# 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
- 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

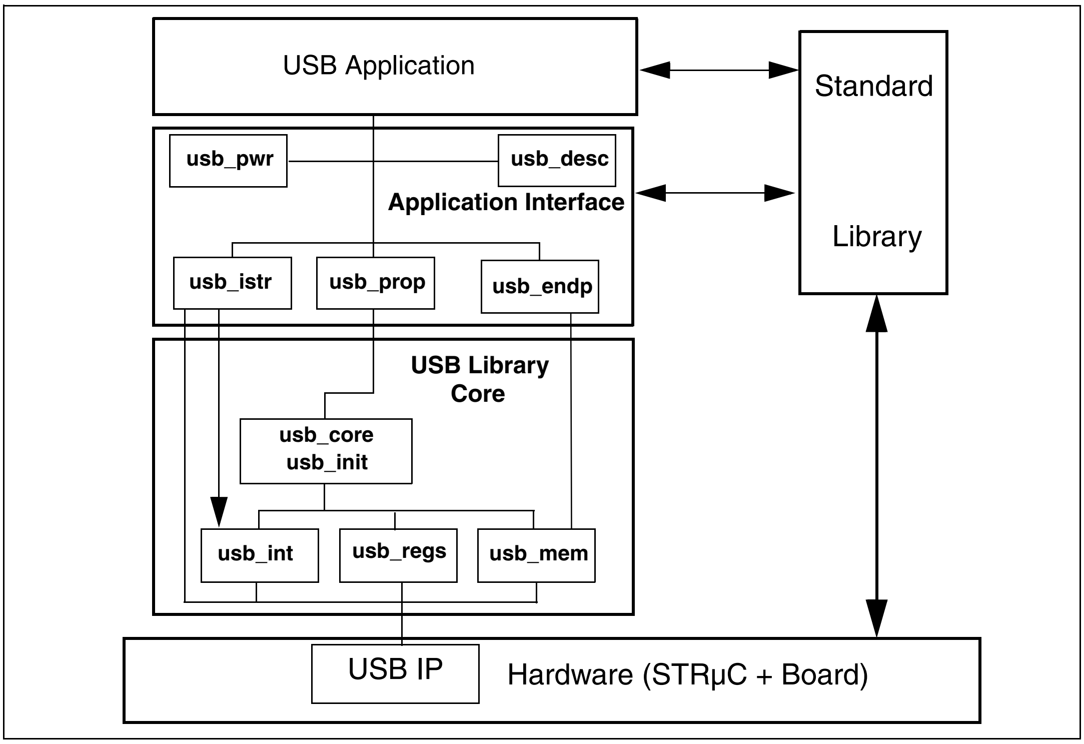
This section describes the firmware interface (called USB Library) used to manage the STR7/9 USB 2.0 full-speed macrocell.

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 in 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 llibrary 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 initializations |
| *usb_core (.h , .c)* | USB protocol management (compliant with chapter 9 of the *USB 2.0 specification*) |
| *usb_mem(.h,.c)* | Data transfer management (from/to Packet Memory Area) |
| *usb_def.h* | USB definitions |

### 1.2.1 usb_type.h

This file provides the main types used in the library. These types are dependant 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 could be used to set or to get the different common USB registers:

Table 2. Common register functions

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

## 2. Endpoint registers functions

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

a) Endpoint set/get value

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

b) Endpoint TYPE field

The EP_TYPE field of the endpoint register 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 in order to set or get the endpoints' receive and transmit buffer addresses and sizes.

a) Tx/Rx buffer address fields

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

b) Tx/Rx buffer counter fields

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

4. **Double-buffered endpoints functions**

To obtain high data transfer throughput in bulk or isochronous modes, *double-buffered* mode has to be programmed.

In this operating mode some fields of the endpoint registers and buffer description table cells have different meanings.

To ease the use of this feature several functions have been developed.

**SetEPDoubleBuff:**

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

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

**FreeUserBuffer:**

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

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

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

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

**FreeUserBuffer:** `void FreeUserBuffer(u8 bEpNum, u8 bDir);`

`bEpNum = endpoint number`

a) Double buffer addresses

These functions set or get buffers 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 buffers 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. This functionality 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;
```

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

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

**Description of the structure fields:**

– **USBbmRequestType** is the copy of *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 they have to use the corresponding functions in this structure.

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

## 1.3 Application interface

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

**Table 3. Application interface modules**

| File | Description |
|---|---|
| *usb_istr (.c,.h)* | USB interrupt handler functions |
| *usb_conf.h* | USB configuration file |
| *usb_prop (.c, .h)* | USB application specific properties |
| *usb_endp.c* | CTR interrupt 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(*) *(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 shall verify the "Interface" and "AlternateSetting" according their own implementation and return the `USB_UNSUPPORT`(*) 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.

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

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

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 shall 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 non control endpoint

The management of the 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:

| | | | |
|---|---|---|---|
| — | **X** | **Y** | — |

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()` (*) gets the user actions and returns the direction of the mouse pointer. The function `Joystick_Send()` (*) formats the data to send to the host and validates the data transaction phase.

*Note:* **File HW_config.c*

# 3 Mass storage demo

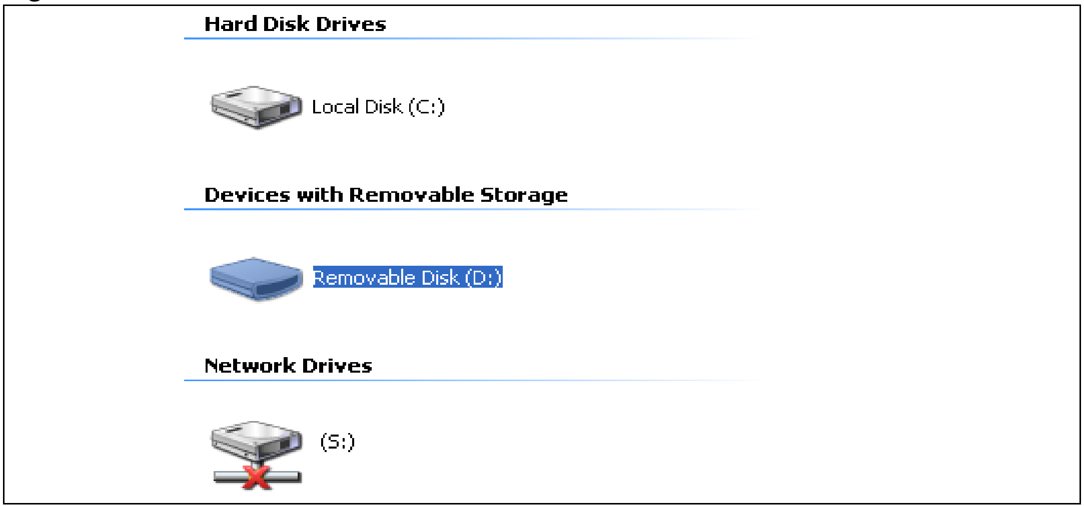
The mass storage demo gives a typical example of how to use the STR USB peripheral to communicate with PC host using the bulk transfer. the demo presents two different implementations: one for each internal transfer mode supported by the STR USB IP (simple and double buffered transfer modes). Please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

## 3.1 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 2*).

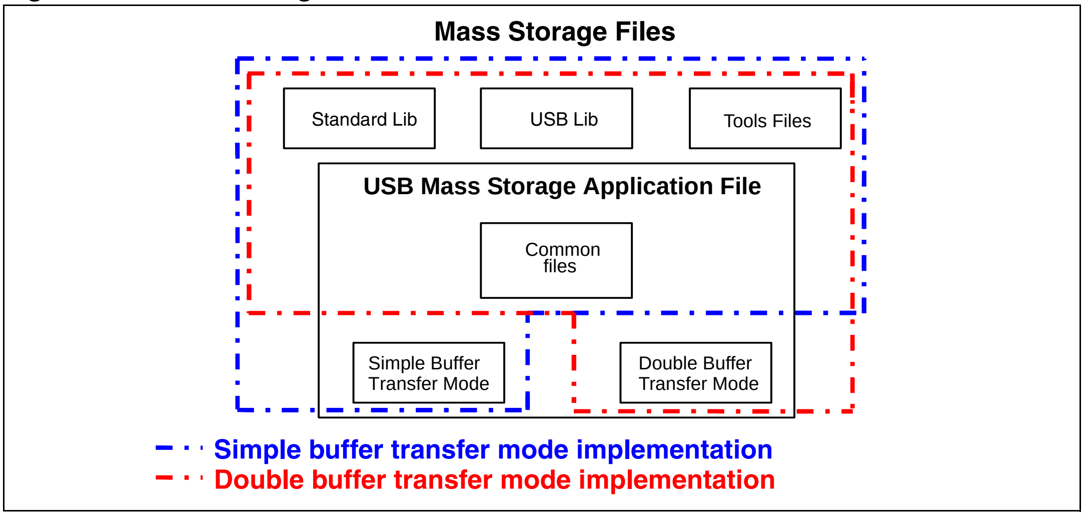
**Figure 2. New Removable Disk in Windows**




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

*Figure 3* shows the demo file architecture including the two implementations of the demo.

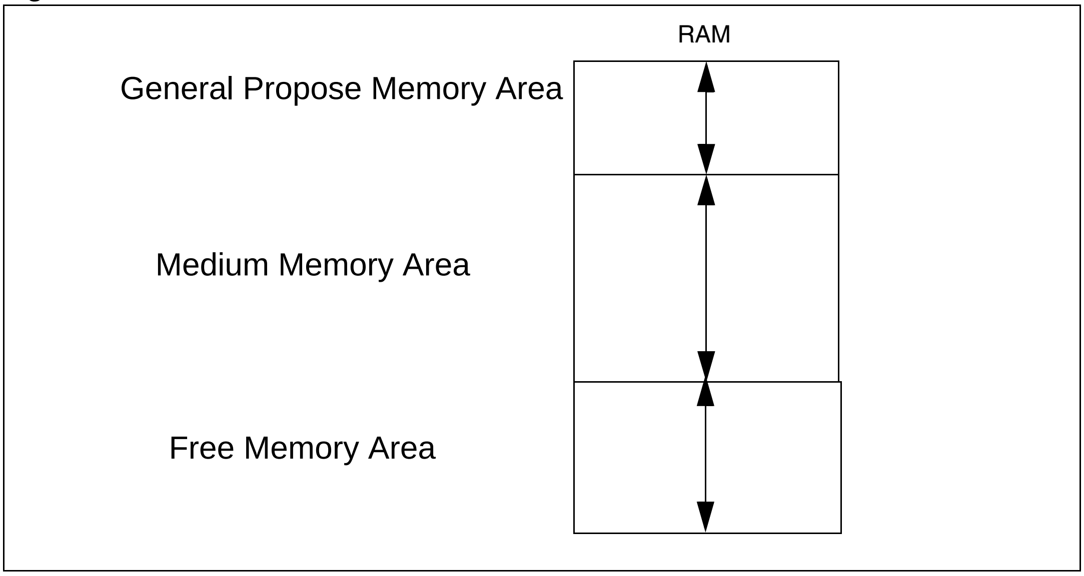
**Figure 3. Mass Storage File architecture**




This implementation uses the internal RAM like memory support. In fact the internal RAM is divided into the following three different areas (see *Figure 4*):

- **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 4. Internal RAM areas**



## 3.2 Mass storage protocol

### 3.2.1 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 5* shows the different fields of a CBW.

**Table 5. 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 shall ignore this bit if the dCBWDataTransferLength field is clear to zero.*

  **Bits 6:0**: reserved (clear to zero).

- ***bCBWLUN***: concerned Logical Unit number.
- ***bCBWCBLength****:* this field specify the length (in bytes) of the command CBWCB.
- ***CBWCB***: the command block to be executed by the device.

#### Status transfer

To inform the host about the status of each received command, the device uses the Command Status Wrapper (CSW). *Table 6* shows the different fields of a CSW.

Table 6. 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 shall set this field with the received value of *dCBWTag* in the concerned CBW.
- ***dCSWDataResidue***: the difference between the expected data (the value of *dCBWDataTransferLength* field of the concerned CBW) and the real value of data received or send by the device.
- ***bCSWStatus***: the status of the concerned command. This field can take the three several values shown below in *Table 7*:

Table 7. Command Block Status Values

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

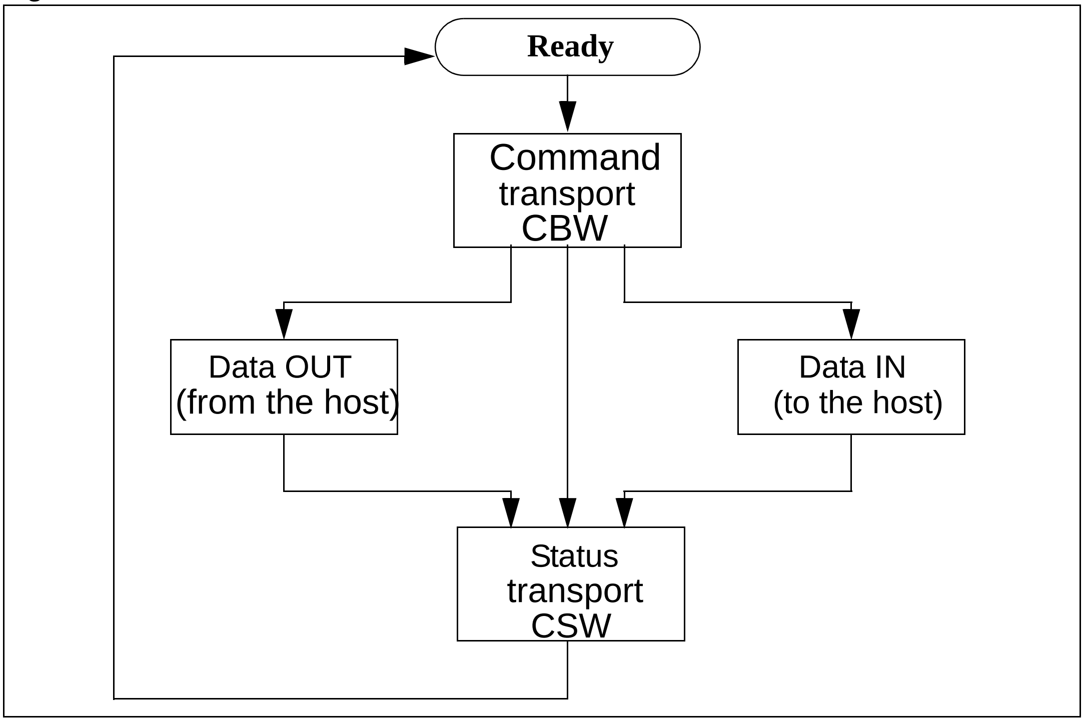
### Data transfer

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

The diagram of the *Figure 5* 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 5. BOT state machine




### 3.2.2 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 8* shows the SCSI command for removable devices (*).

Table 8. SCSI Command Set

| 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 |

| 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 | | | | |

*Note:* *(*) Note that the previous table doesn't show all the SCSI commands. For more information please refer to the SPC and RBC specifications.*

## 3.3 Mass storage demo implementations

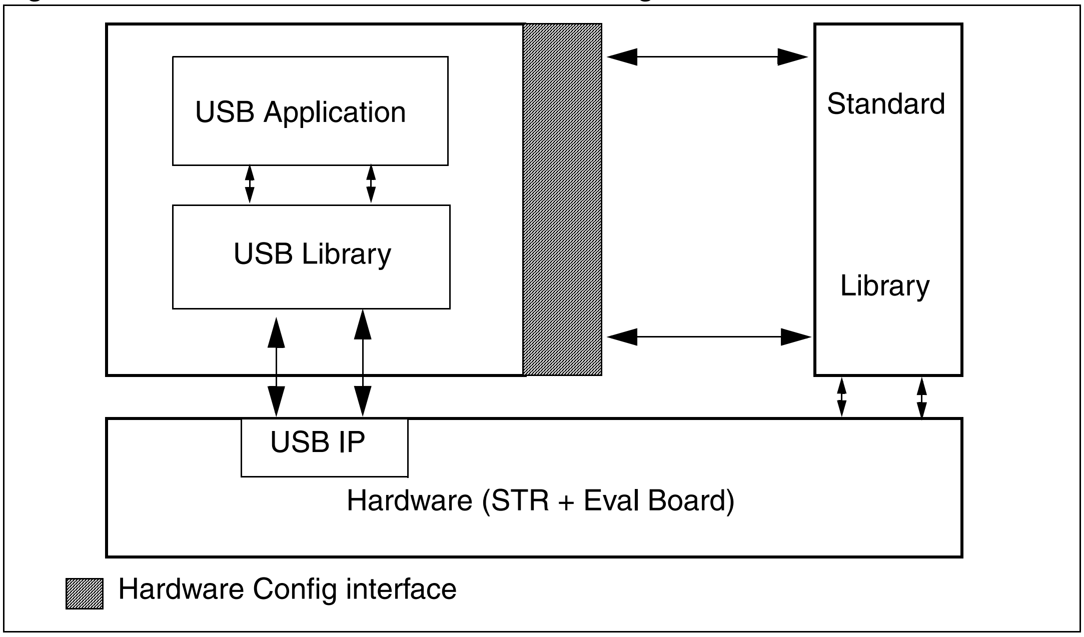
The STR7/9 USB IP presents two modes for the bulk transfer: *simple* and *double* buffer modes. So describe these two modes this demo provides two different implementations, one for each mode. The major difference between the two implementations is the data transfer management.

This chapter presents the two implementations supported by the demo.

### 3.3.1 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 6* shows the interaction between the different firmware components and the hardware environment.

**Figure 6. 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)

### 3.3.2 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 on these several 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*.

### 3.3.3 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 shall ready the device to 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 unit supported by the device.

To issue a Get Max LUN request the host shall 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:* *Note that 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 they have to return the number of implemented LUNs as response to this request.*

### 3.3.4 Standard request requirements

To be compliant with the BOT specification the device shall 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 shall 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 shall 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*.

### 3.3.5 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 one 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 in 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: Note that 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).*

### 3.3.6 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 `Resquest_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):

  Return always a CSW with COMMAND PASSED status.

- **SCSI PREVENT\ALLOW MEDIUM REMOVAL** Command (OpCode = 0x1E):

  Return always 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.

### 3.3.7 Memory management

All the memory management functions are grouped in the two files: *memory.c* and *memory.h.* The memory management consists in 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 (according to the transfer Read or Write) and return 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 into 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).*

## 3.4 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 3.3.6: 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 shall also report the medium characteristics (at least the memory and the block size) to the hardware configuration layer (see *Section 3.1 on page 22*).

### Mass storage descriptors

**Table 9. 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 10. 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:<br>Bus powered |
| *Maxpower* | 0x32 | Maximum power consumption through USB bus: 100 mA |

**Table 11. 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 12. **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 |

# 4 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 and the PC driver. This section provides a brief description of the implementation, and the way to run the demo.

## 4.1 Virtual COM port demo proposal

The demo proposal is to use the Eval 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 7*. The PC application used in the communication is Windows HyperTerminal. See *Figure 8*.

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

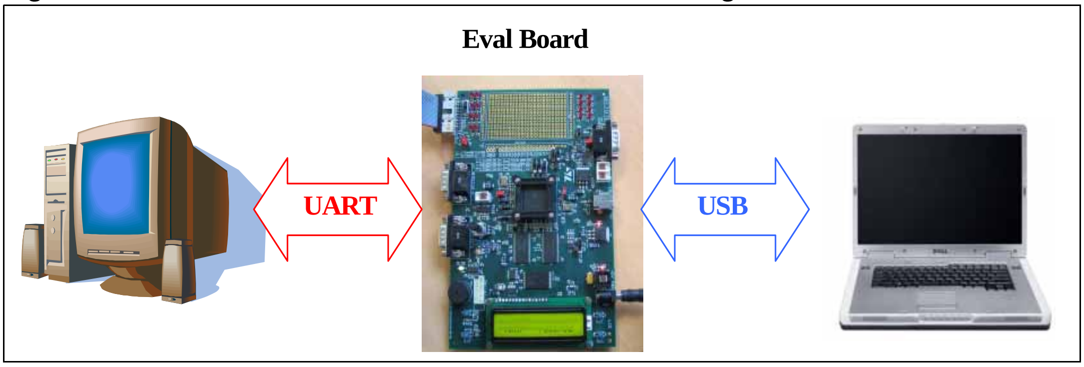
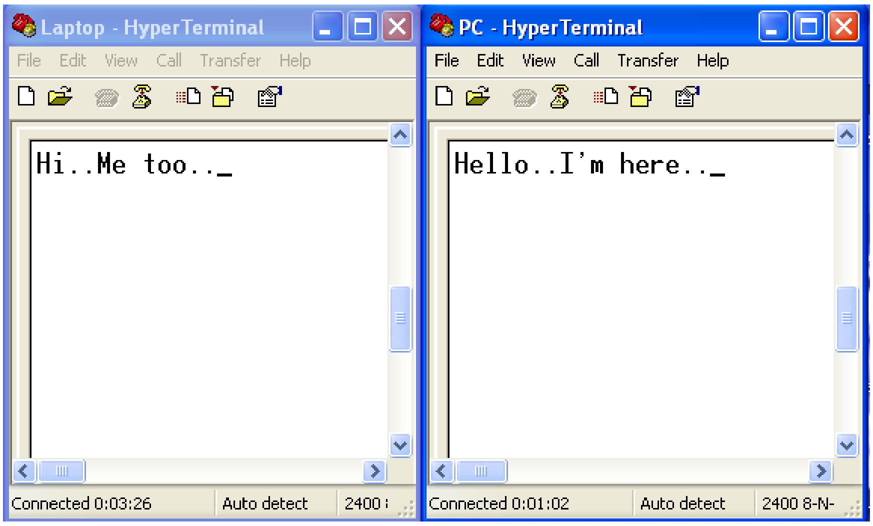



**Figure 8. Communication example**



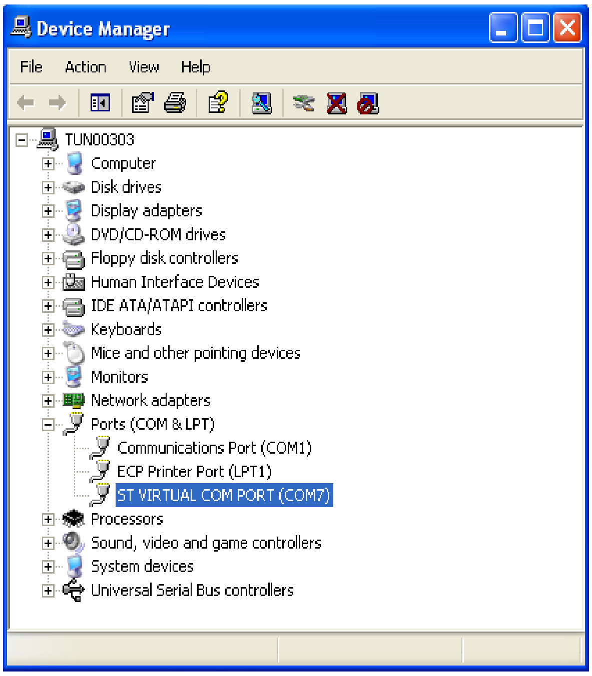
## 4.2 Software driver installation

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

- Load the application and run it on the Eval 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 9*.

**Figure 9. Device Manager windows**




## 4.3 Implementation

### 4.3.1 Hardware implementation

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

### 4.3.2 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 truth 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 interruption
- 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.*

# 5 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.

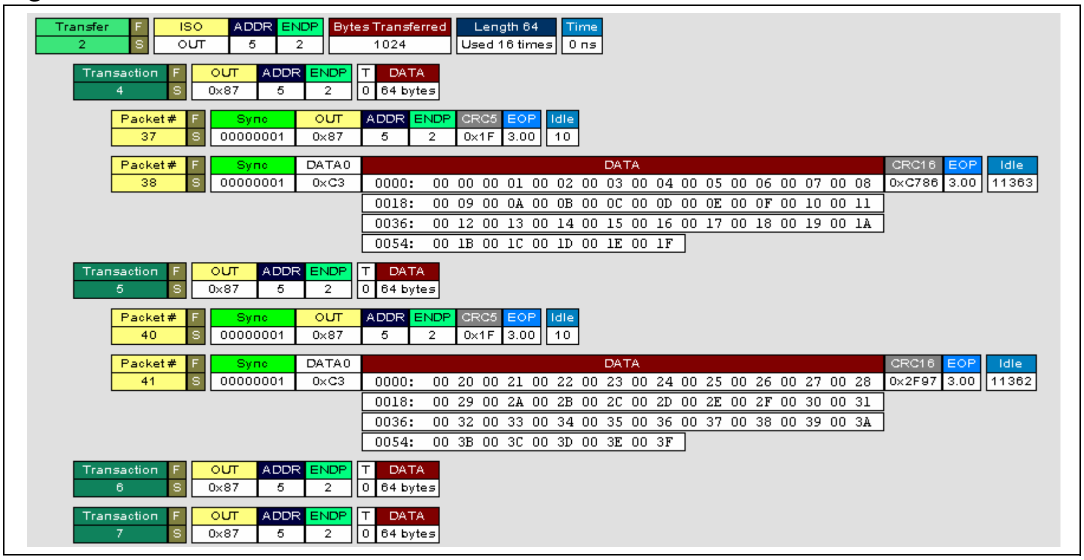
The voice demos described in this user guide are an USB speaker and an USB microphone (only for STR75x and STR91x families).

## 5.1 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 10* shows an example of an isochronous OUT transfer with 64 bytes in the data packet.

**Figure 10. 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.

## 5.2 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 soundrelated 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.

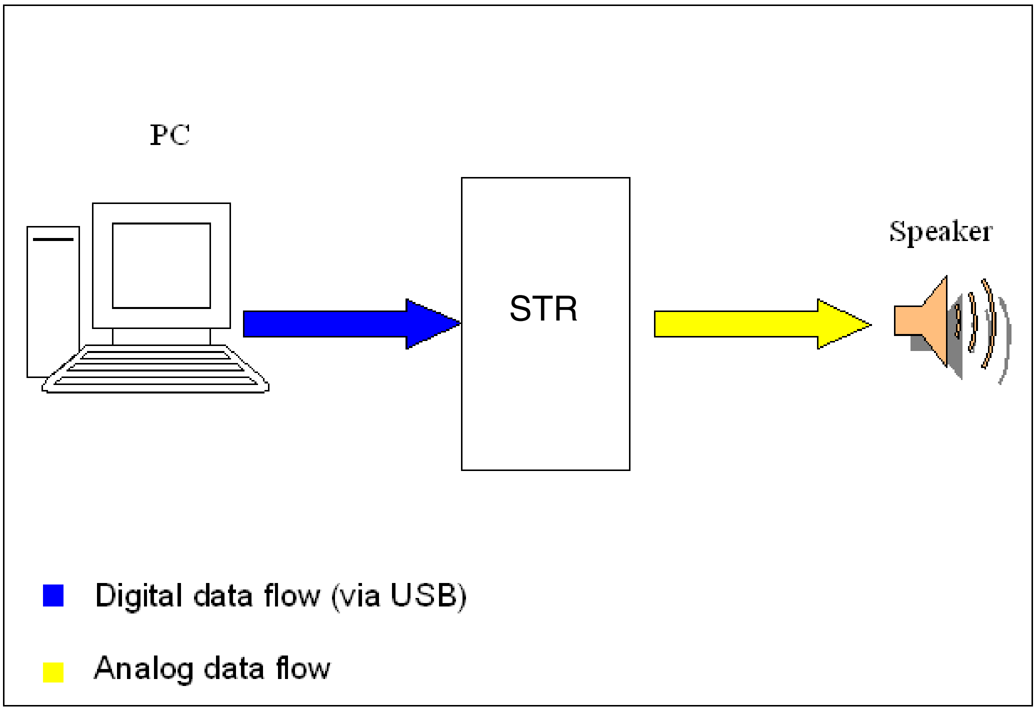
  To be able to manipulate the physical properties of an audio function, its functionality must be divided into addressable entities. Two types of such generic entities are identified and are called *units* and *terminals*. The audio class specification defines seven following types of standard units and 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.

## 5.3 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 11: STR7/9 USB audio speaker demo data flow* represents the data flow between the PC host and the audio speaker.

**Figure 11. STR7/9 USB audio speaker demo data flow**



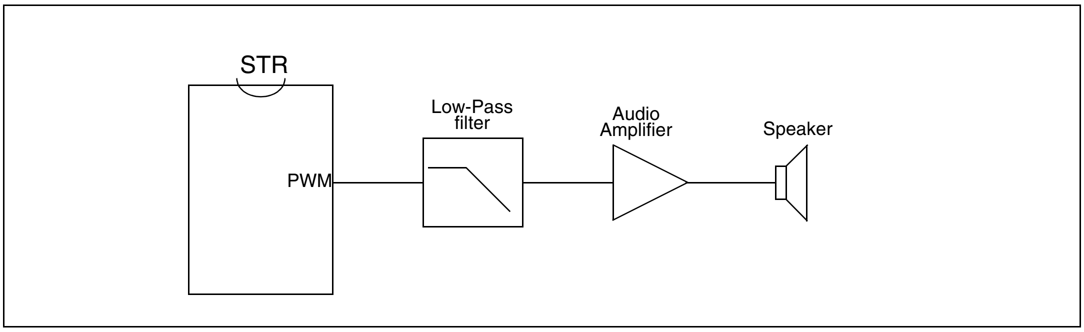

The STR7/9 USB developer kit has many implementations of the USB speaker demo: one for the STR71x, one for the STR75x and four using the STR91x family (one with only CPU and the three others using both CPU and DMA).

### 5.3.1 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 12* shows the Audio playback diagram flow using the built-in PWM.

**Figure 12. Audio playback flow**




### 5.3.2 Implementation

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

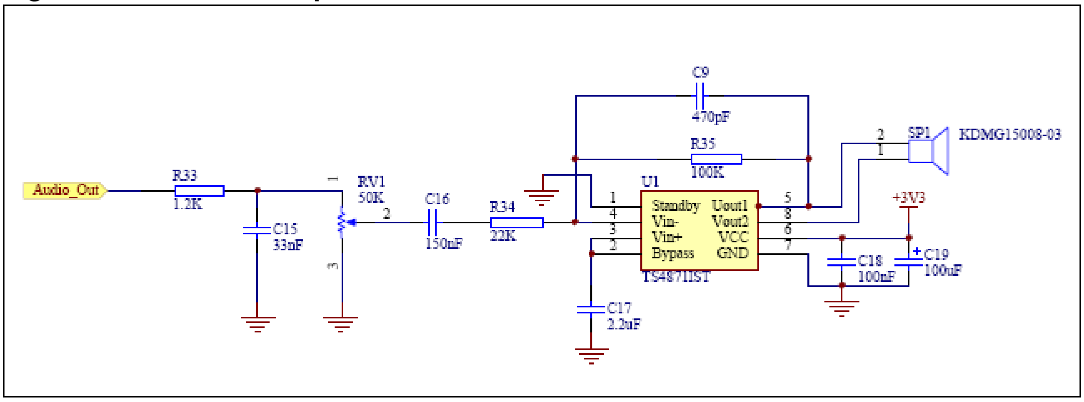
#### Hardware implementation

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 13* shows the Hardware needed implementation to provide audio playback using the build-in PWM:

**Figure 13. Hardware implementation**




*Note:* *For the STR75x and the STR91x, the audio playback hardware using the built-in PWM is already implemented in the corresponding Eval 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 8ohm load at 3V3. The amplifier output is used to drive an 8ohm speaker.

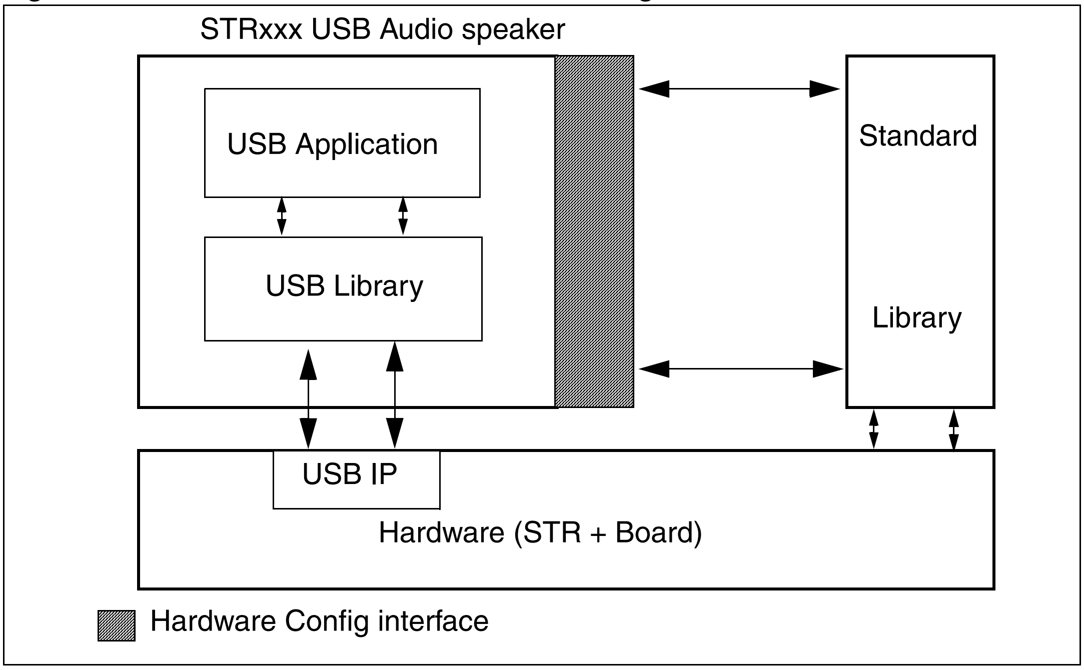
### Firmware implementation

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 14* shows the interaction between the different firmware components and the hardware environment.

**Figure 14. 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.

– Timers configuration.

b) Endpoints 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 the 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: Note that 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 over flow of the "Stream_Buffer".*

### 5.3.3 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 the 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 a LLI.*

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

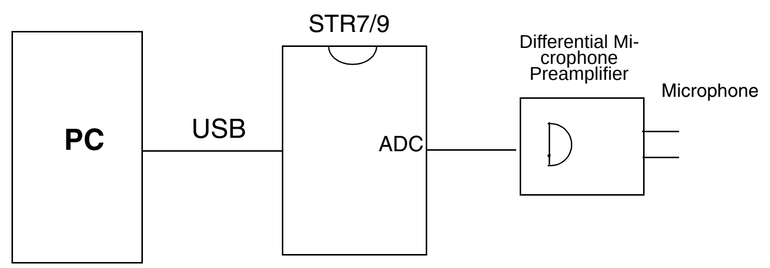
The purpose of the USB microphone demo is to convert the analogue signal generated by the microphone using the built-in ADC of the microcontroller. The resulting digital audio streams are sent to the PC host via USB.

### 5.4.1 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 15* shows the hardware components used in the microphone demo.

**Figure 15. Microphone hardware blocks**




### 5.4.2 Implementation

#### Hardware implementation

The microphone hardware is already implemented in both the STR75x and the STR91x Eval 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 54*). 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 45.45µs.

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 45.45µs.

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 in 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 13. 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 the Endpoint 0:<br>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 14. 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:<br>Bus powered |
| *Maxpower* | 0x32 | Maximum power consumption through USB bus: 100 mA |

**Table 15. 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 15. 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 15. 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 16. 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 17. 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 18. 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:<br>Bus powered |
| *Maxpower* | 0x32 | Maximum power consumption through USB bus: 100 mA |

Table 19. 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 19. 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 19. 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 19. 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 20. 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 |

# 6 Revision history

Table 21. Document revision history

| Date | Revision | Changes |
|---|---|---|
| 16-Jan-2007 | 1 | Initial release. |

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**