15. Serial communication – sampling by ADC

Techniques presented in previous chapters will be used here to connect the test board to a personal computer using the RS232. The program for the test board will initialize UART4 and then wait for a command from RS232. When a suitable command is received the ADC and timer TIM2 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. Once the sampling is finished the content of the array will be sent through UART to the personal computer.

The hardware used in this example and its initialization was all explained in previous chapters, and explanation on this will not be repeated here. This chapter describes only the additions to programming and possible changes to give a usable product, as well as the differences in interrupt functions.

The array to store results of sampling must be declared as global since it will be used within an interrupt function, and this is done at the beginning of the program. The result obtained for one sample from one ADC is 12 bits in width, and could be stored into an integer variable (32 bits). However, the results of sampling will be sent over the RS232 line, and this requires sending a byte (8 bits) at a time. It may be more efficient to organize the storage space for samples in the way needed for transmission, so the array is declared as a string of characters ("char"). Before storing a result from an ADC will be be split in two halves, upper 4 bits to be stored into even and lower 8 bits to be stored into odd element of the array. There are two ADCs, and the result from the first occupies first two elements of the array, while the result from the second occupies next two elements. It has been decided to take 1024 samples in a row by two ADCs, so we need 4096 characters in the array to save all results. A pointer into the array is needed, and is also declared as global.

The main part of the program is given in Fig. 1 and initializes the hardware and then enters the endless loop. The initialization for ports, UART, and the interrupt request for UART (NVIC) is exactly the same as described in chapter 14 on UART4, and need not be explained again. The initialization of the ADCs and the interrupt request for ADC is exactly the same as described in chapter 12 on the use of ADC/DAC, and will not be re-explained here. Only one is to point out: the version that uses the timer TIM4 to directly start the conversion at the ADC is used. The initialization of the timer TIM2 is copied from the same chapter (chapter 12), but with one change: the timer TIM2 is not enabled to count, one statement (TIM2->CR1 |= 0x0001;) in omitted here. Everything is therefore set-up to start the periodic sampling, but the timer is not enabled and there is no sampling yet. However, the UART is ready to receive a byte through RS232.

The endless loop is the same as usually; a status of the pushbutton S370 is checked and the LED D390 is turned on when appropriate. The essence of the program in hidden in interrupt functions.

```
#include "stm32f4xx.h"

char Results[4096];
int Pointer = 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->CR1    |= 0x200c;      // Enable UART for TX, RX
  UART4->CR1    |= 0x0020;      // Enable RX interrupt

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

  // ADC set-up
  RCC->APB2ENR |= 0x00000100;   // Enable clock for ADC1
  RCC->APB2ENR |= 0x00000200;   // Enable clock for ADC2
  ADC->CCR      = 0x00000006;   // Regular simultaneous mode only
  ADC1->CR2     = 0x00000001;   // ADC1 ON
  ADC1->SQR3    = 0x00000002;   // use PA02 as input
  ADC2->CR2     = 0x00000001;   // ADC1 ON
  ADC2->SQR3    = 0x00000003;   // use PA03 as input
  GPIOA->MODER |= 0x000000f0;   // PA02, PA03 are analog inputs

  ADC1->CR2    |= 0x06000000;   // use TIM2, TRG0 as Start Conversion source
  ADC1->CR2    |= 0x10000000;   // Enable external SC, rising edge
  ADC1->CR1    |= 0x00000020;   // Enable ADC Interrupt for EOC

  // NVIC IRQ enable
  NVIC_EnableIRQ(ADC_IRQn);     // Enable IRQ for ADC in NVIC

  // Timer 2 set-up
  RCC->APB1ENR |= 0x0001;       // Enable clock for Timer 2
  TIM2->ARR     = 8400;         // Auto Reload value: 8400 == 100us
  TIM2->CR2    |= 0x0020;       // select TRGO to be update event (UE)

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

Figure 1: A listing of the main part of the program: initialization and endless loop

When a correct byte is received by the UART4, a sampling should start. There is the interrupt function called on an interrupt request from the UART4, and is called “UART4_IRQHandler”, as shown in chapter 14. This function is composed of two parts, one for handling the receipt of a byte, and one for handling the sending. The complete listing is given in Fig. 2.

If the received byte equals ‘a’ then the sampling should start. First the pointer into the array is initialized to point to the first element of the array, since this is the place to store the first result of

```c
// IRQ function - UART
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') {
      Pointer = 0;              // init pointer to results
      TIM2->CR1    |= 0x0001;   // Enable Counting
    };
  };

  // TX IRQ part
  if (UART4->SR & 0x0080) {     // If TXE flag in SR is on then
    if (Pointer < 4096)         // if not end of string
      UART4->DR = Results[Pointer++]; // send next byte and increment pointer
    else {                      // else
      UART4->CR1 &= ~0x0080;    // disable TX interrupt
    };
  };

}
```

Figure 2: A listing of the interrupt function for UART4

sampling. Next the timer TIM2 is enabled to count, then the execution of this interrupt function is terminated. The timer TIM2 issues a timer reload event when a predefined number is reached by the counter within the TIM2, and the ADCs sample input signals. Once the results of sampling are available the ADC issues its own interrupt request, and this transfers the execution of the program to the interrupt function reserved for ADC (this function is properly named “ADC_IRQHandler”), as given in Fig. 3.

Since both first results from both ADCs are now available they should be stored into the array. They are latched into two temporary variables “Ra” and “Rb”; the latching also clears the interrupt request flag to prevent immediate re-entrance into the same interrupt function. Both results are split as described before and stored into four consecutive elements of the array. At the end of this saving the pointer points to element number 4.

A predefined number of samples is to be taken and stored, in our case 1024 (after taking 1024th sample the pointer points to element 4096). If pointer is less than 4096 then the sampling is to continue, and no further actions are needed. The execution of the interrupt function can be

```c
// IRQ function - ADC
void ADC_IRQHandler(void)       // PASS takes approx 400ns of CPU time!
{
  int Ra = ADC1->DR;
  int Rb = ADC2->DR;
  Results[Pointer++] = (Ra & 0x0f00) >> 8;
  Results[Pointer++] = (Ra & 0x00ff);
  Results[Pointer++] = (Rb & 0x0f00) >> 8;
  Results[Pointer++] = (Rb & 0x00ff);
  if (Pointer >= 4096)    {
    TIM2->CR1    &= ~0x0001;    // Disable Counting
    UART4->CR1   |= 0x0080;     // Enable TX IRQ
    Pointer = 0;
    UART4->DR = Results[Pointer++];   // Send first character
  };
}
```

Figure 3: A listing of the interrupt function for ADC

terminated, and the execution continues in 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 content equals 4096: the sampling should stop, and the acquired results should be sent to the PC. In this case the body of the "If" statement within the interrupt function "ADC_IRQHandler" gets executed. Here the counter is first disabled preventing any further start conversion pulses for ADCs. Next the interrupt request for transmission through UART4 is enabled, pointer to the array is initialized again, and the first element of the array is passed to the data register within the UART4. This initiates the transmission of the complete array, as already described in chapter 14.

There is one small difference in sending complete string from the example in chapter 14. Here we know how many bytes to transfer (4096), so instead of testing the value of the byte we test the value of the pointer to the array, see the second part of the interrupt function for UART, Fig. 2. If pointer value is less than 4096 then there are still bytes to be sent, otherwise the transmission gets stopped by clearing the TX interrupt enable flag in UART4.

This program was also used to test the quality of the ADC, and the results are presented in Fig. 4 to 6. The diagram in Fig. 4 shows results obtained when a slowly decreasing signal at about 1/6 of the full scale was applied at the input of the ADC. One can observe the resolution of the ADC: the spread of the results is small, and equals about ±1 LSB. We are therefore dealing with at about 11-bit ADC.

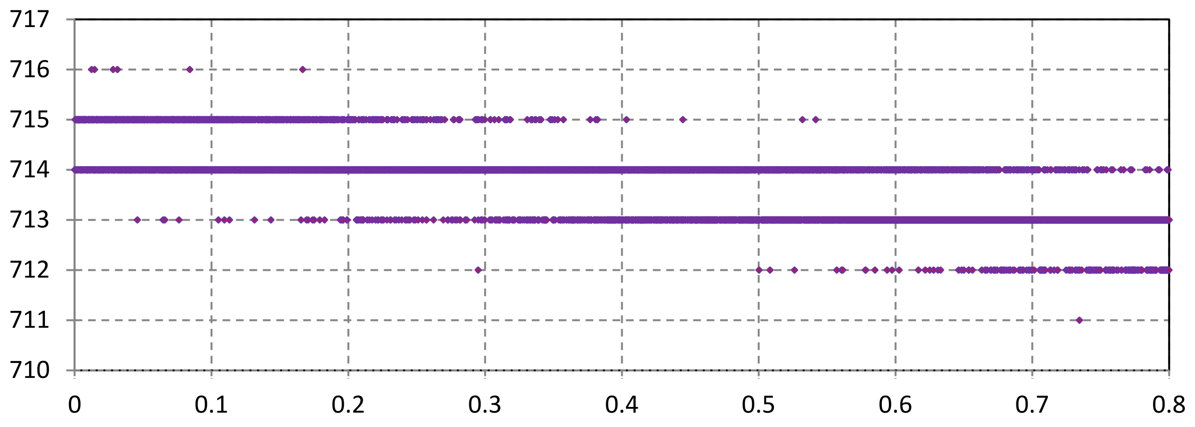

Figure 4: A slowly decreasing input signal as sampled by the board and this software; the spread of results is small, all fall within three (sometimes even only two) adjacent values.
Horizontal axis – time in seconds, vertical axis – result of measurement as coming from the ADC (0 to 4093)

The next diagram in Fig. 5 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 (8196 samples were taken) have the same value of 958. About 15% of results have value of 957, and about 5% are 959.

Next a sinusoidal signal was applied to the input to the board; the signal had a frequency of 500 Hz and amplitude of about 1.5V, biased at about ½ of the full scale. The last diagram in Fig. 6 gives a normalized frequency spectrum of this signal obtained by slightly modified software. The sampling frequency was 10 kHz, and 30000 samples were taken and transferred to PC, where standard software for data processing and presentation (Microsoft EXCEL & DaDisp) calculated the spectrum using a Kaiser window function and standard algorithm for the FFT (Fast Fourier Transform). It can be seen that the first harmonic is about 70 dB below the fundamental, and higher harmonics are even less. It seems a bit strange to have the harmonic at 1000 Hz so strong, and this shall be investigated still.

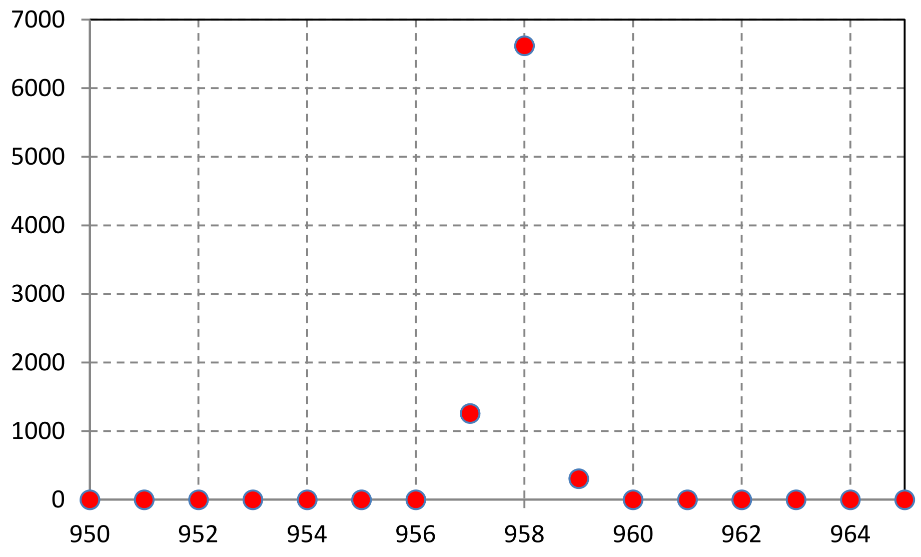

Figure 5: The distribution of samples, constant input signal; horizontal axis – measurement result, vertical axes – frequency of occurrence. Results beside 957, 958 and 959 never occured

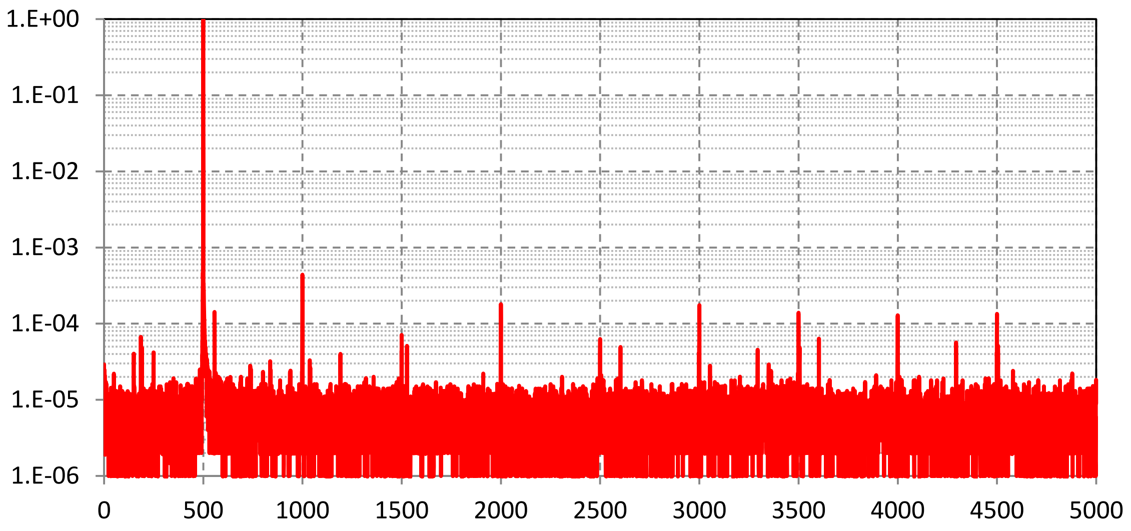

Figure 6: The spectrum of the sinewave signal as obtained by the board; horizontal axis – frequency in Hz, vertical axis – normalized.