AN1071
APPLICATION NOTE

HALF DUPLEX USB-TO-SERIAL BRIDGE USING THE ST72611 USB MICROCONTROLLER

by USB Application Team

INTRODUCTION

This application note describes how to develop a bridge that allows a RS232 peripheral to be connected to a host computer through a USB connection.

Communication with the host computer is based on the HID (Human Interface Device) USB protocol, permitting a maximum speed of 800 Bytes/sec with a USB Low Speed device. It makes development of the user host application easy by using the native HID driver and functions library provided by the OS (Operating System).

A reference firmware has been developed by STMicroelectronics based on a low-cost, low-pin-count USB microcontroller such as the ST72611 or ST72623. It can be obtained by consulting your ST sales office.

A demonstration board has also been developed: It can be obtained by consulting your ST sales office, and the schematic is given in this document.

Table of Contents

1 SYSTEM FEATURES

1.1 USB COMMUNICATION

USB communication with the PC host is implemented in accordance with the HID (Human Interface Device) class specification. This specification uses the following pipes:

- Control pipe (Endpoint 0) is mainly used to enumerate the device. (*Set_Feature command* is also used to set some device parameters).
- Interrupt IN pipe (Endpoint 1) for data sent to the host
- Interrupt OUT pipe (Endpoint 2) for data received from the host

If the host computer does not support the Interrupt OUT pipe (Windows 98), the *Set_Output* command (Control pipe) is used instead to transfer data from host to device.

Communication is based on Interrupt IN/OUT pipes, which implies a fixed 8-byte packet size. Therefore it is necessary to provide information about the number of meaningful or relevant bytes contained in this 8-byte data packet, as described below:

Consequently, the IN and OUT Reports have the following structure:

0	1	2	3	4	5	6	7
Nb	Data	Data	Data	Data	Data	Data	Data

Nb: number of significant bytes

1.2 SERIAL COMMUNICATION

The proposed solution features 5-pin half duplex communication, where the USB-to-Serial Bridge is behaves as a DTE (Data Terminal Equipment) as defined in the RS232 specification. In this configuration, a CTS/RTS handshake is implemented as flow control.

The baud rate is configurable up to 56700 baud and a fixed 10-bit frame format is used. However, different frame formats or flow control can easily be implemented by slightly modifying the firmware.

1.3 FLOW CONTROL

A basic flow control system has been used in order to compensate the asynchronisms and speed differences between the USB and the RS232 port.

1.3.1 From USB to RS232

The basic principle is to keep the receiving endpoint disabled until the RS232 transfer is completed. This ensures no data is sent by the host until the device has correctly processed the previous data.

A second (and independent) flow control level can be implemented on the RS232 port side to check that the device connected is ready to receive data.

It is based on CTS signal monitoring by the bridge, and it must be noted that due to the relative speed differences between RS232 and USB, this protection is rarely activated.

1.3.2 From RS232 to USB

This case is more critical since the most common RS232 Baud rates are higher than the Low Speed USB bandwidth. Two flow control methods can be considered:

1. Use a temporary circular buffer in the bridge.

2. Use handshaking (Hardware or Software) between the bridge and the serial port device if the buffer gets full.

Several implementations are possible and the choice depends on the RS232 port communication characteristics: Speed, total data amount, time between bytes or byte packets.

By default, the proposed solution includes the circular buffer whereas hardware handshaking can be enabled or disabled.

In case the buffer overflows (Hardware handshaking disabled), a “Buffer Overflow” message is sent to the host through the following report.

0	1	2	3	4	5	6	7
AA	FF	FF	FF	FF	FF	FF	FF

2 HARDWARE IMPLEMENTATION

To interface with a RS232 line, a level translator is required. In order to match the USB specification the level translator must be specified at least in the 4.05-5.25V range.

In case of a common 5-pin RS232 configuration, the ST3232 (2 Receivers, 2 Transmitters) can be used, while the ST3243 is needed for a complete RS232 configuration (5 Receivers, 3 Transmitters).

The example provided refers to a 5-pin implementation.

The level translator power consumption can be turned off under software control in order to match the USB Suspend mode behaviour. When using the ST3232, this is done by turning-off its VCC supply through a PNP transistor controlled by the PA2 I/O pin on the microcontroller.

If the ST3243 is used instead, software control has to be adapted to activate the power-down mode of the ST3243.

See the schematic and Bill Of Materials in the appendix for more details.

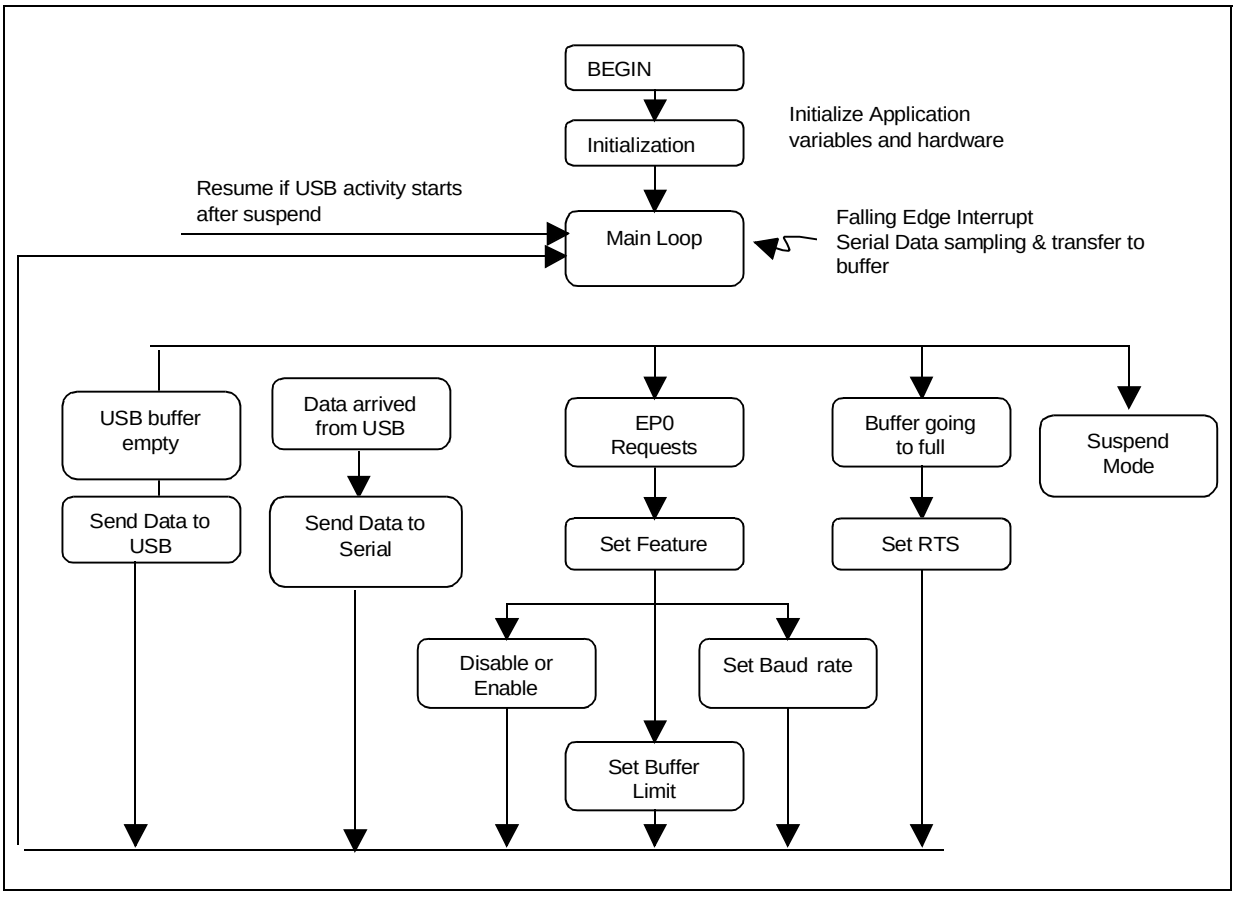
3 SOFTWARE IMPLEMENTATION

3.1 ARCHITECTURE

The firmware is built based on the USB library V4.x for ST7 USB Low Speed devices. This library contains all the necessary functions to handle USB communication, including HID protocol requests.

A key benefit of this library is that some variables are shared by the user application and the USB event manager. This makes the software architecture quite simple, as shown below.

It is highly recommended to look at Application note AN1325 for a detailed description of the shared variables.



3.1.1 Initialization

At the beginning, the firmware initializes the both USB and the application.

The *Init_App_HW()* function, in *My_Init.c* allows users to initialize the program as well as the hardware.

After the initialization, the firmware enters the USB polling main loop.

3.1.2 Main loop tasks

3.1.2.1 Check suspend flag

```
if (USBLibStatus & USB_SUSPEND)
```

The suspend flag (*USB_SUSPEND*) in the *USBLibStatus* variable is set, this indicates there has been no Start Of Frame (SOF) from USB host for more than 3ms. In this case the firmware should enter suspend mode to reduce power consumption (in our hardware example, this turns-off the ST3232 supply).

3.1.2.2 USB Standard Transfer

```
Handle_USB_Events();
if (USBLibStatus & APP_REQUEST)
                    {
Handle_APP_Requests();
Enable_STATUS_Stage();
}
```

If the suspend flag is not set, the firmware will process the standard transfer on Endpoint 0.

3.1.2.3 Check Data from USB

```
//Test if some data arrived on EP2 OUT
                    if ((USBEP2RB & 0x30) == 0x20)
                    {
                          while (ValBit (PADR, CTS)) {asm nop;}
SendSerial (EP2OutBuffer[0],2);
                          Set_EP_Ready(2, EP_OUT, 8);                }
```

Check whether data is received from USB. If yes, the firmware disables all interrupts and then sends the data to the RS232 port.

3.1.2.4 Check Feature Report

```
                    if((Status & SetFeat)&&(Test_EP_Ready(0,EP_OUT)))
                    {
                          if(EP0OutBuffer[0] == 0)
                          …
}
```

If a feature report is received, the firmware will configure itself according to the feature report.

The following features can be set:

Baud rate: 4800, 9600, 19200, 38400, 57600.

Handshake: disabled or enabled

Buffer Limit: a value set to define the buffer full condition.

3.1.2.5 Fill USB Buffer

```
if (Test_EP_Ready(1, EP_IN)) //Checks USB buffer ready
      Send_USB();    //copy data to USB buffer
```

The firmware checks first whether endpoint 1 is ready for transmission: this means the previous IN token has been processed and the USB buffer can be safely overwritten. Secondly it checks if any data are available in the ring buffer. If so, the following sequence is applied:

– Copy bytes from the ring buffer to the Endpoint buffer

– Set the endpoint to ready state with the SetEPReady() routine from the USB library

As a result the data copied are automatically transferred to the host (by the USB embedded state machine) the next time an IN token occurs.

3.1.2.6 Set Handshake

```
      if (Program_Status & USE_HANDSHAKE)
{
…
}
```

If handshaking is enabled, the firmware will use hardware handshaking.

3.1.3 Interrupts

Two interrupt sources have been used in this firmware: the USB interrupt and the Port B external interrupt.

3.1.3.1 USB Interrupt

The USB Interrupt sets flags according to the USB events. The event processing is done within the main loop. Please refer to Application Note AN1325 for more details about the USB Interrupt.

As explained in the RS232 Data Reception section, the USB Interrupt routine of the USB Library has been slightly modified to monitor the Rx pin.

3.1.3.2 Port B external Interrupt

Data reception is based on an interrupt routine triggered by a falling edge on the Rx pin. The falling edge corresponds to the transition “idle to Start bit”. The Interrupt routine manages the serial data sampling, its transfer to the internal data buffer and the update of pointers to this buffer.

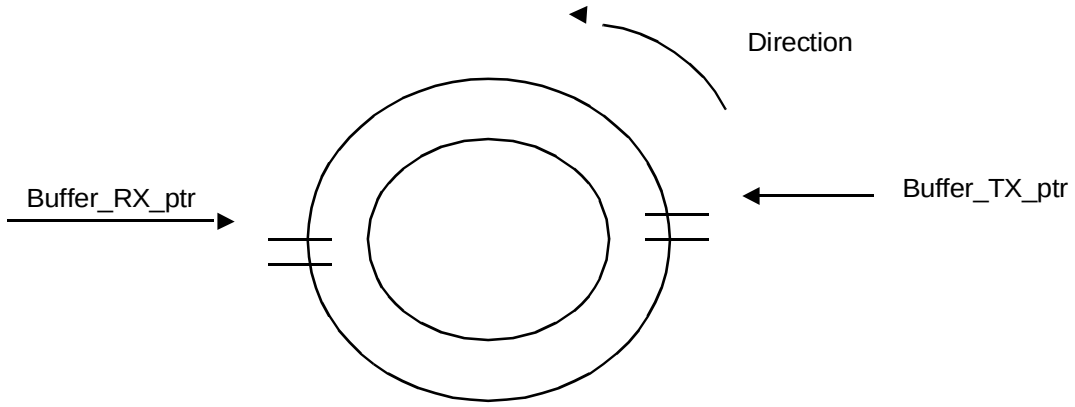
3.2 FLOW CONTROL

3.2.1 Data Buffer

The proposed firmware is based on a ring buffer approach for data transfer RS232 to USB. Its size is defined as 128 bytes by the BUFFER_SIZE constant in User_Def.h.

Also, BUFFER_MASK specifies the buffer mask. The buffer size and limit have the following values unless the program is modified:

Buffer size	16	32	64	128	256
Buffer Mask	0x0F	0x1F	0x3F	0x7F	0xFF



3.2.1.1 Buffer_RX_ptr Pointer

This buffer points to the data received on the RS232 port. When a data has been received, it is stored in the buffer at the location pointed to by Buffer_RX_ptr and its value is incremented by 1. When it reaches the size of the buffer, the pointer is reset to its initial value.

3.2.1.2 Buffer_RX_ptr Pointer

When a Data IN token is received, the firmware moves some data from the ring buffer (Buffer_Tx_ptr) to the EP DMA buffer for transmission. The amount of data fetched depends on the difference (delta) between the RX pointer and the TX pointer. If the difference is larger than 7, only 7 bytes will be fetched and sent to the USB. The Buffer_TX_ptr will then be incremented by 7. If delta is less than 7 then delta bytes will be fetched and sent, and the Buffer_TX_ptr will be incremented by delta.

3.2.1.3 Overflow condition

If the RS232 port transfer rate is higher than the USB transfer rate, the Buffer_RX_ptr pointer might be incremented much faster than the Buffer_Tx_Ptr.

An overflow occurs when Buffer_RX_ptr is equal to Buffer_TX_ptr. This check is performed in the reception interrupt routine.

3.2.2 Handshaking

The safest way to avoid buffer overflow is to enable a handshaking system on the RS232 port.

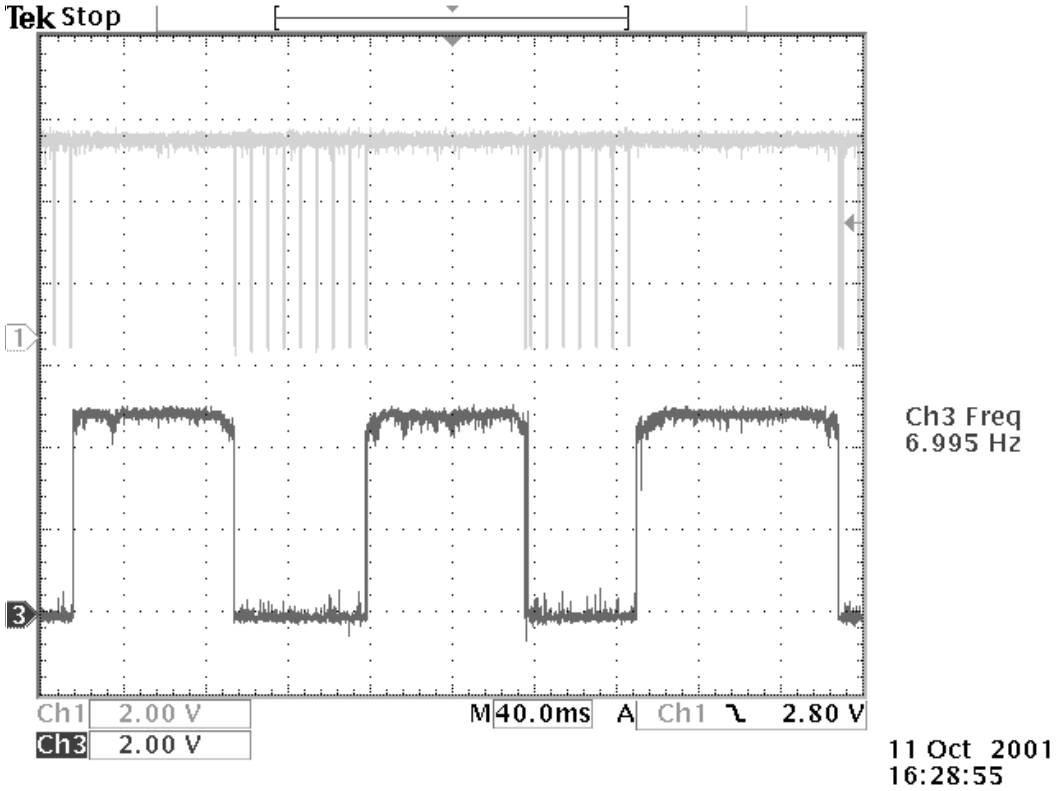
3.2.2.1 RS232 port Data transmission

The principle is to send data on the RS232 port only when the CTS signal is asserted (Active LOW).

```
UART_polling in uart.c.
if ((USBEP2RB & 0x30) == 0x20)          //Test if some data arrived
      {
                  while (ValBit (PADR, CTS)) {asm nop;} // check handshake

                  //If yes, send the EP2OutBuffer[0] bytes to RS232
                   SendSerial (EP2OutBuffer[0], 2);
                  //Sets back the EP ready
                  Set_EP_Ready(2, EP_OUT, 8);
      }
```

The code *"while (ValBit (PADR, CTS)) {asm nop;} "* checks the level of CTS. A high level indicates the device is not able to receive data. Therefore, the firmware waits until CTS becomes low before reenabling the endpoint 2 for USB Data OUT transfer.



The above plot demonstrates the detection of CTS. In the plot, the upper channel is the data, received by the USB, sent from the firmware through RS232. The lower channel is the state of CTS. The CTS signal is generated manually.

3.2.2.2 RS232 port Data Reception

The principle is to assert the RTS signal (Active LOW) only if the internal buffer is able to receive new data.

If, for any reason, the interfacing device does not take it into account, a more complex data flow has to be put in place.

As the connected device may effectively stop sending data only after purging its own transmission buffer (16 bytes on most PC computers), the RTS signal setting must be done before the buffer is completely full: Therefore the firmware sets the RTS when the buffer is 75% full.

Here are the two conditions for setting the RTS and therefore disabling the data transfer on the serial port:

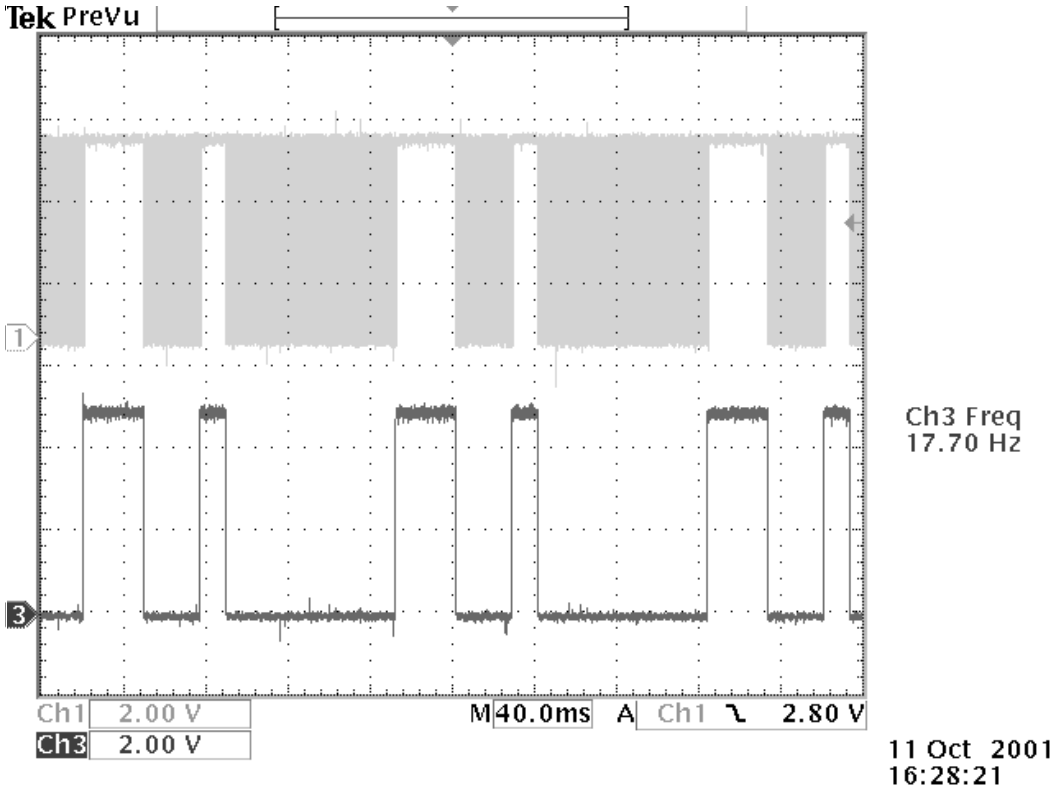
```
(Buffer_TX_ptr>Buffer_RX_ptr) &&(Buffer_TX_ptr-Buffer_RX_ptr)<Buffer_Limit)
```

or

```
(Buffer_TX_ptr<Buffer_RX_ptr) && ((Buffer_TX_ptr+BUFFER_SIZE-
Buffer_RX_ptr)<Buffer_Limit)))
```

Buffer_Limit is the threshold level at which the handshaking must be activated. Its default value has been set to 32 corresponding to 25% of a 128-byte Buffer (32=128 x 0.25).

Its value can be modified by using a Set_Feature report.



In the above plot, the upper channel is the data sent to the Bridge on the serial port while the lower channel is the state of RTS.

3.3 RS232 PORT COMMUNICATION

3.3.1 Overview

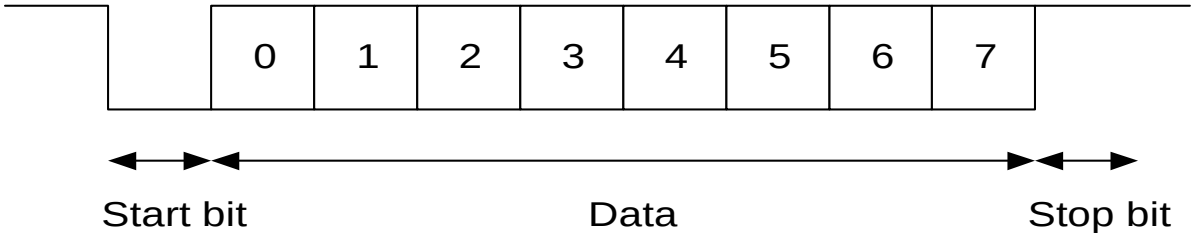
The UART required for RS232 communications is emulated by firmware in order to use a basic and cost effective microcontroller, the ST72611. As a consequence, all timings are handled by software wait loops.

As already mentioned, the proposed firmware has the following features:

– Configurable bit rate from 4800 up to 57600 bps - Default 9600.

– 10-bit frame format (1 start bit, 8 data bits, 1 stop bit)

– Half Duplex



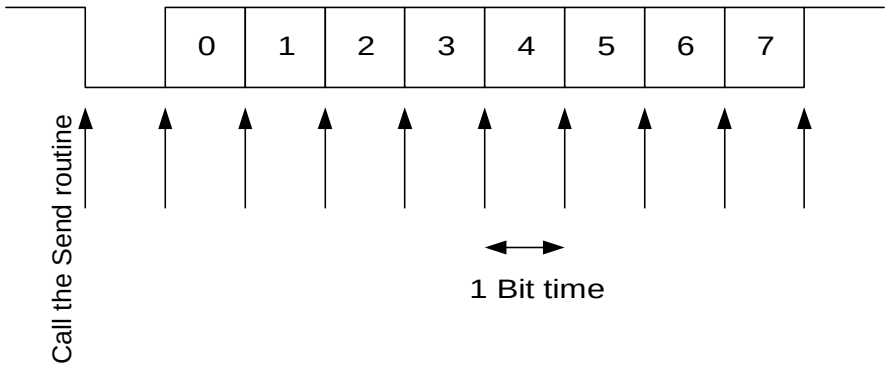
In this implementation we will use the PB.0 I/O port on the ST72611 MCU for TX and the PB.7 for RX. However, the pin could be changed depending on the application. To change TX or RX within Port B, simply change the TX and RX constants in User_Def.h.

3.3.2 Data Transmission

Before transmitting data, interrupts are disabled. In fact, if an interrupt occurs during the transmission, the waiting loop timings will be incorrect.

When the application firmware has to send some data, the *SendSerial* routine is called. The parameter of this routine is the number of bytes to transmit and the endpoint number.

The *SendSerial* routine first generates the Start Bit, then performs 8-bit operations, separated by a timing loop.



```
/* Start bit */
bres PBDR, #TX                          //TX is the RS232 output pin

      ld A, #Val1                             //4 cycles
looptxΔ:
dec A                                         //3 cycles
                  cp A, #0x00                 //2 cycles
                  jrne looptxΔ          //3 cycles
```

So the wait loop duration is (4 + (3 + 2 + 3)***V**al1) clock cycles: As a consequence, Val1 must be defined as follows:

```
Val1= (Osc/Baud-4) / 8
```

Afterward each bit Δ (0 ≤ Δ ≤ 7) of the *Data* variable is tested one by one. If it is '1', the program jumps to the bit Δ label. The *nop* is to have the same execution time for both '1' and '0'.

```
      btjt Data,#Δ,bitΔ         //5 cycles

      //if '0'
      bres PBDR, #TX                    //5 cycles
      jp bit0cnt                              //2 cycles

      //if '1'
      bit01:
                  bset PBDR, #TX    //5 cycles
                  nop                               //2 cycles

      bit0cnt:
```

3.3.3 Data reception

3.3.3.1 Principle

Data reception is based on an interrupt routine triggered by a falling edge on the Rx pin. The falling edge corresponds to the transition "idle to Start bit".

Once started, the interrupt routine performs the following tasks:

– Sample the Rx pin again to check the falling edge was not a noise phenomenon.

– Sample the 1st bit, 1.5 bit times after the falling edge.

– Sample the next bits each bit time, and rebuild the complete byte, bit by bit.

– Check that no new falling edge interrupt has been triggered.

– Store the byte in the Data Buffer, pointed to by Buffer_RX_ptr

– Set a flag if the buffer overflows.

At the end of the interrupt routine, the firmware checks if the data pattern includes a "1 to 0" transition. This is because this transition generates a pending falling edge interrupt request that does not correspond to a Start of bit condition. In order to avoid generating spurious data

reception, a flag is set at the end of the interrupt routine if any “1 to 0” transition occurs during data reception. This flag is read at the start of the interrupt routine, and if it is found to be set, the interrupt routine finishes without any data reception.

3.3.3.2 Sampling the 1st data bit

As the execution of the interrupt routine takes some time to start, the computation of the 1.5 bit times has to take into account this additional delay. This delay can be due to the completion of the current instruction, or to the completion of the whole USB interrupt routine.

The delay has been computed with the following data:

Time needed to enter USB interrupt routine	34 cycles. 4.25µs (Use of REGX options)
Time needed to exit USB interrupt & monitor RX	53 cycles, 6.625µs (Use of REGX options)
Time needed to enter the Falling Edge Interrupt routine	10 cycles, 1.25µs
Time to complete the currently executing instruction	9 cycles, 1.1µs
Execution time of USB interrupt routine	70 cycles 8.75µs

IF...	**THEN...**
The I/O port Falling edge event occurs before or 20µs after the USB Interrupt event. The first instruction of the falling edge IT is executed 2.2µs after (Time to complete the last instruction + time to enter the Interrupt routine).	No Flag set in the USB Interrupt routine
The Falling edge event occurs up to 13µs (4.25+8.75) after the USB Interrupt event. A flag is set at the end of the USB Interrupt routine if the RX line is LOW. The first instruction of the falling edge interrupt routine is executed between 8µs & 21µs after the falling edge. IT MAY MASK THE START BIT AT 57600BPS! * 8µs (6.625+1.25) if the falling edge occurs just at the end of the USB interrupt routine. *21µs (4.25+8.75+6.625+1.25) if the falling edge occurs just after the USB Interrupt event.	Flag set in the USB Interrupt routine. RX sampling for noise immunity is done by monitoring Rx in the USB Interrupt routine.
The Falling edge event occurs between 13µs and 19.6µS after the USB interrupt event. The USB Interrupt does not see Rx at 0, and no flag is set. The first instruction of the falling edge interrupt routine is executed between 1.25µs & 8µs after the falling edge. 1.25µs if the falling edge occurs just when execution returns to the main routine. 8µs (6.625+1.25) if the falling edge occurs just after the end of the USB interrupt routine.	No Flag set in the USB Interrupt routine

In summary:

If no flag is set, the falling edge occurred between 2.2µs and 8µs before the start of the falling edge interrupt routine: an average delay of 1.5 bit times – 4µs is then chosen.

If a flag is set, the falling edge occurred between 8µs and 20µs before the start of the falling edge interrupt routine: an average delay of 1.5 bit times – 15µs is then chosen.

With these values, volume data transfer has been performed successfully at 4800, 9600, 19200, 38400 and 57600 bps.