29. DMA & Serial communication

A serial communication was already demonstrated in experiment 14, where interrupt technique was implemented to handle the transfer of a substantial number of bytes from microcontroller to aPC. However, there are times when a large amount of data needs to be transferred, and the use of interrupts might pose too much of a burden to the CPU. Namely, the operation of the CPU gets interrupted for every byte passed from memory of the microcontroller to the UART block, and this might spoil the CPU timing. To avoid tight situations a special hardware block to handle the transfer is introduced. Such block is called Direct Memory Access block (DMA), and can transfer data from one location within the microprocessor to another one without the intervention of the CPU, which is then free to do other tasks.

29.1 The theory

The theory behind serial communication was already explained in chapter 14. Bytes get transferred from the microcontroller to, for instance, PC byte by byte, and each byte takes some time to transfer. The timing depends on the Baud rate setting, and the CPU should pass the complete string of bytes for transmission one by one at appropriate times; the CPU send a byte, and once it gets absorbed by the UART block, the block signals this by setting a flag TXE (transmit buffer empty); then the CPU send next byte. Two options are available based on the knowledge so far:

a) The CPU can execute a loop and check the flag TXE periodically. Once this flag gets set the CPU passes next byte to the UART block, and this loop continues until all bytes are passed to the UART (and hopefully out of the microcontroller to the PC). This option requires constant CPU attention, and therefore blocks all other CPU activities, rendering a real-time processing impossible. This option has only limited use.

b) The UART unit can be configured to issue an interrupt once the transmit buffer of the UART is free to absorb next byte, and the CPU can be configured to respond to these interrupts, as it was shown in chapter 14. The software needs to send the first byte in a string of bytes, and once the UART block is ready to absorb next byte it issues an interrupt that starts the execution of the interrupt routine within the CPU. The main part of this routine passes next byte to the UART TX buffer; it also checks weather all bytes were sent and breaks the repeating action once all bytes from the string are passed. This option requires periodic intervention of the CPU; the interventions are short since the interrupt routine consists of few lines of code, therefore not much time gets wasted. However, the CPU must execute this short interrupt routine periodically, and this might still interfere with the exact timing needed for real-time data processing (yes, setting interrupt priorities might solve problems here…). This option has wide use, but it might not be adequate in some cases.

In order to relieve the CPU from passing byte-by-byte from the memory of the microcontroller to the UART, a special hardware might be introduced. The busses leading data between memory and the

CPU and other peripherals are not busy all the time; there are times when the CPU has enough work to do and does not need data and address busses. At such times the CPU releases the data and address buses, and then these two buses can be used by other devices to pass data. Such device that can take over the control over the address and data bus is the Direct Memory Access block (DMA). Actually, the STM32F407 houses two DMA blocks that serve different peripherals and pass data in different directions within the microcontroller. Both DMA blocks utilize short but frequent time intervals when both address and data buses are not engaged by signals from the CPU, and the operation of the DMA blocks is completely transparent and does not interfere with the operation of the CPU.

The use of the DMA block opens the third option for feeding the UART unit:

c) The DMA can be configured to take over the transfer of bytes: the DMA must know two addresses, that is the address of the source (the location of the bytes to be transferred within the memory) and the location of the destination (the address of the UART TX register). Additionally, the DMA must know the number of bytes to transfer and the size of each transfer, byte in this case (8 bits per transfer here, but might be 16 of 32 bits in some cases). Next, the DMA must know how to handle both source and destination address and weather to increment them after each transfer. Finally, the DMA must know what triggers the transfer. Once these options are configured the DMA block can be enabled by the software, and the transfer commences. The first transfer of the initial byte is executed by the DMA immediately, and each next transfer gets initiated by the TXE flag from the UART block; this is how we shall configure the DMA block. Once the predefined number of bytes (the complete string in our case) get transferred, the DMA action ends. To emphasize: the transfer is independent of the CPU activity after it commences, and the CPU does not get any interruptions due to transfers (unless we configure the DMA unit for some interrupts).

In general, the DMA block can move large amount of data in three ways from the source to the destination: from one place in memory to another place, from memory to peripheral or from peripheral to memory. When the memory is involved in DMA transfer we expect that the address pointer to the memory, being either the source or the destination, should increment after each transfer; contrary to the above the address pointer to the peripheral involved should stay the same throughout the DMA operation. There is a setting to the DMA specifying whether the corresponding address pointers to the source and/or destination should be incremented after each transfer or left as they are.

A transfer can be initiated by the selected peripheral or a condition within (like the TX empty flag in former case), periodically by most of the timers, and many other sources that are listed in the reference manual RM0090, chapter 10 on DMA, table 43 and 44 (Figure 29.1).

A transfer can be single or there can be whole burst of transfers initiated by the selected event. A transfer can be 8, 16 or 32 bits wide.

There is also a First-In-First-Out (FIFO) buffer associated with the DMA; we will not use it for this experiment, and so we do not need to discuss the FIFO buffer here.

Something might go bad during the transfer. The following errors are trapped by the DMA block and can be deduced from flags and handled by the CPU. These flags can also be configured to issue an interrupt to the CPU to expedite the error handling:

- TEIF: Transfer Error Interrupt Flag, something went wrong during the transfer of the data, and
- FEIF: FIFO Error Interrupt Flag, something went wrong with the FIFO buffer.

There are two additional flags signaling the progression of the DMA transfers:

- HTIF: half of the transfers completed, and

- TCIF: all transfers completed.

Details on these flags can be found in RM0090, chapter 10.3.

Peripheral requests	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	SPI3_RX		SPI3_RX	SPI2_RX	SPI2_TX	SPI3_TX		SPI3_TX
Channel 1	I2C1_RX		TIM7_UP		TIM7_UP	I2C1_RX	I2C1_TX	I2C1_TX
Channel 2	TIM4_CH1		I2S3_EXT_RX	TIM4_CH2	I2S2_EXT_TX	I2S3_EXT_TX	TIM4_UP	TIM4_CH3
Channel 3	I2S3_EXT_RX	TIM2_UP TIM2_CH3	I2C3_RX	I2S2_EXT_RX	I2C3_TX	TIM2_CH1	TIM2_CH2 TIM2_CH4	TIM2_UP TIM2_CH4
Channel 4	UART5_RX	USART3_RX	UART4_RX	USART3_TX	UART4_TX	USART2_RX	USART2_TX	UART5_TX
Channel 5	UART8_TX(1)	UART7_TX(1)	TIM3_CH4 TIM3_UP	UART7_RX(1)	TIM3_CH1 TIM3_TRIG	TIM3_CH2	UART8_RX(1)	TIM3_CH3
Channel 6	TIM5_CH3 TIM5_UP	TIM5_CH4 TIM5_TRIG	TIM5_CH1	TIM5_CH4 TIM5_TRIG	TIM5_CH2		TIM5_UP	
Channel 7		TIM6_UP	I2C2_RX	I2C2_RX	USART3_TX	DAC1	DAC2	I2C2_TX

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 29.1: Sources that can trigger transfers for DMA1 (upper table) and DMA2 block (lower table)

There are two DMA blocks available, and internally each block is organized to serve up to eight streams (numbered from 0 to 7). Each stream can be configured to transfer data due to a trigger from one of the possible eight trigger sources named “Channels”. If, for instance, each transfer is to be triggered at the end of conversion at ADC1 then DMA2 block should be used to perform the transfers, Stream 0 in particular. For this stream Channel 0 should be selected as the triggering source, as seen from tables at Fig. 29.1. Alternatively, Stream4/Channel0 could be used. The difference between these two is the priority; in case many DMA streams for different transfers are configured, and they are triggered by many peripherals simultaneously, all of them cannot be executed at the same time. The one with higher priority and lower stream number has the advantage. The DMA request priority is configurable.

29.2 The outline of the program

Let us program a simple (and non-exact) watch that sends seconds to a PC running terminal program, for instance Realterm, utilizing the DMA. The program should start with the configuration of all peripherals involved and continue to run an infinite loop. Within the loop, the program should:

a) increment the counter of seconds,
b) if the counter of seconds overflows 59 the program should return seconds counter to zero,
c) translate the number of seconds into a string of characters,
d) initiate DMA transfer of the prepared string to the serial communication block (UART3),
e) waste almost one second, and
f) repeat the loop.

The points regarding the DMA transfer will be considered next.

We are going to use USART3 to send characters to PC. The characters will be prepared in the microcontroller memory within a character array, and sent one by one out of the microcontroller. The DMA has to pass next character from the array to the USART3/TX register as soon as possible after the USART3/TX register is free to receive next character. The trigger for DMA transfer should therefore be the USART3_TX flag, and this can be utilized through DMA1 block, Stream3, Channel4 as seen from the upper table at Fig. 29.1.

Characters are eight bits entities and are stored in memory of the microcontroller in a character array. The pointer to this array should be pre-configured into the selected DMA block and stream. Since the characters are stored in consecutive locations of this array, the DMA source address for these transfers should increment after each transfer.

The destination address should be set for this DMA/stream, and it should be the address of the USART3 TX data register; this register is eight bits wide. Since the same register will be used throughout the sequence of transfers, this address should not be incremented after each transfer.

Each transfer should deal with one byte, so these will be single transfers. Since this is a simple example with only one DMA channel used, the setting of the DMA priority is not important.

Let us decide to have four characters in the message from the microcontroller to a PC: two characters are reserved for tens and units of seconds, and two characters are reserved for “carriage return” symbol (0x0a) and “line feed” symbol (0x0d). These two symbol direct the terminal program at the PC to show each message in a new line.

There may be some flags within the DMA block affected during or after the transfer of the current message, and those flags should be dealt with within the program or cleared in time, or the next message might not come out correctly.

29.3 The Software

The program commences with configuration routines for serial block USART3, DMA block DMA1 and LCD. The latter will be used to display seconds to see the progress of the program. Next the program enters an endless loop where it prepares the string if characters to be sent and initiates the DMA transfer between the memory of the microcontroller and the USART3 block. Still within the loop, it writes the seconds counter to the LCD and waits about one second before repeating the loop.

a) Main

The listing of the “main” is given bellow. The character array is declared and initialized first, line 01. A unsigned variable for counting seconds is declared and initialized next at line 02, followed by three calls to subroutines in lines 03 to 05; these calls cause a complete configuration of USART3 with a baud rate of 921600 Bd, configuration of DMA block DMA1 to transfer characters between memory and USART3, and the configuration of the LCD screen. The main infinite loop starts at line 06.

```
char OutString[4] = {'0', '0', 0x0a, 0x0d};                                    // 01

void main(void) {
unsigned Seconds = 0;                                                          // 02

  USART3_init(921600);                                                         // 03
  DMA1fromUSART3_init();                                                       // 04
  LCD_init();                                                                  // 05

  while (1) {                                                                  // 06
      if (++Seconds > 59) Seconds = 0;                                         // 07
      unsigned x = Seconds;                                                    // 08
      OutString[0] = x / 10   + '0';    x %= 10;                               // 09
```

```
      OutString[1] = x        + '0';                                                  // 10
      OutString[2] = '\n';                                                            // 11
      OutString[3] = '\r';                                                            // 12
      USART_DMACmd(USART3, USART_DMAReq_Tx, ENABLE);                                  // 13
      DMA_ClearFlag(DMA1_Stream3, DMA_FLAG_TCIF3 | DMA_FLAG_HTIF3);                   // 14
      DMA_Cmd(DMA1_Stream3, ENABLE);           // note: USART3->SR.TXE requests DMA   // 15

      LCD_uInt16(Seconds, 0x05, 1);            // show Seconds on LCD                 // 16
      for (int i = 0; i<30000000; i++) {};     // waste about 1 s                     // 17
  };
}
```

The seconds counter is incremented and checked against 60 in line 07, then its value is transferred to a new variable x and there sliced into corresponding characters in ASCII to present a two-digit number, followed by a CRLF characters, lines 08 to 12. With this, the string is ready for sending.

Next, transfer requests issued by USART3 are enabled at line 13, followed by clearing all flags associated with the previous transfer at line 14. If these flags were left as they were they would prematurely end the current transfer, since they signal the end of previous transfer; better clear them. These flags are “Transfer Complete” and “Half Transfer”. Finally the DMA1, Stream 3 is enabled at line 15, and with this the transfer starts. All four characters of the prepared string will be transferred to the USART3 character by character immediately after the USART3 is ready to receive them.

The last part of the infinite loop displays the seconds counter at the LCD (line 16) and wastes about one second looping within an empty “for” loop.

b) USART configuration

The configuration of the USART block is almost the same as stated in experiment 14. Both configurations for microcontroller pins and UART3 are exactly the same. However, since we are going to use the DMA transfer instead of interrupts, both lines considering the IRQ settings within the UART and NVIC are gone. The complete listing for configuration of the USART3 is given below.

```
// USART3 initialization, GPIOD 8:9, TX&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_Cmd(USART3, ENABLE);
}
```

c) DMA configuration

The listing for configuration of the DMA block is given below. Firstly, let us recapitulate: the USART3 requests for transfers are to be served by DMA every time TX buffer of this UART becommes empty, therefore from tables at Fig. 29.1 DMA1 block is to be configured, Stream 3, Channel 4.

The listing commences with the declaration of the data structure containing all needed configuration parameters that are to be distributed throughout DMA1 block configuration registers diring the configuration procedure, line 01.

Then the DMA block DMA1 gets its clock signal to become enabled at line 02, and immediatelly gets initialized to a neutral state at line 03.

Data structure declared at line 01 gets initialized next.

- At line 04 the correct Channel is stated, folowed by the initialization of the destination and source address at lines 05 and 06. Note that the address of (&) the USART3-DR gets casted to unsigned integer ((uint32_t)(&USART3->DR)) due to the requirements of the IAR compiler. Similary, the address of the "OutString" gets casted ((uint32_t)(OutString)).
- At line 07 the direction of the transfer is defined; valid options are DMA_DIR_PeripheralToMemory, DMA_DIR_MemoryToPeripheral, and DMA_DIR_MemoryToMemory
- At line 08 the number of transfers is stated, in our case it is 4.
- At lines 09 and 10 we enable or disable the incrementing of the pointers to the source and the destination; in our case we increment the source (points to the character array), and do not increment the destination (the USART3-DR).
- At lines 11 and 12 we define the size of data to be transferred; it is bytes in this case, but valid options are: DMA_PeripheralDataSize_Byte, DMA_PeripheralDataSize_HalfWord, and DMA_PeripheralDataSize_Word
  or alternativelly: DMA_MemoryDataSize_Byte, DMA_MemoryDataSize_HalfWord, DMA_MemoryDataSize_Word.
- At line 13 we opt for "DMA_Mode_Normal" mode of operation for DMA; an alternative is "DMA_Mode_Circular". The difference is that "normal" mode ends when all transfers stated at line 04 are completed, while in "circular" mode next transfer request beyond the number stated at line 04 resets pointers to the initial value and continues the operation.
- At line 14 the priority for this particular Stream is defined; this line has no effect in our experiment, since we have only one DMA stream enabled. Valid optoins here are: DMA_Priority_Low, DMA_Priority_Medium, DMA_Priority_High, DMA_Priority_VeryHigh.
- We are not going to use the FIFO buffer within the DMA block, so we disable it at line 15.
- We are not going to use any interrupts at the end of DMA transmission, and thus line 16 does not matter.
- Each transfer should pass one byte only, from the standpoint of the source and of the destination. This is defined in lines 17 and 18 as "Single" transfers.

Finally, the content of the data structure gets distributed into the registers of the DMA block DMA1 by calling the subroutine "DMA_Init " at line 19. With this, the DMA transfer is completely defined.

```
//DMA1 initialization, TX requests next character to send
void DMA1fromUSART3_init(void) {
DMA_InitTypeDef         DMA_InitStructure;                                          // 01
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA1, ENABLE);                              // 02
```

```
  DMA_DeInit(DMA1_Stream3);                       /* Reset DMA Stream registers */              // 03
  /* Configure DMA Stream */
  DMA_InitStructure.DMA_Channel                 = DMA_Channel_4;                               // 04
  DMA_InitStructure.DMA_PeripheralBaseAddr      = (uint32_t)(&USART3->DR);                     // 05
  DMA_InitStructure.DMA_Memory0BaseAddr         = (uint32_t)(OutString);                       // 06
  DMA_InitStructure.DMA_DIR                     = DMA_DIR_MemoryToPeripheral;                  // 07
  DMA_InitStructure.DMA_BufferSize              = (uint32_t)4;                                 // 08
  DMA_InitStructure.DMA_PeripheralInc           = DMA_PeripheralInc_Disable;                   // 09
  DMA_InitStructure.DMA_MemoryInc               = DMA_MemoryInc_Enable;                        // 10
  DMA_InitStructure.DMA_PeripheralDataSize      = DMA_PeripheralDataSize_Byte;                 // 11
  DMA_InitStructure.DMA_MemoryDataSize          = DMA_MemoryDataSize_Byte;                     // 12
  DMA_InitStructure.DMA_Mode                    = DMA_Mode_Normal;                             // 13
  DMA_InitStructure.DMA_Priority                = DMA_Priority_High;                           // 14
  DMA_InitStructure.DMA_FIFOMode                = DMA_FIFOMode_Disable;                        // 15
  DMA_InitStructure.DMA_FIFOThreshold           = DMA_FIFOThreshold_Full;                      // 16
  DMA_InitStructure.DMA_MemoryBurst             = DMA_MemoryBurst_Single;                      // 17
  DMA_InitStructure.DMA_PeripheralBurst         = DMA_PeripheralBurst_Single;                  // 18
  DMA_Init(DMA1_Stream3, &DMA_InitStructure);                                                  // 19
}
```