8. Analog to digital converter

The use of an ADC is demonstrated.

8.1. Hardware – a ADC block

There are three 12-bit analog to digital converter (ADC) blocks available in STM32F407VG microcontroller. The conversion time for each ADC block is about 1µs. There is a rich variety of possible ADC configurations, and here we demonstrate the simplest one to measure two voltages simultaneously using two ADCs.

A simplified block diagram for one ADC block is given in Fig. 8.1 (excerpt from Fig. 44, RM0090, page 387). The complete ADC block includes two more ADCs with similar features and additional settings.

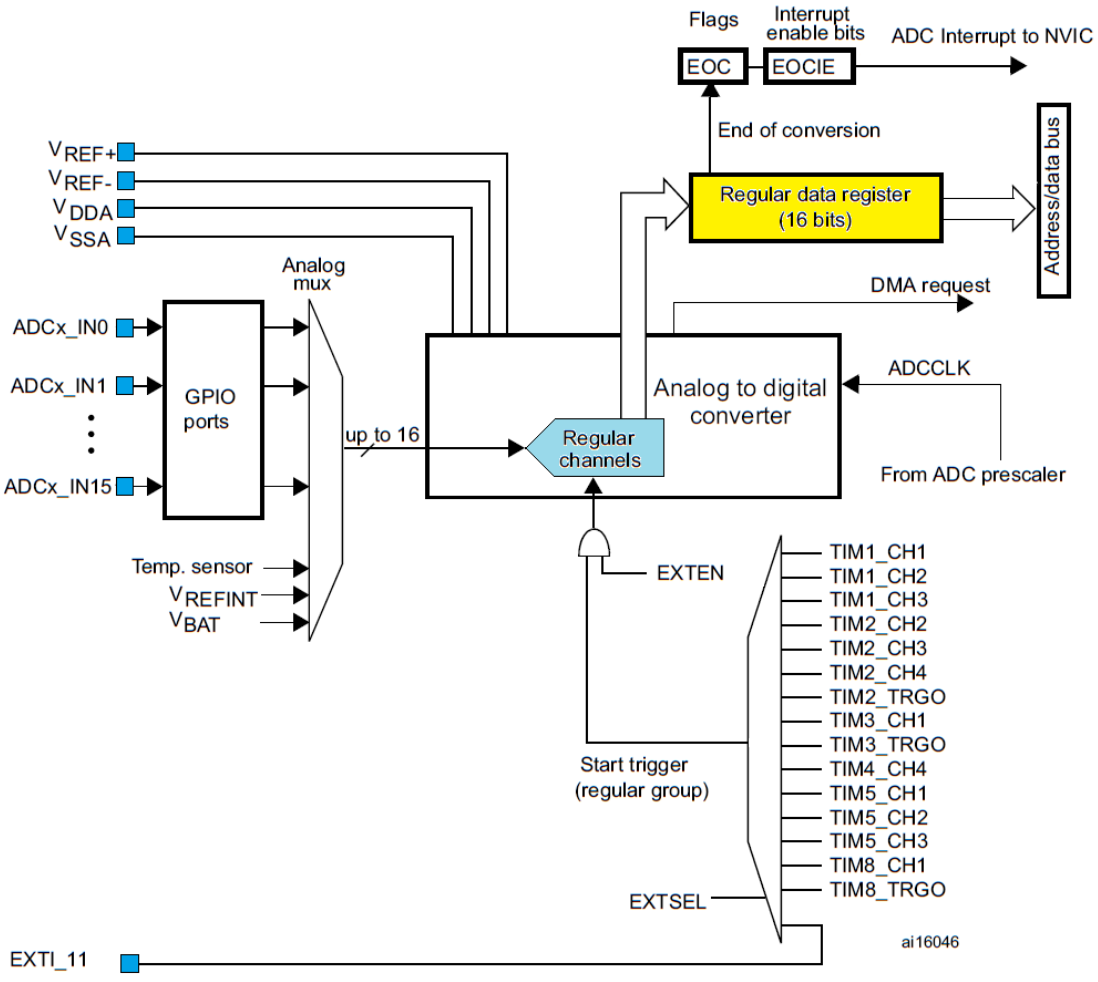

Figure 8.1: The block diagram of a single ADC

The actual ADC is located in the center of the figure. Its operation is supported by power supply lines V_{DDA} and V_{SSA}, and the reference lines V_{REF+} and V_{REF-}, all coming from pins of the microcontroller. Additionally, a clock signal ADCCLK is required, and is generated in the microcontroller. The analog input signals are connected to the designated microcontroller pins ADCx_INxx and are fed to the ADC through a multiplexer named Analog mux. There are altogether 16 pins available to connect analog signals to, and the multiplexer houses additional three inputs to allow the measurement of the chip temperature, power and reference lines. Appropriate input pin is selected by setting bits (five of them) in one of the ADC registers (ADCx_SQR3, if one wants to do it manually). It should be noted that analog signals do not mix well with digital signals, and that the pin used for input must be configured as analog.

The ADC writes the result of conversion into a Regular data register (ADCx_DR). Here x stands for the index of the ADC used (1, 2 or 3).

The ADC block includes hardware to support up to 16 consecutive conversions at pins declared in advance without the assistance from the microcontroller; when one intends to implement consecutive conversions the number of consecutive conversions and the input pins used must be stated before initiating the conversion. However, there is only one register to hold a result of conversion, so a user program must take care of moving consecutive results from the Regular data register before next conversion is finished or the previous result will be lost. The DMA (direct memory access) hardware can be utilized to move results in time.

8.2. Software to test the ADC

The ADC block can be configured by manipulating bits in designated registers of the ADC block. However, this requires a detailed knowledge on the hardware of the ADC, and it is much simpler to use functions available in CMSIS library. The functions that actually manipulate bits are stored in the source file “stm32f4_adc.c”, and definitions and data structures to accompany these functions are stored in the header file “stm32f4x_adc.h”. The first of them must be included from the user program.

A brief recipe for the use of two or more ADCs is again given at the beginning of the source file:

- Configure pins that will be used to input analog signals. They must be designated as analog pins, there must be no terminating resistors.
- Enable clock for ADCs used.
- Configure the control circuitry valid for all three ADCs within the STM32F407VG, use function “ADC_CommonInit()” and the related data structure.
- Configure the use of regular channels for every ADC individually using function “ADC_Init()” and the related data structure. Connect ADCs with input signals.
- Enable ADC(s) using function “ADC_Cmd()”.

The result of conversion can be obtained by reading the content of register ADCx->DR, where x stands for either 1, 2 or 3. Alternativelly, and to enhance portability of the program, a CMSIS function “ADC_GetConversionValue()” can be used.

The recipe for configuration of two ADCs to sample two input signals simultaneously is implemented in the function and listed below.

```
Void ADCinit_SoftTrigger(void)  {
              // 1
GPIO_InitTypeDef        GPIO_InitStructure;                                          // 2
ADC_InitTypeDef         ADC_InitStructure;                                           // 3
ADC_CommonInitTypeDef   ADC_CommonInitStructure;                                     // 4

  RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1 | RCC_APB2Periph_ADC2,  ENABLE);         // 6
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);                              // 7
```

```
  GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_1 | GPIO_Pin_2;                                  // 9
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_AN;                                            // 10
  GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_NOPULL;                                        // 11
  GPIO_Init(GPIOA, &GPIO_InitStructure);                                                   // 12

  ADC_CommonInitStructure.ADC_Mode             = ADC_DualMode_RegSimult;                   // 14
  ADC_CommonInitStructure.ADC_Prescaler        = ADC_Prescaler_Div2;                       // 15
  ADC_CommonInitStructure.ADC_DMAAccessMode    = ADC_DMAAccessMode_Disabled;               // 16
  ADC_CommonInit(&ADC_CommonInitStructure);                                                // 17

  ADC_InitStructure.ADC_Resolution             = ADC_Resolution_12b;                       // 19
  ADC_InitStructure.ADC_ScanConvMode           = DISABLE;                                  // 20
  ADC_InitStructure.ADC_ContinuousConvMode     = DISABLE;                                  // 21
  ADC_InitStructure.ADC_ExternalTrigConvEdge   = ADC_ExternalTrigConvEdge_None;            // 22
  ADC_InitStructure.ADC_ExternalTrigConv       = ADC_ExternalTrigConv_T1_CC1;              // 23
  ADC_InitStructure.ADC_DataAlign              = ADC_DataAlign_Right;                      // 24
  ADC_InitStructure.ADC_NbrOfConversion        = 1;                                        // 25
  ADC_Init(ADC1, &ADC_InitStructure);                                                      // 26
  ADC_RegularChannelConfig(ADC1, ADC_Channel_1, 1, ADC_SampleTime_3Cycles);                // 27

  ADC_Init(ADC2, &ADC_InitStructure);                                                      // 29
  ADC_RegularChannelConfig(ADC2, ADC_Channel_2, 1, ADC_SampleTime_3Cycles);                // 30

  ADC_Cmd(ADC1, ENABLE);                                                                   // 32
  ADC_Cmd(ADC2, ENABLE);                                                                   // 33
}
```

The function starts with the declaration of data structures, three are required for three different function calls. Lines 6 and 7 hold function calls to enable the clock signals for ADC1, ADC2 and port A.

Analog voltages to be measured are connected to pins 1 and 2 of port A, see schematic diagram of the BaseBoard for available pins. These two pins must be configured as analog without terminating resistors, so lines 9 to 11 initialize members of the data structure 'GPIO_InitStructure' to correct values, and line 12 calls the function to actually configure the port A.

The hardware that is common to all three ADCs is configured next, this is done in line 17 with a call to a function "ADC_CommonInit()", the function itself is located in the source file "stm32f4xx_adc.c". This function requires a pointer to the data structure 'ADC_CommonInitStructure' as the only argument, and the structure itself is initialized in lines 14 to 16. There are four members of this structure.

- The first member '.ADC_Mode' defines how the operation of the three ADC is synchronized. Their operation can be independent ('ADC_Mode_Independant'), and each ADC is started independently of others. Their operation can be simultaneous ('ADC_TripleMode_RegSimult'), and all three ADCs are started using a single trigger event. Their operation can be interleaved ('.ADC_TripleMode_Interl'), and each trigger event starts the first ADC, the other two follow after a fixed time interval, one after another. There are similar options for the use of two ADCs, we have selected the option '.ADC_DualMode_RegSimult': two ADCs are used, and they are started simultaneously on the trigger event. Other options can be found in the header file "stm32f4xx_adc.h", lines 115 to 127, and are described in reference manual RM0090.
- The second member '.ADC_Prescaler' selects the clock frequency ADCCLK. This clock determines the conversion time, and is not the same as the clock enabled in line 6 of the function, see the reference manual RM0090, pg. 388, for the difference. The clock ADCCLK is common for all ADCs, and can be either ½, ¼, 1/8 or 1/16 of the microprocessor internal clock f_{PCLK} frequency, which equals to 42 MHz in our case. Since the ADC can use clock frequency up

to 30 MHz it is safe to use the least division factor of ½. The correct option is then 'ADC_Prescaler_Div2', others can be found in the header file, lines 149 to 152.

- The third member '.ADC_DMAAccessMode' defines the use of DMA hardware to move results of conversion from the Regular data register into a reserved part of the microprocessor RAM. We are not going to use this option, so the correct keyword is 'ADC_DMAAccessMode_Disabled'. Other options are listed in header file, lines 165 to 168, and the explanation is given in the reference manual RM0090, chapter 13.
- The fourth member defines the time interval between consecutive starts of ADCs if the interleaved mode is selected. We do not use this mode, so this member need not be defined.

Settings for each ADC individually are configured next. We are going to use two ADCs, and both should work the same way; we therefore call the function "ADC_Init" to initialize the ADC two times in lines 26 and 29, once for 'ADC1' and once for 'ADC2'. The function requires a pointer to the data structure 'ADC_InitStructure', and parameters for the two calls are the same. The data structure 'ADC_InitStructure' has seven members, they are initialized in lines 19 to 25.

- An ADC can operate with different resolutions; lower resolutions allow faster conversion. For this example the best resolution of 12 bits is required, and the corresponding conversion time is less than 1μs. This is selected by initializing the member 'ADC_Resolution' with 'ADC_Resolution_12b'. Other options can be found in the header file, lines 223 to 226.
- An ADC starts a measurement on a trigger event. Within the measurement it can sample one single signal connected to one pin, or it can scan a predefined set of signals connected to several pins. We need to measure a single signal for each trigger event, and this is achieved by disabling the scanning option in line 20.
- An ADC can start a measurement on trigger event and then wait for next trigger event, or can continuously repeat the measurement without repeating the trigger event. We want a single conversion, and this is achieved by disabling the continuous conversion mode in line 21.
- The fourth member '.ADC_ExternalTrigConvEdge' configures the edge of the signal to trigger the conversion. The edge can be rising, falling, both of them, or none. We want to start a conversion using a software command here, so option 'ADC_ExternalTrigConvEdge_None' is used in line 22.
- The fifth member '.ExternalTrigConv' selects one of the 16 signals to be used as trigger event by configuring the multiplexor bellow the ADC, Fig. 8.1. All possible options are listed in the header file, lines 256 to 271. It is not necessary to initialize this member in this example since we disabled the external trigger in line 22 and by this selected the software trigger, but the line 23 remains there for reference.
- The sixth member '.ADC_DataAlign' determines the alignment of the result in the Regular data register; it can be either left or right. The right alignment is selected by 'ADC_DataAlign_Right'.
- The seventh member 'ADC_NbrOfConversion' determines the number of consecutive conversions to be performed by one ADC within one measurement. Since we do not want to scan several analog signals, this member is initialized to 1.

Using this data structure two ADCs are initialized in lines 26 and 29.

The ADCs need to be connected through the multiplexor with sources of analog signals. These are pins 1 and 2 of port A, designated as ADC_Channel_1 and ADC_Channel _2, see the pinout table in the datasheet DM00037051, page 47. This connection can be achieved by setting of several bits residing in ADC control registers, which is best achieved by a call to a CMSIS function "ADC_RegularChannelConfig()", its brief description can be found in the source file, line 636 and on. The function requires 4 arguments.

- The first argument is the name of the ADC to be configured, can be ADC1, ADC2 or ADC3.

- The second argument is used to put the multiplexor in front of the ADC in correct state, and simply declares the number of channel to be measured. It can be any value from 'ADC_Channel_0' to 'ADC_Channel_18'.
- The third argument specifies the order of conversion if a sequence of conversions is initiated. This example does not use the sequenced conversions, and this argument is set to 1 since this is to be the first (and only) conversion.
- The fourth parameter defines the sampling time of the ADC, and can be minimum 3 clock cycles ('ADC_SampleTime_3Cycles') and maximum 480 clock cycles ('ADC_SampleTime_480Cycles'). All possible values are listed in the header file, lines 359 to 366.

Using these arguments the two lines 27 and 30 for individual configuration of ADCs are written.

The last thing is to enable ADCs; this is achieved by two calls in lines 32 and 33, the applicable CMSIS functions are "ADC_Cmd(ADCx, ENABLE)", where x stands for either 1 or 2.

The demo program has been prepared, and its listing is given bellow. The program uses ADC1 and ADC2 to measure voltages at pins 1 and 2, port A. The listing starts with the inclusion of necessary files, note the "stm32f4xx_adc.c", and the executable part starts with the initialization of both ADCs by calling the function "ADCinit()". This is followed by the initialization of the LCD screen, and writing of the introductory strings to the LCD.

The endless 'while' loop is executed next. Within the loop the conversion on ADC1 is initiated by calling the CMSIS function "ADC_SoftwareStartConv(ADC1)", and since the two ADCs were configured as simultaneous, this also starts the conversion on ADC2. Then some time is wasted allowing ADCs to finish the conversion, and both results of conversions are read from two Regular data registers of both ADCs and displayed. Please note that the reading of results is done by a direct access to the hardware; the name of the regular data register is stated as 'ADC1->DR' and 'ADC2->DR', which does not promote porting or CMSIS. There is a CMSIS function to retrieve the result available, its name is "ADC_GetConversionValue()", but it is not used here for simplicity and the speed of operation.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_adc.c"
#include "stm32f4xx_gpio.c"
#include "LCD2x16.c"

int main ()  {
unsigned int j;

  ADCinit_SoftTrigger();

  LCD_init();
  LCD_string("ADC1=", 0);
  LCD_string("ADC2=", 0x40);

  while (1) {
    ADC_SoftwareStartConv(ADC1);
    for (j = 0; j<10000000; j++){};      // waste some time
    LCD_uInt16((int)ADC1->DR, 0x06, 1);  // write result on LCD, 1st line
    LCD_uInt16((int)ADC2->DR, 0x46, 1);  // write result on LCD, 2nd line
  };
}
```