

**POLITECNICO
DI TORINO**

Digital Systems Electronics

Lab8-STM32

Polling, timers and ADC

The TIMx Timer

A general description of the peripheral and the related registers is 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).

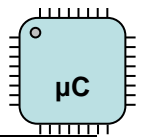
There are four main registers to be considered for a basic use of the timer:

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 it restarts from zero and generates a counter overflow event, also known as update event (UEV). With enabled interrupt, 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 999 (+1) and TIMx_ARR to 07999 (decimal values).

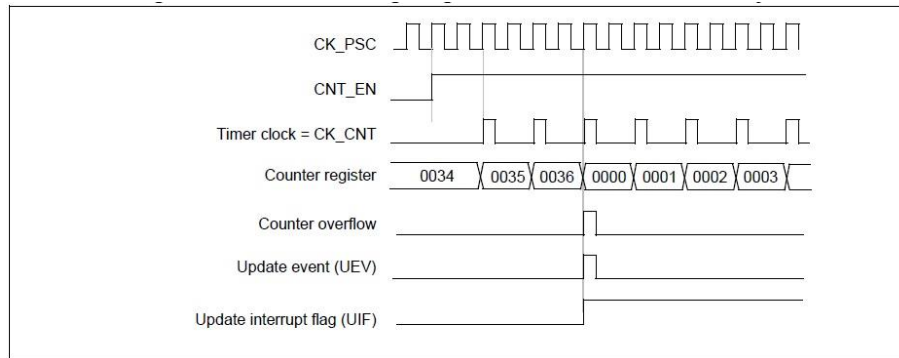


Figure 1: TIMx counter timing diagram in up-counting mode.

Configuration of TIMx clock

The clock of peripherals is obtained by manipulating the input frequency with some prescalers and PLL circuits, which enable the increase or 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 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.

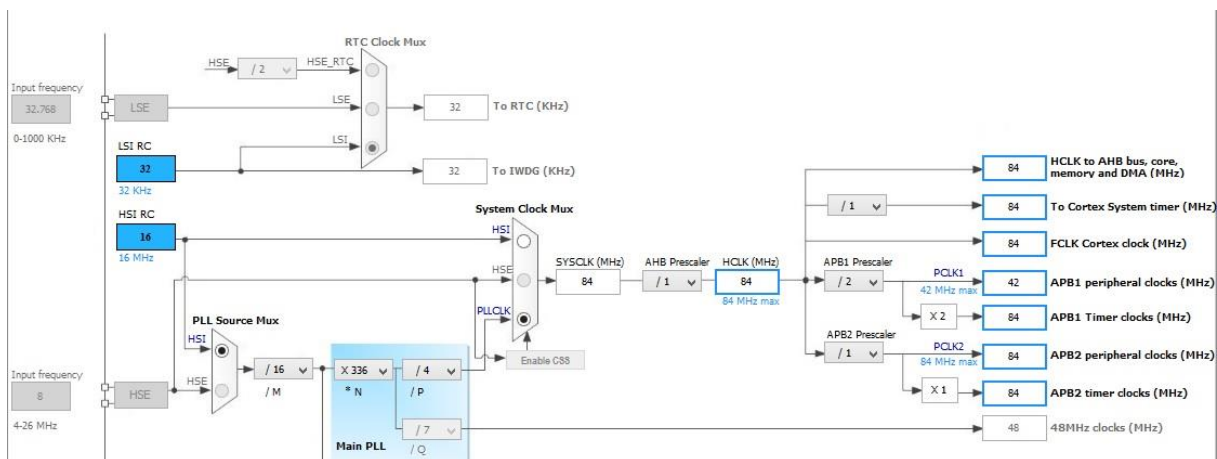


Figure 2: Default Clock Configuration.

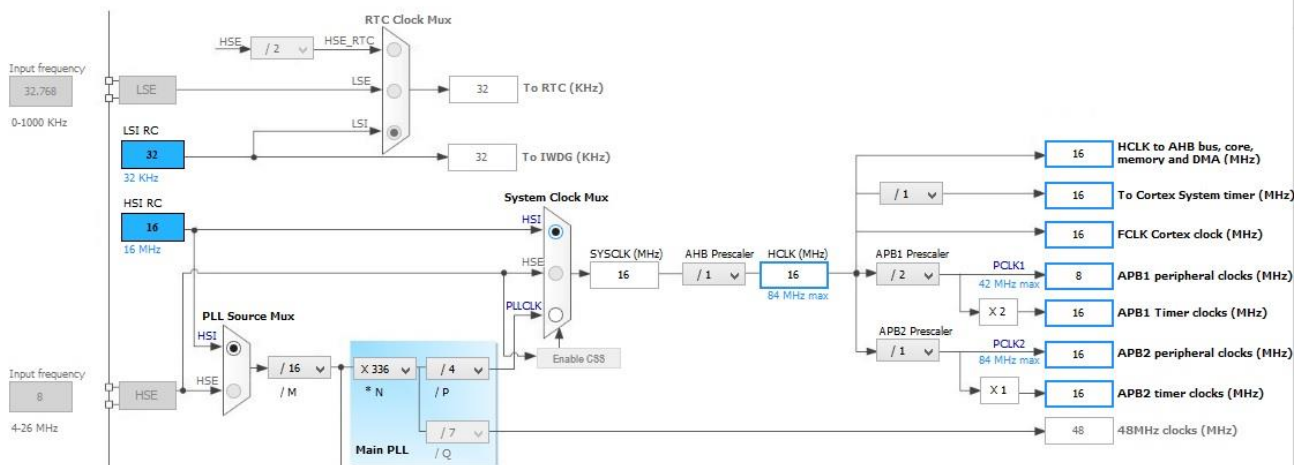
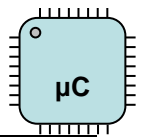


Figure 3: Possible alternative clock configuration.

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 the STM32CubeIDE tool then 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 following options:

- *Frozen* mode allows using Output Compare mode for other reasons. It is used when we do not need any output channel.
- *Toggle on match* mode ensures the automatic toggle of output channel, without any explicit toggle statement in the user code. However, in this case, the OC output pins must be explicitly enabled for the used channels (see register TIMX_CCER).

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).

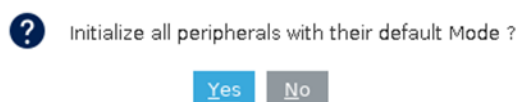


Figure 4: Selection of the default Mode for all the peripherals.

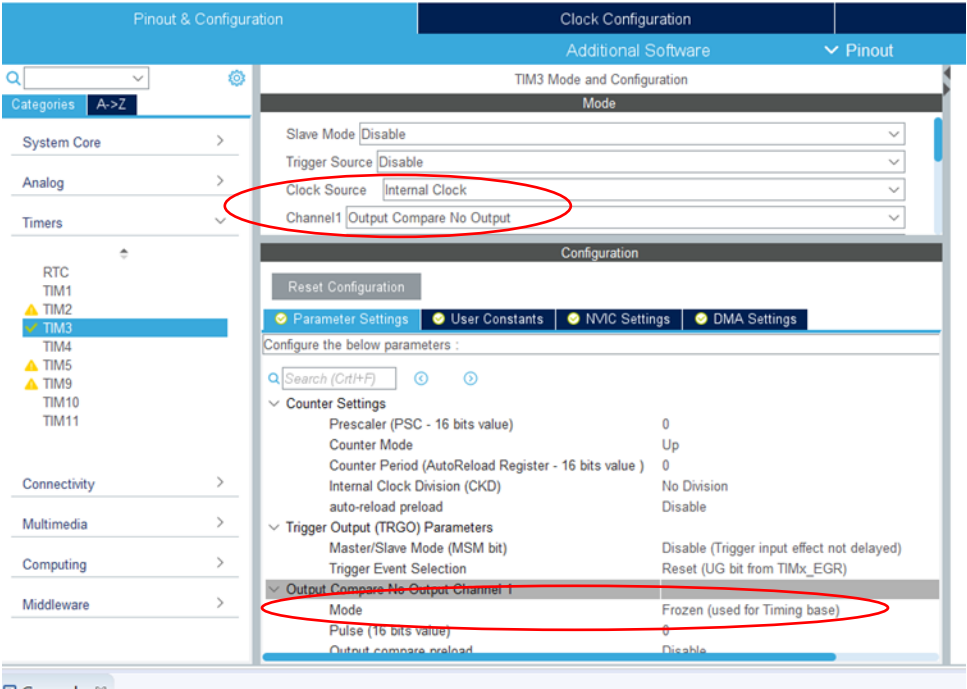
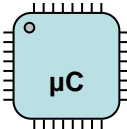


Figure 5: Default pins' configuration.

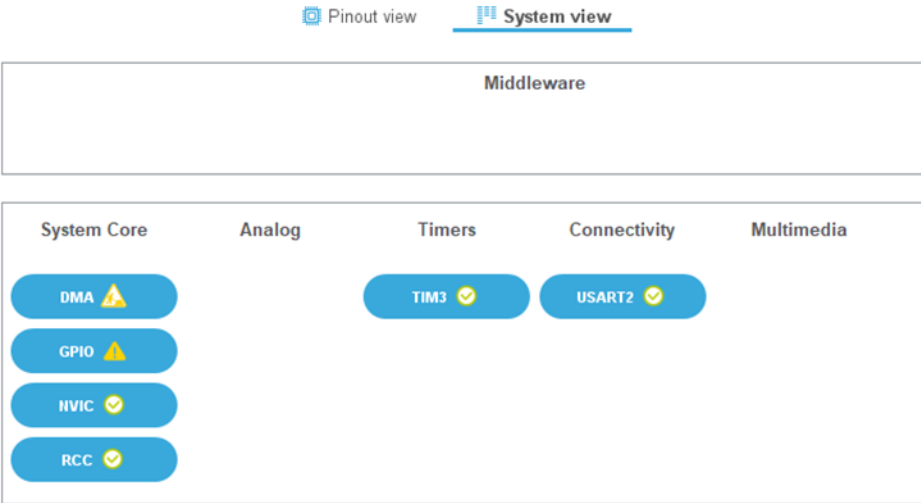
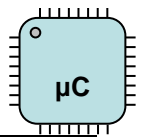


Figure 6: TIM3 appearing in the Control section.



Click on the TIMx button and access the configuration window, which includes the *Parameters Settings* menu. As for the parameters to be set in this menu, it is required to do a distinction between the different approaches you will use in the laboratory session:

- 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 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 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 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 the Update Interrupt Flag (UIF) and the Capture Compare 1 Interrupt Flag (CC1IF) are set whenever the counter TIMx_CNT reaches the values stored in TIMx_ARR and TIMx_CCR1 respectively. Moreover, when the condition TIMx_CNT == TIMx_ARR is reached, the TIMx_CNT counter is reset to 0, while it continues with regular increments after detection of the condition TIMx_CNT == TIMx_CCR1.

Therefore, to generate a periodic square wave with a polling on the UIF flag, it is enough to properly set the TIMx_PSC and TIMx_ARR registers at the beginning. The counter value will increase up to the TIMx_ARR value and then automatically restart from 0.

On the contrary, if the polling is done on the CC1IF flag (using the output compare method, the TIMx_CCR1 register must be not only initialized, but also incremented by an adequate *delta delay* at each detection of the condition TIMx_CNT == TIMx_CCR1. Moreover, we must avoid the automatic reset to 0 occurring on the TIMx_CNT counter when the condition TIMx_CNT == TIMx_ARR is detected. This event is generated if TIMx_ARR is lower than TIMx_CCR1 and it would prevent the counter from reaching the value stored in the Capture Compare Register. In order to guarantee that TIMx_CNT <= TIMx_ARR, we must initialize the *Counter Period* (TIMx_ARR) 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).

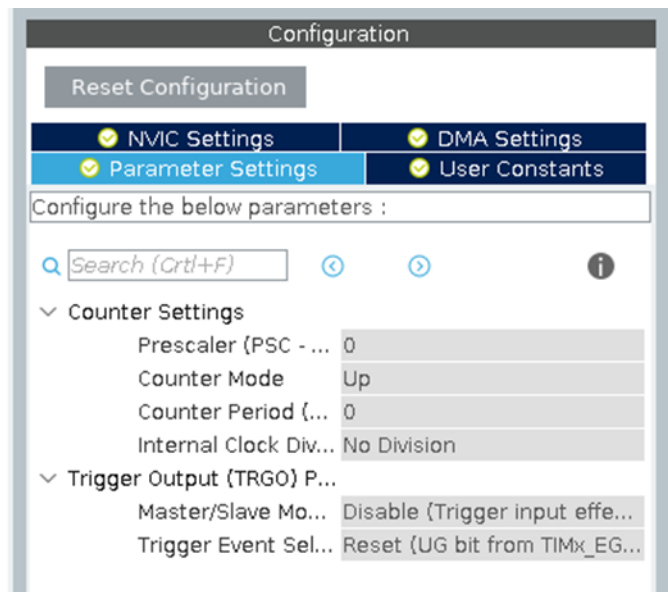


Figure 7: Parameter Settings in the TIM3 Configuration window (polling).

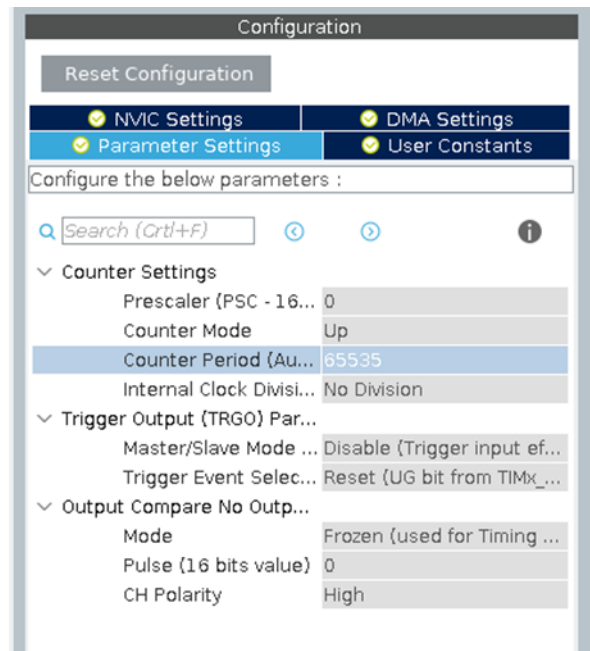
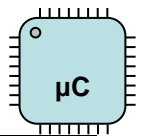


Figure 8: Parameter Settings in the TIM3 Configuration window (Output Compare).

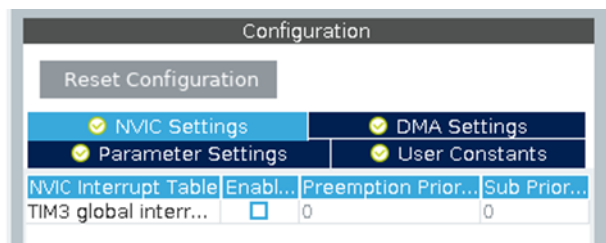


Figure 9: TIM3 global interrupt disabled.

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**.**

Generic Output Compare in Polling

Let us consider an output compare application, in a generic MCU. In particular, let us assume that a square wave on an I/O pin, named IO7, is going to be generated *via*-polling, in output compare mode, by exploiting the timer TCNT and the Capture Compare Register TC2. We assume that the configuration of ports and timer is already done, as well as the enable of the timer. In the main loop, when the flag associated to the comparison between TCNT e TC2 is asserted, we have to clear it; then, we must toggle IO7 and update TC2, such that IO7 can be toggled again after a semi-period of the wanted square wave. Figure 10 shows the generated waveform, with the values assigned to TC2 at each half-period. On the bottom part of the Figure, the key statements required in the main loop are given, together with the progressive update of TC2, which is not affected by the register overflow.

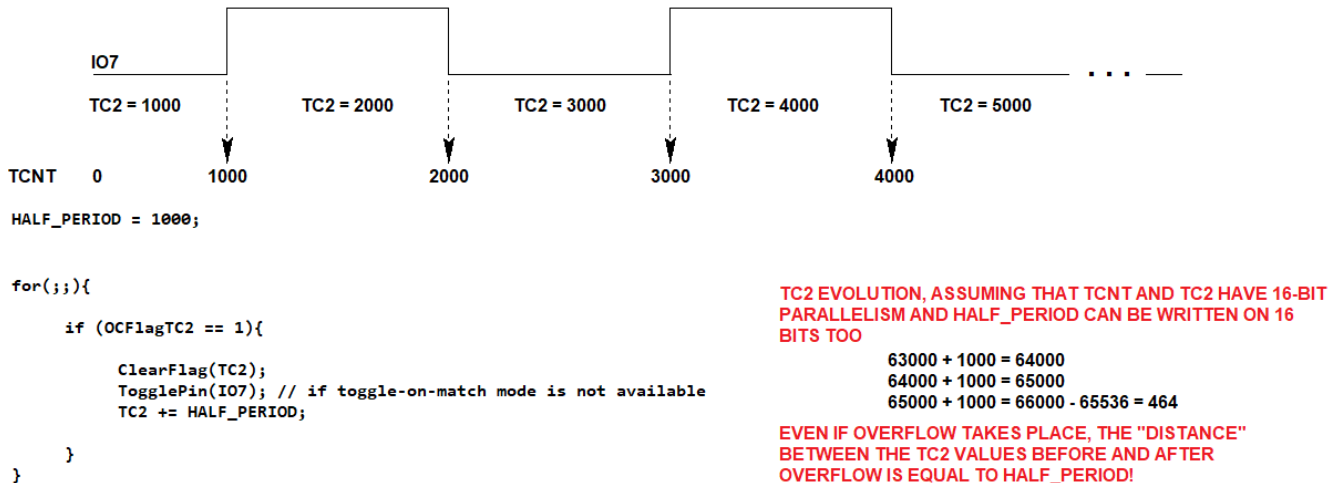
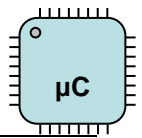


Figure 10: Output Compare Mode (register and flag names here are not referred to the STM32).

TIMx programming using Low-Layer (LL) paradigm

Equivalently to the GPIO case, TIM peripherals can be programmed by exploiting the macros described at pg. 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 TIM register. Three parameters are required:
 - `__INSTANCE__`: TIM Instance
 - `__REG__`: Register to be written
 - `__VALUE__`: Value to be written in the register

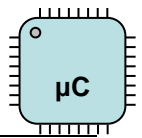
The macro does not return any value.

The instance is nothing but the timer TIMx to be employed. The TIMx registers (see pg. 347-369 of the Reference Manual [4]) to be properly programmed for the right execution of the proposed 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 CC1IF are set by hardware when the counter TIMx_CNT equals TIMx_ARR and TIMx_CCR1 respectively. **You must always clear these registers.**
- TIM count value (TIMx_CNT), that you could force to 0 if you want to reset the counter.

The TIMx Capture/Compare Enable Register (TIMx_CCER) includes the CC1E bit (Capture/Compare 1 output enable), which enables the capture compare mode on channel 1 output pin.

The complete list of register names defined for the TIM peripheral can be found in file *stm32f401xe.h*, available in the CMSIS library (path: *Drivers/CMSIS/Device/ST/STM32F4xx/Include*).



TIM configurations are automatically done by STM32Cube. However, an automatic code generation does not ensure the absence of bugs. In this project, an error might be introduced by the tool (actually, it depends on the tool version): look at the generated code and, if the following statement is detected

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

replace it with the correct version

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

to automatically toggle of the pin in OC mode.

Debugging timer applications

The TIMx peripheral is designed to run freely with respect to the CPU. Thus, the timers are normally running independently of the current CPU task, and this can cause troubles during the debugging. When the core is halted, for example as an effect of a reached breakpoint, the timer continues its counting and wrong behaviors can be observed. To overcome this problem, a dedicated register can be exploited to set an automatic stop of the peripheral (the timer, in this case) when the system is halted. Specifically, the Debug MCU APB1 freeze register (DBGMCU_APB1_FZ) serves 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);
```

The Analog-to-Digital converter

A general description of the ADC 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 ADC registers related 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 a 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, such as for example a channel connected to the internal temperature sensor, one for the internal reference voltage and one for 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 multiple input sources. In single conversion mode, the ADC performs one conversion; at the end of the conversion, a flag is set (EOC) and an interrupt is generated (if enabled). In continuous conversion mode, the ADC starts a new conversion as soon as it finishes the current 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 11).

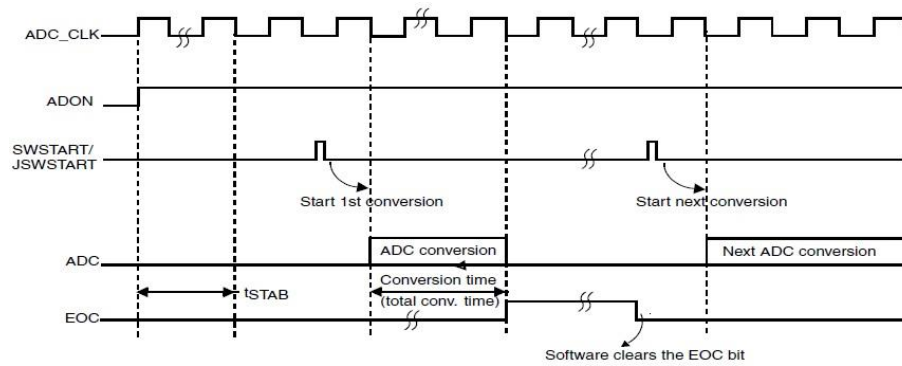
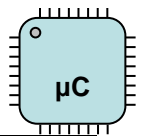


Figure 11: ADC timing diagram.

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, 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 DMA mode.

ADC configuration using STM32Cube

As for the GPIOx and TIMx cases, to configure the Analog-to-Digital converter click on ADC1 option in the Peripherals column and then select one of the inputs. You can now configure the ADC (Figure 12 and Figure 13). An eight-bit conversion is sufficient for our purposes. You can increase the resolution in order to have more accurate results, but be aware 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.

As for the “End of Conversion Selection” field, we have to consider the description given in the reference manual (pg. 232): by setting EOCS = 0 in the ADC control register 2 (ADC_CR2), the EOC (End Of Conversion) bit is automatically set at the end of each sequence of regular conversions. Therefore, the EOC flag can be used in polling mode to determine when a conversion is completed. Thus, we select *EOC flag at the end of all conversions*, which will clear the EOCS bit.

Finally, in the “ADC_Regular_ConversionMode” fields, we select 1 as the number of conversions, and *Regular Conversion launched by Software* as the “External Trigger Conversion Source” [6]. After configuring the ADC in LL mode, you can generate code.

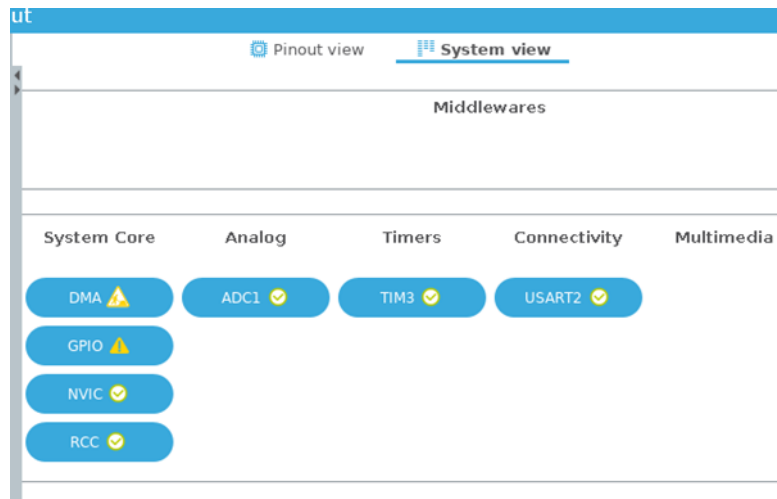
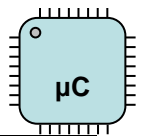


Figure 12: Configuration window after selecting ADC1.

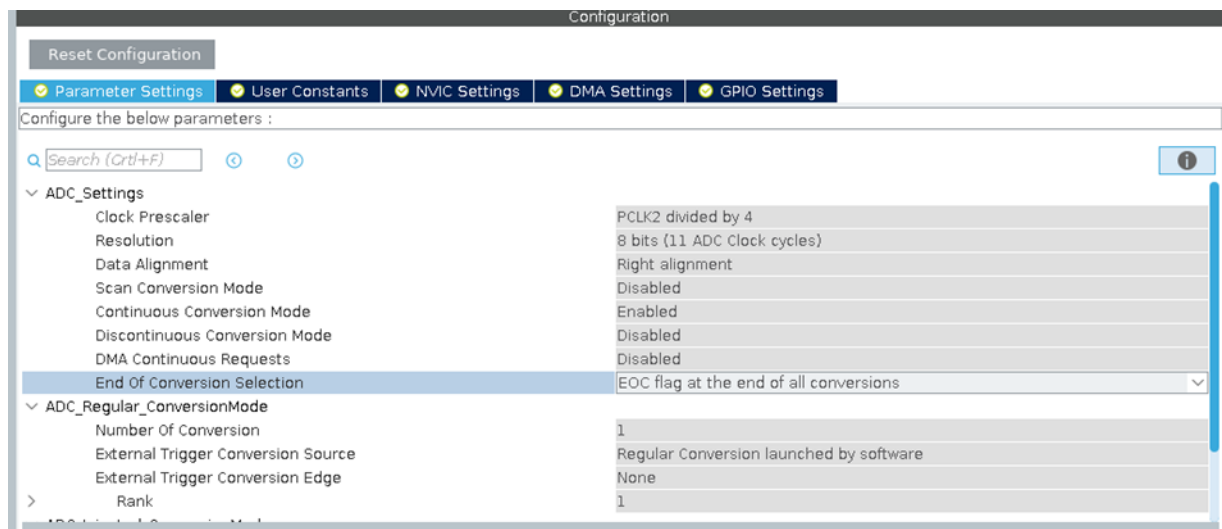
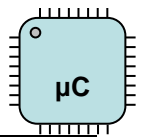


Figure 13: ADC Configuration.

ADC programming using Low-Layer (LL) paradigm

Equivalently to the GPIO case, ADC1 peripherals can be programmed by exploiting the macros at pg. 1082 of the User Manual [8] :

- `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 project 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, in particular 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.

The ADC configuration code is automatically generated by STM32Cube and available in the `MX_ADC1_Init` function.

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>
- [8] UM1725, User Manual, Description of STM32F4 HAL and LL drivers, DocID025834 Rev 5, Feb. 2017