15. Serial communication – sampling by ADC

Techniques presented in previous chapters will be combined here to connect the microcontroller to a personal computer using the RS232. The program for the microcontroller will initialize USART3 and then wait for a command coming through RS232 connection from the PC. When a suitable command (character 'a') is received the ADC and timer TIM5 will be enabled to start a periodic sampling of two analog input signals. The time interval will be defined by the reload value of the timer, and the predefined number of samples will be stored into an array. The content of the array will be sent through UART to the personal computer when a suitable command (character 'b') is received.

15.1. The software

The hardware used in this example and its configuration was discussed in previous chapters, and will not be repeated here. This chapter describes only the additions to programming and possible modifications to give a usable demo. The chapter does give the new interrupt functions to achieve the goal, and reports the quality of the ADC used.

The listing of the main part of the program is given below. Initially all CMSIS files needed by the compiler are included, then an array named 'Results[]' for storing the results from ADC is declared as global in line 9; it will be used in interrupt function. The array consists of characters and this might look strange. However, characters can be sent using the RS232. The results from ADC are 12 bits in width, so a function for storing results has to split the 12 bits into two sections and store both in consecutive characters of the array. The length of the array is 65536 elements, therefore 32k results from an ADC can be stored. A pointer 'ResultsPointer' to the array of results is needed and is declared and initialized in line 10, also as global.

The main part of the program calls four functions to configure the hardware. These functions are exactly the same as described in previous chapters. Note that a very fast baud rate was selected. This is the fastest standard baud rate on a PC, and can be used only if the wire between the PC and the microcontroller is relatively short, about 1 m is safe to use. Note also that the time interval between two consecutive interrupt requests from the timer is fixed to 100μs, and hardware triggering of the start of conversion is used.

All processing is done in either in hardware or interrupt functions, so the processor enters an empty endless loop after the configuration. Everything is set-up to start the periodic sampling but the timer is not enabled and there is no sampling yet, only the USART3 is ready to receive a byte through RS232. The processor might as well be put into sleep mode conserving power; this is not implemented here.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_adc.c"
#include "stm32f4xx_gpio.c"
#include "stm32f4xx_usart.c"
#include "stm32f4xx_tim.c"
```

```
#include "dd.h"

char Results[65536];                                                                    // 9
int ResultsPointer = 0;                                                                 // 10

void main(void) {
  LEDinit();                                                                            // 13
  USART3_init(921600);                                                                  // 14
  ADCinit_T5_CC1_IRQ();                                                                 // 15
  TIM5init_TimeBase_CC1(840);          // 840 == 100us                                  // 16

  while (1) {                          // endless loop
  };
}
```

The processor keeps on executing an endless empty loop until a character is received by the USART. The essence of the program in hidden in interrupt functions.

When a correct byte is received by the USART3, a sampling should start. The programmer should prepare an interrupt function to be called on an interrupt request from the USART3 as shown already in chapter 14. This function is again composed of two parts, the upper one for handling the receipt of a command character, and the lower one for handling the sending. The complete listing is given next.

```
void USART3_IRQHandler(void)     {

  // RX IRQ part
  if (USART3->SR & USART_SR_RXNE) {  // if RXNE flag in SR is on then
    int RXch = USART3->DR;       // save received character & clear flag
    if (RXch == 'a') {                                                                  // 6
      LED_BL_ON;                // blue ON                                              // 7
      ResultsPointer = 0;       // init pointer to results                              // 8
      TIM_Cmd(TIM5, ENABLE);                                                            // 9
    };
    if (RXch == 'b') {                                                                  // 10
      USART_ITConfig(USART3, USART_IT_TXE, ENABLE);                                     // 11
      ResultsPointer = 0;                                                               // 12
      LED_RD_ON;                // red ON                                               // 13
      USART3->DR = Results[ResultsPointer++];   // Send first character                 // 14
    };
  };

  // TX IRQ part
  if (USART3->SR & USART_SR_TXE) { // If TXE flag in SR is on then                      // 19
    if (ResultsPointer < 65536)    // if not end of string                              // 20
      USART3->DR = Results[ResultsPointer++]; // send next byte & pointer++             // 21
    else {                         // else
      USART_ITConfig(USART3, USART_IT_TXE, DISABLE);                                    // 23
      LED_RD_OFF;                  // red OFF                                           // 24
    };
  };

}
```

If the received character equals ‘a’ then the sampling should start. First the blue LED is turned on to signal the start of sampling in line 7, then the pointer into the array is initialized to point to the zeroth element in line 8, since this is the beginning of the place to store results of sampling. Next the timer TIM5 is enabled, and the execution of this interrupt function is terminated; the processor returns to the dull part: the execution of the endless empty loop. Any further action depends on commands received by the USAT3 or interrupt request from the ADC.

When the content of timer TIM5 becomes equal to 1 its output OC1 triggers the start conversion at both ADCs, see chapter 14. Once results are available the ADC issues its own interrupt request, and this causes the execution of the interrupt function reserved for ADC (this function is named “ADC_IRQHandler”), as given bellow.

```
void ADC_IRQHandler(void)       {
  int Ra = ADC1->DR;                                                                      // 2
  int Rb = ADC2->DR;                                                                      // 3
  if (ResultsPointer < 65535)    {                                                        // 4
    Results[ResultsPointer++] = (Rb & 0x0f00) >> 8;                                       // 5
    Results[ResultsPointer++] = (Rb & 0x00ff);                                            // 6
  }
  else  {
    LED_BL_OFF;                 // blue LED OFF                                           // 9
    TIM_Cmd(TIM5, DISABLE);                                                               // 10
  };
}
```

The results from both ADCs are now available, and they are latched into two temporary variables “Ra” and “Rb”, lines 2 and 3; the latching also clears the interrupt request flag in ADC module to prevent immediate re-entrance into the same interrupt function. One of the results (‘Rb’) is split into two bytes and stored into two consecutive elements of the array in lines 5 and 6. At the end of this saving the pointer ‘ResultsPointer’ points to element number 2 of the array.

A predefined number of samples is to be taken and stored, in our case 32768 (after taking 32768th sample the pointer points to element 65536). If pointer is less than 65535 then the sampling is to continue, and no further actions are needed; this is checked in line 4. The execution of the interrupt function can be terminated, and the processor returns to execute the endless loop until next interrupt request when the action described two paragraphs above is repeated.

However, if the all the required samples were already taken the pointer equals 65536; the sampling should stop. In this case lines 9 and 10 get executed. Here the blue LED is turned off to signal the end of acquisition, and the timer TIM5 is disabled preventing any further start conversion pulses for ADCs. This terminates the sampling phase, and the processor is now waiting for the interrupt requests only from USART3.

When next character is received by the USART3, the interrupt function for the USART3 is entered again. If the character received equals ‘b’, then lines 11 to 14 are executed. These lines start the process of sending results from the array of results through the RS232. The USART3 interrupt request line for the end of transmission is enabled first, then the pointer to the array is initialized to 0. The red LED is turned on in line 13 to signal the start of transmission, and the transmission is commenced in line 14 by sending the first element of the array to the USART3 data register, and the pointer to array gets incremented. This finishes the execution of the interrupt function, and the processor returns to the empty loop.

Once the USART3 is ready to accept the next byte of results it issues an interrupt request again, and the processor re-enters the interrupt function for USART3. The checking of the cause of interrupt in line 19 confirms that the next byte is to be sent. This confirmation is followed by a checking if all elements of the array have been sent in line 20. If pointer is less than 65536 this is not true and the next element is passed to the USART3, and the processor returns to the empty loop. Once all elements are sent the checking in line 20 diverts the execution to line 23, where further interrupt requests for sending are disabled and the red LED is turned off.

This program was used to test the quality of the ADC, and the results are presented in Fig. 15.1 to 15.3. The diagram in Fig. 15.1 shows results obtained when a slowly decreasing signal at about ¼ of the full scale was applied to the input of the ADC. The version presented is idealized to an extent, this is explained in the following paragraph. One can observe the resolution of the ADC: the spread of the results is small, and equals to about ±1.5 LSB. We are therefore dealing with about 10.5 - bit ADC, crudely speaking.

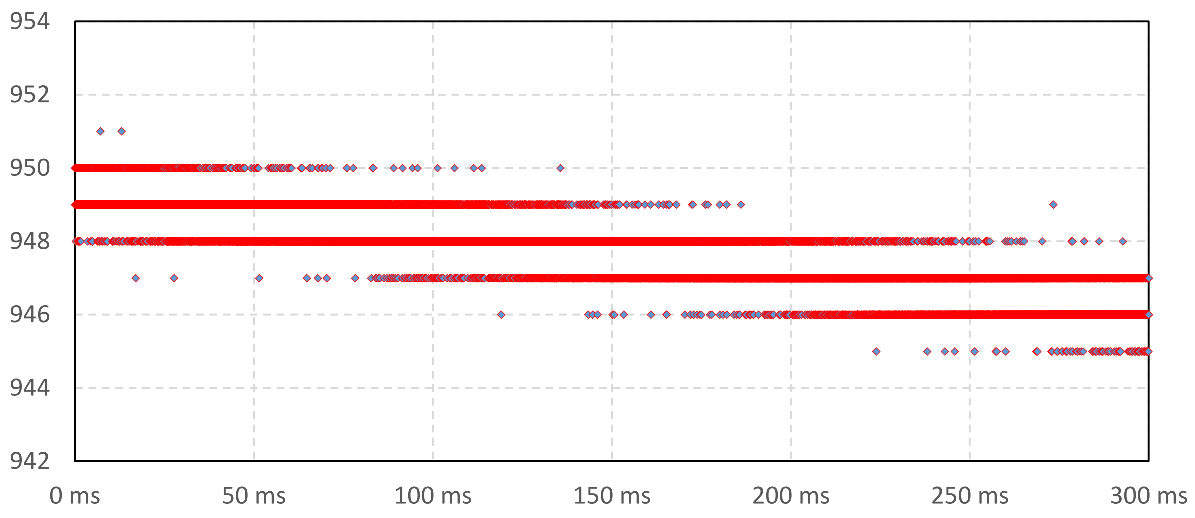

Figure 15.1: A slowly decreasing input signal as sampled by the board and this software; the spread of results is small, all fall within four (sometimes even only three) adjacent values.
Horizontal axis – time, vertical axis – result of measurement as coming from the ADC (0 to 4093)

The next diagram in Fig. 15.2 gives the distribution of samples when a constant voltage of about ¼ of the full scale voltage is applied; again, the spread of the results is small. About 80% of all results (30000 samples were accounted for) have the same value of 971. About 6% of results have value of 970, and about 14% are 972. However, for some reason results far out of these values were also obtained. They are not numerous and account for about 0.16%, and can be best seen on Fig. 15.3, the actual result of sampling, no idealization. The spikes are ascribed to the Discovery board with relatively poor analog layout, but may have other reasons (to be investigated).

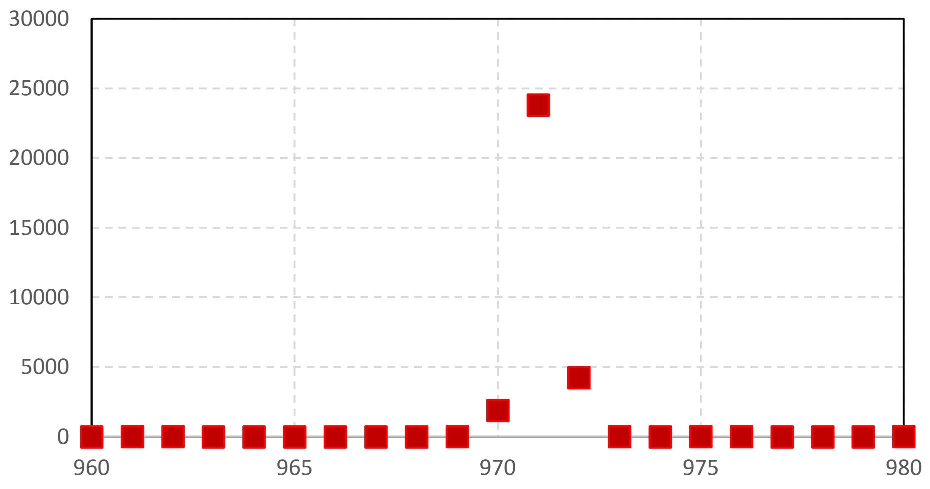

Figure 15.2: The distribution of samples, constant input signal; horizontal axis – measurement result, vertical axes – frequency of occurrence.

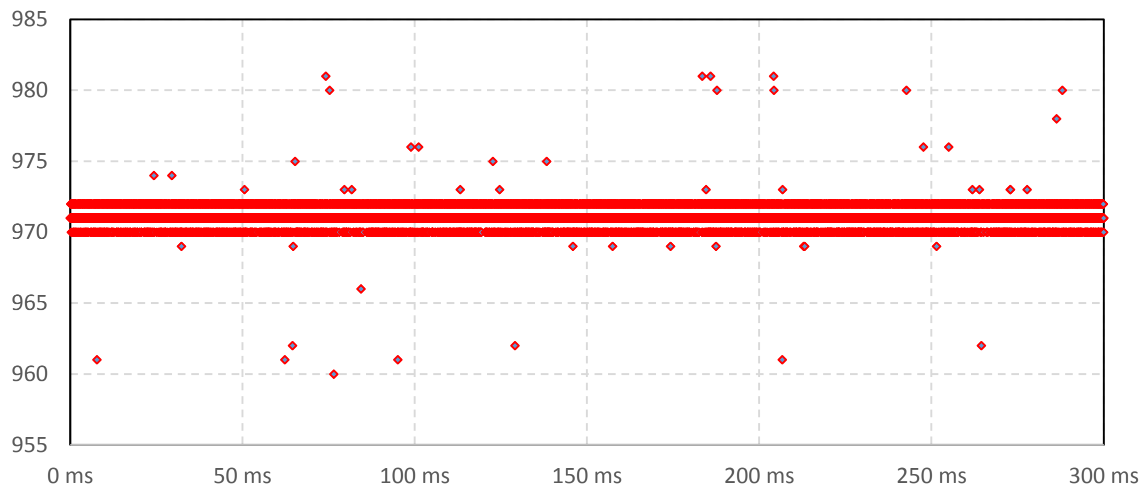

Figure 15.3: A constant signal is sampled, most of the results fall within three channels, but some of them are very far from the expected value. Horizontal axis – time, vertical axis – result.

Next a sinusoidal signal was applied to the input to the board; the signal had a frequency of 5 kHz and amplitude of about 1.5V, biased at about ½ of the ADC range. The last diagram in Fig. 15.4 gives a normalized frequency spectrum of this signal. The spectrum was obtained by commercial software for data processing and presentation (DaDisp & Microsoft EXCEL). Calculation involved the use of Kaiser window function and standard algorithm for the FFT (Fast Fourier Transform). It can be seen that the first harmonic is about 60 dB below the fundamental, and higher harmonics are even lower. It seems a bit strange to have the even multiples harmonics so strong, and this shall be investigated still.

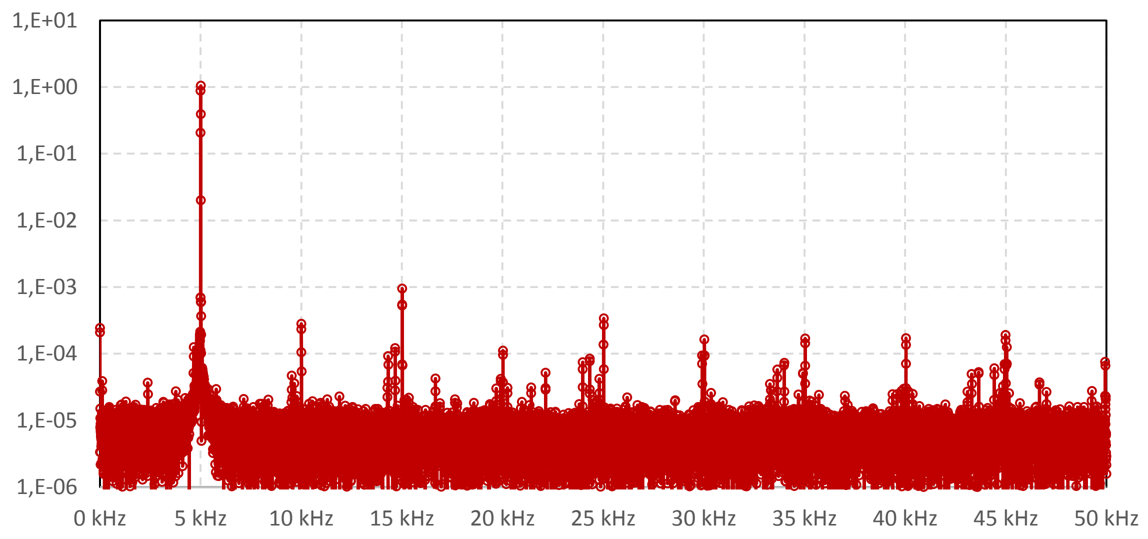

Figure 15.4: The spectrum of the sine wave signal as obtained by the board; horizontal axis – frequency, vertical axis – normalized.