Politecnico di Torino

Digital Systems Electronics

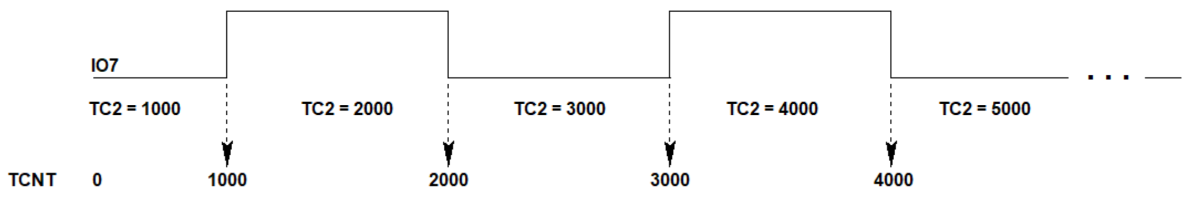
Lab8-STM32

Polling timers and ADC

1. Generic Output Compare in Polling

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*-polling 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 main loop: when the flag associated to the comparison between TCNT e TC2 is asserted, it must be cleared, then IO7 is toggled and TC2 is updated, such that IO7 can be toggled after a semi-period of the square wave. 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;


for(;;){

      if (OCFlagTC2 == 1){

            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!

NB: This procedure is not employed in all the exercises of this laboratory session. Think about when it could be exploited!

2. The TIMx Timer

A general description of the peripheral and a description of the related registers are given in the reference manual **[4]** in chapter 12. In particular, the complete description of the registers related to TIMx is available in Section 12.4, starting at page 283, while a more detailed description of the key registers that we need to handle for a basic use of the timer is available in Section 12.3 of the same document (page 244).
There are five main registers to be considered for a basic use of the timer:

1. Counter register (TIMx_CNT): 16-bit counter. At page 302 of the Reference Manual, we can find the details of the register.
2. Prescaler register (TIMx_PSC): 16-bit prescaler register, used to divide the counter clock frequency by any factor between 1 and 65536. The actual counter clock frequency (CK_CNT) is equal to the frequency of the input clock divided by (TIMx_PSC[15:0] + 1). The input clock can be selected among a few options, as described later in this section.
3. Control register 1 (TIMx_CR1): 16-bit control register containing several configuration fields for TIMx. For a base use of the timer, we have to consider two of them:
    a. CEN (bit 0), Counter enable: we enable the counter by setting this bit to one. The counter starts counting one clock cycle after setting the CEN bit in the TIMx_CR1 register.
    b. DIR (bit 4), Direction: we configure the counter for the up-counting mode by setting this bit to zero.

   The remaining bits of the TIM3_CR1 register can be left to zero, state corresponding to default choices for the other options of the timer.
4. Auto-reload register (TIMx_ARR): 16-bit register set to the actual auto-reload value, *i.e.* the maximum value achievable by the counter. When TIMx_CNT == TIMx_ARR, counter register is reset.
5. Capture/Compare register 1 (TIMx_CCR1), where the value to be compared with the counter TIMx_CNT for Output Compare mode must be stored.

In up-counting mode, the counter counts from zero to the auto-reload value (content of the TIMx_ARR register), then restarts from zero and generates a counter overflow event, also known as update event (UEV). The update event also has the effect of setting the update flag (UIF bit in a further TIMx register, TIMx_SR). If the ARR register is loaded with the value 0, the counter does not start. When we change on the fly the value of ARR, the counter proceeds with the previous value of ARR up to the next UEV, then it starts again with the updated ARR value. Figure **1** shows the behavior of the timer with TIMx_PSC=0x01 and TIMx_ARR=0x36.
In the picture, CK_PSC is the input clock signal, while CK_CNT is the counter clock, with half the frequency as CK_PSC. When the counter register reaches the programmed overflow condition (increment of 0x0036), it is automatically loaded with zero, the UEV pulse is generated (one cycle) and the UIF flag is set.
For example, assuming an internal clock frequency of 8 MHz, we can obtain a LED switching on every second (frequency of the signal equal to 1/2 Hz) by setting TIM3_PSC to 1999(+1) and TIMx_ARR to 4000 (decimal values).

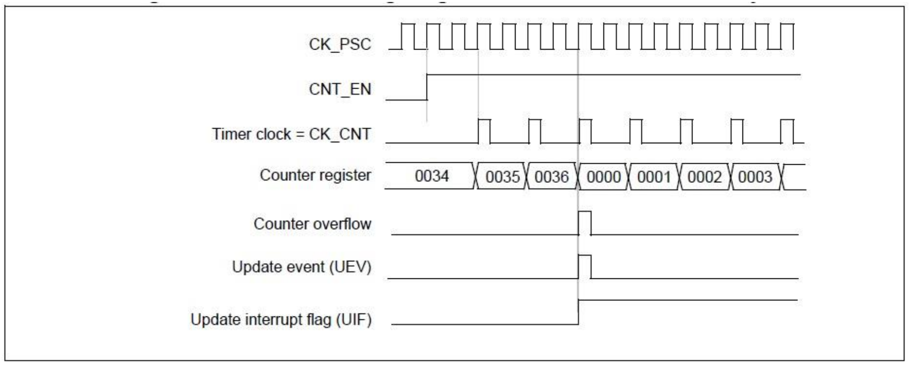


Figure 1: TIMx counter timing diagram in up-counting mode.

2.1. Configuration of TIMx clock

The clock of peripherals is obtained by manipulating the input frequency with some prescalers and PLL circuits which enables the increase or the decrease of the frequency. Then, the main input frequency is distributed through the peripherals. In order to change the frequency of TIMx or other peripherals we can change the values of prescalers and PLL registers.
The default configuration of the clock is shown in Figure 2, where the System Clock is provided by the PLL unit; the default parameters in the PLL leads to a clock frequency of 84 MHz. The values of prescalers and the PLL registers can be modified, a view of a possible clock configuration is given in Figure 3: in the picture, the selected system clock source is the HSI clock, with frequency equal to 16 MHz. In particular, this clock signal is distributed to peripherals connected to APB2 bus, including some of the TIMx timer. With the shown configuration, the TIM1 counter works with a frequency of 16 MHz.
A different frequency is obtained by modifying the available parameters related to the PLL unit and to the multiple prescalers available in the clock management unit. One option to work with these parameters is to use STM32Cube, by operating in the *"Clock Configuration"* section, which generates from the chosen settings the necessary clock configuration code, contained in the *SystemClock_Config* function. See pg. 38-39 of the reference manual to understand which clock drives each peripheral.

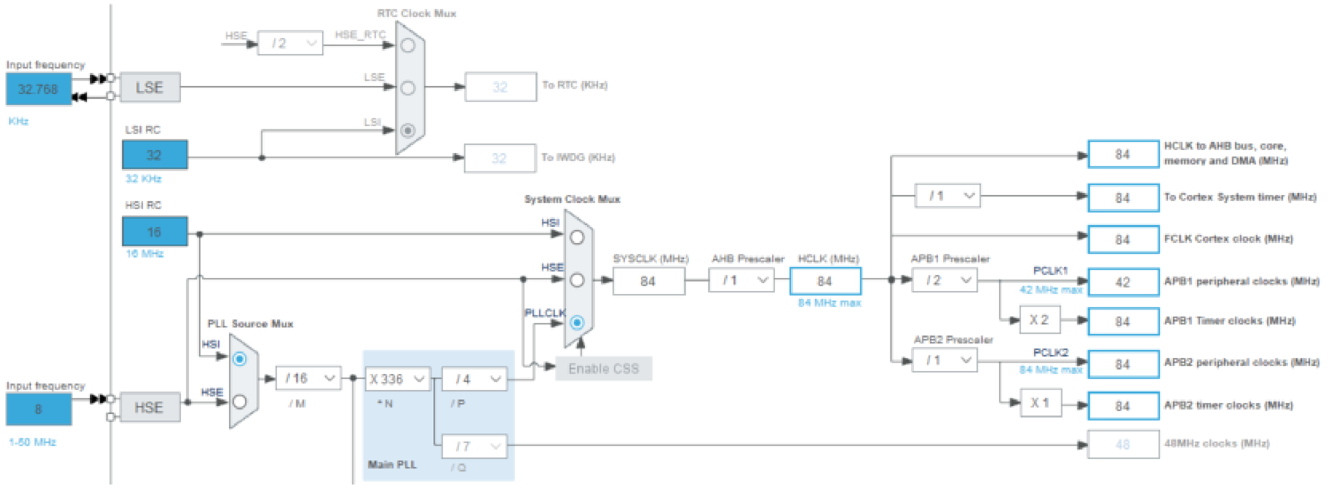


Figure 2: Default Clock Configuration.

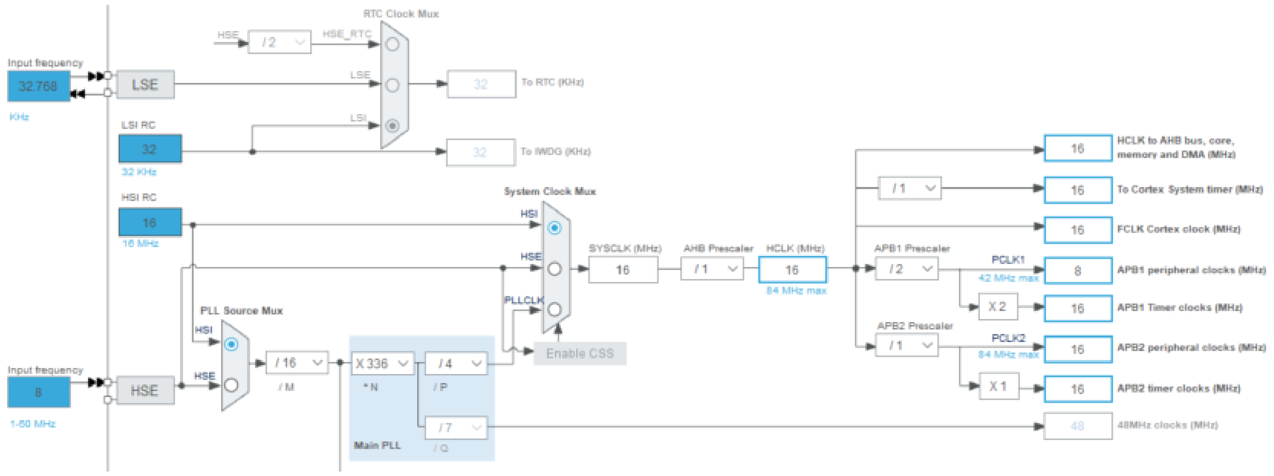


Figure 3: Possible alternative clock configuration.

2.2. TIMx configuration using STM32Cube

We use STM32Cube to obtain the configuration statements for both GPIOx and TIMx peripherals. The timer configuration that is presented in this section can be reliably employed for timers with x = [2;5].
We start by launching STM32CubeIDE tool, we select the correct board from the available list of choices and we create the project by exploiting STM32Cube. At the end of the wizard, a popup will appear (Figure 4); in this laboratory session, we will configure all the peripherals with their default Mode. First, we have to configure the clock signal: open the *Clock Configuration* view. The default configuration is the one shown in Figure 2, where the System Clock source is the PLL clock. The resulting clock frequency distributed to TIMx timer (APB2 peripheral clock) is equal to 84 MHz.
To configure a TIMx timer, click select the "Pinout & Configuration" section. In the left column you can find the Timer category. Expand it to see all the available timers. Select the required timer and a configuration window will appear. Here you can select ***Internal Clock* as the *Clock Source*. This setting must be selected for both polling and Output Compare cases**; moreover, in order to configure the Output Compare functionality for the Channel 1 of TIMx, you must select the correct configuration *(Output Compare No Output* or *Output Compare CH1)* for ***Channel 1***. In order to complete the timer configuration we have to select the operating *Mode* in the Parameter Settings, among the possibilities:

- *Toggle on match* mode ensures the automatic toggle of output channel (**without any explicit toggle in your user code**).
- *Frozen* mode allows using Output Compare mode for other reasons. It is used when we do not need any output channel.

The options to be configured are circled in Figure 5. As an effect of this setting, in the Configuration view, the TIMx button appears under the *System view* (Figure 6).

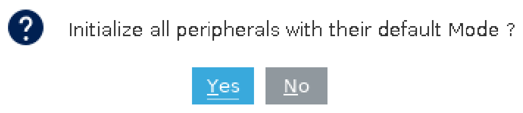
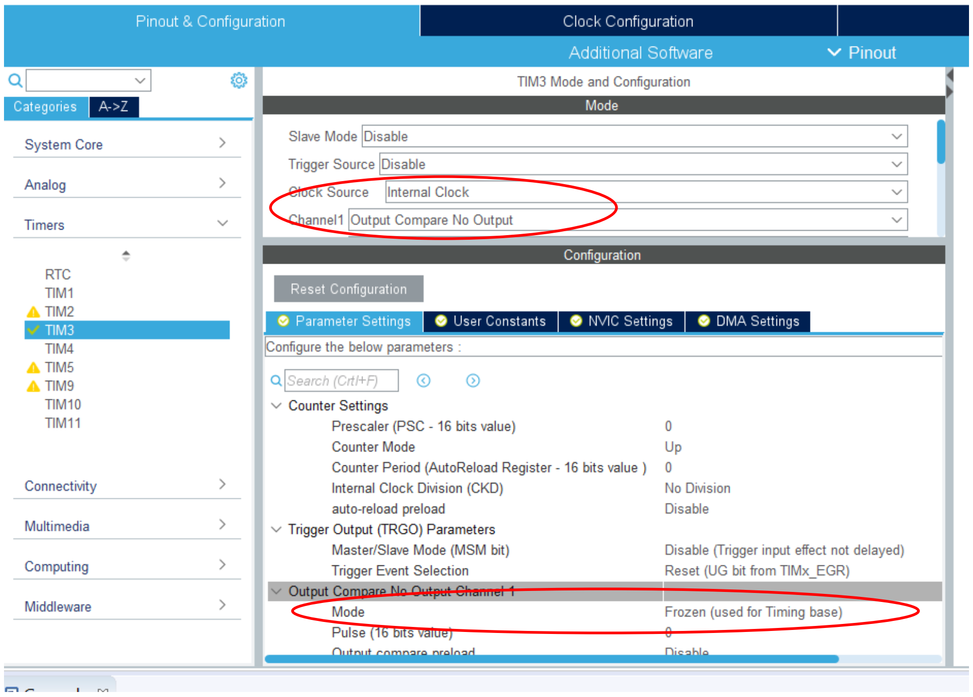

Initialize all peripherals with their default Mode ?

Figure 4: Selection of default Mode for all the peripherals.



```
Slave Mode       Disable
Trigger Source   Disable
Clock Source     Internal Clock
Channel1         Output Compare No Output

Counter Settings
    Prescaler (PSC - 16 bits value)                        0
    Counter Mode                                           Up
    Counter Period (AutoReload Register - 16 bits value )  0
    Internal Clock Division (CKD)                          No Division
    auto-reload preload                                    Disable
Trigger Output (TRGO) Parameters
    Master/Slave Mode (MSM bit)                            Disable (Trigger input effect not delayed)
    Trigger Event Selection                                Reset (UG bit from TIMx_EGR)
Output Compare No Output Channel 1
    Mode                                                   Frozen (used for Timing base)
    Pulse (16 bits value)                                  0
    Output compare preload                                 Disable
```

Figure 5: Default pins' configuration.

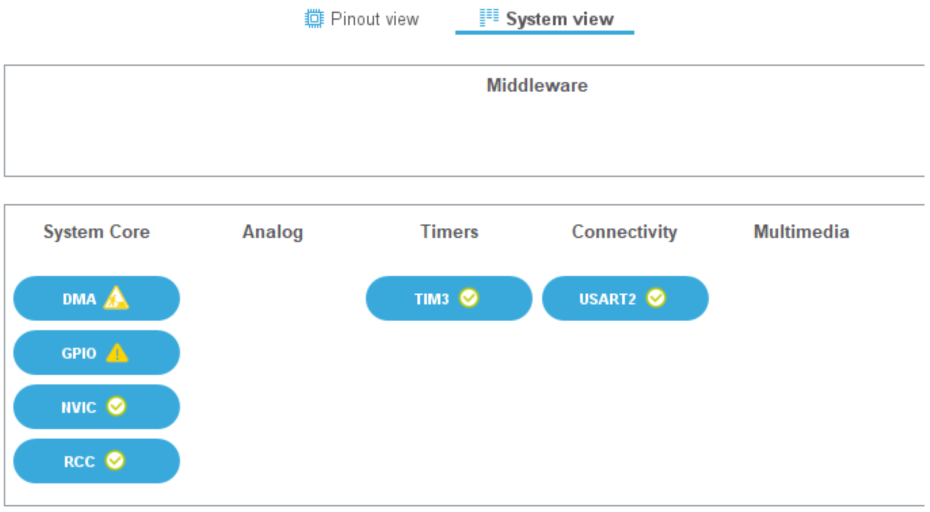
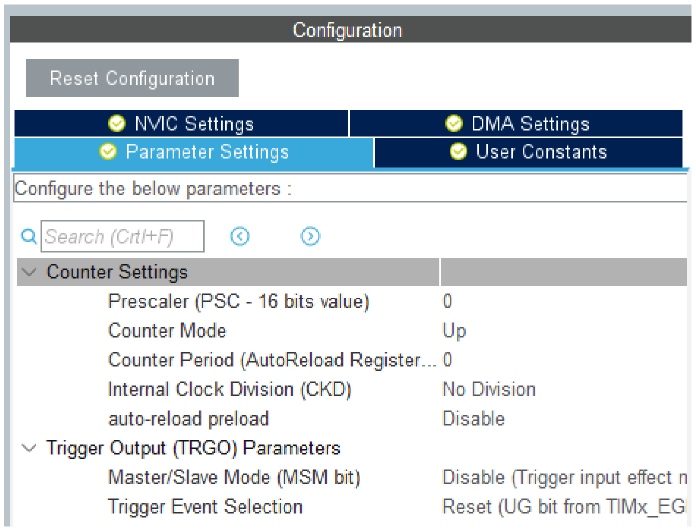


Figure 6: TIM3 appearing in the System View.

Click on the TIMx button and access the configuration window, which includes the *Parameters Settings* menu. For the parameters to be set in this menu, it is required to do a distinction between the different approach you will use in the lab experience:

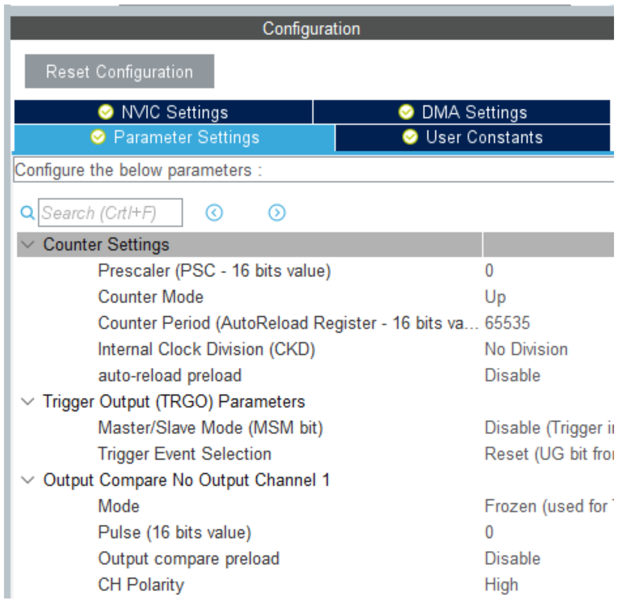
- When polling is done on the value of the counter you have to set only the *Prescaler* and *Counter Period*. Since you will be polling the CNT value, consider setting the TIMx_ARR value to the maximum available.
- When polling is done on TIMx_CCR1, enter the proper *Prescaler* and *Pulse* fields. Insert the remaining parameters according to Figure **8**. The *Pulse* value will be in CCRx register.
- When polling is done on UIF (Figure **7**), enter the proper values for *Prescaler* and *Counter Period* fields (the latter is equivalent to the TIM3_ARR register). Leave the default values for the remaining fields;
- When you use the OC with the automatic toggle of the output pin you have to set all the parameters as in the other OC cases and also the correct *Mode* for the OC function, in this case *toggle on match*.
- Every time you use the OC function remember to update the CCRx register with the new target value each time the match occurs.

It is remarked that Update Interrupt Flag (UIF) and Capture Compare 1 Interrupt Flag (CC1IF) are set whenever the counter TIMx_CNT reaches the values stored in TIMx_ARR and TIMx_CCR1 respectively. The counter value is reset to 0 when the first condition (TIMx_CNT == TIMx_ARR) is reached, thus also implying that counter does not start for TIMx_ARR==0. **In order to generate a periodic square wave in Output Compare**, since the value stored in TIMx_CCR1 must be incremented by a *delta delay* after reaching the equality condition TIMx_CNT == TIMx_CCR1, we must ensure that counter is never reset by reaching the Auto-Reload Register value before arriving to TIMx_CCR1. In fact, if TIMx_ARR was lower than TIMx_CCR1, the counter would never reach the value stored in the Capture Compare Register, with the consequent impossibility to generate a square wave. The condition TIMx_CNT <= TIMx_ARR must always be satisfied and it is ensured by setting the *Counter Period* to the highest value on sixteen bits, *i.e.* 65535.
In order to use the polling mode, the **global interrupt of TIMx must not be enabled** in the *NVIC Settings* menu (see Figure 9).



Counter Settings	
Prescaler (PSC - 16 bits value)	0
Counter Mode	Up
Counter Period (AutoReload Register...	0
Internal Clock Division (CKD)	No Division
auto-reload preload	Disable
Trigger Output (TRGO) Parameters	
Master/Slave Mode (MSM bit)	Disable (Trigger input effect n
Trigger Event Selection	Reset (UG bit from TIMx_EG

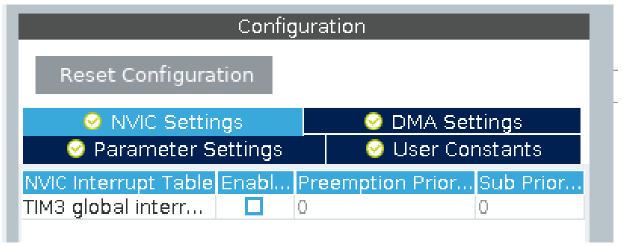
Figure 8: Parameter Settings in the TIM3 Configuration window (polling).



Counter Settings	
Prescaler (PSC - 16 bits value)	0
Counter Mode	Up
Counter Period (AutoReload Register - 16 bits va...	65535
Internal Clock Division (CKD)	No Division
auto-reload preload	Disable
Trigger Output (TRGO) Parameters	
Master/Slave Mode (MSM bit)	Disable (Trigger i
Trigger Event Selection	Reset (UG bit fro
Output Compare No Output Channel 1	
Mode	Frozen (used for
Pulse (16 bits value)	0
Output compare preload	Disable
CH Polarity	High

Figure 7: Parameter Settings in the TIM3 Configuration window (Output Compare).

Figure 9: TIM3 global interrupt disabled.



NVIC Interrupt Table	Enabl...	Preemption Prior...	Sub Prior...
TIM3 global interr...	☐	0	0

This completes the configuration of TIM3 peripheral. Therefore, we can generate the C code from the main STM32Cube 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**.

2.3. 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 TIM 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 CC1IF are set by hardware when the counter TIMX_CNT equals TIMx_ARR and TIMx_CCR1 respectively. **You must always clear these flags.**
- TIM count value (TIMx_CNT), that you could force to 0 if you want to reset the counter.

It is remembered that TIM configurations are automatically done by STM32CubeMX. **However, an automatic code generation does not ensure the absence of bugs. In fact, bugs dramatically influencing the program has been detected. You must replace**

```
TIM_OC_InitStruct.OCState = LL_TIM_OCSTATE_DISABLE;
```

with

```
TIM_OC_InitStruct.OCState = LL_TIM_OCSTATE_ENABLE;
```

when you need the automatic toggle of the pin in OC mode.

3. TIMx debugging tips

This peripheral is designed to run freely with respect to the CPU. Thus, when the CPU is performing its operation, the timers will run: this fact could be a problem during debug. When the core is halted due to a breakpoint the counter will continue their operation and some strange behavior could compromise the debug phase. To overcome this situation specific register can be used to set an automatic stop of peripheral operation when the system is halted. Specifically Debug MCU APB1 freeze register (DBGMCU_APB1_FZ) serve this purpose. You can find more information on the reference manual **[4]** pg 819-822. In LL to set the register for a specific peripheral you can use:

```
LL_DBGMCU_APB1_GRP1_FreezePeriph (uint32_t Periphs)
```

You have to pass the correct value as *periphs*. On pg 1600-1601 of the user manual you can find the list of the available peripherals. For example, if you want to freeze the CAN1 module you could write the following instruction after initializing the peripherals.

```
LL_DBGMCU_APB1_GRP1_FreezePeriph(LL_DBGMCU_APB1_GRP1_CAN1_STOP);
```

4. The Analog-to-Digital converter

A general description of the peripheral and a description of the related registers are given in the reference manual [4] in chapter 11. In particular, the complete description of the registers related to TIMx is available in Section 11.12, starting at page 228, while a more detailed description of the key registers that we need to handle for basic use of the timer is available in Section 11.3 of the same document (page 213).

The ADC is implemented as a Successive Approximation Register ADC. This peripheral can be connected to 16 multiplexed input channels, allowing to measure signals from external sources. A few internal channels are also available. For example, a first channel is connected to the internal temperature sensor, a second one to the internal reference voltage and a third one to the external power supply. It is possible to organize the conversions in two groups: regular and injected. A group consists of a sequence of conversions that can be done on any channel and in any order. The sixteen input channels are fixed and bound to specific MCU pins; moreover, they can be logically reordered to form custom sampling sequences (see pg. 214 of the Reference Manual).

The A/D conversion of the channels can be performed in single, continuous, scan or discontinuous mode. The result of the ADC is stored into a left or right-aligned 16-bit data register. Moreover, the A/D conversion process can be triggered by software or by a variable number of input sources. In single conversion mode, the ADC performs one conversion; at the end of the conversion, a flag is set (EOC). In continuous conversion mode, the ADC starts a new conversion as soon as it finishes one. The ADC needs a stabilization time (t_{STAB}) before it starts converting accurately. After the start of the ADC conversion and after 15 clock cycles, the EOC flag is set and the 16-bit ADC data register contains the result of the conversion (Figure 10).

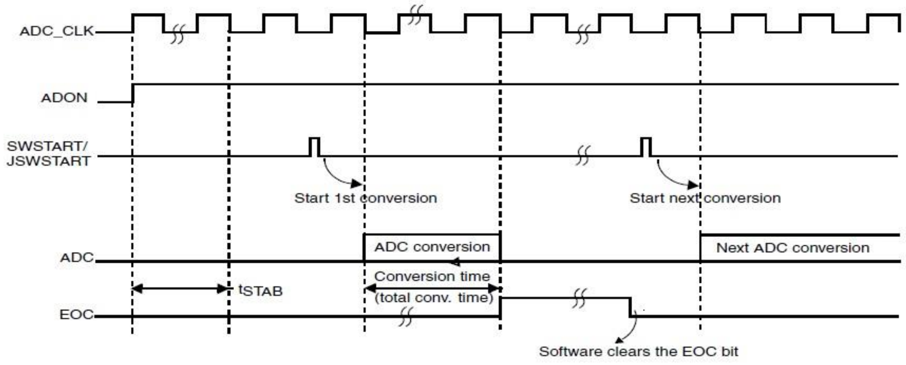


Figure 10: ADC timing diagram.

In scan mode, the ADC scans all the channels selected in the related configuration registers and performs a single conversion for each channel of the group. After each end of conversion signal, the next channel in the group is converted automatically. If the conversions are slow enough, the conversion sequence can be handled by the software: each time a conversion is complete, EOC is set and the ADC_DR register can be read. The ADC peripheral can be driven in three modes: polling, interrupt and Direct Memory Access (DMA) mode.

4.1. ADC configuration using STM32Cube

To configure the Analog-to-Digital converter, as for GPIOx and TIMx cases, click on *Analog > ADC1* option in the left column of the STM32CubeMX perspective, then select the required inputs. You can now configure the ADC (Figure 11 and Figure 12). An eight-bit conversion is enough for our purposes. You can increase the resolution in order to have more accurate results but remember that a higher resolution implies a higher conversion latency. Moreover, the Continuous Conversion Mode must be enabled for starting conversion at the end of another one, without inserting any **additional** ADC enable in your code after the first one. Right alignment ensures that the value stored in the ADC Data Register (DR) is already aligned in the expected way, thus avoiding any swapping between pairs of bits. Since we work with a single channel, Scan Mode must be disabled; interrupts must not be enabled since we work in polling mode.

According to the reference manual (pg. 232), when the End Of Conversion Selection bit EOCS is equal to 0, EOC is set at the end of each sequence of regular conversions. From a practical point of view, EOCS must be cleared or set for continuous or single conversions respectively [6]. After configuring the ADC in LL mode, you can generate code.

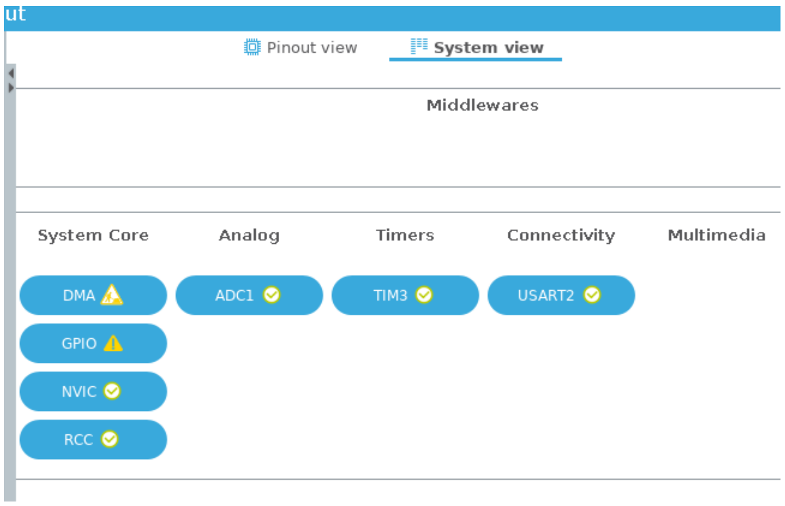
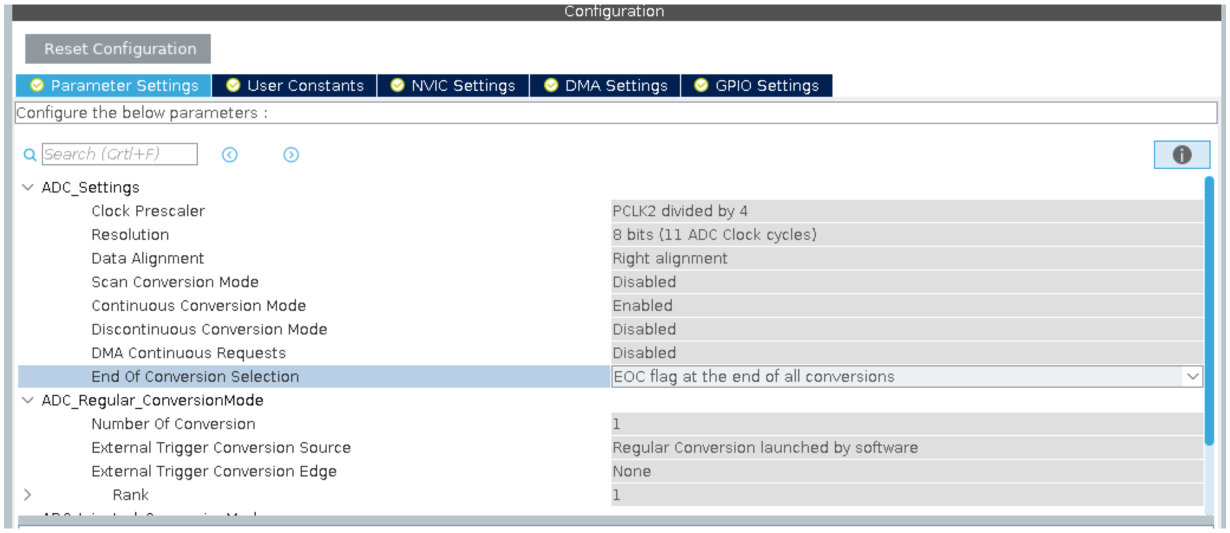


Figure 11: Configuration window after selecting ADC1.



Configure the below parameters :

ADC_Settings	
Clock Prescaler	PCLK2 divided by 4
Resolution	8 bits (11 ADC Clock cycles)
Data Alignment	Right alignment
Scan Conversion Mode	Disabled
Continuous Conversion Mode	Enabled
Discontinuous Conversion Mode	Disabled
DMA Continuous Requests	Disabled
End Of Conversion Selection	EOC flag at the end of all conversions
ADC_Regular_ConversionMode	
Number Of Conversion	1
External Trigger Conversion Source	Regular Conversion launched by software
External Trigger Conversion Edge	None
Rank	1

Figure 12: ADC Configuration.

4.2. ADC programming using Low-Layer (LL) paradigm

Equivalently to the GPIO case, ADC1 peripherals can be programmed by exploiting the macros at pg. 1689 of the User Manual:

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

  The macro returns the value of the register.

- `LL_ADC_WriteReg(__INSTANCE__,__REG__,__VALUE__)`, for writing a value in a ADC register. Three parameters are required:
  - __INSTANCE__: ADC Instance
  - __REG__: Register to be written

- __VALUE__: Value to be written in the register

The macro does not return any value.

The ADC instance is always ADC1. The ADC1 registers (see pg. 228-240 of the Reference Manual) to be properly programmed for the right execution of the proposed exercises are:

- ADC second Control Register (ADC_CR2), whose ADON and SWSTART bits must be set in order to turn on the ADC and to start the conversion. **You must always pay attention to the notes in a datasheet, particularly in this case, please spend one more minute for reading carefully pg. 231 of the Reference Manual.**
- ADC Status Register (ADC_SR), whose EOC bit is set when conversion is ended. You must always reset it.
- ADC regular Data Register (ADC_DR), storing the last acquired value.

It is remembered that ADC configurations are automatically done by STM32Cube. However, an automatic code generation does not ensure the absence of bugs. In fact, bugs dramatically influencing the program has been detected. **IN SOME VERSIONS, you must replace**

```
ADC_REG_InitStruct.DMATransfer = LL_ADC_REG_DMA_TRANSFER_LIMITED;
```

with

```
ADC_REG_InitStruct.DMATransfer = LL_ADC_REG_DMA_TRANSFER_NONE;
```

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