UM1021
User manual

STM32F105xx, STM32F107xx, STM32F2xx and STM32F4xx USB On-The-Go host and device library

Introduction

The USB On-The-Go Host and Device Library is a firmware and application software package for USB (universal serial bus) hosts and devices. This package includes example and demonstration software for developing applications using USB full speed and high speed transfer types (control, interrupt, bulk and isochronous).

The aim of the USB OTG Host and Device Library is to provide at least one firmware example demonstration for each USB transfer type. This library is designed for use with the following evaluation boards:

- STM3210C-EVAL evaluation board (UM0600) for STM32F105/7 devices
- STM3220G-EVAL evaluation board (UM1057) for STM32F20x devices
- STM3221G-EVAL evaluation board (UM1065) for STM32F21x devices
- STM3240G-EVAL evaluation board (UM1461) for STM32F40x devices
- STM3241G-EVAL evaluation board (UM1460) for STM32F41x devices

This document describes all the components of a USB OTG host and device library, including examples for the following types of devices:

- Mass storage, based on the microSD card available on the evaluation boards
- HID joystick, based on the embedded joystick on the evaluation boards
- Virtual COM port
- Direct Firmware Update-based
- Audio (OUT)
- Dual Core, based on mass storage and HID examples (available only for STM322xG-EVAL and STM324xG-EVAL evaluation boards)

And the following examples for hosts:

- Mass storage, using file explorer, write files and slide show
- HID, dynamic support for mice and keyboards
- Dual core, for mass storage on the high speed port and HID (keyboards or mice) on the full speed port

The package also includes an example of a manual dual role device that enables the core to switch between host and device modes depending on user input.

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.

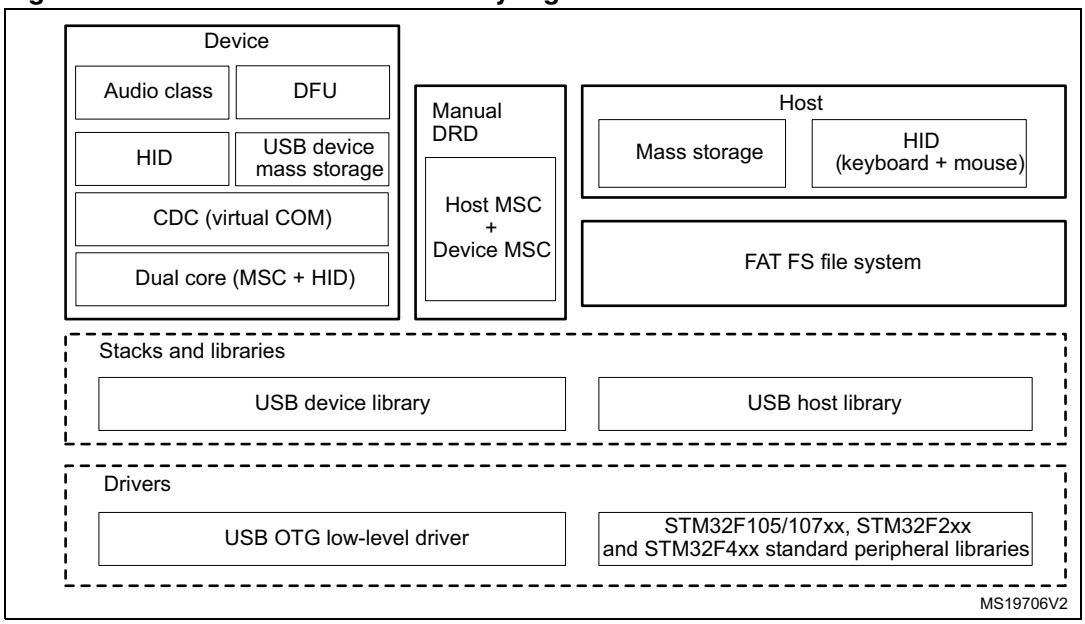
Table 1. List of terms

Term	Meaning
AHB	AMBA High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture
CDC	Communication Device Class
DCD	Device Core Driver
DFU	Device Firmware Upgrade
DRD	Dual Role Device
FIFO	First In, First Out
FS	Full Speed (12 Mbps)
HCD	Host Core Driver
HID	Human Interface Device
HNP	Host Negotiation Protocol
HS	High Speed (480 Mbps)
LS	Low Speed (1.5 Mbps)
Mbps	Megabit per second
MSC	Mass Storage Class
OTG	USB On-The-Go
PHY	Physical Layer (as described in the OSI model)
SRP	Session Request Protocol
USB	Universal Serial Bus

2 USB host and device library overview

The following figure gives an overview of the USB host and device libraries.

Figure 1. USB host and device library organization overview



The USB host and device libraries are built around the common STM32 USB OTG low level driver and the USB device and host libraries.

2.1 Main features

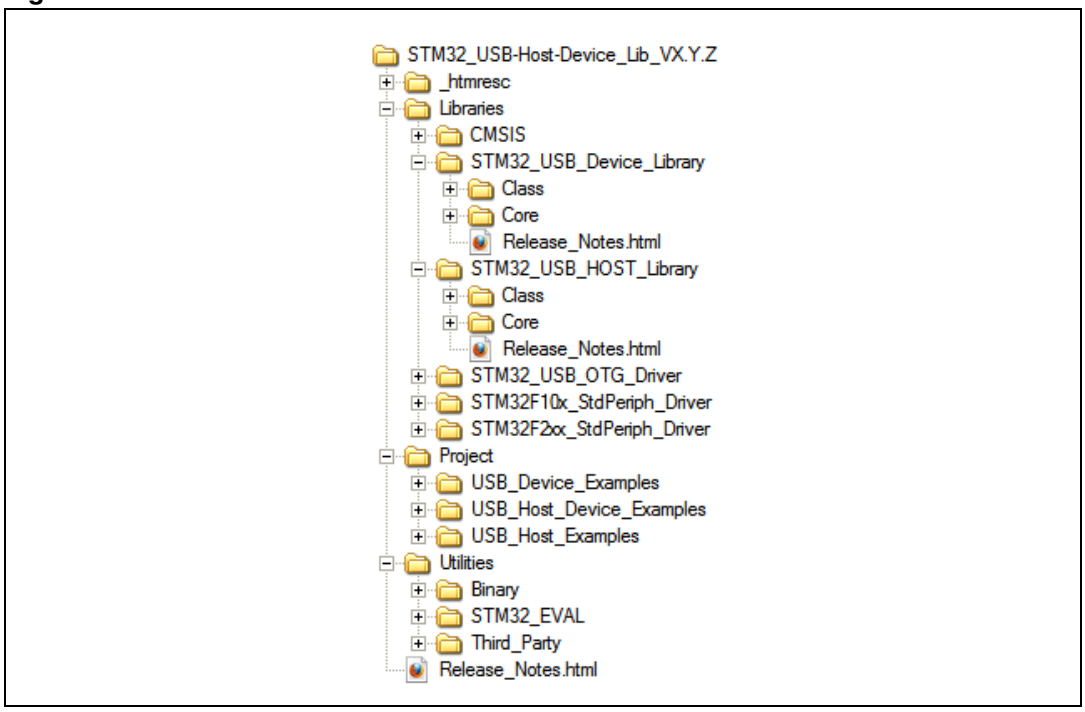
The USB host and device library is:

- Compatible with the STM32F105x, STM32F107x, STM32F2xx and STM32F4xx devices in HS and FS USB modes
- Fully compliant with the Universal Serial Bus Revision 2.0 Specification
- Optimized to work with the USB OTG peripherals (high speed and full speed) and can use all their features
- Built with a reduced footprint, high transfer performance, robustness and a high-quality code and documentation package
- Easily extended to support USB OTG features
- Built following a generic and easy-to-use architecture
  - able to add further specific vendor classes
  - supports multi-interface applications (composite devices)
- Able to support multiple USB OTG cores allowing the use of several cores with the same library

3 USB host and device library folder structure

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

Figure 2. Folder structure



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

1. ***Libraries***: contains the STM32 USB OTG low-level driver, the standard peripherals libraries, the host and the device libraries.
2. ***Project***: 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 (LCD, SD card, buttons, joystick, etc). This folder contains also the FatFs generic file system used for the Host demos.

4 USB OTG core

4.1 USB OTG full speed core

The OTG_FS is a dual-role device (DRD) controller that supports both device and host functions. It is fully compliant with the On-The-Go Supplement to the USB 2.0 Specification. It can also be configured as a host-only or device-only controller, fully compliant with the USB 2.0 Specification. In Host mode, the OTG_FS supports full-speed (12 Mbps) and low-speed (1.5 Mbps) transfers whereas in Device mode, it only supports full-speed transfers.

The OTG_FS supports both HNP (Host Negotiation Protocol) and SRP (Session Request Protocol). The only external device required is a charge pump for the VBUS power supply in Host mode.

4.1.1 OTG_FS interface main features

The OTG_FS interface has the following features:

- Complies with the On-The-Go Supplement to the USB 2.0 Specification (Revision 1.0a)
- Operates in Full Speed (12 Mbps) and Low Speed (1.2 Mbps) modes
- Supports Session Request Protocol (SRP)
- Supports Host Negotiation Protocol (HNP)
- Supports a generic root hub and multi-point capabilities and includes automatic ping capabilities
- Four bidirectional endpoints, including 1 control endpoint and 3 device endpoints which support bulk, interrupt and isochronous transfers
- All device IN endpoints can support periodic transfers
- Eight host channels with periodic OUT support
- Dedicated FIFO transmission for each of the 4 device IN endpoints. Each FIFO can hold multiple packets
- Combined Rx and Tx FIFO size of 320 x 35 bits with Dynamic FIFO sizing (1.25 Kbytes)
- Eight entries in periodic Tx queue, 8 entries in non-periodic Tx queue
- Controls on-chip FS PHY for USB Host, Device or OTG operations
- Requires an external charge pump for VBUS power supply
- 32-bit AHB Slave interface for accessing control and status registers (CSRs) and the data FIFO

4.2 USB OTG high speed core

The OTG_HS is a dual-role device (DRD) controller that supports both peripheral and host functions. It is fully compliant with the On-The-Go Supplement to the USB 2.0 Specification. It can also be configured as a host-only or peripheral-only controller, fully compliant with the USB 2.0 Specification. In Host mode, the OTG_HS supports high-speed (480 Mbps), full-speed (12 Mbps) and low-speed (1.5 Mbps) transfers whereas in Peripheral mode, it only supports high-speed and full-speed transfers.

The OTG_HS supports both HNP (Host Negotiation Protocol) and SRP (Session Request Protocol). The only external device required is a charge pump for the VBUS power supply in OTG mode.

The OTG_HS interface has the following features:

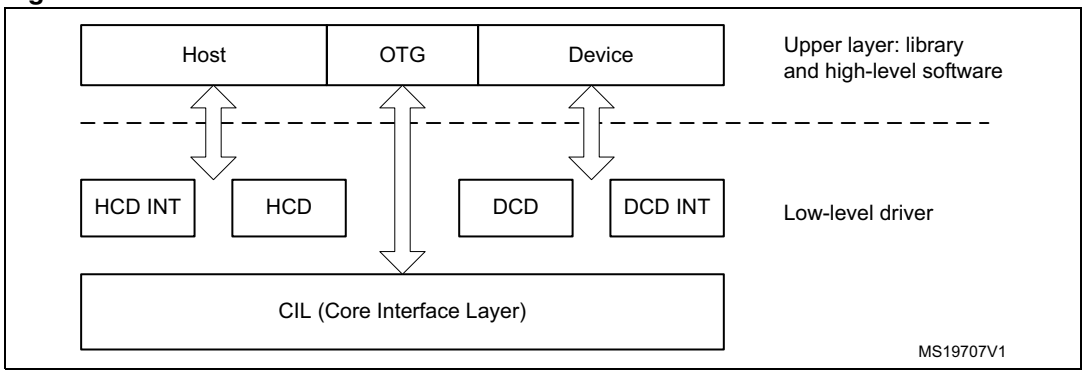
- USB-IF certified with the Universal Serial Bus Revision 2.0 Specification
- Supports two PHY interfaces:
  - An on-chip Full Speed PHY
  - An ULPI (UTMI+ low pin interface) interface for the external High Speed PHY
- Supports the host negotiation protocol (HNP) and the session request protocol (SRP)
- It allows the host to turn VBUS off to save power in OTG applications, with no need for external components
- Can be used to monitor VBUS levels with internal comparators
- Supports dynamic host-peripheral role switching
- Software-configurable to operate as:
  - An SRP-capable USB HS/FS peripheral (B-device)
  - An SRP-capable USB HS/FS/low-speed host (A-device)
  - A USB OTG FS dual-role device
- Supports HS/FS SOF (start-of-frame) pulses as well as low-speed (LS) keep-alive tokens with:
  - SOF pulse pad output capability
  - SOF pulse internal connection to Timer 2 (TIM2)
  - Configurable framing period
  - Configurable end-of-frame interrupt
- Embeds an internal DMA with thresholding support and software selectable AHB burst type in DMA mode
- Powersaving features such as system clock stop during USB suspend, switching off of the digital core internal clock domains, PHY and DFIFO power management
- Dedicated 4-Kbyte data RAM with advanced FIFO management:
  - Memory partition can be configured into different FIFOs to allow flexible and efficient use of RAM
  - Each FIFO can contain multiple packets
  - Memory allocation is performed dynamically
  - FIFO size can be configured to values that are not powers of 2 to allow the use of contiguous memory locations
- Ensures a maximum USB bandwidth of up to one frame without application intervention

STM32F105/07xx devices embed one USB OTG FS core, while STM32F2xx and STM32F4xx devices embed one USB OTG FS core and one HS core.

5 USB OTG low level driver

5.1 USB OTG low level driver architecture

Figure 3. Driver architecture overview



The low level driver can be used to connect the USB OTG 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.

5.2 USB OTG low level driver files

Figure 4. Driver files

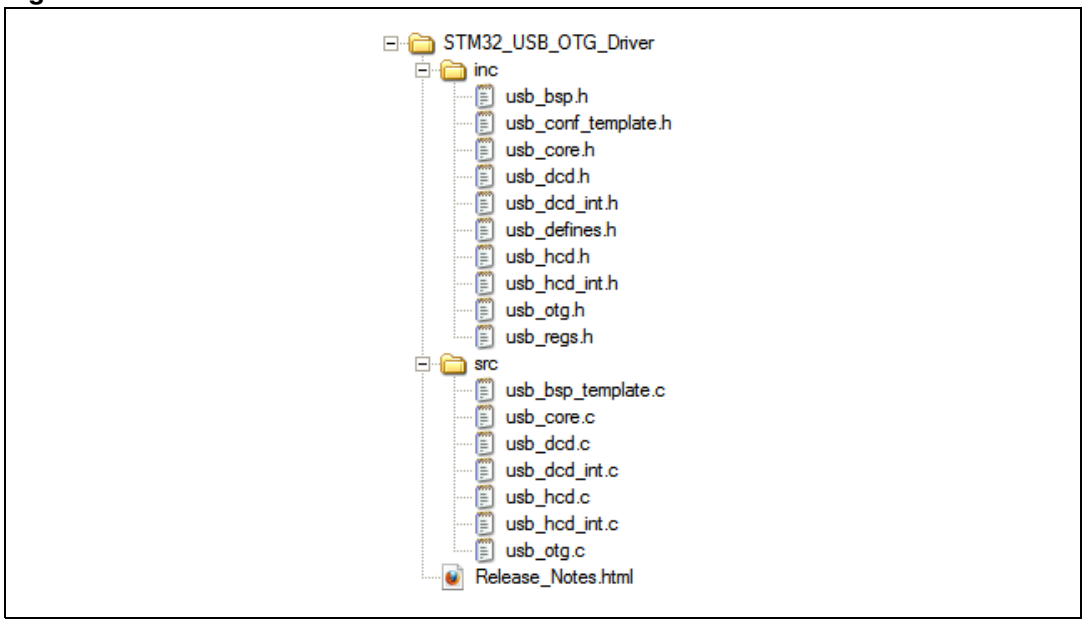


Table 2. USB OTG low level file descriptions

Mode	Files	Description
Common	*usb_core.c/h*	This file contains the hardware abstraction layer and the USB communication operations.
	usb_core.c/h	This file contains the core configuration for Host, Device and OTG modes: Transmit FIFO size, Receive FIFO size, Core mode and selected features...etc. This file should be copied to the application folder and modified depending on the application needs.
	usb_bsp_template.c	This file contains the low level core configuration (interrupts, GPIO). This file should be copied to the application folder and modified depending on the application needs.
Host	*usb_hcd.c/h*	This file contains the host interface layer used by the library to access the core.
	usb_hcd_int.c/h	This file contains the interrupt subroutines for the Host mode.
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.
OTG	*usb_otg.c/h*	This file contains the implementation of the SRP and HNP protocols and the interrupts relative to the OTG mode.

5.3 USB OTG low level driver configuration

The configuration of the USB OTG cores (high and full speed core) is defined in the common configuration file (*usb_conf.h*). The user can enable or disable certain core features, define the Tx and Rx FIFO for the device, the periodic and the non-periodic transmit FIFO and the Rx FIFO for the host. This file is also used to select Host, Device or OTG modes or selecting both Device and Host modes for manual dual role device applications.

The table below gives details of the core configurations defined in the *usb_conf.h* file.

Table 3. Core configurations

Define	Description
`USB_OTG_FS_CORE`	Enables the use of the full speed core.
`USB_OTG_HS_CORE`	Enables the use of the high speed core.
`RX_FIFO_FS_SIZE`	Sets the Receive FIFO size for the full speed core.
`RX_FIFO_HS_SIZE`	Sets the Receive FIFO size for the high speed core.
`TXn_FIFO_FS_SIZE`	Sets the Transmit FIFO size for a device endpoint (Full speed) where *n* is the Index of the endpoint to be used.
`TXn_FIFO_HS_SIZE`	Sets the Transmit FIFO size for a device endpoint (High Speed core) where *n* is the Index of the endpoint to be used.
`TXH_NP_FS_FIFOSIZ`	Sets the non-periodic Transmit FIFO size for Host mode (Full Speed).

Table 3. Core configurations (continued)

Define	Description
TXH_NP_HS_FIFOSIZ	Sets the non-periodic Transmit FIFO size for Host mode (High Speed core).
TXH_P_FS_FIFOSIZ	Sets the Periodic Transmit FIFO size for Host mode (Full Speed).
TXH_P_HS_FIFOSIZ	Sets the Periodic Transmit FIFO size for Host mode (High Speed core).
USB_OTG_ULPI_PHY_ENABLED	Enables the ULPI PHY for High Speed core.
USB_OTG_EMBEDDED_PHY_ENABLED	Enables the embedded FS PHY for High Speed core.
USB_OTG_HS_LOW_PWR_MGMT_SUPPORT	Enables low power management for High Speed core (USB Core clock gating, etc.).
USB_OTG_FS_LOW_PWR_MGMT_SUPPORT	Enables low power management for Full Speed core (USB Core clock gating, etc.).
USB_OTG_HS_INTERNAL_DMA_ENABLED	Enables the internal DMA feature for High Speed core.
USB_OTG_HS_DEDICATED_EP1_ENABLED	Enables the dedicated Endpoint 1 feature for Device mode in High Speed core.

5.4 USB OTG driver programming manual

5.4.1 Low level driver structures

The low level driver does not have any exportable variables. A global structure (USB_OTG_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.

This method allows the application to use the same low level driver for both high- and Full Speed cores in the same project.

The global USB core structure is defined as follows:

```
typedef struct USB_OTG_handle
{
  USB_OTG_CORE_CFGS    cfg;
  USB_OTG_CORE_REGS    regs;
#ifdef USE_DEVICE_MODE
  DCD_DEV     dev;
#endif
#ifdef USE_HOST_MODE
  HCD_DEV     host;
#endif
#ifdef USE_OTG_MODE
  OTG_DEV     otg;
#endif
}
USB_OTG_CORE_HANDLE, *PUSB_OTG_CORE_HANDLE;
```

5.4.2 Programming considerations when using internal DMA

When using the internal DMA with the USB OTG High Speed core, all structures dealing with the DMA (data buffer) during the transaction process should be 32-bit aligned.

Consequently, the `USB_OTG_handle` structure is defined to keep all the internal buffers and variables used to hold the data to be transferred 32-bit aligned.

When the internal DMA is used, the global USB Core structure should be declared as follows:

Figure 5. USB core structure

```
#ifdef USB_OTG_HS_INTERNAL_DMA_ENABLED
  #if defined ( __ICCARM__ ) /*!< IAR Compiler */
    #pragma data_alignment=4
  #endif
#endif /* USB_OTG_HS_INTERNAL_DMA_ENABLED */

__ALIGN_BEGIN USB_OTG_CORE_HANDLE     USB_OTG_dev __ALIGN_END ;
```

Note: *__ALIGN_BEGIN and __ALIGN_END are compiler-specific keywords defined in the* usb_conf.h *file and are used to align variables on a 32-bit boundary.*

Figure 6. C compiler-dependant keywords (defined in usb_conf.h file)

```
/******************* C Compilers dependant keywords ****************************/
/* In HS mode and when the DMA is used, all variables and data structures dealing
   with the DMA during the transaction process should be 4-bytes aligned */
#ifdef USB_OTG_HS_INTERNAL_DMA_ENABLED
  #if defined   (__GNUC__)        /* GNU Compiler */
    #define __ALIGN_END    __attribute__ ((aligned (4)))
    #define __ALIGN_BEGIN
  #else
    #define __ALIGN_END
    #if defined   (__CC_ARM)      /* ARM Compiler */
      #define __ALIGN_BEGIN    __align(4)
    #elif defined (__ICCARM__)    /* IAR Compiler */
      #define __ALIGN_BEGIN
    #elif defined  (__TASKING__)  /* TASKING Compiler */
      #define __ALIGN_BEGIN    __align(4)
    #endif /* __CC_ARM */
  #endif /* __GNUC__ */
#else
  #define __ALIGN_BEGIN
  #define __ALIGN_END
#endif /* USB_OTG_HS_INTERNAL_DMA_ENABLED */

/* __packed keyword used to decrease the data type alignment to 1-byte */
#if defined (__CC_ARM)          /* ARM Compiler */
  #define __packed    __packed
#elif defined (__ICCARM__)      /* IAR Compiler */
  #define __packed    __packed
#elif defined   ( __GNUC__ )    /* GNU Compiler */
  #define __packed    __attribute__ ((__packed__))
#elif defined   (__TASKING__)   /* TASKING Compiler */
  #define __packed    __unaligned
#endif /* __CC_ARM */
```

5.4.3 Selecting USB physical interface

As described in the USB OTG Low Level Driver configuration, the user can select the USB Physical interface (PHY) to be used.

- For the USB OTG Full Speed Core, the embedded Full Speed PHY is used.
- When using the USB OTG High Speed core, the user can select one of the two PHY interfaces:
  - A ULPI interface for the external High Speed PHY: the USB HS Core will operate in High speed mode
  - An on-chip Full Speed PHY: the USB HS Core will operate in Full speed mode

The library provides the capability of selecting the PHY to be used using one of these two defines (in usb_conf.h file)(as described in *Section 5.3: USB OTG low level driver configuration on page 14*) :

- USE_ULPI_PHY: if the USB OTG HS Core is to be used in High speed mode
- USE_EMBEDDED_PHY: if the USB OTG HS Core is to be used in Full speed mode

Note: With the ULPI interface, the user can force the core to work in Full Speed mode by modifying the usb_core.c *file through the ULPIFSLS bit in the OTG_HS_GUSBCFG register.*

In Host mode, the core speed can be modified when a device with a lower speed is connected.

5.4.4 Programming device drivers

Device initialization

The device is initialized using the following function:

```
DCD_Init (USB_OTG_CORE_HANDLE *pdev, USB_OTG_CORE_ID_TypeDef coreID)
```

The Rx and Tx FIFOs size and start address are set inside this function to use one more endpoints in addition to the control Endpoint (0). The user can change the FIFO settings by modifying the default values and changing the FIFO depth for each Tx FIFO in the *usb_conf.h* file.

Endpoint configuration

Once the USB OTG core is initialized, the device mode is selected. 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_OTG_CORE_HANDLE *pdev ,
uint8_t ep_addr,
uint16_t ep_mps,
uint8_t ep_type)
uint32_t DCD_EP_Close (USB_OTG_CORE_DEVICE *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_address;
  uint32_t       DevRemoteWakeup;
  USB_OTG_EP     in_ep   [USB_OTG_MAX_TX_FIFOS];
  USB_OTG_EP     out_ep  [USB_OTG_MAX_TX_FIFOS];
  uint8_t        setup_packet [8*3];
  USBD_Class_cb_TypeDef         *class_cb;
  USBD_Usr_cb_TypeDef           *usr_cb;
  uint8_t        *pConfig_descriptor;
 }
DCD_DEV , *DCD_PDEV;
```

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_OTG_EP0_IDLE          0
#define USB_OTG_EP0_SETUP         1
#define USB_OTG_EP0_DATA_IN       2
#define USB_OTG_EP0_DATA_OUT      3
#define USB_OTG_EP0_STATUS_IN     4
#define USB_OTG_EP0_STATUS_OUT    5
#define USB_OTG_EP0_STALL         6
```

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

```
/* Device Status */
#define USB_OTG_DEFAULT           0
#define USB_OTG_ADDRESSED         1
#define USB_OTG_CONFIGURED        2
```

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_OTG_CORE_HANDLE *pdev,
                      uint8_t   ep_addr,
                      uint8_t *pbuf,
                      uint16_t  buf_len);
uint32_t    DCD_EP_Tx (USB_OTG_CORE_HANDLE *pdev,
                           uint8_t  ep_addr,
                           uint8_t  *pbuf,
                           uint32_t   buf_len);
uint32_t    DCD_EP_Stall (USB_OTG_CORE_HANDLE *pdev,
                          uint8_t   epnum);
uint32_t    DCD_EP_ClrStall (USB_OTG_CORE_HANDLE *pdev,
                              uint8_t epnum);
uint32_t    DCD_EP_Flush (USB_OTG_CORE_HANDLE *pdev,
                           uint8_t epnum);
```

The DCD layer of the USB OTG Low Level Driver has one function that must be called by the USB interrupt (high speed or full speed):

```
uint32_t    DCD_Handle_ISR (USB_OTG_CORE_HANDLE *pdev)
```

The *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) (USB_OTG_CORE_HANDLE *pdev , uint8_t
epnum);
        uint8_t (* DataInStage)  (USB_OTG_CORE_HANDLE *pdev , uint8_t
epnum);
        uint8_t (* SetupStage) (USB_OTG_CORE_HANDLE *pdev);
        uint8_t (* SOF) (USB_OTG_CORE_HANDLE *pdev);
        uint8_t (* Reset) (USB_OTG_CORE_HANDLE *pdev);
        uint8_t (* Suspend) (USB_OTG_CORE_HANDLE *pdev);
        uint8_t (* Resume) (USB_OTG_CORE_HANDLE *pdev);
        uint8_t (* IsoINIncomplete) (USB_OTG_CORE_HANDLE *pdev);
        uint8_t (* IsoOUTIncomplete) (USB_OTG_CORE_HANDLE *pdev);
        uint8_t (* DevConnected) (USB_OTG_CORE_HANDLE *pdev);
        uint8_t (* DevDisconnected) (USB_OTG_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;
```

Specific OUT and IN interrupt

The USB OTG High Speed core embeds two independent interrupts for endpoint 1 IN and endpoint 1 OUT. Consequently, the `USBD_OTG_EP1OUT_ISR_Handler` and `USBD_OTG_EP1IN_ISR_Handler` can be used to lighten the global USB OTG interrupt.

The specific endpoint feature is selected by enabling the `USB_OTG_HS_DEDICATED_EP1_ENABLED` define in the *usb_conf.h* file.

Internal DMA use in High speed mode

The USB OTG High Speed core embeds an internal DMA capable of handling the FIFO I/O request automatically without using the CPU. However the data structures used in DMA mode should be 32-bit aligned.

The internal DMA feature is selected by enabling the `USB_OTG_HS_INTERNAL_DMA_ENABLED` define in the *usb_conf.h* file.

Note: The Internal DMA and Specific OUT and IN interrupt features can be used together to enhance data transfer performance.

5.4.5 Programming host drivers

Host driver initialization

The host is initialized using the following function:

```
HCD_Init (USB_OTG_CORE_HANDLE *pdev, USB_OTG_CORE_ID_TypeDef coreID)
```

This function sets the Rx and periodic/non periodic Tx FIFOs size and start address. The user can change the FIFO settings by modifying the default values and changing the FIFO depth for each periodic and non periodic Tx FIFO in the *usb_conf.h* file.

Host channel initialization

Once the USB OTG core is initialized, the host mode is selected. The upper layer may call the low level driver to open or close a host channel to start transferring data. The following API is used:

```
uint32_t HCD_HC_Init (USB_OTG_CORE_HANDLE *pdev, uint8_t hc_num)
```

Host driver structures

After initializing the Host driver (HCD), the low level driver holds several structures and buffers for data and URB status monitoring. The host channel structures are kept in the host driver and accessed from the upper layer through the host number index.

```
typedef struct _HCD
{
  uint8_t                   Rx_Buffer [MAX_DATA_LENGTH];
  __IO uint32_t             ConnSts;
  __IO uint32_t             ErrCnt[USB_OTG_MAX_TX_FIFOS];
  __IO uint32_t             XferCnt[USB_OTG_MAX_TX_FIFOS];
  __IO HC_STATUS            HC_Status[USB_OTG_MAX_TX_FIFOS];
  __IO URB_STATE            URB_State[USB_OTG_MAX_TX_FIFOS];
  USB_OTG_HC                hc [USB_OTG_MAX_TX_FIFOS];
  uint16_t                  channel [USB_OTG_MAX_TX_FIFOS];
```

```
  USB_OTG_hPort_TypeDef      *port_cb;

} HCD_DEV , *USB_OTG_USBH_PDEV;
```

In this structure,

- `Rx_Buffer`: this buffer holds the IN packet and can be accessed directly from the global Host Core structure as follows: pdev->host.Rx_Buffer.
- `ConnSts`: connection status. It can be accessed directly or by using the `HCD_IsDeviceConnected ()` function.
- `ErrCnt`: holds the number of errors on a channel during one transfer.
- `XferCnt`: holds the number of IN data already received and available in the Rx_Buffer. It can be accessed directly or by using the `GetXferCnt ()` function.
- `HC_Status`: used internally by the driver. It can be accessed also by the upper layer. It keeps the status of the current transaction on a channel.
- `URB_State`: this variable keeps the transfer status on a host channel.
- `Channel`: this variable manages the host channels status (`used` or `free`).
- `port_cb`: host port callbacks that contain variables used to track the use of the connection or disconnection handler to prevent multiple accesses to this handler.

Example:

```
if (!(HCD_IsDeviceConnected(pdev)) &&
       (pdev->host.port_cb->DisconnHandled == 0))
  {
    (…)

    pdev->host.port_cb->DisconnHandled = 1; /* Handle to avoid the Re-
entry*/
    USBH_DeInit(pdev, phost);

    (…)
}
```

Starting a host transfer

Once a host channel is initialized, it can be used to start a host transfer. The following API is used:

```
uint32_t HCD_SubmitRequest (USB_OTG_CORE_HANDLE *pdev , uint8_t hc_num)
```

At this point, the transfer flow is handled by the HCD interrupts (*usb_hcd_int.c/h*) and the upper layer can monitor the transfer progress using the following APIs:

```
URB_STATE HCD_GetURB_State (USB_OTG_CORE_HANDLE *pdev , uint8_t ch_num)
HC_STATUS HCD_GetHCState (USB_OTG_CORE_HANDLE *pdev ,  uint8_t ch_num)
uint32_t HCD_GetXferCnt (USB_OTG_CORE_HANDLE *pdev, uint8_t ch_num)
```

USB host monitoring

It is not possible to start any USB Host transfer without connecting a USB device. The application can probe the USB Host port using the following API:

```
uint32_t HCD_IsDeviceConnected (USB_OTG_CORE_HANDLE *pdev)
```

USB Host interrupt subroutines

The HCD layer of the USB OTG Low Level Driver has one function to be called from the USB interrupt (high speed or full speed):

```
uint32_t    HCD_Handle_ISR (USB_OTG_CORE_HANDLE *pdev)
```

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

Internal DMA use in high speed mode

The USB OTG High Speed core embeds an internal DMA capable of handling the FIFO I/O request automatically without using the CPU. However, the data structures used in DMA mode must be 32-bit aligned.

The internal DMA feature is selected by enabling the `USB_OTG_HS_INTERNAL_DMA_ENABLED` define in the *usb_conf.h* file.

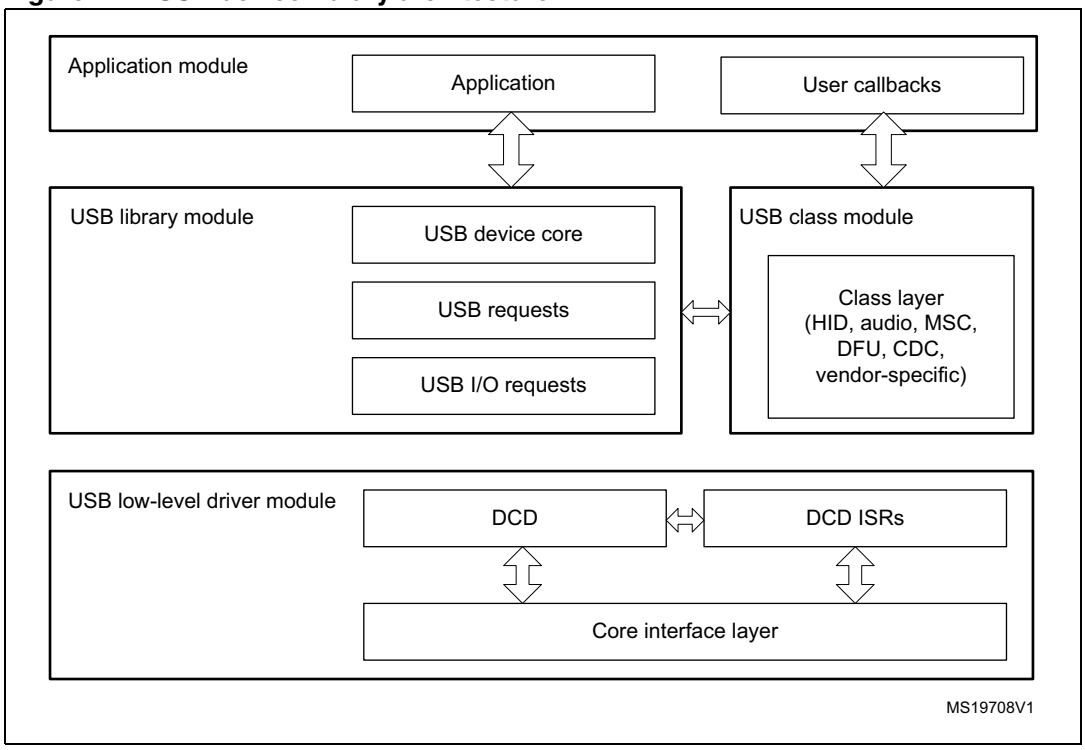
6 USB device library

The USB device library:

- Supports multi-packet transfer features so that a large amount of data can be sent without having to split it into maximum packet size transfers.
- Supports up to three back-to-back transfers on control endpoints (compatible with OHCI controllers).
- Uses configuration files to change the core and the library configuration without changing the library code (Read Only).
- 32-bit aligned data structures to handle DMA-based transfer in High speed modes.
- Supports multi USB OTG core instances from a user level.

6.1 USB device library overview

Figure 7. USB device library architecture



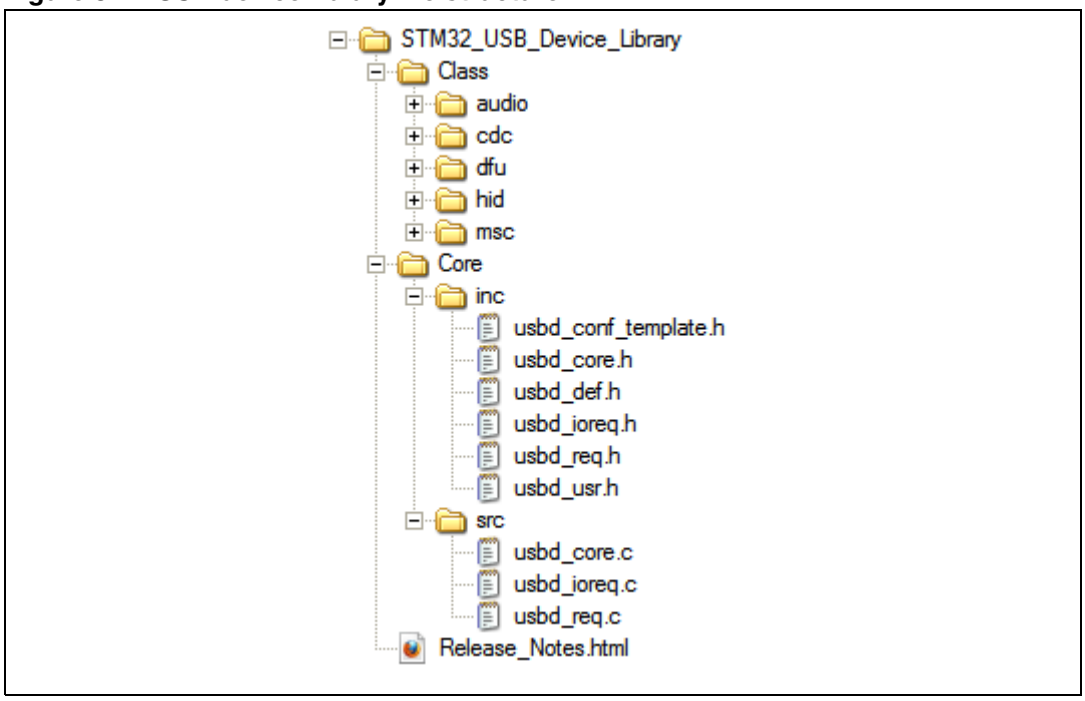
As shown in the above figure, the USB device library is composed of two main parts: the library core and the class drivers.

The library core is composed of three main blocks:

- USB device core
- USB requests
- USB I/O requests

6.2 USB device library files

Figure 8. USB device library file structure



The USB device library is based on the generic USB OTG low level driver which supports Host, Device and OTG modes and works for High speed, Full speed and Low speed (for host mode).

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

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

6.3 USB device library description

6.3.1 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

Most of 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.

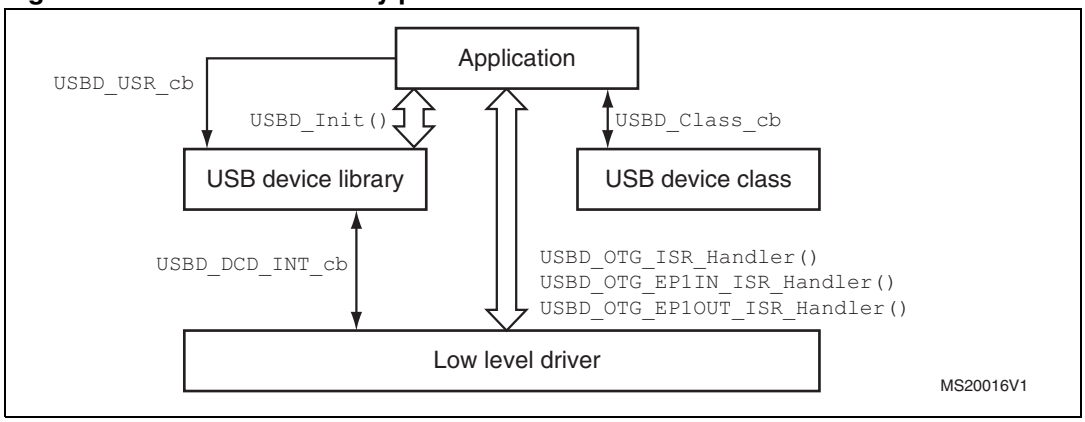
- Status stage

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

6.3.2 USB device library process

Figure 9 shows the different layers interaction between the low level driver, the usb device library and the application layer.

Figure 9. USB device library process flowchart



The Application layer has only to call to 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 application has also to use the general USB ISR and the specific EP1 subroutines when the `USB_OTG_HS_DEDICATED_EP1_ENABLED` define is uncommented in the *usb_conf.h* file.

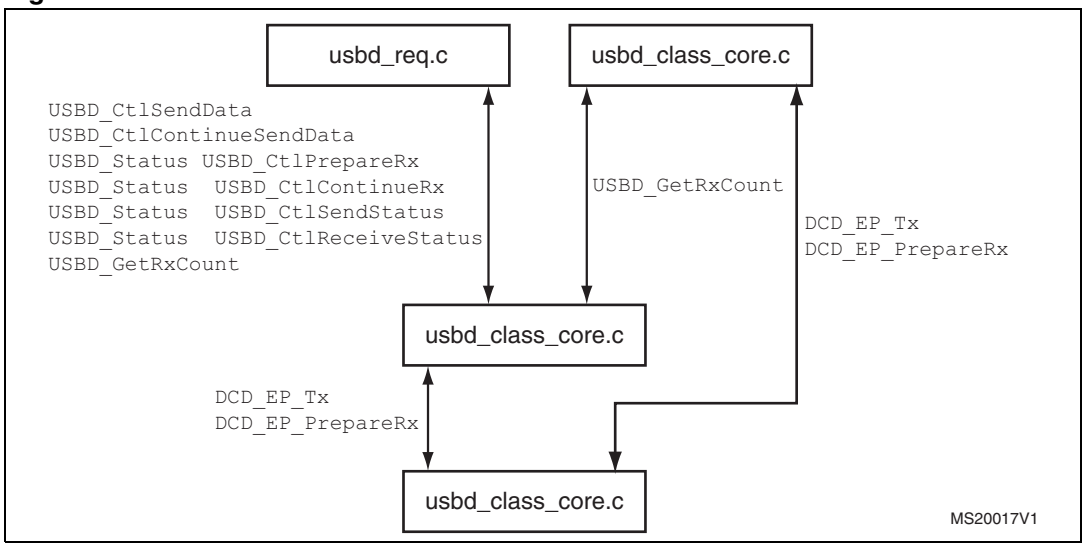
The `USBD_Init` function needs however the user callback structure to inform the user layer of the different library states and messages and the class callback structure to start the class interface.

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.

6.3.3 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 when a wrapping is needed to manage the multi-packet feature on the control endpoint or directly from the *usb_dcd.c* layer when the other endpoints are used since the USB OTG core supports the multi-packet feature. The following figure illustrates this data flow scheme.

Figure 10. USB device data flow



6.3.4 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
#define USB_MAX_STR_DESC_SIZ        64

/****  USB_MSC_Class_Layer_Parameter *********/
#define MSC_IN_EP                        0x81
#define MSC_OUT_EP                      0x01
#define MSC_MAX_PACKET          512
#define MSC_MEDIA_PACKET     4096

/****  USB_HID Class_Layer_Parameter *********/
#define HID_IN_EP                        0x81
#define HID_OUT_EP                     0x01
#define HID_IN_PACKET             4
#define HID_OUT_PACKE            4
```

6.3.5 USB data transfer handling

The USB data transfer handling supports multi-packet transfer features so that a large amount of data can be sent without splitting it into maximum packet size transfers. The multi packet process is handled by the low level driver through the `DCD_HandleRxStatusQueueLevel_ISR` and `DCD_HandleInEP_ISR` when the USB OTG core is running in Slave mode and by the internal DMA when the DMA mode is used (DMA mode available only with the USB OTG HS core).

6.3.6 Using the multi-packet feature

To transmit data, the `DCD_EP_Tx ()` function is called and to receive data `DCD_EP_PrepareRx()` is called, an with unlimited data length. Internally, the USB OTG core checks the available space in the FIFO and processes the transfer, respecting the endpoint size. For example, if the endpoint size is configured to work with 64 bytes of data and the user wants to transmit / receive N bytes of data, the USB core sends / receives several packets of 64 bytes each.

6.3.7 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. Interrupt routine restrictions apply.

Device suspend

When the device detects a suspend condition on the USB, the library stops all the operations and puts the system to suspend state (if low power mode is enabled by 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 to idle state (if low power mode is enabled by in the *usb_conf.h* file).

6.3.8 FIFO size customization

In order to use a new endpoint or change the endpoint already used in the application, the user has to take care of two things:

1. Endpoint initialization: this phase is done generally in the usbd_class_core layer through the following function:

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

   The ep_addr should hold the endpoint address, note the endpoint direction is identified by the MSB bit (ie "0x80| ep" index for ep IN endpoint) and the ep_mps holds the max packet size of the endpoints.

2. The FIFO configuration done in the usb_core.c file in the usb low level driver , the FIFO configuration could be modified by the user through the usb_conf.h file.

```
#ifdef USB_OTG_FS_CORE
   #define RX_FIFO_FS_SIZE128
   #define TX0_FIFO_FS_SIZE64
   #define TX1_FIFO_FS_SIZE128
   #define TX2_FIFO_FS_SIZE0
```

```
   #define TX3_FIFO_FS_SIZE0
#endif
#ifdef USB_OTG_HS_CORE
   #define RX_FIFO_HS_SIZE512
   #define TX0_FIFO_HS_SIZE128
   #define TX1_FIFO_HS_SIZE384
   #define TX2_FIFO_HS_SIZE0
   #define TX3_FIFO_HS_SIZE0
   #define TX4_FIFO_HS_SIZE0
   #define TX5_FIFO_HS_SIZE0
        #endif
```

The configuration of the FIFO is described in detail in reference manuals *RM0033* and *RM0008*. The Rx and the TXs FIFOs can be calculated as follows:

1. Receive data FIFO size = RAM for setup packets + Data Out endpoint control information + Data Out packets + Miscellaneous

Note: Space = ONE 32-bit word

   - RAM for setup packets = 10
   - Data Out endpoint control information = 1 space (one space for status information written to the FIFO along with each received packet).
   - Data Out packets = (largest packet size / 4) + 1 space (MINIMUM to receive packets) OR Data Out packets = at least 2*(largest packet size / 4) + 1 space (if high-bandwidth endpoint is enabled or multiple isochronous endpoints)
   - Miscellaneous = 1 space per Data Out endpoint (one space for transfer complete status information also pushed to the FIFO with each endpoint's last packet)

2. MINIMUM RAM space required for each Data In endpoint Tx FIFO = MAX packet size for that particular Data In endpoint. More space allocated in the Data In endpoint Tx FIFO results in a better performance on the USB and can hide latencies on the AHB.
3. Tx*n* minimum size = 16 words. (where, *n* is the Transmit FIFO index).
4. When a Tx FIFO is not used, the Configuration should be as follows:

   Case 1: n > m and Txn is not used (where, *n,m* are the Transmit FIFO indexes)

   - Tx*m* can use the space allocated for Tx*n*.

   Case 2: n < m and Txn is not used (where, *n,m* are the Transmit FIFO indexes)

   - Tx*n* should be configured with the minimum space of 16 words

5. The FIFO is used optimally when used TxFIFOs are allocated in the top of the FIFO. For example, use EP1 and EP2 as IN instead of EP1 and EP3 as the IN ones.

   The total FIFO size for the used USB OTG core: for the USB OTG FS core, the total FIFO size is 320 * 32 bits while for the USB OTG HS core, the total FIFO size is 1024 * 32 bits.

Example

If the application uses 1 IN endpoint for control with MPS = 64 Bytes, 1 OUT Endpoint for Control with MPS = 64 Bytes and 1 IN Bulk endpoint for the class with MPS = 512 Bytes.

The EP0 IN and OUT are configured by the USB Device library. However the user should open the IN endpoint 1 in the class layer as shown below:

```
DCD_EP_Open (pdev,
      0x81,
      512,
      USB_OTG_EP_BULK)
```

and configure the `TX1_FIFO_FS_SIZE` using the formula described in reference manuals *RM0033*, RM0090 and *RM0008*.

6.4 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

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

Functions	Description
`void USBD_Init (USB_OTG_CORE_HANDLE *pdev, USB_OTG_CORE_ID_TypeDef coreID, 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_OTG_CORE_HANDLE *pdev)`	Un-initializes the device library.
`uint8_t USBD_SetupStage (USB_OTG_CORE_HANDLE *pdev)`	Handles the setup stage.
`uint8_t USBD_DataOutStage (USB_OTG_CORE_HANDLE *pdev , uint8_t epnum)`	Handles the Data Out stage.
`uint8_t USBD_DataInStage (USB_OTG_CORE_HANDLE *pdev , uint8_t epnum)`	Handles the Data In stage.
`uint8_t USBD_Reset (USB_OTG_CORE_HANDLE *pdev)`	Handles the reset event.

Table 6. usbd_core (.c, .h) files (continued)

Functions	Description
`uint8_t USBD_Resume (USB_OTG_CORE_HANDLE *pdev)`	Handles the resume event.
`uint8_t USBD_Suspend (USB_OTG_CORE_HANDLE *pdev)`	Handles the suspend event.
`uint8_t USBD_SOF (USB_OTG_CORE_HANDLE *pdev)`	Handles the SOF event.
`USBD_Status USBD_SetCfg (USB_OTG_CORE_HANDLE *pdev, uint8_t cfgidx)`	Configures the device and starts the interface.
`USBD_Status USBD_ClrCfg (USB_OTG_CORE_HANDLE *pdev, uint8_t cfgidx)`	Clears the current configuration.
`uint8_t USBD_IsoINIncomplete(USB_OTG_CORE_HANDLE *pdev)`	Handles incomplete isochronous IN transfer
`uint8_t USBD_IsoOUTIncomplete(USB_OTG_CORE_HANDLE *pdev)`	Handles incomplete isochronous OUT transfer.
`uint8_t USBD_DevConnected(USB_OTG_CORE_HANDLE *pdev)`	Handles device connection event.
`static uint8_t USBD_DevDisconnected(USB_OTG_CORE_HANDLE *pdev)`	Handles device disconnection event.

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

Functions	Description
`USBD_Status USBD_CtlSendData ( USB_OTG_CORE_HANDLE *pdev, uint8_t *pbuf, uint16_t len)`	Sends the data on the control pipe.
`USBD_Status USBD_CtlContinueSendData (USB_OTG_CORE_HANDLE *pdev, uint8_t *pbuf, uint16_t len)`	Continues sending data on the control pipe.
`USBD_Status USBD_CtlPrepareRx (USB_OTG_CORE_HANDLE *pdev, uint8_t *pbuf, uint16_t len)`	Prepares the core to receive data on the control pipe.
`USBD_Status USBD_CtlContinueRx (USB_OTG_CORE_HANDLE *pdev, uint8_t *pbuf, uint16_t len)`	Continues receiving data on the control pipe.

Table 7. usbd_ioreq (.c, .h) files (continued)

Functions	Description
`USBD_Status USBD_CtlSendStatus (USB_OTG_CORE_HANDLE *pdev)`	Sends a zero length packet on the control pipe.
`USBD_Status USBD_CtlReceiveStatus (USB_OTG_CORE_HANDLE *pdev)`	Receives a zero length packet on the control pipe.

Table 8. usbd_req (.c, .h)

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_OTG_CORE_HANDLE *pdev, USB_SETUP_REQ *req)`	Handles standard USB device requests.
`USBD_Status USBD_StdItfReq (USB_OTG_CORE_HANDLE *pdev, USB_SETUP_REQ *req)`	Handles standard USB interface requests.
`USBD_Status USBD_StdEPReq (USB_OTG_CORE_HANDLE *pdev, USB_SETUP_REQ *req)`	Handles standard USB endpoint requests.
`static void USBD_GetDescriptor (USB_OTG_CORE_HANDLE *pdev, USB_SETUP_REQ *req)`	Handles Get Descriptor requests.
`static void USBD_SetAddress (USB_OTG_CORE_HANDLE *pdev, USB_SETUP_REQ *req)`	Sets new USB device address.
`static void USBD_SetConfig (USB_OTG_CORE_HANDLE *pdev, USB_SETUP_REQ *req)`	Handles Set device configuration request.
`static void USBD_GetConfig (USB_OTG_CORE_HANDLE *pdev, USB_SETUP_REQ *req)`	Handles Get device configuration request.
`static void USBD_GetStatus (USB_OTG_CORE_HANDLE *pdev, USB_SETUP_REQ *req)`	Handles Get Status request.
`static void USBD_SetFeature (USB_OTG_CORE_HANDLE *pdev, USB_SETUP_REQ *req)`	Handles Set device feature request.
`static void USBD_ClrFeature (USB_OTG_CORE_HANDLE *pdev, USB_SETUP_REQ *req)`	Handles Clear device feature request.

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

Functions	Description
`void USBD_ParseSetupRequest ( USB_OTG_CORE_HANDLE *pdev, USB_SETUP_REQ *req)`	Copies request buffer into setup structure.
`void USBD_CtlError ( USB_OTG_CORE_HANDLE *pdev, USB_SETUP_REQ *req)`	Handles USB Errors on the control pipe.

6.5 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  (*IsoINIncomplete)   (void *pdev);
      uint8_t  (*IsoOUTIncomplete)   (void *pdev);
      uint8_t  *(*GetConfigDescriptor)( uint8_t speed , uint16_t *length);
    #ifdef USB_OTG_HS_CORE
      uint8_t  *(*GetOtherConfigDescriptor)( uint8_t speed , uint16_t
*length);
    #endif
    #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.
- ***IsoINIncomplete***: This callback is called when the last isochronous IN transfer is incomplete.
- ***IsoOUTIncomplete***: This callback is called when the last isochronous OUT transfer is incomplete.
- ***GetConfigDescriptor***: This callback returns the USB Configuration descriptor.
- ***GetOtherConfigDescriptor***: This callback returns the other configuration descriptor of the used class in High Speed mode.
- ***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.

6.6 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:

```
typedef struct _USBD_USR_PROP
{
   void (*Init)(void);
   void (*DeviceReset)(uint8_t speed);
   void (*DeviceConfigured)(void);
   void (*DeviceSuspended)(void);
   void (*DeviceResumed)(void);
   void (*DeviceConnected)(void);
   void (*DeviceDisconnected)(void);
}
USBD_Usr_cb_TypeDef;
```

- ***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.
- ***DeviceConnected***: This callback is called when the device is connected to the host.
- ***DeviceDisconnected***: This callback is called when the device is disconnected 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);
#ifdef USB_SUPPORT_USER_STRING_DESC
  uint8_t*  (*Get_USRStringDesc) (uint8_t speed, uint8_t idx ,
    uint16_t *length);
#endif /* USB_SUPPORT_USER_STRING_DESC */
   } 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.

6.7 USB device classes

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

The table below presents the USB device class files for the MSC and HID classes.

Table 9. USB device class files

Class	Files	Description
HID	usbd_hid (.c, .h)	This file contains the HID class callbacks (driver) and the configuration descriptors relative to this class.
MSC	usbd_msc(.c, .h)	This file contains the MSC class callbacks (driver) and the configuration descriptors relative to this class.
	usbd_bot (.c, .h)	This file handles the bulk only transfer protocol.
	usbd_scsi (.c, .h)	This file handles the SCSI commands.
	usbd_info (.c,.h)	This file contains the vital inquiry pages and the sense data of the mass storage devices.
	usbd_mem.h	This file contains the function prototypes of the called functions from the SCSI layer to have access to the physical media
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_otp_if (.c,.h)	This file contains the DFU class callbacks relative to the OTP memory interface.
	usbd_template_if (.c,.h)	This file provides a template driver which allows you to implement additional memory interfaces.
Audio	usbd_audio_core (.c,.h)	This file contains the AUDIO class callbacks (driver) and the configuration descriptors relative to this class.
	usbd_audio_out_if (.c,.h)	This file contains the lower layer audio out driver (from USB host to output speaker).
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.

6.7.1 HID class

HID class implementation

This module manages the MSC 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_OTG_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	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_OTG_CORE_HANDLE *pdev, uint8_t *report, uint16_t len)`	Sends HID reports.

6.7.2 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:

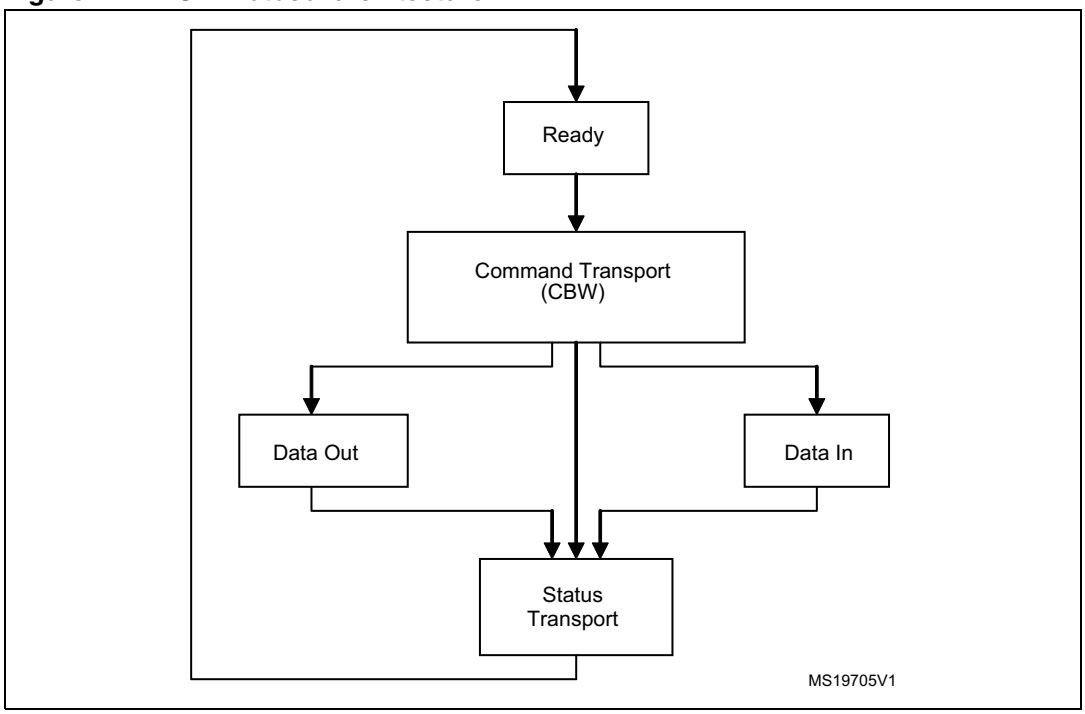
- Bulk-only transport protocol
- Subclass: SCSI transparent command set (ref. SCSI Primary Commands - 3)

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 11. BOT Protocol architecture



The following table shows the supported SCSI commands.

Table 11. SCSI commands

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

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

Table 14. usbd_msc_scsi (.c, .h)

Functions	Description
`int8_t SCSI_ProcessCmd (USB_OTG_CORE_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.
`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 function 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, uint16_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. 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.

Table 15. Functions (continued)

Functions	Description
`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.
`int8_t STORAGE_Write (uint8_t lun, uint8_t *buf, uint32_t blk_addr, uint16_t blk_len)`	Writes data to the medium: – `blk_address` is given in sector unit – `blk_len` is the number of the sector to be processed.
`int8_t STORAGE_GetMaxLun (void)`	Returns the number of supported logical units.

6.7.3 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.

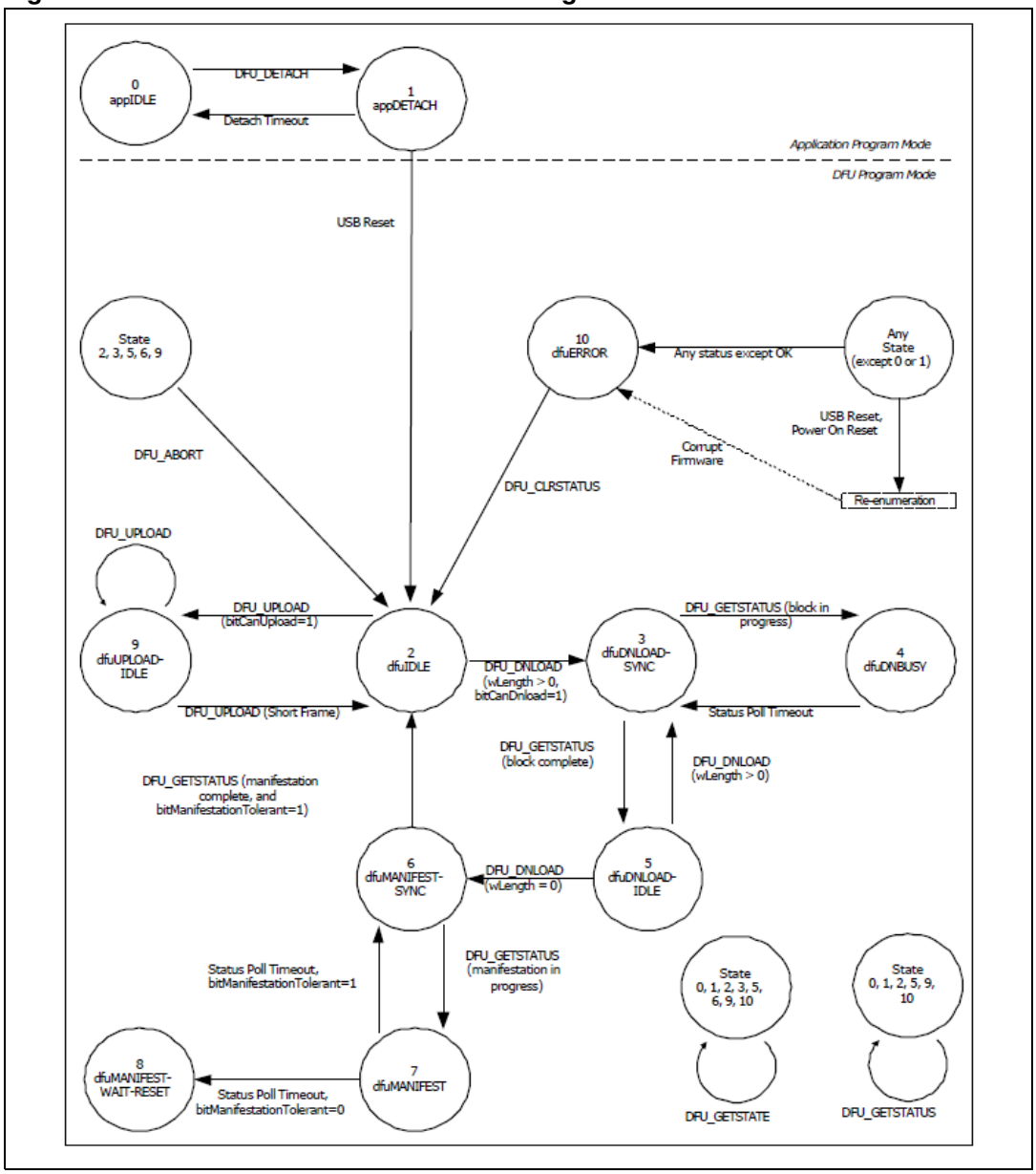
The DFU state machine is based on the following states:

Table 16. DFU states

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

The allowed state transitions are described in the specification document.

Figure 12. DFU Interface state transitions diagram



To protect the application from spurious access before initialization, the initial state of the DFU core (after startup) is `dfuERROR`. Then, the host has to clear this state (by sending a `DFU_CLRSTATE` 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	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 (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 memories (that is, internal Flash, OTP, etc.). 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	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 (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, the relative 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*).

The library provides two default memory drivers for internal Flash memory (*usbd_flash_if.c/.h*) and for OTP memory (*usbd_otp_if.c/.h*). But you can add other memories using the provided template file (*usbd_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	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, through 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 `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.*

In High speed mode, it is not possible to use DMA for writing/reading to/from Flash/OTP memories. In this case, an intermediate buffer is used to interface between memory and DMA controller. This may result in performance degradation for transfers relative to these memories in High speed mode. It is advised to disable DMA mode in this case (comment `USB_OTG_HS_INTERNAL_DMA_ENABLED` *define in file* usb_conf.h*).*

6.7.4 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 and does not allow High Speed transfers. Please refer to “USB Device Class Definition for Audio Devices V1.0 Mar 18, 98" for more details.

These aspects 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. This driver implements simple mechanism of synchronization relying on accuracy of the delivered I2S clock. At each start of frame, the driver checks if the consumption of the previous packet has been correctly performed and aborts it if it is still ongoing. To prevent any data overwrite, two main protections are used:

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

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

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

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

Table 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	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.
`static void AUDIO_Req_SetCurrent(void *pdev, USB_SETUP_REQ *req)`	Handles the `SET_CUR` Audio control request.

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 (*PeriodicTC) (uint8_t cmd);
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. *stm322xg_usb_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	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.
`static uint8_t PeriodicTC (uint8_t cmd)`	Handles the end of current packet transfer (not needed for the current version of the driver).
`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 (i.e. HW Codec, I2S interface, I2C control interface...). 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 *usbd_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_IN_EP` and `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

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

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

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	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.
`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, the relative field is set as NULL value.

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

Table 26. Configurable CDC parameters

Define	Parameter	Typical value	
		Full Speed	High Speed
CDC_DATA_IN_PACKET_SIZE	Size of each IN data packet	64	512
CDC_DATA_OUT_PACKET_SIZE	Size of each OUT data packet	64	512
CDC_IN_FRAME_INTERVAL	Interval time between IN packets sending.	5	40
APP_RX_DATA_SIZE	Total size of circular temporary buffer for IN data transfer.	2048	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	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.

CDC known limitations

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

6.7.6 Adding a custom class

To create a new custom class, the user has to add `USBD_CustomClass_cb` as described in *Section 6.5: USB device class interface*.

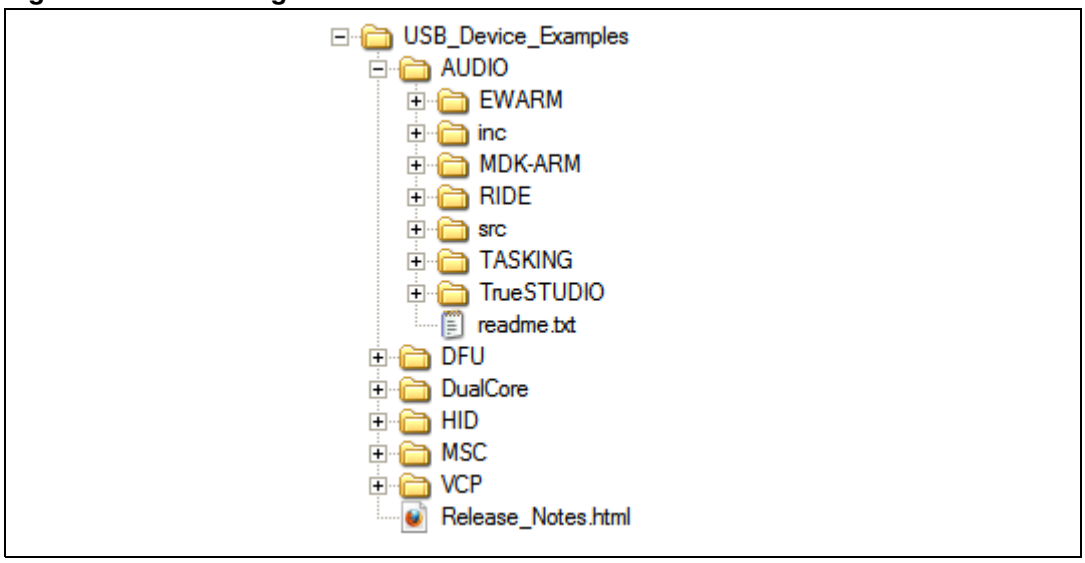
```
typedef struct _Device_cb
{
   uint8_t  (*Init) (void *pdev , uint8_t cfgidx);
   uint8_t  (*DeInit) (void *pdev , uint8_t cfgidx);
   /* Control Endpoints*/
   uint8_t  (*Setup) (void *pdev , USB_SETUP_REQ  *req);
   uint8_t  (*EP0_TxSent)   (void *pdev );
   uint8_t  (*EP0_RxReady)  (void *pdev );
   /* 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  (*IsoINIncomplete)  (void *pdev);
   uint8_t  (*IsoOUTIncomplete)  (void *pdev);

   uint8_t  *(*GetConfigDescriptor)( uint8_t speed ,
   uint16_t *length);
#ifdef USB_OTG_HS_CORE
   uint8_t  *(*GetOtherConfigDescriptor)( uint8_t speed ,
   uint16_t *length);
#endif

#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.

Additional functions could be added as the `IsoINIncomplete` and `IsoOUTIncomplete` could be eventually used to handle incomplete isochronous transfers (for more information, refer to the *USB audio device example*). `EP0_TxSent` and `EP0_RxReady` could be eventually used when the application needs to handle events occurring before the Zero Length Packets (see the *DFU example*).

6.8 Application layer description

Figure 13. Folder organization



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
- *stm32fxxx_it.c*: contains the system interrupt handlers
- *system_stm32fxxx.c*: system clock configuration file for STM32Fxxx devices.
- *usb_bsp.c*: contains the function implementation (declared in the *usb_bsp.h* in the USB OTG low level driver) to initialize the GPIO for the core, time delay methods 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:

- *stm32fxxx_it.h*: header file of the *stm32fxxx_it.c* file
- *usb_conf.h*: configuration files for the USB OTG low level driver.
- *usbd_conf.h*: configuration files for the USB device library.

Note: When using the USB OTG Full speed core, the user should use the CN8 connector on the STM322xG-EVAL and STM324xG-EVAL or the CN2 connector when the STM3210C-EVAL is used.

When using the USB OTG High speed core, the user should use the CN9 connector on the STM322xG-EVAL and STM324xG-EVAL boards.

6.9 Starting the USB device library

Since the USB Library can handle multi-core instances, the user **must first** define the core device handles.

Figure 14. Example of the define for core device handles

```
#ifdef USB_OTG_HS_INTERNAL_DMA_ENABLED
  #if defined ( __ICCARM__ ) /*!< IAR Compiler */
    #pragma data_alignment=4
  #endif
#endif /* USB_OTG_HS_INTERNAL_DMA_ENABLED */

__ALIGN_BEGIN USB_OTG_CORE_HANDLE     USB_OTG_dev __ALIGN_END ;
```

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 15. USBD_Init () function example

```
113
114   USBD_Init(&USB_OTG_dev,
115 #ifdef USE_USB_OTG_HS
116             USB_OTG_HS_CORE_ID,
117 #else
118             USB_OTG_FS_CORE_ID,
119 #endif
120             &USR_desc,
121             &USBD_MSC_cb,
122             &USR_cb);
123
124   while (1)
125   {
126     if (i++ == 0x100000)
127     {
128       STM_EVAL_LEDToggle(LED1);
129       STM_EVAL_LEDToggle(LED2);
130       STM_EVAL_LEDToggle(LED3);
131       STM_EVAL_LEDToggle(LED4);
132       i = 0;
133     }
134   }
135 }
```

6.10 USB device examples

Each project for an example based on a class is given with five configurations, as follows (exception made for USB device dual core example).

1. STM322xG-EVAL_USBD-HS: High-speed example on the STM322xG-EVAL board working with USB OTG HS core and the ULPI PHY
2. STM322xG-EVAL_USBD-FS: Full-speed example on the STM322xG-EVAL board working with USB OTG FS core and the embedded FS PHY
3. STM324xG-EVAL_USBD-HS: High-speed example on the STM324xG-EVAL board working with USB OTG HS core and the ULPI PHY
4. STM324xG-EVAL_USBD-FS: Full-speed example on the STM324xG-EVAL board working with USB OTG FS core and the embedded FS PHY
5. STM3210C-EVAL_USBD-FS: Full-speed example on the STM3210C-EVAL board working with USB OTG FS core and the embedded FS PHY.

For the High speed examples, the following features are selected in the *usb_config.h* file:

- `USB_OTG_HS_ULPI_PHY_ENABLED`: ULPI Phy is used.
- `USB_OTG_HS_INTERNAL_DMA_ENABLED`: internal DMA is used.
- `USB_OTG_HS_DEDICATED_EP1_ENABLED`: endpoint interrupts relative to the class are independent from the global USB OTG interrupt.

For the HID example, 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 [Key] button on the evaluation board.

Note: The USB device examples are using the lcd_log.c module to redirect the Library and User messages on the screen. Depending on the LCD cache depth used to scroll forward and backward within the messages, the application footprints are impacted. With bigger LCD cache depth, the RAM footprint is consequently increased. To prevent this additional RAM footprint, the user can redirect the Library and User messages on another terminal (HyperTerminal or LCD using the native display functions).

Library and user messages are located in the user callbacks in the application layer. They are not mandatory and they are used for information and debug purpose only. They can be modified or even removed.

6.10.1 USB mass storage device example

The Mass storage example uses the microSD Flash embedded in the STM322xG-EVAL, STM324xG-EVAL and STM3210C-EVAL evaluation boards 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.

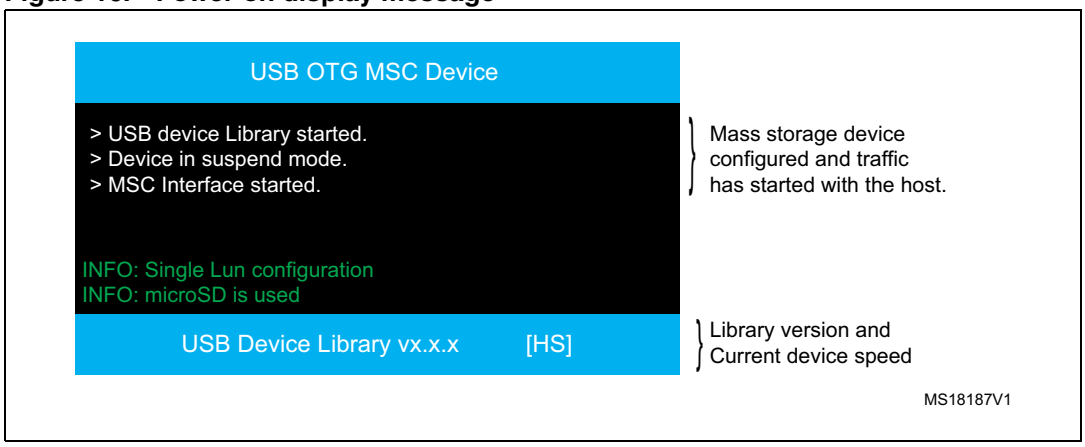
The mass storage example works in High and Full speed modes and has the following USB device information (*usbd_desc.c*).

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

#define USBD_LANGID_STRING              0x409
#define USBD_MANUFACTURER_STRING        "STMicroelectronics"
```

```
#define USBD_PRODUCT_HS_STRING         "Mass Storage in HS Mode"
#define USBD_SERIALNUMBER_HS_STRING    "00000000001A"

#define USBD_PRODUCT_FS_STRING         "Mass Storage in FS Mode"
   #define USBD_SERIALNUMBER_FS_STRING    "00000000001B"

#define USBD_CONFIGURATION_HS_STRING   "MSC Config"
#define USBD_INTERFACE_HS_STRING       "MSC Interface"

#define USBD_CONFIGURATION_FS_STRING   "MSC Config"
#define USBD_INTERFACE_FS_STRING       "MSC Interface"
```

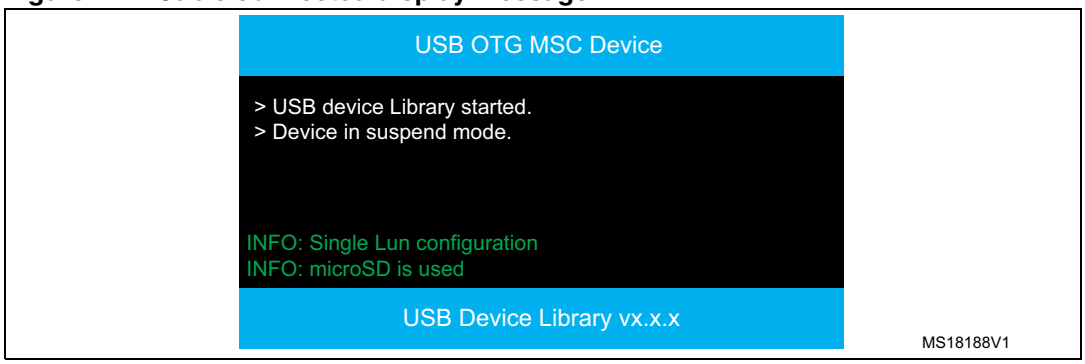
At power on, the LCD displays the following messages.

Figure 16. Power-on display message



When the USB cable is plugged in, the LCD shows the following messages.

Figure 17. Cable connected display message



6.10.2 USB human interface device example

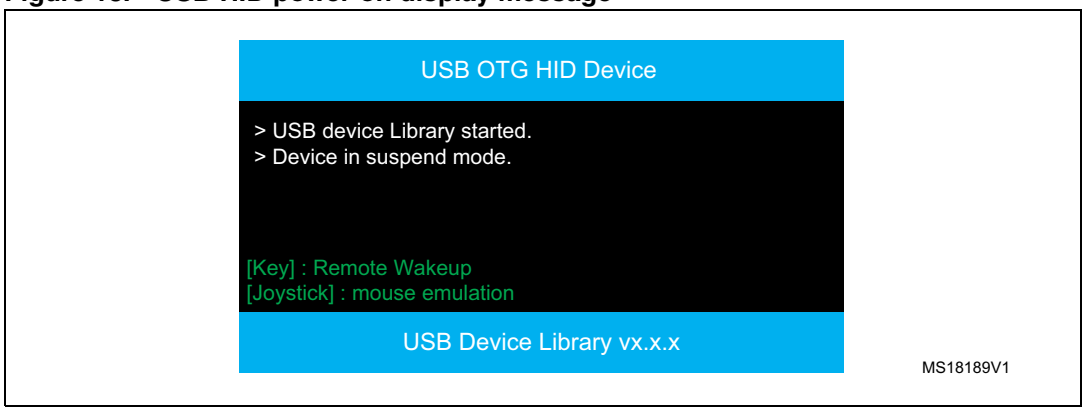
The HID example uses the joystick embedded in the STM322xG-EVAL, STM324xG-EVAL or STM3210C-EVAL evaluation boards.

The HID example works in High and Full speed modes and provides 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_HS_STRING          "Joystick in HS mode"
 #define USBD_SERIALNUMBER_HS_STRING     "00000000011B"

 #define USBD_PRODUCT_FS_STRING          "Joystick in FS Mode"
 #define USBD_SERIALNUMBER_FS_STRING     "00000000011C"

 #define USBD_CONFIGURATION_HS_STRING    "HID Config"
 #define USBD_INTERFACE_HS_STRING        "HID Interface"

 #define USBD_CONFIGURATION_FS_STRING    "HID Config"
 #define USBD_INTERFACE_FS_STRING        "HID Interface"
```

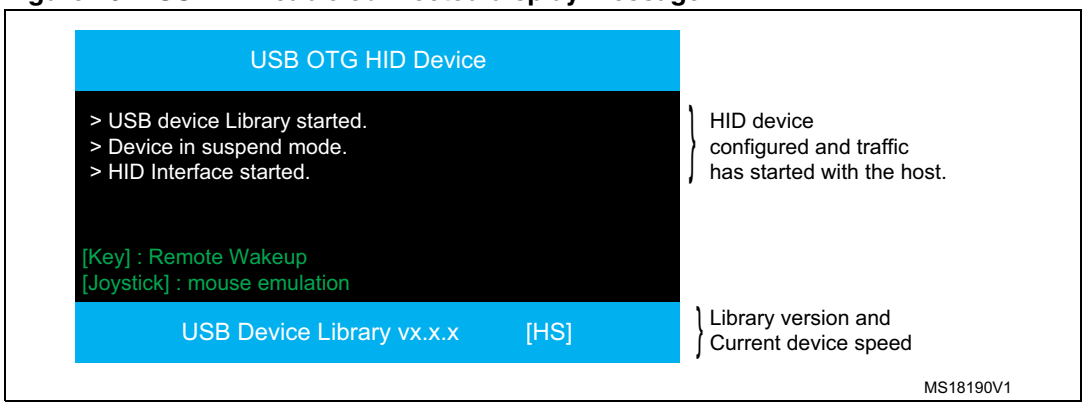
At power on, the LCD displays the following message.

Figure 18. USB HID power-on display message



When the USB cable is plugged in, the LCD displays the following messages.

Figure 19. USB HID cable connected display message



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

6.10.3 Dual core USB device example

The Dual core USB device example integrates the two mass storage and HID example described above in same project and uses the multi core support feature. The Mass storage device is connected to the High speed USB connector while the HID is connected to the Full Speed connector. Note that project comes with only two configurations for the *STM322xG-EVAL and STM324xG-EVAL* boards.

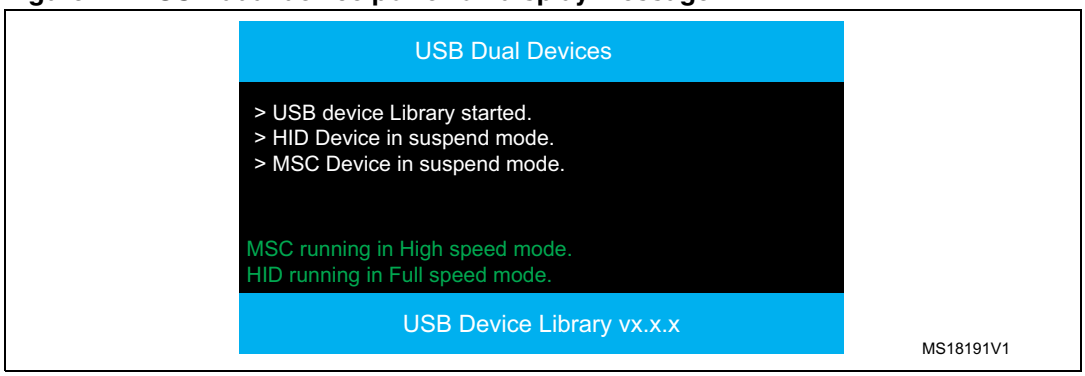
Figure 20. Dual core USB device example

```
81
82 USB_OTG_CORE_HANDLE          USB_OTG_FS_dev;
83 USB_OTG_CORE_HANDLE          USB_OTG_HS_dev;
84
85 /**
86   * @brief  Program entry point
87   * @param  None
88   * @retval None
89   */
90 int main(void)
91 {
92   uint32_t i = 0;
93
94   USBD_Init(&USB_OTG_FS_dev,
95             USB_OTG_FS_CORE_ID,
96             &USR_HID_desc,
97             &USBD_HID_cb,
98             &USR_FS_cb);
99
100  USBD_Init(&USB_OTG_HS_dev,
101            USB_OTG_HS_CORE_ID,
102            &USR_MSC_desc,
103            &USBD_MSC_cb,
104            &USR_HS_cb);
105
106  while (1)
107  {
108    if (i++ == 0x100000)
109    {
110      STM_EVAL_LEDToggle(LED1);
111      STM_EVAL_LEDToggle(LED2);
112      STM_EVAL_LEDToggle(LED3);
113      STM_EVAL_LEDToggle(LED4);
114      i = 0;
115    }
116  }
117 }
```

For this project, two USB device instances are declared and the `USBD_Init()` function is called twice to initialize each USB OTG core and to load the class callbacks for each instance.

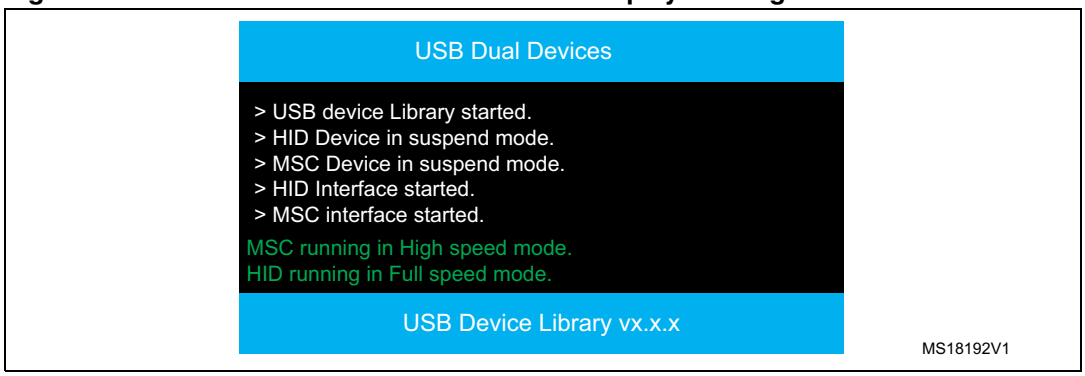
At power on, the LCD displays the following messages.

Figure 21. USB dual device power-on display message



When the USB cable is plugged in, the LCD shows the following messages:

Figure 22. USB dual device cable connected display message



6.10.4 USB device 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.

The supported memories for this example are:

- Internal Flash memory for STM32F105/7, STM32F2xx and STM32F4xx devices
- OTP memory for STM32F2xx and STM32F4xx devices.

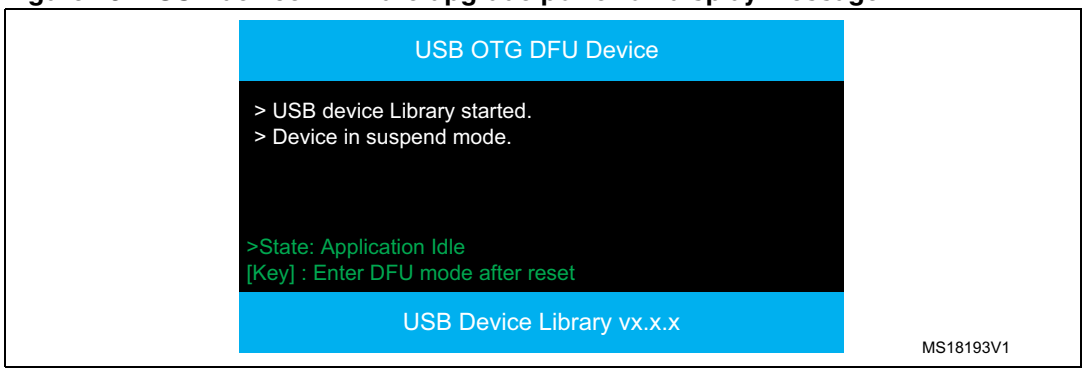
The DFU example works in High and Full speed modes and has the following USB device information.

```
#define USBD_VID                        0x0483
#define USBD_PID                        0xDF11

#define USBD_LANGID_STRING              0x409
#define USBD_MANUFACTURER_STRING        "STMicroelectronics"

#define USBD_PRODUCT_HS_STRING          "DFU in HS mode"
```

```
#define USBD_SERIALNUMBER_HS_STRING     "00000000010B"

#define USBD_PRODUCT_FS_STRING          "DFU in FS Mode"
#define USBD_SERIALNUMBER_FS_STRING     "00000000010C"

#define USBD_CONFIGURATION_HS_STRING    "DFU Config"
#define USBD_INTERFACE_HS_STRING        "DFU Interface"

#define USBD_CONFIGURATION_FS_STRING    "DFU Config"
#define USBD_INTERFACE_FS_STRING        "DFU Interface"
```

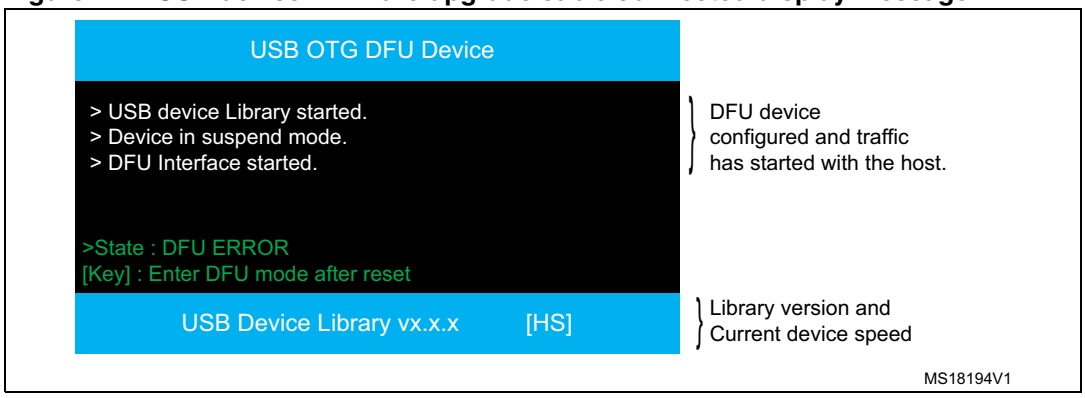
At power on, the LCD displays the following messages.

Figure 23. USB device firmware upgrade power-on display message



When the USB cable is plugged in, the LCD displays the following messages.

Figure 24. USB device firmware upgrade cable connected display message



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 (displayed in the footer on the LCD) 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 ST DFU 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 KEY button is pushed. If the KEY button is released after reset, the example jumps to user image application loaded in the internal Flash.

Note: In High speed mode, when DMA mode is enabled (define `USB_OTG_HS_INTERNAL_DMA_ENABLED` *in file* usb_conf.h*), the DMA cannot directly access (read/write) the internal Flash memory and the OTP memory (due to STM32F2xx and STM32F4xx product architecture). In this case, an intermediate buffer is used to store data before loading them to the memory or before sending them through DMA to the USB interface. This leads to an overall performance equivalent to Full Speed mode. However, if another memory is used (i.e. NOR Flash), it is possible to fully use DMA mode and get a better performance.*

6.10.5 USB virtual com port (VCP) device 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.

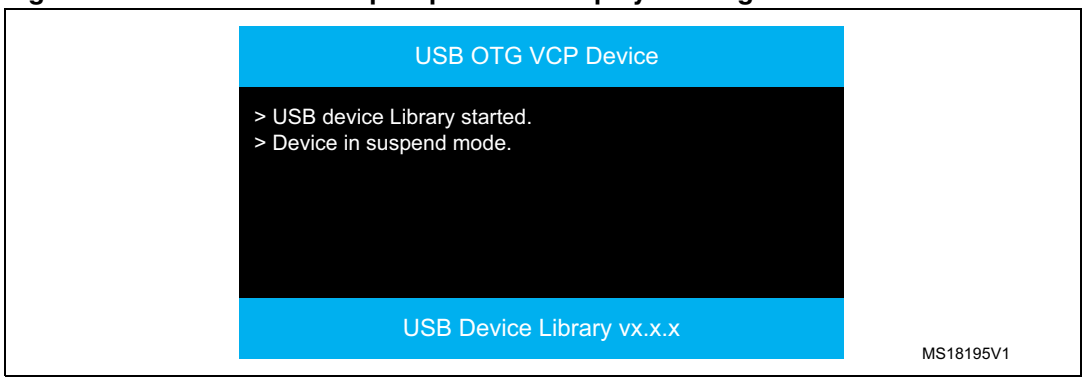
This example can be customized to communicate with interfaces other than USART.

The VCP example works in High and Full speed modes and 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_HS_STRING          "STM32 Virtual ComPort in HS mode"
#define USBD_SERIALNUMBER_HS_STRING     "00000000050B"

#define USBD_PRODUCT_FS_STRING          "STM32 Virtual ComPort  in FS Mode"
#define USBD_SERIALNUMBER_FS_STRING     "00000000050C"

#define USBD_CONFIGURATION_HS_STRING    "VCP Config"
#define USBD_INTERFACE_HS_STRING        "VCP Interface"

#define USBD_CONFIGURATION_FS_STRING    "VCP Config"
#define USBD_INTERFACE_FS_STRING        "VCP Interface"
```

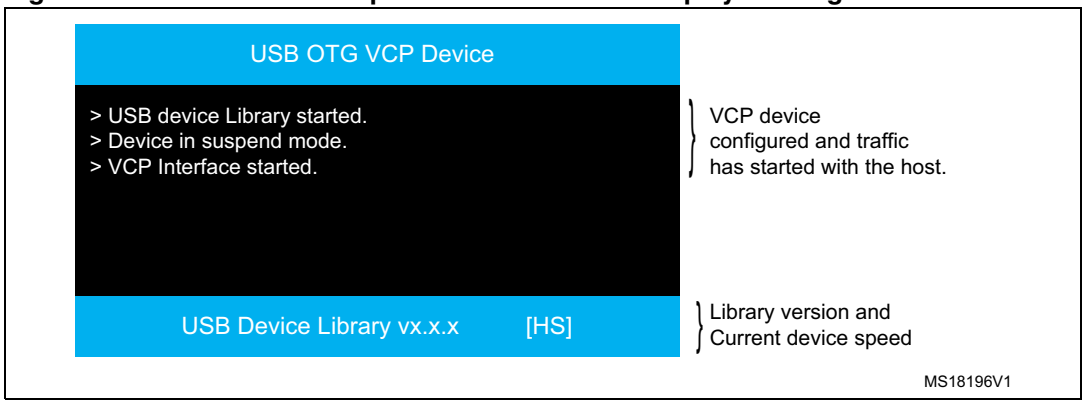
At power on, the LCD displays the following messages.

Figure 25. USB virtual com port power-on display message



When the USB cable is plugged in, the LCD displays the following messages.

Figure 26. USB virtual com port cable connected display message



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

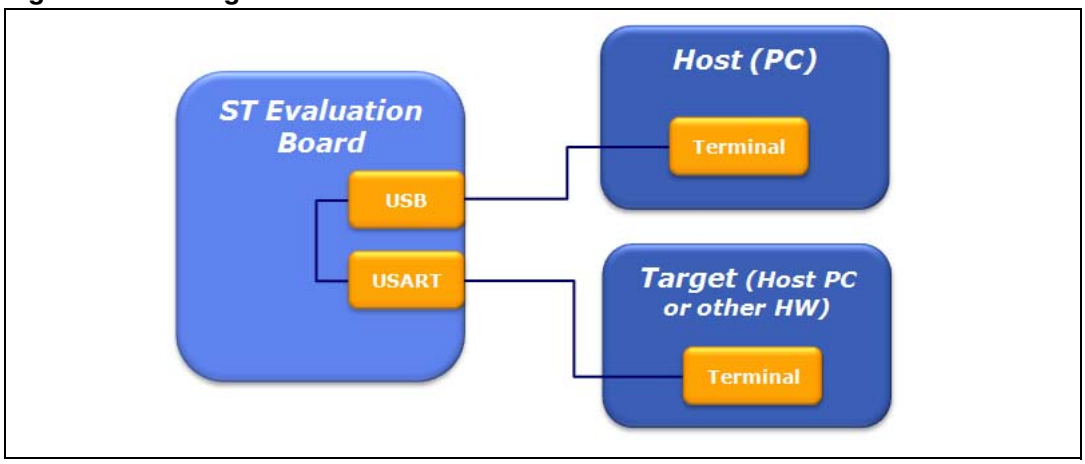


Figure 28. Configuration 1b: One single Host for USB and USART

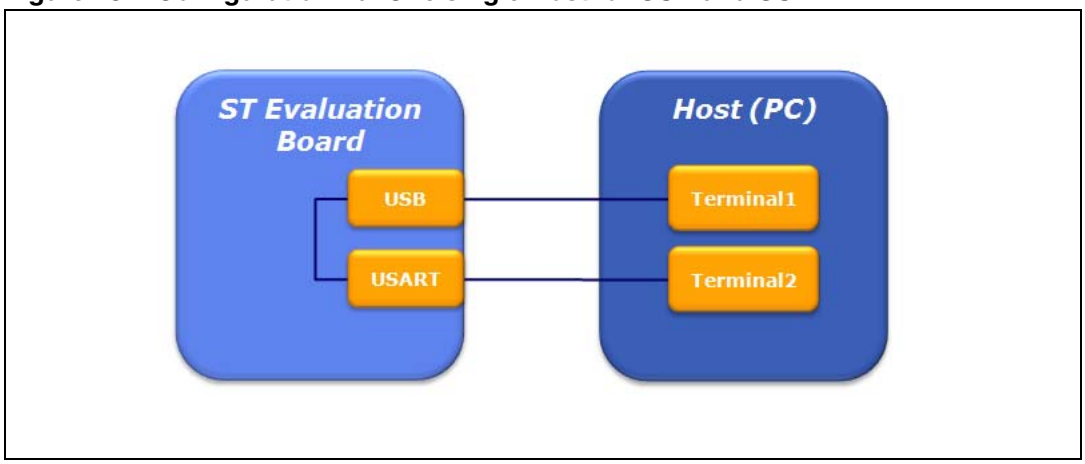
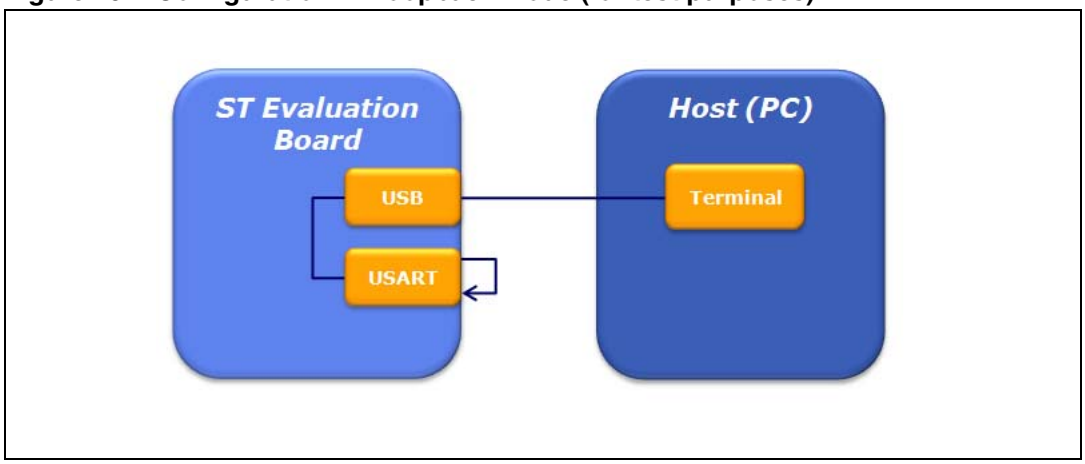


Figure 29. Configuration 2: Loopback mode (for test purposes)



6.10.6 USB audio device 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).

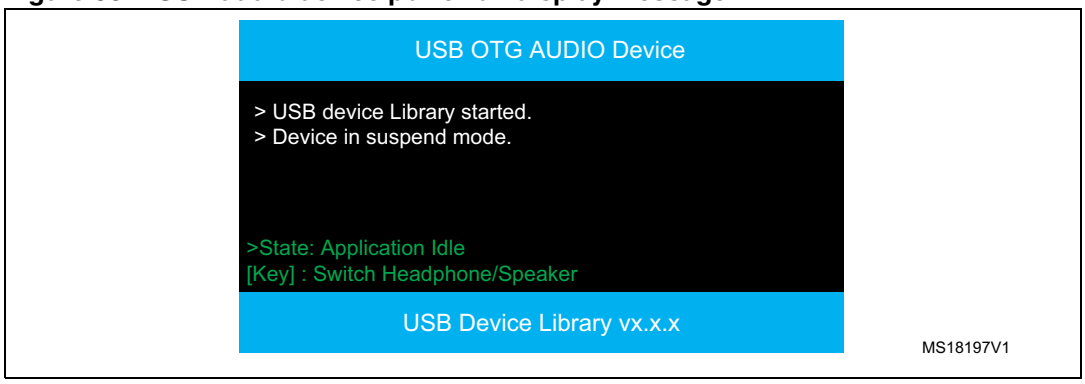
The Audio device is natively supported by most of operating systems (there is no need for specific driver setup).

The Audio device example works in full speed mode only (as specified in the USB Device Class Definition for Audio Devices V1.0 Mar 18, 98) and has the following USB device information.

```
#define USBD_VID                          0x0483
       #ifdef STM32F2XX
         #define USBD_PID                       0x5730
       #else
         #define USBD_PID                       0x5730
       #endif /* STM32F2XX */

      #define USBD_LANGID_STRING              0x409
      #define USBD_MANUFACTURER_STRING        "STMicroelectronics"

      #define USBD_PRODUCT_FS_STRING          "STM32 AUDIO Streaming in FS
Mode"
      #define USBD_SERIALNUMBER_FS_STRING     "00000000034E"

      #define USBD_CONFIGURATION_FS_STRING    "AUDIO Config"
      #define USBD_INTERFACE_FS_STRING        "AUDIO Interface"
```

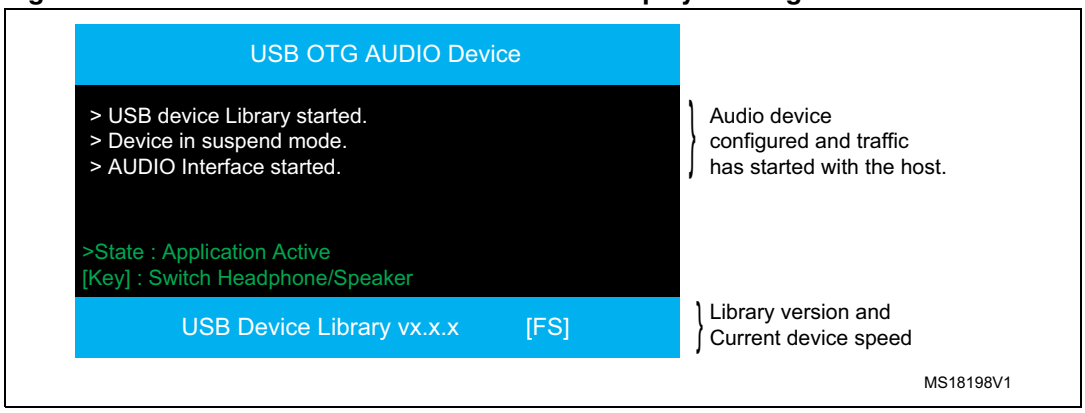
At power on, the LCD displays the following messages.

Figure 30. USB audio device power-on display message



When the USB cable is plugged in, the LCD displays the following messages.

Figure 31. USB audio device cable connected display message



When the Application is ready for audio streaming, the state footer (">State: …") indicates "Application Active" state. When an audio file is being played the state changes to "PLAYING" and when the audio file is paused or stopped, the state updates to "PAUSED".

The last line of the footer (on the LCD screen) indicates the current output state (Headphone or Speaker).

For STM32F2xx and STM32F4xx devices, the Headphone is selected as output by default. When pushing Key button the output is switched to Speaker (or to Headphone if the current output is the Speaker).

For STM32F105/7 devices, the default state is Automatic detection (when the Headphone is plugged in it is used as output, and when it is unplugged, output automatically switches to Speaker). When pushing Key button, the automatic detection is disabled and only the output set by the Key command is configured (Headphone or Speaker).

6.10.7 Known limitations

- If a low audio sampling rate is configured (define `USBD_AUDIO_FREQ` below 24 kHz) it may result in noise issue at pause/resume/stop operations. This is due to software timings tuning between stopping I2S clock and sending mute command to the external codec.
- Supported audio sampling rates are from: 96 kHz to 24 kHz (non-multiple of 1 kHz values like 11.025 kHz, 22.05 kHz or 44.1 kHz are not supported by this driver). For frequencies multiple of 1000 Hz, the Host will send integer number of bytes each frame (1 ms). When the frequency is not multiple of 1000Hz, the Host should send non integer number of bytes per frame. This is in fact managed by sending frames with different sizes (i.e. for 22.05 kHz, the Host will send 19 frames of 22 bytes and one frame of 23 bytes). This difference of sizes is not managed by the Audio core and the extra byte will always be ignored. It is advised to set a high and standard sampling rate in order to get best audio quality (i.e. 96 kHz or 48 kHz). Note that maximum allowed audio frequency is 96 kHz (this limitation is due to the codec used on the Evaluation board. The STM32 I2S cell enables reaching 192 kHz).
- For STM32F105/7 devices, the on-board Speaker output quality with 25MHz external quartz may be not sufficient (noise) due to lack of accuracy on audio output frequency using this quartz.

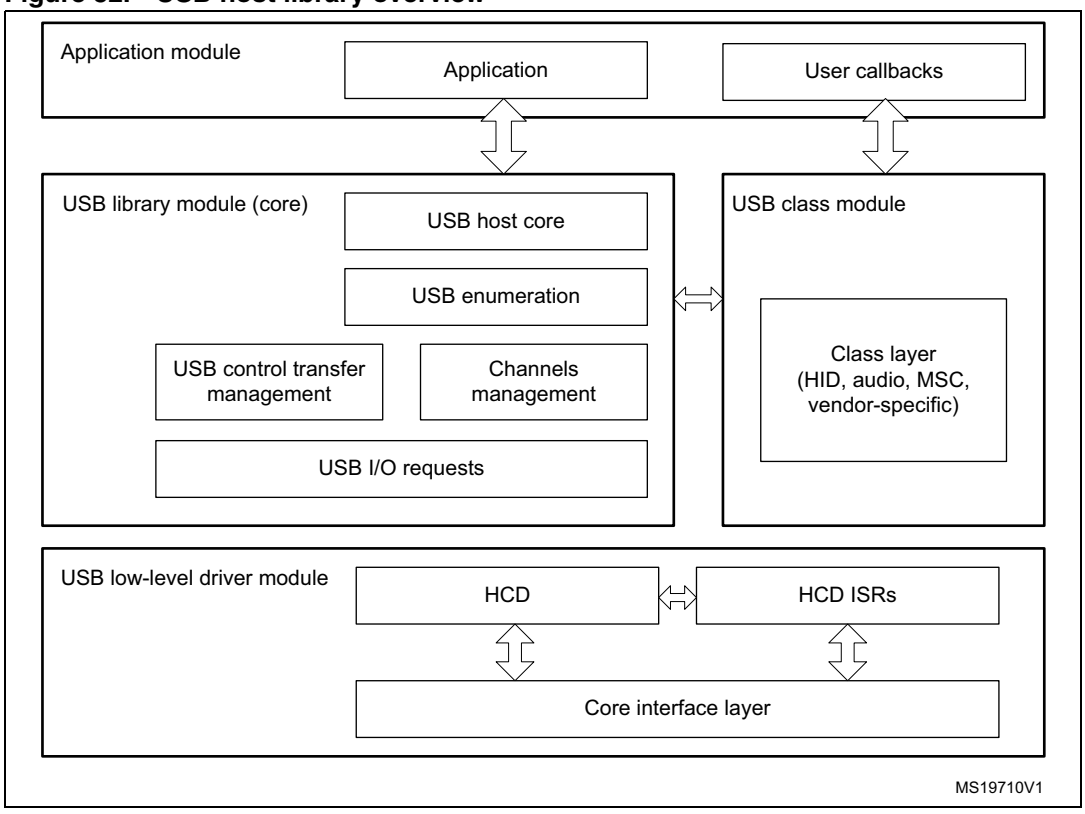
7 USB host library

The USB Host library:

- Supports multi packet transfer features enabling the transmission of large amounts of data; without having to split it into maximum packet size transfers.
- Uses configuration files to change the core and the library configuration without changing the library code (Read Only).
- 32-bit aligned data structures to handle DMA based transfer in High speed modes.
- Supports multi USB OTG core instances from user level.
- Built around global state machine.
- Fully compatible with real time operating system (RTOS).

7.1 Overview

Figure 32. USB host library overview



As shown in the above figure, the USB host library is composed of two main parts: the library core and the class drivers.

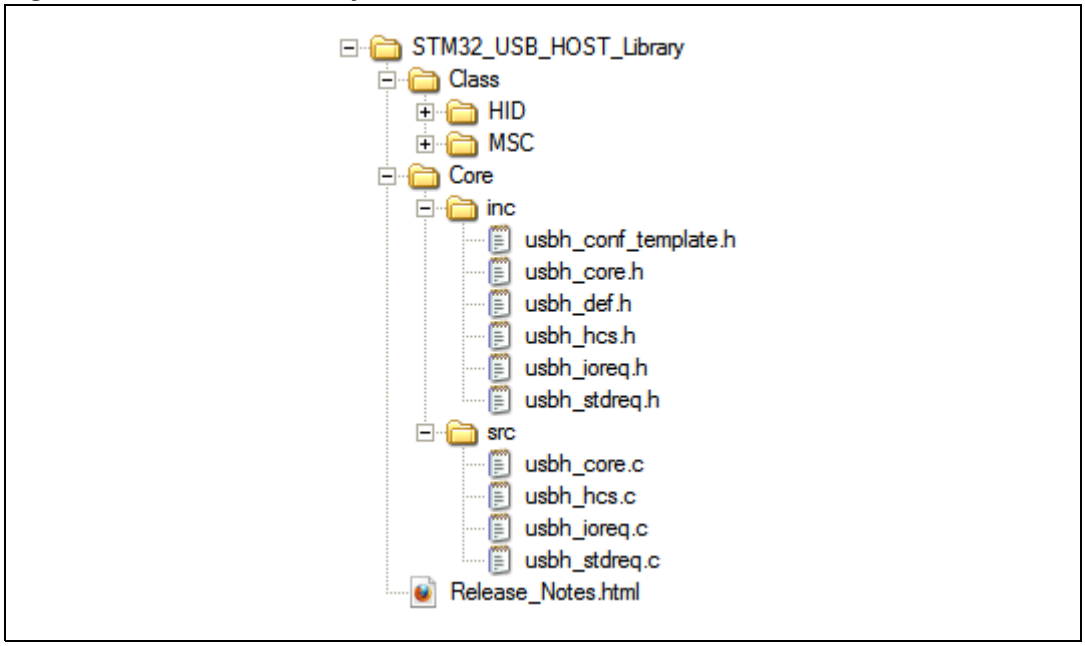
The library core is composed of five main blocks:

- Core host core
- USB enumeration
- USB control transfer management
- USB I/O requests
- Channels management

For all class-related operations, the core state machine hands over operation to a specific class driver. Two class drivers - HID and MSC - are implemented. These class drivers use core layer services for communicating with the low-level driver. Both the core and the class drivers communicate with the user application mainly through defined callback functions. The various host library blocks are described below.

7.2 USB host library files

Figure 33. USB host library file tree structure



The USB Host library is based on the generic USB OTG low level driver which supports Host, device and OTG modes and works for High speed, Full speed and Low speed (for host mode).

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

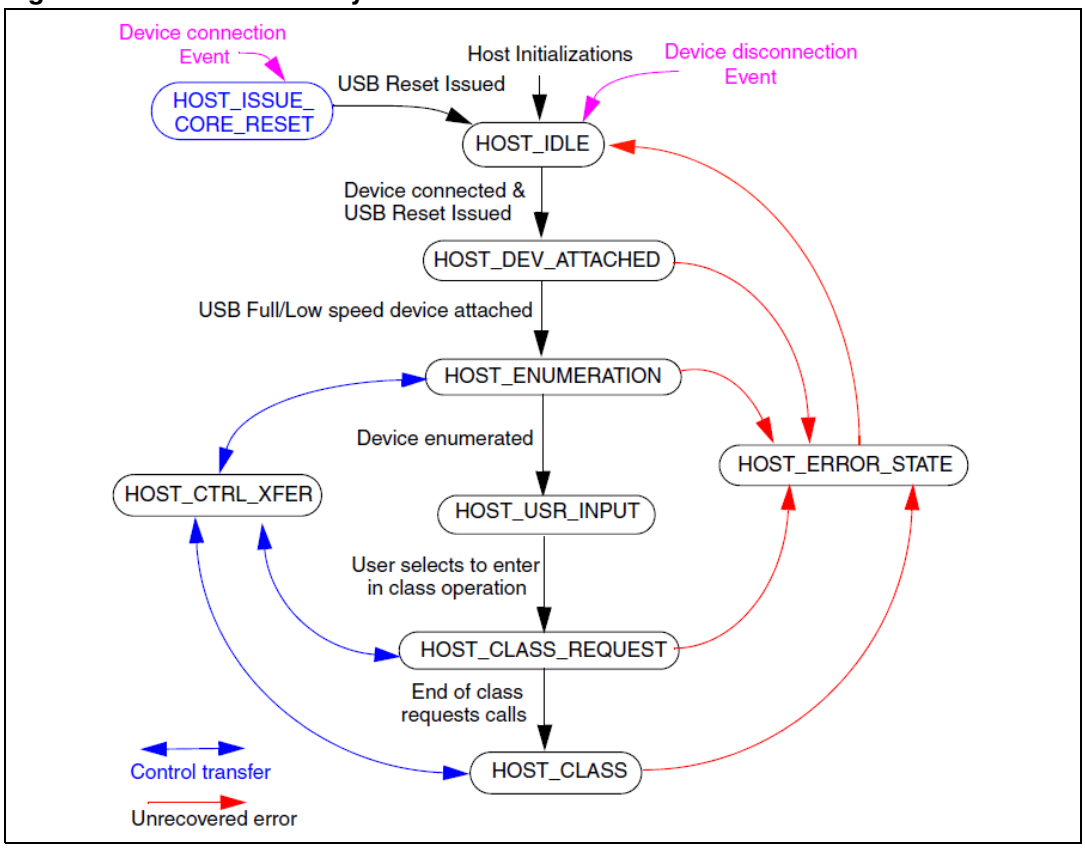
The **Class** folder contains all the files relative to the class implementation and meets with the specification of the protocol built in these classes.

7.3 USB host library description

7.3.1 Host core state machine

The following figure describes the library state machine.

Figure 34. USB host library state machine



The core state machine shows nine states:

- **HOST_IDLE**: after host initialization, the core starts in this state, where it polls for a USB device connection. This state is also entered when a device disconnection event is detected, and also when an unrecovered error occurs.
- **HOST_ISSUE_CORE_RESET**: this state is entered when a device is connected in order to generate a USB bus RESET.
- **HOST_DEV_ATTACHED**: the core enters this state when a device is attached. When a device is detected, the state machine moves to the HOST_ENUMERATION state.
- **HOST_ENUMERATION**: in this state, the core proceeds with a basic enumeration of the USB device. At the end of enumeration process, the default device configuration (configuration 0) is selected.
- **HOST_USR_INPUT**: this is an intermediary state which follows the enumeration and which includes a wait for user input in order to start the USB class operation.
- **HOST_CLASS_REQUEST**: starting from this state, the class driver takes over, and a class request state machine is called in order to handle all the initial class control requests (ex: Get_Report_Descriptor for HID). After finishing the required class requests, the core moves to the HOST_CLASS state.

- **HOST_CLASS**: in this state, the class state machine is called for class-related operation (non-control and control operation).
- **HOST_CTRL_XFER**: this state is entered whenever there is a need for a control transfer.
- **HOST_ERROR_STATE**: this state is entered whenever there is an unrecovered error from any library state machine. In this case, a user-callback function is called (for example for displaying an unrecovered error message). Then the host library is re-initialized.

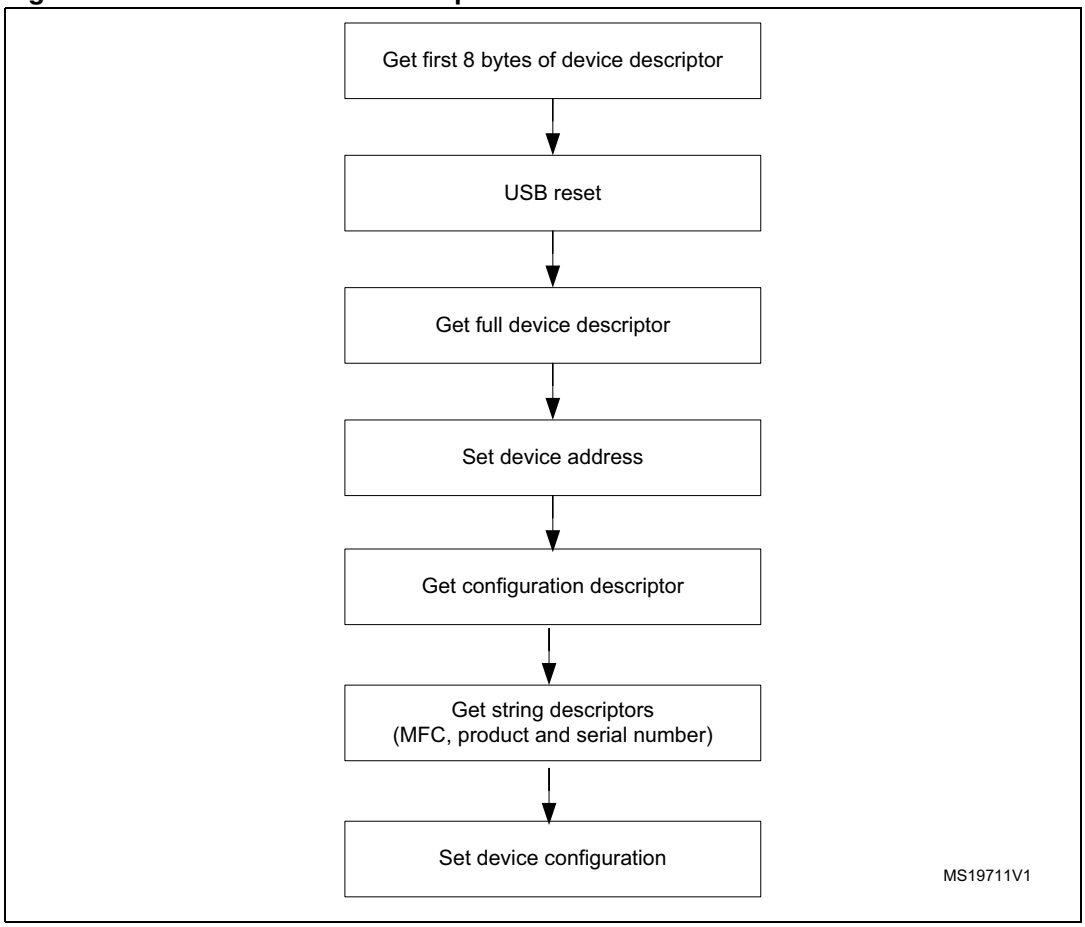
The core state machine process is implemented by the `USBH_Process` function. This function should be called periodically from the application main loop. The initialization of the USB host library is implemented by the `USBH_init` function. This function should be called from the user application during initialization. For further details on this function, refer to *Section 7.7.1: Library user API on page 88*.

7.3.2 Device enumeration

After detecting a device, the host library proceeds with a basic enumeration of the device.

The following diagram shows the different steps involved in the device enumeration.

Figure 35. Device enumeration steps



The enumeration state machine is implemented in the `USBH_HandleEnum` library function, which is called from the core state machine process. `USBH_HandleEnum` function calls the following library routines (implemented in file *usbh_stdreq.c*):

A user callback will be called at the end of enumeration phase in order to enable the user to process the descriptor information (such as displaying descriptor data, for example).

7.3.3 Control transfer state machine

The control transfer state machine is entered from the core or class driver whenever a control transfer is required. This state machine implements the standard stages for a control transfer, i.e. the setup stage, the optional data stage and the status stage.

The control transfer state machine is implemented in the `USBH_HandleControl` function. It is called from the core state machine process.

7.3.4 USB I/O request module

The USB I/O request module is located in the low layer of the core. It interfaces with the USB low-level driver for issuing control, bulk or interrupt USB transactions.

7.3.5 Host channel control module

The host channel control module is located in the lower layer of the core. It allows the configuration of a host channel for a particular operation (control, bulk or interrupt transfer type) and it assigns a selected host channel to a device endpoint for creating a USB pipe.

7.3.6 USB host library configuration

The USB host library can be configured using the *usbh_conf.h* file (a template configuration file is available in the "*Libraries\STM32_USB_Host_Library\Core*" directory of the library).

```
#define USBH_MAX_NUM_ENDPOINTS              2
#define USBH_MAX_NUM_INTERFACES             1
#define USBH_MSC_MPS_SIZE                   0x200
```

7.4 USB host library functions

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

Table 29. USB host core files

Files	Description
usbh_core (.c, .h)	This file contains the functions for handling all USB communication and state machine.
usbh_stdreq(.c, .h)	This file includes the chapter 9 request implementation.
usbh_ioreq (.c, .h)	This file handles the generation of the USB transactions.
usbh_hcs (.c, .h)	This file handles the host channel allocation and triggers processes.
usbh_conf.h	This file contains the configuration of the device interface number, configuration number and maximum packet size.

Table 30. USB I/O request module

Function	Description
USBH_CtlSendSetup	Issues a setup transaction.
USBH_CtlSendData	Issues a control data OUT stage transaction.
USBH_CtlReceiveData	Issues a control data IN stage transaction.
USBH_CtlReq	High level function for generating a control transfer (setup, data, status stages).
USBH_BulkSendData	Issues a bulk OUT transaction.
USBH_BulkReceiveData	Issues a bulk IN transaction.
USBH_InterruptSendData	Issues an interrupt OUT transaction.
USBH_InterruptReceiveData	Issues an interrupt IN transaction.

Table 31. Host channel control module

Function	Description
USBH_Open_Channel	Opens and configures a new host channel.
USBH_Modify_Channel	Modifies an existing host channel.
USBH_Alloc_Channel	Assigns a host channel to a device endpoint (creation of a USB).
USBH_Free_Channel	Frees a host channel.
USBH_DeAllocate_AllChannel	Frees all host channels (used during de-initialization phase).

Table 32. Standard request module

Function	Description
USBH_Get_CfgDesc	Gets a configuration descriptor request.
USBH_Get_DevDesc	Gets a device descriptor request.
USBH_Get_StringDesc	Gets a string descriptor request.
USBH_GetDescriptor	Generic get descriptor request.
USBH_SetCfg1)	Sets a configuration request.
USBH_SetAddress2)	Sets an address request.
USBH_ClrFeature	Clears the feature request.

Note: USBH_SetCfg selects the default configuration (configuration 0).

USBH_SetAddress sets the device address to 0x1.

7.5 USB host class interface

At the end of the enumeration, the core calls a specific class driver function to manage all class-related operations.

Note: The proper class driver selection is not based on the result of device enumeration, but it is "pre-defined" when initializing the host library by calling the `USBH_Init` function.

A class driver is implemented using a structure of type `USBH_Class_cb_TypeDef`:

```
typedef struct _Device_cb
{
      USBH_Status (*Init) (USB_OTG_CORE_HANDLE *pdev ,
USBH_DeviceProp_TypeDef *hdev);
      void (*DeInit) (USB_OTG_CORE_HANDLE *pdev , USBH_DeviceProp_TypeDef
*hdev);
     USBH_Status (*Requests)(USB_OTG_CORE_HANDLE *pdev,
USBH_DeviceProp_TypeDef;*hdev);
     USBH_Status (*Machine) (USB_OTG_CORE_HANDLE *pdev ,
USBH_DeviceProp_TypeDef,*hdev);
}
 USBH_Class_cb_TypeDef;
```

The structure members are described below:

- `Init`: this function is called at the startup of a class operation for assuring all required initializations. This includes:
  - Parsing interface and endpoint descriptors (please note that the current USB host library supports only one interface).
  - Opening and allocating host channels for non-control endpoints,
  - Calling a user callback, in case the device is not supported by the class.
- `Denit`: this function is called for freeing allocated host channels when re-initializing the host. It is called when a device is unplugged or in case of unrecovered error.
- `Requests`: this function implements the class request state machine. It is called during the HOST_CLASS_REQUEST state. It is used to process initial class requests.
- `Machine`: implements the class core state machine. It is called during the HOST_CLASS core state.

7.6 USB host classes

7.6.1 Mass storage class

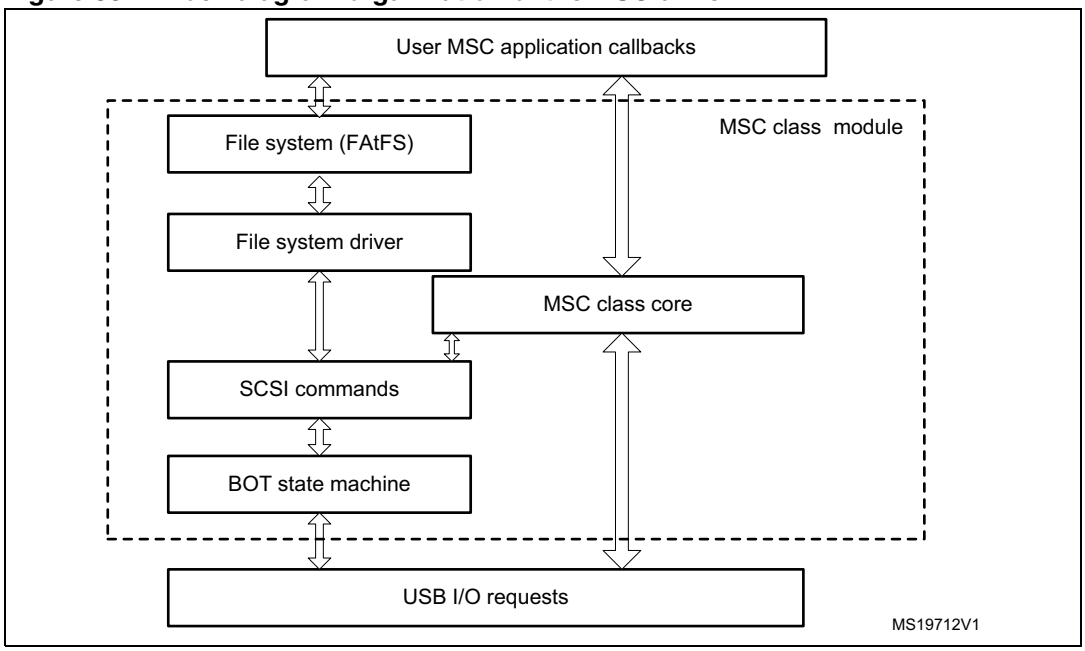
The mass storage class driver is used to support the common USB flash driver, using the BOT "Bulk-Only Transport" protocol and the Transparent SCSI command set. The following modules, located in the "*Libraries\STM32_USB_HOST_Library\Class\MSC*" folder, are used to implement the MSC driver.

Table 33. Modules

Module	Description
usbh_msc_core.c /.h	MSC core state machine implementation.
usbh_msc_bot.c /.h	BOT (Bulk-Only Transport) protocol implementation.
usbh_msc_scsi.c /.h	SCSI command implementation.
usbh_msc_fatfs.c/.h	Functions for interfacing with a file system for file access operations.

The block diagram in the following figure shows the interactions between these modules.

Figure 36. Block diagram organization of the MSC driver



Operation flow description:

The MSC core state machine starts with the required device initializations, which are:

- Issuing `GET_MAX_LUN` class requests for detecting the number of device logical units present on the device. Please note that only devices with one logical unit are supported.
- Issuing `BOT_RESET` class requests for resetting the device BOT state machine.
- Issuing SCSI commands: `MODE_SENSE` for detecting if the device is write-protected and `READ_CAPACITY` for detecting the size of the Flash pendrive. After the above device initializations, the MSC core state machine calls the application user callback.

The user callback can perform any type of file access into the used file system. This operation is translated into a logical page read or write operation. The file system interface provides the connection between the used file system and the MSC driver.

At the SCSI level, the logical page read or write operations are converted into SCSI commands: READ(10) or WRITE(10). These commands are transferred to the Flash pendrive device using the Bulk-Only Transport protocol.

The BOT layer state machine issues the required Bulk IN and Bulk OUT transactions using the core USB I/O request module. Each MSC module is described below.

MSC core module

The MSC core module “usb_msc_core.c” implements the MSC driver, which is defined in the structure `USBH_MSC_cb` of type `USBH_Class_cb_TypeDef` (see *Section 7.5*).

```
USBH_Class_cb_TypeDef USBH_MSC_cb =
{
USBH_MSC_InterfaceInit,
USBH_MSC_InterfaceDeInit,
USBH_MSC_ClassRequest,
USBH_MSC_Handle,
};
```

Table 34. MSC core module description

Function	Description
`USBH_MSC_InterfaceInit`	Parses interface and endpoint descriptors and configures host channels (bulk IN and bulk OUT pipes).
`USBH_MSC_InterfaceDeInit`	De-initialization routine (freeing host channels).
`USBH_MSC_ClassRequest`	In case of MSC, this function only moves the library core state machine to the `HOST_CLASS` state.
`USBH_MSC_Handle`	Implements the MSC handler core state machine.
`USBH_MSC_Issue_BOTReset`	Issues a BOT reset class request.
`USBH_MSC_Issue_GETMaxLUN`	Issues a `GET_MAX_LUN` class request.
`USBH_MSC_ErrorHandle`	MSC error handling.

MSC BOT module

The MSC “Bulk-Only Transport” (BOT) module implements the transport protocol for sending the SCSI commands (such as READ (10) or WRITE(10)). This module is implemented in the “usbh_msc_bot.c” file. For details about the BOT protocol, please refer to the usb.org mass storage class document.

The BOT module has the following functions.

Table 35. MSC BOT module description

Function	Description
`USBH_MSC_Init`	Initialize BOT state machine.
`USBH_MSC_HandleBOTXfer`	BOT transfer state machine.

MSC SCSI module

The SCSI (Small Computer System Interface) module *usb_msc_scsi.c* stands on top of the BOT. It implements the set of SCSI commands required to access the Flash pendrives.

Table 36. MSC SCSI commands

Function	Description
USBH_MSC_Read10	Command for logical Block Read.
USBH_MSC_Write10	Command for logical Block Write.
USBH_MSC_TestUnitReady	Command for checking Device Status.
USBH_MSC_ReadCapacity10	Command for requesting the Device Capacity.
USBH_MSC_ModeSense6	Command for checking the Write-protect status of the mass storage device.
USBH_MSC_RequestSense	Command for getting error information.

MSC file system interface module

The MSC file system interface module "*usbh_msc_fatfs.c*" allows interfacing of file systems with the MSC driver. This module should be ported to the selected file system.

The current USB host library package comes with the open source; FatFS file system support (see next section for an overview about the FatFS API). The functions implemented in the file system interface are:

Table 37. MSC file system interface functions

Function	Description
disk_initialize	Initialize disk drive.
disk_read	Interface function for a logical page read.
disk_write	Interface function for a logical page write.
disk_status	Interface function for testing if unit is ready.
disk_ioctl	Control device-dependent features.

Note: For the FatFS file system, the page size is fixed to 512 bytes. USB pendrives with Flash memories having higher page size are not supported.

FatFS application programming interface

Table 38. FatFS API commands

Function	Description
f_mount	Register/Unregister a work area.
f_open	Open/Create a file.
f_close	Close a file.
f_read	Read file.
f_write	Write file.
f_lseek	Move read/write pointer, expand file size.
f_truncate	Truncate file size.
f_sync	Flush cached data.
f_opendir	Open a directory.
f_readdir	Read a directory item.
f_getfree	Get free clusters.
f_stat	Get file status.
f_mkdir	Create a directory.
f_unlink	Remove a file or directory.
f_chmod	Change attribute.
f_utime	Change timestamp.
f_rename	Rename/Move a file or directory.
f_mkfs	Create a file system on the drive.
f_forward	Forward file data to the stream directly.
f_chdir	Change current directory.
f_chdrive	Change current drive.
f_getcwd	Retrieve the current directory.
f_gets	Read a string.
f_putc	Write a character.
f_puts	Write a string.
f_printf	Write a formatted string.

7.6.2 HID class

The HID class implementation in v1.0 of the USB host library is used to support HID boot mouse and keyboard devices. HID reports are received using the interrupt IN transfer.

The following modules, located in the *Libraries\STM32_USB_HOST_Library\Class\HID* folder, are used to implement the HID class.

Table 39. HID class modules

File	Description
usbh_hid_core.c /.h	This module implements the HID class core state machine.
usbh_hid_mouse.c /.h	HID mouse specific routines.
usbh_hid_keybd.c /.h	HID keyboard specific routines.

The main functions of each module are described below.

HID class core

The HID core module *usb_hid_core.c* implements the HID class driver structure `USBH_HID_cb` of type `USBH_Class_cb_TypeDef` (see *Section 7.5*).

```
USBH_Class_cb_TypeDef  USBH_HID_cb =
{
USBH_HID_InterfaceInit,
USBH_HID_InterfaceDeInit,
USBH_HID_ClassRequest,
USBH_HID_Handle
};
```

The following table summarizes the functions implemented in the HID core module.

Table 40. MSC core module functions

Function	Description
`USBH_HID_InterfaceInit`	Parses interface and endpoint descriptors and configures a host channel in order to have an interrupt IN pipe (for getting HID reports).
`USBH_HID_InterfaceDeInit`	Frees the allocated interrupt IN pipe.
`USBH_HID_ClassRequest`	Implements a state machine of the required class requests for HID mouse and keyboard devices (ex: getting HID report descriptors, setting IDLE time, setting Protocol).
`USBH_HID_Handle`	HID class core state machine (processing of interrupt IN transfers).
`USBH_Get_HID_ReportDescriptor`	Class request for getting HID report descriptor.
`USBH_ParseClassDesc`	Function used for parsing HID report descriptor.
`USBH_Set_Idle`	Class request for setting IDLE time.
`USBH_Set_Report`	Class request for sending Report OUT data (not used in the demonstration software).
`USBH_Set_Protocol`	Class request for setting the HID protocol: Boot or Report(1).

1. USB_Set_Protocol is called to set the Boot protocol mode.

HID mouse and keyboard specific management

The mouse or keyboard device is detected when parsing the interface descriptor in the `USBH_HID_InterfaceInit` function.

The specific initialization for each type of device and the decoding of the received report IN data is performed by two functions which are declared in a structure of type `HID_cb_TypeDef`, which is defined as follows:

```
typedef struct HID_cb
{
void (*Init)(void);
void (*Decode) (uint8_t *data);

} HID_cb_TypeDef;
```

The implementation of the above structures in case a mouse or keyboard is respectively found in HID_MOUSE_cb and HID_MOUSE_cb is as follows:

```
HID_cb_TypeDef HID_MOUSE_cb =
{
MOUSE_Init,
MOUSE_Decode,
};

HID_cb_TypeDef HID_KEYBRD_cb=
{
KEYBRD_Init,
KEYBRD_Decode
};
```

Table 41. Mouse and keyboard initialization & HID report decoding functions

Function	Description
`MOUSE_Init`	Initialization routine for USB mouse.
`MOUSE_Decode`	HID report decoding for mouse (decoding mouse x, y positions, pressed buttons).
`KEYBRD_Init`	Initialization routine for USB keyboard.
`KEYBRD_Decode`	HID report decoding for keyboard (decoding of the key pressed on the keyboard).

Note: You can select AZERTY or QWERTY keyboard through the defines `QWERTY_KEYBOARD` and `AZERTY_KEYBOARD` in the usbh_hid_keybd.h *file.*

7.7 USB host user interface

7.7.1 Library user API

The library user API functions are limited to the two following functions:

- `void USBH_Process (void)`: this function implements core state machine process. It should be called periodically from the user main loop.
- `USBH_Init`: this function should be called for the initialization of the USB host hardware and library.

`USBH_Init` has the following function prototype:

```
void USBH_Init (USB_OTG_CORE_HANDLE *pdev,
                USB_OTG_CORE_ID_TypeDef coreID,
                USBH_HOST *phost,
                USBH_Class_cb_TypeDef *class_cb,
                USBH_Usr_cb_TypeDef *usr_cb);
```

- `pdev`: pointer on the USB host core register structure
- `CoreID`: USB OTG core identifier (select the USB core to be used) .
- `phost`: pointer on the USB host machine structure (reserved for future use).
- `class_cb`: pointer to a class structure of type `USBH_Class_cb_TypeDef`. It can be either `USBH_MSC_cb` for handling MSC devices or `USBH_HID_cb` for handling HID mouse/keyboard devices.
- `usr_cb`: pointer to a structure of type `USBH_Usr_cb_TypeDef`. This structure defines class-independent callbacks (see *Class-independent callback functions on page 89*).

7.7.2 User callback functions

User callbacks are declared in the *usbh_usr.c* user template file. Two types of user callbacks are defined:

- Callback functions related to the class operations (MSC or HID).
- Callback functions independent from class operations. They are mainly called during the enumeration phase. These callbacks are defined in a structure of type `USBH_Usr_cb_TypeDef`. They are generally used to show messages in the different enumeration stage.

7.7.3 Class callback functions

MSC user callback functions

For MSC, the following callback is used: `USBH_USR_MSC_Application ()`. After the end of the class initializations, this function is called by the MSC state machine in order to give hand to the user for file system access operations.

In this callback, the user can implement any access to the FAT file system (file open, file read, file write...) using the FAT FS file system API. The user can also have access to a structure variable exported from the library MSC class driver: `USBH_MSC_Param`.

This variable provides some information about the mass storage key. It is defined using a structure of type `MassStorageParameter_TypeDef`, as described below:

```
typedef struct __MassStorageParameter
{
uint32_t MSCapacity; /*MS device capacity in bytes */
uint32_t MSSenseKey; /*Request Sense SCSI command returned value */
uint16_t MSPageLength; /* MS device Page length */
uint8_t MSBulkOutEp; /* Bulk OUT endpoint address */
uint8_t MSBulkInEp; /*Bulk IN endpoint address */
uint8_t MSWriteProtect; /*Write protection status, 0: non protected,
1:protected */
} MassStorageParameter_TypeDef;
```

HID user callback functions

For the HID class, the following callbacks are defined:

- `void USR_MOUSE_Init(void)`: user initialization for mouse application.
- `void USR_KEYBRD_Init(void)`: user initialization for keyboard application.
- `void USR_MOUSE_ProcessData(HID_MOUSE_Data_TypeDef *data)`: this callback is called when an input parameter data of type `HID_MOUSE_Data_TypeDef` (see Note below) is available.
- `void USR_KEYBRD_ProcessData (uint8_t data)`: this callback is called when a new ASCII character is typed. The character is received in input parameter data.

Note: HID_MOUSE_Data_TypeDef is defined as follows:

```
typedef struct _HID_MOUSE_Data
{
uint8_t x;
uint8_t y;
uint8_t z; /* Not Supported */
uint8_t button; /*Bitmap showing pressed buttons 1:pressed, 0: non pressed
*/
} HID_MOUSE_Data_TypeDef;
```

Class-independent callback functions

The class-independent callback functions are defined in a structure of type `USBH_Usr_cb_TypeDef` as follows:

```
typedef struct _USBH_USR_PROP
{
void (*Init)(void);
void (*DeviceAttached)(void);
void (*ResetDevice)(void);
void (*DeviceDisconnected)(void);
void (*OverCurrentDetected)(void);
void (*DeviceSpeedDetected)(uint8_t DeviceSpeed);
void (*DeviceDescAvailable)(void *);
void (*DeviceAddressAssigned)(void);
```

```
void (*ConfigurationDescAvailable)(USBH_CfgDesc_TypeDef *,
USBH_InterfaceDesc_TypeDef *,
USBH_EpDesc_TypeDef *);
void (*ManufacturerString)(void *);
void (*ProductString)(void *);
void (*SerialNumString)(void *);
void (*EnumerationDone)(void);
USBH_USR_Status (*UserInput)(void);
int (*USBH_USR_MSC_Application) (void);
void (*USBH_USR_DeviceNotSupported)(void);
void (*UnrecoveredError)(void);
}
USBH_Usr_cb_TypeDef;
```

The above callback functions are described below.

- `Init`: called during initialization by `USBH_Init` core function. In this function, the user can implement any specific intialization related to his application.
- `DeviceAttached`: called when a USB device is attached. It can be useful to inform the user of any device attachment using a display screen.
- `DeviceReset`: called after a USB reset is generated from the host.
- `DeviceDisconnect`: called when a device is disconnected.
- `OverCurrentDetected`: called when an overcurrent is detected on USB VBUS.
- `DeviceSpeedDetected`: called when the device speed is detected (see note 1).
- `DeviceDescAvailable`: called when a device descriptor is available (see note 2).
- `DeviceAddressAssigned`: called when the device address is assigned.
- `ConfigurationDescAvailable`: called when configuration, interface and endpoint descriptors are available (see note 3).
- `ManufacturingString`: called when manufacturing string is extracted.
- `ProductString`: called when product string is extractcted.
- `SerialNumString`: called when serial num string is extracted.
- `EnumerationDone`: called when enumeration is finished.
- `UserInput`: called after the end of the enumeration process, for prompting the user for further action, such as pressing a button to start a host class operation (see note 4).
- `USBH_USR_MSC_Application`: called to launch the class application process.
- `USBH_USR_DeviceNotSupported`: called when the detected device is not supported by the current class driver.
- `UnrecoveredError`: called when the core state machine is in "HOST_ERROR_STATE" state. It allows the user to handle any error, by displaying an error message on the LCD screen for example.

Note: Device speed information is returned in the `DeviceSpeed` *parameter. Possible values are: 0x1 for Full speed devices and 0x2 for Low speed devices.*

Device descriptor information is returned in the pointer DeviceDesc, which points to a structure of type `USBH_DevDesc_TypeDef` *defined as follows:*

```
typedef struct _DeviceDescriptor
{
```

```
uint8_t bLength;
uint8_t bDescriptorType;
uint16_t bcdUSB; /* USB Specification Number which device complies too */
uint8_t bDeviceClass;
uint8_t bDeviceSubClass;
uint8_t bDeviceProtocol;
uint8_t bMaxPacketSize;
uint16_t idVendor; /* Vendor ID (Assigned by USB Org) */
uint16_t idProduct; /* Product ID (Assigned by Manufacturer) */
uint16_t bcdDevice; /* Device Release Number */
uint8_t iManufacturer; /* Index of Manufacturer String Descriptor */
uint8_t iProduct; /* Index of Product String Descriptor */
uint8_t iSerialNumber; /* Index of Serial Number String Descriptor */
uint8_t bNumConfigurations; /* Number of Possible Configurations */
}
USBH_DevDesc_TypeDef
```

Device configuration information (config, interface and endpoint descriptors) are returned with pointers on structures `USBH_CfgDesc_TypeDef`, `USBH_InterfaceDesc_TypeDef` *and* `USBH_EpDesc_TypeDef` *defined as follows:*

```
typedef struct _ConfigurationDescriptor
{
uint8_t bLength;
uint8_t bDescriptorType;
uint16_t wTotalLength;
uint8_t bNumInterfaces;
uint8_t bConfigurationValue;
uint8_t iConfiguration;
uint8_t bmAttributes;
uint8_t bMaxPower;
}
USBH_CfgDesc_TypeDef;

typedef struct _InterfaceDescriptor
{
uint8_t bLength;
uint8_t bDescriptorType;
uint8_t bInterfaceNumber;
uint8_t bAlternateSetting; /* Value used to select alternative setting */
uint8_t bNumEndpoints; /* Number of Endpoints used for this interface */
uint8_t bInterfaceClass; /* Class Code (Assigned by USB Org) */
uint8_t bInterfaceSubClass; /* Subclass Code (Assigned by USB Org) */
uint8_t bInterfaceProtocol; /* Protocol Code */
uint8_t iInterface; /* Index of String Descriptor Describing this interface
*/
}
```

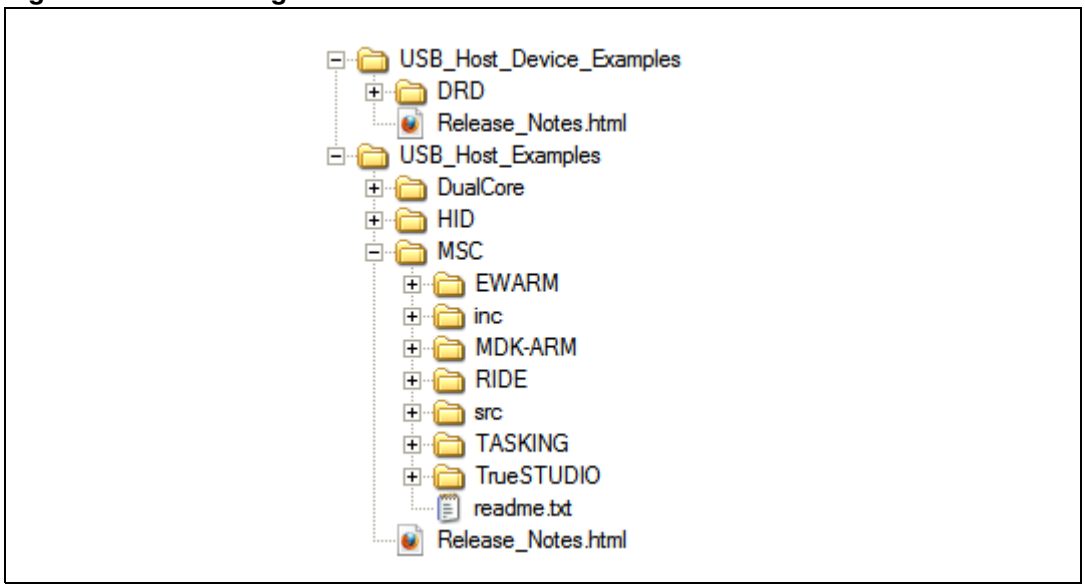
```
USBH_InterfaceDesc_TypeDef;

typedef struct _EndpointDescriptor
{
uint8_t bLength;
uint8_t bDescriptorType;
uint8_t bEndpointAddress; /* indicates what endpoint this descriptor is
describing */
uint8_t bmAttributes; /* specifies the transfer type. */
uint16_t wMaxPacketSize; /* Maximum Packet Size this endpoint is capable of
sending or receiving */
uint8_t bInterval; /* is used to specify the polling interval of certain
transfers. */
}
USBH_EpDesc_TypeDef;
```

In order to move the core state machine to `HOST_CLASS_REQUEST` state, the `UserInput` callback should return the value `USBH_USR_RESP_OK` of type `USBH_USR_Status`.

```
typedef enum {
USBH_USR_NO_RESP = 0, /*no response from user */
USBH_USR_RESP_OK = 1,
}
USBH_USR_Status;
```

7.8 Application layer description

Figure 37. Folder organization



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

  - *stm32fxxx_it.c*: contains the system interrupt handlers
  - *system_stm32fxxx.c*: system clock configuration file for STM32Fxxx devices
  - *usb_bsp.c*: contains the function implementation (declared in the *usb_bsp.h* file in the USB OTG Low Level Driver) to initialize the GPIO for the core, time delay methods and interrupts enabling/disabling process.
  - *usbh_usr.c*: contains the function implementation (declared in the *usbh_usr.h* file in the USB Library) to handle the library events from user layer (event messages).
- The "*includes*" directory contains the following files:
  - *stm32fxxx_it.h*: header file *stm32fxxx_it.c* file
  - *usb_conf.h*: configuration files for the USB OTG low level driver.
  - *usbh_conf.h*: configuration files for the USB Host library.

Note: *For HID demonstration the* usbh_usr_lcd.c *file is used to draw the mouse graphical.*

For Dual core demonstration, additional files are used:
- dual_core_demo.c/.h*: implementation of the demonstration.*
- usbh_msc_usr .c/.h*: contains the user callbacks for mass storage layer.*
- usbh_hid_usr.c/.h*: contains the user callbacks for HID layer.*

When using the USB OTG Full speed core, the user should use the CN8 connector on the STM322xG-EVAL and STM324xG-EVAL or the CN2 connector when the STM3210C-EVAL is used.

When using the USB OTG High speed core, the user should use the CN9 connector on the STM322xG-EVAL and STM324xG-EVAL boards.

7.9 Starting the USB host library

Since the USB Library can handle multi core instances, the user has to define beforehand the core device handle and the host structure pointer in the main file.

```
58
59 /** @defgroup USBH_USR_MAIN_Private_Variables
60 * @{
61 */
62 USB_OTG_CORE_HANDLE          USB_OTG_Core_dev;
63 USBH_HOST                    USB_Host;
64 /**
65 * @}
66 */
67
```

The USB Library is built in interrupt model; from application layer, the user has only to call the `USBH_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.

```
176 int main (void)
177 {
178
179   /* Init HS Core */
180   USBH_Init(&USB_OTG_Core,
181             USB_OTG_HS_CORE_ID,
182             &USB_Host,
183             &USBH_MSC_cb,
184             &USR_MSC_cb);
185    LCD_UsrLog("USB Host High speed initialized.\n");
186   /* Init FS Core */
187   USBH_Init(&USB_OTG_FS_Core,
188             USB_OTG_FS_CORE_ID,
189             &USB_FS_Host,
190             &HID_cb,
191             &USR_HID_cb);
192
193
194   LCD_UsrLog("USB Host Full speed initialized.\n");
195
196 }
197
```

7.10 USB host examples

Each project for an example based on a class is given with five configurations, as follows (exception made for USB Host dual core example).

1. STM322xG-EVAL_USBD-HS: High-speed example on the STM322xG-EVAL board working with USB OTG HS core and the ULPI PHY
2. STM322xG-EVAL_USBD-FS: Full-speed example on the STM322xG-EVAL board working with USB OTG FS core and the embedded FS PHY
3. STM324xG-EVAL_USBD-HS: High-speed example on the STM324xG-EVAL board working with USB OTG HS core and the ULPI PHY
4. STM324xG-EVAL_USBD-FS: Full-speed example on the STM324xG-EVAL board working with USB OTG FS core and the embedded FS PHY
5. STM3210C-EVAL_USBD-FS: Full-speed example on the STM3210C-EVAL board working with USB OTG FS core and the embedded FS PHY.

For the High speed examples, the following features are selected in the *usb_config.h* file:

- `USB_OTG_HS_ULPI_PHY_ENABLED`: ULPI Phy is used
- `USB_OTG_HS_INTERNAL_DMA_ENABLED`: internal DMA is used

Important notes:

Note: The USB Host examples are using the lcd_log.c module to redirect the Library and User messages on the screen. Depending on the LCD cache depth used to scroll forward and backward within the message, the applications footprints are impacted. With bigger LCD cache depth, the RAM footprint is consequently increased. To prevent this additional RAM

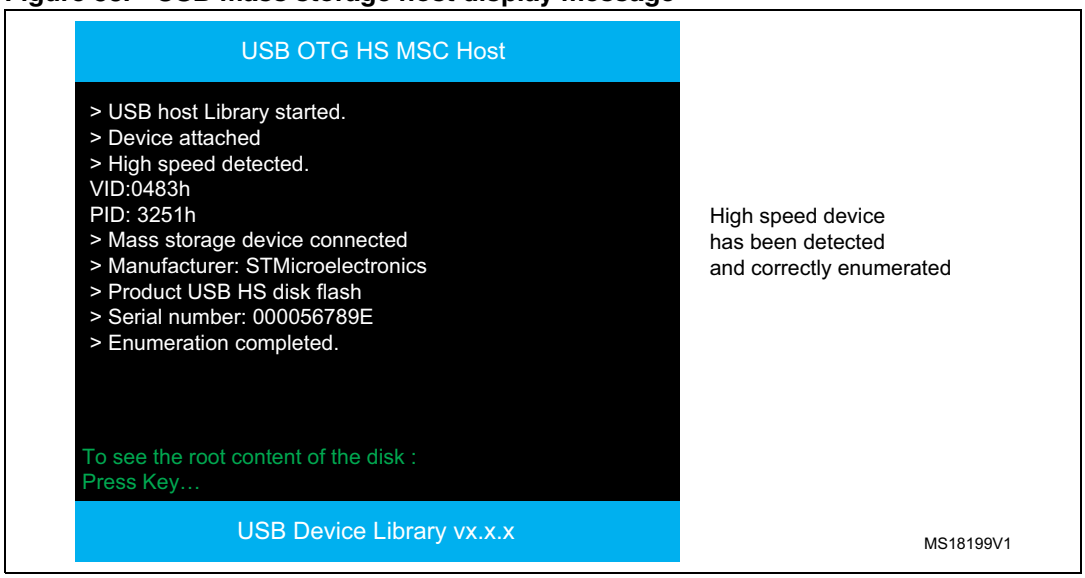
footprint, the user can redirect the Library and User messages on other terminal (HyperTerminal or LCD using the native display functions).

Library and User messages are located in the user callbacks in the application layer. They are not mandatory and they are used for information and debug purpose only. They can be modified or even removed.

7.10.1 USB mass storage host example

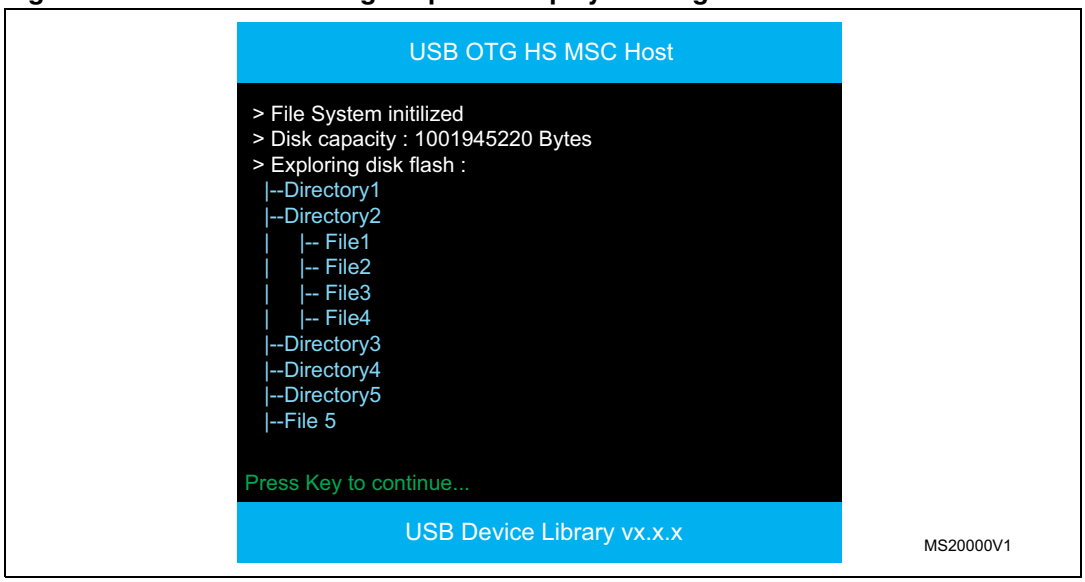
When attaching a mass storage device to the STM322xG-EVAL, STM324xG-EVAL or STM3210C-EVAL board, the LCD displays the following text (for example, when plugging in the Kingston Data Traveler G2 USB Flash drive).

Figure 38. USB mass storage host display message



When the user press the user key [B4], the application explore the USB flash disk content and the LCD displays the following messages:

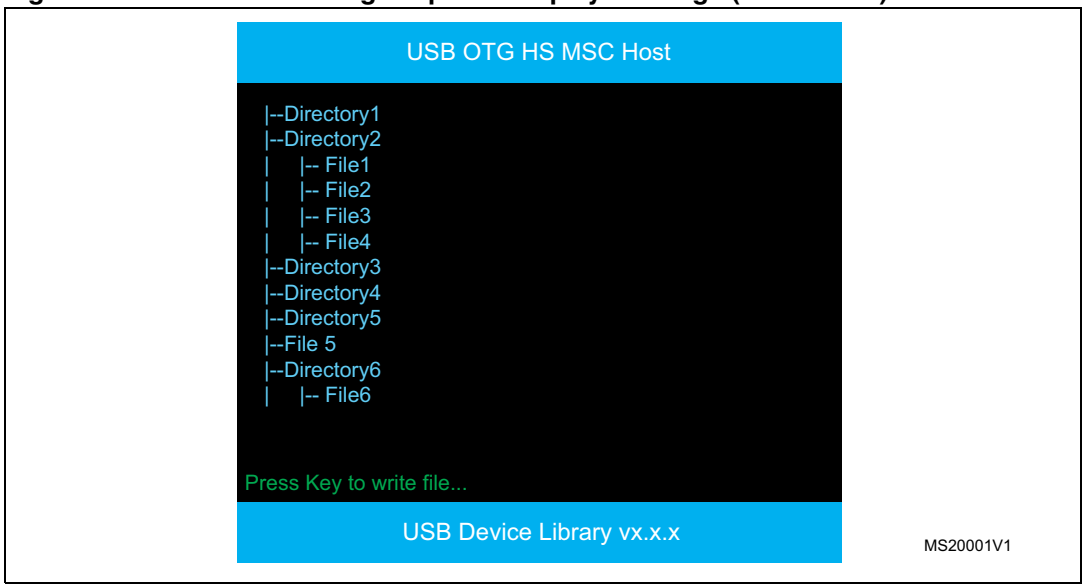
Figure 39. USB mass storage explorer display message



The user has to press the user key [B4] to display the whole disk flash (recursion level 2).

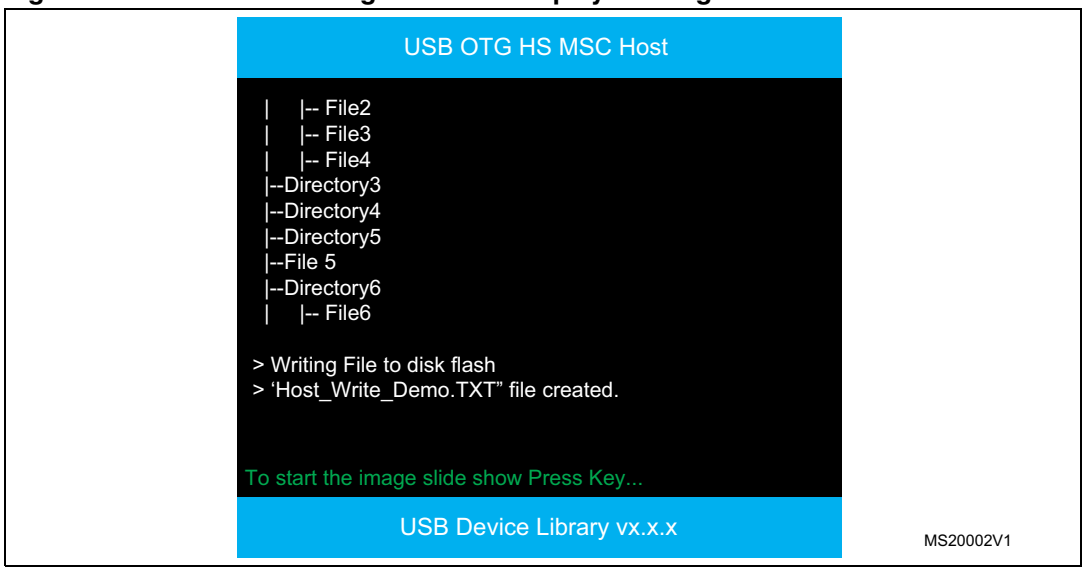
Once the entire disk flash is shown:

Figure 40. USB mass storage explorer display message (last screen)



The user has to press the user key [B4] to write a small file (less to 1 KB) on the disk.

Figure 41. USB mass storage write file display message



After writing the file to the disk, the user can press the user key [B4] and start the Image slide show (BMP file located in the USB Disk root).

Figure 42. USB mass storage slideshow example

The image advancement is done automatically each one second. Once all the images are displayed, the application explores again the disk flash (*Figure 39*).

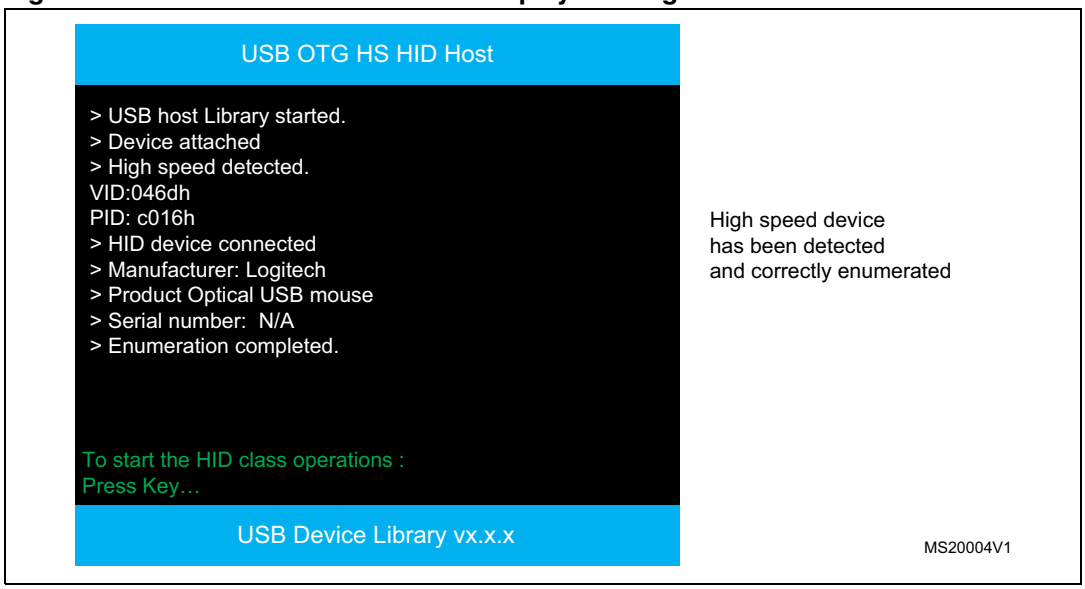
Note: The application header (title) depends on the board in use, once the full speed port is used, the title is: "USB OTG FS MSC Host".

Only the BMP files with the following format are supported: Width = 320, Height = 240, BPP = 16 and Compression = RGB bitmap with RGB masks.

7.10.2 USB HID Host example

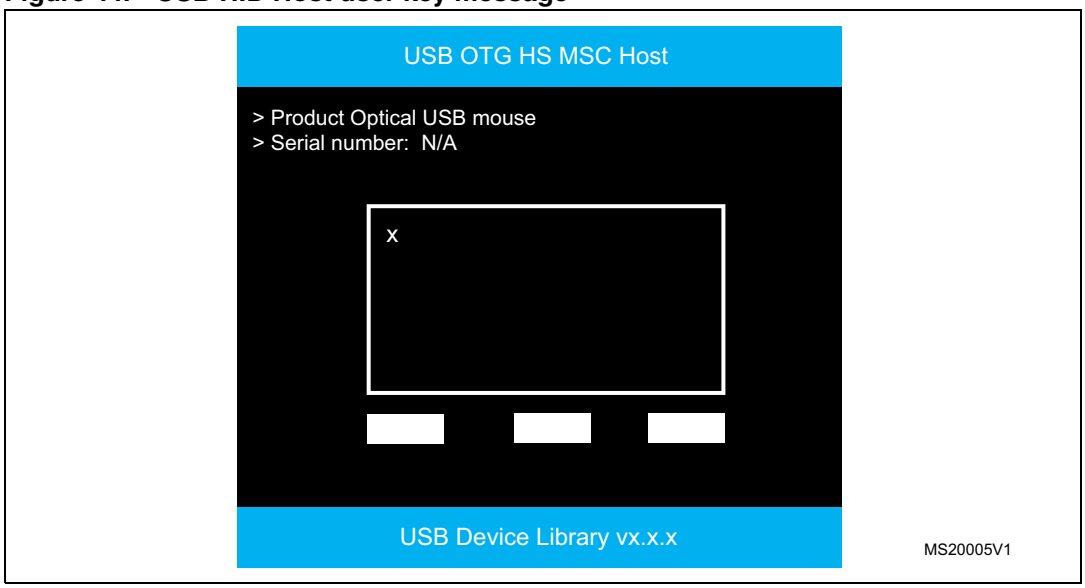
When attaching a HID device to the STM322xG-EVAL, STM324xG-EVAL or STM3210C-EVAL board, the LCD displays the following text (for example, when plugging Logitech USB mouse or a keyboard).

Figure 43. USB HID Host connected display message



When the user press the user key [B4], the application displays the mouse pointer and buttons.

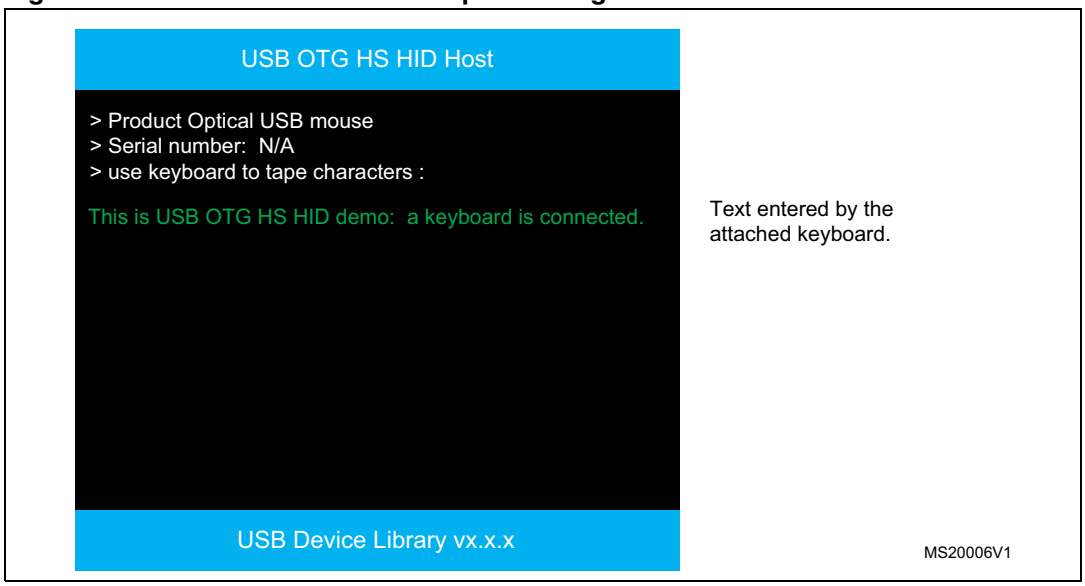
Figure 44. USB HID Host user key message



Moving the mouse will move the pointer in the display rectangle and if a button is pressed, the corresponding rectangle will be highlighted in green

If a keyboard has been attached, the display show the following messages and the taped characters are displayed in green on the display.

Figure 45. USB HID Host text example message

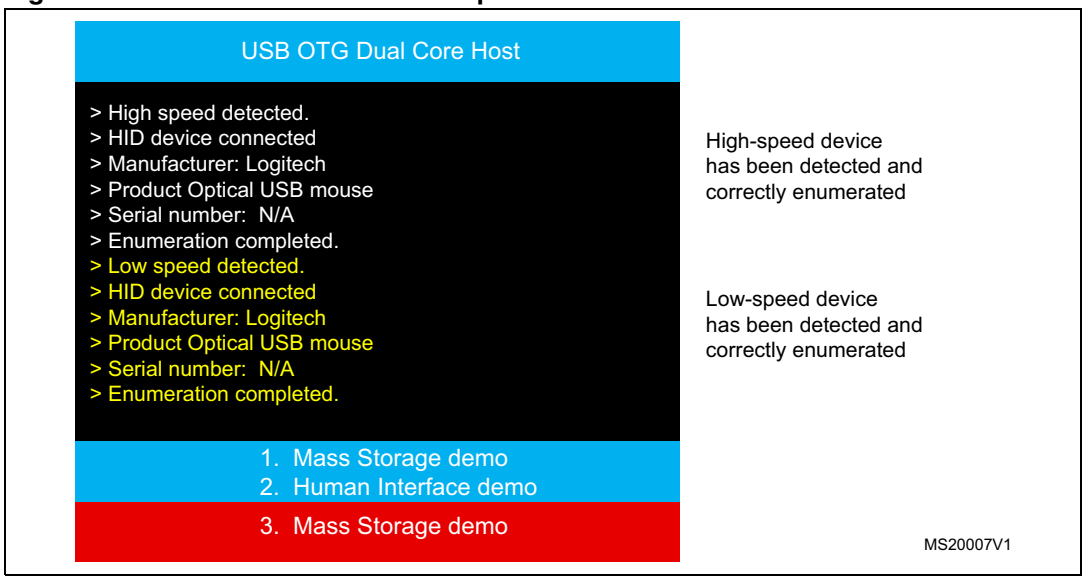


Note: The application header (title) depends on the board in use, once the full speed port is used, the title is: "USB OTG FS MSC Host"

7.10.3 USB dual core host example

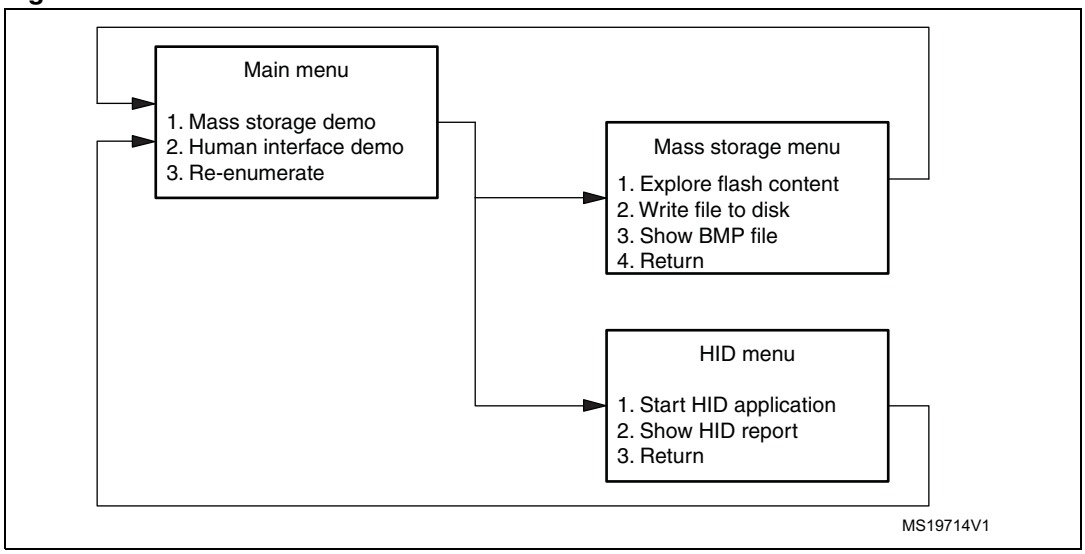
In this demonstration, the user can use one or two devices, the mass storage device should be connected to the high speed port while the HID device should be connected to the full speed port.

Figure 46. USB dual core host example



To move within the menu, the user has to use the embedded joystick, the menu structure is as follows.

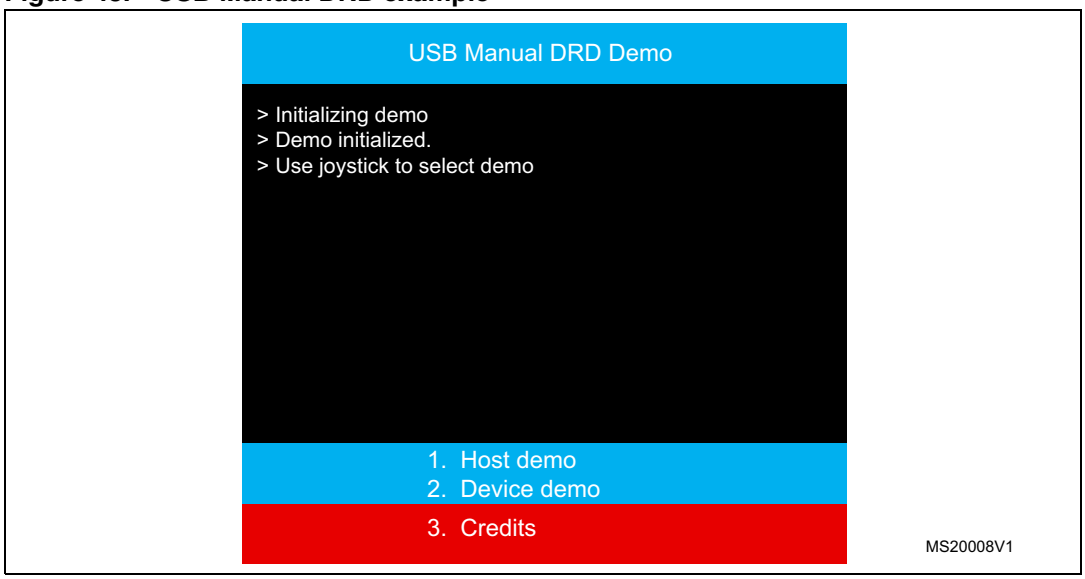
Figure 47. Menu structure



7.10.4 USB manual dual role device example

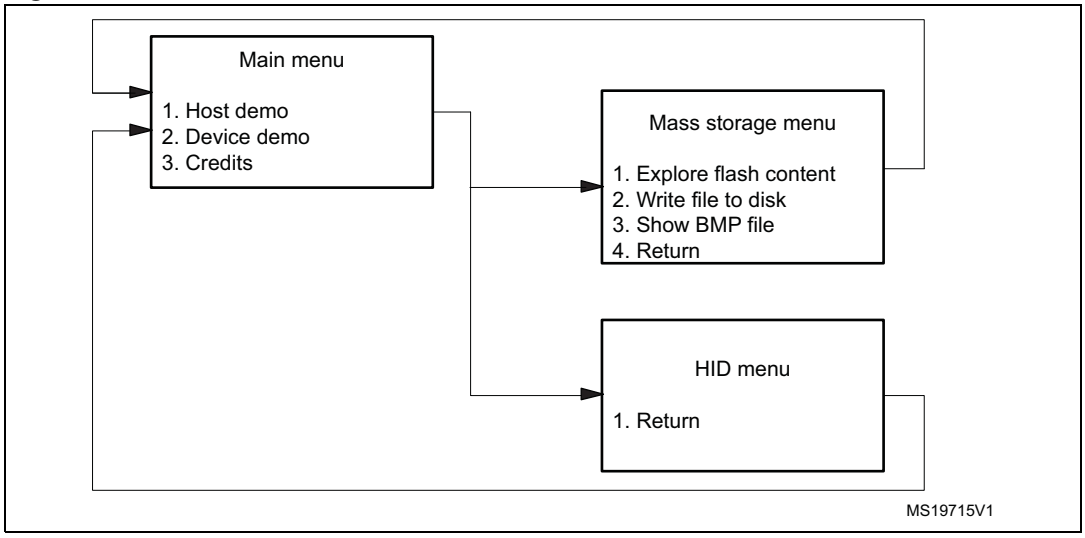
In this demonstration, the user can use Host or Device mode on the same core by selecting through the menu the sub-demo to run. In both Device and Host modes, the mass storage class is used.

Figure 48. USB Manual DRD example



To move within the menu, the user has to use the embedded joystick, the menu structure is as follows.

Figure 49. Menu structure



8 Frequently-asked questions

1. **How can the USB Device Library be configured to run in either High Speed or Full Speed mode?**

   The Library can handle the USB OTG HS and USB OTG FS core, if the USB OTG FS core can only work in Full Speed mode, the USB OTG HS can work in High or Full Speed mode. For that the user has to:

   a) Select the core to be used during the library initalization, issuing one of the two core IDs:

   – `USB_OTG_HS_CORE_ID`

   – `USB_OTG_FS_CORE_ID`

   Example:

   ```
   USBD_Init(&USB_OTG_dev,
   #ifdef USE_USB_OTG_HS
        USB_OTG_HS_CORE_ID,
   #else
        USB_OTG_FS_CORE_ID,
   #endif
        …);
   ```

   b) Select the USB OTG Core, HS or FS, in usb_conf.h file:

   – USE_USB_OTG_HS (*): if the USB OTG HS Core is to be used

   – USE_USB_OTG_FS (*): if the USB OTG FS Core is to be used

   c) Select the PHY to be used, in usb_conf.h file:

   For the USB OTG HS Core, you can select the PHY to be used using one of these two defines:

   – USE_ULPI_PHY (*): if the USB OTG HS Core is to be used in High speed mode

   – USE_EMBEDDED_PHY (*): if the USB OTG HS Core is to be used in Full speed mode

   For USB OTG FS Core, the on-chip Full Speed PHY is used (no need for any configuration).

Note: (*) To avoid modifying these defines each time you need to change the USB configuration, you can declare the needed define in your toolchain compiler preprocessor.

The USE_ULPI_PHY symbol is defined in the project compiler preprocessor as default PHY when the HS core is used.

On STM322xG-EVAL and STM324xG-EVAL boards and for the HS core, only the external ULPI High Speed PHY is available. On-chip Full Speed PHY need a different hardware. For more details refer to your STM32 device datasheet.

2. **How can the used endpoints be changed in the USB Device class driver?**

   To change the endpoints or to add a new endpoint:

   a) Perform the endpoint initialization using `DCD_EP_Open()`.

   b) Configure the TX or the Rx FIFO size of the new defined endpoints in the *usb_conf.h* file.

Note: The total size of the Rx and Tx FIFOs should be lower than the Total FIFO size of the used core (320 x 32 bits for USB OTG FS core and 1024 x 32 bits for the USB OTG HS core).

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

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

4. 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 */

};
```

5. **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.

6. **How can the keyboard layout be changed in the USBH HID class?**

   In the USB Host HID class, two layouts are defined in the *usbh_hid_keybd.h* file and can be used (Azerty and Querty). Uncomment the required keyboard layout:

   ```
   //#define QWERTY_KEYBOARD
   #define AZERTY_KEYBOARD
   ```

   The User can eventually add his own layout by editing the `HID_KEYBRD_Key` array in the *usbh_hid_keybd.c* file.

9 Troubleshooting

1. When resetting USB device applications (HS mode) using the Reset button on the STM322xG-Eval board RevB, the device enters Suspend mode and cannot leave this state until the power cable is removed.

   This issue is due to the fact that the ULPI Phy reset pin is always connected in high state by a hardware pull-up. In order to use the Reset button to reset the ULPI Phy interface, the reset pin of the PHY must be connected by hardware to the Reset button.

2. After removing the USB cable on USB HID devices in Full Speed mode, the core seems to enter Stop mode and the LEDs stop blinking. Is this normal behavior?

   Yes, this is normal behavior. When a suspend state is detected over the USB data bus, the core enters Low Power mode and only a wakeup event or a new connection to the host can wake up the device.

   This behavior can be changed by disabling Low Power mode in the *usb_conf.h* file by commenting the `#define USB_OTG_FS_LOW_PWR_MGMT_SUPPORT` define

3. Even if the `#define USB_OTG_HS_LOW_PWR_MGMT_SUPPORT` is uncommented, the USB HID Device enters Low Power mode but cannot wakeup after a new Host connection.

   This behavior is normal since the wakeup signal in the USB OTG HS Core is available only when the USB OTG HS core is working in Full Speed mode.

10 Revision history

Table 42. Document revision history

Date	Revision	Changes
26-Nov-2010	1	Initial release.
08-Aug-2011	2	Major document revision to include both the USB On-The-Go host and device libraries for STM32F105/7 and STM32F2xx devices.
08-Mar-2012	3	Added STM32F4xx devices. Removed Appendix A. Changed format of the document (for codes).

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.

UNLESS EXPRESSLY APPROVED IN WRITING BY TWO AUTHORIZED ST REPRESENTATIVES, ST PRODUCTS ARE NOT RECOMMENDED, AUTHORIZED OR WARRANTED FOR USE IN MILITARY, AIR CRAFT, SPACE, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS, NOR IN PRODUCTS OR SYSTEMS WHERE FAILURE OR MALFUNCTION MAY RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE. ST PRODUCTS WHICH ARE NOT SPECIFIED AS "AUTOMOTIVE GRADE" MAY ONLY BE USED IN AUTOMOTIVE APPLICATIONS AT USER’S OWN RISK.

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