12. Periodical interrupts and ADC/DAC

The knowledge from previous chapters will be used here to prepare a program which can periodically start a conversion at the ADC, wait for the result and pass the result to the DAC.

12.1. The hardware – idea on sampling and generating

The previously given demonstration program for ADC shows the initialization needed to use two of the built-in ADCs to sample two analog input signals. The results were displayed at the LCD. Here the results will be passed to two DACs generating a replica of the input signal, but quantized in time and value. The picture in Fig. 12.1 shows one of the input signals and the corresponding quantized version at the output from a DAC. Such program could also be prepared by combining former examples on the use of ADC and DAC, and inserting the relevant code for start conversion at the ADC, waiting of the conversion result, and sending the result to DAC in a loop. An additional empty loop could define the time delay between two consecutive iterations of the loop and therefore the sampling frequency. However, using the software to define the time interval is wasting of the processor time, and should be avoided. A additional and much better reason not to use this technique will become evident in the following chapters: we usually want to process the signal before passing to the DAC, and the processing time may vary on the nature of processing.

Figure 12.1: The quantized-in-time version of the input signal

It has been demonstrated in chapter 11 that time intervals can be defined using a timer, and that the reload events of a timer can trigger interrupts. It is therefore only natural to define time intervals between two successive conversions by a timer, and therefore use interrupt function to start the conversion at ADC by issuing the pulse Start Conversion SC, wait for the result, and then pass the result to DAC, as depicted in Fig. 12.2.

12.2. The software – implementation of the idea

The listing of the program to implement the idea above is simple and based on previous three chapters 7, 8, and 11. The program from chapter 11 defines time intervals. A software command is used to start the conversion as demonstrated in program from chapter 7, but this time the command

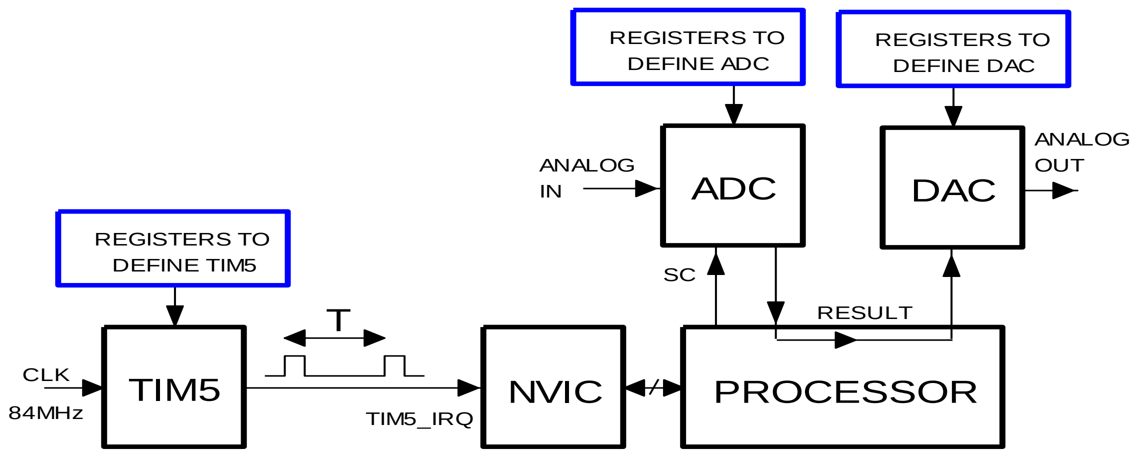

Figure 12.2: The natural operation for periodical sampling and generating of analog signals

is issued from the interrupt function, and therefore issued periodically. The result of conversion is passed to the DAC as demonstrated in chapter 8, still inside the interrupt function. Three hardware blocks require configuration, these are the ADC, the DAC, and the timer. All configuration functions ("ADCinit_SoftTrigger()", "DACinit()", and "TIM5init_TimeBase_ReloadIRQ()") are copied from the relevant chapters. The listing of the program is given bellow.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_adc.c"
#include "stm32f4xx_dac.c"
#include "stm32f4xx_gpio.c"
#include "stm32f4xx_tim.c"

int main ()  {

  ADCinit_SoftTrigger();                                                      // 10
  DACinit();                                                                  // 11
  TIM5init_TimeBase_ReloadIRQ(840);                  // 840 == 10us           // 12

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

void TIM5_IRQHandler(void)       {                                            // 18
  TIM_ClearITPendingBit(TIM5, TIM_IT_Update);        // clear interrupt flag
  DAC->DHR12R1 = ADC1->DR;                                                    // 20
  DAC->DHR12R2 = ADC2->DR;
  ADC_SoftwareStartConv(ADC1);                                                // 22
}
```

The program starts with a set of include statements to gather all CMSIS functions into the user source file. The main part of the program commences with three calls to configuration functions, note that here the argument to define the time interval between two consecutive timer interrupts is reduced to 840 (10μs). The processor enters the infinite empty loop after the configuration, and simply waits for the interrupt request.

The interrupt function starts at line 18. The first thing to do here is to clear the flag in timer TIM5 to prevent a repeated execution of the interrupt function. In order to expedite the execution of the function the order of actions as stated above is reversed. Line 20 copies the current content of the ADC1 into the DAC1, and the line 21 does the same for ADC2 and DAC2. Next in line 22 the ADC conversion is started. It will be finished well before the next repetition of the interrupt function (within 10µs), and the results of this conversion will be waiting then to be copied to DACs.

Such implementation is straightforward, but may not be the best. The digital signal processing theory states that it is very important to take samples at exact time intervals, or the signal to noise ratio will be impaired. The exact period is not guaranteed in the former program since the processor can start executing the interrupt function only after it finishes the execution of the current instruction, and instructions in complex programs can take different time to execute. Furthermore, the execution of the interrupt function for timer TIM5 itself might be delayed in more complex programs utilizing more than one interrupt function if the interrupt request for timer TIM5 is assigned low priority.

12.3. The software – a better implementation

To avoid these problems we should trigger the start of conversion SC directly by a timer TIM5 without the intervention of the software, and use interrupt function only to copy the result of conversion to the DAC. Such interrupt function shall be initiated by the end of conversion signal EOC coming from the ADC, which can also be used to trigger an interrupt at ADC_IRQ as depicted in Fig. 12.3.

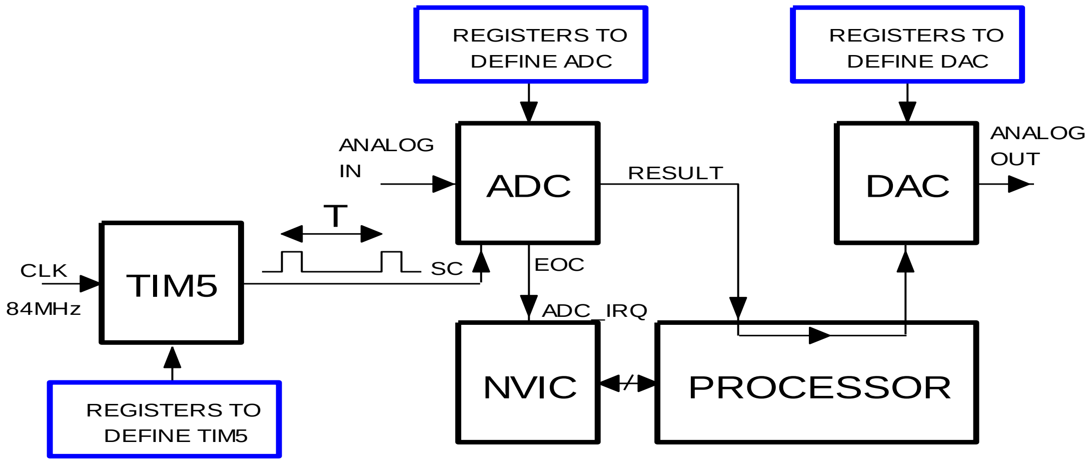

Figure 12.3: The start of conversion can also be triggered by timer TIM5, and the end of conversion signal EOC is used to trigger an interrupt. The interrupt function moves the result of conversion from ADC to DAC.

Few things need to be changed in the configuration section for the ADC, the timer and the interrupt function itself. The configuration section for the DAC remains the same for now.

We start with the configuration of the ADC, the new function is called "ADCinit_T5_CC1_IRQ()", and is listed below.

```
void ADCinit_T5_CC1_IRQ (void)      {
ADC_InitTypeDef         ADC_InitStructure;
GPIO_InitTypeDef        GPIO_InitStructure;
ADC_CommonInitTypeDef   ADC_CommonInitStructure;

  RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1  | RCC_APB2Periph_ADC2,  ENABLE);
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA | RCC_AHB1Periph_GPIOB, ENABLE);

  GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_1 | GPIO_Pin_2;
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_AN;
  GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_NOPULL;
  GPIO_Init(GPIOA, &GPIO_InitStructure);

  ADC_CommonInitStructure.ADC_Mode            = ADC_DualMode_RegSimult;
  ADC_CommonInitStructure.ADC_Prescaler        = ADC_Prescaler_Div2;
  ADC_CommonInitStructure.ADC_DMAAccessMode    = ADC_DMAAccessMode_Disabled;
  ADC_CommonInit(&ADC_CommonInitStructure);

  ADC_InitStructure.ADC_Resolution             = ADC_Resolution_12b;
  ADC_InitStructure.ADC_ScanConvMode           = DISABLE;
  ADC_InitStructure.ADC_ContinuousConvMode     = DISABLE;
  ADC_InitStructure.ADC_ExternalTrigConvEdge   = ADC_ExternalTrigConvEdge_Rising;              // 22
  ADC_InitStructure.ADC_ExternalTrigConv       = ADC_ExternalTrigConv_T5_CC1;                  // 23
  ADC_InitStructure.ADC_DataAlign              = ADC_DataAlign_Right;
  ADC_InitStructure.ADC_NbrOfConversion        = 1;
  ADC_Init(ADC1, &ADC_InitStructure);
  ADC_RegularChannelConfig(ADC1, ADC_Channel_1, 1, ADC_SampleTime_3Cycles);

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

  ADC_Cmd(ADC1, ENABLE);
  ADC_Cmd(ADC2, ENABLE);

  NVIC_EnableIRQ(ADC_IRQn);                // Enable IRQ for ADC in NVIC                       // 35
  ADC_ITConfig(ADC1, ADC_IT_EOC, ENABLE);  // Enable IRQ generation in ADC                     // 36
}
```

The differences start in line 22. We need to use a hardware signal to trigger the start of conversion, and this is configured with initializing the member '.ADC_ExternalTrigConvEdge' to accept the rising edge of an external signal. This line basically enables the output of a multiplexor below the ADC, Fig. 8.1 to start the conversion with its rising edge. There are three inputs to the multiplexor related with timer TIM5, all of them are outputs from Capture and Compare blocks within timer TIM5. They are called CH1, CH1 and CH3, see Fig. 9.2 for reference. Whichever we use we have to configure timer TIM5 to implement capture and compare function, and here we opt to use channel CH1 as the source of start conversion pulses by using the keyword 'ADC_ExternalTrigConv_T5_CC1' to initialize the member '.ADC_ExternalTrigConv'.

The second difference is the addition of statements to allow the end of conversion signal to be used as an interrupt request. Two steps are necessary.

- The first step is used to configure the controller NVIC. A call to function "NVIC_Enable()" with an argument 'ADC_IRQn' allows the controller NVIC to react on an interrupt request signal 'ADC_IRQ' from the ADC, line 35.
- The second step in line 36 configures the ADC to use its end of conversion signal EOC to issue an interrupt request to controller NVIC. There are other signals within the ADC that can be used to issue the interrupt request, a list of them is given in the header file "stm32f4xx_adc.h", lines 478 to 481, and can be seen in the reference manual RM0090, pg.387, where a detailed explanation is provided.

The new configuration function for the timer is dealt next, its listing is given below.

```
void TIM5init_TimeBase_CC1 (int interval)  {
TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStructure;
TIM_OCInitTypeDef       TIM_OCInitStructure;                                                   // 3

  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM5,  ENABLE);

  TIM_TimeBaseInitStructure.TIM_Prescaler = 0;
  TIM_TimeBaseInitStructure.TIM_CounterMode = TIM_CounterMode_Up;
  TIM_TimeBaseInitStructure.TIM_Period = interval;
  TIM_TimeBaseInitStructure.TIM_ClockDivision = TIM_CKD_DIV1;
  TIM_TimeBaseInitStructure.TIM_RepetitionCounter = 0;
  TIM_TimeBaseInit(TIM5, &TIM_TimeBaseInitStructure);

  TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;                               // 14
  TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;                   // 15
  TIM_OCInitStructure.TIM_OutputNState = TIM_OutputState_Disable;                 // 16
  TIM_OCInitStructure.TIM_Pulse = 1;                                              // 17
  TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;                       // 18
  TIM_OCInitStructure.TIM_OCNPolarity = TIM_OCNPolarity_Low;                      // 19
  TIM_OCInitStructure.TIM_OCIdleState = TIM_OCIdleState_Reset;                    // 20
  TIM_OCInitStructure.TIM_OCNIdleState =  TIM_OCIdleState_Set;                    // 21
  TIM_OC1Init(TIM5, &TIM_OCInitStructure);                                        // 22

  TIM_Cmd(TIM5, ENABLE);                                                          // 24
}
```

The configuration is performed in two steps. The first step deals with counting, and is the same as given in chapter 11. The second step deals with capture and compare function needed to generate the SC pulses, and is new. The capture and compare function is described in detail in reference manual RM0090, chapter 18. A CMSIS function “TIM_OC1Init()” is used to ease the configuration of the output ‘OC1’ (equivalent functions are available for configuration of other outputs OC2 to OC4, and a brief description is given in the source file for timer), and this function requires the use of data structure “TIM_OCInitStructure”, which is declared in line 3 of the function. The members of the data structure are initialized in lines 14 to 21, and the function is called in line 22.

- The first member ‘.TIM_OCMode’ is initialized to a pulse width modulation (PWM) mode. Other options are listed in the header file “stm32f4xx_tim.h”, lines 240 to 245, and described in the reference manual RM0090.
- The second member ‘.TIM_OutputState’ is enabled to allow the generation of a pulse at the output from the timer OC1 to start the conversion.
- Next member ‘.TIM_OutputNState’; the negated version of the above signal can be generated as well, but is not needed and therefore disabled.
- Next member ‘.TIM_Pulse’ is initialized to 1 causing the signal OC1 to change its state when the content of the counter CNT equals 1.
- The fifth member ‘.TIM_OCPolarity’ is initialized to ‘TIM_OCPolarity_High’, causing the signal OC1 to change to high when the condition specified by the previous member is fulfilled.
- The sixth member ’.TIM_OCNPolarity’ is not needed here, but still initialized.
- The seventh member ‘.TIM_OCIdleState’ specifies the initial state of the OC1 after the reload event of the counter, and is configured to low (Reset) state.
- The last member ‘.TIM_OCNIdleState’ is not needed here, but still initialized.

The function call to enable timer interrupt requests, as given in chapter 11, is not needed here and has been removed. The configuration function terminates by enabling the counting in line 24.

The interrupt function deals with reading of the result of conversion from the ADC and writing of the same to DAC. Two lines suffice; they are given in the listing bellow.

```
void ADC_IRQHandler(void)      {
  DAC->DHR12R1 = ADC1->DR;
  DAC->DHR12R2 = ADC2->DR;
}
```

Please note the changed name of the interrupt function, since it is called due to a different interrupt request. Note also, that the flag to memorize the interrupt request from the ADC is stored in ADC block itself, and is automatically cleared by reading a result from the ADC, see the reference manual RM0090, section on ADC, for details. There is no need to clear this flag manually. Also, since conversions are now initiated by the timer, there is no need for software command to start the conversion.

The program to implement the timer triggered conversions is given in the listing below. It should be complemented with the inclusion of all functions described above.

```
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_adc.c"
#include "stm32f4xx_dac.c"
#include "stm32f4xx_gpio.c"
#include "stm32f4xx_tim.c"
#include "dd.h"

int main ()  {
  ADCinit_T5_CC1_IRQ();                                                                    // 9
  DACinit();                                                                               // 10
  TIM5init_TimeBase_CC1(840);           // 840 == 10us                                     // 11

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

The main part of the program calls the configuration functions and then enters an empty infinite loop, where it waits for interrupts from ADC. The timer TIM5 triggers the conversion at the ADC, and once the conversion is finished the end of conversion EOC signal from the ADC triggers the interrupt request to the NVIC and the processor. The processor breaks the execution of the empty loop and jumps to execute the interrupt function, where it reads the result of conversion (and simultaneously clears the flag memorizing the interrupt request) and transfers it to the DAC. After this the execution returns to the endless empty loop until the next interrupt.

The remark on periodic sampling and the urge to have equal time intervals between consecutive samples applies to periodic generating by the DAC as well. In order to reduce the harmonic distortion the DAC should move the data from the data holding register DHR to data output register DOR on signal derived from the same timer TIM5. This could be done by utilizing the multiplexor ‘Trigger selectorx’, see Fig. 7.1 for reference, and a change of the configuration of the DAC. However, this detail is omitted here.