14. Serial communication – RS232

The common serial interface can be used to transfer data and commands between microcontrollers or a personal computer and a microcontroller. Here the configuration of the microcontroller to perform serial communication will be demonstrated.

14.1. The protocol RS232 and signals

A popular way to transfer commands or data between a personal computer and a microcontroller is the use of standard serial interface, like the one described by protocols RS232 (older stuff) or USB (newer, more capable). This chapter is devoted to communication conforming to RS232 protocol, the hardware for such interface is provided inside microcontroller STM32F407VG. 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 signals used in communication, and properties of the hardware to transfer signals between devices. There are two signal lines: a TX line is used to output a signal from a device, and the RX line is used to input the signal. There is also a common ground line for both devices, Fig. 14.1 left. The timing 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. 14.1, and would appear on the upper line TX -> RX between devices.

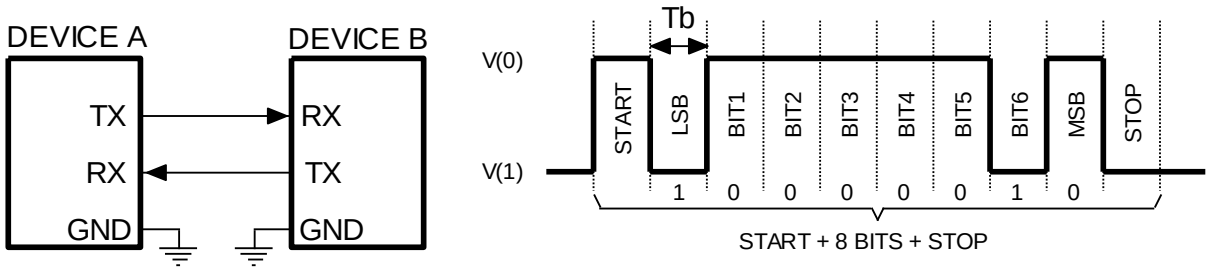

Figure 14.1: A serial communication conforming to RS232 protocol

The standard defines voltage levels V(0) to be at least +5V at the transmitting end of the line TX, and is allowed to degrade 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 microcontroller implements so-called TTL version of RS232 standard.

The standard defines the number of bits to be transferred within one pack, Fig. 14.1 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: 19200 Bd, 38400 Bd, 57600 Bd, and 115200

Bd. These are defined in standard, 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 configuration. The complete transmission of a byte at typical baud rate of 9600 Bd 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 configured to produce 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 any of these options in our experiments.

14.2. The hardware – serial communication

The microcontrollerSTM32F407VG 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 USART3 (Universal Synchronous Asynchronous Receiver Transmitter) will be used in this experiment. Its detailed description can be found in reference manual RM0090, chapter 30. The voltage level translator is added between the PC and the microcontroller since the BaseBoard accepts only TTL version of the RS232 signals TX and RX. The voltage translator can be either connected to the RS232 port of the personal computer or it can derive RS232 signals from the USB port of a personal computer. The signals TX and RX signals are available at connector K600, pins 5 and 4 respectively to ease the connection of the FT USB-RS232 3V3 interface. The RS232 signals RX and TX are available at microprocessor pins as alternate functions replacing the regular port pins, and those must be properly configured.

The block for handling the serial communication is complex, and we will not discuss its options based on the implementations in hardware here. Instead, we will use the CMSIS functions for the configuration of the hardware and will list the proper configuration procedure and comment the use of functions and their arguments.

14.3. The software – serial communication

The software presented demonstrates the use of serial communication. The main part of the program is listed below. The listing starts with the inclusion of CMSIS functions in lines 1 to6; note the include statement in the fourth line for the USART module. A pointer to a character string is declared next, the string will be used for communication purposes. The main function includes three calls to configuration functions for LED diodes on board, four pushbuttons and the LCD display; all these functions were already described in previous chapters and will not be discussed here again. The last configuration function in line 15 deals with the complete configuration of the USART module, and is new here. The execution of the program proceeds into the endless loop, where it periodically checks the state of pushbutton S370. If pressed, the green LED is turned on to verify the continuous execution of the loop.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_gpio.c"
#include "stm32f4xx_usart.c"                                                    // 4
#include "LCD2x16.c"
#include "dd.h"
```

```
char *OutString;        // string must be terminated by '\0'                                    // 8

void main(void) {

  LEDinit();                                                                                   // 12
  SWITCHinit();                                                                                // 13
  LCD_init();                                                                                  // 14
  USART3init(921600);                                                                          // 15

  while (1) {
    if (GPIOE->IDR & S370) LED_GR_ON; else LED_GR_OFF;                                         // 18
    for (int i = 0; i<100000; i++) {};          // waste time                                  // 19
  };
}
```

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 would be a complete waste of processor power to let it wait for signals from RS232. Let it rather do more important work (not in this example), and jump to interrupt function dealing with RS232 only when necessary.

As any other peripheral module in the microcontroller the USART also needs to be configured before used. There are actually three peripherals affected: port D is used to pass the TX and RX signals, the USART module is used to generate and receive RS232 signals, and the controller NVIC is used to process interrupt requests from the USART module. All three must be configured, corresponding function calls to CMSIS functions are put into a configuration function named “USART3init()”. The function requires one argument: the baud rate, and the complete listing of the function is given below.

```
void USART3init(int BaudRate)   {
GPIO_InitTypeDef        GPIO_InitStructure;                                                  // 2
USART_InitTypeDef       USART_InitStructure;                                                 // 3

  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);                                      // 5
  GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_8 | GPIO_Pin_9;                                   // 6
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_AF;                                              // 7
  GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_NOPULL;                                          // 8
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;                                          // 9
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;                                             // 10
  GPIO_Init(GPIOD, &GPIO_InitStructure);                                                     // 11
  GPIO_PinAFConfig(GPIOD, GPIO_PinSource8, GPIO_AF_USART3);                                  // 12
  GPIO_PinAFConfig(GPIOD, GPIO_PinSource9, GPIO_AF_USART3);                                  // 13

  RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART3, ENABLE);                                     // 15
  USART_InitStructure.USART_BaudRate            = BaudRate;                                  // 16
  USART_InitStructure.USART_WordLength          = USART_WordLength_8b;                       // 17
  USART_InitStructure.USART_StopBits            = USART_StopBits_1;                          // 18
  USART_InitStructure.USART_Parity              = USART_Parity_No;                           // 19
  USART_InitStructure.USART_Mode                = USART_Mode_Rx | USART_Mode_Tx;             // 20
  USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;            // 21
  USART_Init(USART3, &USART_InitStructure);                                                  // 22

  USART_ITConfig(USART3, USART_IT_RXNE, ENABLE); // enable IRQ on RX                        // 24
  NVIC_EnableIRQ(USART3_IRQn);   // Enable IRQ for USART3 in NVIC                            // 25

  USART_Cmd(USART3, ENABLE);                                                                 // 27
}
```

Two data structures for two call of CMSIS functions are required, these data structures are declared at the beginning of the function in lines 2 and 3.

The configuration starts with setting-up of the port D in lines 5 to 13 of the listing. The clock for port D is first enabled, then the data structure is initialized to use pins 8 and 9 (in line 6) to serve as alternate functions (in line 7); this data structure is used in line 11 to configure the pins. Next in lines 12 and 13 the correct alternate function is selected by two calls to functions "GPIO_PinConfig()", where the USART3 function is selected.

The configuration continues with setting-up of serial module USART3, lines 15 to 22. The operation of this module is again defined by the content of several configuration registers within the module, and to avoid problems in identifying correct bits in these registers and their function we use the CMSIS function "USART_Init()" instead (line 22). The function is listed in the source file "stm32f4xx_usart.c", and requires the use of a data structure 'USART_InitStructure', declared in the header file "stm32f4xx_usart.h", lines 54 to 81. Brief instruction on the use of USART module are given as usually at the beginning of the source file.

The clock for the USART module is enabled first in line 15. Note that the bit responsible for enabling the clock is not located in the same register as the bit responsible for the clock of ports since the USART module is connected to a different bus within the microcontroller, and the name of the function called differs. The details can be found in reference manual RM0090.

Lines 16 to 21 are used to initialize members of the data structure required in function call in line 22. They are initialized as follows.

- The first member '.USART_BaudRate' has a self-explanatory name. It can be any number, and the CMSIS function will do its best to configure frequency dividers inside the USART module to achieve the baud rate desired. The formula for the calculation is given in the header file, line 56, and the detailed description is to be looked for in the reference manual RM0090. In this example the value is passed into the initialization function as a parameter.
- The second member '.USART_WordLength' defines the number of bits per package sent, and can be either 8 or 9, see the header file, lines 128 and 129 for definitions.
- The third member '.USART_StopBits' defines the duration of the STOP bit, and can be one of the four values listed in the header file, line 141 to 145. Here we select the standard one STOP bit.
- The next member '.Parity' defines the use of the parity bit as one of the options listed in the header file, lines 157 to 159. In this example we do not implement parity checking, so 'USART_Parity_No' is used to initialize this member.
- The fifth member '.USART_Mode' enables the use of the module to receive and/or transmit. In our case both directions are enabled.
- The last member '.USAT_HardwareFlowControl' enables the use of additional signals CTS and RTS, as defined in the header file, lines 181 to 184. We do not use these additional lines, so the keyword 'USART_HardwareFlowControl_None' is used to initialize the member.

The pointer to this data structure is used in line 22 as an argument to a function call. Note that the call configures module USART3, as specified by the first argument.

We have decided to use interrupts to deal with the needs of the USART3 module. The first reason for the interrupt request is obvious: when a module receives a character the processor should remove it from the module and do something with it. In any case there may be more characters coming, and the next one could overwrite the one that was just received, so processor better move it from the USART module fast! On the other hand we do not want to waste valuable processor time by writing a

program that makes processor wait for commands from the USART module or to make processor periodically poll the module for commands.

We must therefore activate the interrupt processing for the incoming characters. This is done in two steps in lines 24 and 25. First the USART module is allowed to issue interrupt request signals, and these interrupt requests should appear due to the reception of a character; the interrupt processing for the RX function is enabled at the module USART3 in line 24. Next the interrupt controller block NVIC is instructed to react on interrupt requests from the USART3 module in line 25. Obviously, there must be an interrupt function prepared in order for interrupt processing to work.

The last line of the configuration function simply enables the USART3 module.

A response to the interrupt request requires an interrupt function. An example of such function to handle interrupts from USART3 is presented next.

```
void USART3_IRQHandler(void)     {

  // RX IRQ part
  if (USART3->SR & USART_SR_RXNE) {   // if RXNE flag in SR is on then                          // 4
    int RXch = USART3->DR;            // save received character & clear flag                  // 5
    LCD_uInt16(RXch, 0x05, 1);        // show ASCII on LCD                                     // 6
    if (RXch == 'a') LED_BL_ON;                                                              // 7
    if (RXch == 'b') LED_BL_OFF;                                                             // 8
    if (RXch >= 'A' && RXch <= 'Z') USART3->DR = ++RXch;  // echo this character             // 9
    if (RXch == 'c') {                // if 'c' => return string                             // 10
      OutString = "ABCDEFGH";         // Init string & ptr                                   // 11
      USART_ITConfig(USART3, USART_IT_TXE, ENABLE);                                          // 12
      USART3->DR = *OutString++;      // Send first character                                // 13
    };
  };

  // TX IRQ part
  if (USART3->SR & USART_SR_TXE) {    // If TXE flag in SR is on then                        // 18
    if (*OutString != '\0')           // if not end of string                                // 19
      USART3->DR = *OutString++;      // send next character & increment pointer             // 20
    else                              // else                                                // 21
      USART_ITConfig(USART3, USART_IT_TXE, DISABLE);                                         // 22
  };

}
```

The interrupt function is named as required in the interrupt vector table (UART3_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. The function is intended to retrieve the received characters from the USART3 module; this is the ‘RX IRQ part’. We are also going to send a string of bytes to demonstrate the capabilities of the microcontroller; such string, actually a pointer to a string, was already declared in the first listing (char *OutString). Such string is expected to be terminated by a ‘\0’, a standard C-language termination character, and the corresponding software is listed in ‘TX IRQ part’.

There are many possible reasons to interrupt the execution of a regular program. When a new byte is received by the UART3, it appears in the USART Data Register (USART3->DR), and should be removed from there and handled before it gets overwritten by a next byte received. This is so-called receive interrupt. There are other reasons for UART3 to request attention and issue an interrupt request; some of them will be dealt with later, they are all listed in the source file, lines 254 to 256 and described in the reference manual RM0090. However, there is only one interrupt request signal available to be

utilized by USART3, and the reason for the interrupt request must be known in the interrupt function. The code of the reason is stored in status register of the USART3. Bit called '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 this bit in line 4; if it is set then the reason for the interrupt is a byte waiting in the USART3 data register, and this byte must be read from there. This gets done in the 5th line of the listing. This read from the data register simultaneously clears the bit RXNE, and the USART3 is ready to receive next byte.

Once the received byte is safely retrieved the interrupt function compares it with some predefined values in lines 7 to 9. If the value is equal to ASCII 'a' then the blue LED 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 of the same USART.

A user program might require 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 in the USART module must be written byte by byte only after the previous byte gets successfully sent over the TX line out of the microcontroller. The processor time would be wasted if the processor is programmed to periodically check whether the module USART is ready to process the next byte, so an interrupt function should be used for transmission as well.

The procedure is as follows. Initially the string of bytes to be transferred is prepared. The string should be terminated with a unique byte as in this example (the C-language automatically terminates a string by '\0' character) to ease the end-of-string detection or the length of the string should be known in advance. Another interrupt request should be enabled, this interrupt request shall be issued when a byte is fully transmitted and the next one is allowed to be written into the data register of the USART; this is done by enabling 'USART_IT_TXE' option, CMSIS function "USART_ITConfig()". The last thing is to write the first byte of the string into the data register USART3->DR sending it out of the microcontroller, increment the pointer to the string of bytes and exit the interrupt function. These three steps are implemented as the fourth option, lines 10 to 14, when a byte ASCII 'c' is received.

Wwhen the first byte of the string is transferred over the USART3, 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. The second IF statement is implemented in listing, line 18, to check the status of bit TXE. When set, the current element of the string is checked against the termination value ('\0'), line 19.

- If the current element of the string differs from the termination value it gets written into the data register, therefore sent through USART3 out of the microcontroller, and the pointer to the string of bytes is incremented. The interrupt request for transmission is left enabled, so the sending of characters can repeat.
- If the current element is equal to the termination value then the interrupt requests for transmission are disabled by a call to the function "USART_ITConfig".

The operation of the program can be verified 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 USART. 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.

Please note that the BaseBoard and the STM32F4-Discovery board accept the TTL version of the RS232 signals. These can be obtained for instance by the use of a USB to TTL-RS232 converter cable, like FT USB-RS232 3V3. Note that the converter must be designed for 3.3V, and that the version for 5V might not work correctly!

Signals available at the RS232 D-9 connector at the back of a personal computer can damage the board! Do not use them for this experiment!
