30. DMA and ADC

The use of ADC and the transfer of results from the microcontroller to the PC was demonstrated in experiment 14. The microcontroller was connected to a PC; the PC sent a command to start the acquisition, then the microcontroller took 32768 samples at two ADC channels simultaneously; the acquisition was controlled by timer, and ADC interrupts were used to move results from ADCs into the memory of the microcontroller. The time interval between consecutive samples was 100 us. The next command issued from the PC initiated the transfer of the acquired results from the microcontroller to the PC, again using serial connection and USART_TX interrupts.
Using interrupts as in experiment 14 is too slow if one wants to sample analog signals at full speed available at STM32F407 ADCs (about 2 MHz); the interrupt routine to move acquisition results from ADCs to memory is the bottleneck. In this experiment we are going to use DMA block to move the acquisition results from ADC into memory, and (since we already know how to do it from experiment 29) later to move them from memory to serial communication block. Basically, the interrupt activity will be replaced by DMA activity, the sampling can be significantly faster but still simultaneous for two ADCs.

30.1 The theory

The same theory as described in experiment 29 about the DMA transfers applies for this experiment as well. The only difference is that appropriate DMA unit, stream and channel should be selected for the transfer of acquisition results from both ADCs to the memory.

30.2 The outline of the program

Consider the following scenario for this experiment (the microcontroller is configured once in advance):

- A PC issues a command ‘s’ to start the acquisition, the command is sent using a serial communication to the microcontroller, USART3.
- The serial communication block USART3 is configured to issue an interrupt on receipt of a character (RXIE, RX interrupt enable), therefore any character received invokes an interrupt routine at the microcontroller (as described in experiment 14).
- Within the interrupt routine the microcontroller checks the character received; if it is an ‘s’, it initiates a set of ADC acquisitions by enabling DMA block for ADCs. The DMA block is now ready to move data from both ADCs to the memory of the microcontroller immediately after ADCs end a conversion. DMA block is pre-configured to move 4096 results, and then issue a DMA interrupt.
- Still within the same interrupt routine a timer is started; this timer is used to trigger the start of conversion at the ADC on overflow, as it was configured in experiment 14. Then the program exits the interrupt routine for USART3 TX.

- Every overflow of the timer now sends a StartOfConversion signal to ADCs. Upon the EndOfConversion signal from ADCs results from both ADCs are moved by the DMA block from ADCs to memory.
- The above action repeats 4096 times, then the DMA block issues an interrupt signaling “acquisition finished”.
- The DMA interrupt routine gets executed. Here the timer is stopped, and the DMA for ADCs is disabled. Then, still within the same interrupt routine, another DMA block for serial transfer is configured to transfer 2 x 2 x 4096 (2 ADCs, 2 bytes per result, 4096 results) bytes from the memory of the microcontroller to USART3, TX register. This DMA block gets enabled at the end of the interrupt routine, then the microcontroller leaves the interrupt routine.
- The complete set of measurement results gets transferred from the microcontroller to PC using serial channel.

Note that the complete transfer of results from ADCs to memory and from memory to USART gets handled by the DMA, not by interrupt routines. The interrupt routine is used once only to stop the acquisition after sufficient number of samples are gathered and to commence the transfer to the PC; the microcontroller CPU is free to perform other tasks during the acquisition and transfers, and the acquisition is fully hardware supported, therefore the time intervals between consecutive samples are exactly equal.

30.3 The software

The program commences with configuration routines for LED outputs, serial block USART3, DMA blocks DMA1 and DMA2, and the timer. Next the program prepares the initial part of the string that is to be sent from the microcontroller to a PC, and enters an endless empty loop.

a) Main

The listing of the “main” part of the program is given below. Using a #define statement the name ‘MAXRAW’ is initialized; this represents the number of samples to acquire and transfer, 4096 for this experiment. Next (line 02) an array for the samples acquired is declared. The array consists words (16 bits, “unsigned short”, the same as registers for results in ADCs), and has three sections:

- First four words of the array will be reserved for four constants; it is a good idea to mark the beginning of the string with some special characters so that the PC (actually the software that shall interpret bytes within the string) can recognize the commencement of the string. Since our ADCs cannot produce results like 0xffff (we use 12-bit ADCs, therefore the highest result can be 0x0fff) a pair of 0xffff words is adequate. Additionally, one might want to pass some additional measurement parameters from the microcontroller to the PC, so additional two words are reserved for this purpose; altogether four words.
- Next ‘MAXRAW’ words are reserved for results from ADC1, and
- last ‘MAXRAW’ words are reserved for results from ADC2.

The “main” routine commences with calls to configuration routines for peripherals used, lines 03 to 09. The names of routines clarify the peripherals in question. Note that the last configuration routine for timer uses a parameter 83, which is the number of clock periods (actually 84 - 1) that cause the overflow of this counter; this number defines the time interval between two overflows as 1 us.

The configuration part is finalized with the initialization of the first four words of the array, then the program enters the infinite empty “while” loop. All action from this moment on is hidden mainly within the hardware and partially within interrupt routines.

```
#define MAXRAW 4096                                                                      // 01
short unsigned volatile ADCraw[4 + MAXRAW + MAXRAW];                                     // 02
```

```
void main(void) {
  GPIOD_LEDS();                                                                     // 03
  USART3_init(921600);                                                              // 04
  DMA1forUSART3_init();                                                             // 05
  ADCinit_T3_TRGO_DMA();                                                            // 06
  DMA2forADC1_init();                                                               // 07
  DMA2forADC2_init();                                                               // 08
  TIM3init_ADCtrigger(83);                     // 1us intervals!                    // 09

  ADCraw[0] = 0xffff;  ADCraw[1] = 0xffff;                                          // 10
  ADCraw[2] = 0x0000;  ADCraw[3] = 0x0100;                                          // 11

  while (1) {  };                                                                   // 12
}
```

b) Configuration of LED outputs

Output pins for LEDs are configured first using a standard configuration procedure, the listing is given below.

```
// initialize port LEDS: pins 12, 13, 14, 15, port D as outputs
void GPIOD_LEDS (void)   {
GPIO_InitTypeDef  GPIO_InitStructure;
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);
  GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_12 | GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15;
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_OUT;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
  GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_NOPULL;
  GPIO_Init(GPIOD, &GPIO_InitStructure);
}
```

c) Configuration of USART block

The configuration of the USART block is almost the same as for experiment 29. The only difference is that this time the USART block should be able to call an interrupt routine for every character received (and not transmitted), so the last three lines are added to enable interrupts in the listing below. Such configuration was used in experiments on serial communication, chapter 14.

```
//USART3 initialization, GPIOD 8:9, TX&RX, IRQ on RX
void USART3_init(int BaudRate)   {
USART_InitTypeDef       USART_InitStructure;
GPIO_InitTypeDef        GPIO_InitStructure;
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);
  GPIO_InitStructure.GPIO_Pin   = GPIO_Pin_8 | GPIO_Pin_9;
  GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_AF;
  GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_NOPULL;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
  GPIO_Init(GPIOD, &GPIO_InitStructure);
  GPIO_PinAFConfig(GPIOD, GPIO_PinSource8, GPIO_AF_USART3);
  GPIO_PinAFConfig(GPIOD, GPIO_PinSource9, GPIO_AF_USART3);

  RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART3, ENABLE);
  USART_DeInit(USART3);
  USART_InitStructure.USART_BaudRate            = BaudRate;
  USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
  USART_InitStructure.USART_Mode                = USART_Mode_Rx | USART_Mode_Tx;
  USART_InitStructure.USART_Parity              = USART_Parity_No;
  USART_InitStructure.USART_StopBits            = USART_StopBits_1;
  USART_InitStructure.USART_WordLength          = USART_WordLength_8b;
  USART_Init(USART3, &USART_InitStructure);

  USART_ITConfig(USART3, USART_IT_RXNE, ENABLE); // enable IRQ on RX, disable on TX
  USART_Cmd(USART3, ENABLE);
  NVIC_EnableIRQ(USART3_IRQn);   // Enable IRQ for USART3 in NVIC
}
```

d) Configuration of DMA block for USART3

The configuration of the DMA block for USART3, transmit action, is the same as it was described in experiment 29, the listing is given below. The complete array will be transferred byte by byte from the memory of the microcontroller to the USART block. One might note that the conversion results within the memory are stored as words, while the same are passed to USART block as bytes; this poses no problem, only an appropriate number of transfers for the DMA block and correct addresses will have to be stated initially. The correct pointer to the start of the array is given at line 01, and correct sizes are provided at lines 02 and 03.

```
//DMA1 initialization, TX requests next character to send
void DMA1forUSART3_init(void) {
DMA_InitTypeDef         DMA_InitStructure;
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA1, ENABLE);
  DMA_DeInit(DMA1_Stream3);             /* Reset DMA Stream registers (for debug purpose) */
  /* Configure DMA Stream */
  DMA_InitStructure.DMA_Channel                 = DMA_Channel_4;
  DMA_InitStructure.DMA_PeripheralBaseAddr      = (uint32_t)(&USART3->DR);
  DMA_InitStructure.DMA_Memory0BaseAddr         = (uint32_t)(&ADCraw[0]);                      // 01
  DMA_InitStructure.DMA_DIR                     = DMA_DIR_MemoryToPeripheral;
  DMA_InitStructure.DMA_BufferSize              = (uint32_t)8;
  DMA_InitStructure.DMA_PeripheralInc           = DMA_PeripheralInc_Disable;
  DMA_InitStructure.DMA_MemoryInc               = DMA_MemoryInc_Enable;
  DMA_InitStructure.DMA_PeripheralDataSize      = DMA_PeripheralDataSize_Byte;                 // 02
  DMA_InitStructure.DMA_MemoryDataSize          = DMA_MemoryDataSize_Byte;                     // 03
  DMA_InitStructure.DMA_Mode                    = DMA_Mode_Normal;
  DMA_InitStructure.DMA_Priority                = DMA_Priority_High;
  DMA_InitStructure.DMA_FIFOMode                = DMA_FIFOMode_Disable;
  DMA_InitStructure.DMA_FIFOThreshold           = DMA_FIFOThreshold_Full;
  DMA_InitStructure.DMA_MemoryBurst             = DMA_MemoryBurst_Single;
  DMA_InitStructure.DMA_PeripheralBurst         = DMA_PeripheralBurst_Single;
  DMA_Init(DMA1_Stream3, &DMA_InitStructure);
}
```

e) Configuration of the ADC

The configuration of the ADC follows the standard established procedure. First both ADC1 and ADC2 are turned on by enabling their respective clocks, then pins for inputting analog signals to these ADCs are configured. Next, the common section for both ADCs is configured at lines 01 to 04. The un-familiar configuration is given at line 01, where the DMA access mode is selected; the DMA block shall be used to transfer conversion results from ADCs to a pre-declared array. There are various DMA modes available, and they serve different number of ADCs; for this example “ADC_DMAAccessMode_2” is appropriate. The details on the mode used are given in RM0009, section 13.9.

Line 05 selects timer TIM3, trigger output, as the source of the StartConversion signal. The rest of this routine is the same as in previous examples on the use of ADCs.

```
// ADC init function
void ADCinit_T3_TRGO_DMA (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_DMAAccessMode    = ADC_DMAAccessMode_2;                          // 01
  ADC_CommonInitStructure.ADC_Mode             = ADC_DualMode_RegSimult;                       // 02
  ADC_CommonInitStructure.ADC_Prescaler        = ADC_Prescaler_Div2;                           // 03
  ADC_CommonInit(&ADC_CommonInitStructure);                                                    // 04

  ADC_InitStructure.ADC_Resolution             = ADC_Resolution_12b;
  ADC_InitStructure.ADC_ScanConvMode           = DISABLE;
  ADC_InitStructure.ADC_ContinuousConvMode     = DISABLE;
  ADC_InitStructure.ADC_ExternalTrigConvEdge   = ADC_ExternalTrigConvEdge_Rising;
  ADC_InitStructure.ADC_ExternalTrigConv       = ADC_ExternalTrigConv_T3_TRGO;                 // 05
  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);
}
```

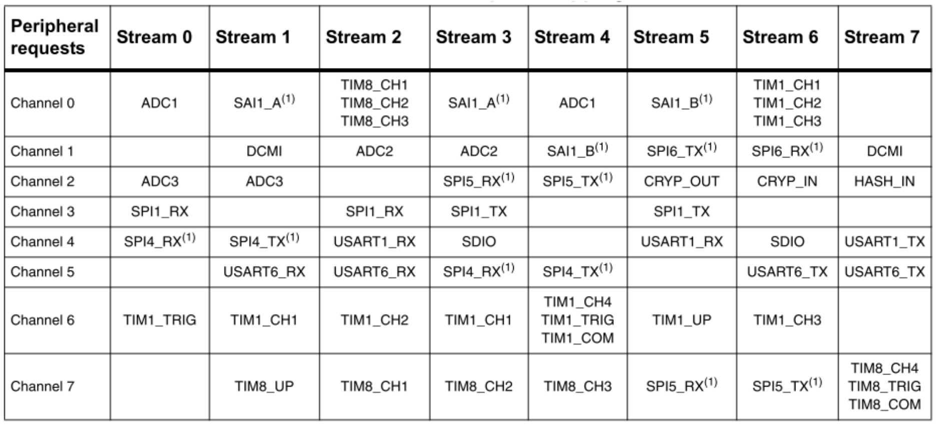
f) Configuration of the DMA for ADC

Conversion results shall be transferred from both ADCs to a predefined space in memory using the DMA block. The EndOfConversion for each of the two ADCs should therefore raise a DMA request, and two DMA channels need to be configured to accomplish both transfers. Let us start with DMA configuration for ADC1.

First the correct stream and channel for ADC1 must be identified from the table at figure 30.1. We might select Stream0/Channel0 or Stream4/Channel0, and will use the first option for no particular reason. The complete configuration routine is given below.

```
//DMA2 configuration, ADC1
void DMA2forADC1_init(void) {
DMA_InitTypeDef         DMA_InitStructure;                                                     // 00
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA2, ENABLE);
  DMA_DeInit(DMA2_Stream3);               /* Reset DMA Stream registers (for debug purpose) */
  /* Configure DMA Stream */
  DMA_InitStructure.DMA_Channel                 = DMA_Channel_0;                          // 01
  DMA_InitStructure.DMA_PeripheralBaseAddr      = (uint32_t)(&ADC1->DR);                  // 02
  DMA_InitStructure.DMA_Memory0BaseAddr         = (uint32_t)(&ADCraw[4]);                 // 03
  DMA_InitStructure.DMA_DIR                     = DMA_DIR_PeripheralToMemory;             // 04
  DMA_InitStructure.DMA_BufferSize              = (uint32_t)MAXRAW;                       // 05
  DMA_InitStructure.DMA_PeripheralInc           = DMA_PeripheralInc_Disable;              // 06
  DMA_InitStructure.DMA_MemoryInc               = DMA_MemoryInc_Enable;                   // 07
  DMA_InitStructure.DMA_PeripheralDataSize      = DMA_PeripheralDataSize_HalfWord;        // 08
  DMA_InitStructure.DMA_MemoryDataSize          = DMA_MemoryDataSize_HalfWord;            // 09
  DMA_InitStructure.DMA_Mode                    = DMA_Mode_Normal;                        // 10
  DMA_InitStructure.DMA_Priority                = DMA_Priority_High;                      // 11
  DMA_InitStructure.DMA_FIFOMode                = DMA_FIFOMode_Disable;                   // 12
  DMA_InitStructure.DMA_FIFOThreshold           = DMA_FIFOThreshold_Full;                 // 13
  DMA_InitStructure.DMA_MemoryBurst             = DMA_MemoryBurst_Single;                 // 14
  DMA_InitStructure.DMA_PeripheralBurst         = DMA_PeripheralBurst_Single;             // 15
  DMA_Init(DMA2_Stream0, &DMA_InitStructure);                                             // 16

  DMA_ITConfig(DMA2_Stream0, DMA_IT_TC, ENABLE);    // interrupt at the end of DMA transfer    // 17
  NVIC_EnableIRQ(DMA2_Stream0_IRQn);   // Enable IRQ for USART3 in NVIC                   // 18
}
```



Peripheral requests	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	ADC1	SAI1_A(1)	TIM8_CH1 TIM8_CH2 TIM8_CH3	SAI1_A(1)	ADC1	SAI1_B(1)	TIM1_CH1 TIM1_CH2 TIM1_CH3	
Channel 1		DCMI	ADC2	ADC2	SAI1_B(1)	SPI6_TX(1)	SPI6_RX(1)	DCMI
Channel 2	ADC3	ADC3		SPI5_RX(1)	SPI5_TX(1)	CRYP_OUT	CRYP_IN	HASH_IN
Channel 3	SPI1_RX		SPI1_RX	SPI1_TX		SPI1_TX		
Channel 4	SPI4_RX(1)	SPI4_TX(1)	USART1_RX	SDIO		USART1_RX	SDIO	USART1_TX
Channel 5		USART6_RX	USART6_RX	SPI4_RX(1)	SPI4_TX(1)		USART6_TX	USART6_TX
Channel 6	TIM1_TRIG	TIM1_CH1	TIM1_CH2	TIM1_CH1	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	
Channel 7		TIM8_UP	TIM8_CH1	TIM8_CH2	TIM8_CH3	SPI5_RX(1)	SPI5_TX(1)	TIM8_CH4 TIM8_TRIG TIM8_COM

Figure 30.1: ADC1 and ADC2 can be served by DMA2, stream0/channel0 and stream2/channel1 for instance

The details regarding this DMA transfer will be considered next.

The configuration of the DMA follows the standard procedure. First a standard data structure is declared (line 00), then this structure is stuffed with parameters (lines 01 to 15), and finally a routine is called to distribute these parameters into corresponding registers (line 16). The parameters selected deserve some explanation:

- We are dealing with Channel 0 (line 01), and are configuring DMA2, Stream 0 (line 16)
- The pointer to ADC data register as the source of word to be transferred is given at line 02
- The pointer to the destination, i.e. the array with conversion results, is given at line 03; note that the first four words of this array are reserved, and that conversion results should be stored at locations from [4] on.
- The number of consecutive transfers is given at line 05. The last transfer (MAXRAW) should trigger an interrupt to stop the acquisition.
- The address to the source (the DAC data register) should not be incremented, while the address of the destination should be, as given at lines 06 and 07.
- The size of the data to be transferred is provided at lines 08 and 09, 'halfword' equals 16 bits.
- We are not changing the default priorities (lines 10 and 11), we are not using DMA buffer (lines 12 and 13).

- The DMA should transfer one word at a time, this means one word for one DMA request. This is stated at lines 14 and 15.

The configuration concludes with enabling the interrupt at the end of DMA transfer, lines 17 and 18. The interrupt must be enabled in two consecutive commands. First the cause of the interrupt is selected (the source is DMA2 block, Stream 0, and the interrupt should occur due to the TransferComplete, TC flag), then the interrupt should be enabled at the NVIC level.

Almost the same configuration can be used for ADC2 as well. The listing is provided bellow. Different Channel/Stream is used here, and the pointer to the destination address is given as element MAXRAW+4 within the array for conversion results.

```
//DMA2 configuration, ADC2
void DMA2forADC2_init(void) {
DMA_InitTypeDef         DMA_InitStructure;
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA2, ENABLE);
  DMA_DeInit(DMA2_Stream3);                                       /* Reset DMA Stream registers (for
debug purpose) */
  /* Configure DMA Stream */
  DMA_InitStructure.DMA_Channel                 = DMA_Channel_1;
  DMA_InitStructure.DMA_PeripheralBaseAddr      = (uint32_t)(&ADC2->DR);
  DMA_InitStructure.DMA_Memory0BaseAddr         = (uint32_t)(&ADCraw[MAXRAW+4]);
  DMA_InitStructure.DMA_DIR                     = DMA_DIR_PeripheralToMemory;
  DMA_InitStructure.DMA_BufferSize              = (uint32_t)MAXRAW;
  DMA_InitStructure.DMA_PeripheralInc           = DMA_PeripheralInc_Disable;
  DMA_InitStructure.DMA_MemoryInc               = DMA_MemoryInc_Enable;
  DMA_InitStructure.DMA_PeripheralDataSize      = DMA_PeripheralDataSize_HalfWord;
  DMA_InitStructure.DMA_MemoryDataSize          = DMA_MemoryDataSize_HalfWord;
  DMA_InitStructure.DMA_Mode                    = DMA_Mode_Normal;
  DMA_InitStructure.DMA_Priority                = DMA_Priority_High;
  DMA_InitStructure.DMA_FIFOMode                = DMA_FIFOMode_Disable;
  DMA_InitStructure.DMA_FIFOThreshold           = DMA_FIFOThreshold_Full;
  DMA_InitStructure.DMA_MemoryBurst             = DMA_MemoryBurst_Single;
  DMA_InitStructure.DMA_PeripheralBurst         = DMA_PeripheralBurst_Single;
  DMA_Init(DMA2_Stream2, &DMA_InitStructure);

  DMA_ITConfig(DMA2_Stream2, DMA_IT_TC, ENABLE);        // interrupt at the end of DMA transfer
  NVIC_EnableIRQ(DMA2_Stream2_IRQn);   // Enable IRQ for USART3 in NVIC
}
```

g) Configuration for timer

Timer TIM3, trigger output TRGO, is selected to start conversion in this experiment. This option can be selected within the ADC common structure using the lower right multiplexor from figure 44, chapter 13.3, RM0090. The complete configuration routine is given below.

```
// Timer 3 init function - time base
void TIM3init_ADCtrigger (int interval)  {
TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStructure;                                     // 00
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3,  ENABLE);                                // 01
  TIM_TimeBaseInitStructure.TIM_ClockDivision = TIM_CKD_DIV1;                          // 02
  TIM_TimeBaseInitStructure.TIM_CounterMode = TIM_CounterMode_Up;                      // 03
  TIM_TimeBaseInitStructure.TIM_Period = interval;                                     // 04
  TIM_TimeBaseInitStructure.TIM_Prescaler = 0;                                         // 05
  TIM_TimeBaseInitStructure.TIM_RepetitionCounter = 0;                                 // 06
  TIM_TimeBaseInit(TIM3, &TIM_TimeBaseInitStructure);                                  // 07
  TIM_SelectOutputTrigger(TIM3, TIM_TRGOSource_Update); // SC (TRGO) on TIM3 update            // 08
}
```

The data structure for the configuration os first declared at line 00, then the timer TIM3 is turned on by enabling its clock at line 01. Next the data structure is stuffed with configuration parameters and the routine to distribute these parameters throughout the timer TIM3 registers is called at line 07.

Finally, trigger output TRGO is configured to issue a pulse on timer overflow (“TIM_TRGOSource_Update”).

This concludes the configuration part of the program.

h) Sequence of events: interrupt routine for USART3

The listing of the USART3 interrupt routine is given below. Only the RX interrupts are served, since only these were enabled within USART3 configuration routine. The status register of USART3 is therefore checked if the receive flag caused the entrance to the interrupt routine at line 00; if this is the case then the rest of the routine gets executed. The character received is fetched from the USART3 DataRegister into a character variable RXch, then this variable gets checked against predefined constants. When ‘a’ the interrupt routine turns on the LED, and when ‘b’ it turns the same LED off. This two commands are added to ease the verification of the serial communication between a PC and the microcontroller. However, if the character received equals ‘s’ (line 01) the control registers CR2 for ADC1 and ADC2 are modified (lines 02 and 03); DMA is first disabled, and enabled immediately afterword. This operation is required and described in RM0090, chapter 13.8.1. The operation seems un-necessary, but it clears some flags that cannot be cleared otherwise. Following this the DMA streams for both ADCs are enabled at lines 04 and 05, and then the timer TIM3 is enabled at line 06. The last thing is the LED that gets turned on at line 07 to signal the start of the acquisition. With this the interrupt routine finishes and the execution returns to the empty “while” loop.

The timer TIM3 is now running and periodically starts a conversion at both ADCs. Once the conversions are finished both results are transferred using two DMA streams from ADC data registers to the memory of the microcontroller at appropriate addresses; the destination addresses get incremented after every transfer. There are ‘MAXRAW’ transfers, then both DMA streams within DMA2 block issue corresponding interrupts, and two interrupt routines get executed. This is described in the following section.

```
// IRQ function for USART3
void USART3_IRQHandler(void)     {
  // RX IRQ part
  if (USART3->SR & USART_SR_RXNE) {   // if RXNE flag in SR is on then                         // 00
    int RXch = USART3->DR;            // save received character & clear flag
    if (RXch == 'a') LED_BL_ON;       // to show we are alive
    if (RXch == 'b') LED_BL_OFF;      // to show we are alive
    if (RXch == 's') {                // actual start of DMA supported sampling                // 01
       ADC1->CR2 &= ~ADC_CR2_DMA;        ADC1->CR2 |=  ADC_CR2_DMA;                            // 02
       ADC2->CR2 &= ~ADC_CR2_DMA;        ADC2->CR2 |=  ADC_CR2_DMA;                            // 03
       DMA_Cmd(DMA2_Stream0, ENABLE);                                                          // 04
       DMA_Cmd(DMA2_Stream2, ENABLE);                                                          // 05
       TIM_Cmd(TIM3, ENABLE);                                                                  // 06
       GPIOD->BSRRL = 0x8000;                                                                  // 07
    };
  };
}
```

i) Sequence of events: interrupt routine for DMA/DAC1 (Stream 0)

The listing of the interrupt routine is given below. First the timer TIM3 is stopped to prevent any further Start Conversion pulses to the ADC block. Next, the DMA stream for ADC1 gets disabled, since this was the last transfer within the sequence of MAXRAW transfers. Finally, some flags within the DMA2, Stream 0 get cleared. These are flags that signal Transfer Complete status and Transfer Error status; we will ignore these messages for this demonstration.

```
void DMA2_Stream0_IRQHandler(void)   {
  TIM_Cmd(TIM3, DISABLE);
  DMA_Cmd(DMA2_Stream0, DISABLE);
  DMA_ClearFlag(DMA2_Stream0, DMA_FLAG_TCIF0 | DMA_FLAG_TEIF0);
}
```

j) Sequence of events: interrupt routine for DMA/DAC2 (Stream 2)

The listing of this interrupt routine is given bellow. It is expected that this routine executed after the above one since Stream 0 has higher priority than Stream 2.

```
void DMA2_Stream2_IRQHandler(void)   {
  TIM_Cmd(TIM3, DISABLE);
  DMA_Cmd(DMA2_Stream2, DISABLE);
  DMA_ClearFlag(DMA2_Stream2, DMA_FLAG_TCIF2 | DMA_FLAG_TEIF2);

  GPIOD->BSRRH = 0x8000;
  TIM3->CNT = 0;

  DMA_SetCurrDataCounter(DMA1_Stream3, MAXRAW * 4 + 8);                                       // 01
  USART_DMACmd(USART3, USART_DMAReq_Tx, ENABLE);                                              // 02
  DMA_ClearFlag(DMA1_Stream3, DMA_FLAG_TCIF3 | DMA_FLAG_HTIF3 | DMA_FLAG_FEIF3);              // 03
  DMA_Cmd(DMA1_Stream3, ENABLE);            // note: USART3->SR.TXE requests DMA              // 04
}
```

The routine commences with stopping the timer TIM3 (this might not be needed here since the same timer was already disabled within the “DMA2_Stream0_IRQHandler”). Next, the DMA stream for ADC2 gets disabled, and some flags for this stream get cleared. Since the acquisition is now done the LED, signaling the acquisition, gets turned off, and the content of the timer TIM3 gets cleared and prepared for next time.

It is now the time to transfer the acquired results from the memory of the microcontroller to PC using the USART3, in particular the DMA 1, Stream 3. The DMA block was already configured for the transfer, all it takes now is to refresh the required number of bytes to be sent in line 01 (there are MAXRAW * 2 bytes * 2 channels + 8 bytes to be sent), then the USART3 DMA request should be enabled. Some flags should again be cleared within the DMA1 block for this stream at line 03, and finally this stream should be enabled. With this the transfer of bytes from the memory to USART3 data register commences, and the next byte will be transferred without the intervention of the processor under the control of the DMA1 block until all bytes are sent.

One might want to use another interrupt at the end of the transfer to signal the end of operation, but this is not implemented in this example.