UM0424
User manual

STM32F10xxx USB development kit

Introduction

The STM32F10xxx USB development kit is a complete firmware and software package including examples and demos for all USB transfer types (control, interrupt, bulk and isochronous). It supports all STM32F10xxx Microcontroller families.

The aim of the STM32F10xxx USB development kit is to use the STM32F10xxx USB library with at least one firmware demo per USB transfer type.

This document presents a description of all the components of the STM32F10xxx USB development kit, including:

- STM32F10xxx USB library: All processes related to default endpoint and standard requests
- Device Firmware Upgrade Demo: Control transfer
- Joystick mouse demo: Interrupt transfer
- Mass storage demo: Bulk transfer
- Virtual COM port: Bulk transfer
- USB voice demo (USB speaker): Isochronous transfer

Contents

List of tables

List of figures

1 STM32F10xxx USB firmware library

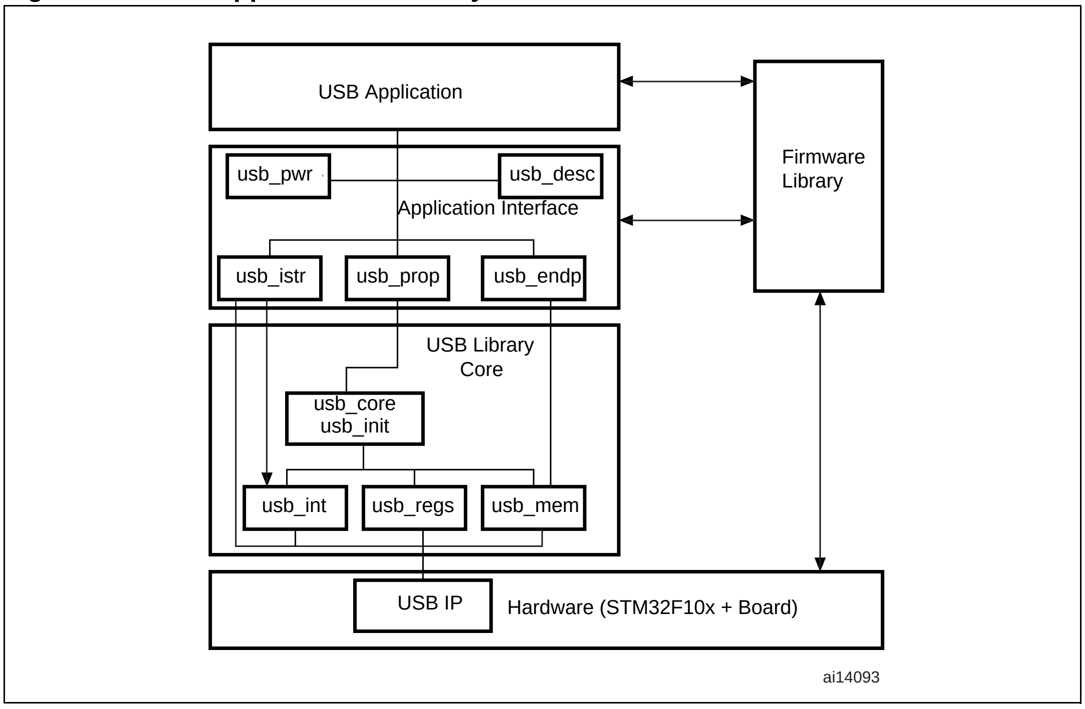
This section describes the firmware interface (called USB Library) used to manage the STM32F10xxx USB 2.0 full-speed Device peripheral.

The main purpose of this firmware library is to provide resources to ease the development of applications using the USB macrocell in the STM32F10xxx microcontroller family.

1.1 USB application hierarchy

Figure 1 shows the interaction between the different components of a typical USB application and the USB library.

Figure 1. USB application hierarchy



The USB library is divided into two layers:

- **USB Library Core layer:** this layer manages the direct communication with the USB IP hardware and the USB standard protocol. The USB Library Core is compliant with the USB 2.0 specification and is separate from the standard STM32F10xxx Firmware Library.
- **Application Interface layer:** this layer provides the user with a complete interface between the library core and the final application.

Note: The application interface layer and the final application can communicate with the Firmware Library to manage the hardware needs of the application.

A detailed description of these two layers with coding rules is provided in the next two sections.

1.2 USB library core

Table 1 presents the USB library core modules:

Table 1. **USB library core modules**

File	Description
usb_type.h	Types used in the library core. This file is used to guarantee the independence of the USB library
usb_reg (.h, .c)	Hardware abstraction layer
usb_int.c	Correct transfer interrupt service routine
usb_init (.h,.c)	USB initializations
usb_core (.h , .c)	USB protocol management (compliant with chapter 9 of the *USB 2.0 specification*)
usb_mem(.h,.c)	Data transfer management (from/to Packet Memory Area)
usb_def.h	USB definitions

1.2.1 usb_type.h

This file provides the main types used in the library. These types are dependent on the used microcontroller family.

Note: *The type definitions used in the USB library are the same as those used in the* STM32F10xxx *Firmware library to guarantee the autonomy of the whole code.*

1.2.2 usb_reg(.c, .h)

The **usb_regs** module implements the hardware abstraction layer, it offers a set of basic functions for accessing the USB macrocell registers.

Note: *The available functions have two call versions:*

– *As a macro: the call is:* *_NameofFunction(parameter1,...)*
– *As a subroutine: the call is:* *NameofFunction(parameter1,...)*

Common register functions:

These functions could be used to set or to get the different common USB registers:

Table 2. Common register functions

Register	function
CNTR	`void SetCNTR (u16 wValue)`
	`u16 GetCNTR (void)`
ISTR	`void SetISTR (u16 wValue)`
	`u16 GetISTR (void)`
FNR	`u16 GetFNR  (void)`
DADDR	`void SetDADDR  (u16 wValue)`
	`u16 GetDADDR  (void)`
BTABLE	`void SetBTABLE (u16 wValue)`
	`u16 GetBTABLE (void)`

Endpoint registers functions

All operations with endpoint registers can be obtained with the `SetENDPOINT` and `GetENDPOINT` functions. However, many functions are derived from these to offer the advantage of a direct action on a specific field.

a) Endpoint set/get value

```
SetENDPOINT :  void SetENDPOINT(u8 bEpNum,u16 wRegValue)
bEpNum = Endpoint number, wRegValue = Value to write
GetENDPOINT :  u16 GetENDPOINT(u8 bEpNum)
bEpNum = Endpoint number
return value: the endpoint register value
```

b) Endpoint TYPE field

The EP_TYPE field of the endpoint register can assume the defined values below:

```
#define EP_BULK                (0x0000)   // Endpoint BULK
#define EP_CONTROL             (0x0200)   // Endpoint CONTROL
#define EP_ISOCHRNOUS          (0x0400)   // Endpoint ISOCHRONOUS
#define EP_INTERRUPT           (0x0600)   // Endpoint INTERRUPT
```

```
SetEPType :   void SetEPType (u8 bEpNum, u16 wtype)
bEpNum = Endpoint number, wtype = Endpoint type (value from the
above define’s)
GetEPType :   u16 GetEPType (u8 bEpNum)
bEpNum = Endpoint number
return value: a value from the above define’s
```

c) Endpoint STATUS field

The STAT_TX / STAT_RX fields of the endpoint register can assume the defined values below:

```
#define EP_TX_DIS           (0x0000)   // Endpoint TX DISabled
#define EP_TX_STALL         (0x0010)   // Endpoint TX STALLed
#define EP_TX_NAK           (0x0020)   // Endpoint TX NAKed
#define EP_TX_VALID         (0x0030)   // Endpoint TX VALID
#define EP_RX_DIS           (0x0000)   // Endpoint RX DISabled
#define EP_RX_STALL         (0x1000)   // Endpoint RX STALLed
#define EP_RX_NAK           (0x2000)   // Endpoint RX NAKed
#define EP_RX_VALID         (0x3000)   // Endpoint RX VALID
SetEPTxStatus :  void SetEPTxStatus(u8 bEpNum,u16 wState)
SetEPRxStatus :  void SetEPRxStatus(u8 bEpNum,u16 wState)
bEpNum = Endpoint number, wState = a value from the above define’s
GetEPTxStatus :  u16 GetEPTxStatus(u8 bEpNum)
GetEPRxStatus :  u16 GetEPRxStatus(u8 bEpNum)
bEpNum = endpoint number
return value:a value from the above define’s
```

d) Endpoint KIND field

```
SetEP_KIND    :  void SetEP_KIND(u8 bEpNum)
ClearEP_KIND  :  void ClearEP_KIND(u8 bEpNum)
bEpNum = endpoint number
Set_Status_Out   :  void Set_Status_Out(u8 bEpNum)
Clear_Status_Out :  void Clear_Status_Out(u8 bEpNum)
bEpNum = endpoint number
SetEPDoubleBuff    :  void SetEPDoubleBuff(u8 bEpNum)
ClearEPDoubleBuff  :  void ClearEPDoubleBuff(u8 bEpNum)
bEpNum = endpoint number
```

e) Correct Transfer Rx/Tx fields

```
ClearEP_CTR_RX :  void ClearEP_CTR_RX(u8 bEpNum)
ClearEP_CTR_TX :  void ClearEP_CTR_TX(u8 bEpNum)
bEpNum = endpoint number
```

f) Data Toggle Rx/Tx fields

```
ToggleDTOG_RX :  void ToggleDTOG_RX(u8 bEpNum)
ToggleDTOG_TX :  void ToggleDTOG_TX(u8 bEpNum)
bEpNum = endpoint number
```

g) Address field

```
SetEPAdress :  void SetEPAddress(u8 bEpNum,u8 bAddr)
bEpNum = endpoint number
bAddr  = address to be set
GetEPAdress :  BYTE GetEPAddress(u8 bEpNum)
bEpNum = endpoint number
```

Buffer description table functions

These functions are used in order to set or get the endpoints' receive and transmit buffer addresses and sizes.

a) Tx/Rx buffer address fields

```
SetEPTxAddr : void SetEPTxAddr(u8 bEpNum,u16 wAddr);
SetEPRxAddr : void SetEPRxAddr(u8 bEpNum,u16 wAddr);
bEpNum = endpoint number
wAddr  = address to be set (expressed as PMA buffer address)
GetEPTxAddr : u16 GetEPTxAddr(u8 bEpNum);
GetEPRxAddr : u16 GetEPRxAddr(u8 bEpNum);
bEpNum = endpoint number
return value : address value (expressed as PMA buffer address)
```

b) Tx/Rx buffer counter fields

```
SetEPTxCount : void SetEPTxCount(u8 bEpNum,u16 wCount);
SetEPRxCount : void SetEPRxCount(u8 bEpNum,u16 wCount);
bEpNum = endpoint number
wCount = counter to be set
GetEPTxCount : u16 GetEPTxCount(u8 bEpNum);
GetEPRxCount : u16 GetEPRxCount(u8 bEpNum);
bEpNum = endpoint number
return value : counter value
```

Double-buffered endpoints functions

To obtain high data-transfer throughput in bulk or isochronous modes, *double-buffered* mode has to be programmed. In this operating mode some fields of the endpoint registers and buffer description table cells have different meanings.
To ease the use of this feature several functions have been developed.

SetEPDoubleBuff: An endpoint programmed to work in bulk mode can be set as double-buffered by setting the EP-KIND bit. The function `SetEPDoubleBuff()` accomplishes this task.

```
SetEPDoubleBuff : void SetEPDoubleBuff(u8 bEpNum);
bEpNum = endpoint number
```

FreeUserBuffer: In double-buffered mode the endpoints become mono-directional and buffer description table cells of the unused direction are applied to handle a second buffer.

Addresses and counters must be handled in a different way. Rx and Tx Addresses and counter cells become **Buffer0** and **Buffer1** cells. Functions dedicated to this operating mode are provided for in the library.

During a bulk transfer the line fills one buffer while the other buffer is reserved to the application. A user application has to process data before the arrival of bulk needing a buffer. The buffer reserved to the application has to be freed in time.

To free the buffer in use from the application the `FreeUserBuffer` function is provided:

```
FreeUserBuffer: void FreeUserBuffer(u8 bEpNum, u8 bDir);
bEpNum = endpoint number
```

a) Double buffer addresses

These functions set or get buffer address value in the buffer description table for double buffered mode.

SetEPDblBuffAddr : `void SetEPDblBuffAddr(u8 bEpNum,u16 wBuf0Addr,u16 wBuf1Addr);`

SetEPDblbuf0Addr : `void SetEPDblBuf0Addr(u8 bEpNum,u16 wBuf0Addr);`

SetEPDblbuf1Addr : `void SetEPDblBuf1Addr(u8 bEpNum,u16 wBuf1Addr);`

`bEpNum = endpoint number`

`wBuf0Addr, wBuf1Addr = buffer addresses (expressed as PMA buffer addresses)`

GetEPDblBuf0Addr : `u16 GetEPDblBuf0Addr(u8 bEpNum);`

GetEPDblbuf1Addr : `u16 GetEPDblBuf1Addr(u8 bEpNum);`

`bEpNum = endpoint number`

`return value :  buffer addresses`

b) Double buffer counters

These functions set or get buffer counter value in the buffer description table for double buffered mode.

SetEPDblBuffCount: `void SetEPDblBuffCount(u8 bEpNum, u8 bDir, u16 wCount);`

SetEPDblBuf0Count: `void SetEPDblBuf0Count(u8 bEpNum, u8 bDir, u16 wCount);`

SetEPDblBuf1Count: `void SetEPDblBuf1Count(u8 bEpNum, u8 bDir, u16 wCount);`

```
bEpNum = endpoint number
bDir   = endpoint direction
wCount = buffer counter
```

GetEPDblBuf0Count : `u16 GetEPDblBuf0Count(u8 bEpNum);`

GetEPDblBuf1Count : `u16 GetEPDblBuf1Count(u8 bEpNum);`

```
bEpNum  = endpoint number
return value : buffer counter
```

c) Double buffer STATUS

The simple and double buffer modes use the same functions to manage the Endpoint STATUS except for the STALL status for double buffer mode. This functionality is managed by the function:

SetDouBleBuffEPStall: `void SetDouBleBuffEPStall(u8 bEpNum,u8 bDir)`

```
bEpNum = endpoint number
bDir   = endpoint direction
```

1.2.3 usb_int (.c , .h)

The **usb_int** module handles the correct transfer interrupt service routines; it offers the link between the USB protocol events and the library core.

The STM32F10xxx USB peripheral provides two correct transfer routines:

- Low-priority interrupt: managed by the function `CTR_LP()` and used for control, interrupt and bulk (in simple buffer mode).
- High-priority interrupt: managed by the function `CTR_HP()` and used for faster transfer mode like Isochronous and bulk (in double buffer mode).

1.2.4 usb_core (.c , .h)

The usb_core module is the kernel of the library. It implements all the functions described in chapter 9 of the *USB 2.0 specification*.

The available subroutines cover handling of USB standard requests related to the control endpoint (EP0), offering the necessary code to accomplish the sequence of enumeration phase.

A state machine is implemented in order to process the different stages of the setup transactions.

The USB core module also implements a dynamic interface between the standard request and the user implementation using the structure **User_Standard_Requests.**

The USB core dispatches the class specific requests and some bus events to user program whenever it is necessary. User handling procedures are given in the **Device_Property** structure.

The different data and function structures used by the kernel are described in the following paragraphs.

1. **Device table structure**

   The core keeps device level information in the **Device_Table** structure. **Device_Table** is of the type: **DEVICE**.

   ```
   typedef struct _DEVICE {
    u8 Total_Endpoint;
    u8 Total_Configuration;
   } DEVICE;
   ```

2. **Device information structure**

The USB core keeps the setup packet from the host for the implemented USB device in the **Device_Info** structure. This structure has the type: **DEVICE_INFO**.

```
typedef struct _DEVICE_INFO {
 u8 USBbmRequestType;
 u8 USBbRequest;
 u16_u8 USBwValues;
 u16_u8 USBwIndexs;
 u16_u8 USBwLengths;
 u8 ControlState;
 u8 Current_Feature;
 u8 Current_Configuration;
 u8 Current_Interface;
u8 Current_AlternateSetting;
 ENDPOINT_INFO Ctrl_Info;
} DEVICE_INFO;
```

An union **u16_u8** is defined to easily access some fields in the **DEVICE_INFO** in either **u16** or **u8** format.

```
typedef union {
 u16 w;
 struct BW {
 u8 bb1;
 u8 bb0;
 } bw;
} u16_u8;
```

Description of the structure fields:

– **USBbmRequestType** is the copy of the *bmRequestType* of a setup packet

– **USBbRequest** is the copy of the *bRequest* of a setup packet

– **USBwValues** is defined as type: **WORD_BYTE** and can be accessed through 3 macros:

```
#define USBwValue USBwValues.w
#define USBwValue0 USBwValues.bw.bb0
#define USBwValue1 USBwValues.bw.bb1
```

USBwValue is the copy of *the wValue* of a setup packet
USBwValue0 is the low byte of *wValue*, and **USBwValue1** is the high byte of *wValue*.

– **USBwIndexs** is defined as USBwValues and can be accessed by 3 macros:

```
#define USBwIndex USBwIndexs.w
#define USBwIndex0 USBwIndexs.bw.bb0
#define USBwIndex1 USBwIndexs.bw.bb1
```

USBwIndex is the copy of the *wIndex* of a setup packet
USBwIndex0 is the low byte of *wIndex*, and **USBwIndex1** is the high byte of *wIndex*.

– **USBwLengths** is defined as type: **WORD_BYTE** and can be accessed through 3 macros:

```
#define USBwLength USBwLengths.w
#define USBwLength0 USBwLengths.bw.bb0
#define USBwLength1 USBwLengths.bw.bb1
```

  USBwLength is the copy of the *wLength* of a setup packet
  USBwLength0 and **USBwLength1** are the low and high bytes of *wLength*, respectively.

– **ControlState** is the state of the core, the available values are defined in CONTROL_STATE.

– **Current_Feature** is the device feature at any time. It is affected by the SET_FEATURE and CLEAR_FEATURE requests and retrieved by the GET_STATUS request. User code does not use this field.

– **Current_Configuration** is the configuration the device is working on at any time. It is set and retrieved by the SET_CONFIGURATION and GET_CONFIGURATION requests, respectively.

– **Current_Interface** is the selected interface.

– **Current_Alternatesetting** is the alternative setting which has been selected for the current working configuration and interface. It is set and retrieved by the SET_INTERFACE and GET_INTERFACE requests, respectively.

– **Ctrl_Info** has type ENDPOINT_INFO.

  Since this structure is used everywhere in the library, a global variable **pInformation** is defined for easy access to the **Device_Info** table, it is a pointer to the **DEVICE_INFO** structure.

  Actually, **pInformation = &Device_Info**.

3. **Device Property Structure**

The USB core dispatches the control to the user program whenever it is necessary. User handling procedures are given in an array of **Device_Property**. The structure has the type: **DEVICE_PROP**:

```
typedef struct _DEVICE_PROP {
void (*Init)(void);
void (*Reset)(void);
void (*Process_Status_IN)(void);
void (*Process_Status_OUT)(void);
RESULT (*Class_Data_Setup)(u8 RequestNo);
RESULT (*Class_NoData_Setup)(u8 RequestNo);
RESULT  (*Class_Get_Interface_Setting)(u8 Interface,u8
AlternateSetting);
u8* (*GetDeviceDescriptor)(u16 Length);
u8* (*GetConfigDescriptor)(u16 Length);
u8* (*GetStringDescriptor)(u16 Length);
u8 MaxPacketSize;
} DEVICE_PROP;
```

4. **User Standard Request Structure**

   The User Standard Request Structure is the interface between the user code and the management of the standard request. The structure has the type: **USER_STANDARD_REQUESTS:**

```
typedef struct _USER_STANDARD_REQUESTS {
void(*User_GetConfiguration)(void);
void(*User_SetConfiguration)(void);
void(*User_GetInterface)(void);
void(*User_SetInterface)(void);
void(*User_GetStatus)(void);
void(*User_ClearFeature)(void);
void(*User_SetEndPointFeature)(void);
void(*User_SetDeviceFeature)(void);
void(*User_SetDeviceAddress)(void);
} USER_STANDARD_REQUESTS;
```

   If the user wants to implement specific code after receiving a standard USB request they have to use the corresponding functions in this structure.

   An application developer must implement tree structures having the **DEVICE_PROP**, **Device_Table** and **USER_STANDARD_REQUEST** types in order to manage class requests and application specific controls. The different fields of these structures are described in the next section.

1.3 Application interface

The modules of the Application interface are provided as a template, they must be tailored by the application developer for each application. *Table 3* shows the different modules used in the application interface.

Table 3. Application interface modules

File	Description
usb_istr (.c,.h)	USB interrupt handler functions
usb_conf.h	USB configuration file
usb_prop (.c, .h)	USB application-specific properties
usb_endp.c	CTR interrupt handler routines for non-control endpoints
usb_pwr (.h, .c)	USB power management module
usb_desc (.c, .h)	USB descriptors

1.3.1 usb_istr(.c)

USB_istr module provides a function named **`USB_Istr()`** which handles all USB macrocell interrupts.

For each USB interrupt source a callback routine named `XXX_Callback` (for example, `RESET_Callback`) is provided in order to implement a user interrupt handler. To enable the processing of each callback routines, a preprocessor switch named `XXX_Callback` must be defined in the USB configuration file ***USB_conf.h***.

1.3.2 usb_conf(.h)

The *usb_conf.h* is used to:

- Define the BTABLE and all endpoint addresses in the PMA.
- Define the interrupt mask according to the needed events.

1.3.3 usb_endp (.c)

USB_endp module is used for handling the CTR's "correct transfer" routines for endpoints different from endpoint 0 (EP0).

For enabling the processing of these callback handlers a pre-processor switch named EPx_IN_Callback (for IN transfer) or EPx_OUT_Callback (for OUT transfer) must be defined in the ***USB_conf.h*** file.

1.3.4 usb_prop (.c , .h)

The USB_prop module is used for implementing the **Device_Property**, **Device_Table** and **USER_STANDARD_REQUEST** structures used by the USB core.

Device property implementation

The device property structure fields are described below:

– **void Init(void)**: Init procedure of the USB IP. It is called once at the start of the application to manage the initialization process.

– **void Reset(void)**: Reset procedure of the USB IP. It is called when the macrocell receives a RESET signal from the bus. The user program should set up the endpoints in this procedure, in order to set the default control endpoint and enable them to receive.

– **void Process_Status_IN(void)**: Callback procedure, it is called when a status in a stage is finished. The user program can take control with this callback to perform class- and application-related processes.

– **void Process_Status_OUT(void)**: Callback procedure, it is called when a status out stage is finished. As with Process_Status_IN, the user program can perform actions after a status out stage.

– **RESULT** (see note below) ***(Class_Data_Setup)(BYTE RequestNo)**: Callback procedure, it is called when a class request is recognized and this request needs a data stage. The core cannot process such requests. In this case, the user program gets the chance to use custom procedures to analyze the request, prepare the data and pass the data to the USB core for exchange with the host. The parameter RequestNo indicates the request number. The return parameter of this function has the type: RESULT. It indicates the result of the request processing to the core.

– **RESULT (*Class_NoData_Setup)(BYTE RequestNo)** Callback procedure, it is called when a non-standard device request is recognized, that does not need a data stage. The core cannot process such requests. The user program can have the chance to use custom procedures to analyze the request and take action. The return parameter of this function has type: RESULT. It indicates the result of the request processing to the core.

– **RESULT (*Class_GET_Interface_Setting)(u8 Interface, u8 AlternateSetting)**: This routine is used to test the received set interface standard request. The user must verify the "Interface" and "AlternateSetting" according to their own implementation and return the USB_UNSUPPORT in case of error in these two fields.
– **BYTE* GetDeviceDescriptor(WORD Length)**: The core gets the device descriptor.
– **BYTE* GetConfigDescriptor(WORD Length)**: The core gets the configuration descriptor.
– **BYTE* GetStringDescriptor(WORD Length)**: The core gets the string descriptor.
– **WORD MaxPacketSize**: The maximum packet size of the device default control endpoint.

*Note: The **RESULT** type is the following:*

```
typedef enum _RESULT {
USB_SUCCESS = 0,/* request process sucessfully */
USB_ERROR,       /* error
USB_UNSUPPORT,   /* request not supported
USB_NOT_READY/* The request process has not been finished,*/
/* endpoint will be NAK to further requests*/
} RESULT;
```

Device table implementation

Description of the structure fields:

– **Total_Endpoint** is the number of endpoints the USB application uses.
– **Total_Configuration** is the number of configurations the USB application has.

USER_STANDARD_REQUEST implementation

This structure is used to manage the user implementation after receiving all standard requests (except Get descriptors). The fields of this structure are:

– **void (*User_GetConfiguration)(void)**: Called after receiving the Get Configuration Standard request.
– **void (*User_SetConfiguration)(void)**: Called after receiving the Set Configuration Standard request.
– **void (*User_GetInterface)(void)**: Called after receiving the Get interface Standard request.
– **void (*User_SetInterface)(void)**: Called after receiving the Set interface Standard request.
– **void (*User_GetStatus)(void)**: Called after receiving the Get interface Standard request.
– **void (*User_ClearFeature)(void)**: Called after receiving the Clear Feature Standard request.
– **void (*User_SetEndPointFeature)(void)**: Called after receiving the set Feature Standard request (only for endpoint recipient).
– **void (*User_SetDeviceFeature)(void)**: Called after receiving the set Feature Standard request (only for Device recipient).
– **void (*User_SetDeviceAddress)(void)**: Called after receiving the set Address Standard request.

1.3.5 usb_pwr (.c , .h)

This module manages the power management of the USB device. It provides the functions shown in *Table 4*.

Table 4. Power management functions

Function name	Description
`RESULT Power_on(void)`	Handles switch-on conditions
`RESULT Power_off(void)`	Handles switch-off conditions
`void Suspend(void)`	Sets suspend mode operation conditions
`void Resume(RESUME_STATE eResumeSetVal)`	Handles wake-up operations

1.4 Implementing a USB application using the STM32F10xxx USB library

1.4.1 Implementing a no-data class-specific request

All class-specific requests without a data transfer phase implement the field `RESULT (*Class_NoData_Setup)(BYTE RequestNo)` of the structure device property. The `USBbRequest` of the request is available in the `RequestNo` parameter and all other request fields are stored in the device info structure.

The user has to test all request fields. If the request is compliant with the class to implement, the function returns the USB_SUCCESS result. However if there is a problem in the request, the function returns the UNSUPPORT result status and the library responds with a STALL handshake.

1.4.2 How to implement a data class-specific request

In the event of class requests requiring a data transfer phase, the user implementation reports to the USB library the length of the data to transfer and the data location in the internal memory (RAM if the data is received from the host and, RAM or Flash memory if the data is sent to the host). This type of request is managed in the function `RESULT (*Class_Data_Setup)(BYTE RequestNo)`. For each class data request the user has to create a specific function with the format:

```
u8* My_First_Data_Request (u16 Length)
```

If this function is called with the `Length` parameter equal to zero, it sets the `pInformation->Ctrl_Info.Usb_wLength` field with the length of data to transfer and returns a NULL pointer. In other cases it returns the address of the data to transfer. The following C code shows a simple example:

```
u8* My_First_Data_Request (u16 Length)
 {
    if (Length == 0)
    {
       pInformation->Ctrl_Info.Usb_wLength  = My_Data_Length;
       return NULL;
    }
    else
       return (&My_Data_Buffer);
 }
```

The function `RESULT (*Class_Data_Setup)(BYTE RequestNo)` manages all data requests as described in the following C code:

```
RESULT Class_Data_Setup(u8 RequestNo)
{
   u8 *(*CopyRoutine)(u16);
   CopyRoutine = NULL;

   if (My_First_Condition)// test the filds of the first request
      CopyRoutine = My_First_Data_Request;
   else if(My_Second_Condition) // test the filds of the second request
      CopyRoutine = My_Second_Data_Request;
   /*
   ...   same implementation for each class data requests
   ...
   */
   if (CopyRoutine == NULL) return USB_UNSUPPORT;

   pInformation->Ctrl_Info.CopyData = CopyRoutine;
   pInformation->Ctrl_Info.Usb_wOffset = 0;
   (*CopyRoutine)(0);
   return USB_SUCCESS;
} /*End of Class_Data_Setup */
```

1.4.3 How to manage data transfers in non-control endpoint

The management of the data transfer using a pipe that is not the default one (Endpoint 0) can be managed in the *usb_endp.c* file.

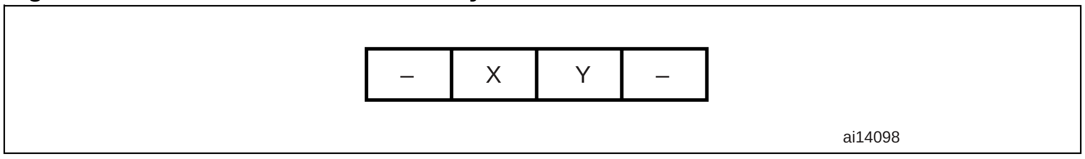
The user has to uncomment the line corresponding to the endpoint (with direction) in the file *usb_conf.h*.

2 Joystick mouse demo

A USB mouse (Human Interface Device class) is a simple example of a complete USB application. The joystick mouse uses only one interrupt endpoint (endpoint 1 in the IN direction). After normal enumeration, the host requests the HID report descriptor of the mouse. This specific descriptor is presented (with standard descriptors) in the *usb_desc.c* file.

To get the mouse pointer position the host requests four bytes of data with the format shown in *Figure 2*, using pipe 1 (endpoint 1).

Figure 2. Format of the four data bytes



The purpose of the mouse demo is to set the X and Y values according to the user actions with a joystick button. The `JoyState()` function gets the user actions and returns the direction of the mouse pointer. The `Joystick_Send()` function formats the data to send to the host and validates the data transaction phase.

Note: See the hw_config.c file for details on the functions.

3 Device firmware upgrade

This part of the document presents the implementation of a device firmware upgrade (DFU) capability in the STM32F10xxx microcontroller. It follows the DFU class specification defined by the USB Implementers Forum for reprogramming an application through USB. The DFU principle is particularly well suited to USB applications that need to be reprogrammed in the field:

The same USB connector can be used for both the standard operating mode and the reprogramming process.

This operation is made possible by the IAP capability featured by most of the STMicroelectronics USB Flash microcontrollers, which allows a Flash MCU to be reprogrammed by any communication channel.

The DFU process, like any other IAP process, is based on the execution of firmware located in one small part of the Flash memory and that manages the erase and program operations of the others Flash memory modules depending on the device capabilities: it could be the main program/Code Flash, data Flash/EEPROM or any other memory connected to the microcontroller even a serial Flash (Through SPI or I^2C etc.). For the STM32F10xxx the DFU Demo is used to program the internal Flash memory and the SPI Flash memory available in the STM3210B-EVAL evaluation board.

Refer to the UM0412, *DfuSe USB device firmware upgrade STMicroelectronics extension*, for more details on the driver installation and PC user interface.

Note: If the internal Flash memory where the user application is to be programmed is write- or/and read-protected, it is required to first disable the protection prior to using the DFU.

3.1 DFU extension protocol

3.1.1 Introduction

The DFU class uses the USB as a communication channel between the microcontroller and the programming tool, generally a PC host. The DFU class specification states that, all the commands, status and data exchanges have to be performed through Control Endpoint 0. The command set, as well as the basic protocol are also defined, but the higher level protocol (Data format, error message etc.) remain vendor-specific. This means that the DFU class does not define the format of the data transferred (.s19, .hex, pure binary etc.).

Because it is impractical for a device to concurrently perform both DFU operations and its normal runtime activities, those normal activities must cease for the duration of the DFU operations. Doing so means that the device must change its operating mode; that is, a printer is **not** a printer while it is undergoing a firmware upgrade; it is a Flash/Memory programmer. However, a device that supports DFU is not capable of changing its mode of operation on its own volition. External (human or host operating system) intervention is required.

3.1.2 Phases

There are four distinct phases required to accomplish a firmware upgrade:

1. Enumeration:

   The device informs the host of its capabilities. A DFU class-interface descriptor and associated functional descriptor embedded within the device's normal run-time descriptors serve this purpose and provide a target for class-specific requests over the control pipe.

2. **DFU Enumeration**

   The host and the device agree to initiate a firmware upgrade. The host issues a USB reset to the device, and the device then exports a second set of descriptors in preparation for the Transfer phase. This deactivates the run-time device drivers associated with the device and allows the DFU driver to reprogram the device's firmware unhindered by any other communications traffic targeting the device.

3. **Transfer**

   The host transfers the firmware image to the device. The parameters specified in the functional descriptor are used to ensure correct block sizes and timing for programming the non-volatile memories. Status requests are employed to maintain synchronization between the host and the device.

4. **Manifestation**

   Once the device reports to the host that it has completed the reprogramming operations, the host issues a USB reset to the device. The device re-enumerates and executes the upgraded firmware.

To ensure that only the DFU driver is loaded, it is considered necessary to change the *id-Product* field of the device when it enumerates the DFU descriptor set. This ensures that the DFU driver will be loaded in cases where the operating system simply matches the vendor ID and product ID to a specific driver.

3.1.3 Requests

A number of DFU class-specific requests are needed to accomplish the upgrade operations. *Table 5* summarizes the DFU class-specific requests.

Table 5. Summary of DFU class-specific requests

bmRequest	bRequest	wValue	wIndex	wLength	Data
00100001b	DFU_DETACH (0)	wTimeout	Interface	Zero	None
00100001b	DFU_DNLOAD (1)	wBlockNum	Interface	Length	Firmware
10100001b	DFU_UPLOAD (2)	wBlockNum	Interface	Length	Firmware
10100001b	DFU_GETSTATUS(3)	Zero	Interface	6	Status
00100001b	DFU_CLRSTATUS (4)	Zero	Interface	Zero	None
10100001b	DFU_GETSTATE (5)	Zero	Interface	1	State
00100001b	DFU_ABORT (6)	Zero	Interface	Zero	None

For additional information about these requests, please refer to the DFU Class specification.

3.2 DFU mode selection

The host should be able to enumerate a device with DFU capability in two ways:

- as a single device with only DFU capability
- as a composite device: HID, Mass storage, or any functional class, and with DFU capability.

During the enumeration phase, the device exposes two distinct and independent descriptor sets, each one at the appropriate time:

- Run-time descriptor set: shown when the device performs normal operations
- DFU mode descriptor set: shown when host and device agree to perform DFU operations

3.2.1 Run-time descriptor set

During normal run-time operation, the device exposes its normal set of descriptors plus two additional descriptors:

- Run-Time DFU Interface descriptor
- Run-Time DFU Functional descriptor

Note: The number of interfaces in each configuration descriptor that supports the DFU must be incremented by one to accommodate the addition of the DFU interface descriptor.

3.2.2 DFU mode descriptor set

After the host and the device agree to perform DFU operations, the host re-enumerates the device. At this time the device exports the descriptor set shown below:

- DFU Mode Device descriptor
- DFU Mode Configuration descriptor
- DFU Mode Interface descriptor
- DFU Mode Functional descriptor: identical to the Run-Time DFU Functional descriptor

DFU mode device descriptor

This descriptor is only present in the DFU mode descriptor set.

Table 6. DFU mode device descriptor

Offset	Field	Size	Value	Description
0	`bLength`	1	12h	Size of this descriptor, in bytes.
1	`bDescriptorType`	1	01h	DEVICE descriptor type.
2	`bcdUSB`	2	0100h	USB specification release number in binary coded decimal.
4	`bDeviceClass`	1	00h	See interface.
5	`bDeviceSubClass`	1	00h	See interface.
6	`bDeviceProtocol`	1	00h	See interface.
7	`bMaxPacketSize0`	1	8,16,32, 64	Maximum packet size for endpoint zero. 8: for ST7 Low Speed 32: 16: 64: For ST7 Full speed, STR7/9 & STM32F10xxx devices
8	`idVendor`	1	0483h	Vendor ID
10	`idProduct`		DF11h	Product ID
12	`bcdDevice`		011Ah	Version of the STMicroelectronics DFU ExtensionSpecification release
14	`iManufacturer`		Index	Index of string descriptor.
15	`iProduct`		Index	Index of string descriptor.
16	`iSerialNumber`		Index	*Index of string descriptor.*
17	`bNumbConfigurations`		01h	One configuration only for DFU.

DFU mode configuration descriptor

This descriptor is identical to the standard configuration descriptor described in the USB specification version 1.0, with the exception that the `bNumInterfaces` field must contain the value 01h.

- DFU Mode Interface Descriptor

This is the descriptor for the only interface available when operating in DFU mode. Therefore, the value of the `bInterfaceNumber` field is always zero.

Table 7. DFU mode interface descriptor

Offset	Field	Size	Value	Description
0	`bLength`	1	09h	Size of this descriptor, in bytes.
1	`bDescriptorType`	1	04h	INTERFACE descriptor type.
2	`bInterfaceNumber`	1	00h	Number of this interface.
3	`bAlternateSetting`	1	Number	Alternate setting
4	`bNumEndpoints`	1	00h	Only the control pipe is used.
5	`bInterfaceClass`	1	FEh	Application Specific Class Code
6	`bInterfaceSubClass`	1	01h	Device Firmware Upgrade Code
7	`bInterfaceProtocol`	1	00h	The device does not use a class-specific protocol on this interface
8	`iInterface`	1	Index	Index of string descriptor for this interface

- Alternate settings and string descriptor definition

This section describes the STMicroelectronics implementation for Alternate settings and the corresponding string descriptor set that is not specified by the standard DFU specification in section 4.2.3.

Alternate settings have to be used to access additional memory segments and other memories (Flash memory, RAM, EEPROM) that may be physically implemented in the CPU memory mapping or not, such as external serial SPI Flash memory or external NOR/NAND Flash memory.

In this case, each alternate setting employs a string descriptor to indicate the target memory segment as shown below:

```
@Target Memory Name/Start Address/Sector(1)_Count*Sector(1)_Size
Sector(1)_Type,Sector(2)_Count*Sector(2)_SizeSector(2)_Type,......,
Sector(n)_Count*Sector(n)_SizeSector(n)_Type
```

Another example, for STM32F10xxx Flash microcontroller, is shown below:

```
@Internal Flash   /0x08000000/12*001Ka,116*001Kg"
```

Each Alternate setting string descriptor must follow this memory mapping else the PC Host Software would be able to decode the right mapping for the selected device:

- @: To detect that this is a special mapping descriptor (to avoid decoding standard descriptor)
- /: for separator between zones
- Maximum 8 digits per address starting by “0x”
- /: for separator between zones
- Maximum of 2 digits for the number of sectors
- `*`: For separator between number of sectors and sector size
- Maximum 3 digits for sector size between 0 and 999
- 1 digit for the sector size multiplier. Valid entries are: B (byte), K (Kilo), M (Mega)
- 1 digit for the sector type as follows:

  a (0x41): Readable

  b (0x42): Erasable

  c (0x43): Readable and Erasable

  d (0x44): Writeable

  e (0x45): Readable and Writeable

  f (0x46): Erasable and Writeable

  g (0x47): Readable, Erasable and Writeable

Note: If the target memory is not contiguous, the user can add the new sectors to be decoded just after a slash"/" as shown in the following example:

```
"@Flash /0xF000/1*4Ka/0xE000/1*4Kg/0x8000/2*24Kg"
```

- DFU Functional descriptor

This descriptor is identical for both the run-time and the DFU mode descriptor sets.

Table 8. DFU functional descriptor

Offset	Field	Size	Value	Description
0	bLength	1	09h	Size of this descriptor, in bytes.
1	bDescriptorType	1	21h	DFU FUNCTIONAL descriptor type.
2	bmAttributes	1	00h	*DFU attributes:* – Bit7: if bit1 is set, the device will have an accelerated upload speed of 4096 byes per upload command (*bitCanAccelerate)* 0: No 1:Yes – Bits 6:4: reserved – Bit 3: device will perform a bus detach-attach sequence when it receives a DFU_DETACH request. 0 = no 1 = yes *Note: The host must not issue a USB Reset. (bitWillDetach)* – Bit 2: device is able to communicate via USB after Manifestation phase (bitManifestation tolerant) 0 = no, must see bus reset 1 = yes – Bit 1: upload capable (bitCanUpload) 0 = no 1 = yes – Bit 0: download capable (bitCanDnload) 0 = no 1 = yes
3	wDetachTimeOut	2	Number	Time, in milliseconds, that the device waits after receipt of the DFU_DETACH request. If this time elapses without a USB reset, then the device terminates the Reconfiguration phase and reverts to normal operation. This represents the maximum time that the device can wait (depending on its timers, etc.). The host may specify a shorter timeout in the DFU_DETACH request.
5	wTransferSize	2	Number	Maximum number of bytes that the device can accept per control-write transaction: wTransferSize depends on the firmware implementation on each MCU.
7	bcdDFUVersion	2	011Ah	Version of the STMicroelectronics DFU ExtensionSpecification release.

3.3 Reconfiguration phase

Once the operator has identified the device and supplied the filename, the host and the device must negotiate to perform the upgrade.

- The host issues a DFU_DETACH request to Control Endpoint EP0.
- The host issues a USB reset to the device. This USB Reset is not possible on some PC Host OS versions. To bypass this issue, the USB reset is performed by the MCU depending on the corresponding implementation.
- The device enumerates with the DFU Mode descriptor set, as described above.

Note: 1 *Some Device application may not be using USB in their run-time mode such as a Motor control application or security system, and USB may be used only for memory upgrade. Those devices are called non-USB application in the scope of this document and the above sequences are not applicable.*

2 *Non-USB applications have to carry out the right procedure to enter the DFU mode. This can be done simply by plugging the USB cable or by jumping to the DFU firmware code while performing an USB reset so that the device would enumerate with the DFU descriptor set.*

3.4 Transfer phase

The transfer phase begins after the device has processed the USB reset and exported the DFU Mode descriptor set. Both downloads and uploads of firmware can take place during this phase. This transfer phase consists of a succession of DFU requests according to the state diagram described in the following sections.

3.4.1 Requests

A number of DFU class-specific requests are needed to accomplish the upgrade/upload operations. *Table 9* summarizes these requests.

Table 9. Summary of DFU upgrade/upload requests

bmRequest	bRequest	wValue	wIndex	wLength	Data
00100001b	DFU_DNLOAD (1)	wBlockNum	Interface	Length	Firmware
10100001b	DFU_UPLOAD (2)	wBlockNum	Interface	Length	Firmware
10100001b	DFU_GETSTATUS(3)	Zero	Interface	6	Status
00100001b	DFU_CLRSTATUS (4)	Zero	Interface	Zero	None
10100001b	DFU_GETSTATE (5)	Zero	Interface	1	State
00100001b	DFU_ABORT (6)	Zero	Interface	Zero	None

For additional information about these requests, please refer to the DFU Class specification.

3.4.2 Special command/protocol descriptions

In order to support all features (Address decoding and Memory block to erase etc.) of the DFU Extension implementation from STMicroelectronics, a few format rules are added to the DFU_DNLOAD request. They are defined as shown in *Table 10*.

Table 10. Special command descriptions

Command	Request	wBlockNum	wLength	Data
Get Commands	DFU_DNLOAD	0	1	**0x00**
Set Address Pointer	DFU_DNLOAD	0	5	**21h**, Address (4bytes)
Erase Sector containing address	DFU_DNLOAD	0	5	**41h**, Address (4bytes)

This new custom DFU implements only three supported basic commands:

- **Get Commands**

  Byte0 = 0x00 then no additional bytes.

  The next DFU_UPLOAD request with wBlockNum = 0 should give the supported commands.

  The maximum size of the supported commands buffer is **256** bytes and the buffer **must** support the following commands:
  - 0x00 (Get Commands)
  - 0x21 (Set Address Pointer)
  - 0x41 (Erase Sector containing address)
- **Set Address Pointer**

  Byte0 = 0x21 then 4 bytes containing the address Pointer from which the Blocks will be downloaded or uploaded starting from the next DFU_DNLOAD or DFU_UPLOAD request with wBlockNum >1.
- **Erase Sector containing address**

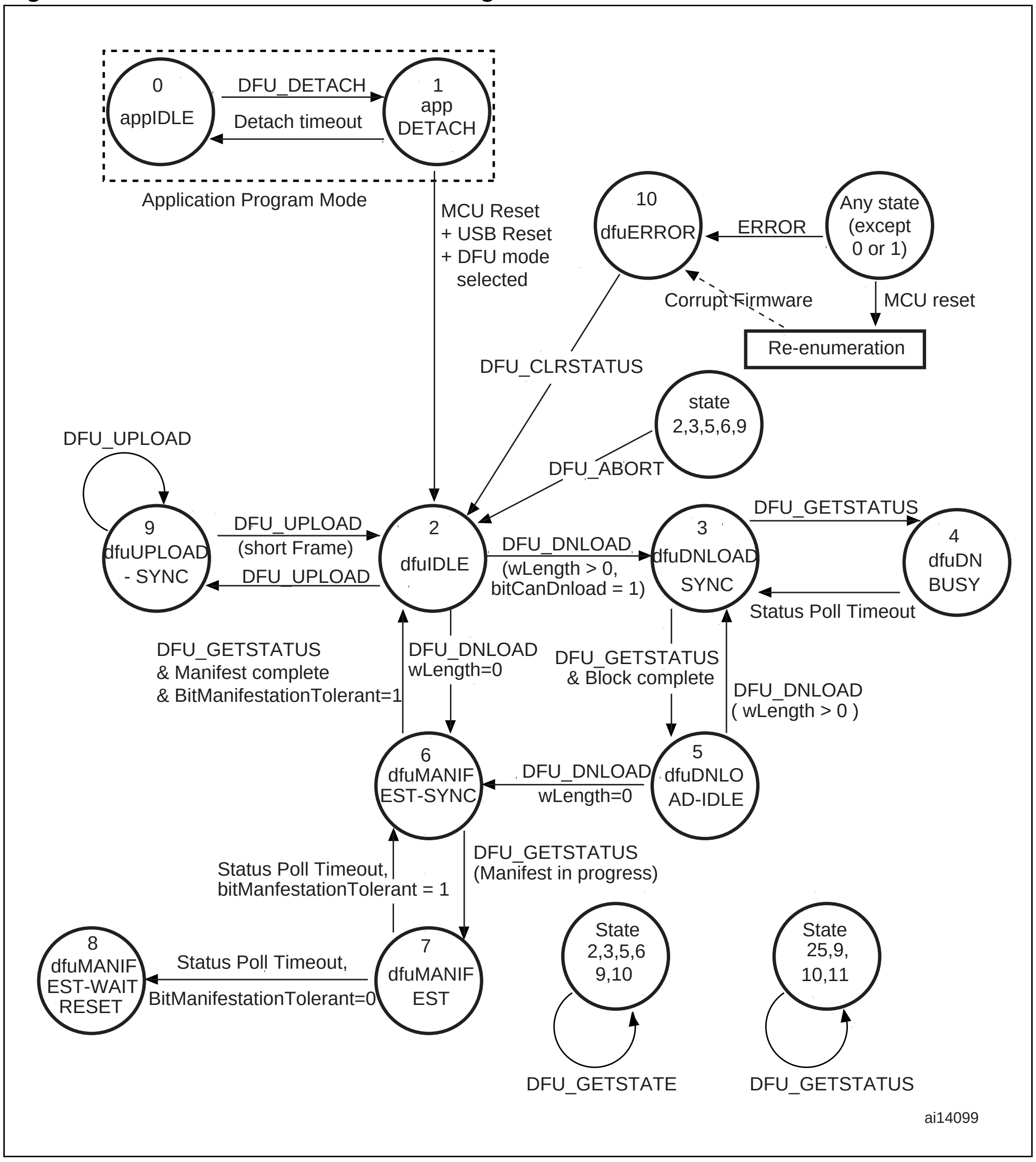
  Byte0 = 0x41 then 4 bytes containing a valid address contained in a memory sector to be erased and as already exported by the string descriptors of the Alternate settings.

Note: wBlockNum = 1 for both DFU_DNLOAD and DFU_UPLOAD requests is reserved for future STMicroelectronics use.

3.4.3 DFU state diagram

Figure 3 summarizes the DFU interface states and the transitions between them. The events that rigger state transitions can be thought of as arriving on multiple “input tapes” as in the classic Turing machine concept.

Figure 3. Interface state transition diagram



Note: *The state transition diagram shown in Figure 3 is almost the same as that defined in the DFU Class specification (Fig A1 page 28), with the exception of the new transition from state 2 to state 6, which is additional and may or not be implemented in the device firmware.*

3.4.4 Downloading and uploading

The host slices the firmware image file into N pieces and sends them to the device by means of control-write operations in the default endpoint (Endpoint 0).

The maximum number of bytes that the device can accept per control-write transaction is specified in the `wTransferSize` field of the DFU Functional Descriptor.

There are several possible download mechanisms depending on the MCU device memory mapping and the Type of the memory (that is Readable, Erasable, Writeable or a combination).

The most generic mechanism is described below, where we have a readable, erasable and writeable sector of memory:

- In addition to the data collected after the enumeration phase about the whole memory mapping, the device capabilities etc., the Host starts to send a GetCommands command in order to know additional device capabilities and which commands are supported by the DFU implementation.
- The host sends an Erase Sector Containing Address command using a DFU_DNLOAD request with `wBlockNum` = 0 and `wLength` = 5. At this stage, the device erases the memory block where the address sent by the host is located. After the erase operation, the DFU firmware is able to write application data into the erased block.
- The host begins by sending the Set Address Pointer command using a DFU_DNLOAD request with `wBlockNum` = 0 and `wLength` = 5. This address pointer is saved in the device RAM as an Absolute Offset.
- The host continues to send the N pieces to the device by means of DFU_DNLOAD requests with `wBlockNum` starting from 2 and with the maximum number of bytes that the device can accept per control-write transaction specified in the `wTransferSize` field of the DFU Functional Descriptor.

  So the last data written into the memory will be located at device address:

  Absolute Offset + (`wBlockNum` – 2) × `wTransferSize` + `wLength`, where `wBlockNum` and `wLength` are the parameters of the last DFU_DNLOAD request.

If the Host wants to upload the memory data for verification, or to retrieve and archive a device firmware, by definition the reverse of a Download is performed:

- The host begins by sending a Set Address Pointer command using a DFU_DNLOAD request with `wBlockNum` = 0 and `wLength` = 5. This address pointer is saved in the device RAM as an Absolute Offset.
- The host continues to send N DFU_UPLOAD requests with `wBlockNum` starting from 2 and with the maximum number of bytes that the device can accept per control-write transaction specified in the `wTransferSize` field of the DFU Functional Descriptor if *bitCanAccelerate = 0.* If *bitCanAccelerate = 1* in the DFU Functional Descriptor, the value in the `wTransferSize` field is fixed to *0x4096 bytes.*

  So the last data retrieved from the memory will be located at device address:

  Absolute Offset + (`wBlockNum` – 2) × *`wTransferSize`* + `wLength`, where `wBlockNum` and `wLength` are the parameters of the last DFU_UPLOAD request.

3.4.5 Manifestation phase

After the transfer phase completes, the device is ready to execute the new firmware. This is achieved by performing a USB reset to re-enumerate the device in normal run-time operation.

3.5 STM32F10xxx DFU implementation

3.5.1 Supported memories

For the STM32F10xxx the DFU implementation supports the following memories:

- **Internal Flash memory**: the first 12 pages are reserved for the DFU (read-only pages) and the other 116 pages can be programmed by the DFU (applicative zone).
- External serial Flash memory (M25P64): consists of 128 sectors of 64 Kbytes each.

Note: 1 To create a DFU image for the internal Flash memory select the Alternate Setting 00 in the DFU file Manager.

2 To create a DFU image for the external serial Flash memory, select the Alternate Setting 01 in the DFU file Manager.

3.5.2 DFU mode entry mechanism

For the STM32F10xxx the DFU mode is entered after an MCU reset if:

- The DFU mode is forced by the user: the user presses the key push-button after a reset.
- There is no correct code available in the applicative area: before jumping to the applicative code, the DFU code tests if there is a correct top-of-stack address in the first address in the applicative area of the internal Flash memory (for the STM32F10xxx the first applicative address is 0x0800 3000). This is done by reading the value of the first applicative address and verifying if the MSB half-word is equal to 0x2000 (base address of the RAM area in the STM32F10xxx).

3.5.3 Available DFU image for the STM32F10xxx

The available DFU images in the STM32F10xxx USB development kit are:

- Joystick Mouse Demo
- Mass Storage Demo
- Virtual COM Demo
- Audio Speaker Demo

4 Mass storage demo

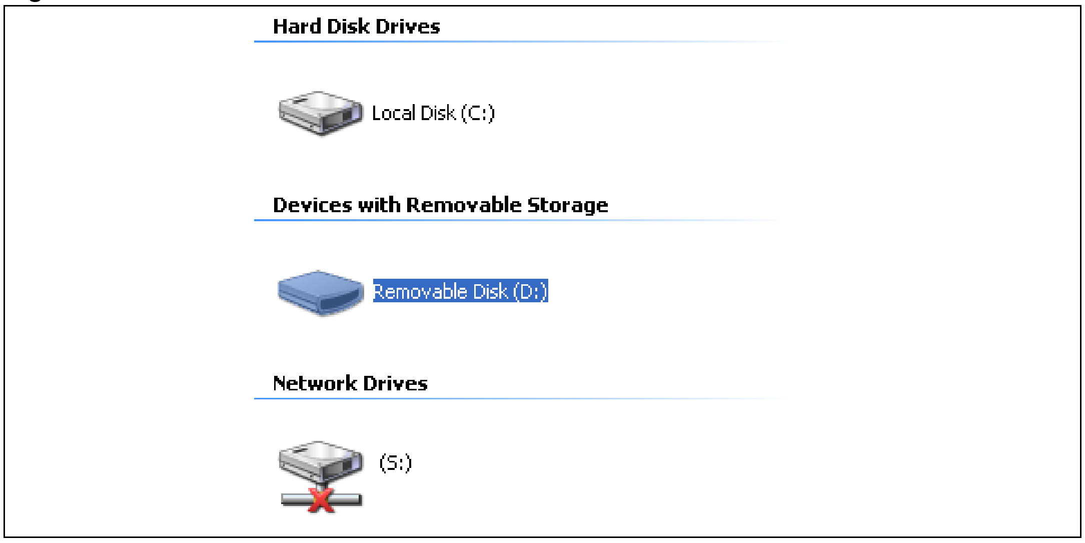
The mass storage demo gives a typical example of how to use the STM32F10xxx USB peripheral to communicate with the PC host using the bulk transfer.

This demo supports the BOT (bulk only transfer) protocol and all needed SCSI (small computer system interface) commands, and is compatible with both Windows XP (SP1/SP2) and Windows 2000 (SP4).

4.1 Mass storage demo overview

The mass storage demo complies with USB 2.0 and USB mass storage class (bulk-only transfer subclass) specifications. After running the application, the user just has to plug the USB cable into a PC Host and the device is automatically detected without any additional drive (with Win 2000 and XP). A new removable drive appears in the system window and write/read/format operations can be performed as with any other removable drive (see *Figure 4*).

Figure 4. New removable disk in Windows



This implementation uses a microSD card as memory support. All related firmware used to initialize, read from and write to the microSD card are available in the “msc.c/.h” files.

Note: For mass storage class, the device firmware does not need to know or take into account the file system the host is using. The firmware just stores and sends blocks of data as requested by the host.

4.2 Mass storage protocol

4.2.1 Bulk-only transfer (BOT)

The BOT protocol uses only bulk pipes to transfer commands, status and data (no interrupt or control pipes). The default pipe (pipe 0, or in other words, Endpoint 0) is only used to clear the bulk pipe status (clear STALL status) and to issue the two class-specific requests: Mass Storage reset and Get Max LUN.

Command transfer

To send a command, the host uses a specific format called command block wrapper (CBW). The CBW is a 31-byte length packet. *Table 11* shows the different fields of a CBW.

Table 11. CBW packet fields

	7	6	5	4	3	2	1	0
0-3	dCBWSignature							
4-7	dCBWTag							
8-11	dCBWDataTransferLength							
12	bmCBWFlags							
13	Reserved (0)				bCBWLUN			
14	Reserved (0)			bCBWCBLength				
15-30	CBWCB							

- ***dCBWSignature***: 43425355 *USBC* (little Endian)
- ***dCBWTag***: The host specifies this field for each command. The device should return the same *dCBWTag* in the associated status.
- ***dCBWDataTransferLength***: total number of bytes to transfer (expected by the host).
- ***bmCBWFlags***: This field is used to specify the direction of the data transfer (if any). The bits of this field are defined as follows:
  - **Bit 7**: Direction bit:
    0: Data Out transfer (host to device).
    1: Data In transfer (device to host).
    Note: The device ignores this bit if the `dCBWDataTransferLength` field is cleared to zero.
  - **Bits 6:0**: reserved (cleared to zero).
- ***bCBWLUN***: concerned Logical Unit number.
- ***bCBWCBLength****:* this field specify the length (in bytes) of the command CBWCB.
- ***CBWCB***: the command block to be executed by the device.

Status transfer

To inform the host about the status of each received command, the device uses the command status wrapper (CSW). *Table 12* shows the different fields of a CSW.

Table 12. CSW packet fields

	7	6	5	4	3	2	1	0
0-3	dCSWSignature							
4-7	dCSWTag							
8-11	dCSWDataResidue							
12	bCSWStatus							

- ***dCSWSignature***: 53425355 *USBS* (little Endian).
- ***dCSWTag***: the device sets this field to the received value of *dCBWTag* in the concerned CBW.
- ***dCSWDataResidue***: the difference between the expected data (the value of the *dCBWDataTransferLength* field of the concerned CBW) and the real value of the data received or sent by the device.
- ***bCSWStatus***: the status of the concerned command. This field can assume one of the three non-reserved values shown in *Table 13*.

Table 13. Command block status values

Value	Description
00h	Command passed
01h	Command failed
02h	Phase error
03h=>FFh	Reserved

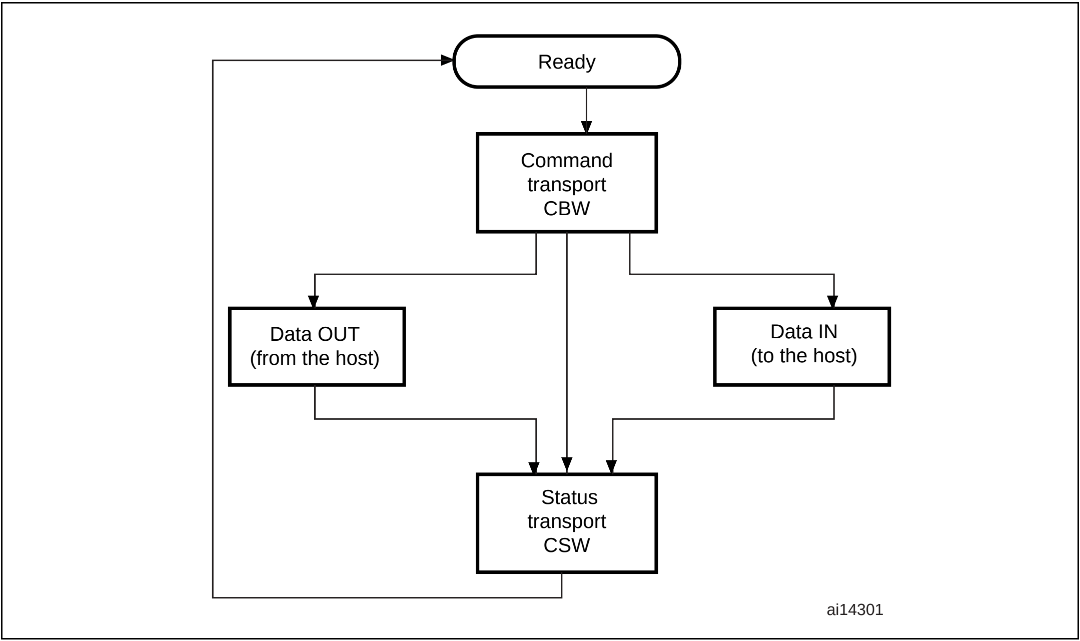
Data transfer

The data transfer phase is specified by the *dCBWDataTransferLength* and *bmCBWFlags* of the correspondent CBW. The host attempts to transfer the exact number of bytes to or from the device.

The diagram shown in *Figure 5* shows the state machine of a BOT transfer.

Note: *For more information about the BOT protocol please refer to the “Universal Serial Bus Mass Storage Class Bulk-Only Transport” specification.*

Figure 5. BOT state machine



4.2.2 Small computer system interface (SCSI)

The SCSI command set is designed to provide efficient peer-to-peer operation of SCSI devices like, for example, hard desks, tapes and mass storage devices. In other words these are used to ensure the communication between an SCSI device and an operating system in a PC host.

Table 14 shows SCSI commands for removable devices. Not all commands are shown. For more information, please refer to the SPC and RBC specifications.

Table 14. SCSI command set

Command name	OpCode	Command support(1)	Description	Reference
Inquiry	0x12	M	Get device information	SPC-2
Read Format Capacities	0x23	M	Report current media capacity and formattable capacities supported by medium	SPC-2
Mode Sense (6)	0x1A	M	Report parameters to the host	SPC-2
Mode Sense (10)		M	Report parameters to the host	SPC-2
Prevent\ Allow Medium Removal	0x1E	M	Prevent or allow the removal of media from a removable media device	SPC-2
Read (10)	0x28	M	Transfer binary data from the medium to the host	RBC
Read Capacity (10)	0x25	M	Report current medium capacity	RBC
Request Sense	0x03	O	Transfer status sense data to the host	SPC-2
Start Stop Unit	0x1B	M	Enable or disable the Logical Unit for medium access operations and controls certain power conditions	RBC
Test Unit Ready	0x00	M	Request the device to report if it is ready	SPC-2
Verify (10)	0x2F	M	Verify data on the medium	RBC
Write (10)	0x2A	M	Transfer binary data from the host to the medium	RBC

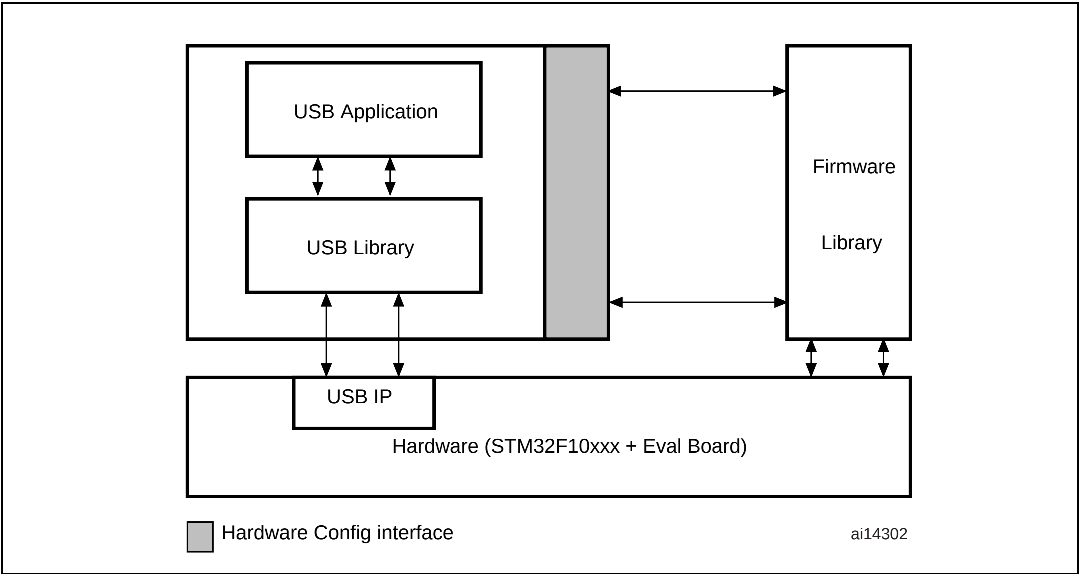
1. Command Support key: M = support is mandatory, O = support is optional.

4.3 Mass storage demo implementations

4.3.1 Hardware configuration interface

The hardware configuration interface is a layer between the USB application (in our case the Mass Storage demo) and the internal/external hardware of the STM32F10xxx microcontroller. This internal and external hardware is managed by the STM32F10xxx Firmware Library, so from the firmware point of view, the hardware configuration interface is the firmware layer between the USB application and the Firmware Library. *Figure 6* shows the interaction between the different firmware components and the hardware environment.

Figure 6. Hardware and firmware interaction diagram


USB Application
USB Library
Firmware
Library
USB IP
Hardware (STM32F10xxx + Eval Board)
Hardware Config interface
ai14302

The hardware configuration layer is represented by the two files *HW_config.c* and *HW_config.h*. For the Mass Storage demo, the hardware management layer manages the following hardware requirements:

- System and USB IP clock configuration
- Read and write LED configuration
- LED command
- Initialize the microSD card
- Get the characteristics of the memory medium (the block size and the memory capacity)

4.3.2 Endpoint configurations and data management

This section provides a description of the configuration and the data flow according to the transfer mode.

Endpoint configurations

The endpoint configurations should be done after each USB reset event, so this part of code is implemented in the `MASS_Reset` function (file *usp_prop.c*).

To configure endpoint 0 it is necessary to:

- Configure Endpoint 0 as the default control endpoint
- Configure the Endpoint 0 Rx and Tx count and buffer addresses in the BTABLE (*usb_conf.h* file)
- Configure the Endpoint Rx status as VALID and the Tx status as NAK.

The configuration of the bulk pipes (endpoints 1 and 2) is done as follows:

- Configure endpoint 1 as bulk IN
- Configure the endpoint 1 Tx count and data buffer address in the BTABLE (*usb_conf.h* file)
- Disable the endpoint 1 Rx
- Configure the endpoint 1 Tx status as NAK
- Configure the endpoint 2 as bulk OUT
- Configure the endpoint 2 Rx count and data buffer address in the BTABLE (*usb_conf.h* file)
- Disable the endpoint 2 Tx
- Configure the endpoint 2 Rx status as VALID.

Data management

Data management consists of the transfer of the needed data directly from the specified data buffer address in the PMA, according to the related endpoint (IN: ENDP1TXADDR; OUT: ENDP2RXADDR). For these transfers, the following two functions are used (*usb_mem.c* file):

- **PMAToUserBufferCopy ()**: this function transfers the specified number of bytes from the Packet Memory Area to the internal RAM. This function is used to copy the data sent by the host to the device.
- **UserToPMABufferCopy ()**: this function transfers the specified number of bytes from the internal RAM to the Packet Memory Area. This function is used to send the data from the device to the host.

4.3.3 Class-specific requests

The Mass Storage Class specification describes two class-specific requests:

Bulk-Only Mass Storage reset

This request is used to reset the Mass Storage device and its associated interface. This class-specific request makes the device ready for the next CBW sent by the PC host.

To issue the Bulk-Only Mass Storage Reset, the host issues a device request on the default pipe (endpoint 0) of:

- *bmRequestType*: Class, Interface, Host to device
- *bRequest* field set to 0xFF
- *wValue* field set to 0
- *wIndex* field set to the interface number (0 for this implementation)
- *wLength* field set to 0

This request is implemented as a no-data class-specific request in the `MASS_NoData_Setup()` function (*usb_prop.c* file).

After receiving this request, the device clears the data toggle of the two bulk endpoints, initializes the CBW signature to the default value and sets the BOT state machine to the BOT_IDLE state to be ready to receive the next CBW.

GET MAX LUN request

A Mass Storage Device may implement several logical units that share common device characteristics. The host uses bCBWLUN to designate 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 issue a Get Max LUN request the host must issue a device request on the default pipe (endpoint 0) of:

- *bmRequestType*: Class, Interface, Host to device
- *bRequest* field set to 0xFE
- *wValue* field set to 0
- *wIndex* field set to the interface number (0 for this implementation)
- *wLength* field set to 1

This request is implemented as a data class-specific request in the `MASS_Data_Setup()` function (*usb_prop.c* file).

Note: *In the two implementations, there is only one LUN so the Get Max LUN is returned directly with the 0x00 Value. In order to implement other LUNs, the user has to return the number of implemented LUNs as a response to the request.*

4.3.4 Standard request requirements

To be compliant with the BOT specification the device must respond to the two following requirements after receiving the same standard requests:

- When the device switches from the unconfigured to the configured state, the data toggle of all endpoints must be cleared. This requirement is served by the `Mass_Storage_SetConfiguration()` function in the *usb_prop.c* file.
- When the host sends a CBW command with an invalid signature or length, the device must keep endpoints 1 and 2 both as STALL until it receives the Mass Storage Reset class-specific request. This functionality is managed by the `Mass_Storage_ClearFeature()` function in the *usb_prop.c* file.

4.3.5 BOT state machine

To provide the BOT protocol, a specific state machine with five states is implemented. The states are described below:

- **BOT_IDLE**: this is the default state after a USB Reset, Bulk-Only Mass Storage Reset or after sending a CSW. In this state the device is ready to receive a new CBW from the host
- **BOT_DATA_OUT**: the device enters this state after receiving a CBW with data flow from the host to the device
- **BOT_DATA_IN**: the device enters this state after receiving a CBW with data flow from the device to the host
- **BOT_DATA_IN_LAST**: the device enters this state when sending the last of the data asked for by the host
- **BOT_CSW_SEND**: the device moves to this state when sending the CSW. When the device is in this state and a correct IN transfer occurs, the device moves to the BOT_IDLE state to be able to receive the next CBW
- **BOT_ERROR:** Error state

The BOT state machine is managed using the functions described below (*usb_bot.c* and *usb_bot.h* firmware files):

- **Mass_Storage_In ()**; **Mass_Storage_Out ()**: these two functions are called when a correct transfer (IN or OUT) occurs. The aim of these two functions is to provide the next step after receiving/sending a CBW, data or CSW
- **CBW_Decode ()**: this function is used to decode the CBW and to dispatch the firmware to the corresponding SCSI command
- **DataInTransfer ()**: this function is used to transfer the characteristic device data to the host
- **Set_CSW ()**: this function is used to set the CSW fields with the needed parameters according to the command execution
- **Bot_Abort ()**: this function is used to STALL the endpoints 1 or 2 (or both) according to the Error occurring in the BOT flow

4.3.6 SCSI protocol implementation

The aim of the SCSI Protocol is to provide a correct response to all SCSI commands needed by the operating system on the PC host. This section details the method of management for all implemented SCSI commands.

- **INQUIRY** command (OpCode = 0x12):

  Send the needed inquiry page data (in this demo only page 0 and the standard page are supported) with the needed data length according to the *ALLOCATION LENGTH* field of the command.
- **SCSI READ FORMAT CAPACITIES** command (OpCode = 0x23):

  Send the Read Format Capacity data response (`ReadFormatCapacity_Data[ ]` from the *SCSI_data.c* file).
- **SCSI READ CAPACITY (10)** command (OpCode = 0x25):

  Send the Read Capacity (10) data response (`ReadCapacity10_Data[ ]` from the *SCSI_data.c* file).
- **SCSI MODE SENSE (6)** command (OpCode = 0x1A):

  Send the Mode Sense (6) data response (`Mode_Sense6_data[ ]` from the *SCSI_data.c* file).
- **SCSI MODE SENSE (10)** command (OpCode = 0x5A):

  Send the Mode Sense (10) data response (`Mode_Sense10_data[ ]` from the *SCSI_data.c* file).
- **SCSI REQUEST SENSE** command (OpCode = 0x03):

  Send the Request Sense data response. Note that the `Resquest_Sense_Data [ ]` array (*SCSI_data.c* file) is updated using the `Set_Scsi_Sense_Data()` function in order to set the *Sense key* and the *ASC* fields according to any error occurring during the transfer.
- **SCSI TEST UNIT READY** command (OpCode = 0x00):

  Always return a CSW with COMMAND PASSED status.
- **SCSI PREVENT\ALLOW MEDIUM REMOVAL** command (OpCode = 0x1E):

  Always return a CSW with COMMAND PASSED status.
- **SCSI START STOP UNIT** command (OpCode = 0x1B):

  This command is sent by the PC host when a user right-clicks on the device (in Windows) and selects the Eject operation. In this case the firmware programs the data in the internal Flash memory using the `Stor_Data_In_Flash()` function.
- **SCSI READ 10** command (OpCode = 0x28) and **SCSI WRITE 10** command (OpCode = 0x2A):

  The host issues these two commands to perform a read or a write operation. In these cases the device has to verify the address compatibility with the memory range and the direction bit in the bmFlag of the command. If the command is validated the firmware launches the read or write operation from the microSD card.
- **SCSI VERIFY 10** command (OpCode =0x2F):

  The SCSI VERIFY 10 command requests the device to verify the data written on the medium. In this case no Flash-like memory support is used, so when the SCSI VERIFY 10 command is received, the device tests the BLKVFY bit. If the BLKVFY bit is set to one, a Command Passed status is returned in the CSW.

4.3.7 Memory management

All the memory management functions are grouped in the two files: *memory.c* and *memory.h.* Memory management consists of two basic processes:

- Management and validation of the address range for the SCSI READ (10) and SCSI WRITE (10) commands: this process is done by the `Address_Management_Test()` function. The role of this function is to extract the real address and memory offset in the microSD card and test if the current transfer (Read or Write) is in the memory range. If this is not the case, the function STALLs endpoint 1 or both endpoints (according to the transfer Read or Write) and returns a bad status to disable the transfer.
- Management of the Read and Write processes: this process is done by the two functions `Read_Memory()` and `Write_Memory()`. These two functions manage the microSD card access based on the two functions “MSD_WriteBlock” and “MSD_ReadBlock” from the *msc.c* file. After each access, the current memory offset and the next Access Address are updated using the length of the previous transfer.

4.4 How to customize the mass storage demo

The implemented firmware is a simple example used to demonstrate the STM32F10xxx USB peripheral capability in bulk transfer. However it can be customized according to user requirements. This customizing can be done in the three layers of the implemented mass storage protocol:

- **Customizing of the BOT layer:** the user can implement their own BOT state machine or modify the implemented one just by modifying the two files *usb_BOT.c* and *usb_BOT.h* and by keeping the same data transfer method.
- **Customizing of the SCSI layer:** the implemented SCSI protocol presents, more than the supported command listed in *Section 4.3.6: SCSI protocol implementation*, a list of unsupported commands. When the host sends one of these commands, a corresponding function is called by the `CBW_Decode()` function like a common command. However, all the functions related to unsupported commands are defined by the `SCSI_Invalid_Cmd()` function, (see *usb_scsi.c* file). The `SCSI_Invalid_Cmd()` function STALLs the two endpoints (1 and 2), sets the Sense data to *invalid command key* and sends a CSW with a *Command Failed* status.
  To support one of the invalid commands, the user has to comment-out the concerned line and implement their own process. For example, for the need to support the `SCSI_FormatUnit` command, comment the line:
  ```
  // #define SCSI_FormatUnit_Cmd SCSI_Invalid_Cmd
  ```
  And implement a process in a function with the same name in the *usb_scsi.c* file:
  ```
  void SCSI_Invalid_Cmd (void)
  {
  // your implementation
  }
  ```
  In this way the custom function is called automatically by the `CBW_Decode()` function (*usb_BOT.c* file).

  However if a user needs to implement a command not listed in the previous list they have to modify the `CBW_Decode()` and implement the protocol of the new command.

Mass storage descriptors

Table 15. Device descriptor

Field	Value	Description
bLength	0x12	Size of this descriptor in bytes
bDescriptortype	0x01	Descriptor type (device descriptor)
bcdUSB	0x0200	USB specification release number: 2.0
bDeviceClass	0x00	Device Class
bDeviceSubClass	0x00	Device subclass
bDeviceProtocol	0x00	Device protocol
bMaxPacketSize0	0x40	Max Packet Size of Endpoint 0: 64 bytes
idVendor	0x0483	Vendor identifier (STMicroelectronics)
idProduct	0x5720	Product identifier
bcdDevice	0x0100	Device release number: 1.00
iManufacturer	4	Index of the manufacturer String descriptor: 4
iProduct	42	Index of the product String descriptor: 42
iSerialNumber	96	Index of the serial number String descriptor
bNumConfigurations	0x01	Number of possible configurations: 1

Table 16. Configuration descriptor

Field	Value	Description
bLength	0x09	Size of this descriptor in bytes
bDescriptortype	0x02	Descriptor type (configuration descriptor)
wTotalLength	32	Total length (in bytes) of the returned data by this descriptor (including interface endpoint descriptors)
bNumInterfaces	0x0001	Number of interfaces supported by this configuration (only one interface)
bConfigurationValue	0x01	Configuration value
iConfiguration	0x00	Index of the Configuration String descriptor
bmAttributes	0x80	Configuration characteristics: Bus powered
Maxpower	0x32	Maximum power consumption through USB bus: 100 mA

Table 17. Interface Descriptors

Field	Value	Description
bLength	0x09	Size of this descriptor in bytes
bDescriptortype	0x04	Descriptor type (Interface descriptor)
bInterfaceNumber	0x00	Interface number
bAlternateSetting	0x00	Alternate Setting number
bNumEndpoints	0x02	Number of used Endpoints: 2
bInterfaceClass	0x08	Interface class: Mass Storage class
bInterfaceSubClass	0x06	Interface subclass: SCSI transparent
bInterfaceProtocl	0x50	Interface protocol: 0x50
iInterface	106	Index of the interface String descriptor

Table 18. Endpoint descriptors

Field	Value	Description
IN ENDPOINT		
bLength	0x07	Size of this descriptor in bytes
bDescriptortype	0x05	Descriptor type (Endpoint descriptor)
bEndpointAddress	0x81	IN Endpoint address 1.
bmAttributes	0x02	Bulk Endpoint
wMaxPacketSize	0x40	64 bytes
bInterval	0x00	Does not apply for bulk Endpoints
OUT ENDPOINT		
bLength	0x07	Size of this descriptor in bytes
bDescriptortype	0x05	Descriptor type (Endpoint descriptor)
bEndpointAddress	0x02	Out Endpoint address 2
bmAttributes	0x02	Bulk Endpoint
wMaxPacketSize	0x40	64 bytes
bInterval	0x00	Does not apply for bulk Endpoints

5 Virtual COM port demo

In modern PCs, USB is the standard communication port for almost all peripherals. However many industrial software applications still use the classic COM Port (UART). The Virtual COM Port Demo provides a simple solution to bypass this problem. It uses the USB as a COM port by affecting the legacy PC application designed for COM Port communication.

The Virtual COM Port demo provides the firmware examples for the STM32F10xxx family and the PC driver. This section provides a brief description of the implementation, and shows how to run the demo.

5.1 Virtual COM port demo proposal

The demo proposal is to use the STM3210B-EVAL evaluation board as a USB-to-USART bridge and to provide communication between a laptop (without RS-232 Port) and a normal PC as shown in *Figure 7*. The PC application used in the communication is Windows HyperTerminal. See *Figure 8*.

Figure 7. Virtual COM Port demo as USB-to-USART bridge.

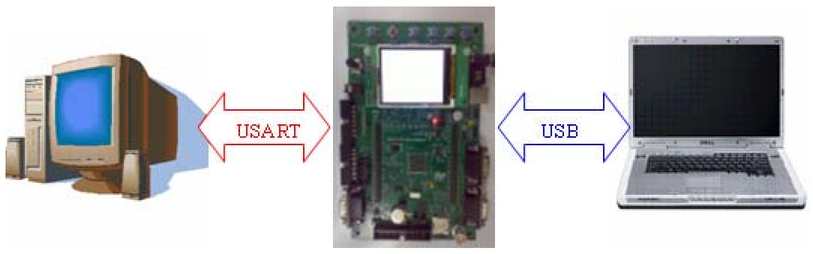
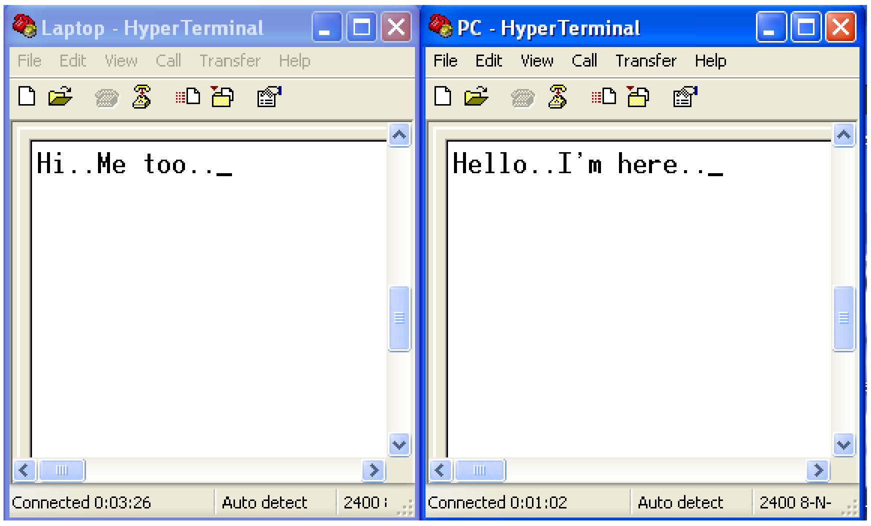


Figure 8. Communication example


Laptop - HyperTerminal
File Edit View Call Transfer Help
PC - HyperTerminal
File Edit View Call Transfer Help

Hi..Me too.._

Connected 0:03:26 Auto detect 2400 8

Hello..I'm here.._

Connected 0:01:02 Auto detect 2400 8-N-

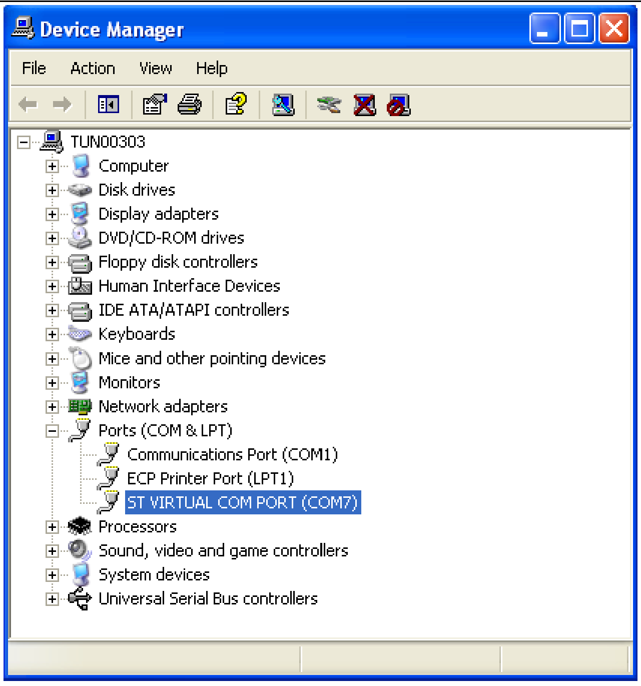
5.2 Software driver installation

To install the software driver of the Virtual COM port, follow the following steps:

- Load the application and run it on the STM3210B-EVAL evaluation board
- Plug the USB cable into the PC
- Indicate to the PC the location of the *stmcdc.inf* file (already provided in the Kit)

At the end of the installation a new COM port appears in the Device Manager window as shown in *Figure 9*.

Figure 9. Device Manager window



```
TUN00303
  Computer
  Disk drives
  Display adapters
  DVD/CD-ROM drives
  Floppy disk controllers
  Human Interface Devices
  IDE ATA/ATAPI controllers
  Keyboards
  Mice and other pointing devices
  Monitors
  Network adapters
  Ports (COM & LPT)
    Communications Port (COM1)
    ECP Printer Port (LPT1)
    ST VIRTUAL COM PORT (COM7)
  Processors
  Sound, video and game controllers
  System devices
  Universal Serial Bus controllers
```

5.3 Implementation

5.3.1 Hardware implementation

The Virtual COM port demo uses the USART 1 present in the STM3210B-EVAL board. There is no need to add external hardware to run the demo.

5.3.2 Firmware implementation

In order to be considered a COM port, the USB device has to implement two interfaces according to the Communication Device Class (CDC) specification:

- Abstract Control Model Communication, with 1 Interrupt IN endpoint: in our implementation this interface is declared in the descriptor but the related endpoint (endpoint 2) is not used
- Abstract Control Model Data, with 1 Bulk IN and 1 Bulk OUT endpoint: this interface is represented in the demo by endpoint 1 (IN) and endpoint 3 (OUT). Endpoint 1 is used to send the data received from USART 1 to the PC truth USB. Endpoint 3 is used to receive the data from the PC and send it through the UART.

For more information on the CDC class please refer to the *Universal Serial Bus Class Definitions for Communication Devices* specification provided by the www.usb.org website.

Class-specific requests

To implement a virtual COM port, the device supports the following class-specific requests:

- **SET_CONTROL_LINE_STATE**: RS-232 signal used to tell the device that the Date Terminal Equipment device is now present. This request always returns a USB_SUCCESS status in the `Virtual_Com_Port_NoData_Setup()` function (*usb_porp.c* file).
- **SET_COMM_FEATURE**: controls the settings for a particular communication feature. This request always returns a USB_SUCCESS status in the `Virtual_Com_Port_NoData_Setup()` function (*usb_porp.c* file).
- **SET_LINE_CODING**: sends the configuration of the device. It includes the baud rate, stop-bits, parity, and number-of-character bits. The received data is stored in a specific data structure called “linecoding” and used to update the UART 0 parameters.
- **GET_LINE_CODING**: This command requests the device current baud rate, stop-bits, parity, and number-of-character bits. The device responds to this request with the data stored in the “linecoding” structure.

Hardware configuration interface

The hardware configuration interface (*hw_config.c* and *.h*) in the Virtual COM port manages the following routines:

- Configure the system and IPs (USB & USART1) clock and interruption
- Initialize USART 1 to default values
- Configure USART 1 with the parameters received by the SET_LINE_CODING request
- Send the data received by the USART 1 to the PC through PC
- Send the data received by the USB through USART 1

Note: *For the* STM32F10xxx *the supported data formats are 7 & 8 bits (in the* HyperTerminal*) and the bandwidth range is from 1200 to 115200.*

6 USB voice demo

The USB voice demos give examples of how to use the STM32F10xxx USB peripheral to communicate with the PC host in the isochronous transfer mode. They provides a demonstration of the correct method for configuring an isochronous endpoint, receiving or transmitting data from/to the host. They also show how to use the data in a real-time application.

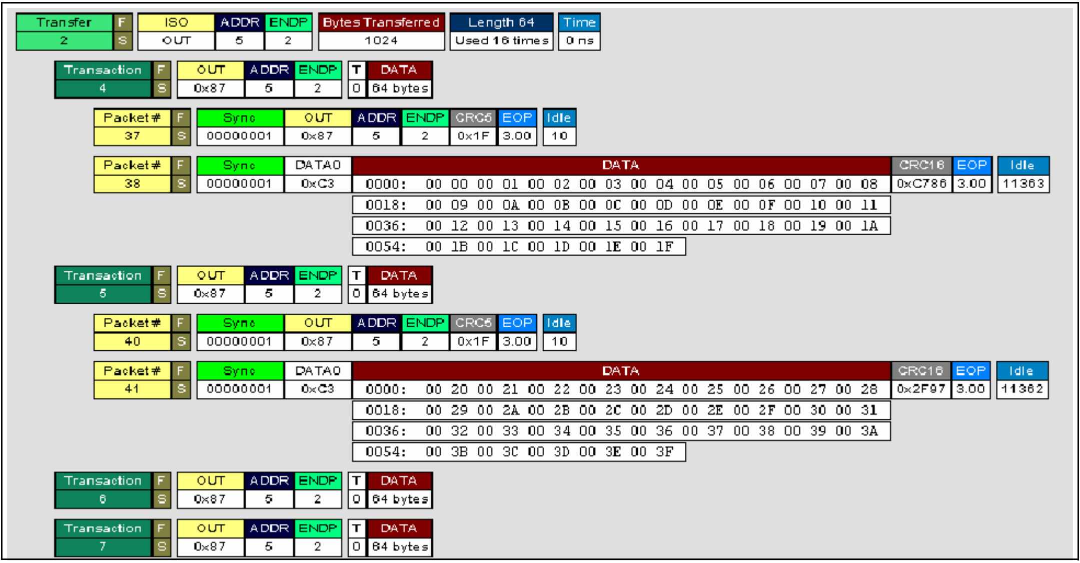
The available voice demo described in this user guide is a USB speaker.

6.1 Isochronous transfer overview

The isochronous transfer is used when the application needs to guarantee the access to the USB bandwidth with bounded latency, constant data rate and without attempting a new data transfer operation in case of failure.

In fact, an isochronous transaction does not have a handshake phase and no ACK packet is expected or sent after the data packet. *Figure 10* shows an example of an isochronous OUT transfer with 64 bytes in the data packet.

Figure 10. Isochronous OUT transfer



```
Transfer F | ISO  ADDR ENDP | Bytes Transferred | Length 64       | Time
   2     S | OUT   5    2   |      1024         | Used 16 times   | 0 ns

  Transaction F | OUT  ADDR ENDP | T | DATA
       4      S | 0x87  5    2   | 0 | 64 bytes

    Packet# F | Sync      OUT   ADDR ENDP CRC5 EOP  | Idle
      37    S | 00000001  0x87   5    2   0x1F 3.00 | 10

    Packet# F | Sync      DATA0 | DATA                                                      | CRC16  EOP  | Idle
      38    S | 00000001  0xC3  | 0000:  00 00 00 01 00 02 00 03 00 04 00 05 00 06 00 07 00 08 | 0xC786 3.00 | 11363
                                | 0018:  00 09 00 0A 00 0B 00 0C 00 0D 00 0E 00 0F 00 10 00 11
                                | 0036:  00 12 00 13 00 14 00 15 00 16 00 17 00 18 00 19 00 1A
                                | 0054:  00 1B 00 1C 00 1D 00 1E 00 1F

  Transaction F | OUT  ADDR ENDP | T | DATA
       5      S | 0x87  5    2   | 0 | 64 bytes

    Packet# F | Sync      OUT   ADDR ENDP CRC5 EOP  | Idle
      40    S | 00000001  0x87   5    2   0x1F 3.00 | 10

    Packet# F | Sync      DATA0 | DATA                                                      | CRC16  EOP  | Idle
      41    S | 00000001  0xC3  | 0000:  00 20 00 21 00 22 00 23 00 24 00 25 00 26 00 27 00 28 | 0x2F97 3.00 | 11362
                                | 0018:  00 29 00 2A 00 2B 00 2C 00 2D 00 2E 00 2F 00 30 00 31
                                | 0036:  00 32 00 33 00 34 00 35 00 36 00 37 00 38 00 39 00 3A
                                | 0054:  00 3B 00 3C 00 3D 00 3E 00 3F

  Transaction F | OUT  ADDR ENDP | T | DATA
       6      S | 0x87  5    2   | 0 | 64 bytes

  Transaction F | OUT  ADDR ENDP | T | DATA
       7      S | 0x87  5    2   | 0 | 64 bytes
```

Typical examples of application use of the isochronous transfer mode are audio samples, compressed video streams and, in general, any sort of sampled data with strict requirements for the accuracy of the delivered frequency.

Please see the USB 2.0 specifications for more details on the USB isochronous transfer mode characteristics.

6.2 Audio device class overview

An audio device, as defined by the Universal Serial Bus Class Definition for Audio Devices specification, is a device or a function embedded in composite devices that are used to manipulate audio, voice, and sound-related functionality. This includes both audio data (analog and digital) and the functionality that is used to directly control the audio environment, such as *volume* and *tone control*.

All audio devices are grouped, from the USB point of view, in the audio interface class. This class is divided into several subclasses. The Universal Serial Bus Class Definition for Audio Devices specification details the three following subclasses:

- **AudioControl Interface subclass (AC)**: each audio function has a single AudioControl interface. The AC interface is used to control the functional behavior of a particular audio function. To achieve this functionality, this interface can use the following endpoints:
  - A control endpoint (endpoint 0) for manipulating unit and terminal settings and retrieving the state of the audio function using class-specific requests.
  - An interrupt endpoint for status returns. This endpoint is optional.

  The AudioControl interface is the single entry point to access the internals of the audio function. All requests that are concerned with the manipulation of certain audio controls within the audio function's units or terminals must be directed to the AudioControl interface of the audio function. Likewise, all descriptors related to the internals of the audio function are part of the class-specific AudioControl interface descriptor.

  The AudioControl interface of an audio function may support multiple alternate settings. Alternate settings of the AudioControl interface could for instance be used to implement audio functions that support multiple topologies by presenting different class-specific AudioControl interface descriptors for each alternate setting.
- **AudioStreaming Interface Subclass (AS):** AudioStreaming interfaces are used to interchange digital audio data streams between the host and the audio function. They are optional. An audio function can have zero or more AudioStreaming interfaces associated with it, each possibly carrying data of a different nature and format. Each AudioStreaming interface can have at most one isochronous data endpoint.
- **MIDIStreaming Interface Subclass (MIDIS)**: MIDIStreaming interfaces are used to transport MIDI data streams into and out of the audio function.

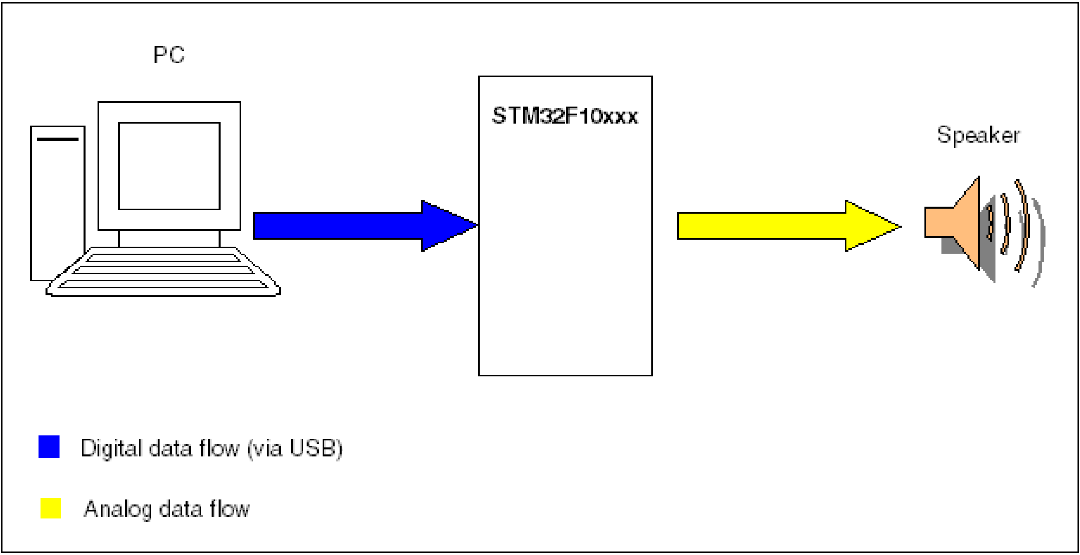
  To be able to manipulate the physical properties of an audio function, its functionality must be divided into addressable entities. Two types of such generic entities are identified and are called *units* and *terminals*. The Universal Serial Bus Class Definition for Audio Devices specification defines seven types of standard units and terminals that are considered adequate to represent most audio functions. These are:
  - Input Terminal
  - Output Terminal
  - Mixer Unit
  - Selector Unit
  - Feature Unit
  - Processing Unit
  - Extension Unit.

For more information about the audio class characteristics and requirements please refer to the *Universal Serial Bus Device Class Definition for Audio Devices specification* provided by the usb.org website.

6.3 STM32F10xxx USB audio speaker demo

The purpose of the USB audio speaker demo is to receive the audio Stream (data) from a PC host using the USB and to play it back via the STM32F10xxx MCU. *Figure 11: STM32F10xxx USB audio speaker demo data flow* represents the data flow between the PC host and the audio speaker.

Figure 11. STM32F10xxx USB audio speaker demo data flow

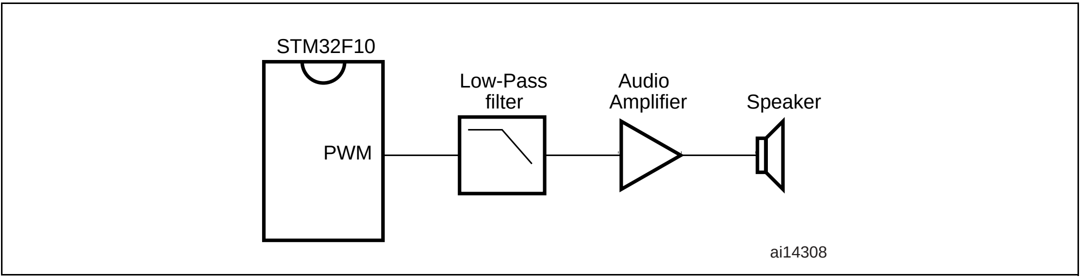


6.3.1 General characteristics

- USB characteristics:
  - Endpoint 0: used to enumerate the device and to respond to class-specific requests. The maximum packet size of this endpoint is 64 bytes.
  - Endpoint 1 (OUT): used to receive the audio stream from the PC host with a maximum packet size up to 22 bytes.
- Audio characteristics:
  - Audio data format: Type I / PCM8 format / Mono.
  - Audio data resolution: 8 bits.
  - Sample frequency: 22 kHz.
- Hardware requirements:

  As the STM32F10xxx MCU does not have an on-chip DAC to generate the analog data flow, an alternate method is used to implement 1 channel DAC. This method consists in using the build-in pulse width modulation (PWM) module to generate a signal whose pulse width is proportional to the amplitude of the sample data. The PWM output signal is then integrated by a low-pass filter to remove high-frequency components, leaving only the low-frequency content. The output of the low-pass filter provides a reasonable reproduction of the original analog signal. *Figure 12* shows the Audio playback diagram flow using the built-in PWM.

Figure 12. Audio playback flow



6.3.2 Implementation

This section describes the hardware and software solution used to implement a USB audio speaker using the STM32F10xxx microcontroller.

Hardware implementation

To implement the PWM feature the following STM32F10xxx built-in timers are used:

- TIM2 in output compare timing mode to act as SysTimer.
- TIM4 in PWM mode

Firmware implementation

The aim of the STM32F10xxx speaker demo is to store the data (Audio Stream) received from the host in a specific buffer called *Stream_Buffer* and to use the PWM to play one stream (8-bit format) every 45.45 µs (~ 22 kHz).

a) Hardware configuration interface:

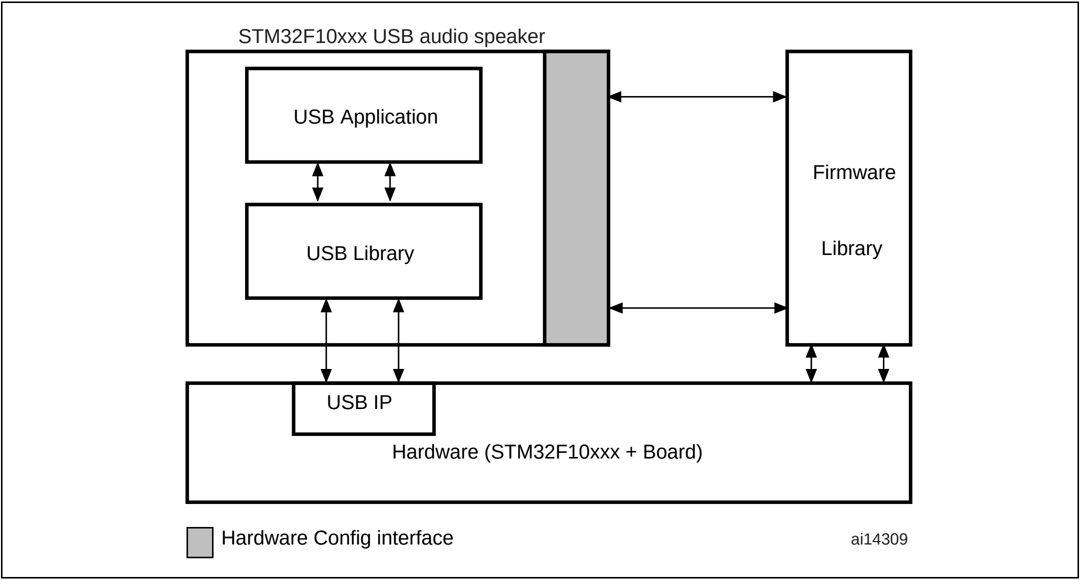
The hardware configuration interface is a layer between the USB application (in our case the USB Audio Speaker) and the internal/external hardware of the STM32F10xxx microcontroller. This internal and external hardware is managed by the STM32F10xxx Firmware library, so from the firmware point of view, the hardware configuration interface is the firmware layer between the USB application and the Firmware library. *Figure 13* shows the interaction between the different firmware components and the hardware environment.

The Hardware configuration layer is represented by the two files *hw_config.c* and *hw_config.h*. For the USB audio speaker demo, the hardware management layer manages the following hardware requirements:

– System and USB IP clock configuration

– Timer configuration

Figure 13. Hardware and firmware interaction diagram


STM32F10xxx USB audio speaker
USB Application
USB Library
Firmware
Library
USB IP
Hardware (STM32F10xxx + Board)
Hardware Config interface
ai14309

b) Endpoints Configurations:

In the STM32F10xxx USB speaker demo, two endpoints are used to communicate with the PC host: endpoint 0 and endpoint 1. Note that endpoint 1 is an Isochronous OUT endpoint and this kind of endpoint is managed by the STM32F10xxx USB IP using the double buffer mode so the firmware has to provide two data buffers in the Packet Memory Area for this endpoint. The following C code describes the method used to configure an isochronous OUT endpoint (see the *usb_prop.c* file, `Speaker_Reset ()` function).

```
/* Initialize Endpoint 1 */
        SetEPType(ENDP1, EP_ISOCHRONOUS);
        SetEPDblBuffAddr(ENDP1,ENDP1_BUF0Addr,ENDP1_BUF1Addr);
        SetEPDblBuffCount(ENDP1, EP_DBUF_OUT, 22);
        ClearDTOG_RX(ENDP1);
        ClearDTOG_TX(ENDP1);
        ToggleDTOG_TX(ENDP1);
        SetEPRxStatus(ENDP1, EP_RX_VALID);
        SetEPTxStatus(ENDP1, EP_TX_DIS);
```

c) Class-specific request

This implementation supports only Mute control. This feature is managed by the `Mute_command` function (*usb_prop.c* file).

d) Isochronous data transfer management

As detailed before, the STM32F10xxx manages the isochronous data transfer using the double buffer mode. So to copy the received data from the PMA to the *Stream_Buffer*, the swapping between the two PMA buffers (ENDP1_BUF0Addr and ENDP1_BUF1Addr) has to be managed. Swapping access to the PMA is managed according to the buffer usage between the USB IP and the firmware. This operation is provided by the `EP1_OUT_Callback ()` function (*usb_endp.c* file). After the end of the copy process, a global variable called *IN_Data_Offset* is updated by the number of bytes received and copied in the Stream_Buffer.

e) Audio Playing Implementation:

To play back the audio samples received from the host, Timer TIM4 is programmed to generate a 125.5 kHz PWM signal and the TIM2 is programmed to generate an interrupt at a frequency equal to 22 kHz. On each TIM2 interrupt one Audio Stream is used to update the pulse of the PWM. A global variable (*Out_Data_Offset*) is used to point to the next Stream to play in Stream buffer.

Note: Note that both “IN_Data_Offset” and “Out_Data_Offset” are initialized to 0 in each Start of frame interrupt (see usb_istr.c file, SOF_Callback() function) to avoid the overflow of the “Stream_Buffer”.

Audio speaker descriptors

Table 19. Device descriptors

Field	Value	Description
bLength	0x12	Size of this descriptor in bytes
bDescriptortype	0x01	Descriptor type (Device descriptor)
bcdUSB	0x0200	USB specification Release number: 2.0
bDeviceClass	0x00	Device class
bDeviceSubClass	0x00	Device subclass
bDeviceProtocol	0x00	Device protocol
bMaxPacketSize0	0x40	Max packet size of Endpoint 0: 64 bytes;
idVendor	0x0483	Vendor identifier (STMicroelectronics)
idProduct	0x5730	Product identifier
bcdDevice	0x0100	Device release number: 1.00
iManufacturer	0x01	Index of the manufacturer string descriptor: 1
iProduct	0x02	Index of the product string descriptor: 2
iSerialNumber	0x03	Index of the serial number string descriptor: 3
bNumConfigurations	0x01	Number of possible configurations: 1

Table 20. Configuration descriptors

Field	Value	Description
bLength	0x09	Size of this descriptor in bytes
bDescriptortype	0x02	Descriptor type (Configuration descriptor)
wTotalLength	0x6D	Total length (in bytes) of the returned data by this descriptor (including interface endpoint descriptors)
bNumInterfaces	0x0002	Number of interfaces supported by this configuration (two interfaces)
bConfigurationValue	0x01	Configuration value
iConfiguration	0x00	Index of the Configuration String descriptor
bmAttributes	0x80	Configuration characteristics: Bus powered
Maxpower	0x32	Maximum power consumption through USB bus: 100 mA

Table 21. Interface descriptors

Field	Value	Description
USB speaker standard interface AC descriptor (Interface 0, Alternate Setting 0)		
bLength	0x09	Size of this descriptor in bytes
bDescriptortype	0x04	Descriptor type: Interface descriptor
bInterfaceNumber	0x00	Interface number
bAlternateSetting	0x00	Alternate setting number
bNumEndpoints	0x00	Number of used endpoints: 0 (only endpoint 0 is used for this interface)
bInterfaceClass	0x01	Interface class: USB DEVICE CLASS AUDIO
bInterfaceSubClass	0x01	Interface subclass: AUDIO SUBCLASS AUDIOCONTROL
bInterfaceProtocol	0x00	Interface protocol: AUDIO PROTOCOL UNDEFINED
iInterface	0x00	Index of the interface string descriptor
USB speaker class-specific AC interface descriptor		
bLength	0x09	Size of this descriptor in bytes
bDescriptortype	0x24	Descriptor type: AUDIO INTERFACE DESCRIPTOR TYPE
bDescriptorSubtype	0x01	Descriptor Subtype: AUDIO CONTROL HEADER
bcdADC	0x0100	bcdADC:1.00
wTotalLength	0x0027	Total Length: 39
bInCollection	0x01	Number of streaming interfaces: 1
baInterfaceNr	0x01	baInterfaceNr: 1
USB speaker input terminal descriptor		
bLength	0x0C	Size of this descriptor in bytes: 12
bDescriptortype	0x24	Descriptor type: AUDIO INTERFACE DESCRIPTOR TYPE
bDescriptorSubtype	0x02	Descriptor Subtype: AUDIO CONTROL INPUT TERMINAL
bTerminalID	0x01	Terminal ID: 1
wTerminalType	0x0101	Terminal Type: AUDIO TERMINAL USB STREAMING
bAssocTerminal	0x00	No association
bNrChannels	0x01	One channel
wChannelConfig	0x0000	Channel Configuration: MONO
iChannelNames	0x00	Unused
iTerminal	0x00	Unused
USB speaker audio feature unit descriptor		
bLength	0x09	Size of this descriptor in bytes
bDescriptortype	0x24	Descriptor type: AUDIO INTERFACE DESCRIPTOR TYPE
bDescriptorSubtype	0x06	DescriptorSubtype: AUDIO CONTROL FEATURE UNIT

Table 21. Interface descriptors (continued)

Field	Value	Description
bUnitID	0x02	Unit ID: 2
bSourceID	0x01	Source ID:1
bControlSize	0x01	Control Size:1
bmaControls	0x0001	Only the control of the MUTE is supported
iTerminal	0x00	Unused
USB speaker output terminal descriptor		
bLength	0x09	Size of this descriptor in bytes
bDescriptortype	0x24	Descriptor type: AUDIO INTERFACE DESCRIPTOR TYPE
bDescriptorSubtype	0x03	Descriptor subtype: AUDIO CONTROL OUTPUT TERMINAL
bTerminalID	0x03	Terminal ID: 3
wTerminalType	0x0301	Terminal Type: AUDIO TERMINAL SPEAKER
bAssocTerminal	0x00	No association
bSourceID	0x02	Source ID:2
iTerminal	0x00	Unused
USB speaker standard AS interface descriptor - audio streaming zero bandwidth (Interface 1, Alternate Setting 0)		
bLength	0x09	Size of this descriptor in bytes
bDescriptortype	0x24	Descriptor type: AUDIO INTERFACE DESCRIPTOR TYPE
bInterfaceNumber	0x01	Interface Number: 1
bAlternateSetting	0x00	Alternate Setting: 0
bNumEndpoints	0x00	not used (zero bandwidth)
bInterfaceClass	0x01	Interface class: USB DEVICE CLASS AUDIO
bInterfaceSubClass	0x02	Interface subclass: AUDIO SUBCLASS AUDIOSTREAMING
bInterfaceProtocol	0x00	Interface protocol: AUDIO PROTOCOL UNDEFINED
iInterface	0x00	Unused
USB speaker standard AS interface descriptor - audio streaming operational (Interface 1, Alternate Setting 1)		
bLength	0x09	Size of this descriptor in bytes
bDescriptortype	0x24	Descriptor type: AUDIO INTERFACE DESCRIPTOR TYPE
bInterfaceNumber	0x01	Interface number: 1
bAlternateSetting	0x01	Alternate Setting: 1
bNumEndpoints	0x01	One Endpoint.
bInterfaceClass	0x01	Interface class: USB DEVICE CLASS AUDIO
bInterfaceSubClass	0x02	Interface subclass: AUDIO SUBCLASS AUDIOSTREAMING
bInterfaceProtocol	0x00	Interface protocol: AUDIO PROTOCOL UNDEFINED

Table 21. Interface descriptors (continued)

Field	Value	Description
iInterface	0x00	Unused
USB speaker audio type I format interface descriptor		
bLength	0x0B	Size of this descriptor in bytes
bDescriptortype	0x24	Descriptor type: AUDIO INTERFACE DESCRIPTOR TYPE
bDescriptorSubtype	0x03	Descriptor subtype: AUDIO STREAMING FORMAT TYPE
bFormatType	0x01	Format type: Type I
bNrChannels	0x01	Number of channels: one channel
bSubFrameSize	0x01	Subframe size: one byte per audio subframe
bBitResolution	0x08	Bit resolution: 8 bits per sample
bSamFreqType	0x01	One frequency supported
tSamFreq	0x0055F0	22 kHz

Table 22. Endpoint descriptors

Field	Value	Description
Endpoint 1 - standard descriptor		
bLength	0x07	Size of this descriptor in bytes
bDescriptortype	0x05	Descriptor type (endpoint descriptor)
bEndpointAddress	0x01	OUT Endpoint address 1.
bmAttributes	0x01	Isochronous Endpoint
wMaxPacketSize	0x0016	22 bytes
bInterval	0x00	Unused
Endpoint 1 - Audio streaming descriptor		
bLength	0x07	Size of this descriptor in bytes
bDescriptortype	0x25	Descriptor type: AUDIO ENDPOINT DESCRIPTOR TYPE
bDescriptor	0x01	AUDIO ENDPOINT GENERAL
bmAttributes	0x80	*bmAttributes*: 0x80
bLockDelayUnits	0x00	Unused
wLockDelay	0x0000	Unused

7 Revision history

Table 23. Document revision history

Date	Revision	Changes
28-May-2007	1	Initial release.
04-Oct-2007	2	Evaluation board name corrected. Reference to UM0412 added to *Section 3: Device firmware upgrade*. Note added in *Section 4.1: Mass storage demo overview*.