Politecnico di Torino

Digital Systems Electronics

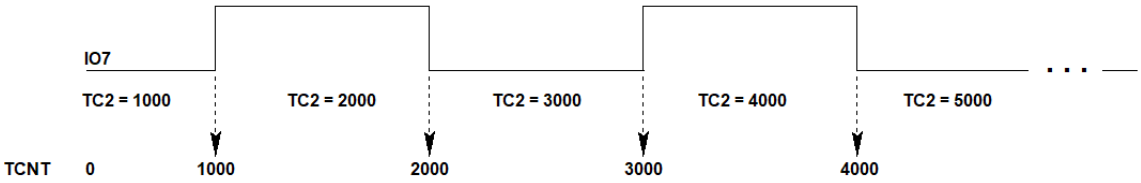
Lab9-STM32

Interrupts

1. Generic Output Compare in Interrupt

NB: The following figure does not refer to the STM32 microcontroller, but to a generic one. The Output Compare principle is exactly the same, but it is required to arrange the scheme to the employed microcontroller (in terms of register names, name of routines, *etc.*).

Let us assume that a square wave on a I/O pin named IO7 is going to be generated *via*-interrupt in output compare, by exploiting TCNT and a Capture Compare Register named TC2. It is assumed that the configuration of ports and timer is already done, such as the enable of timer. Focus is on what is written in the Interrupt Service Routine, accessed when the flag associated to the comparison between TCNT e TC2 is asserted: the flag must be cleared, then IO7 is toggled and TC2 is updated. On the bottom right it is possible to observe that with an unsigned counter output compare timing is not affected by overflow.




```
HALF_PERIOD = 1000;

@interrupt void tc2_isr(void){

      ClearFlag(TC2);
      TogglePin(IO7); // if toggle-on-match mode is not available
      TC2 += HALF_PERIOD;

}
```

TC2 EVOLUTION, ASSUMING THAT TCNT AND TC2 HAVE 16-BIT PARALLELISM AND HALF_PERIOD CAN BE WRITTEN ON 16 BITS TOO

63000 + 1000 = 64000
64000 + 1000 = 65000
65000 + 1000 = 66000 - 65536 = 464

EVEN IF OVERFLOW TAKES PLACE, THE "DISTANCE" BETWEEN THE TC2 VALUES BEFORE AND AFTER OVERFLOW IS EQUAL TO HALF_PERIOD!

2. Interrupts configuration using STM32Cube

We use STM32Cube to obtain the configuration statements for both GPIOx and TIMx peripherals. ADC configuration and programming are exactly the same of the previous laboratory session, with the exception that the Continuous Conversion Mode is disabled in the second exercise. If you want, you can increase the ADC resolution.

We start by launching the STM32CubeIDE tool and creating a new project, with all the peripherals configured with their default Mode.
To configure a TIMx timer, click on the corresponding option in the Peripherals column and select *Internal Clock* as the *Clock Source*. In this laboratory, we will use timers in Output Compare mode. In Figure **1** the two possible Output Compare configurations for Channel 1 of TIM3 (the approach is exactly the same for each channel of each timer) are marked:

- with *Output Compare CHx*, STM32Cube configures the corresponding ***Channel x*** as output pin (green pin in the pinout);
- if output channel is not required, we can select *Output Compare No Output*. This Output Compare mode can be employed for the periodic manipulation of a register.

As an effect of this setting, the TIMx button appears under the *Control* section in the Configuration view (Figure **2**). Double click on the TIMx button and access the configuration window, which includes the *Parameters Settings* menu (Figure **3**). Enter the proper *Prescaler* and *Pulse* fields **for each channel and each timer**. Moreover, the **global interrupt of TIMx must be enabled** in the *NVIC Settings* menu (see Figure **4**).
For the configuration of the user button interrupt, click on the GPIO button in the *System* section of the *Configuration* view. The *Pin Configuration* popup appears, and you must enable the external interrupt in the *NVIC* section (Figure **5**).
To select the pre-emptive options for the different interrupts in the first exercise open the NVIC configuration from the *System Core* category menu. In the *Priority Group* select *"1 bit for pre-emption priority 3 bits for sub-priority"*. Enabling 1 bit for pre-emption and leaving all the interrupt at "0" priority means that every interrupt coming later will block the execution of the previous one. When you are requested to disable pre-emption to see the difference you have multiple options:

- Change the *Priority Group* from the STM32Cube and generate again the code.
- Change preemption bit for TIM3 to 1 (higher value means lower priority) in the STM32Cube and generate again the code.
- Change one of the two values directly in the code avoiding a new generation.

NB: each time you generate the code you may need to correct the bug for the ADC DMA configuration.

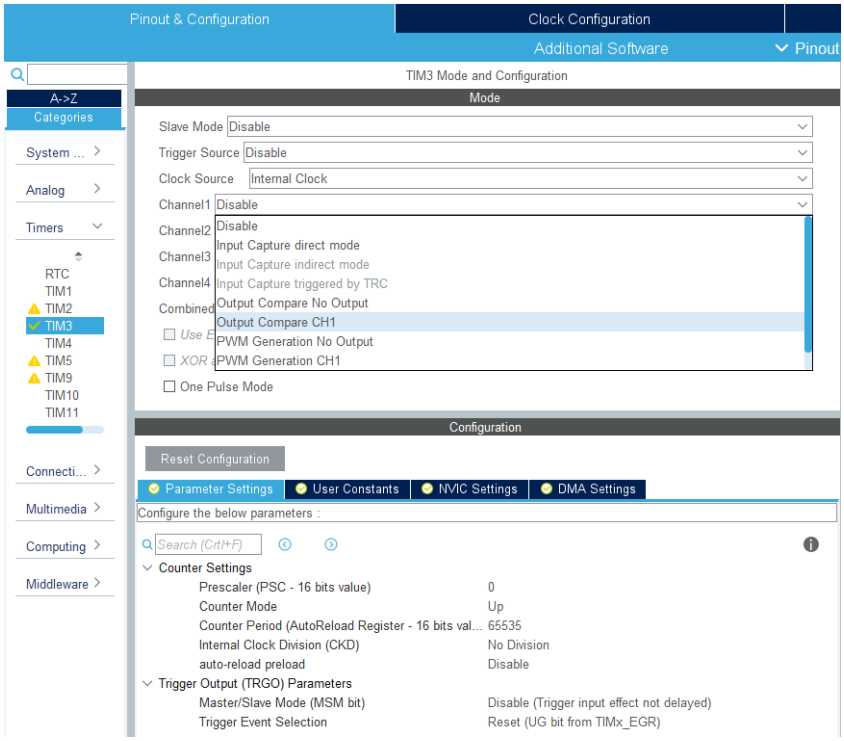


Figure 1: Configurations of Output Compare.

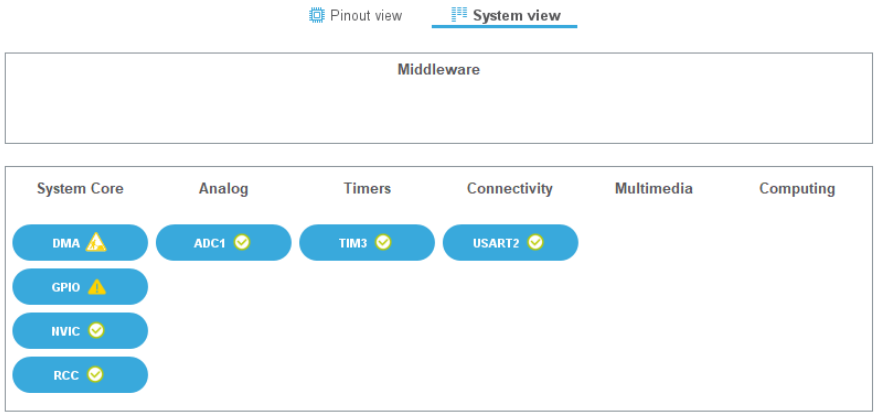


Figure 2: Configuration view.

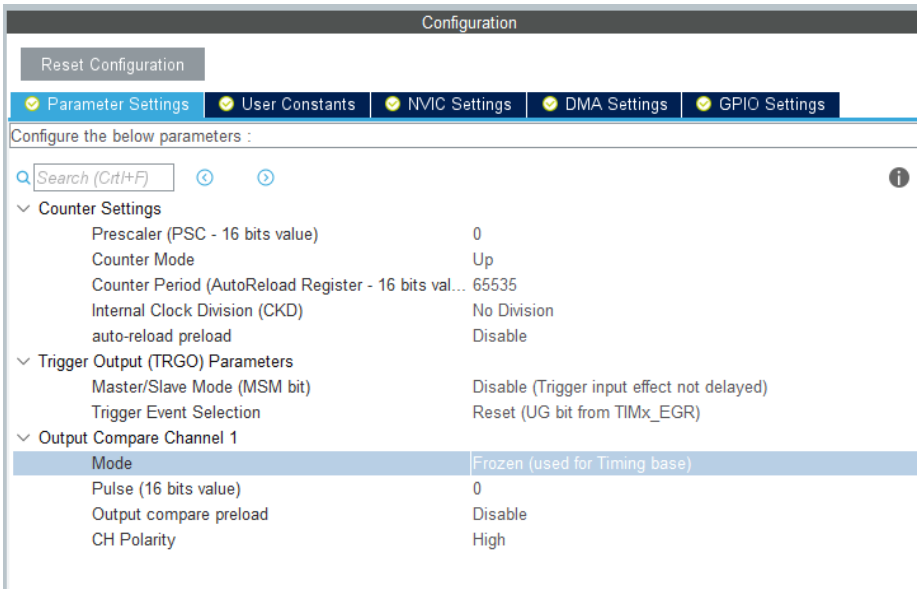


Figure 3: Parameter Settings in the TIM3 Configuration window.

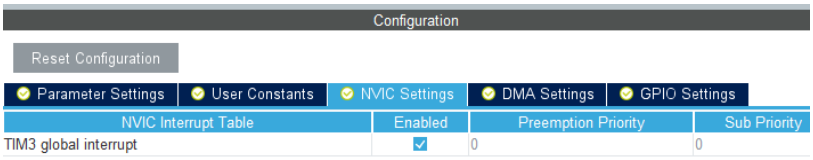


Figure 4: TIM3 interrupt enable.

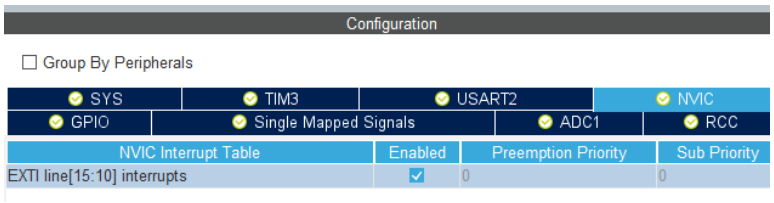


Figure 5: External interrupt enable.

This completes the configuration of TIM3 peripheral. Therefore, we can generate the C code from the main STM32CubeIDE window, by selecting proper names for the project and the folders. You must also select in *Advanced Settings* the Low-Layer (LL) mode for all drivers.
Interrupts are a key aspect of microcontrollers. Each vendor adopts its own way to handle interrupts inside the code. ARM and ST for these MCU use the concept of handler. The interrupt vector is hidden to the user through the abstraction layer (both HAL and LL). In LL each interrupt calls a specific routine of the library that will perform some operation and then call the respective handler. Different interrupts will call the same handler, in this way the code can be simpler, but the user may have to check which particular peripheral caused the interrupt. For example, the same interrupt handler is called by all the channel of a timer. It's up to the user to check and serve the correct peripheral.

3. TIMx programming using Low-Layer (LL) paradigm

Equivalently to the GPIO case, TIM peripherals can be programmed by exploiting the macros at pg. 1689 of the User Manual:

- `LL_TIM_ReadReg(__INSTANCE__,__REG__)`, for reading a value in a TIM register. Two parameters are required:
  - __INSTANCE__: TIM Instance
  - __REG__: Register to be read

  The macro returns the value of the register.

- `LL_TIM_WriteReg(__INSTANCE__,__REG__,__VALUE__)`, for writing a value in a GPIO register. Three parameters are required:
  - __INSTANCE__: TIM Instance
  - __REG__: Register to be written
  - __VALUE__: Value to be written in the register

  The macro does not return any value.

The instance is nothing but the timer TIMx to be employed. The TIMx registers (see pg. 347-369 of the Reference Manual) to be properly programmed for the right execution of the proposed exercises are:

- TIM first Control Register (TIMx_CR1), to enable the counter by setting the CEN bit.
- TIM Status Register (TIMX_SR), whose bits UIF and CCxIF are set by hardware when the counter TIMX_CNT equals TIMx_ARR and TIMx_CCRx respectively. **You must always clear these flags.**
- TIMx DMA/Interrupt Enable Register (TIMx_DIER), whose CCxE bit enables the interrupt for capture compare mode on channel x.
- TIMx Capture/Compare output Enable Register (TIMx_CCER), whose CCxE bit enables the output for capture/compare mode on channel x.
- TIMx capture/compare register (TIMx_CCR1 for channel 1), where the value to be compared with the counter TIMx_CNT must be stored.

It is remembered that Output Compare configuration is automatically done by STM32Cube. Yet, enabling the interrupt in the NVIC is not enough. You need to arm the interrupts for the timer setting the right bit in the TIMx_DIER.

An example of interrupt handler for Timer 3 is reported. **This routine is characteristic of the timer, so it is employed in all the available timer modes (Output Compare, Input Capture, etc.).**

```
void TIM3_IRQHandler(void)
{
      /* Insert your code here */
}
```

In this routine you must detect the channel calling the interrupt, clear the corresponding status register and – in the Output Compare case - update the value in the Capture Compare Register. To detect the caller simply check for the correspondent flag. Use an IF/THEN/ELSE structure to understand the cause of the interrupt. Remember to clear the flag and perform the needed actions. If you want to program with "higher level" Low-Layer (LL) macros, you can read the CCxIF (here we report CC1IF as example, but the approach is the same for all channels) with

```
LL_TIM_IsActiveFlag_CC1(__INSTANCE__)
```

Where `__INSTANCE__` is nothing but the TIMx instance. In order to clear these flags with

```
LL_TIM_ClearFlag_CC1(__INSTANCE__)
```

Notice that the ISR routines are by default automatically generated by STM32Cube in the file stm32f4xx_it.c, a redefinition of the IST in the main.c file generates compiling errors as the function would be present twice in the code. **Modify the stm32f4xx_it.c to properly program the NUCLEO board.**

Hint: global variables (i.e. variables which are defined outside the main() or other functions/handlers) can be used to share information among several functions. Yet, the definition of the same global variables in other files is not allowed and will cause linking error. At this purpose, it could be useful to define the variable as *external* in the other files. Thus, it can be modified also in functions saved in a file which is different from the one defining the variable. An example follows:

File: main.c

```
/* USER CODE BEGIN PV */
int foo_variable;
/* USER CODE END PV */

int main(void)
{
   foo_variable = 1;
   while (1)
   {
   };
}
```

File: stm32f4xx_it.c

```
/* External variables --------------------*/
/* USER CODE BEGIN EV */
extern int foo_variable;
/* USER CODE END EV */

void TIM3_IRQHandler(void)
{
   /* Insert your code here */
   foo_variable++;
}
```

4. External interrupt handler

STM32Cube will automatically generate the following Interrupt Service Routine

```
void EXTI15_10_IRQHandler(void)
{
  /* USER CODE BEGIN EXTI15_10_IRQn 0 */

  /* USER CODE END EXTI15_10_IRQn 0 */
  if (LL_EXTI_IsActiveFlag_0_31(LL_EXTI_LINE_13) != RESET)
  {
    LL_EXTI_ClearFlag_0_31(LL_EXTI_LINE_13);
    /* USER CODE BEGIN LL_EXTI_LINE_13 */

    /* USER CODE END LL_EXTI_LINE_13 */
  }
  /* USER CODE BEGIN EXTI15_10_IRQn 1 */

  /* USER CODE END EXTI15_10_IRQn 1 */
}
```

We can clearly observe that the routine is the same for lines from 15 to 10. The control and clear routines for the interrupt flag associated to line 13 (**related to all GPIO pins with number 13, *i.e.* Px13**) are automatically inserted. You must insert your user code in the right section.

References

[1] https://www.st.com/en/development-tools/stm32cubeide.html
[2] http://www.st.com/content/st_com/en/products/evaluation-tools/product-evaluation-tools/mcu-eval-tools/stm32-mcu-eval-tools/stm32-mcu-nucleo/nucleo-f401re.html#design-scroll
[3] UM1724 User manual (STM32 Nucleo-64 board), Nov. 2016 (description of the Nucleo board).
[4] RM0368 Reference manual, May 2015 (guide to the use of memory and peripherals in the STM32).
[5] Agus Kurniawan , “Getting Started With STM32 Nucleo Development”, 1st Edition, ISBN 9781329075559, 2015
[6] Carmine Noviello, “Mastering STM32”, 2017, available for sale at http://leanpub.com/mastering-stm32
[7] http://www.st.com/en/ecosystems/stm32cube.html?querycriteria=productId=SC2004