UM0290
User manual

STR7/STR9 USB developer kit

Introduction

The STR7/9 USB developer 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 ST 32-bit USB microcontrollers (STR71x, STR75x and STR91x).

The aim of the STR7/9 USB developer kit is to use the same certified USB library across the STR7/STR9 microcontroller families and present for each at least one firmware demo per USB transfer type.

This document presents a description of all the components of the STR7/9 USB Developer kit, including:

- Common STR7/9 USB library: All processes related to default endpoint and standard requests
- Joystick mouse demo: Interrupt transfer
- Custom HID: interrupt transfer
- Device Firmware Upgrade (DFU): control transfer
- Mass storage demo: Bulk transfer
- Virtual COM port: Bulk transfer
- USB voice demo (speaker and microphone): Isochronous transfer

Contents

1 STR7/STR9 USB firmware library

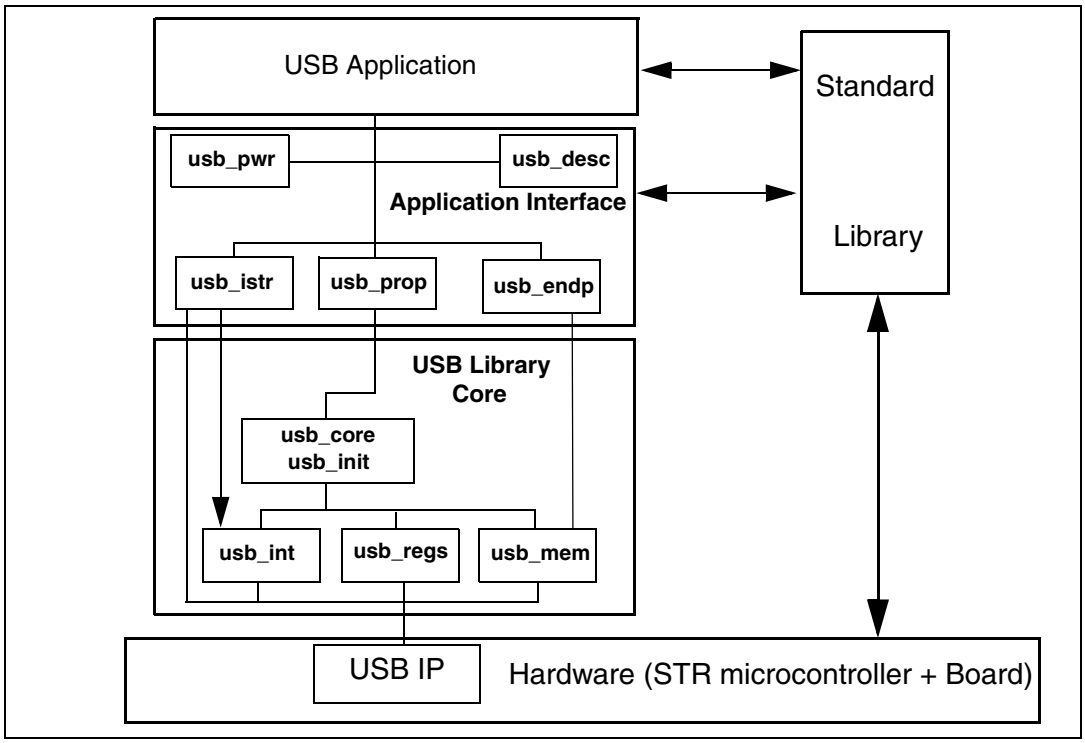
This section describes the firmware interface (called USB Library) used to manage the STR7/9 USB 2.0 full-speed macrocell.

The main purpose of this firmware library is to provide resources to ease the development of applications using the USB macrocell in all STR7/9 microcontrollers (STR71x, STR75x and the STR91x families).

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 doesn't have dependences with any Standard Software Library of STR7/9 microcontrollers.
- **Application Interface layer:** This layer presents to the user a complete interface between the library core and the final application.

Note: The application interface layer and the final application can communicate with the Standard Software 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 independency of the USB library
usb_reg (.h, .c)	Hardware abstraction layer
usb_int.c	Correct transfer interrupt service routine
usb_init (.h,.c)	USB initialization
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 STRxxx Standard 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,...)

1. **Common register functions:** These functions can be used to set or to get the 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)`

2. **Endpoint register functions:** All operations related to endpoint registers can be performed with the `SetENDPOINT` and `GetENDPOINT` functions. However, several functions are derived from these to permit 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 could have these defined values:

```
#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 could have these defined values:

```
#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
```

3. **Buffer description table functions:** These functions are used 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
```

4. **Double-buffered endpoint 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 the 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 the 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 the STALL status for double buffer mode which is managed by the function:

```
SetDouBleBuffEPStall: void SetDouBleBuffEPStall(u8 bEpNum,u8 bDir)
bEpNum = endpoint number
bDir   = endpoint direction
```

5. **Direct Memory Access (DMA) functions:** (only available for STR91x) For the STR91x, the data transfer from/to the PMA can be managed using the Direct Memory Access Controller (DMAC) to decrease the CPU usage. This section describes the different implemented functions used to manage the two DMA transfer modes (unlinked and linked).

a) DMA Unlinked mode functions

In this mode only a single data packet can be transferred by the DMA. Multiple endpoints can be mapped on the channel (3 in Tx mode and 10 in Rx mode). The CPU has to configure the DMA when the new CTR_TX/CTR_RX interrupt is received.

In fact the CPU needs to decode the endpoint to be served (to get the source and the destination addresses) before programming the next DMA transfer because multiple endpoints are mapped on the same channel (Tx/Rx).

Note: In unlinked mode the DMA interface doesn't mask/clear any CTR_TX/CTR_RX interrupt (the CPU is responsible for this task).

– IN Endpoint:

In unlinked mode only three endpoints can be mapped on the DMA channel using the following functions:

`void` **`DMAUnlinkedModeTxConfig`**`(u8 bEpNum ,u8 index):configure a IN endpoint to use the DMA in unlinked mode.`

`bEpNum = Endpoint number: 0 to 9.`

`index = Endpoint index: 0,1 or 2.`

`void` **`DMAUnlinkedModeTxEnable`**`(u8 index):Enable a IN Endpoint to trigger Tx DMA request.`

`index = Endpoint index: 0,1 or 2.`

`void` **`DMAUnlinkedModeTxDisable`**`(u8 index):Disable the trigger Tx DMA request for the IN Endpoint.`

`index = Endpoint index: 0,1 or 2.`

– OUT Endpoint

In unlinked mode up to 10 endpoints can be mapped in the DMA channel using the following functions:

`void` **`DMAUnlinkedModeRxEnable`**`(u8 bEpNum): Enable a OUT Endpoint to trigger OUT DMA request.`

`bEpNum = Endpoint number: 0 to 9.`

`void` **`DMAUnlinkedModeRxDisable`**`(u8 bEpNum): Disable the trigger Rx DMA request for the OUT Endpoint.`

`bEpNum = Endpoint number: 0 to 9.`

b) DMA Linked Mode Functions

In this mode only a single endpoint can be mapped on the DMA channel (Tx/Rx). The DMA can prepare linked lists (LLI) in order to manage multiple data packet transfer without CPU intervention at the end of the single data packet transfer. The DMA interface provides transfer requests to the DMA controller until the LLI is completed. The CPU is only responsible for configuring the linked lists (descriptor

chains) before enabling the DMA and, on termination of the DMA transfer (terminal count interrupt from the DMAC).

– IN Endpoint:

`void` **`DMALinkedModeTxConfig`**`(u8 bEpNum): Configure a IN endpoint to trigger DMA Tx linked mode request.`

`bEpNum = Endpoint number: 0 to 9.`

`void` **`DMALinkedModeTxEnable`**`(void): Enable a IN endpoint to trigger DMA Tx linked mode.`

`void` **`DMALinkedModeTxDisable`**`(void): Disable the trigger Tx DMA Linked request for the IN Endpoint.`

`void` **`SetDMALLITxLength`**`(u8 length): set the DMA linked list length for IN Endpoint.`

`length = length of the linked list. This value can be up to 255.`

– OUT Endpoint:

`void` **`DMALinkedModeRxConfig`**`(u8 bEpNum): Configure a OUT endpoint to trigger DMA Rx linked mode request.`

`bEpNum = Endpoint number: 0 to 9.`

`void` **`DMALinkedModeRxEnable`**`(void):Enable a OUT endpoint to trigger DMA Rx linked mode.`

`void` **`DMALinkedModeRxDisable`**`(void):Disable the trigger Rx DMA Linked request for the OUT Endpoint.`

`void` **`SetDMALLIRxLength`**`(u8 length):set the DMA linked list length for OUT Endpoint.`

`length = length of the linked list. This value can be up to 255.`

`void` **`SetDMALLIRxPacketNum`**`(u8 PacketNum): Set the number of packets to be received for each single descriptor of the Linked List.`

`PacketNum = number of packet. It can be up to 127 packets.`

`u8` **`GetDMALLIRxPacketNum`**`(void): Get the number of packets to be received for each single descriptor of the Linked List.`

c) DMA common functions:

To configure the DMA in both unlinked and linked modes the USB library provides some common functions used to synchronize the USB and the DMAC IPs and to configure the burst size for IN or OUT endpoints.

`void` **`SetDMABurstTxSize`**`(u8 DestBsize): Set the burst size for IN endpoint (destination burst size)`

`DestBsize = Destination burst size.`

`void` **`SetDMABurstRxSize`**`(u8 SrcBsize):Set the burst size for OUT endpoint (source burst size)`

`SrcBsize = Source burst size.`

`void` **`DMASynchEnable`**`(void): Enable the Synchronisation between the DMAC and the USB IP.`

`void` **`DMASynchDisable`**`(void): Disable the Synchronisation between the DMAC and the USB IP.`

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 STR7 USB IP provides two correct transfer routines:

- Low priority interrupt: managed by the function `CTR_LP()` and used for the 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 implements also 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 functions structures used by the kernel are described in the following paragraphs.

1. **Device table structure**: The core keeps device level information in the structure Device_Table. Device_Table with 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;
```

A 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 *bmRequestType* of a setup packet
- **USBbRequest** is the copy of *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 *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 *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 *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 is 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 he has 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 handlers 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:

- Select the microcontroller: the user has to select the microcontroller. For the STR91x the user has also to select the access mode to the USB IP (buffered or non-buffered) by uncommenting the dedicated line:

```
//#define STR7xx
//#define STR71x
//#definr STR75x
//#define STR91x
//#define STR91x_USB_BUFFERED
//#define STR91x_USB_NON_BUFFERED
```

- 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 “correct transfer” routines for endpoints different from endpoint 0 (EP0).

For enabling the processing of these call-back 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.

1. 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**[(a)] ***(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 can not process such requests. In this case, the user program gets the chance to use custom procedures to analyze the request, and prepare the data and pass the data to the USB core to exchange with the host. The parameter RequestNo indicates the request number. The return parameter of this function has the type: RESULT. It indicates to the core the result of the request processing.
- **RESULT (*Class_NoData_Setup)(BYTE RequestNo)** Callback procedure, it is called when a non-standard device request is recognized which does not need a data stage. The core can not 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 to the core the result of the request processing.
- **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" depending on the specific implementation and return the `USB_UNSUPPORT`[(a)] and in case of error in this 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.

---

a. * 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;
```

2. **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.

3. **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 following functions:

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 STR7/9 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 parameter `RequestNo` and all other request fields are stored in the device info structure.

The user has to implement the test of all request fields. If the request is compliant with the class to implement the function must return the USB_SUCCESS result. However if there is any problem in the request the function returns the status UNSUPPORT result 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 in case of receiving the data from the host and RAM or Flash in case of sending data to the host). This type of request is managed in the function `RESULT (*Class_NoData_Setup)(BYTE RequestNo)`. The user has to create for each class data request a specific function with the format:

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

If this function is called with the parameter Length equal to zero, it sets the `pInformation->Ctrl_Info.Usb_wLength` field with the length of data to transfer and return 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_NoData_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 a non-control endpoint

The management of data transfer using a pipe other than the default one (Endpoint 0) can be managed in the file *usb_endpoint.c*.

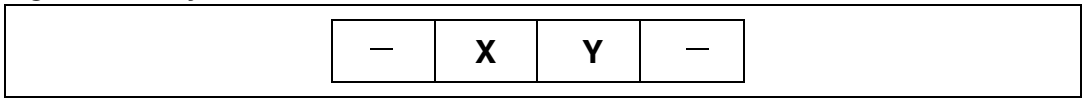
The user has to uncomment the line corresponding to the endpoint (with direction) in the file *usb_conf.h*.

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 file *usb_desc.c*.

To get the mouse pointer position the host requests four bytes of data using the pipe 1 (endpoint 1) with the following format:

Figure 2. Joystick mouse data format



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

b. File HW_config.c

3 Custom HID

3.1 General description

The HID (human interface device) class primarily consists of devices that are used by humans to control the operation of computer systems. Typical examples of HID class devices are standard mouse devices, keyboards, Bluetooth adaptors etc.

For more details on the HID device class, please refer to the “Device Class Definition for HID 1.11” available from the usb.org website.

The custom HID demo is a simple HID demo provided with a small PC applet to give an example of how to create a customized HID based on the native Windows HID driver. It consists of simple data exchanges between the STR7xx/STR91x evaluation boards and the PC Host using two interrupt pipes (IN and OUT).

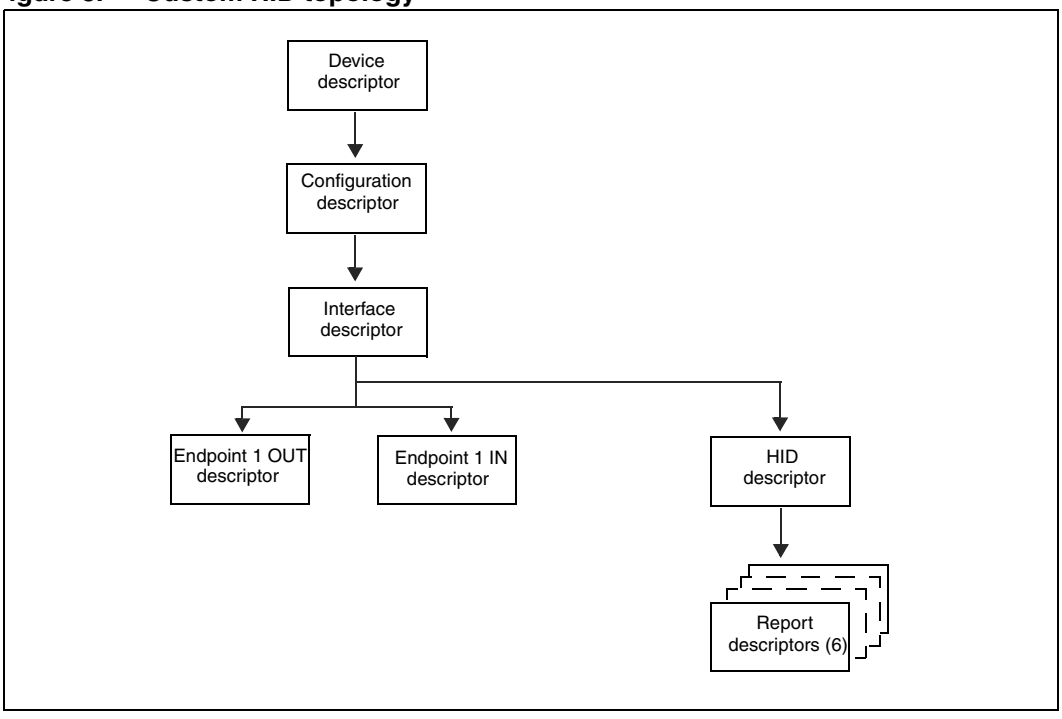
The exchanged data are related to LED commands, push-button state reports and ADC conversion values.

For more details on how to use the PC applet of the custom HID, please refer to the UM0551user manual “*USB HID demonstrator*” available from the STMicroelectronics microcontroller website www.st.com.

3.2 Descriptor topology

The custom HID topology is based on two interrupt pipes used to handle the data transfer for seven different reports. The following chart shows the custom HID topology.

Figure 3. Custom HID topology



Each Report descriptor is related to a specific component in the evaluation board (LEDs, push button or ADC). The following section described the functionality of these reports.

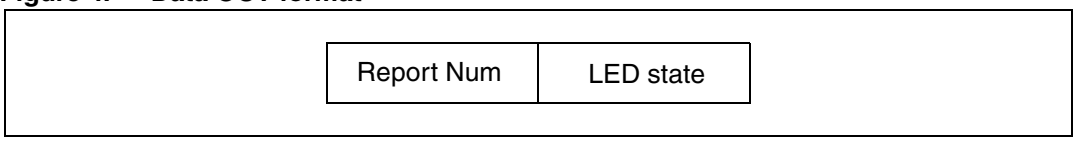
3.3 Custom HID implementation

3.3.1 LEDs control

The STR7xx/91x evaluation boards have four LEDs. In the Custom HID demo each LED correspond to a specific report (report 1 to 4) and the LEDs states (ON/OFF) are set by the PC applet.

When the device receive data on the endpoint 1 OUT the function “EP1_OUT_Callback()” is called to dispatch the received state to the correspondent LED according to the report number. The received data has the following format:

Figure 4. Data OUT format



Report Num: report number from 1 to 4.

LED State: 0 -> LED OFF

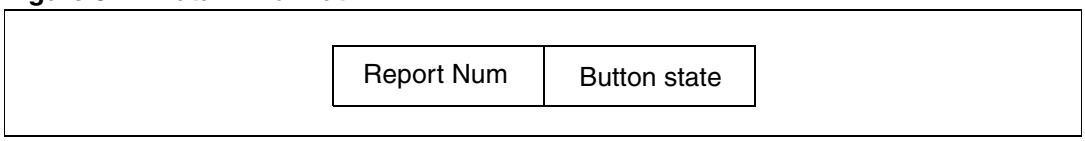
1 -> LED ON

3.3.2 Push button states report

The state of the Key push button is reported to the PC host using the endpoint 1 IN.

The Key button corresponds to the report 5. When this button is pressed the device sends to the host the related report number and the button state using the following format:

Figure 5. Data IN Format



Report Num: report number (5).

Button State: 1 -> button pressed.

3.3.3 ADC converted data transfer

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

Note: The data format is the same as the push buttons, but the report number (7) is followed by the MSB of the ADC conversion result.

4 Device firmware upgrade

4.1 General description

This part of the document presents the implementation of a device firmware upgrade (DFU) capability in the 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 other 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.).

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.

4.2 DFU extension protocol

4.2.1 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.

4.2.2 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 *idProduct* 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.

4.2.3 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.

4.3 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

4.3.1 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.

4.3.2 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. 64 bytes for the STR7 and STR9 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 STR91x Flash microcontroller, is shown below:

```
"@Internal Flash 0 /0x00000000/8*064Kg" in case of the STR9 512KB
"@Internal Flash 0 /0x00000000/32*064Kg" in case of the STR9 2MB
```

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 Extension specification release.

4.4 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 In some device applications, such as motor control or security systems, it could be the case that USB is not used in the application run-time mode, and USB might only be used for memory upgrade. These devices are called non-USB applications 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 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 enumerates with the DFU descriptor set.

4.5 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.

4.5.1 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.

4.5.2 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**

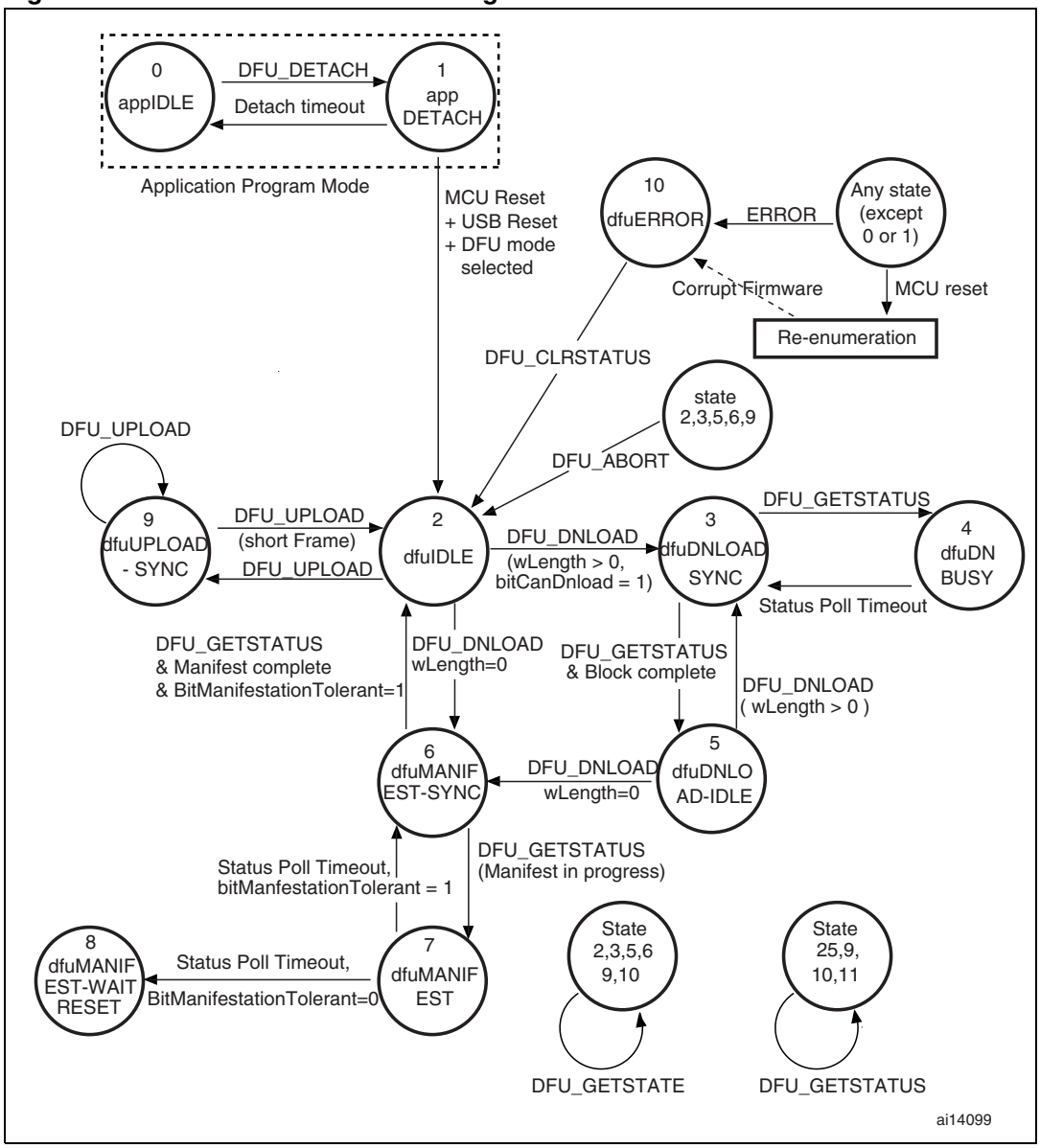
  Byte0 = 0x41 then 4 bytes containing a valid address contained in a memory sector to be erased and as already exported by the string descriptors of the Alternate settings.

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

4.5.3 DFU state diagram

Figure 6 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 6. Interface state transition diagram



Note: The state transition diagram shown in Figure 6 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.

4.5.4 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.

4.5.5 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.

4.6 DFU implementation

4.6.1 DFU mode entry mechanism

For the STR7/9 devices 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.

4.6.2 Available DFU images in the STR7/9 USB development kit

The available DFU images in the STR7/9 USB development kit are:

- Joystick Mouse Demo
- Custom HID Demo
- Mass Storage Demo (only for STR71x and STR91x)
- Virtual COM Demo
- Audio Speaker Demo
- Audio Microphone demo (only for STR75x and STR91x)

4.6.3 How to create a DFU Image

DFU Image for STR7xx devices

To create a DFU image for STR7xx devices three steps are needed:

1. Create a binary image from one of the available applications by adjusting the ROM base to the application address and remap the RAM to the address 0x0.
2. Remap the interrupt vectors to the first address of the RAM
3. Using the DFU file manager given with the DFU demo package, generate the DFU file by setting target ID to 0 (internal Flash) and the starting the application address.

DFU Image for STR91x devices

To create a DFU image for STR91x devices two steps are needed:

1. Create a binary image from one of the available applications [(c)]
2. Using the DFU file manager given with the DFU demo package, generate the DFU file by setting target ID to 0 (internal Flash) and the starting address to 0x0.

The STR7/9 USB Developer kit includes already a fully configured project examples to demonstrate how to create a DFU image for each supported device with all the available tool chains.

c. Remove all FMI initialization from the init file.

5 Mass storage demo

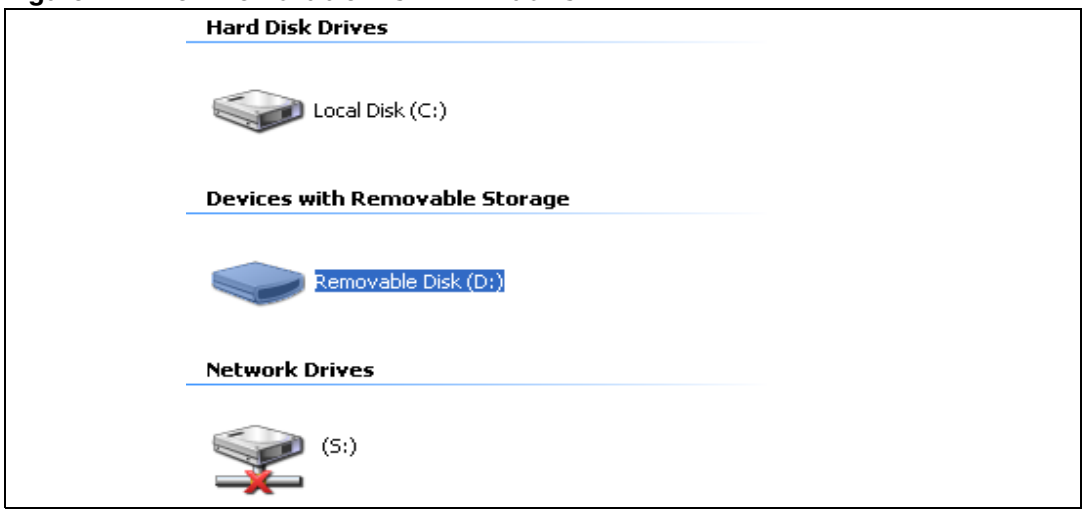
The mass storage demo gives a typical example of how to use the STR USB peripheral to communicate with a PC host using bulk transfer. The demo presents two different implementations: one for each internal transfer mode supported by the STR USB IP (simple and double buffered transfer modes). Please refer to the STR71x/91x reference manual USB section for more information about these two internal transfer modes.

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).

5.1 Mass storage demo overview

The mass storage demo complies with USB 2.0 and USB mass storage class (bulk-only transfer sub class) specifications. After running the application, the user has just 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 7*).

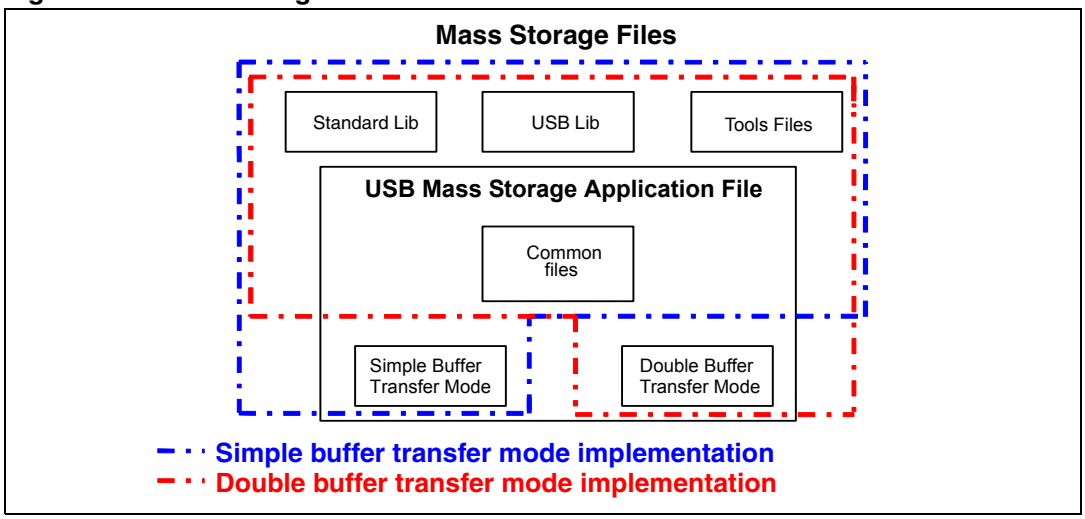
Figure 7. New Removable Disk in Windows



As described above, this demo presents two different implementations, single and double buffer transfer mode implementations.

Figure 8 shows the demo file architecture including the two implementations of the demo.

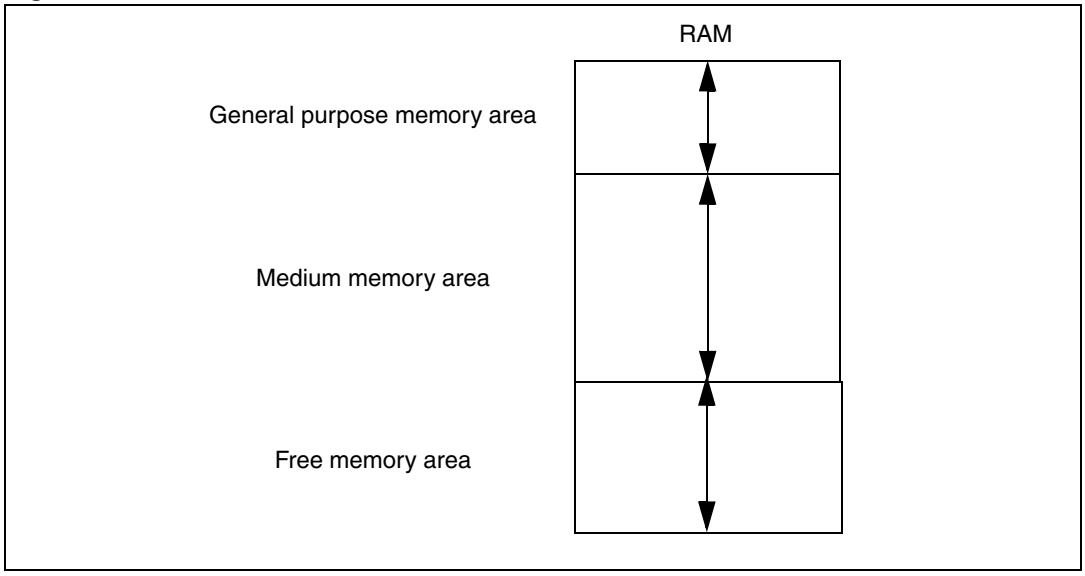
Figure 8. Mass storage file architecture



This implementation uses the internal RAM like memory support. In fact the internal RAM is divided into the following three different areas (see *Figure 9*):

- **General purpose memory area:** this part of memory is used by the firmware to store internal variables.
- **Medium memory area:** this part of memory is dedicated to the user data. It is used to store/read/write data from Windows.
- **Free area:** this part of memory is not used.

Figure 9. Internal RAM areas



5.2 Mass storage protocol

5.2.1 Bulk Only Transfer (BOT)

The BOT protocol uses only bulk pipes to transfer command, 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(s) 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. The *Figure 10* 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 must ignore 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 specifies 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 must set this field with the received value of *dCBWTag* in the related CBW.
- ***dCSWDataResidue***: the difference between the expected data (the value of *dCBWDataTransferLength* field of the related CBW) and the real value of data received or sent by the device.
- ***bCSWStatus***: the status of the related command. This field can take any of the three values shown below in *Table 13*:

Table 13. Command block status values

Value	Description
00h	Command Passed
01h	Command Failed
02h	Phase Error
03h=>FFh	Reserved

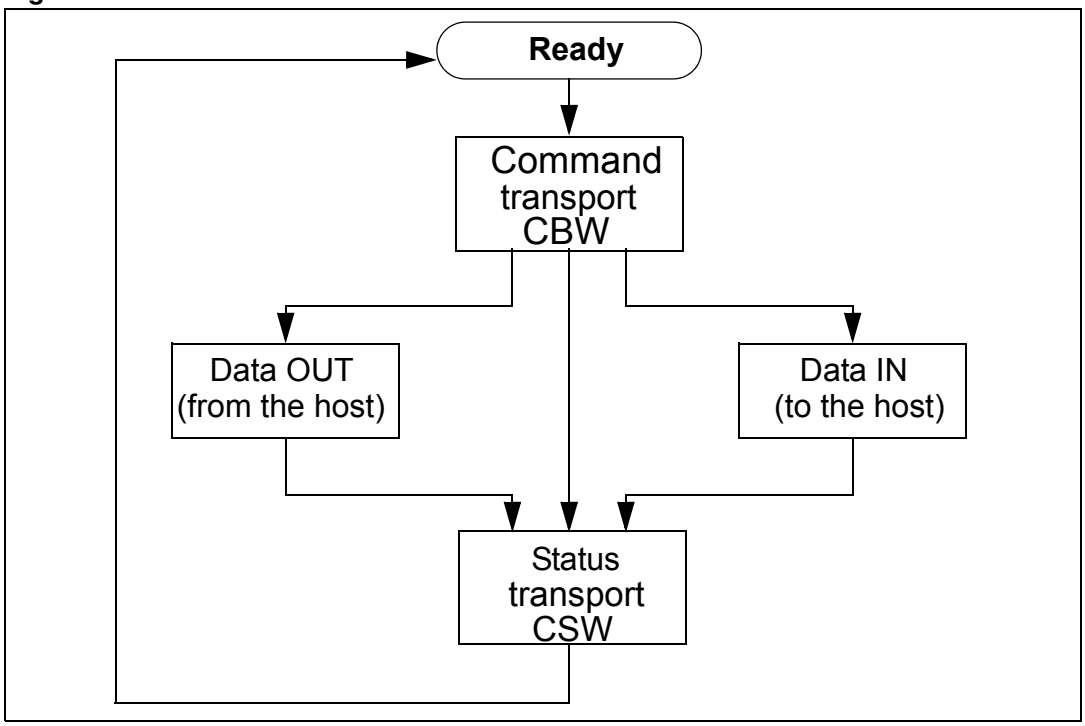
Data transfer

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

The diagram of the *Figure 10* shows the state machine of a BOT transfer.

Note: For more information about the BOT protocol please refer to the specification "Universal Serial Bus Mass Storage class Bulk-only transport".

Figure 10. BOT state machine



5.2.2 Small Computer System Interface (SCSI)

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

Table shows the SCSI command for removable devices.

Table 14. SCSI Command Set(1)

Command Name	OpCode	Command Support	Description	Reference
Inquiry	0x12	M	Get device information	SPC-2
Read Format Capacities	0x23	M	Report current media capacity and formattable capacities supported by media	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 media to the host	RBC
Read Capacity (10)	0x25	M	Report current media capacity	RBC
Request Sense	0x03	O	Transfer status sense data to the host	SPC-2

Table 14. SCSI Command Set(1) (continued)

Command Name	OpCode	Command Support	Description	Reference
Start Stop Unit	0x1B	M	Enable or disable the Logical Unit for media 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 media	RBC
Write (10)	0x2A	M	Transfer binary data from the host to the media	RBC
Command Support key: M = support is mandatory, O = support is optional				

1. This table doesn't show all the SCSI commands. For more information please refer to the SPC and RBC specifications.

5.3 Mass storage demo implementations

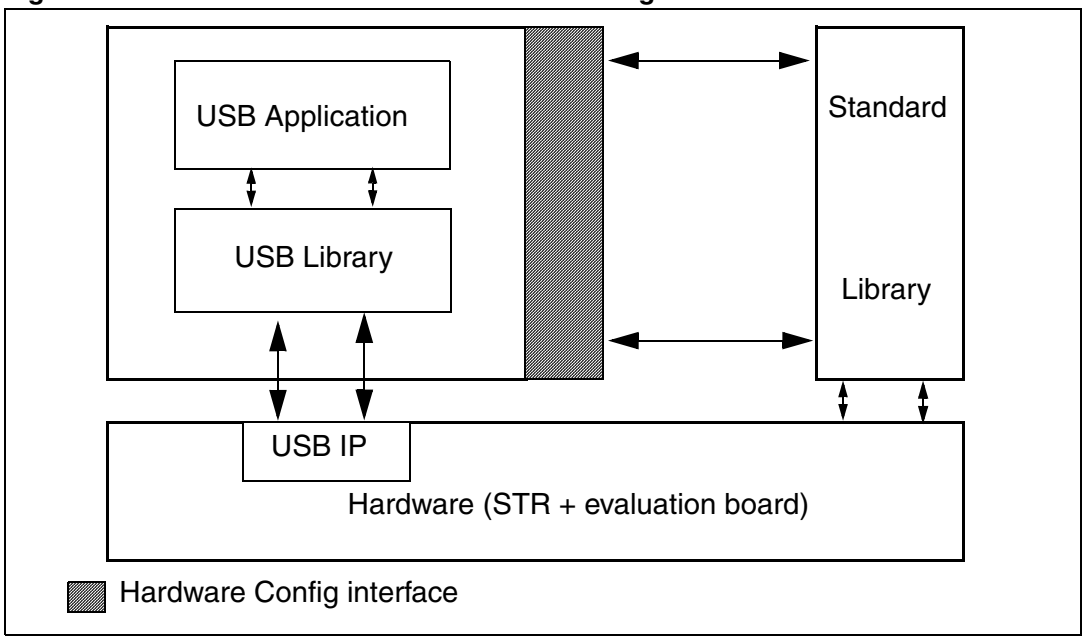
The STR7/9 USB IP presents two modes for the bulk transfer: *simple* and *double* buffer modes. So describe these two modes this demo provides two different implementations, one for each mode. The major difference between the two implementations is the data transfer management.

This chapter presents the two implementations supported by the demo.

5.3.1 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 STR microcontroller. This internal and external hardware is managed by the STR Standard Software Library, so from a firmware point of view, the hardware configuration interface is the firmware layer between the USB application and the standard library. *Figure 11* shows the interaction between the different firmware components and the hardware environment.

Figure 11. Hardware and firmware interaction diagram



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 LEDs configuration
- LEDs command
- Get the characteristics of the memory medium (the block size and the memory capacity)

5.3.2 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 function `MASS_Reset` (file *usp_prop.c*).

The default endpoint (pipe 0) configuration is the same for the two transfer modes.

To configure the endpoint 0 it is necessary to:

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

For the bulk pipes (endpoints 1 and 2) the configuration depends on the transfer mode. For the **simple buffered mode** the endpoint configuration consists of these steps:

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

In the **double buffer mode**, each unidirectional endpoint uses two buffers in the Packet Memory Area (PMA). For additional information about the double buffer transfer mode please refer to the *STR71x/91x Reference Manual*, USB SLAVE INTERFACE, section Double-Buffered Endpoints.

To configure the bulk endpoint in double buffer mode, it is necessary to use the same steps as with the simple buffer mode and to modify the transfer mode by setting the EP_KIND bit in the USB_EPxR register of each bulk endpoint. Also, provide for each bulk endpoint tow data buffer (ENDPx_BUF0Addr and ENDPx_BUF1Addr, x can be 1 or 2 according to the endpoint number) in the BTABLE (file *usb_conf.h*).

To manage the double buffer Endpoint configuration the firmware uses dedicated functions:

- **SetEPDoubleBuff()**: to set the EP_KIND bit in the USB_EPxR register
- **SetEPDblBuffCount()**: used to set the double buffer endpoint count
- **SetEPDblBuffAddr()**: used to set the buffer address

Data Management

Data transfer management depends on the transfer mode. In fact, for the **simple buffer mode**, only one data buffer is used for each endpoint. So the 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 (file *usb_mem.c*):

- **PMAToUserBufferCopy ()**: this function transfers the specified number of bytes from the Packet Memory Area to the internal RAM. This function is used 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.

However, for the **double buffer mode**, each endpoint uses two data buffers in the Packet Memory Area. So, to transfer the data from/to the PMA, it is necessary to take care of the current usage of each buffer. In fact, the firmware has to manage the swap between the ENDPx_BUF0Addr and ENDPx_BUF1Addr. The swap depends directly on the firmware and IP buffer usage, so before the access to the PMA (for write operations) the firmware tests the SW_BUF bit or/and the DTOG bit to select the free buffer.

This operation is managed in the demo by the `EP2_OUT_Callback()` function for data out transfer and by both `Send_Data()` and `EP1_IN_Callback()` function for data in transfer.

For more information about the double buffer transfer mode please refer to the *STR71x reference manual*.

5.3.3 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 is to make the device ready for the next CBW sent by the PC host.

To issue the BOT 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 function `MASS_NoData_Setup()` (file *usb_prop.c*).

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 state BOT_IDLE 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 function `MASS_Data_Setup()` (file *usb_prop.c*).

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

5.3.4 Standard request requirements

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

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

5.3.5 BOT state machine

To provide the BOT protocol, a specific state machine is implemented with 5 states, described below:

- **BOT_IDLE**: this state is the default state after a USB Reset, BOT 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 moves to this state when it has to send the last part of data asked for by the host.
- **BOT_CSW_SEND**: the device moves to this state when it has to send 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 management of this state machine is done using the functions described below (files *usb_bot.c* and *usb_bot.h* in the firmware):

- **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

Note: The BOT state machine is the same for the two transfer modes (simple and double buffer mode). The difference is only related on the data transfer (managed by the function Send_Data () in the double buffer mode).

5.3.6 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 the 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 Capacities data response (`ReadFormatCapacity_Data[ ]` from the files *SCSI_data.c*).

- **SCSI READ CAPACITY (10)** Command (OpCode = 0x25):

  Send the Read Capacity (10) data response (`ReadCapacity10_Data[ ]` from the file *SCSI_data.c*).

- **SCSI MODE SENSE (6)** Command (OpCode = 0x1A):

  Send the Mode Sense (6) data response (`Mode_Sense6_data[ ]` from the file *SCSI_data.c*).

- **SCSI MODE SENSE (10)** Command (OpCode = 0x5A):

  Send the Mode Sense (10) data response (`Mode_Sense10_data[ ]` from the file *SCSI_data.c*).

- **SCSI REQUEST SENSE** Command (OpCode = 0x03):

  Send the Request Sense data response. Note that the `Request_Sense_Data [ ]` array (file *SCSI_data.c*) is updated using the function `Set_Scsi_Sense_Data()` 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 using the function `Stor_Data_In_Flash()`.

- **SCSI READ10** (OpCode = 0x28) and **SCSI WRITE 10** (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 correctly validated the firmware launches the read or write operation from the internal RAM.

- **SCSI VERIFY 10** (OpCode =0x2F):

  The Verify 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 command Verify is received, the device tests the BLKVFY bit. If set to one, a Command Passed status is returned in the CSW.

5.3.7 Memory management

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

- Management and the validation of the address range for the command Read (10) and Write (10): this process is done by the function `Address_Management_Test()`. The role of this function is to extract the real Address and memory offset in the Medium Memory Area 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 (depending on whether the transfer is 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 medium memory access. After each access, the current memory offset and the next Access Address are updated using the length of the previous transfer.

Note: For the STR91x the read and write transfer from/to PMA is managed by the two functions PMA_Read() and PMA_Write() (file usb_rw.c). This two functions are optimized to speed up the transfer of data packets with a size multiple of 32 bits (4 bytes).

5.4 How to customize the mass storage demo

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

- **Customization 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.
- **Customization of the SCSI layer:** the implemented SCSI protocol presents, more than the supported command listed in *Section 5.3.6: 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 an unsupported command are defined by the function `SCSI_Invalid_Cmd()`, (see file *usb_scsi.c*). 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. If a user wants to support one of the previous commands they have to comment-out the

concerned line and implement their own process. For example, for the need to support the command `SCSI_FormatUnit`, comment the line:

```
// #define SCSI_FormatUnit_Cmd SCSI_Invalid_Cmd
```

And implement a process in a function with the same name in the file *usb_scsi.c*:

```
void SCSI_Invalid_Cmd (void)
{
// your implementation
}
```

In this way the custom function is called automatically by the `CBW_Decode()` function (file *usb_BOT.c*).

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.

- **Customization of the memory management layer:** this layer can be modified according to the user medium memory (external NAND Flash, SPI Flash, SD-MMC card...).

  The user has to provide specific management of this memory medium. In other words, provide a specific driver to manage the Read/Write operation and interface the driver with the memory management layer (*memory.h* and *memory.c*). The driver must also report the medium characteristics (at least the memory and the block size) to the hardware configuration layer (see *Section 5.1 on page 38*).

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 sub class
bDeviceProtocol	0x00	Device protocol
bMaxPacketSize0	0x40	Max packet size of the endpoint 0: 64 bytes;
idVendor	0x0483	Vendor identifier (STMicroelectronics)
idProduct	0x5715	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 interfaces endpoints 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 sub class: 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

6 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 with affecting the legacy PC application designed for COM Port communication.

The Virtual COM Port demo provides the firmware examples for all STRxxx family devices and the PC driver. This section provides a brief description of the implementation, and the way to run the demo.

6.1 Virtual COM port demo proposal

The demo proposal is to use the evaluation board as an USB-to-UART bridge and to provide communication between a laptop (without UART Port) and a normal PC as shown in the below in *Figure 12*. The PC application used in the communication is Windows HyperTerminal. See *Figure 13*.

Figure 12. Virtual COM Port demo as USB-to-UART bridge.

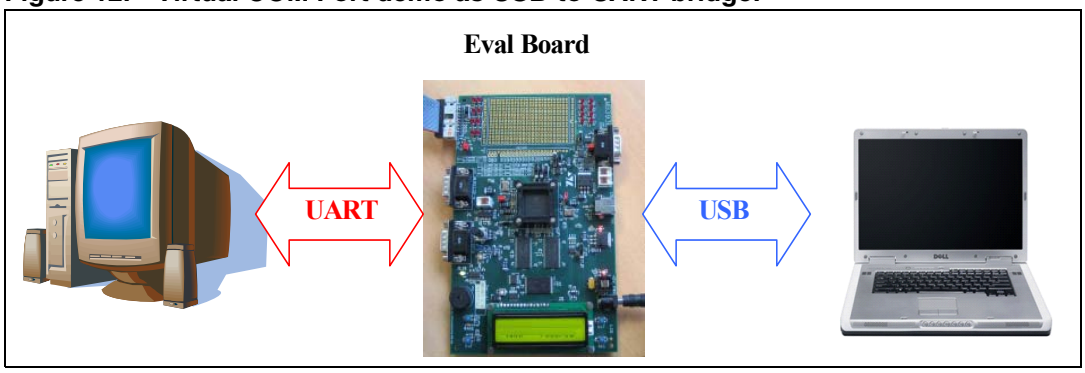
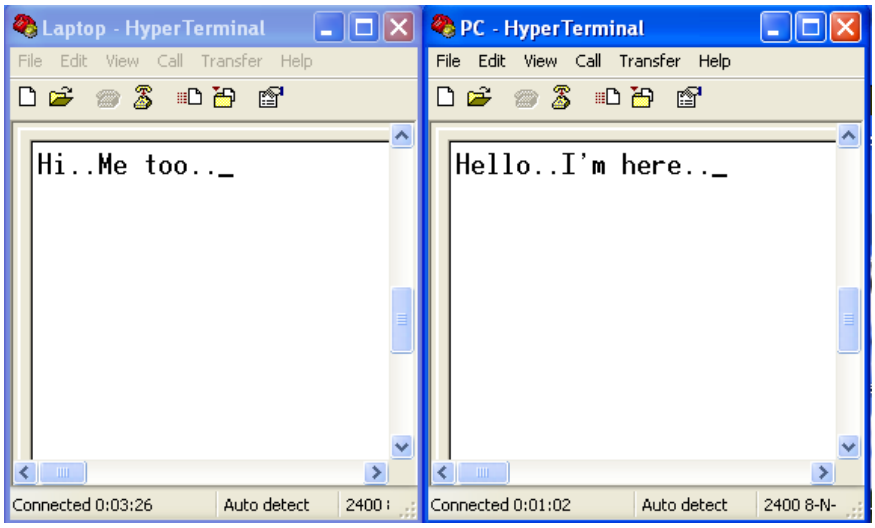


Figure 13. Communication example



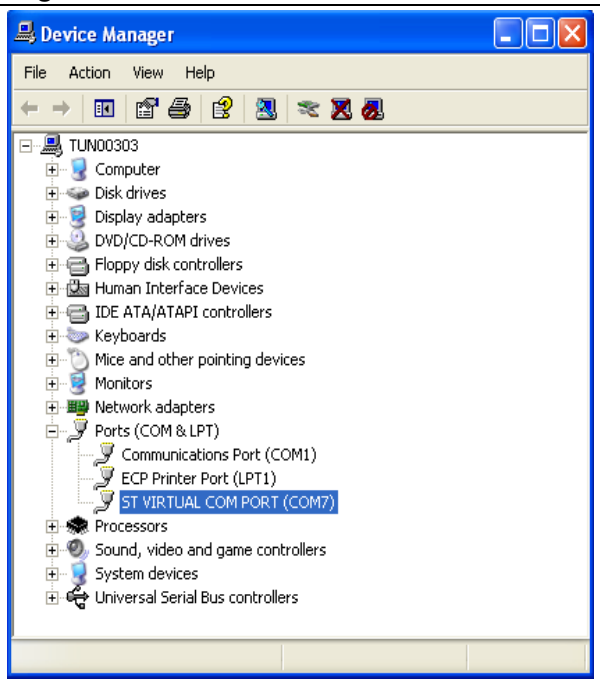
6.2 Software driver installation

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

- Load the application and run it on the 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 below in *Figure 14*.

Figure 14. Device Manager windows



6.3 Implementation

6.3.1 Hardware implementation

The Virtual COM port demo uses the UART 0 present in the evaluation board (for all micros). There is no need to add any external hardware to run the demo.

6.3.2 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 the endpoint 1 (IN) used to send the data received from the

UART 0 to the PC through USB and the endpoint 3 (OUT) used to receive the data from the PC and send it to through UART.

For more information on the CDC class please refer to the *Class Definitions for Communication Devices* specification provided by *www.usb.org* web site.

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 with a USB_SUCCESS status in the function `Virtual_Com_Port_NoData_Setup()` (file *usb_porp.c*).
- **SET_COMM_FEATURE**: Controls the settings for a particular communication feature. This request always returns with a USB_SUCCESS status in the function `Virtual_Com_Port_NoData_Setup()` (file *usb_porp.c*).
- **SET_LINE_CODING**: send 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 structure “linecoding”.

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 (UAB & UART0) clock and interrupt
- Initialize the UART 0 at Default parameters
- Configure the UART with the parameters received by the SET_LINE_CODING request
- Send the data received by the UART to the PC through PC
- Send the data received by the USB through UART

Note: The core of these routines depends on the used microcontroller.

7 USB voice demos

The USB voice demos give examples of how to use the STR USB peripheral to communicate with the PC host in the isochronous transfer Mode. This provides a demonstration of the correct method of configuring an isochronous endpoint, of receiving or transmitting data from/to the host and of how to use this data in a real-time application.

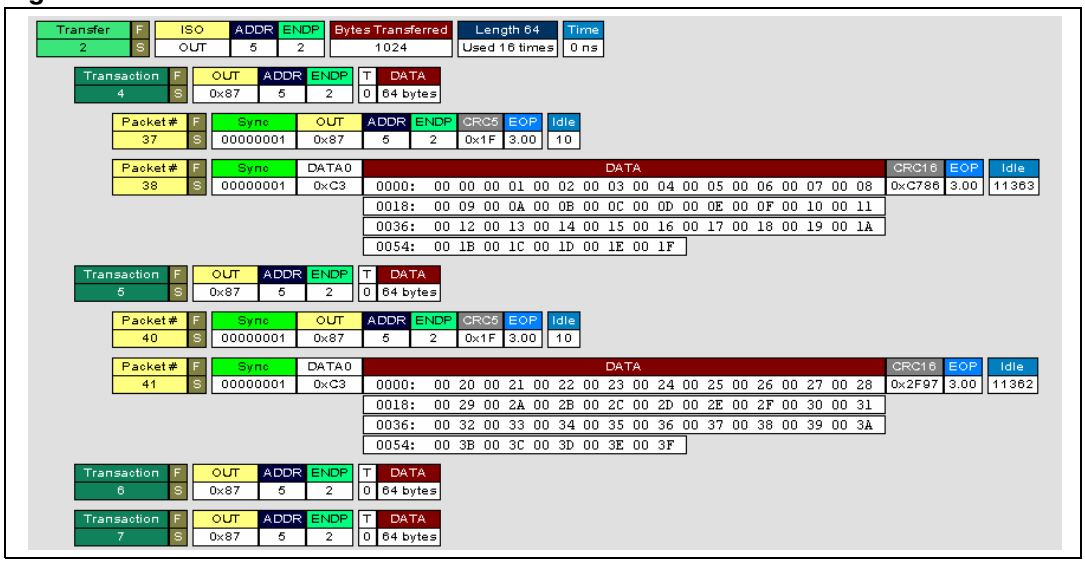
The voice demos described in this user guide are an USB speaker and an USB microphone (only for STR75x and STR91x families).

7.1 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 retrying the attempt in case of error in the data transfer operation.

In fact, an isochronous transaction doesn't have a handshake phase and no ACK packet is expected or sent after the data packet. *Figure 15* shows an example of an isochronous OUT transfer with 64 bytes in the data packet.

Figure 15. Isochronous OUT transfer



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

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

7.2 Audio device class overview

An audio device, as defined by the audio device class 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 regrouped, from a USB point of view, in the audio interface class. This class is divided into several subclasses. The audio class 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.

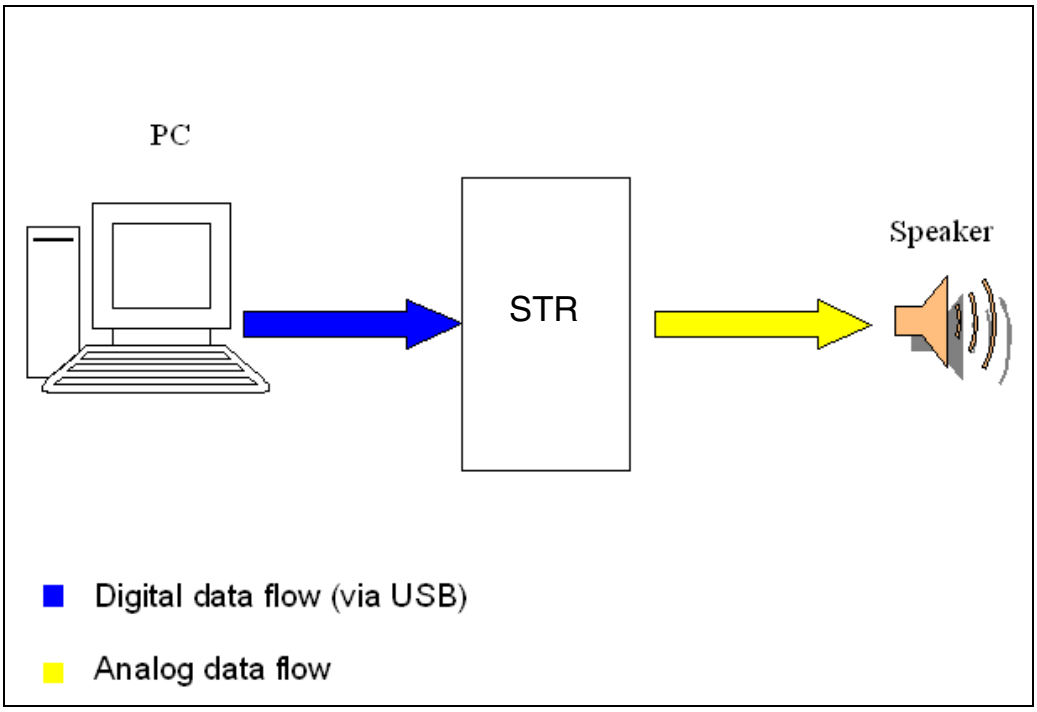
  To be able to manipulate the physical properties of an audio function, its functionality must be divided into addressable entities. Two types of such generic entities are identified and are called *units* and *terminals*. The audio class specification defines seven following types of standard units and terminals that are considered adequate to represent most audio functions:
  - 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* web site.

7.3 STR7/9 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 STR7/9 MCU. *Figure 16: STR7/9 USB audio speaker demo data flow* represents the data flow between the PC host and the audio speaker.

Figure 16. STR7/9 USB audio speaker demo data flow



The STR7/9 USB developer kit has many implementations of the USB speaker demo: one for the STR71x, one for the STR75x and four using the STR91x family (one with only CPU and the three others using both CPU and DMA).

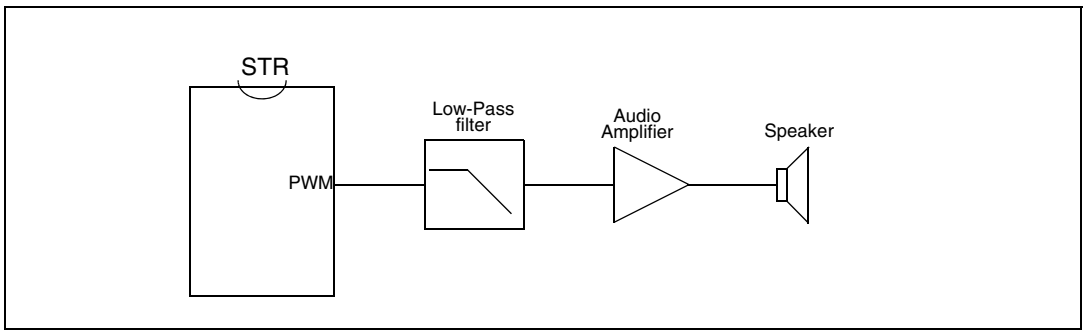
7.3.1 General characteristics

- USB characteristics:
  - Endpoint 0: used to enumerate the device and to respond to the class specific requests. The maximum packet size of this endpoint is 64 bytes.
  - Endpoint 1 (OUT): This endpoint is 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. (24 kHz for the implementation using the DMA with linked lists in STR91x microcontrollers).
- Hardware requirements:

  As the STR7/9 MCU doesn't have an on-chip DAC to generate the analog data flow, an alternative method is used to implement 1 channel DAC. Such a method is the use of 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 of the original analog signal. The *Figure 17* shows the Audio playback diagram flow using the built-in PWM.

Figure 17. Audio playback flow



7.3.2 Implementation

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

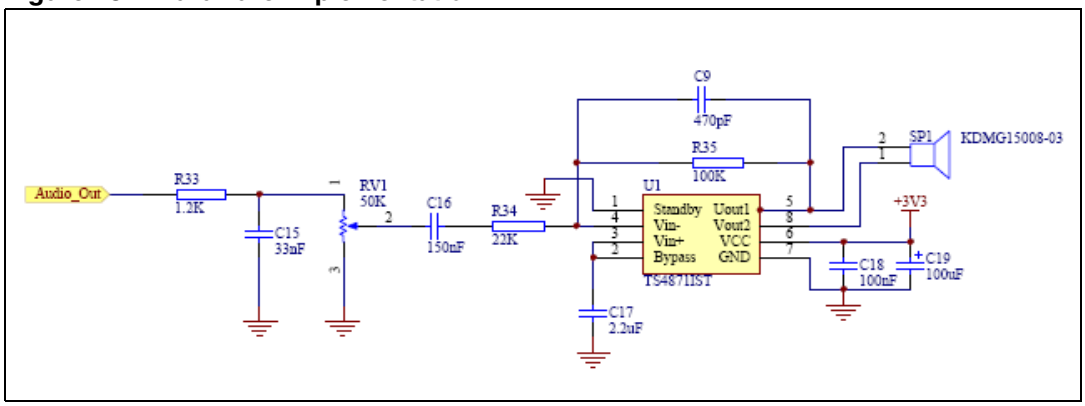
Hardware implementation

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

- TIM0 in output compare timing mode to act as SysTimer.
- TIM3 in PWM mode

Figure 18 shows the require hardware implementation to provide audio playback using the built-in PWM:

Figure 18. Hardware implementation



Note: For the STR75x and the STR91x, the audio playback hardware using the built-in PWM is already implemented in the corresponding evaluation board. There is no need to add any external hardware to run the USB speaker demo on these boards.

The T3.OCMPA pin is used to output the PWM signal. This signal is passed through a simple low-pass RC (R33 and C15) filter. The cut-off frequency of this filter is set at 4 kHz.

This ensures that almost the entire audio spectrum is passed through, but the PWM carrier is cut off.

Since the cut-off frequency of the low-pass filter is set at 4 kHz, the R and C component values result as: R = 1.2 kΩ, C = 33 nF.

The filter output is fed to the input of a common TS4871 audio amplifier through a potentiometer to provide continuous adjustability of the audio volume.

The TS4871 is monolithic integrated audio power amplifier chip from STMicroelectronics capable of delivering 0.5W of output power into an 8 Ω load at 3.3 V. The amplifier output is used to drive an 8 Ω speaker.

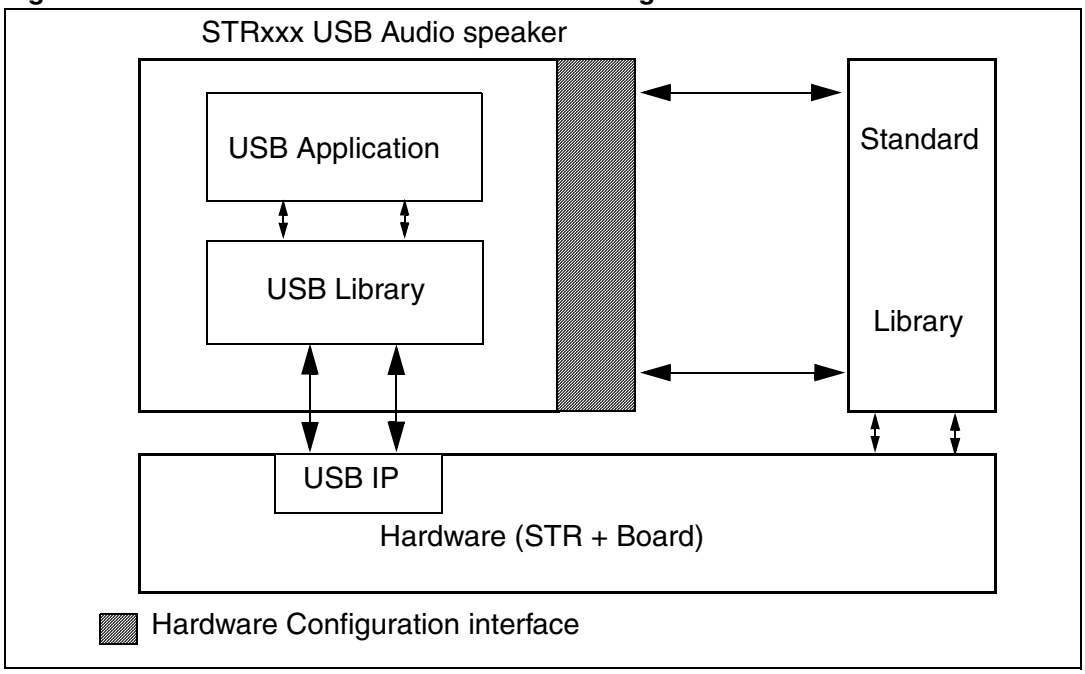
Firmware implementation

The aim of the STR 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) each 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 STR microcontroller. This internal and external hardware is managed by the STR7/9 Standard Software Library, so from a firmware point of view, the hardware configuration interface is the firmware layer between the USB application and the Standard library. *Figure 19* shows the interaction between the different firmware components and the hardware environment.

Figure 19. Hardware and firmware interaction diagram



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

– Configuration of the Timers

b) Endpoint Configurations:

In the STR USB speaker demo, two endpoints are used to communicate with the PC host: endpoint 0 and endpoint 1. Note that the endpoint 1 is an Isochronous OUT endpoint and this kind of endpoint is managed by the STR 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 file *usb_prop.c* function `STRSpeaker_Reset ()`).

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

d) Isochronous Data Transfer Management:

As detailed before, the STR7/9 manages the isochronous data transfer using double buffer mode. So to copy the received data from the PMA to the *Stream_Buffer*, we have to manage the swap between the two PMA buffers (ENDP1_BUF0Addr and ENDP1_BUF1Addr). This swapping access to the PMA is done according to the buffer usage between the USB IP and the firmware. This operation is provided by the function `EP1_OUT_Callback ()` (file *usb_endp.c*). 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 playback the audio samples received from the host, Timer TIM3 is programmed to generate a 125.5 kHz PWM signal and the TIM0 is programmed to generate an interrupt at frequency equal to 22 kHz. On each TIM0 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: Both "IN_Data_Offset" and "Out_Data_Offset" are initialized to 0 in each Start of frame interrupt (see file usb_istr.c function SOF_Callback()) to avoid the overflow of the "Stream_Buffer".

7.3.3 STR91x USB audio speaker using the DMA

For the STR91x, in order to decrease CPU usage, the Direct Memory Access Controller (DMAC) can be used to transfer the data from/to the PMA. This section describes the way to implement all DMA transfer modes in the USB speaker demo.

STR91x USB audio speaker demo using DMA unlinked mode

In this mode the CTR interrupt is not masked or cleared by the DMA. The CPU is interrupted by both the CTR and DMA terminal count interrupt.

The initial configuration of the DMA is done in the file *hw_config.c* / *.h* by the function `DMA_config()`.

The `DMA_config()` function sets the different values of the DMA_InitStruct structure with the source and destination addresses and widths, the correct transfer flow controller (USB)

and the trigger source (USB Rx). The function `DMAUnlinkedModeRxEnable()` is called to enable the DMA Rx unlinked mode.

As said before, the STR USB IP handles the isochronous transfer using the double buffer mode. So to manage the transfer of the data received by the device, after each end of DMA transfer (DMA terminal count interrupt) the `Switch_DMA_Src_Addr()` function is called to set the source address field with the next buffer to use (ENDP1_BUF0Addr or ENDP1_BUF1Addr) and to re-enable the DMA in unlinked data transfer mode.

STR91x USB speaker demo using DMA linked mode (single data packet transfer)

In linked mode, the CTR_RX is cleared and the related source is masked by the DMA controller. In this case the CPU is not interrupted by the CTR interrupt and programs the linked list descriptor before receiving the next Rx data.

In this mode the user can program one or more transfers (up to a maximum of 256 transfers).

In the case of one data packet transfer, the management of the DMA is virtually the same as in unlinked mode. The only difference is in the fact that the CPU is not interrupted by the CTR of the related endpoint, so the CTR_HP is not used and the file usb_endp.c is deleted from the project. The DMA terminal count interrupt is used to switch the source address and to re-enable the DMA linked data transfer mode.

STR91x USB speaker demo using DMA linked mode (linked list)

In this demo two stream buffers are used: one to manage the data transfer by DMA from the Packet Memory Area (USB IP) to the RAM and the second is used by the CPU to update the timer pulse. Moreover the DMA transfers up to four packets using an LLI list programmed in a specific buffer called *Linked_List_Descriptor_Table*. At the end of the four transfers, the DMA generates a terminal count interrupt, the stream buffers are switched and a new LLI is programmed to manage the next four transfers.

In this way, the CPU is interrupted (by the USB transfer) only once every four transfers and is used only to set the new LLI descriptors.

Note: In this mode the DMA can manage the transfer of 256 data packets. The first one is programmed in the DMA at startup and the others in an LLI.

7.4 STR7/9 USB microphone (only for STR75x and STR91x families)

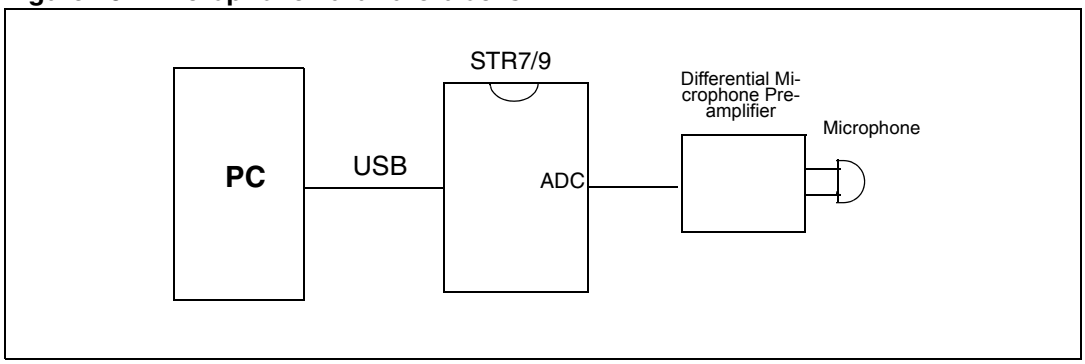
The purpose of the USB microphone demo is to convert the analog signal generated by the microphone using the built-in ADC of the microcontroller. The resulting digital audio streams are sent to the PC host via USB.

7.4.1 General characteristics

- USB characteristics:
  - Endpoint 0: used to enumerate the device and to respond to the class specific requests. The maximum packet size of this endpoint is 64 bytes.
  - Endpoint 1 (IN): This endpoint is used to transfer the audio stream data converted from the STR7/9 to 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:

  To record the audio streams, the analog signal issued from the microphone is amplified using a differential preamplifier. The resulting signal is converted by the built-in ADC (channel 12). *Figure 20* shows the hardware components used in the microphone demo.

Figure 20. Microphone hardware blocks



7.4.2 Implementation

Hardware implementation

The microphone hardware is already implemented in both the STR75x and the STR91x evaluation board. There is no need to add any external hardware to run the demo in this board.

Firmware implementation

The aim of the microphone demo is to convert the signal issued from the microphone and send it via USB to the PAC host.

The audio sampling frequency declared in the USB Micro Audio Type I Format Interface Descriptor is 22 kHz (see *USB microphone descriptors on page 70*). This means that the host asks the device for 22 audio streams (bytes) each of 1ms using endpoint 1. These audio streams represent the digital value of the audio signal picked up by the microphone every 45.45 µs.

To manage this functionality, the built-in ADC (channel) is configured in continuous conversion mode and the Timer 1 is configured to generate an interrupt every 45.45 µs.

In each timer interrupt, the ADC conversion value is stored in a buffer with a size of 22 bytes. This buffer is copied to the PMA (endpoint 1) and sent to the host after receiving the IN token.

a) Hardware configuration interface:

The hardware configuration interface is used in the USB microphone demo for:

– System and USB IP clock configuration.
– Timer1 configuration.
– ADC configuration & command.

b) Endpoint configurations:

In the STR USB microphone demo two endpoints are used to communicate with the PC host: endpoint 0 (control endpoint) and endpoint 1 (isochronous double buffer endpoint). The following C code describes how to configure an isochronous IN endpoint (see the file *usb_prop.c* function `Micro_Reset()`).

```
SetEPType(ENDP1, EP_ISOCHRONOUS);
SetEPDblBuffAddr(ENDP1,ENDP1_BUF0Addr,ENDP1_BUF1Addr);
SetEPDblBuffCount(ENDP1, EP_DBUF_IN,22);
ClearDTOG_RX(ENDP1);
ToggleDTOG_RX(ENDP1);
ClearDTOG_TX(ENDP1);
SetEPTxStatus(ENDP1, EP_TX_VALID);
SetEPRxStatus(ENDP1, EP_RX_DIS);
```

c) Isochronous data transfer management:

As with the speaker demo, the data transfer in the microphone example is based on the double buffer mode. There are two PMA buffers used for the endpoint 1 and the copy of the needed data is switched between these two buffers according to the IP and SW buffer usage.

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 sub class
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 interfaces endpoints 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 sub class: 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	DescriptorSubtype: 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

Table 21. Interface descriptors (continued)

Field	Value	Description
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
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	DescriptorSubtype: 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

Table 21. Interface descriptors (continued)

Field	Value	Description
bInterfaceProtocol	0x00	InterfaceProtocol: 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	InterfaceProtocol: AUDIO PROTOCOL UNDEFINED
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	DescriptorSubtype: 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

USB microphone descriptors

Table 23. 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 sub class
bDeviceProtocol	0x00	Device protocol
bMaxPacketSize0	0x40	Max Packet Size of the Endpoint 0: 64 bytes;
idVendor	0x0483	Vendor identifier (STmicroelectronics)
idProduct	0x5731	Product identifier
bcdDevice	0x0100	Device release number: 1.00
iManufacturer	0x01	Index of the manufacturer String descriptor: 4
iProduct	0x02	Index of the product String descriptor: 42
iSerialNumber	0x03	Index of the serial number String descriptor: 72
bNumConfigurations	0x01	Number of possible configurations: 1

Table 24. Configuration descriptor

Field	Value	Description
bLength	0x09	Size of this descriptor in bytes
bDescriptortype	0x02	Descriptor type (Configuration descriptor)
wTotalLength	0x64	Total length (in bytes) of the returned data by this descriptor (including interfaces endpoints 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 25. Interface descriptors

Field	Value	Description
USB microphone 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 sub class: AUDIO SUBCLASS AUDIOCONTROL
bInterfaceProtocol	0x00	Interface protocol: AUDIO PROTOCOL UNDEFINED
iInterface	0x00	Index of the interface String descriptor
USB microphone 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	0x001E	Total Length: 30

Table 25. Interface descriptors (continued)

Field	Value	Description
bInCollection	0x01	Number of streaming interfaces : 1
baInterfaceNr	0x01	baInterfaceNr: 1
USB microphone input terminal descriptor		
bLength	0x0C	Size of this descriptor in bytes: 12
bDescriptortype	0x24	Descriptor type: AUDIO INTERFACE DESCRIPTOR TYPE
bDescriptorSubtype	0x02	DescriptorSubtype: AUDIO CONTROL INPUT TERMINAL
bTerminalID	0x01	Terminal ID: 1
wTerminalType	0x0201	Terminal Type: USB MICROPHONE
bAssocTerminal	0x00	No association
bNrChannels	0x01	One channel
wChannelConfig	0x0000	Channel Configuration: MONO
iChannelNames	0x00	Unused
iTerminal	0x00	Unused
USB microphone 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
bUnitID	0x02	Unit ID: 2
wTerminalType	0x0101	AUDIO_USB_STREAMING
bAssocTerminal	0x00	Unused
bSourceID	0x01	Source ID: 1
iTerminal	0x00	Unused
USB microphone 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

Table 25. Interface descriptors (continued)

Field	Value	Description
bInterfaceSubClass	0x02	Interface SubClass: AUDIO SUBCLASS AUDIOSTREAMING
bInterfaceProtocol	0x00	Interface Protocol: AUDIO PROTOCOL UNDEFINED
iInterface	0x00	Unused
USB microphone 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
iInterface	0x00	Unused
USB microphone class-specific AS general interface descriptor		
bLength	0x07	Size of this descriptor in bytes
bDescriptortype	0x24	Descriptor type: AUDIO INTERFACE DESCRIPTOR TYPE
bDescriptorSubtype	0x01	Descriptor Subtype: AUDIO STREAMING GENERAL
bTerminalLink	0x02	Format Type: Type I
bDelay	0x01	Interface delay: 0x01
wFormatTag	0x0002	AUDIO FORMAT PCM 8
USB microphone 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

Table 25. Interface descriptors (continued)

Field	Value	Description
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 26. Endpoint descriptors

Field	Value	Description
Endpoint 1 - Standard descriptor		
bLength	0x07	Size of this descriptor in bytes
bDescriptortype	0x05	Descriptor type (endpoint descriptor)
bEndpointAddress	0x81	IN 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	0x00	No sampling frequency control, no pitch control, no packet padding.
bLockDelayUnits	0x00	Unused
wLockDelay	0x0000	Unused

8 Revision history

Table 27. Document revision history

Date	Revision	Changes
16-Jan-2007	1	Initial release.
13-Nov-2008	2	Added *Section 3: Custom HID on page 24* Added *Section 4: Device firmware upgrade on page 26*

Please Read Carefully:

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

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

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

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

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

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

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

ST and the ST logo are trademarks or registered trademarks of ST in various countries.

Information in this document supersedes and replaces all information previously supplied.

The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

STMicroelectronics group of companies

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

www.st.com