11. Interrupts & Timer TIM5

The explanation of interrupt processing was given in chapter 10 (Ports & Interrupts). Here an example of the use of timer and interrupt processing will demonstrate how to make periodic interruptions to the execution of the regular program in such a way that the processor can do some useful work at precisely defined time intervals. In our case the processor will generate two consecutive pulses at port E, bit 8, for every interrupt request.

The block representing the behavior of the microcontroller for this experiment is given in Fig. 1.

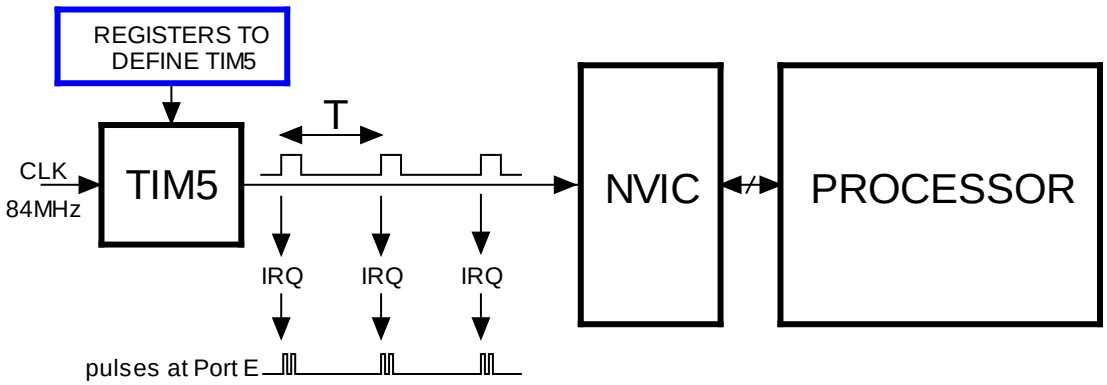

Figure 1: The chain used to implement periodic interrupt requests - simplified

- The timer TIM5 is used for the demonstration. It is the same as previously described timer TIM2; it uses 32-bit counter and has additional hardware to select the input signal and control the counting. The timer TIM5 is driven by a clock signal with a frequency of 84 MHz from inside of the microcontroller. This is also the fastest clock this counter can handle, and is connected to the input of the counter through a previously described set of multiplexers by default.
- The timer TIM5 counts up to a predefined number and then resets back to zero; the return to zero is called update event. The predefined number is stored in register TIM5_ARR (Auto Reload Register). If the content of register TIM5_ARR is zero then the counter counts up to 2^{32}-1. By setting the content of the register TIM5_ARR to 84000 and using the clock signal with the frequency of 84 MHz the time interval T between two consecutive update events equals to 1 ms, as shown in Fig. 1. These update events can be used as interrupt requests.
- The update events are mapped as interrupt requests to the input to the controller NVIC, which must be enabled at the channel for interrupt requests from timer TIM5. This is done by a call to a function “NVIC_EnableIRQ” to enable the controller NVIC.

- The interrupt function must be prepared to be executed on interrupt request. Within the interrupt function the actual work should be done. In our case four statements are needed to toggle port E, bit 8, high-low-high-low. Additionally, the update event is stored in register TIM5_SR (Status Register), bit 0 when update event takes place. This bit must be cleared within the interrupt function to prevent immediate repetition of the same interrupt request.

The order of events is then as follows. When the counter TIM5 content reaches the predefined value stored in the reload register TIM5_ARR, the content of the counter register TIM5_CNT returns to zero triggering the reload event, then the counting continues from zero on. The reload event is saved into the status register TIM5_SR, bit 0, and simultaneously passed to the controller NVIC as an interrupt request from timer TIM5. Since the controller NVIC is enabled to respond to this particular interrupt request, it forces the processor to interrupt the execution of the regular program and starts executing the interrupt function. Within the interrupt function the processor toggles the port E, bit 8, four times to make two consecutive pulses and clears the update event stored in the status register TIM5_SR, bit 0, to acknowledge the execution of the interrupt function. After this the processor continues with the execution of the regular program as if nothing had happened.

The complete listing of the demo program is given in Fig. 2. The program starts with the initialization by enabling the clock for ports A and E, and the clock for timer TIM5. Next a pin on port E is defined as output, here the pulses will be available, and a pin at port A is defined as output, here the LED is connected and will be turned on when the pushbutton S370 is pressed to confirm that the regular program is running.

```
#include "stm32f4xx.h"

int main ()  {

  RCC->AHB1ENR |= 0x0010;       // Enable clock for GPIOE & GPIOA
  RCC->APB1ENR |= 0x00000008;   // Enable Clock for Timer 5

  GPIOE->MODER |= 0x00010000;   // Output pin for time mark at port E
  GPIOA->MODER |= 0x00001000;   // output pin for LED D390

  NVIC_EnableIRQ(TIM5_IRQn);    // Enable IRQ for TIM5 in NVIC

  TIM5->ARR     = 84000;        // Auto Reload Register value => 1ms
  TIM5->DIER   |= 0x0001;       // DMA/IRQ Enable Register - enable IRQ on update
  TIM5->CR1    |= 0x0001;       // Enable Counting

  while (1) {
    if (GPIOE->IDR & 0x01) GPIOA->ODR |=  0x0040;   // press S370 to turn on D390
    else                   GPIOA->ODR &= ~0x0040;
  };
}

void TIM5_IRQHandler(void)
{
  TIM5->SR &= ~TIM_SR_UIF;      // clear IRQ flag in TIM5
  GPIOE->ODR |=  0x0100;        // PE08 up
  GPIOE->ODR &= ~0x0100;        // PE08 down
  GPIOE->ODR |=  0x0100;        // PE08 up
  GPIOE->ODR &= ~0x0100;        // PE08 down
}
```

Figure 2: A listing of the program to utilize interrupt requests at port E, bit 3

The interrupt controller is enabled for interrupt request signals coming from timer TIM5 by a call to function NVIC_EnableIRQ. The initialization is finished by setting-up the timer TIM5. The reload value is written into the reload register TIM5_ARR, and the update event is mapped as a valid interrupt request from this timer by setting the LS bit of the register TIM5_DIER. Finally, the timer is enabled to run by setting the LS bit in control register TIM5_CR.

Within the main loop the state of the pushbutton S370 is checked and the LED390 is turned on when this pushbutton is pressed.

The interrupt function is named as required by the interrupt vector table (TIM5_IRQHandler). The function does not pass any variables, therefore the declaration section of this function uses “void” keywords. Within the function the flag to memorize the update event is first cleared, then the two pulses are generated.