UM1720
User manual

STM32Cube USB host library

Introduction

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

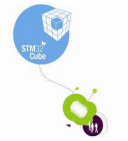
STM32Cube Version 1.x includes:

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

The Universal Serial Bus (USB) is known to be the defacto standard for communication between Personal Computer (PC) as host and connected USB peripherals. Today with the increasing number of embedded portable devices, USB as a host is no more restricted to PC but it is becoming more present in embedded consumer and industrial devices as embedded host with limited functionality like support of a particular USB class (for example Mass-storage class, virtual comport) or particular vendor device.

This user manual describes the STM32Cube USB Host library which is part of the STM32Cube firmware package that can be downloaded free from ST website (*http://www.st.com/stm32cube*). It allows using an STM32 microcontroller as an USB embedded host for communication with USB peripherals of various USB classes (MSC, HID, CDC, Audio and MTP).

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



Contents

List of tables

List of figures

1 STM32Cube USB Host library overview

This document describes the STM32Cube USB host library middleware module.

The USB host library sits on top of the STM32Cube USB host HAL driver. This library offers the APIs used to access USB devices of various classes.

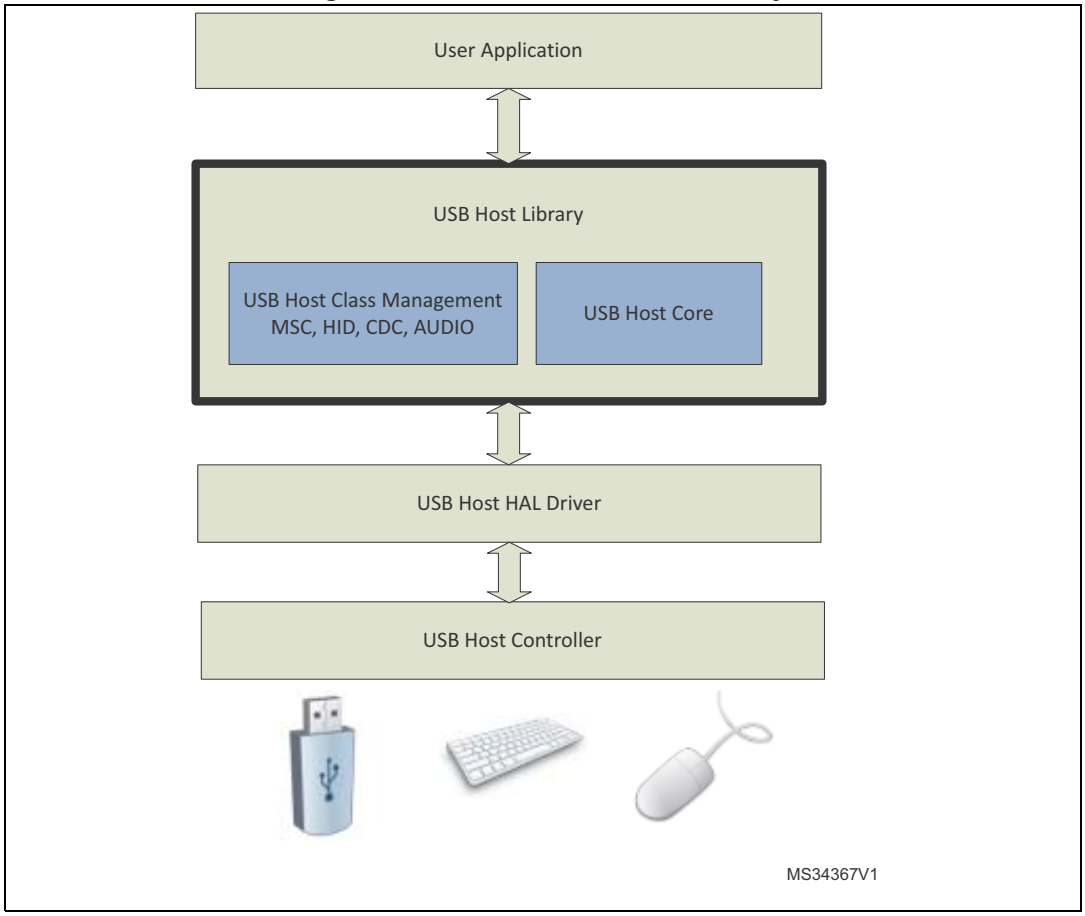
The USB host module can be used for the implementation of the major USB classes:

- Mass-storage class (MSC)
- Human interface mouse and keyboard class(HID)
- Communication device class (CDC)
- Audio class (AUDIO)
- Media Transfer protocol class (MTP)

In addition to the above-listed classes, the users can build their own class using the available library APIs.

The library is built with the possibility to work in standalone mode or in RTOS mode. The library also supports multi-instance, as it can work simultaneously on two or more USB host peripherals.

Figure 1. STM32Cube USB host library

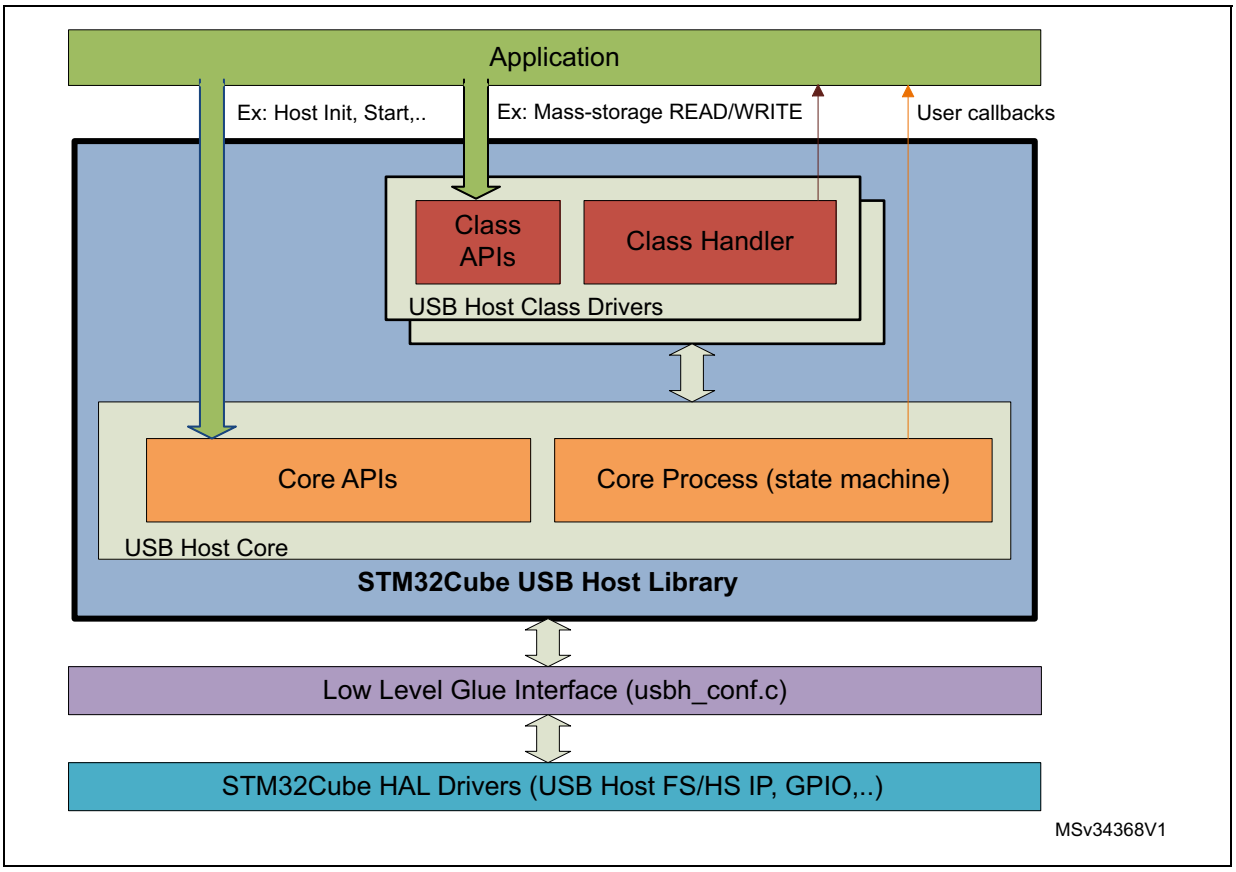


2 USB host library architecture and file organization

2.1 USB host library architecture

Figure 2 shows an overview of the library architecture.

Figure 2. USB host library architecture



As seen in *Figure 2*, the USB host library is organized into two main parts: the core and the class drivers.

The host core handles the core services of the library which are mainly used for the following:

- Device connection/disconnection management and standard enumeration,
- Host pipes control and USB transfer management (control, interrupt, bulk, isochronous).

The core block can be schematically subdivided into two main sub-blocks:

- The core APIs which are called from the user application or from a class driver,
- The core process which handles the host state machine.

The class drivers are used for the implementation of the various USB classes.

A class driver can be seen as the following:

- A set of class specific APIs (for example disk_read, disk_write) which can be called from an application layer.
- A class handler which is called from the core host state machine (Core Process) to handle the class operation (class initialization, class de-initialization, class process).

Both the host core and the class drivers call user callback functions following some defined events (for example device connection/disconnection, data reception).

The USB host library interfaces with the STM32Cube HAL drivers using an interface layer.

A template file implementing the low-level interface layer is provided in the library. The user can customize the sections in the file that are related to board support package requirements.

2.2 USB host library file organization

2.2.1 USB host core files

The USB host core files are located in the *Core* folder (*STM32_USB_Host_Library\Core*). *Table 1* lists the core files.

Table 1. USB host core files

File	Description
usbh_core.c/.h	Main host core file. Implements the host state machine. Manages the device detection and enumeration and handles control to class module for class operation.
usbh_ctlreq.c/.h	Implements the standard control requests (USB chapter9)
usbh_ioreq.c/.h	APIs for USB transfer management (control, bulk, interrupt, isochronous)
usbh_pipes.c/.h	APIs for pipe control (for example allocate, open, close)
usbh_conf_template.c/.h	Template file for the low layer interface file, should be customized by user and included with application file (see details in section 5.x)
usbh_def.h	Common library defines

2.2.2 USB host class files

Table 2 lists the class files, they are located in the *Class* folder (*STM32_USB_Host_Library\Core*)

Table 2. Class drivers files

USB class	File	Description
Mass-Storage	usbh_msc.c	Mass-storage class handler
	usbh_msc_bot.c	Bulk-Only Transfer (BOT) protocol
	usbh_msc_scsi.c	SCSI commands
HID mouse & keyboard	usbh_hid.c	HID class state handler
	usbh_hid_parser.c	HID descriptor parser
	usbh_hid_mouse.c	HID mouse subclass handler
	usbh_hid_keybd.c	HID Keyboard subclass handler
	usbh_hid_usage.h	Common defines for HID
Audio speaker	usbh_audio.c	Audio class handler
CDC virtual com port	usbh_cdc.c	CDC virtual comport handler
MTP class	usbh_mtp.c	MTP class handler
	usbh_mtp_ptp	MTP class PTP spec implementation

3 Host library core module

3.1 Core features

The USB host core has the following main features:

- Device attachment management and enumeration independently from the device class
- State machine based, runs in the main loop in background or as an RTOS task thread
- Use of user event callbacks to inform application layer about host events such as device connection/disconnection, error state
- User event log that can be redirected to any interface (for example serial port, LCD)
- Error management and reporting

3.2 Core APIs, user callbacks and data structures

As detailed in *Section 2.1: USB host library architecture*, the core APIs can be used by both the application and the class drivers.

The following sections present each core API. For more details please refer to function header in the C code.

The main APIs usage is further detailed in *Chapter 5*.

3.2.1 Core APIs for application

The user application APIs are listed in *Table 3*.

Table 3. Core APIs for application

Function	Description
USBH_Init	Initializes the host stack and the low level. Should be called at application startup
USBH_DeInit	Re-initializes the host stack variables and runs a low level clean-up (for example closes all open pipes, clears interrupt flags)
USBH_RegisterClass	Registers a supported USB class handler. After enumeration the host checks if the current device class corresponds to the one of the registered classes
USB_ReEnumerate	Initiates a device re-enumeration by re-initializing the host stack and by forcing a device VBUS disconnection/connection
USBH_Start	Enables the host port VBUS power and starts the low-level operation
USBH_Stop	Disables the host port VBUS power and stops the low-level operation
USBH_GetActiveClass	Returns the current active USB class following the device enumeration and class level initialization
USBH_Process	Host process function that implements the core state machine in standalone mode operation. This function should be called in background in the main loop to handle the host state machine.

3.2.2 Core user callbacks

The host core communicates the USB events to the application layer by calling a user callback function. This function is passed as parameter when calling the *USBH_Init* API.

The user callback function should be of the following prototype:

void (*pUsrFunc)(USBH_HandleTypeDef * phost, uint8_t event)

The core user callbacks events are set as *event* parameter. *Table 4* describes the core user callback events.

Table 4. Core user callbacks events

Core user callback event	Description
HOST_USER_CONNECT	Informs the application about a device connection
HOST_USER_DISCONNECT	Informs the application about a device disconnection
HOST_USER_CLASS_ACTIVE	Informs the application about the end of class initialization process
HOST_USER_SET_CONFIGURATION	Informs the application about end of the device standard enumeration
HOST_USER_CLASS_SELECTED	Informs than a supported class is found

3.2.3 Core APIs for class handlers

The core APIs dedicated to class handlers are classified as follows:

- I/O requests APIs
- Pipe control APIs
- Standard class control requests APIs
- Interface utility APIs

Table 5 lists the core APIs dedicated to class handlers.

Table 5. Core APIs dedicated to class handlers

Function	Category	Description
USBH_BulkReceiveData	IO requests	Receives data on bulk pipe
USBH_BulkSendData		Sends data on bulk pipe
USBH_CtlReceiveData		Receives data on control pipe
USBH_CtlSendData		Sends data on control endpoint
USBH_CtlSendSetup		Issues a control request
USBH_InterruptReceiveData		Receives data from the interrupt pipe
USBH_InterruptSendData		Sends data to the interrupt pipe
USBH_IsocReceiveData		Receives data from the isochronous pipe
USBH_IsocSendData		Sends data to the isochronous pipe
USBH_OpenPipe	Pipe control	Opens a pipe
USBH_ClosePipe		Closes a pipe
USBH_AllocPipe		Allocates a new pipe
USBH_FreePipe		Sets free an allocated pipe
USBH_GetDescriptor	Standard control requests	Generic function to get a descriptor
USBH_SetInterface		Standard control request to set alternate setting value for an interface
USBH_FindInterface	Interface utility	Parses a configuration descriptor to find an interface descriptor corresponding to a specific class, subclass and protocol
USBH_FindInterfaceIndex		Parses a configuration descriptor to find the index of an interface descriptor with a particular interface number and alternate setting value

3.2.4 Main host core data structures and enumerated typedefs

The data structures and enumeration types of the core are defined in the file *usbh_def.h*. The main data structures are the following:

- the core handle structure of the type *USBH_HandleTypedef,*
- the device handle structure of the type *USBH_DeviceTypedef,*
- the class handle structure of the type *USBH_ClassTypedef*.

The core handle and the device handle structure are detailed in the following sub sections. The class handle structure will be detailed in the class section of *Chapter 4*.

Host core handle structure

The main structure used in the host library is the host handle which is of the type *USBH_HandleTypedef*.

Figure 3. USBH_HandleTypedef

```
typedef struct _USBH_HandleTypeDef
{
  HOST_StateTypeDef     gState;
  ENUM_StateTypeDef     EnumState;
  CMD_StateTypeDef      RequestState;
  USBH_CtrlTypeDef      Control;
  USBH_DeviceTypeDef    device;
  USBH_ClassTypeDef*    pClass[USBH_MAX_NUM_SUPPORTED_CLASS];
  USBH_ClassTypeDef*    pActiveClass;
  uint32_t              ClassNumber;
  uint32_t              Pipes[15];
  __IO uint32_t         Timer;
  void*                 pData;
  void                 (* pUser )(struct _USBH_HandleTypeDef *pHandle,
uint8_t id);

#if (USBH_USE_OS == 1)
  osMessageQId          os_event;
  osMessageQId          class_ready_event;
  osThreadId            thread;
#endif
} USBH_HandleTypeDef;
```

Table 6 details the structure of the host handler.

Table 6. Host handle structure

Structure member	Description
gState	Gives the current state in the global host state machine
EnumState	Gives the current state in the enumeration state machine
RequestState	Gives the current state for a control request (IDLE, SEND or WAIT)
Control	Structure maintaining the information related to the control transfer management
Device	Structure maintaining the informations about the connected device

Table 6. Host handle structure (continued)

Structure member	Description
pClass[USBH_MAX_NUM_SUPPORTED_CLASS]	Array assigned with pointers to the registered class handlers structures
pActiveClass	Points to the active class handler structure
ClassNumber	Gives the total count of registered classes
Pipes[15]	Maintains the status for each host pipe (allocated or free). Indicates also the associated endpoint number if any
Timer	Counting variable for time management. Automatically incremented at each start of frame
pData	Pointer initialized with low level host controller data structure (in the *usbh_conf.c configuration* file), allows the interfacing of the library with the low-level driver
(* pUser)(struct _USBH_HandleTypeDef *pHandle, uint8_t id);	Host user event callback function

Note: The members under the define USBH_USE_OS shown in Figure 3 are related to operation with RTOS. They maintain the internal information regarding RTOS message queue events.

Host core device structure

The host core device structure maintains the informations regarding the connected device. This structure is of the type *USBH_DeviceTypeDef* which is declared as shown in *Figure 4*.

Figure 4. Device descriptor

```
typedef struct {
#if (USBH_KEEP_CFG_DESCRIPTOR == 1)
  uint8_t
CfgDesc_Raw[USBH_MAX_SIZE_CONFIGURATION];
#endif
  uint8_t                          Data[USBH_MAX_DATA_BUFFER];
  uint8_t                          address;
  uint8_t                          speed;
  uint8_t                          is_connected;
  uint8_t                          current_interface;
  USBH_DevDescTypeDef              DevDesc;
  USBH_CfgDescTypeDef              CfgDesc;
}USBH_DeviceTypeDef;
```

Table 7 lists the members of the host device structure:

Table 7. Host device structure

Structure member	Description
CfgDesc_Raw	Full device configuration descriptor
Data	Data buffer allocated to receive the descriptors
Address	Device address
Speed	Device speed (low, full, high speed)
is_connected	Device connection status
current_interface	Current selected interface
DevDesc	Structure with device descriptor data
CfgDesc	Structure with configuration descriptor data (only the first 9 bytes of the configuration descriptor)

Main enumerated typedefs

USBH_StatusTypeDef

Almost all library functions return a status of the type *USBH_StatusTypeDef*. The application should always check the returned status.

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

Table 8 describes the above-mentioned returned status.

Table 8. USB host status

Status	Description
USBH_OK	Returned when the operation is completed successfully
USBH_BUSY	Retuned when the operation is still not completed (busy)
USBH_FAIL	Returned when the operation has failed due to a low-level error or protocol fail
USBH_NOT_SUPPORTED	Returned when the requested operation/feature is not supported
USBH_UNRECOVERED_ERROR	Returned when the host process cannot self recover from a error
USBH_ERROR_SPEED_UNKNOWN	Returned after a device attachment to inform the application that the device speed cannot be detected

The following enumerated typedefs are used internally by the USB core. However, the user can access these through the host handle structure for debug purpose.

HOST_StateTypeDef

This enumerated typedef lists the possible host core states (see host state machine in *Section 3.3*).

```
typedef enum
{
  HOST_IDLE =0,
  HOST_DEV_ATTACHED,
  HOST_DEV_DISCONNECTED,
  HOST_DETECT_DEVICE_SPEED,
  HOST_ENUMERATION,
  HOST_CLASS_REQUEST,
  HOST_INPUT,
  HOST_SET_CONFIGURATION,
  HOST_CHECK_CLASS,
  HOST_CLASS,
  HOST_SUSPENDED,
  HOST_ABORT_STATE,
}HOST_StateTypeDef;
```

ENUM_StateTypeDef

This enumerated typedef lists the states during device enumeration process.

```
typedef enum
{
  ENUM_IDLE = 0,
  ENUM_GET_FULL_DEV_DESC, /* get device descriptor */
  ENUM_SET_ADDR,          /* set device address */
  ENUM_GET_CFG_DESC,      /* get configuration descriptor (first 9 bytes)*/
  ENUM_GET_FULL_CFG_DESC, /* get full configuration descriptor */
  ENUM_GET_MFC_STRING_DESC, /* get manufacturer string descriptor */
  ENUM_GET_PRODUCT_STRING_DESC, /* get product string descriptor */
  ENUM_GET_SERIALNUM_STRING_DESC, /* get serial number string descriptor */
} ENUM_StateTypeDef;
```

CTRL_StateTypeDef

This enumerated typedef lists the states during a control transfer.

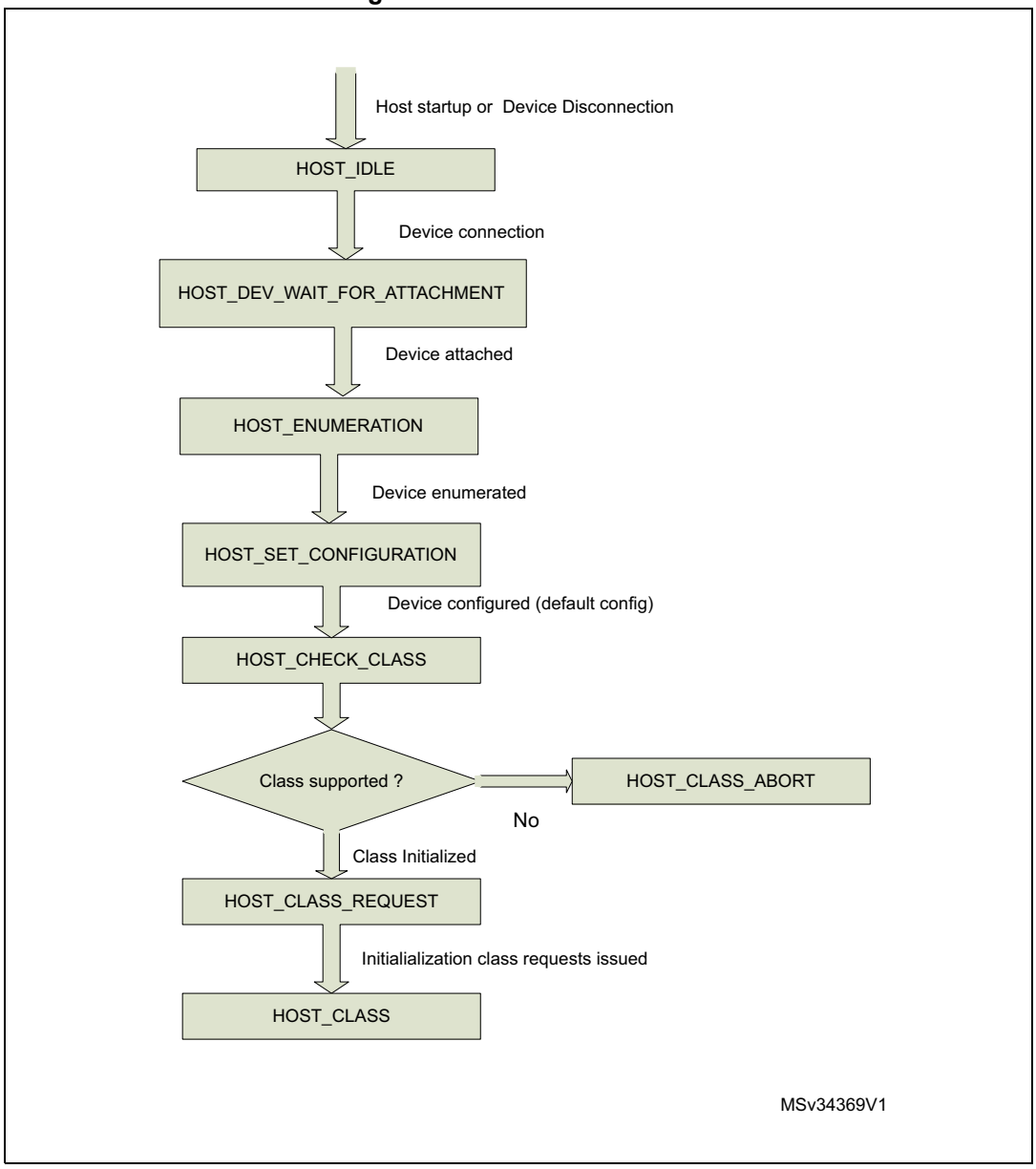
```
typedef enum
{
  CTRL_IDLE =0,
  CTRL_SETUP, /* issue setup packet */
  CTRL_SETUP_WAIT, /* wait setup ACK */
```

```
  CTRL_DATA_IN, /* start a data stage IN */
  CTRL_DATA_IN_WAIT, /* wait data IN stage complete*/
  CTRL_DATA_OUT, /* start a data stage OUT */
  CTRL_DATA_OUT_WAIT, /* wait data OUT stage complete*/
  CTRL_STATUS_IN, /* start status IN stage */
  CTRL_STATUS_IN_WAIT, /* wait status IN stage complete */
  CTRL_STATUS_OUT, /*start status OUT stage */
  CTRL_STATUS_OUT_WAIT, /* wait status OUT stage complete */
  CTRL_ERROR, /* control error */
  CTRL_STALLED, /* request stalled */
  CTRL_COMPLETE    /*full transfer complete */
}CTRL_StateTypeDef;
```

3.3 Overview of the core state machine

Figure 5 presents an overview of the core state machine. The state machine is handled in the core function USBH_Process which can be called in the main polling loop (in standalone mode) or in RTOS task.

Figure 5. Core state machine



The various states are described hereafter.

HOST_IDLE: entered after host start-up or when the device is disconnected

HOST_DEV_WAIT_FOR_ATTACHMENT: entered when the device is connected. While in this state, the host handles the attachment process of the device.

HOST_ENUMERATION: entered when the device attachment is finished. In this state the host runs a standard enumeration of the device.

HOST_SET_CONFIGURATION: in this state, a Set_Config request is issued to select the default configuration.

HOST_CHECK_CLASS: this state checks if one of the registered classes is supported by the enumerated device.

HOST_CLASS_ABORT: this state is entered when the enumerated device class is not supported.

HOST_CLASS_REQUEST: the host enters in this state after initialization of the class interfaces. In this state the host issues the required class control requests.

HOST_CLASS: in this state the host calls the class process function to handle the class background process.

3.4 Core interface with low-level driver

As mentioned in *Section 2.1*, the USB host library interfaces with the STM32Cube HAL low-layer drivers using a low-level interface layer which acts as a glue layer with the STM32Cube HAL.

The low level interface layer implements the following:

- the USB host library low-level interface APIs defined in *Table 9*,
- the USB host BSP functions (clocks, GPIOs, interrupts),
- the USB Host HAL driver callback functions that call the USB host library callbacks defined in *Table 10*.

In the STM32Cube solution the implementation of the low-level interface is provided as part of the USB host examples as some parts of the low-level interface are board- and system-dependent.

Table 9 lists the low level API functions.

Table 9. Low-level interface APIs

API	Description
USBH_LL_Init	Low-level initialization
USBH_LL_DeInit	Low-level de-initialization
USBH_LL_Start	Low-level start
USBH_LL_Stop	Low-level stop
USBH_LL_GetSpeed	Allows to get the detected speed of the connected device
USBH_LL_ResetPort	Issues a USB reset
USBH_LL_GetLastXferSize	Gets the last completed transfer size
USBH_LL_DriverVBUS	Enables or disables VBUS

Table 9. Low-level interface APIs (continued)

API	Description
USBH_LL_OpenPipe	Opens a pipe
USBH_LL_ClosePipe	Closes a pipe
USBH_LL_SubmitURB	Submits a host transfer request
USBH_LL_GetURBState	Gets the state of a pipe
USBH_LL_SetToggle	Sets the initial data toggle for transfer (DATA0 or DATA1)
USBH_LL_GetToggle	Get the data toggle information

Table 10 presents the host library callback functions which are called from the low-level interface following some USB events.

Table 10. Low-level event callback functions

Callback functions	Description
USBH_LL_SetTimer	Should be called by the USB Host HAL driver during USB Host start-up, to initialize the host timer
USBH_LL_IncTimer	Should be called each SOF event, to increment the host timer variable
USBH_LL_SOF	Should be called in HAL SOF event callback to handle the USB class processing which should be synchronized with SOF
USBH_LL_Connect	Should be called on the USB Host HAL event callback for device connection
USBH_LL_Disconnect	Should be called on the USB Host HAL event callback for device disconnection
USBH_LL_NotifyURBChange	When using RTOS mode, this callback function should be called in USB Host HAL event callback for URB state change

4 USB host library class module

4.1 Class implementation model

A class driver is defined as a combination of the following:

- A set of class-specific APIs that can be called from the application layer.
- A set of event callbacks (when applicable).
- A class-handler implemented using a structure of the type *USBH_ClassTypedef*. The structure member functions are invoked by the USB core process.

Figure 6 shows the definition of the *USBH_ClassTypedef*.

Figure 6. Class structure

```
typedef struct
{
  uint8_t             *Name;
  uint8_t              ClassCode;
  USBH_StatusTypeDef  (*Init) (struct _USBH_HandleTypeDef *phost);
  USBH_StatusTypeDef  (*DeInit) (struct _USBH_HandleTypeDef *phost);
  USBH_StatusTypeDef  (*Requests) (struct _USBH_HandleTypeDef *phost);
  USBH_StatusTypeDef  (*BgndProcess) (struct _USBH_HandleTypeDef
*phost);
   USBH_StatusTypeDef  (*SOFProcess)(struct _USBH_HandleTypeDef
*phost);
  void*                pData;
} USBH_ClassTypeDef;
```

Table 11 lists the members of the class-handler structure:

Table 11. Host class handler structure

Structure member	Description
Name	Class name
ClassCode	USB class code
Init	Class interface Init: initializes the pipes needed for handling the class. Called during the core HOST_CHECK_CLASS
Deinit	Class interface DeInit: de-initializes the interface. Called during the device disconnection or when performing a host stop
Requests	Class control requests: state machine for handling class requests called during class initialization in host core state HOST_CLASS_REQUEST
BgndProcess	Class operation background process. Called from the core state machine during HOST_CLASS state.

Table 11. Host class handler structure (continued)

Structure member	Description
SOFProcess	Class SOF process: handles the class operation that should be called periodically from SOF interrupt handler. Should be used to schedule periodic transfers (interrupt, isochronous).
pData	Initialized during class initialization with a class handle structure that maintains the class process variables

Note: Control class requests can be also issued in HOST_CLASS state.

4.2 USB mass-storage class (MSC)

The MSC background class is used to access the common USB flash pendrives, using the BOT “Bulk-Only Transport” protocol and the Transparent SCSI command set.

The MSC class is implemented using the files listed in *Table 12*.

Table 12. Files used for USB MSC implementation

File	Description
usbh_msc.c	Implements mass-storage class handler and class APIs
usbh_msc_bot.c	Implements the Bulk-Only Transfer (BOT) protocol state machine
usbh_msc_scsi.c	Implements SCSI commands (Read10, Write10, ...)

The MSC class handler is implemented using a structure of the type USBH_ClassTypeDef:

```
USBH_ClassTypeDef  USBH_msc =
{
   "MSC",
   USB_MSC_CLASS,
   USBH_MSC_InterfaceInit,
   USBH_MSC_InterfaceDeInit,
   USBH_MSC_ClassRequest,
   USBH_MSC_BgndProcess,
   NULL,
   USBH_MSC_Handle
};
```

4.2.1 MSC class interface initialization

During the MSC class interface initialization, two bulk pipes are created: one bulk IN and one bulk OUT for handling BOT protocol.

4.2.2 MSC control class requests

Table 13 lists the implemented class requests.

Table 13. USB host mass storage class handlers

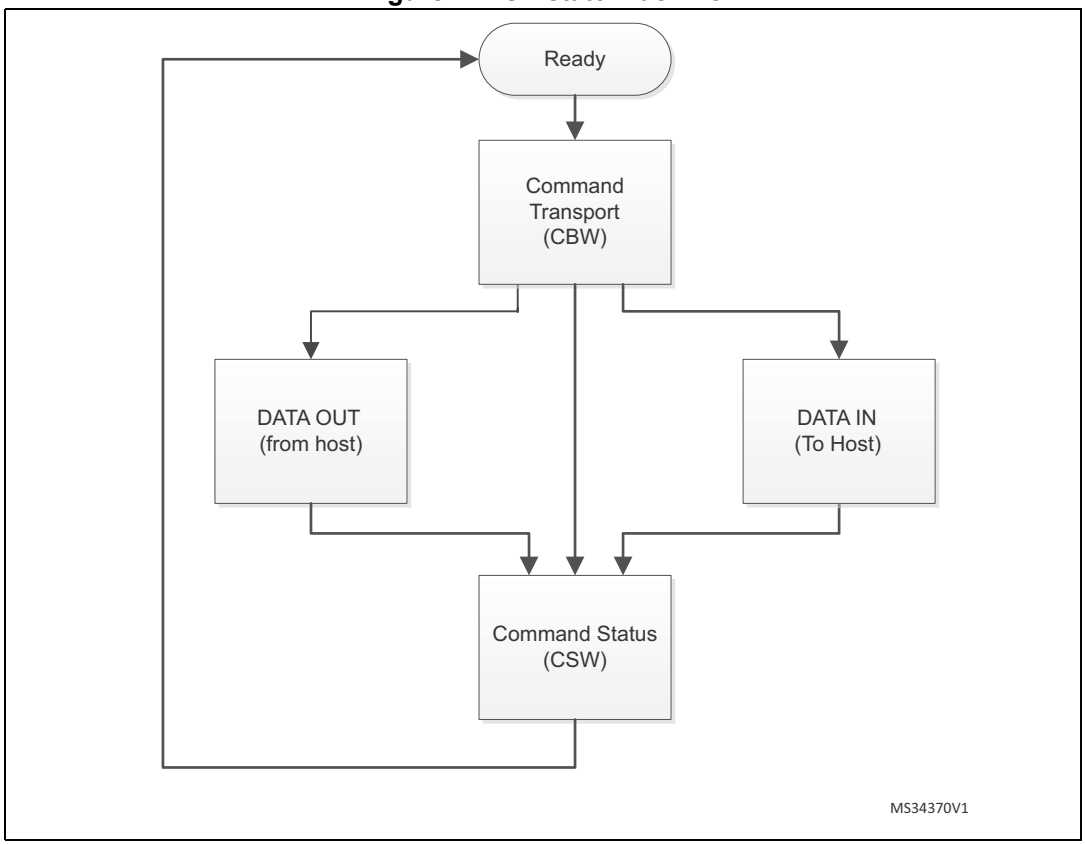
Class request	Description
Get_Max_LUN	Issued to get the number of supported logical units. This command is issued during the class initialization in the HOST_CLASS_REQUEST state
BOT_Reset	This class request is issued in case of BOT error in HOST_CLASS state

4.2.3 MSC class process

The MSC class process handles the issuing of the SCSI commands to get the information about each supported logical unit and to process the disk read/write operations.

The SCSI commands use the Bulk Only Transport (BOT) protocol which implements the three stages of the state machine: command, data and status (*Figure 7*).

Figure 7. BOT state machine



For more details about the BOT, you can refer to the specification available on the website. (*http://www.usb.org*).

Table 14 provides the list of supported SCSI commands.

Table 14. SCSI commands

SCSI command	Description
TestUnitReady	Tests if media is ready
ReadCapacity10	Reads the media capacity
Inquiry	Gets some information about the mass-storage device (for example vendor, version)
RequestSense	Used to get error information
Write10	Writes a data block (defined by a number of sectors and a start sector address)
Read10	Reads a data block (defined by a number of sectors and a start sector address)

4.2.4 MSC class-specific APIs

The MSC class offers the following class-specific APIs which are called by a file system interface (like the FATFS diskio interface in STM32Cube).

Table 15. MSC class specific APIs

API	Description
USBH_MSC_Read	Reads a number of sectors from a logical unit (this function blocks until the operation ends)
USBH_MSC_Write	Writes a number of sectors in a logical unit (this function blocks until the operation ends)
USBH_MSC_GetMaxLUN	Gets the number of logical units (LUNs)
USBH_MSC_GetLUNInfo	Returns a data structure filled with information about a logical unit (for example the capacity)
USBH_MSC_IsReady	Checks if the mass-storage device is ready for read/write operations

4.2.5 MSC class typical usage flow

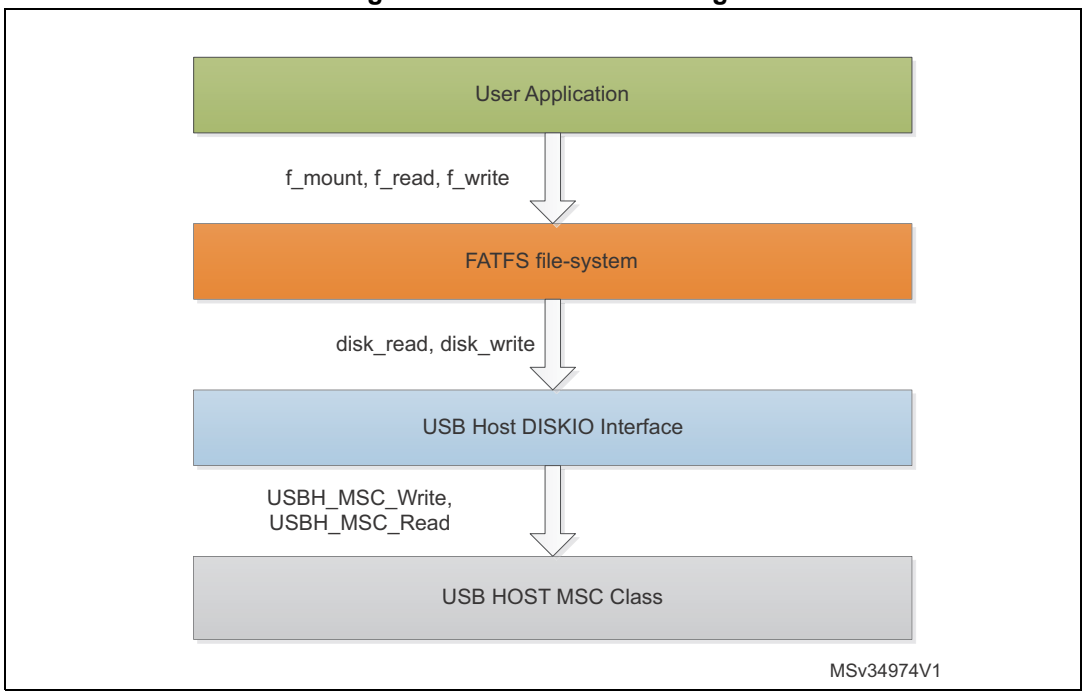
To access a mass-storage class device, the host should implement a file-system. In STM32Cube solution, the FATFS file-system is used to access the FAT based mass-storage device.

As shown in figure below, the interface between the FATFS filesystem and the USB host MSC class is done through a diskio interface.

When not using FATFS, you can easily interface your own file system to host the MSC library by writing a diskio driver similar to the one provided in the STM32Cube host library MSC examples for FATFS.

You can refer to the file *usbh_diskio.c* in USB Host MSC_Standalone or MSC_RTOS example to see implementation of the diskio interface.

Figure 8. USB MSC class usage



Note: in the standalone diskio driver implementation, the disk acess function (disk_read, disk_write) are blocking function that exit only after end of disk read or write operation or on a timeout.

For the RTOS diskio driver implementation, MSC class APIs USBH_MSC_Write and USBH_MSC_Read functions are blocked when the bulk data transfer is on-going, to allow other tasks to run while the hardware access is on-going.

4.3 USB HID mouse and keyboard class (HID)

The HID class is used to access the mouse and keyboard devices using the boot protocol. The class offers also generic HID APIs allowing to customize the class to handle other HID devices.

The HID class is implemented using the files described in *Table 16*.

Table 16. Files used for the implementation of the HID class

File	Description
usbh_hid.c	Implements the HID class
usbh_hid_parser.c	HID descriptor and report parsing functions
usbh_hid_mouse.c	HID mouse-specific functions
usbh_hid_keybd.c	HID keyboard-specific functions
usbh_hid_usage.h	Common defines for HID class

The HID class handler is implemented using a structure of type USBH_ClassTypeDef which is defined as follows:

```
USBH_ClassTypeDef  HID_Class =
{
   "HID",
   USB_HID_CLASS,
   USBH_HID_InterfaceInit,
   USBH_HID_InterfaceDeInit,
   USBH_HID_ClassRequest,
   USBH_HID_BgndProcess,
   USBH_HID_SOF_Process,      /* SOF process to handle Interrupt pipe */
   USBH_HID_Handle            /* Handle structure to maintain HID
                                 background process variable*/
};
```

4.3.1 HID class interface initialization

During interface initialization and depending on the device endpoint descriptors, one interrupt IN endpoint and/or one interrupt OUT endpoint are allocated and opened. For boot mouse and keyboards only the interrupt IN pipe is used.

4.3.2 HID class requests

The control class requests issued in the HID mouse/keyboard class are described in *Table 17*.

Table 17. HID class requests

Class request	Description
Set Idle	Sets the HID mouse/keyboard polling period. Issued during HOST_CLASS_REQUEST state.
Set Protocol	Sets the HID protocol to boot protocol for the mouse and HID. Issued during HOST_CLASS_REQUEST state.

4.3.3 HID class process

The handling of the boot mouse/keyboard is done through the HID report reception using the interrupt IN pipe.

The boot mouse/keyboard handling is assured by two processes, the background process and the SOF process.

- The SOF process handles the issuing of IN transactions synchronized with SOF event.
- The background process handles the data reception and informs the application layer through a data reception callback function.

A HID handle structure *USBH_HID_Handle* is used internally to keep the process variables.

4.3.4 HID specific APIs and event callbacks

Table 18 lists the APIs and event callbacks defined in the HID class. Some APIs are not used to handle the boot mouse/keyboard but they are provided in case the user wants to customize the HID class for another usage than mouse/keyboard.

Table 18. HID APIs and event callbacks

API	Description
USBH_HID_GetReport	Gets report data through control channel (not used in case of HID boot mouse/keyboard)
USBH_HID_SetReport	Sends report data through control channel (not used in case of HID boot mouse/keyboard)
USBH_HID_SetIdle	Sets HID polling period
USBH_HID_EventCallback	Application event callback: called when the mouse or keyboard HID report data is received on IN interrupt pipe
USBH_HID_SetProtocol	Issues a Set_Protocol control request
USBH_HID_GetHIDDescriptor	Gets the HID descriptor
USBH_HID_GetHIDReportDescriptor	Gets the HID report descriptor (not used for boot mouse/keyboard)
USBH_HID_GetDeviceType	Returns the HID device type: mouse of keyboard
USBH_HID_GetMouseInfo	Gets the mouse report data in the structure of type HID_MOUSE_Info_TypeDef (see below)
USBH_HID_GetKeybdInfo	Gets keyboard report data in the structure of type HID_KEYBD_Info_TypeDef (see below)
USBH_HID_GetASCIICode	Converts the keyboard key to ASCII code

Below are the structures used for the mouse and the keyboard report data, respectively.

```
typedef struct _HID_MOUSE_Info
{
   int8_t               x;
   int8_t               y;
   uint8_t            buttons[3];
}
HID_MOUSE_Info_TypeDef;
typedef struct
{
  uint8_t lctrl;     /* left control*/
  uint8_t lshift;    /* left shift  */
  uint8_t lalt;      /* left alt    */
  uint8_t lgui;      /* left alt graph */
  uint8_t rctrl;     /* right control */
  uint8_t rshift;    /* right shift */
  uint8_t ralt;      /* right alt */
  uint8_t rgui;      /* right alt graph */
  uint8_t keys[KBR_MAX_NBR_PRESSED];   /* array of keyboard keys pressed
simulataneously*/
}
HID_KEYBD_Info_TypeDef;
```

4.3.5 HID class usage flow

When interfacing to a HID mouse or keyboard device, the application can get the HID mouse or keyboard report data by periodically polling for HID reports using *USBH_HID_GetKeybdInfo* or *USBH_HID_GetMouseInfo*, these APIs allow to read HID report from a dedicated software FIFO memory.

The application can also use HID class callback function *USBH_HID_EventCallback( )* to get notified in a synchronous way for HID report reception.

For custom HID devices (other than boot mouse/keyboard), the application can use the *USBH_HID_SetReport* or *USBH_HID_GetReport* to send or get HID report through the control pipe.

4.4 USB communication device class (CDC)

The CDC class implementation is used to access the CDC virtual comport devices compliant with the Abstract Control Model (ACM) subclass.

The CDC class is implemented in the class file *usbh_cdc.c/h*. This file defines the CDC class-handler and the class-specific API. The CDC class- handler is of the type *USBH_ClassTypeDef* defined as follows:

```
USBH_ClassTypeDef  CDC_Class =
{
   "CDC",
```

```
  USB_CDC_CLASS,
  USBH_CDC_InterfaceInit,
  USBH_CDC_InterfaceDeInit,
  USBH_CDC_ClassRequest,
  USBH_CDC_BgndProcess,
  NULL,
  USBH_CDC_Handle           /*CDC handle to maintain process variable*/
};
```

4.4.1 CDC interface initialization

During the class initialization, the CDC/ACM class interface allocates and opens three pipes:

- two bulk pipes: one bulk pipe IN and one bulk pipe OUT for CDC/ACM data transfer
- one Interrupt IN pipe for CDC/ACM event notification (not used in current CDC class implementation).

4.4.2 CDC class requests

Table 19 lists the CDC class requests that are issued.

Table 19. CDC class requests

Class request	Description
Get coding line	Gets the current coding line parameters. Issued in HOST_CLASS_REQUEST state.
Set coding line	Sets the line coding parameters (par example baud rate, parity, stop bit). Issued in HOST_CLASS state.

4.4.3 CDC class process

The CDC class background process handles the data reception and transmission on the bulk pipes.

The application is informed of the operation completion by using the callback functions.

The USBH_CDC_Handle stucture maintains internally the variables used during the data transfer. This allows a multi-instance usage of the CDC class handler.

4.4.4 CDC specific APIs and callback functions

The APIs listed in *Table 20* are used to handle the CDC class.

Table 20. CDC class APIs and callback functions

Function	Description
USBH_CDC_GetLineCoding	Gets the current line coding
USBH_CDC_SETLineCoding	Sets the current line coding
USBH_CDC_Transmit	Transmits data
USBH_CDC_Receive	Receives data

Table 20. CDC class APIs and callback functions (continued)

Function	Description
USBH_CDC_TransmitCallback	Callback for data transmitted event
USBH_CDC_ReceiveCallback	Callback for data received event

The line coding data is managed in a structure of the type *CDC_LineCodingTypeDef* defined as follows:

```
typedef union _CDC_LineCodingStructure
{
  uint8_t Array[LINE_CODING_STRUCTURE_SIZE];
  struct
  {
    uint32_t dwDTERate;     /*Data terminal rate, in bits per second*/
    uint8_t bCharFormat;   /*Stop bits  0 - 1 Stop bit
                                        1 - 1.5 Stop bits
                                        2 - 2 Stop bits*/
    uint8_t bParityType;   /* Parity    0 - None
                                        1 - Odd
                                        2 - Even
                                        3 - Mark
                                        4 - Space*/
    uint8_t bDataBits;     /* Data bits (5, 6, 7, 8 or 16). */
  }b;
}
CDC_LineCodingTypeDef;
```

4.4.5 CDC class usage flow

When using the CDC class, the application needs first to set the coding line parameters for virtual comport device through the function *USBH_CDC_SETLineCoding*. Then the application can start to send or receive data using API functions *USBH_CDC_Transmit*/ *USBH_CDC_Receive*. These functions are non-blocking functions. End or transmit or receive operation is notified by using the callback functions *USBH_CDC_TransmitCallback* / *USBH_CDC_TransmitCallback* .

4.5 USB audio class

The USB audio class can be used to access a USB speaker device that is compliant with the USB Audio class 1.0 specification.

The USB Audio class is implemented in class file *usbh_audio.c/.h*, using the following class handler:

```
USBH_ClassTypeDef  AUDIO_Class =
{
```

```
  "AUDIO",
   AC_CLASS,
   USBH_AUDIO_InterfaceInit,
   USBH_AUDIO_InterfaceDeInit,
   USBH_AUDIO_ClassRequest,
   USBH_AUDIO_Bgnd_Process,
   USBH_AUDIO_SOF_Process, /* SOF process to manage isochronous pipe(Note) */
   USBH_AUDIO_Handle
};
```

Note: in the V3.0.0 of the STM32 host library, the SOF callback is not used for handling the isochronous traffic, all handling is done in the background process. This is changed starting from V3.1.0 release of the library, in order to keep isochronous audio data packets transfered by the SOF process, and the background process is used for handling the audio play state machine.

4.5.1 Audio class interface initialization

During the interface initialization up to three pipes can be allocated and opened:

- One isochronous OUT pipe used to handle the speaker audio data
- One isochronous IN pipe used when a microphone is detected (this pipe is not used in the current audio class implementation)
- One interrupt IN pipe used to handle the audio control.

Note: this host audio class implementation does not handle the feedback pipe for audio synchronization. The audio device should support some other synchronization methods.

4.5.2 Audio class control requests

Table 21 lists the audio class control requests that are supported.

Table 21. Audio class control requests

Request	Description
Audio control SET request (CUR, MAX, MIN, RES)	Issued by the audio process state machine to set the volume control information
Audio control GET request (CUR, MAX, MIN, REST)	Issued during the HOST_CLASS_REQUEST state to get volume control information (Max, Min, Resolution)
Audio set sampling frequency	Issued by the Audio class process to set the audio sampling frequency before audio play

4.5.3 AUDIO class process

The Audio class process is handled by both the background process and the SOF process.

The SOF process is used for isochronous audio data transfer and the background process is used to handle a state machine for audio play and audio volume control.

The user application needs to call *USBH_AUDIO_Play* to start an audio buffer play, this function provides the audio data buffer pointer from which out streaming starts and it

changes the background process state machine to AUDIO_PLAYBACK_PLAY state for the audio play operation.

After starting the play process, it is up to the user application to poll for the current read position in the audio data buffer. This is done by using *USBH_AUDIO_GetOutOffset* API function.

If the current buffer read position returned by *USBH_AUDIO_GetOutOffset* reaches a defined threshold, the user application needs to assign a new data buffer that is used later by the audio play process when the current buffer end is reached. This can be done using the *USBH_AUDIO_ChangeOutBuffer* API function.

The application audio data buffer size should be calculated based on the two following parameters:

- Application throughput for filling the audio data buffer (for example when reading audio file from SD card)
- The needed audio sampling rate, which gives the number of bytes that need to be sent every 1ms frame.

4.5.4 AUDIO class APIs

Table 22 lists the audio class APIs that are supported.

Table 22. Audio class APIs

API	Description
USBH_AUDIO_SetVolume	Sets the audio volume
USBH_AUDIO_SetFrequency	Sets the sampling frequency
USBH_AUDIO_Play	Starts play of audio stream from a data source buffer
USBH_AUDIO_Pause	Stops audio stream playing
USBH_AUDIO_Resume	Resume Audio stream playing
USBH_AUDIO_ChangeOutBuffer	Changes audio stream data source buffer
USBH_AUDIO_GetOutOffset	Returns the current read position in audio data source buffer

4.5.5 Audio class usage flow

Before starting the audio play, the application should call the function *USBH_AUDIO_SetFrequency* to set the needed sampling frequency for the device, optionally it can also set the default play volume by calling the function *USBH_AUDIO_SetVolume*.

Then when a first audio stream data buffer is ready, the application can start the audio play by calling the function *USBH_AUDIO_Play*.

The application needs to poll for the current read position in the audio buffer using the function *USBH_AUDIO_GetOutOffset*. When a defined offset is reached, the application needs to provide a new audio databuffer that will be used by the audio class when the end of the current buffer user for play is reached. The function to be called to allocate this new buffer is *USBH_AUDIO_ChangeOutBuffer*.

4.6 USB Media Transport Protocol class (MTP)

The Media Transport Protocol class is widely used in portable devices like Android smartphones. This protocol is similar to the mass-storage class, as it allows access to media files (for example audio and pictures), with the difference that media in the device is not seen as a file system but as a set of objects that can be accessed in transactional way (for example get object or delete object) operations and not as file-system with sectors read/write access.

The MTP protocol comes as an extension to PTP (Picture Transfer Protocol) which is defined in ISO 15740 specification. More details about the MTP protocol can be found in the USB-IF MTP class specification document: Media Transfer Protocol rev 1.1.

The MTP class is implemented in the two files: usbh_mtp.c/.h and usbh_mtp_ptp.c/.h, it uses the following class handler

```
USBH_ClassTypeDef  MTP_Class =
{
   "MTP",
   USB_MTP_CLASS,
   USBH_MTP_InterfaceInit,
   USBH_MTP_InterfaceDeInit,
   USBH_MTP_ClassRequest,
   USBH_MTP_Process,
   USBH_MTP_SOFProcess,
   USBH_MTP_Handle,
};
```

4.6.1 MTP interface initialization

The MTP class intializes three channels:

- one Bulk IN channel for MTP data IN transfer,
- one Bulk OUT channel for MTP data OUT transfer,
- one Interrupt IN channel for MTP events notification.

4.6.2 MTP class control requests

The MTP class does not implement any class specific control requests.

4.6.3 MTP class process

The MTP class is handled by a background process and the SOF process.

The background process handles the opening of a session with MTP device then it acquires the device information and storage unit information.

The SOF process handles the interrupt IN channel to get the MTP device event notifications.

After opening a session with device, all subsequent operations are done using the MTP user application APIs which are blocking APIs (for example USBH_MTP_GetObjectInfo, USBH_MTP_GetObject).

4.6.4 MTP user application APIs and callbacks

The following table lists the APIs and callbacks offered by MTP class:

Table 23. MTP APIs and callbacks

API	Description
USBH_MTP_IsReady	checks if MTP device is ready (session opened)
USBH_MTP_GetNumStorage	gets number of storage units in MTP device
USBH_MTP_SelectStorage	selects a particular storage unit
USBH_MTP_GetStorageInfo	gets storage unit infos
USBH_MTP_GetNumObjects	gets number of objects with particular object format in a storage unit
USBH_MTP_GetObjectHandles	returns an array contaings handlers of the objects in a storage unit
USBH_MTP_GetObjectInfo	Gets an object infos
USBH_MTP_DeleteObject	Deteles an object
USBH_MTP_GetObject	Gets an object
USBH_MTP_GetPartialObject	Gets a part of an object
USBH_MTP_GetObjectPropsSupported	Checks supported object properties
USBH_MTP_GetObjectPropDesc	Gets object properties
USBH_MTP_SendObject	Sends an object to MTP device
USBH_MTP_GetDevicePropDesc	Gets device properties description data set
USBH_MTP_EventsCallback	User callback for MTP event notification

4.6.5 MTP class usage flow

Before accessing the MTP device, the application needs to check that the session with the device is correctly opened using the API function *USBH_MTP_IsReady*.

If it is the case, the application can then start to check for objects with specific format (for example WAV objects). This can be done by using the function *USBH_MTP_GetNumObjects* to check the number of available objects with specific format. The handling of these objects can be retrieved by using the function *USBH_MTP_GetObjectHandles*.

To read the object information (for example object filename), the application needs to call the function *USBH_MTP_GetObjectInfo*.

Finally, the application can start the access to the MTP object by using the APIs *USBH_MTP_GetPartialObject* or *USBH_MTP_GetObject*.

5 Using the USB host library

5.1 USB host library configuration options

The configuration options for the host library are defined in file *usbh_conf.h*, and described in the following table.

Table 24. USB host library configuration options

Configuration options	description
USBH_MAX_NUM_ENDPOINTS	maximum number of supported endpoints
USBH_MAX_NUM_INTERFACES	maximum number of supported interfaces
USBH_MAX_NUM_CONFIGURATION	maximum number of supported configurations
USBH_MAX_NUM_SUPPORTED_CLASS	maximum number of supported classes
USBH_KEEP_CFG_DESCRIPTOR	when defined as 1, all the configuration descriptor will be kept in memory
USBH_MAX_SIZE_CONFIGURATION	defines the maximum size of configuration descriptor
USBH_MAX_DATA_BUFFER	defines the maximum data buffer for data transfer
USBH_DEBUG_LEVEL	defines the log level: – 0: no logs – 1: user messages logs – 2: user and error messages logs – 3: user, error and debug messages logs
USBH_USE_OS	when defined as 1, configures host to work in OS mode

The *usbh_conf.h* file provides also the redefinition for memory management functions used in the library: *USBH_malloc*, *USBH_free USBH_memset*, *USBH_memory.*

5.2 Using the host library in standalone mode

When using the library in standalone mode, a typical main function, should contain the following:

```
void main ()
{
...
/* Init Host Library */
  USBH_Init(&hUSBHost, USBH_UserProcess, 0);

  /* Add Supported Class: example HID class*/
  USBH_RegisterClass(&hUSBHost, USBH_HID_CLASS);


/* Start Host Process */
```

```
  USBH_Start(&hUSBHost);

  /* Application main loop*/
  while (1)
  {
    /*Application background process */
    Application_Process();

    /* USB Host process : should be called in the main loop to handle host
       stack*/
    USBH_Process();
  }
```

The USBH_UserProcess callback handles theUSB host core events (for example disconnection, connection, class active). Typically, it should be implemented as shown hereafter to handle application process state machine:

```
void USBH_UserProcess  (USBH_HandleTypeDef *phost, uint8_t id)
{
  switch (id)
  {
  case HOST_USER_DISCONNECTION:
    Appli_state = APPLICATION_DISCONNECT;
    break;

   /* when HOST_USER_CLASS_ACTIVE event is received, application can start
      communication with device*/
  case HOST_USER_CLASS_ACTIVE:
    Appli_state = APPLICATION_READY;
    break;

  case HOST_USER_CONNECTION:
    Appli_state = APPLICATION_START;
    break;

   default:
    break;
  }
}
```

Please note that the application can register multiple classes, for example:

```
/* Add Supported Class: example HID and MSC classes*/
  USBH_RegisterClass(&hUSBHost, USBH_HID_CLASS);
              USBH_RegisterClass(&hUSBHost, USBH_MSC_CLASS);
```

The user application can determine the enumerated class using core API function *USBH_GetActiveClass()* when HOST_USER_CLASS_ACTIVE event occurs.

5.3 Using the host library in RTOS mode

In RTOS mode, the application needs to define the USBH_USE_OS in the file *usbh_conf.h*.

When the RTOS mode is used the host core background process runs as a separate RTOS task. The communication between the application task and the host core task uses the RTOS message queue mechanism. The CMSIS RTOS APIs are used to support the RTOS mode.

From user point of view, using the host library in RTOS mode is almost transparent as the same APIs are used in RTOS or in Standalone mode.

5.3.1 Typical operation in RTOS mode

In RTOS mode, the user needs to define at least one application, and also define an initialization thread for the application and host initialization. The following is an extract from the MSC_RTOS mode example of the library. The example initializes a user thread and a start thread.

The start thread initializes the application and the USB host by registering the supported class and by starting the host operation in RTOS mode.

```
void main()
{
   ...
   /*defining a User thread and a Start thread*/
  osThreadDef(USER_Thread, StartThread, osPriorityNormal, 0,
8onfigMINIMAL_STACK_SIZE);
  osThreadCreate(osThread(USER_Thread), NULL);
   ...
  for( ;; );
}


static void StartThread(void const *argument)
{
  osEvent event;

  /* Init MSC Application */
  MSC_InitApplication();

  /* Start Host Library*/
  USBH_Init(&hUSBHost, USBH_UserProcess, 0);

  /* Add Supported Class */
  USBH_RegisterClass(&hUSBHost, USBH_MSC_CLASS);

  /* Start Host Process as task (since USBH_USE_OS =1) */
  USBH_Start(&hUSBHost);
```

```
  for( ;; )
  {
    event = osMessageGet(AppliEvent, osWaitForever);

    if(event.status == osEventMessage)
    {
      switch(event.value.v)
      {
      case APPLICATION_DISCONNECT:
        Appli_state = APPLICATION_DISCONNECT;
        break;

      case APPLICATION_READY:
        Appli_state = APPLICATION_READY;

      default:
        break;
      }
    }
  }
}
```

As seen in the above extracted code, the Start thread first initializes first the application, then it initializes the host (as in standalone mode) and finally it blocks the waiting for USB application events (Application_Disconnect, Application Ready). These events are notified using the host user process callback function which should be implemented as shown hereafter:

```
static void USBH_UserProcess(USBH_HandleTypeDef *phost, uint8_t id)
{
  switch(id)
  {
  case HOST_USER_DISCONNECTION:
    osMessagePut(AppliEvent, APPLICATION_DISCONNECT, 0);
    break;

  case HOST_USER_CLASS_ACTIVE:
    osMessagePut(AppliEvent, APPLICATION_READY, 0);
    break;
  default:
    break;
  }
}
```

Note: as shown in the USBH_UserProcess function, the message queue mechanism is used to notify the Start thread of USB events.

5.4 Customizing the low interface file usbh_conf.c

To customize the low level interface file *usbh_conf.c* it is recommended to start from the implementation provided in the examples.

As a reference example we can take the *usbh_conf.c* file used for the MSC class standalone example running on the STM324xG evaluation board. This file is available in the STM32Cube_FW_F4 folder from the following path:
`\Projects\STM324xG_EVAL\Applications\USB_Host\MSC_Standalone\Src`

When opening the *usbh_conf.c* file, you can find three groups of functions:

- HCD BSP Routines: USB Host controller board support package functions,
- Low Level Driver Callbacks: USB Host controller HAL driver callbacks implementations,
- LL driver Interface: USB host library Low-Level interface APIs implementation (refer to *Table 9: Low-level interface APIs*).

5.4.1 USB Host Controller BSP functions

The HCD BSP routines include two functions:

- void HAL_HCD_MspInit(HCD_HandleTypeDef *hhcd)
- void HAL_HCD_MspDeInit(HCD_HandleTypeDef *hhcd)

The function HAL_HCD_MspInit is used for the following:

- USB Host controller and GPIO ports clock enable,
- The needed IOs configuration,
- Configuring and enabling the interrupt for USB Host.

As seen in file *usbh_conf.c*, two implementations of the BSP are available depending on the Full-speed (FS) or High-speed (HS) Host controller.

The users need to customize the IO configuration according to the pin selection on their hardware. For example different alternate functions remapping options are available for ULPI DIR line.

In addition to configuring the USB IOs, the users may need to configure one GPIO pin to be able to drive a charge pump IC for VBUS voltage generation. See in the example *usbh_conf.c* file where for FS controller the pin PH5 was configured as output push-pull to be used for charge pump IC driving. For the HS controller no pin is used since the charge pump IC control is done by the PHY.

5.4.2 USB Host controller HAL driver callbacks

The USB Host controller (HCD) HAL driver implemented callbacks are listed in the following table.

Table 25. USB Host controller (HCD) HAL driver callbacks

USB Host HAL driver callback	Description
HAL_HCD_SOF_Callback	called by host driver on SOF event, it calls host library callback USBH_LL_IncTimer, for USB classes using interrupt and isochronous pipes, this callback should call also USBH_LL_SOF host library callback function
HAL_HCD_Connect_Callback	called by host driver on Connect event, it calls host library callback USBH_LL_Connect
HAL_HCD_Disconnect_Callback	called by host driver on disconnect event, it calls host library callback USBH_LL_Disconnect

Please use the above callback functions «as is» and do not modify them.

5.4.3 USB host library low-level interface APIs

This group of functions implements the low-level API layer of the USB host library. It acts as link layer with the USB Host controller HAL driver APIs.

The implemented functions are listed in *Table 9: Low-level interface APIs*.

When starting from the *usbh_conf.c* file given an example, three functions can be customized by the user while the other functions must be kept “as is”.

- USBH_LL_Init
- USBH_LL_DriverVBUS
- USBH_Delay

The user can use the parameters listed in the table below to configure the function *USBH_LL_Init()*.

Table 26. USBH_LL_Init configuration options

Config option	Description
hhcd.Instance	can be USB_OTG_FS or USB_OTG_FS
hhcd.Init.dma_enable	when using the OTG_HS peripheral, you can enable or disable DMA
hhcd.Init.low_power_enable	For future use. Allows to handle low power mode during host suspend mode
hhcd.Init.phy_itface	Selects the PHY interface, it can be HCD_PHY_EMBEDDED or HCD_PHY_ULPI
hhcd.Init.Sof_enable	when enabled, it allows to output SOF pulse on SOF pin
hhcd.Init.speed	Host speed, can be HCD_SPEED_FULL or HCD_SPEED_HIGH
hhcd.Init.use_external_vbus	When USB HS host controller is used, this parameter enabled VBUS driving from the ULPI PHY

The user needs to customize the function U*SBH_LL_DriverVBUS()* to correctly drive the USB VBUS charge pump IC. The following is the implementation in the example *usbh_conf.c* file. It drives the GPIO pin PH5:

```
USBH_StatusTypeDef USBH_LL_DriverVBUS(USBH_HandleTypeDef *phost, uint8_t
state)
{
  if(state == 0)
  {
    HAL_GPIO_WritePin(GPIOH, GPIO_PIN_5, GPIO_PIN_SET);
  }
  else
  {
    HAL_GPIO_WritePin(GPIOH, GPIO_PIN_5, GPIO_PIN_RESET);
  }

  HAL_Delay(200); // 200ms delay VBUS stabilization time
  return USBH_OK;
}
```

Finally, the function *USBH_LL_Delay* is used to handle delays in ms for the USB host library. It can be implemented by calling HAL library delay function *HAL_Delay()* or by using other custom delay routine.

5.5 FAQs

Does the USB host library support composite devices (for example Mass-storage + HID)?

Yes, providing the users write a custom composite class handler for the composite device.

Can I use the host library with my own USB host controller driver?

Yes, you just need to implement the low-level link interface layer (*usbh_conf.c* file) that sets the adaptation with your USB host controller driver.

Can I use at the same time the OTG HS and the OTG FS host controller?

Yes, you can refer to the dual core example available in `\Applications\USB_Host\DualCore_Standalone`

Can the USB host library handle devices connected through a USB HUB?

No.The USB host library does not support HUB class.

Does the USB host library allow to handle multiple configuration devices?

No. Only single configuration devices are supported

6 Revision history

Table 27. Document revision history

Date	Revision	Changes
21-May-2014	1	Initial release.
05-Feb-2015	2	Updated *Section : Introduction*.
27-May-2015	3	*Section : Introduction* updated and merged with section *STM32Cube overview*.