
# Digital Systems Electronics

## Lab10-STM32

# Input Capture, Hardware Abstraction Level, Pulse Width Modulation and USART

## 1. The Hardware Abstraction Layer (HAL)

In the previous laboratories, the programming of the microcontroller has been always performed using the Low-Layer abstraction. In general, this is not the only way to program it. Three modes are available:

a. Standard Peripheral Library: it enables the programming of the microcontroller directly dealing with the registers.

b. Low-Layer (LL) paradigm: it enables the programming of the microcontroller dealing with registers but introduces macros that simplify the register writing/reading operation and some high-level macros that simplify the programming (e.g. flag clear)

c. Hardware Abstraction Layer: it is intended to enhance the abstraction level providing macros that enable the programming of the microcontroller without working at register level.

The Hardware Abstraction Layer (HAL) has the role of abstracting from the specific peripheral mapping, by separating the development of applications from the detailed knowledge of the underlying hardware. This is done by defining several handlers for each peripheral. A handler is nothing more than a C struct, whose references are used to point to real peripheral address.

For example, for the general-purpose I/O ports, GPIO_InitTypeDef is the C struct used to configure the GPIO, and it is defined in the following way

```
/**
  * @brief  GPIO Init structure definition
  */
typedef struct
{
  uint32_t GPIO_Pin;           /*!< Specifies the GPIO pins to be configured.
                                    This param. can be any value of @ref GPIO_pins_define */
  GPIOMode_TypeDef GPIO_Mode;  /*!< Specifies the operating mode for the selected pins.
                                    This param. can be a value of @ref GPIOMode_TypeDef */
  GPIOSpeed_TypeDef GPIO_Speed;/*!< Specifies the speed for the selected pins.
                                    This param.  can be a value of @ref GPIOSpeed_TypeDef */
  GPIOOType_TypeDef GPIO_OType;/*!< Specifies the operating output type for the selected pins.
```

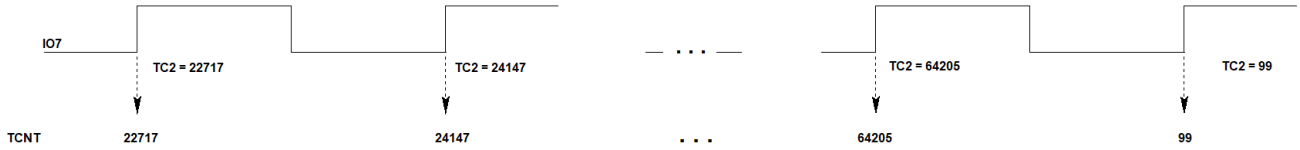
```
                                          This param.  can be a value of @ref GPIOOType_TypeDef */
  GPIOPuPd_TypeDef GPIO_PuPd;  /*!< Specifies the operating Pull-up/down for selected pins.
                                    This param.  can be a value of @ref GPIOPuPd_TypeDef */
}GPIO_InitTypeDef;
```

Notice that the struct associated to each peripheral can be generated by Cube and modified by hand.

# Signal generation setup


When you connect the signal generator to the NUCLEO board you have to verify that **the signal is compatible with the supply voltage range of the board**. In this case, the MCU is supplied with +3.3 V. **Verify with the oscilloscope** that the output signal is a square wave ranging 0-3.3 V.

## 1. Generic Input Capture in Interrupt




```
volatile unsigned short numOverflows, oldEdge, edge;

@ interrupt void tc2_isr(void){

     if OVFFlag(TC2){
          ClearOVFFlag(TC2);
          numOverflows++;
     }

     if EdgeFlag(TC2){
          ClearEdgeFlag(TC2);
          oldEdge = edge;
          edge = TC2;
     }

}
```

**TWO CASES TO BE CONSIDERED:**

**1) NO TCNT OVERFLOW**

**DELTA = edge - oldEdge = 24147 - 22717 = 1430**

**2) TCNT OVERFLOW**

**DELTA = numOverflows*65536 + edge - oldEdge = 65536 + 99 - 64205 = 1430**

**WHEN PRESCALER IS SET SUCH THAT AT MOST ONE OVERFLOW TAKES PLACE (AS IN THIS EXAMPLE), CODE CAN BE SIMPLIFIED. IN FACT, numOverflows VARIABLE DOES NOT HAVE TO BE MANAGED AND**

**DELTA = 65536 + edge - oldEdge = 65536 + 99 - 64205 = 1430**

An Input Capture procedure for measuring the period of a square wave on an I/O pin IO7 of a generic microcontroller is reported in Figure 1. The access to the Interrupt Service Routine (ISR) takes place on the rising edge of the signal and the value of the free-running counter TCNT in correspondence of these edges is stored in TC2. The ISR presents a possible procedure for counting TCNT overflows.

It is assumed that the parallelism of all registers and variables is 16 bit.

## 2. The Input Capture function of TIMx

Let us assume that the TIM3 timer is configured to work at a given TIM3_CLK clock frequency, which is obtained by dividing the APB timer clock frequency by (1+PSC), where PSC is the value programmed in the TIM3_PSC register. The timer increments the TIM1_CNT register up to the Period value (set in the TIM3_ARR register) every 1/TIM3_CLK seconds.

The Input Capture is a feature of the TIMx module which enables us to save the exact time an event occurs on a specified input channel (e.g. the rising edge of a square wave signal). In Input Capture mode, if we apply a square waveform to the input channel 1 and configure the unit to detect every rising edge of the input signal, we have that the TIM3_CCR1 register will be updated with the content of the TIM3_CNT register at every detected transition. When this happens, the timer will generate a corresponding interrupt or a DMA request, allowing keeping track of the counter value.

The Input Capture mode enables us to measure the period of a square wave.

To get the external signal period, we need to perform two consecutive captures and to calculate their distance in terms of counter content. If the first and second captures are equal to CNT0 and CNT1 respectively, their relative distance is simply given by CNT1-CNT0, if CNT1 > CNT0. If, on the contrary, CNT1 < CNT0, the distance is obtained as CNT1 – CNT0 + TIM3_Period. The Auto-Reload Register value, differently from the Output Compare, is not critical for measuring an input frequency or a duty cycle. In fact, since in Input Capture mode the value stored in each Capture-Compare Register TIMx_CCRy depends only on the value of TIMx_CNT when a certain edge occurs (*i.e.* it is not modified by any other phenomenon or instruction), it would be always lower than TIMx_ARR. The creation and the monitoring of a variable that counts the number of overflows permit to obtain the right results in any case.

Once we have the distance, DIST, the period of the received waveform is calculated as DIST / TIM3_CLK. If a prescaler is applied to the used channel, we have to consider it and multiply the previous expression by the prescaling factor. The accuracy of the measure depends on the TIM3_CLK frequency: the higher the clock frequency of the timer, the better the accuracy of the measure. However, be aware that the UEV frequency (an entire cycle of the counter 0 -> TIMx_ARR) must be lower than the frequency of the sampled signal: if the timer runs too fast, it will overflow (that is, it runs out the Period counter) before it can sample the signal edges. We can compensate for a single overflow by comparing the values of the first and second captures and adding TIM3_Period when the second capture is lower than the first one, as shown above. However, if the distance between the two captures is larger than TIM3_Period, we have a wrong result. To avoid this condition, we can increase the Period or the channel prescaler factor.

## 3.1. Input capture configuration using STM32Cube

To bound one channel to the corresponding input you have to select the Input capture direct mode for the desired channel from the Pinout view, for example Channel 1 for TIM3 (Figure 1).

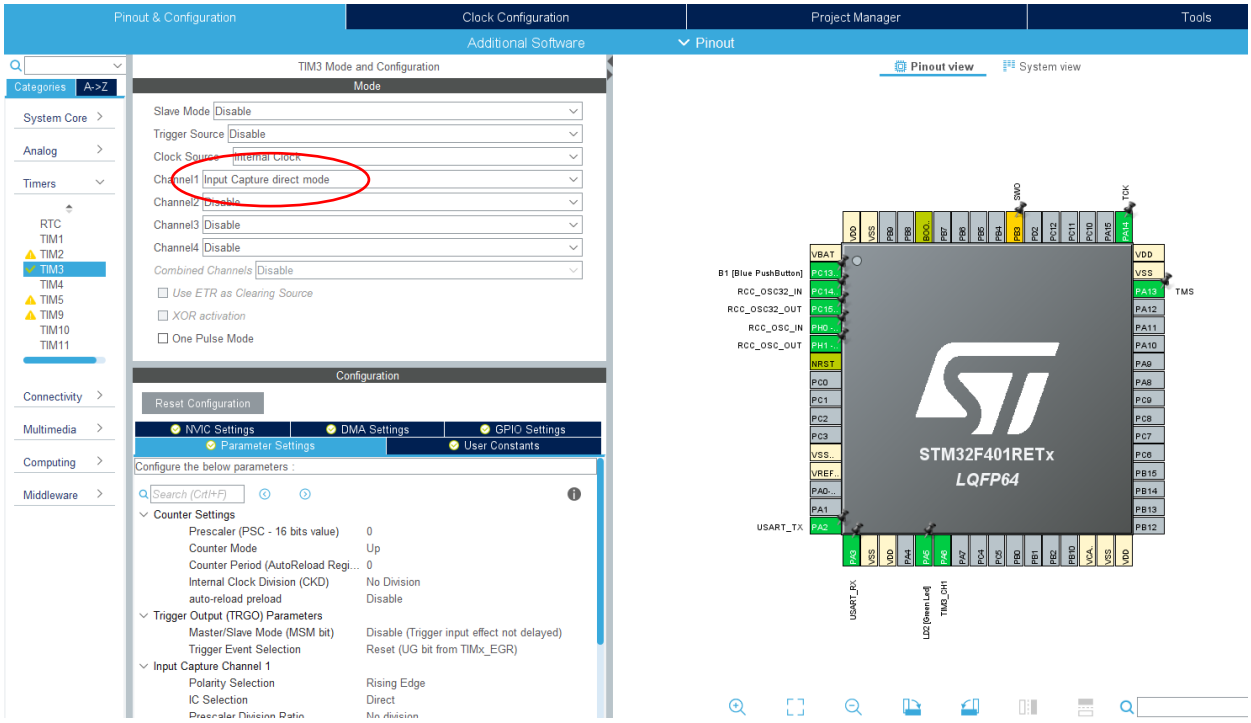



*Figure 1: Input Capture on Channel 1 of TIM3*

From the TIMx configuration view, we can configure the other input capture parameters (see Figure 2). Leave the default parameters, except for:

- The Prescaler value. For example, given an APB clock frequency of 84 MHz, we obtain with PSC = 1999 a counter increment frequency of 42 kHz. This choice leaves 420 increment periods between two consecutive rising edges of a 100 Hz input squarewave.
- In the example TIMx_ARR is set to 65535, in order to minimize the number of overflows.
- The Polarity Selection. Here you can select the edge (rising, falling or both) to be monitored according to the application.

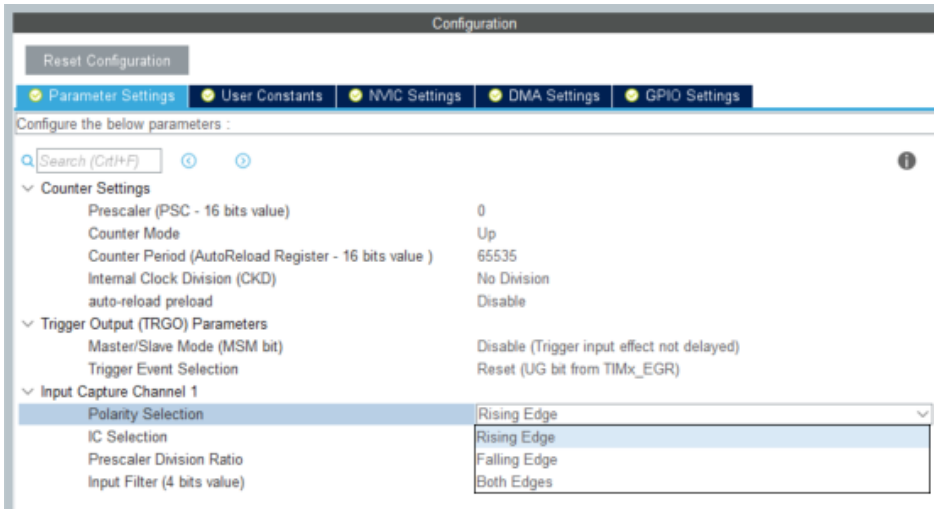



*Figure 2: TIMx Configuration for Input Capture.*

In the NVIC Settings window, we have to enable the TIMx capture compare interrupt (Figure 3), in order to have an interrupt when the desired edge occurs.

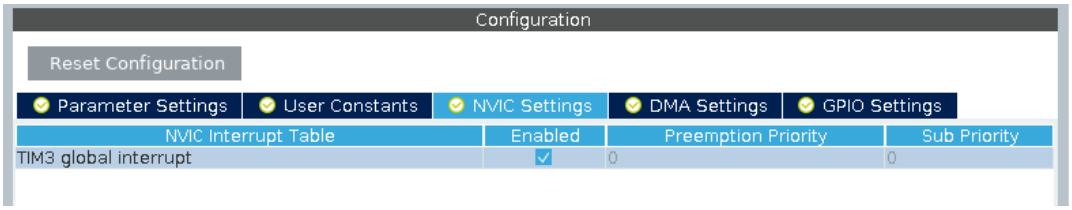
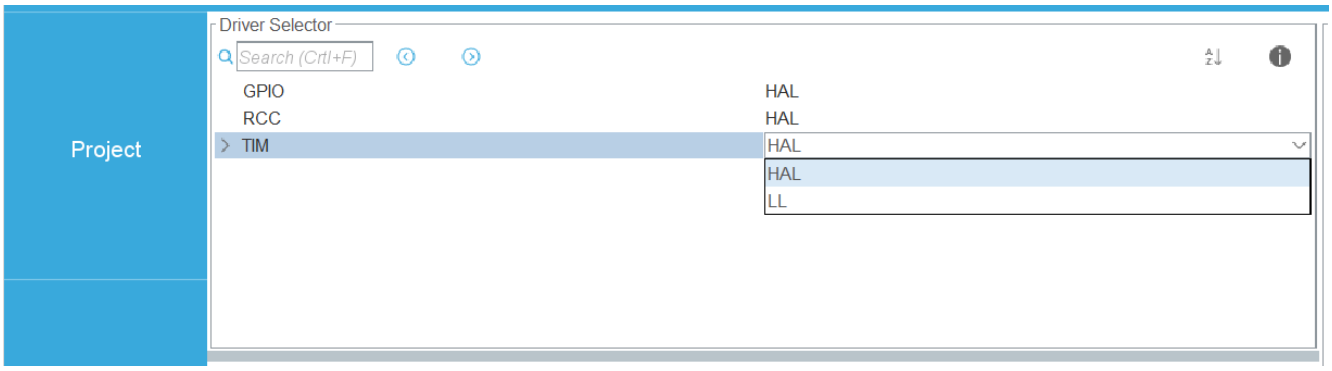



*Figure 3: TIMx global interrupt enable*

After configuring the TIMx peripheral, from the Project menu, select *Settings* and enter project name, project location, toolchain folder location and toolchain (SW4STM32). Then, in the *Advanced Settings* window, select HAL for each peripheral and verify that the "Not Generate Function Call" option is not set for the MX_TIMx_Init function. Finally, generate the configuration code.
In order to use the HAL paradigm, we have to change the configuration of the STM32Cube so that it generates HAL configuration code. To do this, we select "HAL" in the "Project Manager" tab, "Advanced Settings" section.




Once the code is generated with STM32Cube, some automatic code which is similar to the one obtained using the LL configuration shows up in the main.c file. It is necessary to recognize the function HAL_Init(), equivalent to LL_Init(), it resets all peripherals and initializes the flash interface and the Systick.

Spend some time to read and understand the configuration functions MX_TIM3_Init(). In the latter function, all the parameters that you have inserted in the STM32Cube show up.

Running the code as it is does not produce any result. With the LL paradigm, we should start the counter, enable the interrupts, and set the TIMx operating mode (OC or IC).

# 4. Interrupt using the HAL paradigm

Managing interrupts using the HAL is slightly different with respect to LL. The ISR is already generated by the STM32Cube and the interrupt flag is automatically cleared. Differently from LL, you don't have to modify the handler located in the stm32f4xx_it.c, but you can exploit different callbacks for the different peripherals and modes. By looking at the stm32f4xx_it.c file, you can see that the ISR of, as an example, TIM3 already calls a function named `HAL_TIM_IRQHandler.` If you open the definition of this function (handler), you will see a series of IF/THEN/ELSE that call a specific callback depending on the actual interrupt generated: these functions are where you have to write your code. To simplify the programmer's life, the callbacks are defined using the *weak* keyword. This means that you can redefine the needed callbacks in your main function without any error.

Imagine you need to perform some tasks when a counter reaches the ARR value and generates the correspondent interrupt. You can define in your main file the following function:

```
void HAL_TIM_PeriodElapsedCallback (TIM_HandleTypeDef *htim) {
    // Operations to be performed each time the Timer counter reaches the value of the ARR register
}
```

where the htim is a pointer to a TIM_HandleTypeDef structure that contains the configuration information for TIM module. If you have more than one timer running htim is used to determine which timer has generated the interrupt. You can find all the callbacks their usage in the User Manual [3].

## 4.1. Input Capture using HAL paradigm

TIM peripherals can be programmed by exploiting the macros at pg. 887-899 of the HAL User Manual, specified also in the pseudocode reported in LL_vs_HAL.pdf file on Portale della Didattica:

```
int main()
{
      /* after configuration done by Cube and before while(1) you must insert */
      HAL_TIM_IC_Start_IT(&htimx, TIM_CHANNEL_y);
      /* with this command it is possible to start the input compare mode in
      interrupt on channel y of timer x (it merges the three commands in LL mode)
      */
      while(1)
      {
      //code
      }
      return 0;
}
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)
{
      if(htim == &htimx){
        capture= HAL_TIM_ReadCapturedValue(htim,TIM_CHANNEL_y); /*all flag clears
        are done by HAL_TIM_IRQHandler(&htimx) in void TIMx_IRQHandler(void) in
        stm32f4xx_it.c*/
      }
}
```

Where **x** is the channel number.

- `HAL_TIM_IC_Start_IT (TIM_HandleTypeDef *htim, uint32_t Channel)`, Starts the TIM Input Capture measurement in interrupt mode. Two parameters are required:
  a. htim: pointer to a TIM_HandleTypeDef structure that contains the configuration information for TIM module.
  b. Channel: TIM Channels to be enabled. This parameter can be one of the following values:
     – TIM_CHANNEL_1: TIM Channel 1 selected
     – TIM_CHANNEL_2: TIM Channel 2 selected
     – TIM_CHANNEL_3: TIM Channel 3 selected
     – TIM_CHANNEL_4: TIM Channel 4 selected

  The function returns the status.
- `HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)`, Input Capture callback in non blocking mode:
  a. htim: pointer to a TIM_HandleTypeDef structure that contains the configuration information for TIM module.

  The function returns void.
- `HAL_TIM_ReadCapturedValue(TIM_HandleTypeDef *htim, uint32_t Channel)` Read the captured value from Capture Compare unit.
  a. htim: pointer to a TIM_HandleTypeDef structure that contains the configuration information for TIM module.
  b. Channel: TIM Channels to be enabled. This parameter can be one of the following values:
     – TIM_CHANNEL_1: TIM Channel 1 selected
     – TIM_CHANNEL_2: TIM Channel 2 selected
     – TIM_CHANNEL_3: TIM Channel 3 selected
     – TIM_CHANNEL_4: TIM Channel 4 selected

The function returns the captured value.

Using the MCU peripherals in HAL is easier than LL. Essentially, HAL functions set for you all the registers required to use the peripheral. The steps are:

- Enable TIM peripheral in Input Capture mode with interrupt (`HAL_TIM_IC_Start_IT`). From an LL point of view, this function automatically sets:
  a. TIM first Control Register (TIMx_CRy), to enable the counter by setting the CEN bit.
  b. TIMx DMA/Interrupt Enable Register (TIMx_DIER), whose CCxE bit enables the interrupt for capture compare mode on channel x.
  c. TIMx Capture/Compare Enable Register (TIMx_CCER), whose CCxE bit enables the capture compare mode on channel x.
- Description of the Input Capture ISR (`HAL_TIM_IC_CaptureCallback`). In the example, in this routine the TIMx capture/compare register (TIMx_CCRy for channel y) is read by means of `HAL_TIM_ReadCapturedValue`. In this register, there will be the value of the timer TIMx_CNT when an edge to be monitored occurs.

It is remembered that Input Capture configuration is automatically done by STM32Cube.

Pay attention that when you must monitor both rising and falling edges, the program enters into the Input Capture ISR. Since the pin to which the external signal is connected belongs to one of the I/O peripherals, we can understand the type of edge by reading the value of its corresponding bit in the GPIOx_IDR register. This procedure is conceptually similar to that of the VHDL statement

```
if clock'event and clock='1'
```

for a rising edge.

## 4.2. GPIO with the HAL programming paradigm

The configuration of GPIO with STM32CubeMX is the same described in the previous laboratory sessions. Before describing the functions to be exploited in your code, pay attention that HAL programming presents the `GPIO_PinState` data type for the value of a bit belonging to a GPIO register/port. It can assume two values:

- *`GPIO_PIN_RESET`*, equal to 0;
- *`GPIO_PIN_SET`*, equal to 1.

The functions for reading or writing a GPIO bit are the following:

- `HAL_GPIO_ReadPin (GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin);`
  - `GPIOx`: GPIO port of interest, where x can be between A and I.
  - `GPIO_Pin`, that specifies the port bit to read. This parameter can be one of GPIO_PIN_x where x can be between 0 and 15.

The function returns a `GPIO_PinState` value, that is nothing but the input port pin value.

- `HAL_GPIO_WritePin (GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin, GPIO_PinState PinState);`
  - `GPIOx`: GPIO port of interest, where x can be between A and I.
  - `GPIO_Pin`, that specifies the port bit to written. This parameter can be one of GPIO_PIN_x where x can be between 0 and 15.

- `PinState`, specifying the value to be written to the selected bit.

It is very interesting to observe that – differently from Low-Layer programming – HAL functions permit to read a single bit, so masking the register belonging to a GPIO port is not necessary anymore. Furthermore, on the manual you can find also a function to toggle one pin.

## 4.3. Output Compare with the HAL programming paradigm

For sake of completeness, we report here the configuration of Output Compare with STM32CubeMX using HAL.
Output Compare and Input Capture are exploited in *Non-blocking mode*, that is the HAL equivalent of Interrupt mode.

The function to be exploited in your code for enabling Output Compare is:

- `HAL_TIM_OC_Start_IT (TIM_HandleTypeDef * htim, uint32_t Channel);`
  - htim, that is the pointer to a structure containing the configuration information for TIM module to be exploited.
  - Channel, the TIM Channel to be enabled. You must assign to this parameter the value TIM_CHANNEL_x, where x is between 1 and 4.

The following user callback (the interrupt service routines of HAL programming) will be executed when the Output Compare events take place.

```
void HAL_TIM_OC_DelayElapsedCallback (TIM_HandleTypeDef *htim)
{
      // insert your output compare code
}
```

# 5. The Pulse Width Modulation module

PWM is a technique used to generate several pulses with different duty cycles in a given period. Figure  shows three waveforms with the same frequency and three different duty cycles, 50%, 20% and 80%. By varying the duty cycle of the waveform, it is possible to regulate the average voltage proportionally. Therefore, it is possible to control the amount of energy transferred to a given load, such as for example a motor. By using external amplifiers and power transistors, we can also increase the voltage range and control high currents. In another case, we can control the average voltage applied to a LED, in order to obtain a given amount of light.

To well understand the functioning of the PWM, take as reference page 263 of the reference manual. The PWM module is associated to the timer peripheral. The value of the signal frequency is determined by the auto-reload register TIMx_ARR register whereas the duty cycle is determined by the Capture Compare Register TIMx_CCRx register (Pulse).

Two PWM modes are available and can be selected by properly managing the register TIMx_CCMRx. In both PWM modes, TIMx_CNT and TIMx_CCRx are compared to determine whether TIMx_CNT ≤ TIMx_CCRx (or the contrary, according to the counter direction).

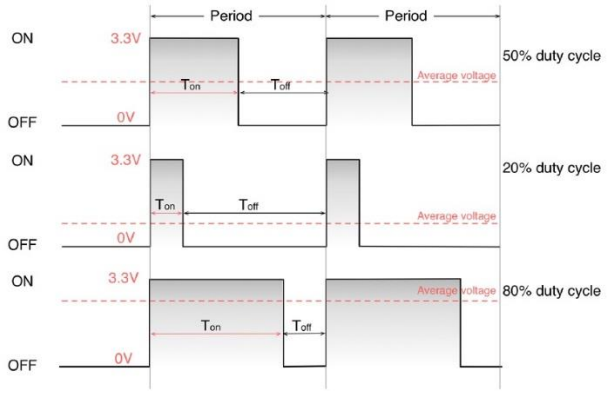



*Figure 4: Examples of PWM*

## 5.1. PWM configuration using STM32Cube

We use STM32Cube to obtain the configuration statements for and TIMx peripheral. We start by launching the STM32CubeIDE tool and creating a new project, with all the peripherals configured by the STM32Cube with their default mode. To configure the TIMx timer, click on corresponding option in the Peripherals column and select *Internal Clock* as the *Clock Source*. Then enable the PWM on Channel1 of TIMx, the corresponding pin will be green in the Pinout View (Figure 4).

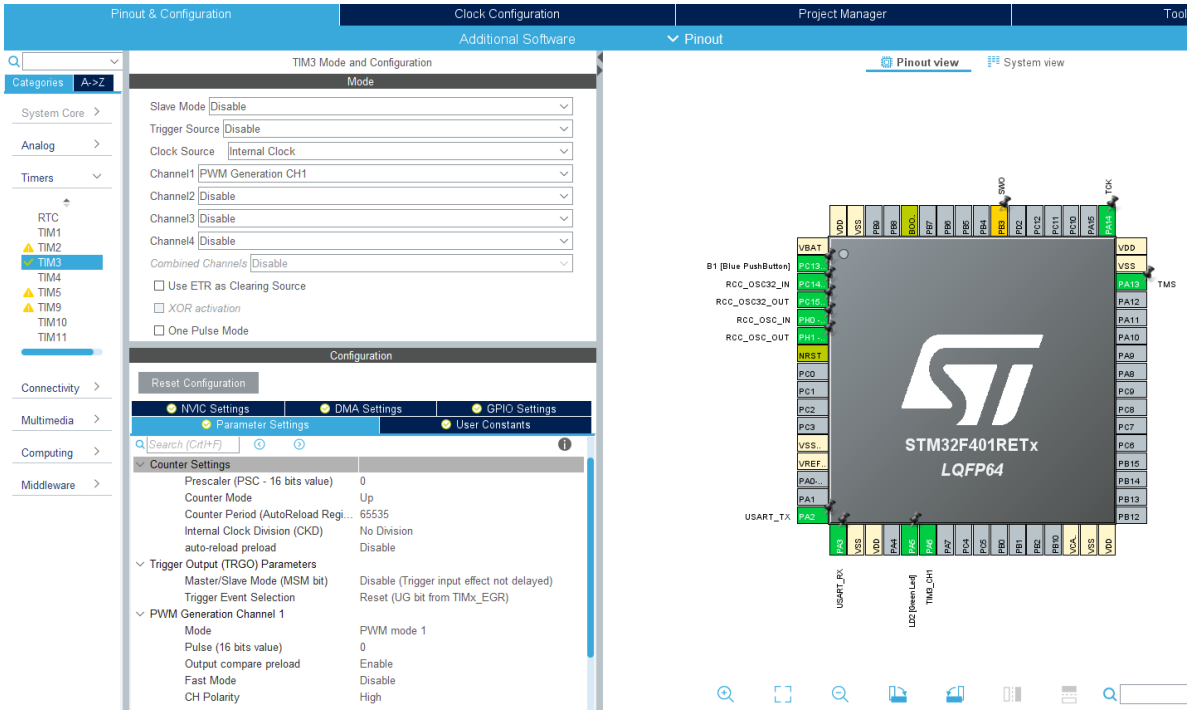



*Figure 4: PWM on Channel 1 of TIM3*

At this point, you must Configure TIMx: in the Parameter Settings you have to insert the values of Prescaler, Counter Period and Pulse to accomplish laboratory specification, while the other parameters must be kept to the default values (Figure 5). For PWM, the period of the output signal is configured through the value stored in the Auto-Reload Register (Counter Period), while the duty cycle is related to the value stored in the Capture Compare Register (Pulse) associated to Channel y. In particular, this value stores the number of counter cycles in which the output voltage is high.

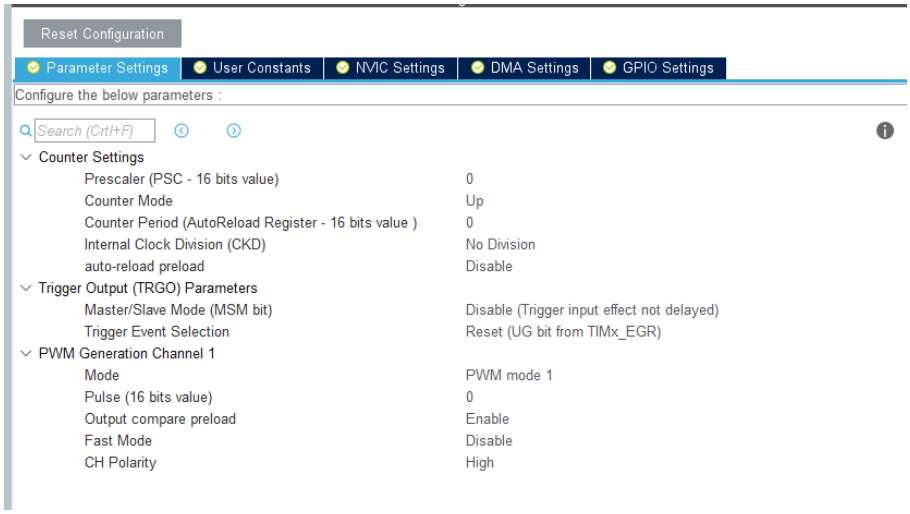



*Figure 5: TIMx configuration for PWM*

Finally, generate the code in HAL.

# 6. PWM using the HAL paradigm

With the HAL, the PWM interface must be initialize using the function HAL_TIM_PWM_Start:

- HAL_TIM_PWM_Start(__INSTANCE__, __CHANNEL__): for starting the PWM.
  Two parameters are required:
  a. __INSTANCE__: TIM Instance. Differently from LL here it is a reference to the *init structure* (&htim2 if you are using TIM2)
  b. __CHANNEL__: timer channel connected to the PWM as TIM_CHANNEL_x
  If we want to start TIM3_CH4 in PWM we write HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_4);

After this instruction, the counter starts and the PWM is enabled. The PWM duty cycle can be modified at runtime using the macro __HAL_TIM_SET_COMPARE.

d. __HAL_TIM_SET_COMPARE(__INSTANCE__, __CHANNEL__, __VALUE__), for writing a value in the CCxR register. Three parameters are required:
   a. ___ INSTANCE__: TIM Instance
   b. ___CHANNEL__: timer channel connected to the PWM as TIM_CHANNEL_x
   c. ___VALUE__: value (uint16_t) to be set as duty cycle

# 7. LED dimmer

In order to change the led brightness in one second it suggested that you configure the timer in the following way:

1. Set the prescaler in order to get a counter increment rate of hundreds of kHz.
2. Set the *Period* to an easy number in order to get a frequency of the PWM of hundreds of Hz.
3. The duty will vary from 0 to *Period* (0% → 100% duty cycle) with unit increments.
4. Divide 1 second (the requested time for a complete sweep) by the number of needed increments: this is the time that you need to wait among two successive modification of the PWM duty cycle.

Be sure that in the STM32Cube the LED pin is set as TIM2_CH1 output and not standard GPIO.

Notice that the generation of delays used for varying the PWM duty cycle may be generated in several ways. The easiest way is to exploit the following HAL function:

HAL_Delay(< uint32_t time[milliseconds]>);

Of course, delays block the execution of the code, freezing the processor. No other instructions will be executed in the meantime. An alternative consists in using the OC function of a timer. It is necessary to enable the interrupt using STM32Cube in the timer NVIC configuration panel, as we did for the Input Capture mode, see Figure 3. Then, the timer can be started in output compare interrupt mode using:

- HAL_TIM_OC_Start_IT(__INSTANCE__, __CHANNEL__): for starting the OC in interrupt mode.
  Two parameters are required:
  a. __INSTANCE__: TIM Instance.
  b. __CHANNEL__: timer channel connected to the OC as TIM_CHANNEL_x

Each time the output compare generates an interrupt, it will call the function HAL_TIM_OC_DelayElapsedCallback. This function can be rewritten in the main.c file to manage the interrupt of the Output Compare according to our need. Define the callback and write the needed code.

```
void HAL_TIM_OC_DelayElapsedCallback (TIM_HandleTypeDef *htim) {
    // Operations to be performed each time the OC is called.
}
```

The following functions can be useful to set or red the value of the Pulse OC register:

- HAL_TIM_ReadCapturedValue (__INSTANCE__, __CHANNEL__): for reading the value of the Pulse register. Three parameters are required:
  a. __INSTANCE__: TIM Instance.
  b. __CHANNEL__: timer channel connected to the OC as TIM_CHANNEL_x

- __HAL_TIM_SET_COMPARE(__INSTANCE__, __CHANNEL__,__VALUE__): for setting the value of the Pulse register. Three parameters are required:
  a. __INSTANCE__: TIM Instance.
  b. __CHANNEL__: timer channel connected to the OC as TIM_CHANNEL_x
  c. __VALUE__: value of the Pulse register

As a third alternative, you can use the interrupt generated by the timer when the counter reaches the value of the ARR register. In this case, we do not need the OC function. The timer can be started with the instruction:

- HAL_TIM _Start_IT(__INSTANCE__): for starting the Timer in interrupt mode.
  One parameter is required:
  a. __INSTANCE__: TIM Instance.

Then, the interrupt can be managed properly defining the already mentionedand described callback HAL_TIM_PeriodElapsedCallback. All the TIM callbacks can be found at page 877 of [3].

# 8. The Universal Synchronous Asynchronous Receiver Transmitter (USART)

The Universal Synchronous Asynchronous Receiver Transmitter (USART) unit of the microprocessor provides the capability of receiving and transmitting data to other devices.

The USART unit provides different features for the implementation of ad-hoc transmission among devices:

- Full duplex asynchronous communication
- Fractional baud rate generation
- Programmable data word length
- Configurable stop bits
- Single-wire communication
- Enable bits
- Transfer detection flags
- Error detection
- Parity control
- Interrupt

A detailed description of the USART is available from page 501 of the reference manual. For a standard serial communication, a minimum of two pins is required (one for RX, one for TX). The RX channel is used to receive data (from device to NUCLEO) whereas the TX channel is used to transmit data (from NUCLEO to device). Oversampling techniques are used to discriminate between data and noise. When the transmission is not enabled, the TX channel is put at high level.

The USART might be programmed using the Hardware Abstraction Level. The most important registers are:

e. Status register
f. Data register
g. Baud rate register

For the details about the registers refer to page 542 of the reference manual.

## 8.1. USART configuration using STM32Cube

We use STM32Cube to obtain the configuration statements for the USART module. We start by launching the STM32Cube tool and creating a new project, with all the peripherals configured with their default Mode. **The default USART configuration is sufficient for the completion of the proposed exercises.** By enabling the default Mode, the USART2 is typically already configured on PA2 and PA3 pins, TX and RX channel respectively. The USART2 is connected to the STLINK chip present on the board (the small chip in the portion of the board with the USB connection). This enables the communication between the board and the personal computer through the same USB that you use to power and program the NUCLEO board. Alternatively, USART1 can be used to implement a communication with a third device, Figure 7 shows the standard configuration of the USART2 module. Check that in the default configuration *Mode* is set as *Asynchronous* and the *Hardware Flow Control (RS232)* is *Disable*.

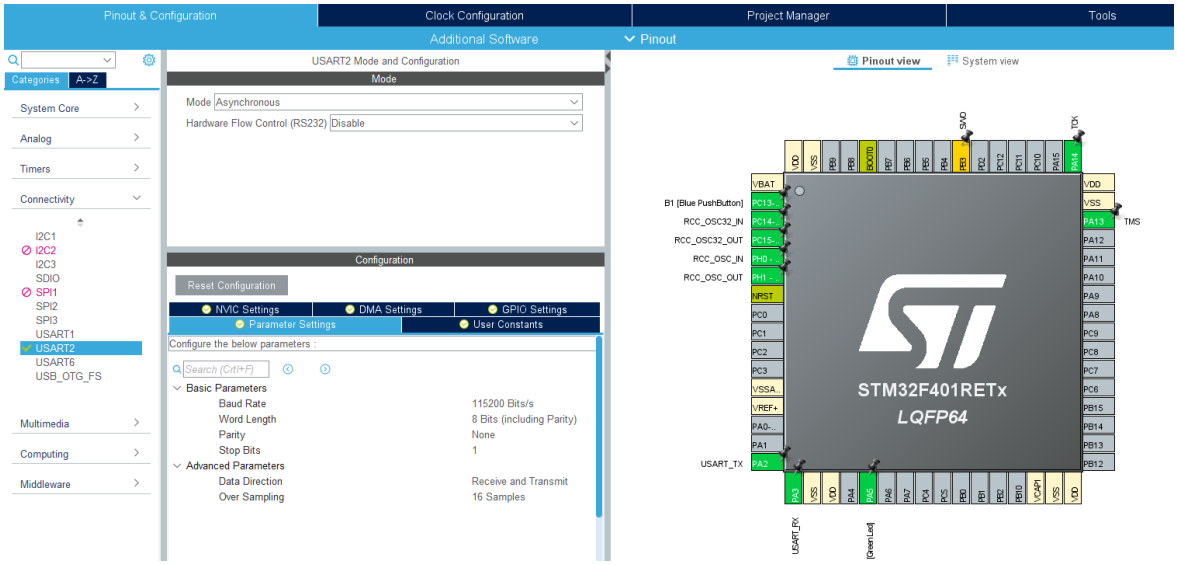



*Figure 7: Pinout with USART2 enabled.*

In the Configuration View, open the USART2 panel and check the Parameter Settings. All the parameters of the transmission should be inserted as in Figure 8. In *NVIC Settings* enable the USART2 global interrupt. The code generate by STM32Cube automatically configures the serial interface. You must only insert the specific code for your application.

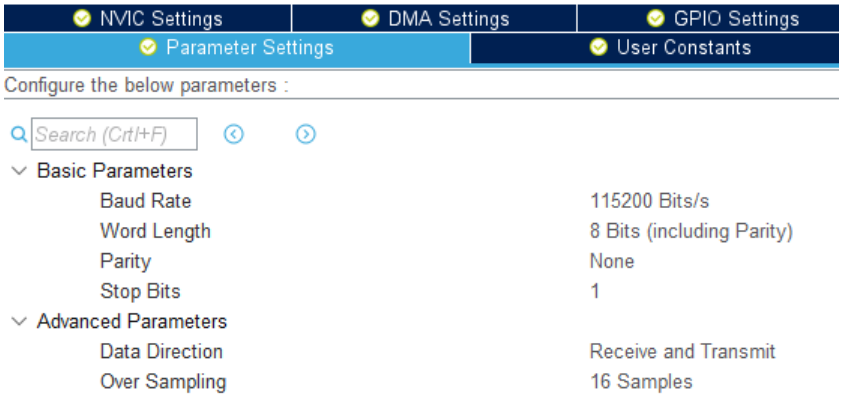



*Figure 8: configuration of the USART2.*

## 8.2. USART with the HAL programming paradigm

Communication can be performed in two modes:

1. *Blocking mode*. The communication is managed by exploiting Polling. The status of the transmission is returned by the same function after finishing the data transfer. The main program is blocked until the communication is ended.

2. *Non-blocking mode*. The communication is performed using Interrupts or DMA, these APIs return the HAL status. The end of the data processing will be indicated through the dedicated UART IRQ when using Interrupt mode or the DMA IRQ when using DMA mode.

In this laboratory, non-blocking communication is required. It is automatically configured by STM32Cube.

The Hardware Abstraction Layer provides some macros useful to manage the USART module and the communication. In particular, the `HAL_UART_Transmit_IT()` function allows sending a message from the board to the serial communication interface. At the end of the transmission, the operation generates an interrupt. Obviously, the interrupt must be enabled in the NVIC configuration using STM32Cube.

- `HAL_UART_Transmit_IT(__INSTANCE__,__DATA__,__SIZE__);`
    - `__INSTANCE__`: USART instance, properly pointer to a UART_HandleTypeDef structure that contains the configuration information for the specified UART module.
    - `__DATA__`: pointer to uint8_t data intended to be sent
    - `__SIZE__`: amount of data (**in bytes**) to be sent.

The opposite function is the `HAL_UART_Receive_IT()`. It configures the UART peripheral to receive a data from the serial interface. The execution of the program is continued (non-blocking interrupt mode), then the execution is blocked as soon as the data are received and an interrupt is generated.

- `HAL_UART_Receive_IT(__INSTANCE__, __DATA__,__SIZE__);`
    - `__INSTANCE__`: USART instance, properly pointer to a UART_HandleTypeDef structure that contains the configuration information for the specified UART module.
    - `__DATA__`: pointer to uint8_t data intended as RX buffer
    - `__SIZE__`: amount of data (**in bytes**) to be received.

The following user callbacks will be executed respectively at the end of the receive or transmit process.

```
void HAL_UART_RxCpltCallback (UART_HandleTypeDef *huart)
{
      // insert your code
}

void HAL_UART_TxCpltCallback (UART_HandleTypeDef *huart)
{
      // insert your code

}
```

Starting a new UART reception in the callback of the previous one could lead to problem during execution. A better approach consists in setting a flag in the callback and reading the flag in the main loop in order to trigger a new receive. Furthermore, you should minimize the code in the callbacks: it is better to perform all the requested instruction in the main loop.

For the sake of completeness, we report a commented example. Notice that this example cannot be executed as it is, use it as a mean of understanding how the UART can be managed.

```
int volatile correctlySentData = 0;
int volatile correctlyReceivedData = 0;

int main ()
{

  // Print welcome message
  HAL_UART_Transmit_IT(&huart2, "Hello world!!", sizeof("Hello world!!"));

  // Prepare UART to receive a single character
  char character[2]; // Pointer to received data
  HAL_UART_Receive_IT(&huart2, character, 1);

  while (1)
  {
         // Check TX flag
         if (correctlySentData == 1) {
             correctlySentData = 0;
             // …
         }

         // Check RX flag
         if (correctlyReceivedData == 1) {
             correctlyReceivedData = 0;
             // …
         }
  }

}

void HAL_UART_TxCpltCallback (UART_HandleTypeDef *huart)
{
       \\ Set TX flag
       correctlySentData = 1;
}

void HAL_UART_RxCpltCallback (UART_HandleTypeDef *huart)
{
       \\ Set RX flag
       correctlyReceivedData = 1;
}
```

# 9. Connecting the board to the PC

You will use a script written in Python to connect the PC to the NUCLEO board. **The script has all the functionalities to handle all the exercises**. In the top you will have to setup connection parameters and select the correct COM port. On the left you

have a log with all the messages received from the board and a line edit in the bottom where you can input your commands (for all the exercises). Even if you are not using it in this lab, on the right there is the graph that can be used to display the motor speed, see Figure 9. The script will automatically load the value and plot the trace of the motor speed, but you have to use specific text messages to send the information through the USART:

- "Speed: XX" when you need to send the speed value.
- "New target speed: XX rpm" when you update the target speed (OPTIONAL only)

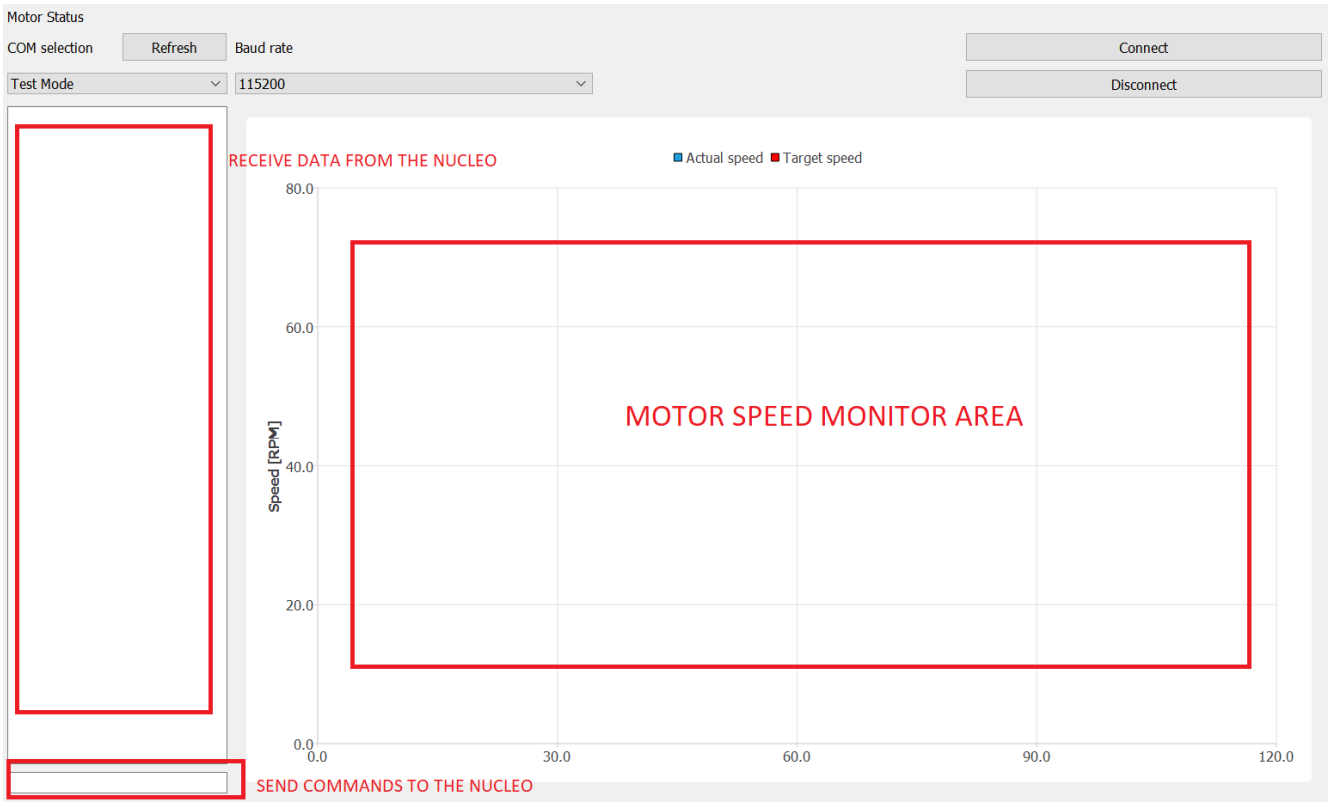



*Figure 9: Script for the transmission control.*

# 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