# 14. Serial communication – RS232

A popular way to transfer commands and data between a personal computer and a microcontroller is the use of standard interface, like the one described by protocols RS232 (older) or USB (newer). This chapter is devoted to communication conforming to RS232 protocol, the hardware for such interface is provided on board. An example will be presented showing the processing of commands received through RS232 interface, and sending of a string of numbers using the same interface.

The protocol RS232 defines the signals used in communication, and the hardware to transfer signals between devices. The time diagram of the typical signal used to transfer character 'A' (ASCII: $65_{10}$ or 0x41) from device A to device B is given in Fig. 1, and would appear on the upper line TX -> RX between devices.

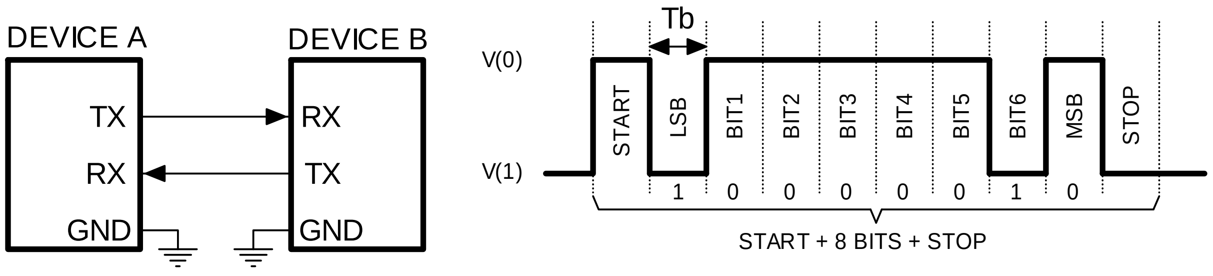

*Figure 1: A signal conforming to RS232 standard*

The standard defines voltage levels V(0) to be at least +5V at the transmitting end of the line TX, and can be degraded along the line to become at least +3V at the receiving and of the line. Similarly voltage level V(1) must be at least -5V at TX, and at least -3V at RX. The standard also defined the upper limit for these voltages to be up to ±15V. Logic high is transferred as V(0). The microcontroller cannot handle such voltage levels, so typically a voltage level translator is inserted between the microcontroller and the connector where the RS232 signals are available. The connectors are typically so-called D9 connectors, and the electric wiring in between two connectors at devices A and B is shown in Fig. 2, for two female type connectors at both devices.

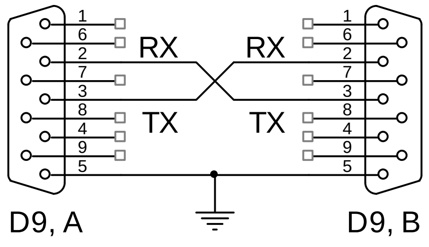

*Figure 2: Wiring for RS232 communication*

The standard defines the number of bits to be transferred within one pack, Fig. 1 right, as eight for regular transmission, and nine for special purposes. The duration *Tb* of each bit defines the speed of transmission and is called the baud-rate. The typical baud-rate is 9600 bits per second (Baud, Bd), and the time *Tb* equals 104.16µs. Other baud rates are also common: 19200 Bd, 38400 Bd, 57600 Bd, and 115200 Bd. Other baud-rates are standardized, but used less frequently. The beginning of the pack of bits is signaled by a so called "START bit", which has value 0 by definition. Its duration is equal to *Tb*. The pack of bits is terminated by so called "STOP bit" with a typical duration of *Tb*, bat can also last either 0.5, 1.5 or 2 *Tb*, depending on the application. The complete transmission of a byte at typical baud rate of 9600 Bd therefore takes 1.0416 ms.

To verify the validity of the transmission the protocol RS232 provides a so called "parity bit". A single bit is added by the transmitter before the stop bit, and its value is selected to give either odd or even number of ones in a string. The number of ones in a string can be verified at the receiving end, and if it does not match the required value, the transmission should be repeated. Other, higher level protocols to ensure the valid transmission, can be implemented in software. The protocol RS232 also accounts for testing if the receiver is capable of receiving incoming bytes and defines two additional wires called RTS (Request To Send) and CTS (Clear To Send) between devices. We will not use either of these options in our experiments.

The microcontroller includes up-to six hardware modules to deal with RS232 signals. Some of the modules additionally implement other communication protocols, like I2C, CAN, SPI; module named UART4 will be used in this experiment. Its detailed description can be found in RM0090, chapter 26. The voltage level translator is added on the test board, and is industry standard chip MAX3232. The signals TX and RX are available at connector P580, pins 3 and 2 respectively. The RS232 signals RX and TX are available as alternate functions replacing the regular port bits, and corresponding port pins must be properly initialized in software.

The software written demonstrates the use of serial communication. The program starts with the initialization, and proceeds into the endless loop, where it periodically checks the status of pushbutton S370. If pressed, the LED D390 is turned on to verify the execution of the regular program. All communication tasks are processed within an interrupt function. The interrupt function is used for a good reason: the RS232 communication is rather slow, and it is a complete waste of processor to let it wait for signals from RS232. Let it rather do more important work, and jump to interrupt function dealing with RS232 only when necessary.

The initialization starts with defining enabling clock for ports A and E, and defining the port pin types. Pin PA06 is used to drive an LED, and pin PE00 is used to check the pushbutton.

The initialization continues with enabling the clock for UART4; the corresponding bit is located in register RCC_AHB1ENR (UART4EN, bit 19, RM0090, page 148). Since the TX and RX signals are mapped as alternate functions to regular port pins, we can find their location and corresponding alternate function in Chip description (DM00037051, page 59). The signals UART4_TX and UART4_RX are available as alternate function AF8 at port C, pins PC10 and PC11 respectively. In order to use port C the clock for this port must first be enabled, and then corresponding pins can be put into alternate function mode by setting bits 21 and 23 in the mode register for port C (GPIOC_MODER). Finally, the correct alternate mode is selected b writing 8 (01000b) into register AFR at positions reserved for pins 10 and 11. This ends the initialization for routing the signals RS232.

The module UART4 needs initialization as well. Baud rate must be selected first, and this is done by writing a constant into register BRR (Baud Rate Register, UART4_BRR). The derivation of the constant is described in RM0090, and several constants for different baud rates are offered as a comment. Next the UART4 module needs to be enabled; this is done by setting bits in control register (UART4_CR1). There is one bit (bit 13, UE) to turn on the complete UART4, and then two additional bits to turn on the transmitting part (bit 3, TE) and receiving part (bit 2, RE). To set all three in one step a 0x200c is OR-ed to the content of register UART4_CR1. Finally, UART4 should be allowed to issue interrupt requests whenever a new byte is received and is available to be read by the software in the microcontroller. This gets allowed by setting a bit (bit 5, RXNEIE, Receiver Not Empty Interrupt Enable) in the same register.

Since the module UART is to issue interrupt requests, the processor should be allowed to respond, and this is done by enabling the interrupt controller NVIC by a call to a function NVIC_EnableIRQ with the argument “UART4_IRQn”.

After this the execution is passed to the endless loop to check the status of the pushbutton and control the LED. The initialization part of the program is given in Fig. 3.

```
#include "stm32f4xx.h"

char *OutString;        // string must be terminated by '\0'

void main(void) {

  // GPIO clock enable, digital pin definitions
  RCC->AHB1ENR  |= 0x00000001;  // Enable clock for GPIOA
  RCC->AHB1ENR  |= 0x00000010;  // Enable clock for GPIOE
  GPIOA->MODER  |= 0x00001000;  // output pin PA06: LED D390

  //UART4 initialization
  RCC->APB1ENR  |= 0x00080000;  // Enable clock for UART4
  RCC->AHB1ENR  |= 0x00000004;  // Enable clock for GPIOC
  GPIOC->MODER  |= 0x00a00000;  // PC10, PC11 => AF mode
  GPIOC->AFR[1] |= 0x00008800;  // select AF8 (UART4,5,6) for PA10, PA11
  UART4->BRR     = 0x1120;      // 9600 baud
  //UART4->BRR     = 0x0890;      // 19200 baud
  //UART4->BRR     = 0x0450;      // 38400 baud
  //UART4->BRR     = 0x02e0;      // 57600 baud
  //UART4->BRR     = 0x016d;      // 115200 baud
  UART4->CR1    |= 0x200c;      // Enable UART for TX, RX
  UART4->CR1    |= 0x0020;      // Enable RX interrupt

  //NVIC init
  NVIC_EnableIRQ(UART4_IRQn);   // Enable IRQ for UART4 in NVIC

  // endless loop
  while (1) {
    if (GPIOE->IDR & 0x0001) GPIOA->ODR |=  0x0040;   // LED on
    else                     GPIOA->ODR &= ~0x0040;   // else LED off
  };
}
```

*Figure 3: A listing of the program to initialize the UART4*

A response to the interrupt request requires an interrupt function. An example of such function to handle interrupts from UART4 is presented in Fig. 4, and should simply be appended to the listing in Fig. 3 before the compilation.

The interrupt function is named as required in the interrupt vector table (UART4_IRQHandler), and neither receives not returns any variables. Any variables used and expected to last longer than the execution of the interrupt function must be declared as global. We are going to send a string of bytes to demonstrate the capabilities of the microcontroller, and such string, actually a character pointer to a string, was already declared in listing, Fig. 3, 3rd line. Such string is expected to be terminated by a ‘\0’, a standard C-language termination character.

There are many possible reasons to interrupt the execution of the regular program. When a new byte is received by the UART4, it appears in the UART4_DR (Data Register), and should be removed from there and handled before it gets overwritten by a next byte received. This is so-called receive interrupt, and was already explained and enabled in listing from Fig. 3. There are even other reasons for UART4 to request attention and issue an interrupt request; those reasons will be dealt with later. However, there is only one interrupt request available for UART4, and the reason an interrupt function was evoked is stored in status register of the UART4 (UART4_SR). Bit RXNE (bit 5, Receiver data register Not Empty) is set when the receipt of a byte causes the interrupt request, and this can be checked within the interrupt function. The body of the interrupt function therefore starts by testing the bit 5; if it is set then the reason for the interrupt is a byte waiting in the UART4 data register, and this byte must be read from there. This gets done in the 6th line of the listing in Fig. 4. This read from the data register simultaneously clears the bit RXNE, and the UART is ready to receive next byte.

Once the received byte is safe the software compares it with some predefined values. If the value is equal to ASCII ‘a’ then an LED connected to port A, bit 6 is turned on. If its value is equal to ASCII ‘b’ then the same LED is turned off. If the value of the byte received is between ASCII ‘A’ and ASCII ‘Z’ then the byte is echoed back to the sender by a simple write back into the data register.

Many times a string of bytes is to be transferred. This possibility is demonstrated next. Since many bytes are to be transferred, this is expected to take some time. Unfortunately, the data register must be written byte by byte only after the previous byte is successfully sent over the UART. The processor time would be wasted if the processor is to wait for UART to transmit a byte, so the interrupt function should be used for transmission as well.

The procedure is as follows. We prepare the string of bytes to be transferred. We should terminate the string with a unique byte as in this example (the C-language automatically terminates a string by ‘\0’ character) or know the length of the string. Then another interrupt request should be enabled, this shall be issued when a byte is fully transmitted and the next one is allowed to be written into the data register; this is done by setting a bit TXEIE in the control register (Transmit data register Empty Interrupt Enable, bit 7, UART4_CR1). The last thing is to write the first byte of the string into the data register sending it through UART4, increment the pointer to the string of bytes and exit the interrupt function. These three steps are implemented as the fourth option when a byte ASCII ‘c’ is received in listing from Fig. 4, RX IRQ part.

Now when the first byte of the string is transferred over the UART4, an interrupt request calls the interrupt function again, but this time bit TXE (Transmit data register Empty) is set signaling next byte can be written into the data register, and the bit RXNE is cleared. Another IF statement is implemented in listing from Fig. 4 to check the status of bit TXE. When this bit is set, the current element of the string checked for the termination byte (‘\0’). If it differs from the termination byte then it is written into the data register, therefore sent through UART4, and the pointer to the string of bytes is incremented. The interrupt request for transmission is left enabled, so the sending of characters can continue. If the current element is equal to the termination character then the interrupts requests for transmission are disabled by clearing the bit TXEIE in control register UART4_CR1.

The program can be checked by the use of HyperTerminal (Windows, pre-7 edition), or one of the freely available terminal emulators.

Some caution should be exercised when such program is debugged. Placing a breakpoint into the interrupt function stops the execution of the program and simultaneously causes a read from registers within the microcontroller by the debugging software. The read includes the data registers of the UART. Some flags may be affected by this read, for instance the flag on TXE and RXNE, causing wrong execution of the interrupt function in step-by-step mode of the debugger! Such traps are difficult to identify and the author of this text spent one afternoon in looking for an error in this program, but there was no error at all! Errors were introduced by misplaced breakpoints, and the program worked fine after the breakpoints were moved to a better place.

```
// IRQ function
void UART4_IRQHandler(void)     {

  // RX IRQ part
  if (UART4->SR & 0x0020) {     // if RXNE flag in SR is on then
    int RXch = UART4->DR;       // save received character & clear flag
    if (RXch == 'a') GPIOA->ODR |=  0x0080;   // if 'a' => LED on
    if (RXch == 'b') GPIOA->ODR &= ~0x0080;   // if 'b' => LED off
    if (RXch >= 'A' && RXch <= 'Z') UART4->DR = RXch;   // echo character
    if (RXch == 'c') {                        // if 'c' => return string
      OutString = "ABCDEFGH";   // Init string & ptr
      UART4->CR1 |= 0x0080;     // Enable TX IRQ
      UART4->DR = *OutString++; // Send first character and increment the pointer
    };
  };

  // TX IRQ part
  if (UART4->SR & 0x0080) {     // If TXE flag in SR is on then
    if (*OutString != '\0')     // if not end of string
      UART4->DR = *OutString++; // send next character and increment pointer
    else                        // else
      UART4->CR1 &= ~0x0080;    // disable TX interrupt
  };

}
```

*Figure 4: A listing of the interrupt function to react to the UART*