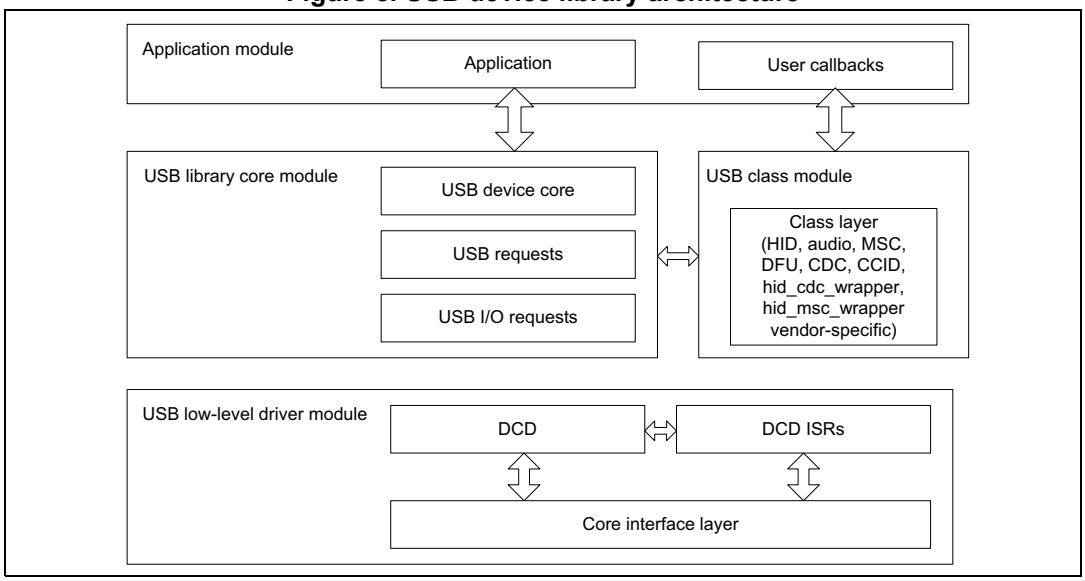
Example:

```
USBD_DCD_INT_cb_TypeDef  *USBD_DCD_INT_fops = &USBD_DCD_INT_cb;
```

# 5 USB device library

## 5.1 USB device library overview

**Figure 5. USB device library architecture**

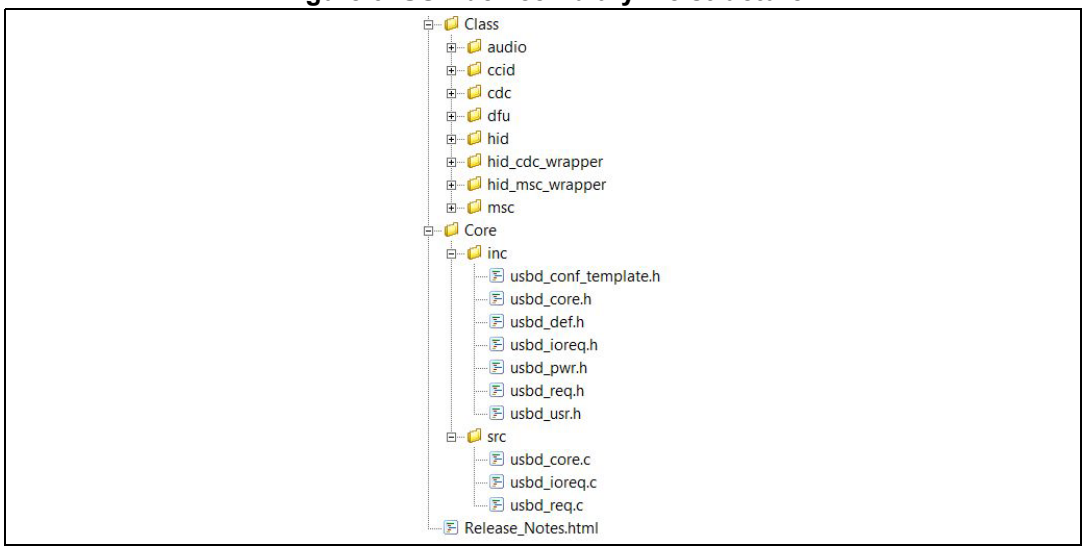



The USB device library is based on the generic USB low level driver and developed to work in Full speed mode. It is composed of two main modules: the USB Library core and the USB Class.

The USB Library core module includes three main blocks: USB device core, USB requests and USB I/O requests. It implements the USB device library machines as defined by the revision 2.0 Universal Serial Bus Specification. This module functionalities are covered by the files under "Core" folder within the USB device library firmware package (see Figure 6).

The USB class module is the class layer built in compliance with the protocol specification.

**Figure 6. USB device library file structure**



## 5.2 USB device library description

### 5.2.1 USB device library flow

#### Handling control endpoint 0

The USB specification defines four transfer types: control, interrupt, bulk and isochronous transfers.

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 library answers requests without the intervention of the user application if the library has enough information about these requests. Otherwise, the library calls user application defined callback functions to accomplish the requests when some application actions or application information are needed. The format and the meaning of the class-specific requests and the vendor specific requests are not common for all USB devices.

The library does not handle any of the requests in these categories. Whenever the library receives a request that it does not know, the library calls a user-defined callback function and passes the request to the user application code. 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:

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

} USB_SETUP_REQ;
```

### Standard requests

All the requests specified in the following table of the USB specification are handled as standard requests in the library. The table lists all the standard requests and their valid parameters in the library. Requests that are not in this table are considered as non-standard requests.

**Table 4. Standard requests**

| | State | bmRequestT | Low byte of | High byte of | Low byte of | 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. |
| 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->dev.class_cb->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 (for example, the library does not support the Halt feature on endpoint 0 but the user application wants so).

Invalid requests are the requests that are not standard requests and are not user-interpreted requests. Since *pdev->dev.class_cb->Setup (pdev, req)* is called after the SETUP stage and before the data stage, user code is responsible, in the ***pdev->dev.class_cb->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->dev.class_cb->Setup (pdev, req)*

For a user-interpreted request, the user code then 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->dev.class_cb->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 and the data in ep_size packet.

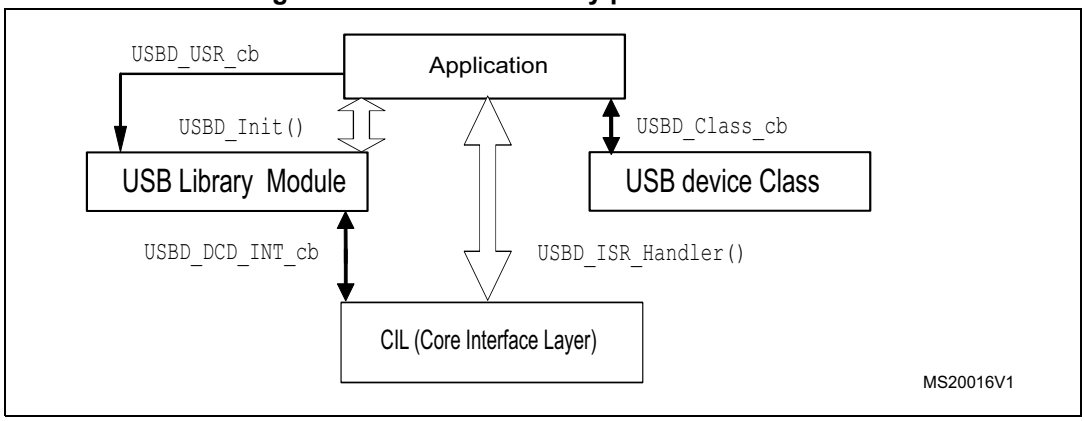
– Status stage

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

## 5.2.2 USB device library process

*Figure 7* shows the different layers interaction between the low level driver, the usb device library and the application layer.

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



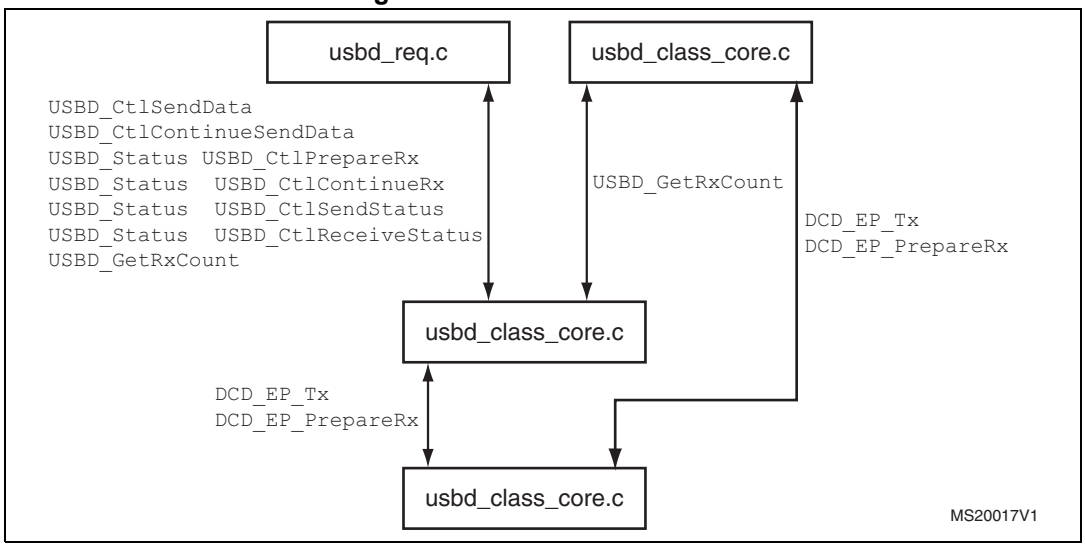

The Application layer has only to call one function (`USBD_Init`) to initialize the USB low level driver, the USB device library, the hardware on the used board (BSP) and to start the library. The `USBD_Init` function needs however the user callback structure to inform the user layer of the different library states and messages and the class callback structure to start the class interface.

The USB Low level driver can be linked to the USB device library through the `USBD_DCD_INT_cb` structure. This structure ensures a total independence between the USB device library and the low level driver; enabling the low level driver to be used by any other device library.

### 5.2.3 USB device data flow

The USB Library (USB core and USB class layer) handles the data processing on Endpoint 0 (EP0) through the IO request layer . The following figure illustrates this data flow scheme.

**Figure 8. USB device data flow**



### 5.2.4 USB device library configuration

The USB device library can be configured using the *usbd_conf.h* file (a template configuration file is available in the "*Libraries\STM32_USB_Device_Library\Core*" directory of the library).

```
#define USBD_CFG_MAX_NUM     1 : Indicates the number of configurations
#define USB_MAX_STR_DESC_SIZ 64: Indicates max string Descriptor Size

/**** USB_MSC_Class_Layer_Parameter *********/
#define MSC_IN_EP            0x81: MSC Interrupt IN endpoint 1
#define MSC_OUT_EP           0x01: MSC Interrupt OUT endpoint 1
#define MSC_MAX_PACKET       64: Maximum packet size of the Endpoint
#define MSC_MEDIA_PACKET     512 This indicates the size (512 Bytes) of the intermediary
buffer which is used to receive/send data from/to USB

/**** USB_HID Class_Layer_Parameter *********/
#define HID_IN_EP            0x81:HID Interrupt IN endpoint 1
#define HID_OUT_EP           0x01:HID Interrupt OUT endpoint 1
#define HID_IN_PACKET        4:Maximum packet size of the Endpoint (IN)
#define HID_OUT_PACKE        4:Maximum packet size of the Endpoint (OUT)
```

### 5.2.5 USB control functions

User applications can benefit from a few other USB functions included in a USB device.

#### Device reset

When the device receives a reset signal from the USB, the library resets and initializes the application on both 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 operations and puts the system in suspend state (if low power mode management is enabled in the *usb_conf.h* 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 *usb_conf.h* file).

## 5.3 USB device library functions

The ***Core*** layer contains the USB device library machines as defined by the revision 2.0 Universal Serial Bus Specification. The following table presents the USB device core files.

**Table 5. USB device core files**

| Files | 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_ioreq (.c, .h) | This file handles the results of the USB transactions. |
| usbd_conf.h | This file contains the configuration of the device: vendor ID, Product Id, Strings…etc |
| usbd_pwr.h | This file provides functions prototypes for the power management |
| usbd_def.h | This file provides general defines for the USB device library |
| usbd_usr.h | This file provides user callback function prototypes USB event management |

**Table 6. usbd_core (.c, .h) files functions**

| Functions | Description |
|---|---|
| void USBD_Init(USB_CORE_HANDLE *pdev, USBD_DEVICE *pDevice, USBD_Class_cb_TypeDef *class_cb, USBD_Usr_cb_TypeDef *usr_cb); | Initializes the device library and loads the class driver and the user call backs. |
| USBD_Status USBD_DeInit (USB_CORE_HANDLE *pdev) | Un-initializes the device library. |
| uint8_t USBD_SetupStage (USB_CORE_HANDLE *pdev) | Handles the setup stage. |
| uint8_t USBD_DataOutStage (USB_CORE_HANDLE *pdev , uint8_t epnum) | Handles the Data Out stage. |
| uint8_t USBD_DataInStage (USB_CORE_HANDLE *pdev , uint8_t epnum) | Handles the Data In stage. |
| uint8_t USBD_Reset (USB_CORE_HANDLE *pdev) | Handles the reset event. |
| uint8_t USBD_Resume (USB_CORE_HANDLE *pdev) | Handles the resume event. |
| uint8_t USBD_Suspend (USB_CORE_HANDLE *pdev) | Handles the suspend event. |
| uint8_t USBD_SOF (USB_CORE_HANDLE *pdev) | Handles the SOF event. |
| USBD_Status USBD_SetCfg (USB_CORE_HANDLE *pdev, uint8_t cfgidx) | Configures the device and starts the interface. |
| USBD_Status USBD_ClrCfg (USB_CORE_HANDLE *pdev, uint8_t cfgidx) | Clears the current configuration. |

**Table 7. usbd_ioreq (.c, .h) files functions**

| Functions | Description |
|---|---|
| `USBD_Status USBD_CtlSendData ( USB_CORE_HANDLE *pdev, uint8_t *pbuf, uint16_t len)` | Sends the data on the control pipe. |
| `USBD_Status USBD_CtlContinueSendData (USB_CORE_HANDLE *pdev, uint8_t *pbuf, uint16_t len)` | Continues sending data on the control pipe. |
| `USBD_Status USBD_CtlPrepareRx (USB_CORE_HANDLE *pdev, uint8_t *pbuf, uint16_t len)` | Prepares the core to receive data on the control pipe. |
| `USBD_Status USBD_CtlContinueRx (USB_CORE_HANDLE *pdev, uint8_t *pbuf, uint16_t len)` | Continues receiving data on the control pipe. |
| `USBD_Status USBD_CtlSendStatus (USB_CORE_HANDLE *pdev)` | Sends a zero length packet on the control pipe. |
| `USBD_Status USBD_CtlReceiveStatus (USB_CORE_HANDLE *pdev)` | Receives a zero length packet on the control pipe. |
| `uint16_t USBD_GetRxCount (USB_CORE_HANDLE *pdev , uint8_t epnum)` | Returns the received data length |

**Table 8. usbd_req (.c, .h) functions**

| Functions | Description |
|---|---|
| `void USBD_GetString(uint8_t *desc, uint8_t *unicode, uint16_t *len)` | *Converts an ASCII string into Unicode one to format a string descriptor.* |
| `static uint8_t USBD_GetLen(uint8_t *buf)` | Returns the string length. |
| `USBD_Status USBD_StdDevReq (USB_CORE_HANDLE *pdev, USB_SETUP_REQ *req)` | Handles standard USB device requests. |
| `USBD_Status USBD_StdItfReq (USB_CORE_HANDLE *pdev, USB_SETUP_REQ *req)` | Handles standard USB interface requests. |
| `USBD_Status USBD_StdEPReq (USB_CORE_HANDLE *pdev, USB_SETUP_REQ *req)` | Handles standard USB endpoint requests. |
| `static void USBD_GetDescriptor (USB_CORE_HANDLE *pdev, USB_SETUP_REQ *req)` | Handles Get Descriptor requests. |
| `static void USBD_SetAddress (USB_CORE_HANDLE *pdev, USB_SETUP_REQ *req)` | Sets new USB device address. |

**Table 8. usbd_req (.c, .h) functions (continued)**

| Functions | Description |
|---|---|
| `static void USBD_SetConfig (USB_CORE_HANDLE *pdev, USB_SETUP_REQ *req)` | Handles Set device configuration request. |
| `static void USBD_GetConfig (USB_CORE_HANDLE *pdev, USB_SETUP_REQ *req)` | Handles Get device configuration request. |
| `static void USBD_GetStatus (USB_CORE_HANDLE *pdev, USB_SETUP_REQ *req)` | Handles Get Status request. |
| `static void USBD_SetFeature (USB_CORE_HANDLE *pdev, USB_SETUP_REQ *req)` | Handles Set device feature request. |
| `static void USBD_ClrFeature (USB_CORE_HANDLE *pdev, USB_SETUP_REQ *req)` | Handles Clear device feature request. |
| `void USBD_ParseSetupRequest ( USB_CORE_HANDLE *pdev, USB_SETUP_REQ *req)` | Copies request buffer into setup structure. |
| `void USBD_CtlError ( USB_CORE_HANDLE *pdev, USB_SETUP_REQ *req)` | Handles USB Errors on the control pipe. |

## 5.4 USB device class interface

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:

```
typedef struct _Device_cb
{
  uint8_t  (*Init)          (void *pdev , uint8_t cfgidx);
  uint8_t  (*DeInit)        (void *pdev , uint8_t cfgidx);
 /* Control Endpoints*/
  uint8_t  (*Setup)         (void *pdev , USB_SETUP_REQ  *req);
  uint8_t  (*EP0_TxSent)    (void *pdev );
  uint8_t  (*EP0_RxReady)   (void *pdev);
  /* Class Specific Endpoints*/
  uint8_t  (*DataIn)        (void *pdev , uint8_t epnum);
  uint8_t  (*DataOut)       (void *pdev , uint8_t epnum);
  uint8_t  (*SOF)           (void *pdev);
  uint8_t  *(*GetConfigDescriptor)( uint8_t speed , uint16_t *length);

#ifdef USB_SUPPORT_USER_STRING_DESC
```

```
  uint8_t  *(*GetUsrStrDescriptor)( uint8_t speed ,uint8_t index,  uint16_t
*length);
#endif

} USBD_Class_cb_TypeDef;
```

- ***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 Start of frame.
- ***GetConfigDescriptor***: This callback returns the USB Configuration descriptor.
- ***GetUsrStrDescriptor***: This callback returns the user defined string descriptor.

*Note: When a callback is not used, it can be set to NULL in the callback structure.*

## 5.5 USB device user interface

The Library provides user callback structure to allow user to add special code to manage the USB events. This user structure is defined as follows:

- ***Init***: This callback is called when the device library starts up.
- ***DeviceReset***: This callback is called when the device has detected a reset event from the host.
- ***DeviceConfigured***: this callback is called when the device receives the set configuration request.
- ***DeviceSuspended***: This callback is called when the device has detected a suspend event from the host.
- ***DeviceResumed***: This callback is called when the device has detected a resume event from the host.

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

```
typedef struct _Device_TypeDef
{
  uint8_t  *(*GetDeviceDescriptor)( uint8_t speed ,
    uint16_t *length);
  uint8_t  *(*GetLangIDStrDescriptor)( uint8_t speed ,
    uint16_t *length);
  uint8_t  *(*GetManufacturerStrDescriptor)( uint8_t speed ,
    uint16_t *length);
```

```
uint8_t  *(*GetProductStrDescriptor)( uint8_t speed ,
   uint16_t *length);
uint8_t  *(*GetSerialStrDescriptor)( uint8_t speed ,
   uint16_t *length);
uint8_t  *(*GetConfigurationStrDescriptor)( uint8_t speed ,
   uint16_t *length);
uint8_t  *(*GetInterfaceStrDescriptor)( uint8_t speed ,
   uint16_t *length);
 } USBD_DEVICE, *pUSBD_DEVICE;
```

- ***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.
- ***Get_USRStringDesc***: This callback returns the user defined string descriptor.

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

## 5.6 USB device classes

The class module contains all the files related to the class implementation. It complies with the specification of the protocol built in these classes.

The table below presents the USB device class file for the MSC, HID, DFU, Audio, CDC and CCID classes.

**Table 9. USB device class files**

| Class | Files | Description |
|---|---|---|
| HID | usbd_hid_core (.c, .h) | This file contains the HID class callbacks (driver) and the configuration descriptors related to this class. |
| MSC | usbd_msc_core( .c, .h) | This file contains the MSC class callbacks (driver) and the configuration descriptors relative to this class. |
| | usbd_msc_bot (.c, .h) | This file handles the bulk only transfer protocol. |
| | usbd_msc_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_mem.h | This file contains the function prototypes of the called functions from the SCSI layer to have access to the physical media |

**Table 9. USB device class files (continued)**

| Class | Files | Description |
|---|---|---|
| DFU | usbd_dfu_core (.c,.h) | This file contains the DFU class callbacks (driver) and the configuration descriptors relative to this class. |
| | usbd_flash_if (.c,.h) | This file contains the DFU class callbacks relative to the internal Flash memory interface. |
| | usbd_mem_if_template (.c,.h) | This file provides a template driver which allows you to implement additional memory interfaces. |
| | usbd_dfu_mal.c/.h | This file provides the generic media access layer for DFU applications |
| Audio | usbd_audio_core (.c,.h) | This file contains the AUDIO class callbacks (driver) and the configuration descriptors relative to this class. |
| CDC | usbd_cdc_core (.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. |
| CCID | usbd_ccid_cmd (.c,.h) | This file provides the CCID command handling |
| | usbd_ccid_core (.c,.h) | This file contains the CCID class callbacks (driver) and the configuration descriptors relative to this class. |
| | usbd_ccid_if(.c,.h) | This file provides all the functions for USB Interface for CCID |

### 5.6.1 HID class

#### 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". This driver implements the following aspects of the specification:

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

#### HID user interface

The `USBD_HID_SendReport` can be used by the 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:

```
static uint8_t HID_Buffer [4];
USBD_HID_SendReport (&USB_FS_dev,
USBD_HID_GetPos(),
4);
static uint8_t *USBD_HID_GetPos (void)
{
```

```
HID_Buffer[0] = 0;
HID_Buffer[1] = GetXPos();;
HID_Buffer[2] = GetXPos();
HID_Buffer[3] = 0;
return HID_Buffer;
}
```

### HID core files

**Table 10. usbd_hid_core (.c, .h) files functions**

| Functions | Description |
|---|---|
| `static uint8_t USBD_HID_Init (void *pdev, uint8_t cfgidx)` | Initializes the HID interface and open the used endpoints. |
| `static uint8_t USBD_HID_DeInit (void *pdev, uint8_t cfgidx)` | Un-Initializes the HID layer and close the used endpoints. |
| `static uint8_t USBD_HID_Setup (void *pdev, USB_SETUP_REQ *req)` | Handles the HID specific requests. |
| `uint8_t USBD_HID_SendReport (USB_CORE_HANDLE *pdev, uint8_t *report, uint16_t len)` | Sends HID reports. |

## 5.6.2 Mass storage class

### 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:

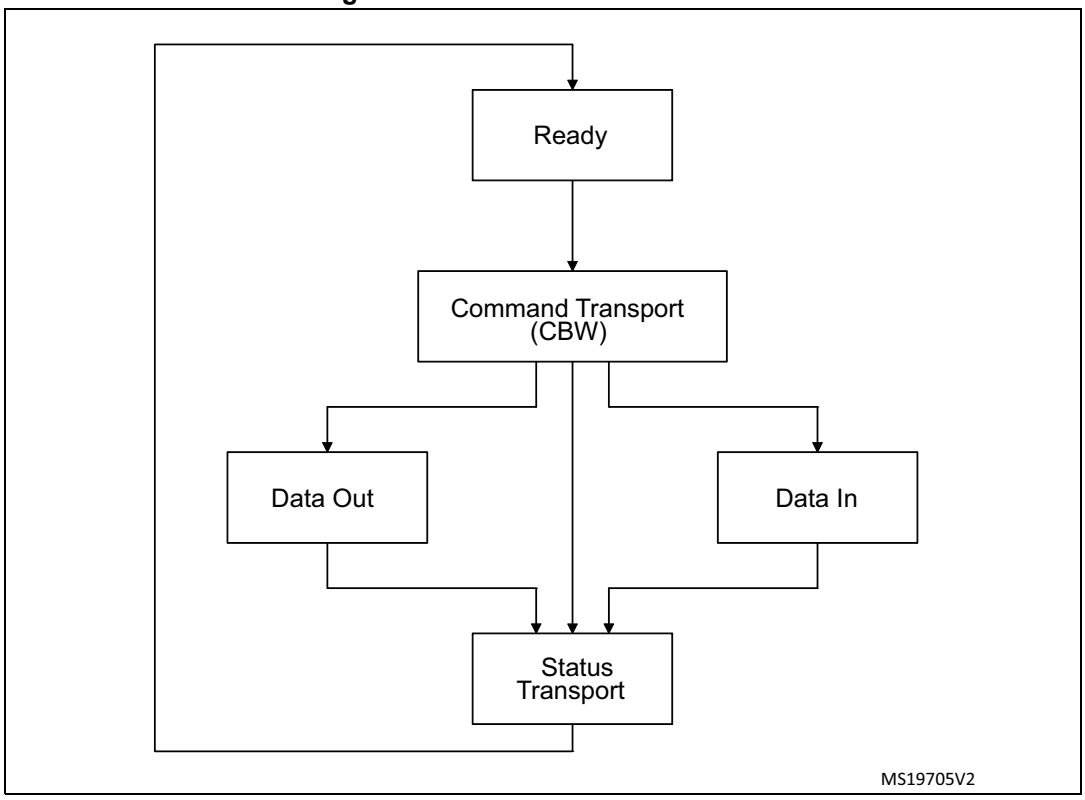
- Bulk-only transport protocol
- Subclass: SCSI transparent command set (ref. SCSI Primary Commands - 3)

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 with ready state (idle state) and if a CBW is received from the host, three cases can be managed:

- DATA-OUT-STAGE: when direction flag is set to “0”, the device must be prepared to receive an amount of data indicated in `dCBWDataTransferLength` 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 `dCBWDataTransferLength` 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 needed: the CSW block is sent immediately after the CBW one.

**Figure 9. BOT Protocol architecture**




The following table shows the supported SCSI commands

**Table 11. SCSI commands**

| Command specification | Command |
|---|---|
| SCSI | SCSI_START_STOP_UNIT,<br>SCSI_TEST_UNIT_READY,<br>SCSI_INQUIRY,<br>SCSI_READ_CAPACITY10,<br>SCSI_MODE_SENSE6,<br>SCSI_MODE_SENSE10<br>SCSI_READ10,<br>SCSI_WRITE10,<br>SCSI_VERIFY10 |

As required by the BOT specification, the following requests are implemented:

– **Bulk-only mass storage reset (class-specific request)**

This request is used to reset the mass storage device and its associated interface. This class-specific request 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'

### Get Max LUN (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'

### MSC Core files

**Table 12. usbd_msc_core (.c, .h) files functions**

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

**Table 13. usbd_msc_bot (.c, .h) files functions**

| Functions | Description |
|---|---|
| `void MSC_BOT_Init (USB_CORE_HANDLE *pdev)` | Initializes the BOT process and physical media. |
| `void MSC_BOT_Reset (USB_CORE_HANDLE *pdev)` | Resets the BOT Machine. |
| `void MSC_BOT_DeInit (USB_CORE_HANDLE *pdev)` | De-Initializes the BOT process. |

**Table 13. usbd_msc_bot (.c, .h) files functions (continued)**

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

**Table 14. usbd_msc_scsi (.c, .h) functions**

| Functions | Description |
|---|---|
| int8_t SCSI_ProcessCmd (USBCORE_HANDLE *pdev, uint8_t lun, uint8_t *params) | Processes the SCSI commands. |
| static int8_t SCSI_TestUnitReady (uint8_t lun, uint8_t *params) | Processes the SCSI Test Unit Ready command. |
| static int8_t SCSI_Inquiry (uint8_t lun, uint8_t *params) | Processes the Inquiry command. |
| static int8_t SCSI_ReadCapacity10 (uint8_t lun, uint8_t *params) | Processes the Read Capacity 10 command. |
| static int8_t SCSI_ReadFormatCapacity (uint8_t lun, uint8_t *params) | Processes the Read Format Capacity command. |
| static int8_t SCSI_ModeSense6 (uint8_t lun, uint8_t *params) | Processes the Mode Sense 6 command. |
| static int8_t SCSI_ModeSense10 (uint8_t lun, uint8_t *params) | Processes the Mode Sense 10 command. |
| static int8_t SCSI_RequestSense (uint8_t lun, uint8_t *params) | Processes the Request Sense command. |
| void SCSI_SenseCode(uint8_t lun, uint8_t sKey, uint8_t ASC) | Loads the last error code in the error list. |

Table 14. usbd_msc_scsi (.c, .h) functions (continued)

| Functions | Description |
|---|---|
| `static int8_t SCSI_StartStopUnit(uint8_t lun, uint8_t *params)` | Processes the Start Stop Unit command. |
| `static int8_t SCSI_Read10(uint8_t lun , uint8_t *params)` | Processes the Read10 command. |
| `static int8_t SCSI_Write10 (uint8_t lun , uint8_t *params)` | Processes the Write10 command. |
| `static int8_t SCSI_Verify10(uint8_t lun , uint8_t *params)` | Processes the Verify10 command. |
| `static int8_t SCSI_CheckAddressRange (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 (uint8_t lun)` | Handles the Burst Read process. |
| `static int8_t SCSI_ProcessWrite (uint8_t lun)` | Handles the Burst Write process. |

### usbd_msc_mem (.h)

This file contains the prototypes of the functions called from the SCSI layer to have access to the physical media.

## Disk operation structure definition

```
typedef struct _USBD_STORAGE
{
int8_t (* Init) (uint8_t lun);
int8_t (* GetCapacity) (uint8_t lun, uint32_t *block_num, uint32_t *block_size);
int8_t (* IsReady) (uint8_t lun);
int8_t (* IsWriteProtected) (uint8_t lun);
int8_t (* Read) (uint8_t lun, uint8_t *buf, uint32_t blk_addr, uint16_t blk_len);
int8_t (* Write)(uint8_t lun, uint8_t *buf, uint32_t blk_addr, uint16_t blk_len);
int8_t (* GetMaxLun)(void);
int8_t *pInquiry;
}USBD_STORAGE_cb_TypeDef;
```

In the media access file from user layer, once the `USBD_STORAGE_cb_TypeDef` structure is defined, it should be assigned to the `USBD_STORAGE_fops` pointer.

Example:

```
USBD_STORAGE_cb_TypeDef *USBD_STORAGE_fops = &USBD_MICRO_SDIO_fops;
```

The standard inquiry data are given by the user inside the `STORAGE_Inquirydata` array. It should be defined as:

```
const int8_t STORAGE_Inquirydata[] = {//36
```

```
/* 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', ' ', ' ', ' ',
'0', '.', '0' ,'1', /* Version : 4 Bytes */
};
```

### Disk operation functions

**Table 15. 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: `blk_address` is given in sector unit `blk_len` is the number of the sector to be processed. |

## 5.6.3 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.

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):

– Manifestation Tolerant mode

### 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.

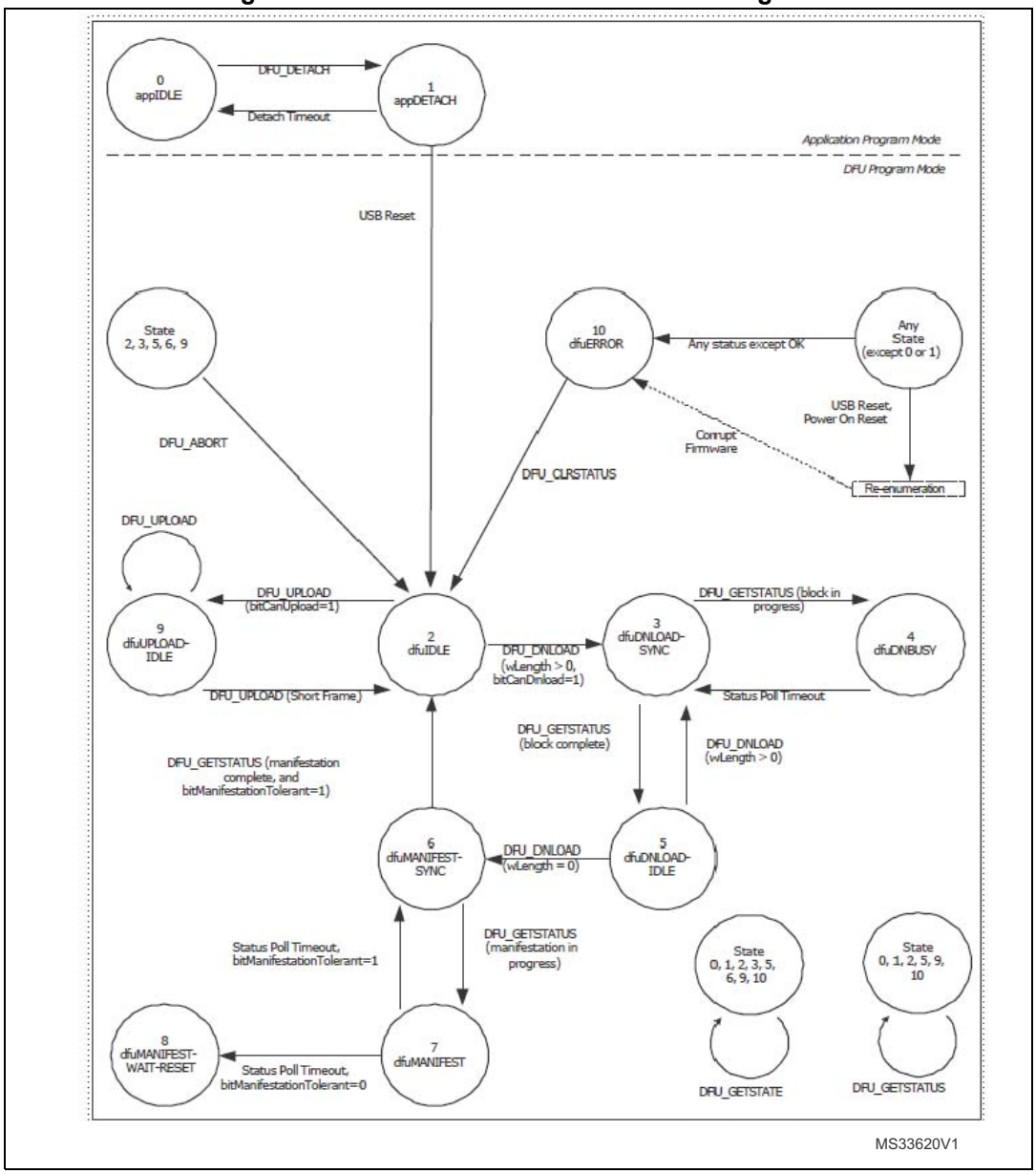
The DFU state machine is based on the following states:

**Table 16. DFU states**

| State | State code |
|---|---|
| STATE_appIDLE | 0x00 |
| STATE_appDETACH | 0x01 |
| STATE_dfuIDLE | 0x02 |
| STATE_dfuDNLOAD-SYNC | 0x03 |
| STATE_dfuDNBUSY | 0x04 |
| STATE_dfuDNLOAD-IDLE | 0x05 |
| STATE_dfuMANIFEST-SYNC | 0x06 |
| STATE_dfuMANIFEST | 0x07 |
| STATE_dfuMANIFEST-WAIT-RESET | 0x08 |
| STATE_dfuUPLOAD-IDLE | 0x09 |
| STATE_dfuERROR | 0x0A |

The allowed state transitions are described in the specification document.

**Figure 10. DFU Interface state transitions diagram**




To protect the application from spurious access before initialization, the initial state of the DFU core (after startup) is `STATE_dfuERROR`. Then, the host has to clear this state (by sending a `DFU_CLRSTATUS` request) before generating any other request.

The DFU core manages all supported requests.

**Table 17. Supported requests**

| Request | Code | Details |
|---|---|---|
| DFU_DETACH | 0x00 | When bit 3 in *bmAttributes* (bit *WillDetach*) 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 *DFU_DNLOAD* 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 *DFU_GETSTATUS* request to facilitate synchronization with the device. |
| DFU_CLRSTATUS | 0x04 | Upon receipt of *DFU_CLRSTATUS*, the device sets a status of OK and transitions to the *dfuIDLE* state. |
| DFU_GETSTATE | 0x05 | This request solicits a report about the state of the device. |
| DFU_ABORT | 0x06 | The *DFU_ABORT* request enables the host to exit from certain states and to return to the *DFU_IDLE* state. |

Each transfer to the control endpoint can be considered into 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).

### Device firmware upgrade (DFU) core files

usbd_dfu_core (.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 18. usbd_dfu_core (.c, .h) files functions**

| Functions | Description |
|---|---|
| `static uint8_t usbd_dfu_Init (void *pdev, uint8_t cfgidx)` | Initializes the DFU interface. |
| `static uint8_t usbd_dfu_DeInit (void *pdev, uint8_t cfgidx)` | De-initializes the DFU layer. |
| `static uint8_t usbd_dfu_Setup (void *pdev, USB_SETUP_REQ *req)` | Handles the DFU request parsing. |
| `static uint8_t EP0_TxSent (void *pdev)` | Handles the DFU control endpoint data IN stage. |
| `static uint8_t EP0_RxReady (void *pdev)` | Handles the DFU control endpoint data OUT stage. |

**Table 18. usbd_dfu_core (.c, .h) files functions (continued)**

| Functions | Description |
|---|---|
| `static uint8_t* Get_USRStringDesc (void *pdev, uint8_t idx)` | Manages the transfer of memory interfaces string descriptors. |
| `static void DFU_Req_DETACH (void *pdev, USB_SETUP_REQ *req)` | Handles the DFU DETACH request. |
| `static void DFU_Req_DNLOAD (void *pdev, USB_SETUP_REQ *req)` | Handles the DFU DNLOAD request. |
| `static void DFU_Req_UPLOAD (void *pdev, USB_SETUP_REQ *req)` | Handles the DFU UPLOAD request. |
| `static void DFU_Req_GETSTATUS (void *pdev)` | Handles the DFU GETSTATUS request. |
| `static void DFU_Req_CLRSTATUS (void *pdev)` | Handles the DFU CLRSTATUS request. |
| `static void DFU_Req_GETSTATE (void *pdev)` | Handles the DFU GETSTATE request. |
| `static void DFU_Req_ABORT (void *pdev)` | Handles the DFU ABORT request. |
| `static void DFU_LeaveDFUMode (void *pdev)` | Handles the sub-protocol DFU leave DFU mode request (leaves DFU mode and resets device to jump to user loaded code). |

**usbd_dfu_mal (.c, .h):**

This driver is the entry point for the memory low layer access. It allows the parsing of the memory control/access requests through the available memory (internal Flash). Depending on the address parameter, it dispatches the control/access request to the relative memory driver (or returns error code if the address is not supported).

**Table 19. usbd_dfu_mal (.c, .h) files functions**

| Functions | Description |
|---|---|
| `uint16_t MAL_Init (void)` | Calls memory interface initialization functions supported by the low layer. |
| `uint16_t MAL_DeInit (void)` | Calls memory interface de-initialization functions supported by the low layer. |
| `uint16_t MAL_Erase (uint32_t SectorAddress)` | Calls the memory interface Erase functions supported by the low layer (if Erase is not supported, this function has no effect). |
| `uint16_t MAL_Write (uint32_t SectorAddress, uint32_t DataLength)` | Calls memory interface Write functions supported by the low layer. |
| `uint8_t *MAL_Read (uint32_t SectorAddress, uint32_t DataLength)` | Calls the memory interface Read functions supported by the low layer. |

**Table 19. usbd_dfu_mal (.c, .h) files functions (continued)**

| Functions | Description |
|---|---|
| `uint16_t MAL_GetStatus(uint32_t SectorAddress ,uint8_t Cmd, uint8_t *buffer)` | Returns the low layer memory interface status. |
| `static uint8_t MAL_CheckAdd (uint32_t Add)` | Checks which memory interface supports the current address (returns error code if the address is not supported). |

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

```
typedef struct _DFU_MAL_PROP
{
const uint8_t* pStrDesc;
uint16_t (*pMAL_Init) (void);
uint16_t (*pMAL_DeInit) (void);
uint16_t (*pMAL_Erase) (uint32_t Add);
uint16_t (*pMAL_Write) (uint32_t Add, uint32_t Len);
uint8_t *(*pMAL_Read) (uint32_t Add, uint32_t Len);
uint16_t (*pMAL_CheckAdd) (uint32_t Add);
const uint32_t EraseTiming;
const uint32_t WriteTiming;
}
DFU_MAL_Prop_TypeDef;
```

Each memory interface driver should provide a structure pointer of type `DFU_MAL_Prop_TypeDef`. 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, its related field is set as NULL value.

**usbd_xxxx_if (.c, .h): (i.e. usbd_flash_if (.c,.h))**

This is the low layer driver managing the memory interface. Each memory interface should be managed by a separate low level driver (that is, *usbd_flash_if.c/.h*, *usbd_otp_if.c/.h*).

*Note: The library provides one memory driver for internal Flash memory (usbd_flash_if.c/.h), you can add other memories using the provided template file (usbd_mem_if_template_if.c/.h).*

This driver provides the structure pointer:

```
extern DFU_MAL_Prop_TypeDef DFU_Flash_cb;
extern DFU_MAL_Prop_TypeDef DFU_OTP_cb;
```

**Table 20. usbd_flash_if (.c,.h) files functions**

| Functions | Description |
|---|---|
| `const uint8_t* pStrDesc` | Pointer to the memory interface descriptor that allows the host to get memory interface organization (name, size, number of sectors/pages, size of sectors/pages, read/write rights). |
| `uint16_t (*pMAL_Init) (void)` | Handles the memory interface initialization. |
| `uint16_t (*pMAL_DeInit) (void)` | Handles the memory interface de-initialization. |
| `uint16_t (*pMAL_Erase) (uint32_t Add)` | Handles the block erase on the memory interface. |
| `uint16_t (*pMAL_Write) (uint32_t Add, uint32_t Len)` | Handles the data writing to the memory interface. |
| `uint8_t *(*pMAL_Read) (uint32_t Add, uint32_t Len)` | Handles the data reading from the memory interface. |
| `uint16_t (*pMAL_CheckAdd) (uint32_t Add)` | Returns MAL_OK result if the address is in the memory range. |
| `const uint32_t EraseTiming` | Mean time for erasing a memory block (sector/page…). It is possible to set this timing value to the maximum value allowed by the memory. |
| `const uint32_t WriteTiming` | Mean time for writing a memory block (sector/page). It is possible to set this timing value to the maximum value allowed by the memory. |

How to use the driver:

Using the file `usbd_conf.h`, you can configure:

– The number of media (memories) to be supported (define `MAX_USED_MEDIA`).
– The device string descriptors.
– The application default address (where the image code should be loaded): define `APP_DEFAULT_ADD`.

Call `usbd_dfu_Init()` function to initialize all memory interfaces and DFU state machine.

All control/request operations are performed through control endpoint 0, using the functions: `usbd_dfu_Setup()` and `EP0_TxSent()`. 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.

To close the communication, call the `usbd_dfu_DeInit()` function.

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

How to add a new memory interface:

Use the file *usbd_mem_if_template.c* as reference (modify file name, fill functions allowing to read/write/erase/get status and the mean timings for write and erase

operations in DFU_Mem_cb structure). If a functionality is not supported (i.e. Erase), fill the relative field in the DFU_MAL_Prop_TypeDef structure.

Configure the new memory string descriptor allowing to determine the memory size, number of sectors, and possibilities of read/write/erase operations on each group of sectors (MEM_IF_STRING in *usbd_mem_if_template.h*).

Configure the start and end addresses of the memory using define MEM_START_ADD and MEM_START_ADD in file usbd_mem_if_template.h.

Update the number of memory interfaces in *usbd_conf.h* file (define MAX_USED_MEDIA)

Update the file *usbd_dfu_mal.c* by:

– Including the new memory header file.
– Adding the new memory callback structure in “tMALTab” table.
– Adding the pointer to the new memory string descriptor in “usbd_dfu_StringDesc” table.

*Note: It is advised to modify the names of defines/variable/files/structures in* usbd_mem_if_template.c/.h *files for each new memory interface.*

### 5.6.4 Audio class

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

This 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. Please refer to “USB Device Class Definition for Audio Devices V1.0 Mar 18, 98" for more details.*

These aspects may 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)

### 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.

The implemented synchronization mechanism allows to overcome the difference between USB clock domain and STM32 clock domain: Clock update synchronization consists on slightly modifying the period of the timer triggering the DAC peripheral.

This solution is based on speeding-up and slowing-down the trigger clock in order to match the USB clock domain. This clock allows to keep all data samples (no data loss) but modifies slightly the audio frequency (this modification is not perceived by human ear). For this method, the distance between write pointer and read pointer is periodically measured and depending on the measured distance a correction action is performed:

- If the measured distance is higher than 3/4 buffer size, the timer **(trigger of DAC)** is speed-up

If the measured distance is lower than 1/4 buffer size, the timer (trigger of DAC) is slow-downThis 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.

*Note: This implementation is based on STM32F072B Eval Board Hardware, Which use the DAC as output Path as shown in the Figure 11*

**Figure 11. USB Audio Block Diagram**

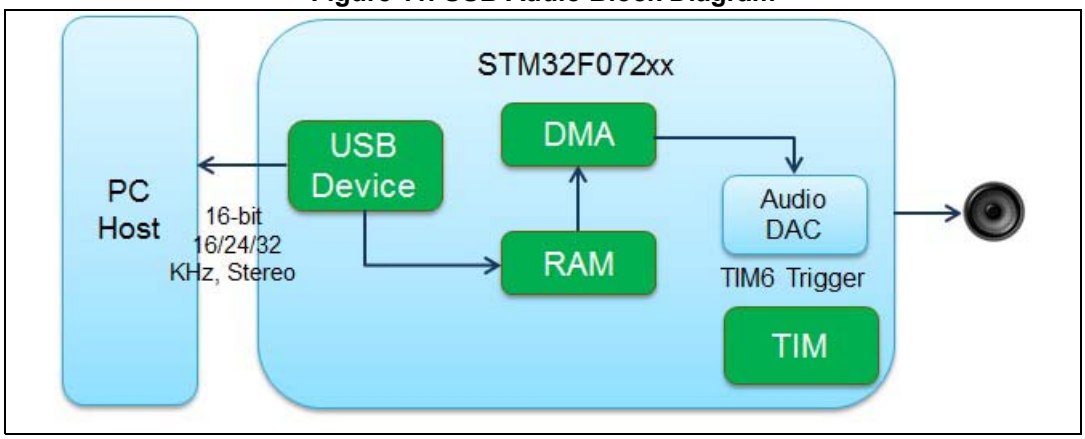


**Table 21. Audio control requests**

| 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 |

## Audio core files

*usbd_audio_core (.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 22. usbd_audio_core (.c, .h) files functions**

| Functions | Description |
|---|---|
| `static uint8_t usbd_audio_Init (void *pdev, uint8_t cfgidx)` | Initializes the Audio interface. |
| `static uint8_t usbd_audio_DeInit (void *pdev, uint8_t cfgidx)` | De-initializes the Audio interface. |
| `static uint8_t usbd_audio_Setup (void *pdev, USB_SETUP_REQ *req)` | Handles the Audio control request parsing. |
| `static uint8_t usbd_audio_EP0_RxReady(void *pdev)` | Handles audio control requests data. |
| `static uint8_t usbd_audio_DataIn (void *pdev, uint8_t epnum)` | Handles the Audio In data stage. |
| `static uint8_t usbd_audio_DataOut (void *pdev, uint8_t epnum)` | Handles the Audio Out data stage. |
| `static uint8_t usbd_audio_SOF (void *pdev)` | Handles the SOF event (data buffer update and synchronization). |
| `static void AUDIO_Req_GetCurrent(void *pdev, USB_SETUP_REQ *req)` | Handles the GET_CUR Audio control request. |

**Table 22. usbd_audio_core (.c, .h) files functions (continued)**

| Functions | Description |
|---|---|
| `static void AUDIO_Req_SetCurrent(void *pdev, USB_SETUP_REQ *req)` | Handles the `SET_CUR` Audio control request. |
| void usbd_audio_BuffXferCplt (uint8_t** pbuf, uint32_t* pSize) | Manage end of buffer transfer and Adjust DAC trigger clock |

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

```
typedef struct _Audio_Fops
{
uint8_t (*Init) (uint32_t AudioFreq, uint32_t Volume, uint32_t options);
uint8_t (*DeInit) (uint32_t options);
uint8_t (*AudioCmd) (uint8_t* pbuf, uint32_t size, uint8_t cmd);
uint8_t (*VolumeCtl) (uint8_t vol);
uint8_t (*MuteCtl) (uint8_t cmd);
uint8_t (*OptionCtrl) (uint8_t size);
uint8_t (*GetState) (void);
}AUDIO_FOPS_TypeDef;
```

Each audio hardware interface driver should provide a structure pointer of type `AUDIO_FOPS_TypeDef`. 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_xxx_if (.c, .h): (i.e. usbd_audio_out_if (.c, .h))*

This driver manages the low layer audio hardware. *usbd_audio_out_if.c/.h* driver manages the Audio Out interface (from USB to audio speaker/headphone). It calls lower layer codec driver (i.e. *stm32072b_audio_codec.c/.h*) for basic audio operations (play/pause/volume control...).

This driver provides the structure pointer:

```
extern AUDIO_FOPS_TypeDef AUDIO_OUT_fops;
```

**Table 23. usbd_audio_xxx_if (.c, .h) files functions**

| Functions | Description |
|---|---|
| `static uint8_t Init (uint32_t AudioFreq, uint32_t Volume, uint32_t options)` | Initializes the audio interface. |
| `static uint8_t DeInit (uint32_t options)` | De-initializes the audio interface and free used resources. |
| `static uint8_t AudioCmd (uint8_t* pbuf, uint32_t size, uint8_t cmd)` | Handles audio player commands (play, pause…) |
| `static uint8_t VolumeCtl (uint8_t vol)` | Handles audio player volume control. |
| `static uint8_t MuteCtl (uint8_t cmd)` | Handles audio player mute state. |

**Table 23. usbd_audio_xxx_if (.c, .h) files functions (continued)**

| Functions | Description |
|---|---|
| `static uint8_t OptionCtrl (uint8_t size)` | Switch audio digital streaming to new sampling rate |
| `static uint8_t GetState (void)` | Returns the current state of the driver audio player (Playing/Paused/Error …). |

The Audio player state is managed through the following states:

**Table 24. Audio player states**

| State | Code | Description |
|---|---|---|
| `AUDIO_STATE_INACTIVE` | 0x00 | Audio player is not initialized. |
| `AUDIO_STATE_ACTIVE` | 0x01 | Audio player is initialized and ready. |
| `AUDIO_STATE_PLAYING` | 0x02 | Audio player is currently playing. |
| `AUDIO_STATE_PAUSED` | 0x03 | Audio player is paused. |
| `AUDIO_STATE_STOPPED` | 0x04 | Audio player is stopped. |
| `AUDIO_STATE_ERROR` | 0x05 | Error occurred during initialization or while executing an audio command. |

**How to use this driver**

This driver uses an abstraction layer for hardware driver . This abstraction is performed through a lower layer (i.e. *usbd_audio_out_if.c*) which you can modify depending on the hardware available for your application.

To use this driver:

Through the file *audio_app_conf.h*, you can configure:

- The audio sampling rate (define `USBD_AUDIO_FREQ`)
- The default volume level (define `DEFAULT_VOLUME`)
- The endpoints to be used for each transfer (defines `AUDIO_OUT_EP`)
- The device string descriptors

Call the function `usbd_audio_Init()` 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_out_if.c*) and can be modified by user depending on the application needs.

The entire transfer is managed by the following functions (no need for user to call any function for out transfers):

- `usbd_audio_SOF` which synchronizes the low layer interface at each start of frame. For out transfers, at each SOF event, this function controls the low layer to stop the previous transfer if it is not stopped yet and start playing next sub-buffer. Each time the reading buffer (`IsocOutRdPtr`) is incremented.
- `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 (`IsocOutWrPtr`) is incremented.

– 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_out_if.c*). In the current version, only `SET_CUR` and `GET_CUR` requests are managed and are used for mute control only.

#### Audio known limitations

– The following situation has not been validated: When dynamic frequency switching is enabled (SUPPORTED_FREQ_NBR define in usbd_conf.h is higher than 1) and when the host uses this feature to optimize the bus usage by switch audio frequency multiple times while the audio file is playing (Windows XP SP3 and Windows 7 drivers don't use this mechanism).
– While application is playing an audio stream, if the USB cable is unplugged without stopping the audio stream and ejecting the device correctly, then a "noise" may result from this operation. This issue is due to missing management of cable disconnected event(will be fixed in next versions)

### 5.6.5 Communication device class (CDC)

This 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 may 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.

### 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:

**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).

On a periodic interval (defined through `CDC_IN_FRAME_INTERVAL` in *usbd_conf.h* file) the driver checks if there are available data in the buffer. It sends them into successive packets to the host through data IN endpoint.

**Data OUT transfer management (from host to device)**

In general, the USB is much faster than the output terminal (i.e. the USART maximum bit rate is 115.2 Kbps while USB bit rate is 12 Mbps for Full 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).

**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.

### Communication device class (CDC) core files

*usbd_cdc_core (.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 25. usbd_cdc_core (.c, .h) files functions**

| Functions | Description |
|---|---|
| `static uint8_t usbd_cdc_Init (void *pdev, uint8_t cfgidx)` | Initializes the CDC interface. |
| `static uint8_t usbd_cdc_DeInit (void *pdev, uint8_t cfgidx)` | De-initializes the CDC interface. |
| `static uint8_t usbd_cdc_Setup (void *pdev, USB_SETUP_REQ *req)` | Handles the CDC control requests. |

**Table 25. usbd_cdc_core (.c, .h) files functions (continued)**

| Functions | Description |
|---|---|
| `static uint8_t usbd_cdc_EP0_RxReady (void *pdev)` | Handles CDC control request data. |
| `static uint8_t usbd_cdc_DataIn (void *pdev, uint8_t epnum)` | Handles the CDC IN data stage. |
| `static uint8_t usbd_cdc_DataOut (void *pdev, uint8_t epnum)` | Handles the CDC Out data stage. |
| `static uint8_t usbd_cdc_SOF (void *pdev)` | Handles the SOF event (data buffer update and synchronization). |
| `static void Handle_USBAsynchXfer (void *pdev)` | Handles the IN data buffer packaging. |

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

```
typedef struct _CDC_IF_PROP
{
uint16_t (*pIf_Init) (void);
uint16_t (*pIf_DeInit) (void);
uint16_t (*pIf_Ctrl) (uint32_t Cmd, uint8_t* Buf, uint32_t Len);
uint16_t (*pIf_DataTx) (uint8_t* Buf, uint32_t Len);
uint16_t (*pIf_DataRx) (uint8_t* Buf, uint32_t Len);
}
CDC_IF_Prop_TypeDef;
```

Each hardware interface driver should provide a structure pointer of type `CDC_IF_Prop_TypeDef`. The functions pointed by this structure are listed in the following sections.

If a functionality is not supported by a given memory interface, its related 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_conf.h *file):*

**Table 26. Configurable CDC parameters**

| Define | Parameter | Typical value |
|---|---|---|
| | | Full Speed |
| `CDC_DATA_IN_PACKET_SIZE` | Size of each IN data packet | 64 |
| `CDC_DATA_OUT_PACKET_SIZE` | Size of each OUT data packet | 64 |
| `CDC_IN_FRAME_INTERVAL` | Interval time between IN packets sending. | 5 |
| `APP_RX_DATA_SIZE` | Total size of circular temporary buffer for IN data transfer. | 2048 |

**usbd_cdc_xxx_if (.c, .h): (i.e. usbd_cdc_vcp_if (.c, .h))**

This driver can be part of the user application. It is not provided in the library, but a template 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_xxx_if.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:

```
extern CDC_IF_Prop_TypeDef APP_FOPS;
```

where `APP_FOPS` should be defined in the *usbd_conf.h* file as the low layer interface structure pointer. (i.e. `#define APP_FOPS VCP_fops)` for using Virtual COM Port interface provided in the Virtual COM Port example).

**Table 27. usbd_cdc_xxx_if (.c, .h) files functions**

| Functions | Description |
|---|---|
| `uint16_t pIf_Init (void)` | Initializes the low layer CDC interface. |
| `uint16_t pIf_DeInit (void)` | De-initializes the low layer CDC interface. |
| `uint16_t pIf_Ctrl (uint32_t Cmd, uint8_t* Buf, uint32_t Len)` | Handles CDC control request parsing and execution. |
| `uint16_t pIf_DataTx (uint8_t* Buf, uint32_t Len)` | Handles CDC data transmission from low layer terminal to USB host (IN transfers). |
| `uint16_t pIf_DataRx (uint8_t* Buf, uint32_t Len)` | Handles CDC data reception from USB host to low layer terminal (OUT transfers). |

In order to accelerate data management for IN transfers, the low layer driver (*usbd_cdc_xxx_if.c/.h*) should use two global variables exported from CDC core:

**Table 28. Variables used by usbd_cdc_xxx_if.c/.h**

| Variable | Usage |
|---|---|
| `extern uint8_t APP_Rx_Buffer []` | Writes CDC received data in this buffer. These data will be sent over USB IN endpoint in the CDC core functions. |
| `extern uint32_t APP_Rx_ptr_in` | Increments this pointer or rolls it back to start the address when writing received data in the buffer APP_Rx_Buffer. |

**How to use this driver**

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

To use this driver:

Through the file *usbd_conf.h* you can configure:

– The Data IN and OUT and command packet sizes (defines `CDC_DATA_IN_PACKET_SIZE`, `CDC_DATA_OUT_PACKET_SIZE`, `CDC_CMD_PACKET_SZE`)

- The interval between IN packets (define `CDC_IN_FRAME_INTERVAL`)
- The size of the temporary circular buffer for IN data transfer (define `APP_RX_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_vcp_if.c*) and can be modified by user depending on the application needs.

CDC IN and OUT data transfers are managed by two functions:

- `APP_DataTx` (i.e. `VCP_dataTx`) 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.
- `APP_DataRx` (i.e. `VCP_dataRx`) 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).

CDC control requests should be handled by the function `APP_Ctrl` (i.e. `VCP_Ctrl`). 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.

To close the communication, call the function `usbd_cdc_DeInit()`. This closes the used endpoints and calls lower layer de-initialization functions.

### 5.6.6 CCID (Specification for Integrated Circuit(s) Cards Interface Devices)

The USB device library provide an embedded CCID firmware application that is compliant with the USB CCID Class Specification of the *USB Device Class Specification for USB Chip/Smart Card Interface Devices Specification*, Revision 1.1

The USB CCID driver implements the following aspects of the specification:

- Bulk transfers of CCID commands
- Interrupt Transfers of the CCID status

#### CCID Device Class

- CCID Endpoints

The configuration and usage of endpoints shall follow the CCID specification section 3

In addition to the default (control) endpoint, the CCID requires three endpoints to communicate with the Host Computer:

- Control Endpoint: For setup and Control purpose
- Bulk OUT: For Command to be sent from Host to STM32 (CCID)
- Bulk IN: To send responses and transfer data from the device to the host in reply to commands received on the Command Pipe
- Interrupt IN: Used by the CCID to notify the host of an insertion and removal events of the card in case of hardware errors

*Note: The data packet size for Bulk IN and Bulk OUT endpoints is 64 bytes*

*The data packet size for Interrupt IN endpoint is 8 bytes*

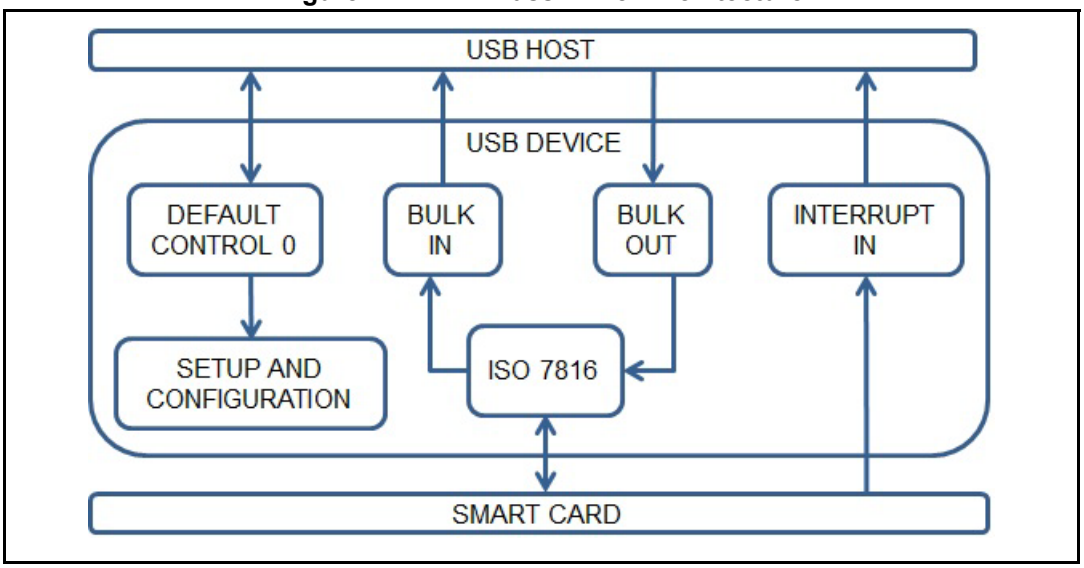
– Communication Protocol:

The control commands are sent on control pipe, these include Class-specific requests and USB standard requests. Commands that are sent on the default pipe report information back to the host on the on the default pipe.

CCID events are sent on Bulk Out endpoint. Each command sent to the STM32(CCID) has an associated ending response.

CCID response are sent on Bulk IN endpoint. All commands sent to the STM32(CCID) have to be sent synchronously.

**Figure 12. CCID Class Driver Architecture**




## USB CCID Core files

usbd_ccid_core(.c,.h)

**Table 29. usbd_ccid_core(.c,.h) files functions**

| Functions | Description |
|---|---|
| static uint8_t USBD_CCID_Init (void *pdev, uint8_t cfgidx) | Initializes the Interface. Opens the EP channels Initializes the parameters for the CCID |
| static uint8_t USBD_CCID_DeInit (void *pdev, uint8_t cfgidx) | De-Initializes the Interface. Close the EP channels De-initializes the parameters for the CCID |
| Static uint8_t USBD_CCID_Setup (void *pdev, USB_SETUP_REQ *req) | Handles the CCID specific class requests. There are three class requests defined by CCID specifications |
| static uint8_t USBD_CCID_DataIn (void *pdev, uint8_t epnum) | Handles data IN Stage. The function calls CCID_BulkMessage_In |

**Table 29. usbd_ccid_core(.c,.h) files functions (continued)**

| Functions | Description |
|---|---|
| static uint8_t USBD_CCID_DataOut (void *pdev, uint8_t epnum) | Handles data OUT Stage The function calls CCID_BulkMessage_Out |
| static uint8_t *USBD_CCID_GetCfgDesc (uint8_t speed, uint16_t *length) | Returns configuration descriptor |

usbd_ccid_if.c(.c,.h)

**Table 30. usbd_ccid_if.c(.c,.h) files functions**

| Functions | Description |
|---|---|
| `void CCID_Init (USB_CORE_HANDLE *pdev)` | Initializes the CCID USB Layer |
| `void CCID_DeInit (USB_CORE_HANDLE *pdev)` | Uninitializes the CCID Machine |
| `void CCID_BulkMessage_In (USB_CORE_HANDLE *pdev, uint8_t epnum)` | Handles Bulk IN data stage : CCID_BulkMessage_In |
| `void CCID_BulkMessage_Out (USB_CORE_HANDLE *pdev, uint8_t epnum)` | Processes CCID OUT data |
| `void CCID_CmdDecode(USB_CORE_HANDLE *pdev)` | Decodes the received Commands and call the related CCID commands |
| `void Transfer_Data_Request(uint8_t* Data_Pointer, uint16_t Data_Len)` | Prepares the request response to be sent to the host |
| `static void CCID_Response_SendData(USB_CORE_HANDLE *pdev, uint8_t* buf, uint16_t len)` | Sends the pre-filled data to the host |
| `void CCID_IntMessage(USB_CORE_HANDLE *pdev)` | Sends the Interrupt-IN datato the host |
| `void CCID_ReceiveCmdHeader(uint8_t* pDst, uint8_t u8length)` | Receives the Data from USB BulkOut Buffer to the CCID structure for easy parsing |
| `uint8_t CCID_IsIntrTransferComplete (void)` | Provides the status of previous Interrupt transfer status |
| `void CCID_SetIntrTransferStatus (uint8_t xfer_Status)` | Sets the value of the Interrupt transfer status |

usbd_ccid_cmd(.c,.h)

**Table 31. usbd_ccid_cmd(.c,.h) files functions**

| Functions | Description |
|---|---|
| `uint8_t PC_to_RDR_IccPowerOn(void)` | PC_TO_RDR_ICCPOWERON message execution, Apply the ICC VCC, Fills the Response buffer with ICC ATR |
| `uint8_t PC_to_RDR_IccPowerOff(void)` | Icc VCC is switched Off |
| `uint8_t PC_to_RDR_GetSlotStatus(void)` | Provides the Slot status to the host |
| `uint8_t PC_to_RDR_XfrBlock(void)` | Handles the Block transfer from Host. |
| `uint8_t PC_to_RDR_GetParameters(void)` | Provides the ICC parameters to the host |
| `uint8_t PC_to_RDR_ResetParameters(void)` | Sets the ICC parameters to the default |
| `uint8_t PC_to_RDR_SetParameters(void)` | Sets the ICC parameters to the host defined parameters |
| `uint8_t PC_to_RDR_Escape(void)` | Executes the Escape command. This is user specific Implementation |
| `uint8_t PC_to_RDR_IccClock(void)` | Executes the Clock specific command from host |
| `uint8_t PC_to_RDR_Abort(void)` | Executes the Abort command from host, This stops all Bulk transfers from host and ICC |
| `uint8_t CCID_CmdAbort(uint8_t slot, uint8_t seq)` | Executes the Abort command from Bulk EP or from Control EP. This stops all bulk transfers from Host and ICC |
| `void RDR_to_PC_DataBlock(uint8_t errorCode)` | Provides the data block response to the host |
| `void RDR_to_PC_SlotStatus(uint8_t errorCode)` | Provides the Slot status response to the host |
| `void RDR_to_PC_Parameters(uint8_t errorCode)` | Provides the data block response to the host |
| `void RDR_to_PC_Escape(uint8_t errorCode)` | Provides the Escape data block response to the host |
| `void RDR_to_PC_NotifySlotChange(void)` | Interrupt message to be sent to the host, Checks the card presence status and updates the buffer accordingly |
| `void CCID_UpdSlotStatus (uint8_t slotStatus)` | Updates the variable for the slot status |
| `void CCID_UpdSlotChange (uint8_t changeStatus)` | Updates the variable for the slot change status |
| `uint8_t CCID_IsSlotStatusChange (void)` | Provides the value of the variable for the slot change status |
| `static uint8_t CCID_CheckCommandParams (uint32_t param_type)` | Checks the specific parameters requested by the function and update status accordingly. This function is called from all |

### CCID Descriptors and specific requests

CCID Descriptors

– Device Descriptor:

The device descriptor of the CCID class is shown below:

```
0x12,                       /*bLength */
USB_DEVICE_DESCRIPTOR_TYPE, /*bDescriptorType*/
LOBYTE(BCD_USB_VER),        /*bcdUSB */
HIBYTE(BCD_USB_VER),
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*/
LOBYTE(USBD_BCD_DEVICE_VER), /*bcdDevice rel. 2.00*/
HIBYTE(USBD_BCD_DEVICE_VER),
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_CFG_MAX_NUM            /*bNumConfigurations*/
```

– Configuration Descriptor

```
0x09,   /* bLength: Configuration Descriptor size */
USB_DESC_TYPE_CONFIGURATION,   /* bDescriptorType: Configuration */
SMARTCARD_SIZ_CONFIG_DESC,
0x00,
0x01,   /* bNumInterfaces: 1 interface */
0x01,   /* bConfigurationValue: */
0x04,   /* iConfiguration: */
0x80, /*bmAttributes: bus powered */
0x32,   /* MaxPower 100 mA */
```

– Interface Descriptor

The interface descriptor is for the CCID Class Interface. It should indicates the Smart Card Class code (0Bh).

```
0x09,   /* bLength: Interface Descriptor size */
0x04,   /* bDescriptorType: */
0x00,   /* bInterfaceNumber: Number of Interface */
0x00,   /* bAlternateSetting: Alternate setting */
0x03,   /* bNumEndpoints: 3 endpoints used */
0x0B,   /* bInterfaceClass: user's interface for CCID */
0x00,   /* bInterfaceSubClass : */
0x00,   /* nInterfaceProtocol : None */
0x05,   /* iInterface: */
```

*Note: The bNumEndpoints field is set to 3, because a CCID shall support a minimum of **two** endpoints **one bulk-out** and **one bulk-in**, in addition to the default, control endpoint(This is not taken into account here)*

*A CCID that reports ICC insertion or removal events must also support an **interrupt** endpoint.*

– CCID Descriptor

The STM32 (CCID) supported CCID features are indicated in its Class descriptor as it is shown below:

```
0x36,   /* bLength: CCID Descriptor size */
0x21,  /* bDescriptorType: Functional Descriptor type. */
0x10,  /* bcdCCID(LSB): CCID Class Spec release number (1.00) */
0x01,   /* bcdCCID(MSB) */
0x00,  /* bMaxSlotIndex :highest available slot on this device */
0x03,  /* bVoltageSupport: bit Wise OR for 01h-5.0V 02h-3.0V 04h 1.8V*/
0x01,0x00,0x00,0x00,/* dwProtocols: 0001h = Protocol T=0 */
0x10,0x0E,0x00,0x00,/* dwDefaultClock: 3.6Mhz = 3600kHz = 0x0E10*/
0x10,0x0E,0x00,0x00,/* dwMaximum*/
0x00, /* bNumClockSupported*/
0xCD,0x25,0x00,0x00,/* dwDataRate*/
0xCD,0x25,0x00,0x00,/* dwMaxDataRate*/
0x00,/* bNumDataRatesSupported*/
0x00,0x00,0x00,0x00, /* dwMaxIFSD: 0 (T=0 only)*/
0x00,0x00,0x00,0x00,/* dwSynchProtocols  */
0x00,0x00,0x00,0x00,/* dwMechanical: no special characteristics */
0x38,0x00,EXCHANGE_LEVEL_FEATURE,0x00,/* dwFeatures*/
0x0F,0x01,0x00,0x00, /* dwMaxCCIDMessageLength*/
0x00, /* bClassGetResponse*/
0x00, /* bClassEnvelope */
0x00,0x00,/* wLcdLayout : 0000h no LCD. */
0x00, /* bPINSupport : no PIN verif and modif  */
0x01, /* bMaxCCIDBusySlots*/
```

– Endpoint Descriptor

As mentioned previously, there are bulk OUT, bulk IN and Interrupt IN endpoints

```
0x07,   /*Endpoint descriptor length = 7*/
0x05,   /*Endpoint descriptor type */
CCID_BULK_IN_EP,   /*Endpoint address (IN, address 1) */
0x02,   /*Bulk endpoint type */
LOBYTE(CCID_BULK_EPIN_SIZE),
HIBYTE(CCID_BULK_EPIN_SIZE),
0x00,   /*Polling interval in milliseconds */
0x07,   /*Endpoint descriptor length = 7 */
0x05,   /*Endpoint descriptor type */
CCID_BULK_OUT_EP,   /*Endpoint address (OUT, address 1) */
0x02,   /*Bulk endpoint type */
```

```
LOBYTE(CCID_BULK_EPOUT_SIZE),
HIBYTE(CCID_BULK_EPOUT_SIZE),
0x00,   /*Polling interval in milliseconds*/
0x07,   /*bLength: Endpoint Descriptor size*/
0x05,   /*bDescriptorType:*/
CCID_INTR_IN_EP,    /*bEndpointAddress: Endpoint Address (IN)*/
0x03,   /* bmAttributes: Interrupt endpoint */
LOBYTE(CCID_INTR_EPIN_SIZE),
HIBYTE(CCID_INTR_EPIN_SIZE),
0x18    /*Polling interval in milliseconds */
```

### Class-Specific Requests

The USB CCID supports following Class specific requests:

**Table 32. Summary of supported Class Specific Requests**

| Request | Implemented | Comments |
|---|---|---|
| ABORT | Yes | The ABORT request allows the host to abort the response portion of a command/response message pair. This may be necessary to recover from error conditions and put the CCID into a state where it can receive a new command message. |
| GET_DATA_RATES | No | CCID with bNumDataRatesSupported equal to 00h does not have to support this request |
| GET_CLOCK_FREQUENCIES | No | CCID with bNumClockSupported equal to 00h does not have to support this request. |

## 5.6.7 Adding a custom class

To create a new custom class, the user has to add `USBD_CustomClass_cb` as described in *Section 5.4*

```
typedef struct _Device_cb
{
   uint8_t  (*Init) (void *pdev , uint8_t cfgidx);
   uint8_t  (*DeInit) (void *pdev , uint8_t cfgidx);
   /* Control Endpoints*/
   uint8_t  (*Setup) (void *pdev , USB_SETUP_REQ  *req);
   uint8_t  (*EP0_TxSent)   (void *pdev );
   uint8_t  (*EP0_RxReady)  (void *pdev, uint8_t epnum );
   /* Class Specific Endpoints*/
   uint8_t  (*DataIn)       (void *pdev , uint8_t epnum);
   uint8_t  (*DataOut)      (void *pdev , uint8_t epnum);
   uint8_t  (*SOF)          (void *pdev);
```

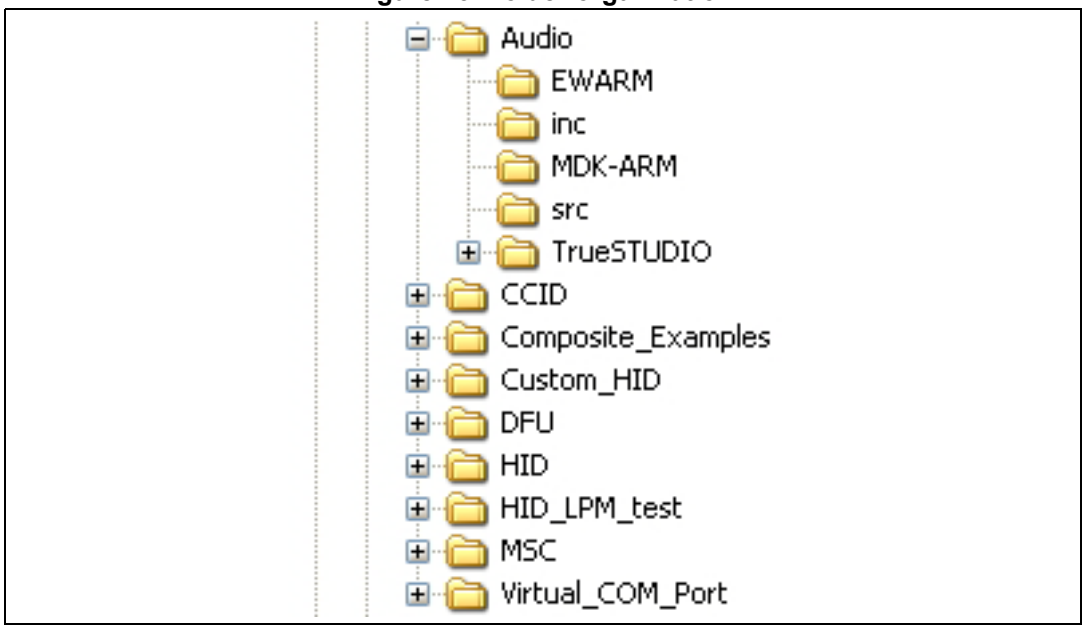
```
   uint8_t  *(*GetConfigDescriptor)( uint8_t speed ,
   uint16_t *length);
#ifdef USB_SUPPORT_USER_STRING_DESC
   uint8_t  *(*GetUsrStrDescriptor)( uint8_t speed ,
   uint8_t index,  uint16_t *length);
#endif

} USBD_Class_cb_TypeDef;
```

In the `DataIn` and `DataOut` functions, the user can implement the internal protocol or state machine, while 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.

`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*).

## 5.7 Application layer description

**Figure 13. Folder organization**




All the examples provided within the STM32F0x2 USB FS device Library firmware package are developed and validated on the STM32072B-EVAL evaluation board. For each example, the source folder is split into *src* (sources) and *inc* (includes).

The *sources* directory includes the following files:

- *app.c*: contains the main function
- *stm32_it.c*: contains the system interrupt handlers
- *system_stm32f0xx.c*: system clock configuration file for STM32F072 devices.

- *usb_bsp.c*: contains the function implementation (declared in the *usb_bsp.h* in the USB device low level driver) to initialize the GPIO for the core and interrupts enabling/disabling process.
- *usbd_usr*: contains the function implementation (declared in the *usbd_usr.h* in the USB library) to handle the library events from user layer (event messages).
- *usbd_desc.c*: This file is provided within USB Device examples and implements callback bodies. This file offers a set of functions used to change the device and string descriptors at application runtime.

The *includes* directory contains the following files:

- *stm32_it.h*: header file of the *stm32_it.c* file
- *usb_conf.h*: configuration files for the USB device low level driver
- *usbd_conf.h*: configuration files for the USB device library
- *usbd_pwr.c:* this file provides API for power management

The user should use CN4 connector of the STM32072B-EVAL to connect the board to a PC host through USB cable

## 5.8 Starting the USB library

The USB Library is built as an interrupt model; from application layer the user has only to call the *USBD_Init ()* function and pass the user and class callbacks. The USB internal process is handled internally by the USB library and triggered by the USB interrupts from the USB driver.

**Figure 14. USBD_Initf unction example**

```
109
110   USBD_Init(&USB_Device_dev,
111             &USR_desc,
112             &USBD_HID_cb,
113             &USR_cb);
114
115   while (1)
116   {
117   }
```

## 5.9 USB examples

### 5.9.1 USB mass storage example

The Mass storage example uses the microSD Flash embedded in the STM32072B-EVAL evaluation board as media for data storage.

In addition to the source files mentioned above, additional files for the disk access were added to handle the microSD driver and microSD access operations.

The mass storage example has the following USB device information (*usbd_desc.c*).

```
#define USBD_VID                    0x0483
#define USBD_PID                    0x5720
#define USBD_LANGID_STRING          0x409
```

```
#define USBD_MANUFACTURER_STRING      "STMicroelectronics"
#define USBD_PRODUCT_FS_STRING        "Mass Storage in FS Mode"
#define USBD_CONFIGURATION_FS_STRING  "MSC Config"
#define USBD_INTERFACE_FS_STRING      "MSC Interface"
```

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

### 5.9.2 USB human interface example

The HID example uses the joystick embedded in the STM32072B-EVAL evaluation board.

The HID example works in Full speed modes and has the following USB device information (*usbd_desc.c*).

```
#define USBD_VID                      0x0483
#define USBD_PID                      0x5710
#define USBD_LANGID_STRING            0x409
#define USBD_MANUFACTURER_STRING      "STMicroelectronics"
#define USBD_PRODUCT_FS_STRING        "Joystick in FS Mode"
#define USBD_CONFIGURATION_FS_STRING  "HID Config"
#define USBD_INTERFACE_FS_STRING      "HID Interface"
```

The user can use the embedded joystick on the evaluation board to move the mouse pointer on the host screen.

*Note: The Low power mode is enabled, allowing entering the core into Low power mode by the USB Suspend event, the core wakes up when the USB wakeup event is received on the USB. The HID example supports also the remote wakeup feature allowing the device to wake up the host by pressing the [Tamper] button on the evaluation board.*

### 5.9.3 USB firmware upgrade example

The DFU example allows a device firmware upgrade using the DFU drivers provided by ST (ST DFUse and ST DFU Tester) available for download from www.st.com.

Refer to the UM0412, DfuSe USB device firmware upgrade STMicroelectronics extension,

for more details on the driver installation and PC user interface.

The Internal flash memory is the only supported memory for this example

The DFU example has the following USB device information.

```
#define USBD_VID                      0x0483
#define USBD_PID                      0xDF11
#define USBD_LANGID_STRING            0x409
#define USBD_MANUFACTURER_STRING      "STMicroelectronics"
#define USBD_PRODUCT_FS_STRING        "DFU in FS Mode"
#define USBD_CONFIGURATION_FS_STRING  "DFU Config"
```

```
#define USBD_INTERFACE_FS_STRING        "DFU Interface"
```

When the DFU application starts, the default state is DFU ERROR in order to prevent spurious access to the application before it is correctly configured. Once the application is running, the state is updated depending on the current operation.

After downloading a DFU image into the internal Flash and exiting from DFU mode (using command “Leave DFU mode” of the DfuSe applet), a hardware reset may be performed (using RESET button on the evaluation board). After reset, the DFU example jumps and executes the loaded user application in the internal Flash memory.

To go back to the DFU example, you have to reset the device (using RESET button or software reset) while the Tamper button is pushed. If the Tamper button is released after reset, the example jumps to user image application loaded in the internal Flash.

### 5.9.4 USB virtual com port (VCP) example

The VCP example illustrates an implementation of the CDC class following the PSTN sub-protocol. The VCP example allows the STM32 device to behave as a USB-to-RS232 bridge.

- On one side, the STM32 communicates with host (PC) through USB interface in Device mode.
- On the other side, the STM32 communicates with other devices (same host, other host, other devices…) through the USART interface (RS232).

The support of the VCP interface is managed through the ST Virtual Com Port driver available for download from www.st.com.

This example can be customized to communicate with interfaces other than USART.

The VCP example has the following USB device information.

```
#define USBD_VID                        0x0483
#define USBD_PID                        0x5740
#define USBD_LANGID_STRING              0x409
#define USBD_MANUFACTURER_STRING        "STMicroelectronics"
#define USBD_PRODUCT_FS_STRING          "STM32 Virtual ComPort  in FS Mode"
#define USBD_CONFIGURATION_FS_STRING    "VCP Config"
#define USBD_INTERFACE_FS_STRING        "VCP Interface"
```

When the VCP application starts, the USB device is enumerated as serial communication port and can be configured in the same way (baudrate, data format, parity, stop bit length…).

To test this example, you can use one of the following configurations:

- **Configuration 1**: Connect USB cable to host and USART (RS232) to a different host (PC or other device) or to the same host. In this case, you can open two hyperterminal-like terminals to send/receive data to/from host to/from device.
- **Configuration 2**: Connect USB cable to Host and connect USART TX pin to USART RX pin on the evaluation board (Loopback mode). In this case, you can open one terminal (relative to USB com port or USART com port) and all data sent from this terminal will be received by the same terminal in Loopback mode. This mode is useful for test and performance measurements.

Figure 15. Configuration 1a: Two different hosts for USB and USART

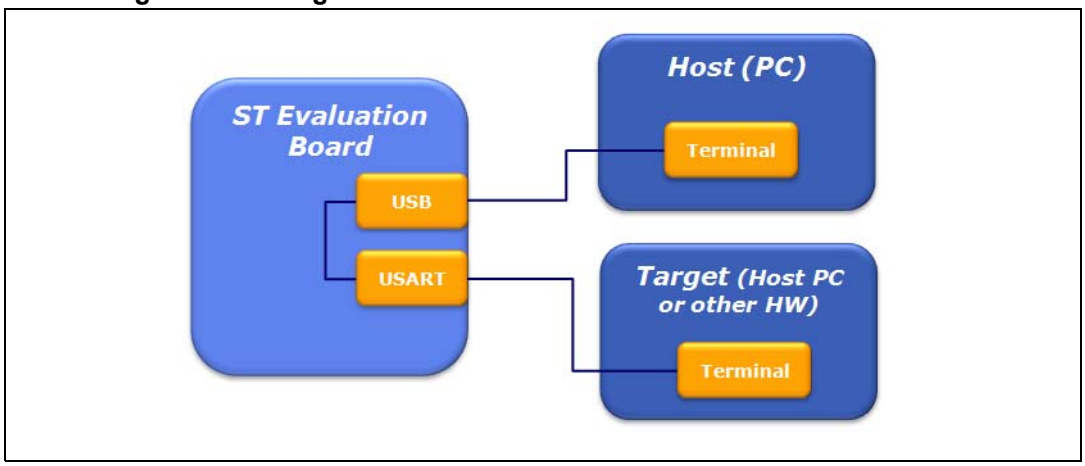



Figure 16. Configuration 1b: One single Host for USB and USART

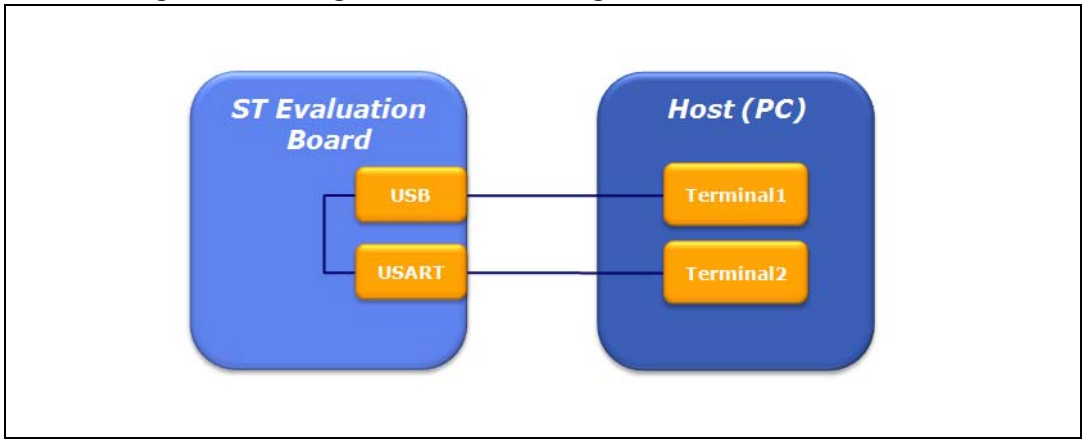
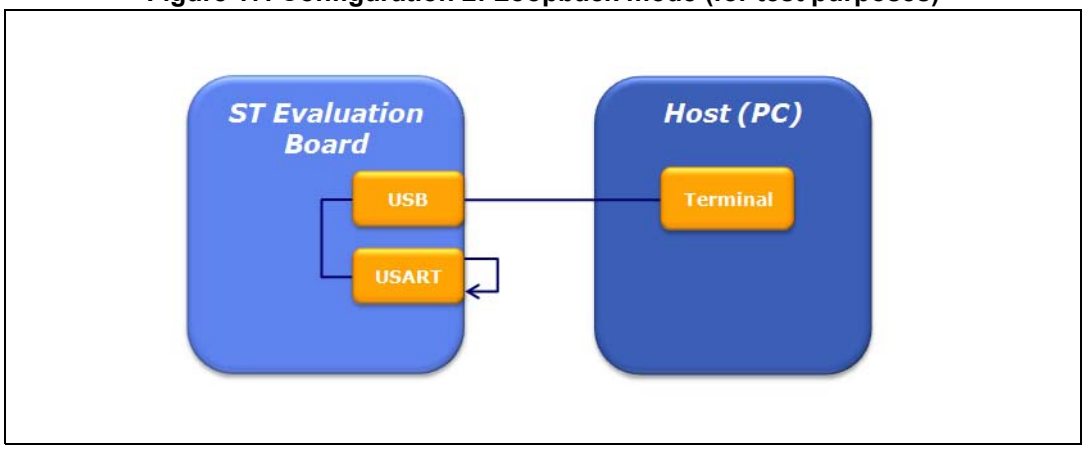



Figure 17. Configuration 2: Loopback mode (for test purposes)




### 5.9.5 USB audio example

The Audio device example allows device to communicate with host (PC) as USB Speaker using isochronous pipe for audio data transfer along with some control commands (i.e. Mute).

The Audio device is natively supported by most of operating systems (there is no need for specific driver setup).

The Audio device example has the following USB device information.

```
#define USBD_VID                          0x0483
#define USBD_PID                          0x5730
#define USBD_LANGID_STRING                0x409
#define USBD_MANUFACTURER_STRING          "STMicroelectronics"
#define USBD_PRODUCT_FS_STRING            "STM32 AUDIO Streaming in FS Mode"
#define USBD_CONFIGURATION_FS_STRING      "AUDIO Config"
#define USBD_INTERFACE_FS_STRING          "AUDIO Interface"
```

The Audio example uses the DAC interface to stream audio data from USB to the audio amplifier implemented on the evaluation board. The audio downstream is driven with 48 kHz sampling rate, a 16-bit depth, and 2 channels (stereo) to headset. It supports single audio frequency for the output: the host PC driver manages the sampling rate conversion from original audio file sampling rate to the sampling rate supported by the example. It is advised to set high audio frequencies to guarantee high audio quality. It is also possible to modify the default volume through define DEFAULT_VOLUME in file *audio_app_conf.h*.

This example provides a synchronization mechanism allowing to overcome the difference between USB clock domain and STM32 clock domain:

The Clock update synchronization consists on slightly modifying the period of the timer triggering the DAC peripheral. This solution is based on speeding-up and slowing-down the trigger clock in order to match the USB clock domain.

This clock allows to keep all data samples (no data loss) but modifies slightly the audio frequency (this modification is not perceived by human ear).

For this method, the distance between write pointer and read pointer is periodically measured and depending on the measured distance a correction action is performed:

– If the measured distance is higher than 3/4 buffer size, the timer (trigger of DAC) is speed-up
– If the measured distance is lower than 1/4 buffer size, the timer (trigger of DAC) is slow-down

### 5.9.6 USB CCID example

The CCID example illustrates an implementation of the CCID class following the USB-CCID “Specification for Integrated Circuit(s) Cards Interface Revision 1.1”

The USB CCID is built around the Smart-Card Integrated Circuit(s) Card Interface Devices specifications. The driver implements the following aspects of the specification:

– Bulk transfers of CCID commands
– Interrupt Transfers of the CCID status

#### Driver Selection and Installation:

When the STM32072B-EVAL evaluation board containing the CCID firmware application is plugged into the USB port of a host PC (Windows), the STM32 device appears as a smart card reader device using the windows native CCID driver. For Windows XP users, a windows update may be needed in order to download the CCID driver "usbccid.sys".

The features provided by Microsoft's usbccid.sys are fully detailed in Microsoft Class Drivers for USB CCID Smart Cards (http://www.microsoft.com/whdc/device/input/smartcard/USB_CCID.mspx).

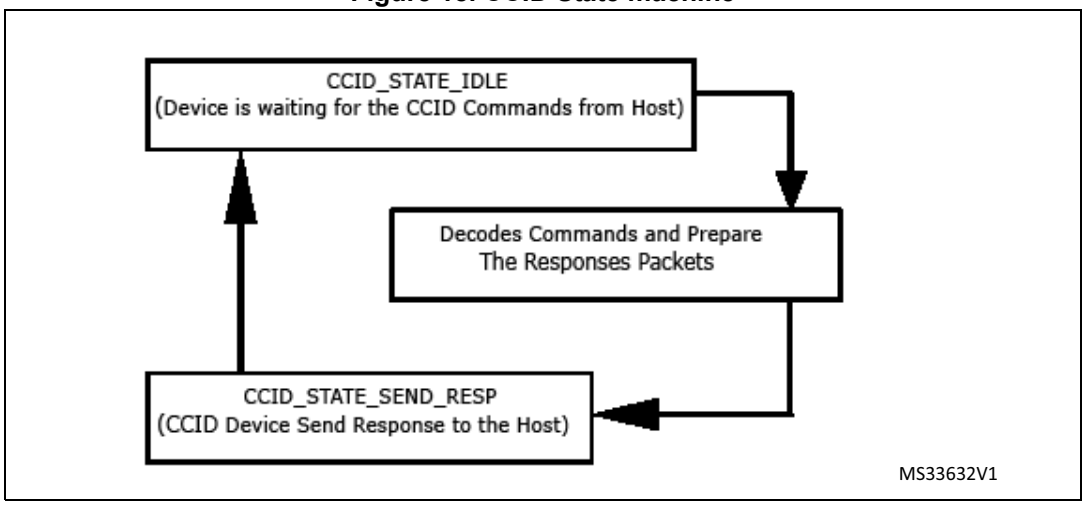
### Commands Description:

The CCID commands are in specified format of 10 bytes. Additionally data transfer may take place depending on commands and their responses. A general BOT transaction is based on a simple basic state machine: it begins with ready state `(CCID_STATE_IDLE)` and if a Command is received from the host, the commands are decoded and handled by CCID layers.

When the commands are received, the endpoint (EP) Interrupt calls the appropriate function for BulkOut Transfer. The state machine in CCID decodes the commands. Each command is then responded with a Bulk-In response.

Most of the Commands are short commands. These commands do not have 2nd stage of the data transfer. The commands are directly responded with the response. The State machine of these commands are shown below:

**Figure 18. CCID State machine**




The Bulk-out/in messages are managed by a state machine. When the Bulk-out message reception is completed, the message function is executed and returns only after the completion of the action to be done by the USART. The response is sent to the host by Bulk-in message. If there are any interrupt messages, they are sent just before the Bulk-in message is sent.

CCID Command Pipe Bulk IN/OUT Messages

The STM32 (CCID) shall follow the CCID Bulk OUT Messages as specified in CCID Rev 1.1 Section 6.1. This CCID part uses a 271-byte buffer for all messages. Messages are composed of two parts:

- header (10 bytes: fixed size)
- data (up to 261 bytes).

The `CCID_BulkMessage_Out()` function uses several Bulk-out USB transactions to verify the message header and store the entire message in the buffer. This function manages the state machine during USB Bulk Out message reception. It fills the message buffer with a maximum of 271 bytes.

The Bulk IN messages are used in response to the Bulk OUT messages.

The `CCID_BulkMessage_In()` function handleBulk IN and Interrupt IN data stage.

This section lists the CCID Bulk IN/OUT message to be supported by the STM32 (CCID):

PC_to_RDR_IccPowerOn

This command activates the card slot and return ATR data from the card. The response to this command message is the `RDR_to_PC_DataBlock` response message and the data returned is the the Answer To Reset (ATR) data

PC_to_RDR_IccPowerOff

This command desactivates the card slot. The response to this message is the `RDR_to_PC_SlotStatus`.

PC_TO_RDR_GETSLOTSTATUS

This command gets the current status of the slot. The response to this message is the `RDR_to_PC_SlotStatus`

PC_TO_RDR_GETPARAMETERS:

This command gets the slot parameters. The respnse to this message is the `RDR_to_PC_Parameters`

PC_TO_RDR_RESETPARAMETERS

This command resets the slot parameters to the default value.The respnse to this message is the `RDR_to_PC_Parameters`

PC_TO_RDR_SETPARAMETERS

This command sets the slot parameters The respnse to this message is the `RDR_to_PC_Parameters`

PC_TO_RDR_ESCAPE

This command allows the CCID manufacturer to define and access extended features. Information sent via this command is processed by the CCID control logic. The respnse to this message is the `RDR_to_PC_Escape`

PC_TO_RDR_ICCCLOCK

This command stops or restarts the clock. The respnse to this message is the `RDR_to_PC_SlotStatus`

PC_TO_RDR_ABORT

This command is used with the Control pipe Abort request to tell the CCID to stop any current transfer at the specified slot and return to a state where the slot is ready to accept a new command pipe Bulk-OUT message.The respnse to this message is the `RDR_to_PC_SlotStatus`

PC_TO_RDR_T0APDU

This command changes the parameters used to perform the transportation of APDU messages by the T=0 protocol.The respnse to this message is the `RDR_to_PC_SlotStatus`

PC_TO_RDR_MECHANICAL

This command is used to manage motorized type CCID functionality. The Lock Card function is used to hold the ICC. This prevents an ICC from being easily removed from the CCID. The Unlock Card function is used to remove the hold initiated by the Lock Card function. The respnse to this message is the `RDR_to_PC_SlotStatus`PC_TO_RDR_SETDATARATEANDCLOCKFREQUENCY

This command is used to manually set the data rate and clock frequency of a specific slot. The respnse to this message is the `RDR_to_PC_DataRateAndClockFrequency`

PC_TO_RDR_SECURE

This command is used to manually set the data rate and clock frequency of a specific slot.This is a command message to allow entering the PIN for verification or modification. The respnse to this message is the `RDR_to_PC_DataBlock`

### 5.9.7 USB Composite examples

The USB Composite Device is a general way to integrate two or more functions into one single device. It is defined in the USB Specification Revision 2.0, as "A device that has multiple interfaces controlled independently of each other".

Intefaces:

Two kinds of composite are described: Single (MSC+HID) and Multi (HID+CDC) interface functions. There are classes like CDC or audio, which consists of multiple interfaces,for these devices an additional Interface Association Descriptor (IAD) should be used.

The interfaces descriptors for each functions are merged in one interface descriptor for the composite.

For example in the MSC+HID composite demo *usbd_hid_msc_wrapper* was created by combining

- Both HID and MSC Interfaces and endpoints descriptors
- Handlers of class-specific requests for these classes

```
USBD_Class_cb_TypeDef  USBD_HID_MSC_cb =
{
  USBD_HID_MSC_Init,
  USBD_HID_MSC_DeInit,
  USBD_HID_MSC_Setup,
  NULL,
  USBD_HID_MSC_EP0_RxReady,
  USBD_HID_MSC_DataIn,
  USBD_HID_MSC_DataOut,
  NULL,
  USBD_HID_MSC_GetConfigDescriptor,
};
```

*Figure 19* shows the device descriptor of a composite device with single interface function and *Figure 20* shows the device descriptor of a composite device with Multi interfaces function.

*Note: As specified by the Universal Serial Bus Specification bDeviceClass, bDeviceSub-Class and bDeviceProtocol set to zero for a device without multi-interface function; bDeviceClass,*

*bDeviceSub-Class and bDeviceProtocolset to EFH, 02H, 01H for a device with multiinterface function inside*

**Figure 19. Device descriptor of a composite device with single interface function**

```
/* USB Standard Device Descriptor */
const uint8_t USBD_DeviceDesc[USB_SIZ_DEVICE_DESC] =
{
  0x12,                       /*bLength */
  USB_DEVICE_DESCRIPTOR_TYPE, /*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_CFG_MAX_NUM            /*bNumConfigurations*/
} ; /* USB_DeviceDescriptor */
```

**Figure 20. Device descriptor of a composite device with single interface function**

```
/* USB Standard Device Descriptor */
const uint8_t USBD_DeviceDesc[USB_SIZ_DEVICE_DESC] =
{
  0x12,                       /*bLength */
  USB_DEVICE_DESCRIPTOR_TYPE, /*bDescriptorType*/
  0x00,                       /*bcdUSB */
  0x02,
  0xEF,                       /*bDeviceClass*/
  0x02,                       /*bDeviceSubClass*/
  0x01,                       /*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_CFG_MAX_NUM            /*bNumConfigurations*/
} ; /* USB_DeviceDescriptor */
```

Endpoints:

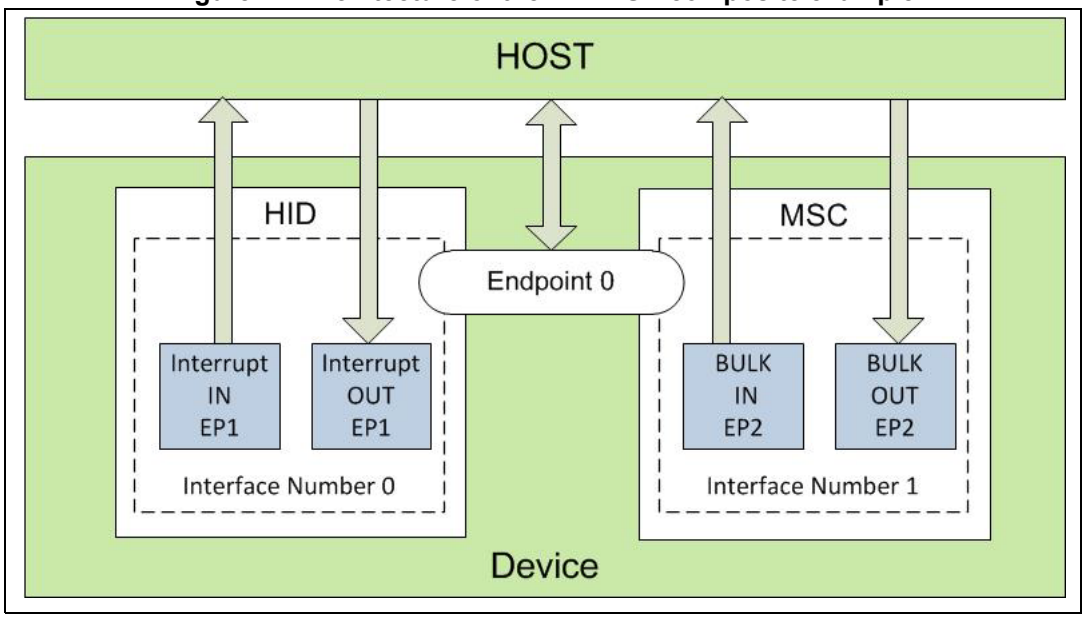
Excluding the default control endpoint (EP0), the composite device should define a number of endpoints equal to the sum of the number of endpoints required for each individual function.

For more details on composite devices, please refer to “usb_20.pdf 5.2.3”, which is available on the usb.org website

### Mass Storage-HID composite example

This example was created by combining the code in the Custom HID and USB MSC example projects. Starting from the Custom HID example, a new interface and Endpoint (EP2) descriptor were added for mass storage, and the total length in the configuration descriptor was modified. The block diagram in *Figure 21* shows the architecture of the HID MSC composite example.

**Figure 21. Architecture of the HID MSC composite example**




### CDC-HID composite example

Interface association descriptor

the CDC Class uses two different interfaces, each with one endpoint. That's why the CDC interfaces are part of a composite device class.

Interface 0 is the device management interface (communication). It provides the host with a mechanism to control the device and to receive notification of events.

Interface 1 is the data interface. It provides the data transfer mechanism for the virtual UART.

Endpoint 2 (`CDC_CMD_EP`) belongs to Interface 0. It is a USB IN endpoint (Interrupt), transmitting notifications to the host.

Endpoint 3 (`CDC_IN_EP/CDC_OUT_EP`) belongs to Interface 1. It is used in both directions as an IN and OUT endpoint (Bulk).

The interface association descriptor (IAD) is used to report that interfaces 0 and 1use the same function (CDC) to the host.

**Figure 22. Standard Interface Association Descriptor**

```
139
140  /******** IAD should be positioned just before the CDC interfaces ******
141                IAD to associate the two CDC interfaces */
142
143  0x08, /* bLength */
144  0x0B, /* bDescriptorType */
145  0x01, /* bFirstInterface */
146  0x02, /* bInterfaceCount */
147  0x02, /* bFunctionClass */
148  0x02, /* bFunctionSubClass */
149  0x01, /* bFunctionProtocol */
150  0x00, /* iFunction (Index of string descriptor describing this function) */
151
```

For more details about the IAD descriptor please refer to:

http://www.usb.org/developers/whitepapers/iadclasscode_r10.pdf

Windows Driver Files

To run this example, you will need to modify the following lines in the .inf file of the cdc driver: *stmcdc.inf* (you will find it in "C:\Program Files\STMicroelectronics\Software\Virtual COM Port Driver).

The stmcdc.inf exists to assist in the loading of the proper USB serial drivers in the Windows OSs for the composite device to operate correctly as a virtual COM interface.

The following Lines associates the CDC device with the OS's usbser.sys driver file

and causes the OS to load this driver during the enumeration process.

The "MI" stands for Multiple Interface.

;VID/PID Settings

;------------------------------------------------------------------------------

[DeviceList.NT]

%DESCRIPTION%=DriverInstall,USB\VID_0483&PID_3256&MI_01

[DeviceList.NTamd64]

%DESCRIPTION%=DriverInstall,USB\VID_0483&PID_3256&MI_01

*To avoid USB devices with the same VID&PID (Which may cause conflict on Windows) each Composite example (MSC+HID) or (CDC+HID)uses a different PID (VID stays unchanged)*

.

### 5.9.8 Custom HID example

This demo uses the HID(human interface device) class for general purpose I/O operations. Typically, the HID class is used to implement human interface products, such as standard mouse devices, keyboards, Bluetooth adaptors etc.

The HID protocol is however quite flexible, and can be adapted and used to send/receive general purpose data to/from a USB device. HID Input/Output reports can be exchanged

over both the interrupt endpoints, and over the default endpoint (Get_/Set_Report requests). The custom HID demo is a simple HID demo provided with a small PC applet to give an example of how to create a customized HID based on the native Windows HID driver. It consists of simple data exchanges between the STM32 evaluation board and the PC Host using two interrupt pipes (IN and OUT).

The Interrupt IN and OUT pipes are used for sending asynchronous data to the host, and to receive low-latency information.

All data transferred must be formatted as reports which structure is defined in the report descriptor. Reports are discussed in detail later

The custom HID demo implements feature request handling, which allows the user to send a control command to the device. This command is sent through endpoint 0, and must be treated as a set_report request. For more details on the HID device class, please refer to the “Device Class Definition for HID 1.11” available from the usb.org website. The data exchanged is related to LED commands, push-button state reports and ADC conversion values.

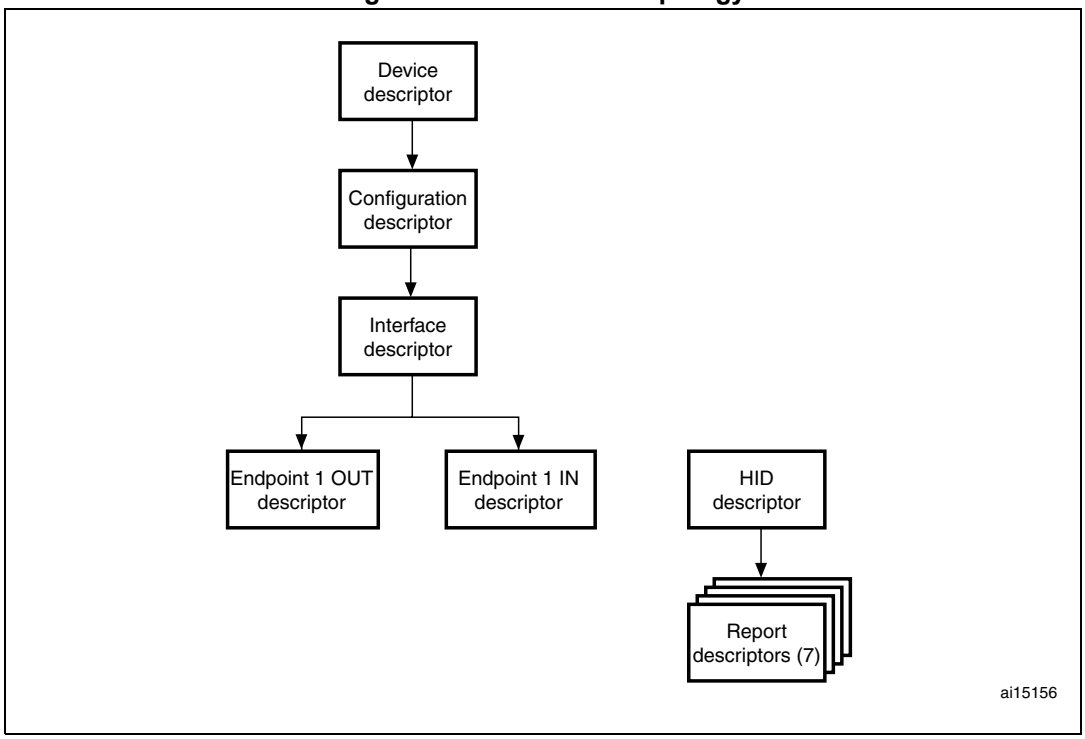
In order to begin sending/receiving packets to the device, you must first plug in the USB device, then run the USB HID demonstrator. As configured by default, the application is looking for HID class USB devices with VID = 0x0483 and PID = 0x5750

For more details on how to use the PC applet of the custom HID, please refer to the UM0551 user manual “USB HID demonstrator” available from the STMicroelectronics microcontroller website www.st.com.

Report Descriptors

All data transferred to and from an HID device must be structured in the form of reports. The report descriptor defines the report structure, which contains all the information a host needs to determine the data format and how the data should be processed. See next Figure for a report descriptor topology:

**Figure 23. Custom HID topology**




Each report descriptor is related to a specific component in the evaluation board (LEDs, Push-buttons or ADC). The following section describes the functionality of these reports.

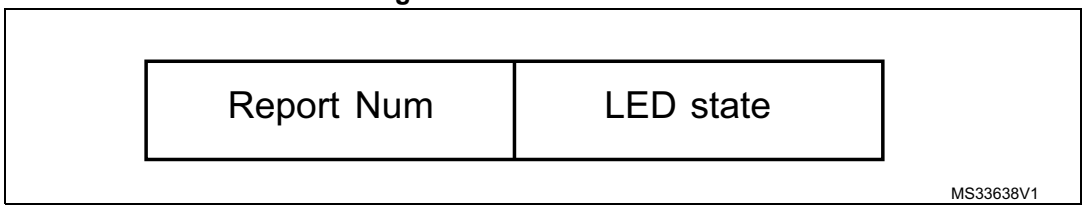
LED control

The STM32F072B evaluation board have four LEDs. In this demo each LED corresponds to a specific report (reports 1 to 4), and the LED states (ON/OFF) are set by the PC applet. Reports generated by the host to the device are transmitted through either the interrupt (OUT) endpoint or the default endpoint (Control) using the *Set_Report* request. In the PC applet, the output mode is set by default to *SET_REPORT*, and interrupt transfer is applied.When the device receives data on endpoint 1 OUT, the USBD_HID_DataOut() function is called to dispatch the received state to the corresponding LED according to the report number. When switching to the *SET_FEATURE* mode, control transfer is applied. The USBD_HID_EP0_RxReady() function is called, and the host initiates a control endpoint transfer, which causes IN and OUT reports to be sent and received. Report_Buf[] contains both the report and the number of bytes to transmit. The data received has the format shown in Figure 5, where:

– Report Num: report number from 1 to 4.
– LED state:

a) 0 -> LED off
b) 1 -> LED on

**Figure 24. Data OUT format**

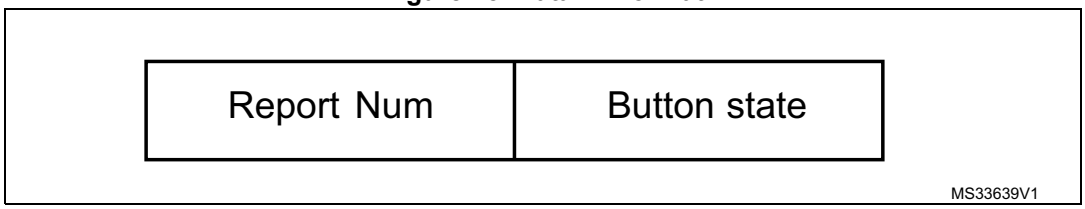



Tamper button state report

The state of the Tamper button on the STM32F072B evaluation board is reported to the PC host using the endpoint 1 IN. The tamper button corresponds to Report 6. When the tamper button is pressed, the device sends the related report number and the tamper button state to the host. Figure 6 shows the used format, where:

– Report Num: report number 6
– Button state: 1 -> button pressed

**Figure 25. Data IN Format**




ADC-converted data transfer

This part of the demo consists in transferring the result of the converted voltage connected to the potentiometer of the evaluation board to the PC host. The ADC is configured in continuous mode with DMA data transfer to a RAM variable (ADC_ConvertedValueX). After each conversion the converted value is tested against an old one (ADC_ConvertedValueX_1) and if there is a difference between the two values (potentiometer value changed by a user), the new value is sent to the PC using the endpoint 1 IN.

*Note: The data format is the same as the one used for the tamper button, but the report number (7) is followed by the MSB of the ADC conversion result.*

# 6 Frequently-asked questions

1. How to define the number of endpoints to be used?

   The `usb_conf.h` file is used to define the number of endpoints to be used (through the define `EP_NUM`: Which take the maximum used endpoint number + 1 (default endpoint EP0) for example in the MSC demo we are using two endpoints: EP1 for MSC IN and EP2 for MSC OUT, in this case `EP_NUM = 2 + 1=3`.

   You can add also a new endpoint through this file.

2. How to set the buffer configuration for endpoints?

   The usb_conf.h file is used also to configure the BTABLE and all endpoint addresses in the PMA (by modifying and/or adding relative address defines: BTABLE_ADDRESS, ENDP0_RXADDR,ENDP0_TXADDR ...).

   Packet memory area (PMA) is used to store buffer description table. Depending on number and type of used endpoints you need to reserve enough space. Please refer to the *RM0091 "STM32F0x1/STM32F0x2/STM32F0x8 advanced ARM-based 32-bit MCUs"* Reference manual available from STMicroelectronics website www.st.com.

3. How to select the USB Clock source?

   The USB clock can be from one of the below two sources:

   - The system clock : PLL and Quartz (Native)

   - The HSI 48MHz RC oscillator: Special Clock Recovery circuitry dedicated to provide an high precision reference clock.

   In the `usb_conf.h` the CRS is used as default USB clock source, through these defines:#define USB_CLOCK_SOURCE_CRS (Default).

   If you need to use PLL and Quartz as USB clock you have to comment this line.

4. Is the New STM32 USB FS Device Library fully compatible with the STM32 USB OTG Host and Device Library V2.1.0 ?

   Yes, The low layer driver (DCD and DCD ISR) manage the difference between the two USB IPs, that's why we have built a new USB low level core with the same architecture as the USB OTG one. The aim of this approach is to simplify the migration for user.

   We use the same way to connect the new low level driver with USB Full speed Core, **which provides the same APIs for user.**

   Also the new *STM32 USB FS Device Library* uses the same way (as OTG Library) for different layers interaction between the low level driver, the usb device library and the application layer.

5. What about other audio synchronization mechanism?

   More advanced application with state of art audio synchronization mechanism is available. For any questions regarding this demo please contact your local FAE.

6. Which USB audio class does the Audio demo support?

   The current firmware is using the USB audio class version 1.

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

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

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

In the *usbd_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_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 0 */
  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 */

};
```

9. How can the DFU class driver support more than one memory interface?

   To add an additional memory interface:

a) In the *usbd_conf.h* file (under *Project\USB_Device_Examples\DFU\inc*), change the following define: `#define MAX_USED_MEDIA`

For example:

```
#define MAX_USED_MEDIA       2
```

b) Implement the APIs given by the following structure: `DFU_MAL_Prop_TypeDef` to implement the media I/O requests (Read, Write, Erase …etc), the prototype of each API is given in the *usbd_dfu_mal.h* file.

c) Add the interface string of the new medium to be added in the `usbd_dfu_StringDesc` table defined in the *usbd_dfu_mal.c* file.

10. Can I use Different endpoints than those used in the demo?

Yes, this can be done in the usbd_conf.h file by changing this define (MSC demo case for example)

#define MSC_IN_EP 0x81 > For Endpoint 1 IN

#define MSC_OUT_EP 0x02 > For Endpoint 2 OUT

11. What about USB device stack footprint?

This library is built with a reduced footprint to provide optimum solution for low STM32 memory products (e.g HID Joystick application consumes as low as 7.5KB of flash and 1.5KB of RAM in high compiler optimization).

# 7 Revision history

Table 33. Document revision history

| Date | Revision | Changes |
|---|---|---|
| 11-Feb-2014 | 1 | Initial release |

**Please Read Carefully:**

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST’s terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

**UNLESS OTHERWISE SET FORTH IN ST’S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.**

**ST PRODUCTS ARE NOT DESIGNED OR AUTHORIZED FOR USE IN: (A) SAFETY CRITICAL APPLICATIONS SUCH AS LIFE SUPPORTING, ACTIVE IMPLANTED DEVICES OR SYSTEMS WITH PRODUCT FUNCTIONAL SAFETY REQUIREMENTS; (B) AERONAUTIC APPLICATIONS; (C) AUTOMOTIVE APPLICATIONS OR ENVIRONMENTS, AND/OR (D) AEROSPACE APPLICATIONS OR ENVIRONMENTS. WHERE ST PRODUCTS ARE NOT DESIGNED FOR SUCH USE, THE PURCHASER SHALL USE PRODUCTS AT PURCHASER’S SOLE RISK, EVEN IF ST HAS BEEN INFORMED IN WRITING OF SUCH USAGE, UNLESS A PRODUCT IS EXPRESSLY DESIGNATED BY ST AS BEING INTENDED FOR “AUTOMOTIVE, AUTOMOTIVE SAFETY OR MEDICAL” INDUSTRY DOMAINS ACCORDING TO ST PRODUCT DESIGN SPECIFICATIONS. PRODUCTS FORMALLY ESCC, QML OR JAN QUALIFIED ARE DEEMED SUITABLE FOR USE IN AEROSPACE BY THE CORRESPONDING GOVERNMENTAL AGENCY.**

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.
Information in this document supersedes and replaces all information previously supplied.
The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.



STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Philippines - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

**www.st.com**