POLITECNICO DI TORINO

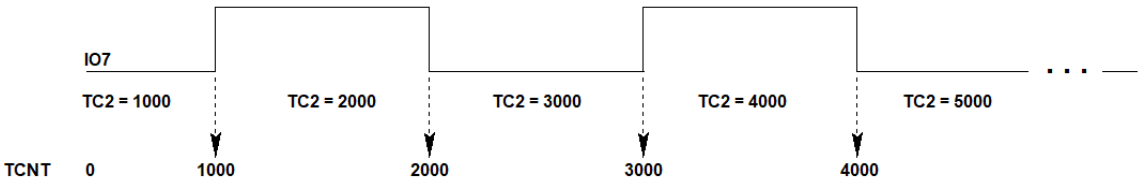
Digital Systems Electronics

Lab9-STM32

Interrupts

Generic Output Compare in Interrupt

Let us assume that a square wave on I/O pin IO7 is going to be generated by using the output compare unit in interrupt mode. To this purpose, we exploit a timer counter TCNT and a Capture Compare Register TC2. We also assume that the configuration of ports and timer is already done and that the timer is already enabled. We need a proper Interrupt Service Routine, which is called as soon as the flag associated to the comparison between TCNT e TC2 is asserted. The necessary tasks in the routine are: (i) clearing the flag, (ii) toggling the IO7 pin, and (iii) updating the TC2 content. The Interrupt Service Routine and the mentioned operations are given in the following picture. The bottom right part of the same picture shows that the overflow of the unsigned counter does not affect the correct timing.




```
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!

Notice that the example in this picture is generic, and therefore the names of the counters, routines and flag do not refer to the STM32 microcontroller

Interrupts configuration using STM32Cube

We use STM32Cube to obtain the configuration code for both GPIOx and TIMx peripherals.
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 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 handling 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). In order to complete the timer configuration, *Toggle on match* for *Mode* field ensures the automatic toggle of output channel (without any explicit toggle in your user code), we select *Frozen* when we do not need any output channel.

To configure the user push button as an external interrupt, goes to the Pinout view, click on the proper pin and select the GPIO_EXTIxx mode. In the STM32F4 MCU, up to 114 GPIOs can be connected to 16 EXTI lines. However, only seven of these lines have an independent interrupt associated with them. All Px0 pins (with x equal to A, B, C, D …) are connected to EXTI0, all Px10 pins are connected to EXTI10 and all Px15 pins are connected to EXTI15. However, as shown in Figure 6, EXTI lines 10 and 15 share the same IRQ inside the NVIC (and hence they are serviced by the same ISR). STM32Cube does the required mapping between the user button pin and an external interrupt request (e.g. line 13) by means of a specific LL driver, called from the function MX_GPIO_Init:

```
LL_SYSCFG_SetEXTISource(LL_SYSCFG_EXTI_PORTC, LL_SYSCFG_EXTI_LINE13);
```

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 project, 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, to avoid a new generation.

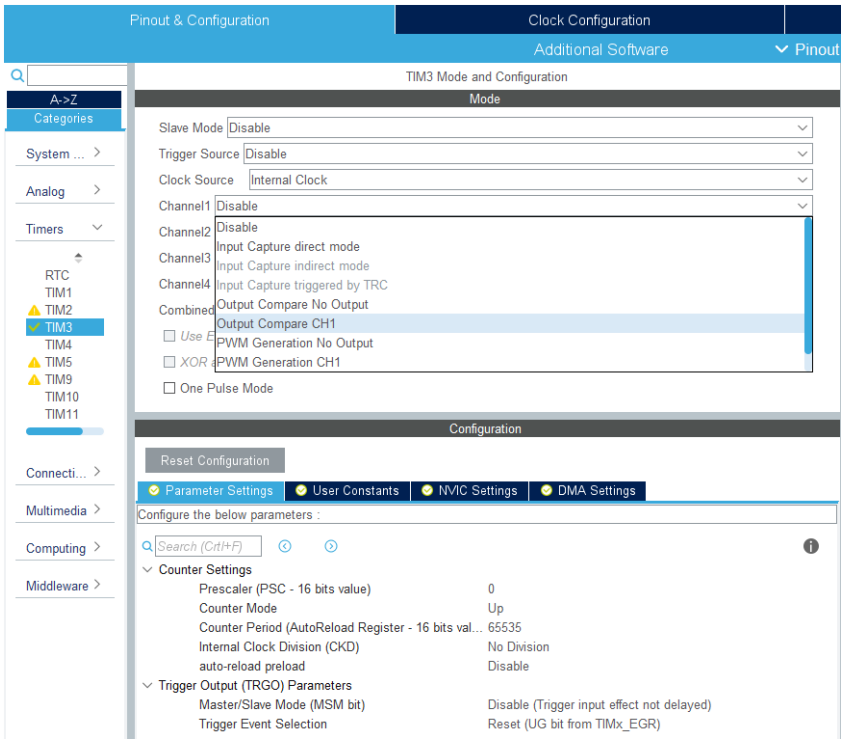


Figure 1: Configurations of Output Compare.

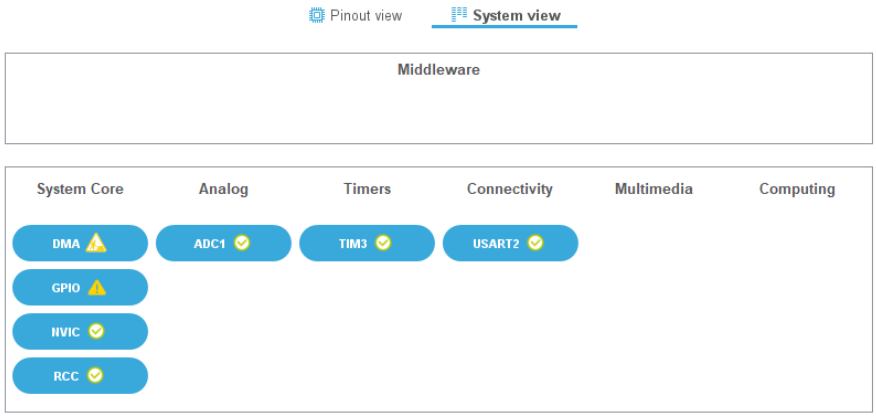


Figure 2: Configuration view.

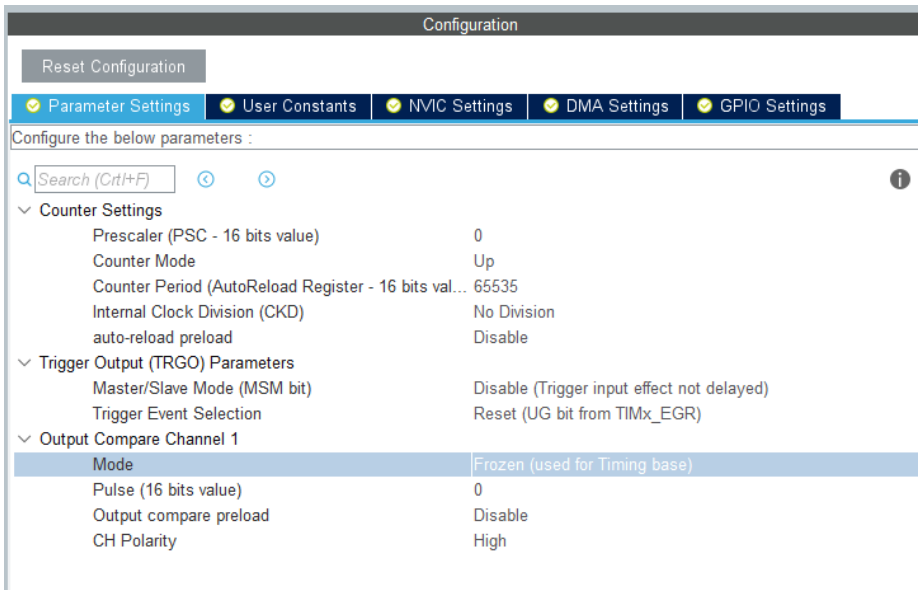


Figure 3: Parameter Settings in the TIM3 Configuration window.

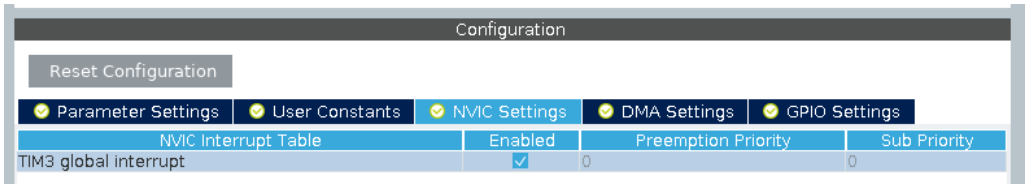


Figure 4: TIM3 interrupt enable.

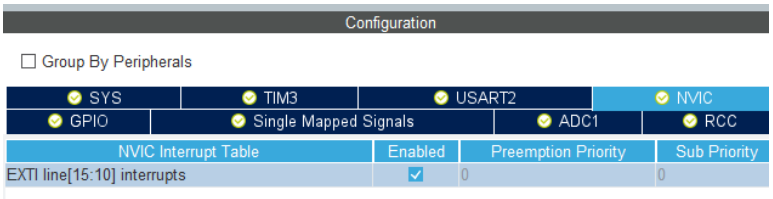


Figure 5: External interrupt enable.

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 micro controllers. Each vendor adopts its own way to handle interrupts inside the code. ARM and ST for these MCUs 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 operations and then call the respective handler. Different interrupts will call the same handler, which simplifies the code; however, the user may have to check which particular peripheral caused the interrupt. For example, all the channels of a timer call the same interrupt handler and the user must check which peripheral needs service, based on the status of the interrupt flags (e.g. CC1IF in register SR).

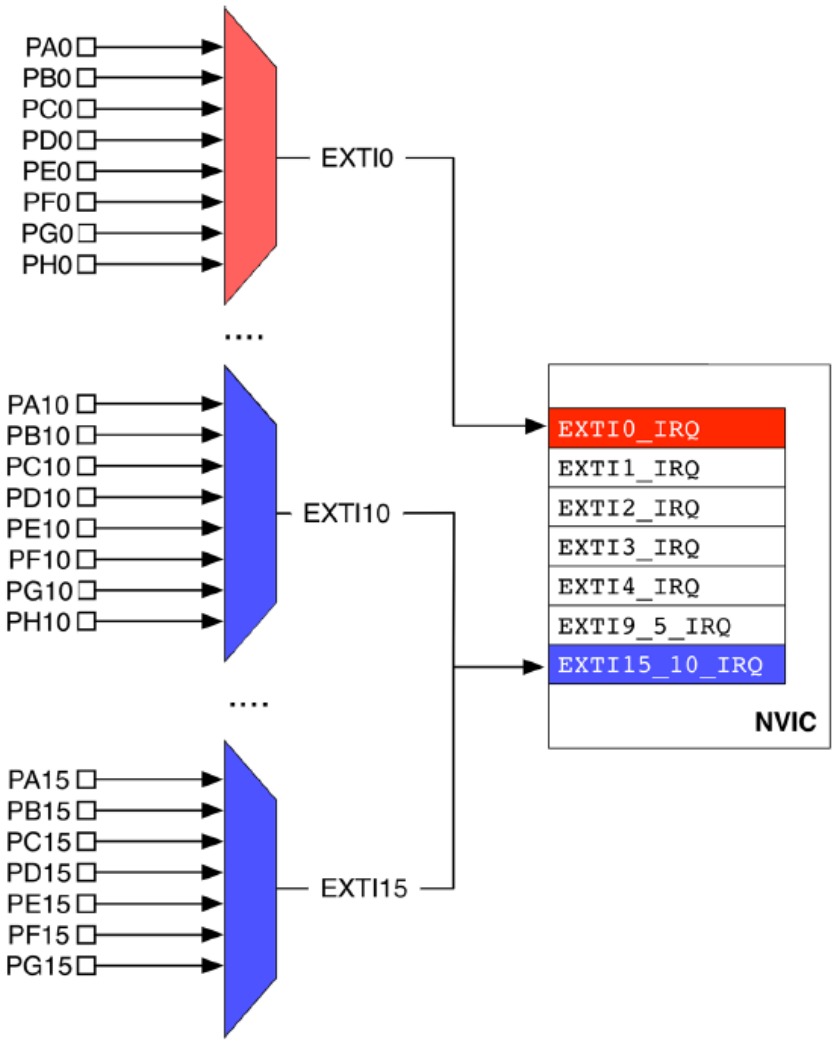


Figure 6: The relation between GPIO, EXTI lines and corresponding ISR.

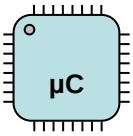
TIMx programming using Low-Layer (LL) paradigm

Equivalently to the GPIO case, TIM peripherals can be programmed by exploiting the macros at pag. 1689 of the User Manual [8] :

- `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 assigned projects 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 registers.**
- TIMx DMA/Interrupt Enable Register (TIMx_DIER), whose CCxIE bit enables the interrupt for capture compare mode on channel x.
- TIMx Capture/Compare Enable Register (TIMx_CCER), whose CCxE bit enables the 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.

The Output Compare configuration can be automatically done by STM32Cube. However, we need to explicitly enable the used interrupt, by setting the proper bits in register TIMx_DIER.

An example of interrupt handler for Timer 3 is reported (see file *stm32f4xx_it.c*). **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. To 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 */
```

File: stm32f4xx_it.c

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

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

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

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 */
}
```

In this code, we can notice the control and clear routines for the interrupt flag associated to line 13 (**related to all GPIO pins with number 13, *i.e.* Px13**), which are automatically inserted by the tool. You must add your own code in the proper section.

ADC configuration for interrupt and external trigger

Several elements of the ADC configuration and programming are the same as in the previous laboratory session. However, in this project, we use the converter in single conversion mode. Therefore, the ADC conversion has to be triggered at each sample we want to obtain. When the conversion is complete, an interrupt must be raised to call the ISR that is in charge to read and process the sample. Moreover, the ADC stops and waits for another trigger.

To enable the interrupt, the ADC1 global interrupt flag must be set in the NVIC Settings window associated with the converter. In the Parameters settings window of the ADC, the other choices must be entered. Particularly we have to set the resolution, disable Scan Conversion Mode, Continuous Conversion Mode, Discontinuous Conversion Mode and DMA Continuous Requests. In addition, we must set the EOC flag at the end of each conversion.

In order to trigger a new conversion from a timer, we must enter the proper settings for both the ADC and the selected timer. As for the ADC, in the Parameters settings window, we can choice the External trigger conversion source; for example, we can select the Timer 4 capture compare 4 event, which means that a new conversion will be started as soon as an input

capture or a capture compare event is detected on the channel 4 of timer 4. We can also decide on the trigger conversion edge.

To configure the selected timer, we have to enter the usual settings, such as the clock source, the operating mode of the selected channel (e.g. output compare), the values of the PSC and ARR registers. If we want to use the output compare event as the trigger, then the "Toggle on match" mode must be selected for the decoded channel.

References

[1] http://www.openstm32.org/
[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
[8] UM1725 User Manual, Description of STM32F4 HAL and LL drivers, DocID025834 Rev 5, Feb. 2017