

# Timers and working modes

Understand about timers and working modes to configure the periodic tasks, Pulse-width modulation, and Capture mode.

`#arm #stm32 #timer #pwm #capture`

---

Last update: May 6, 2021

## Table of Content

1. Timer overview
2. Timer modes
3. HAL Software
4. Lab: Basic Timer
  - 4.1. Configure IO
  - 4.2. Enable Timer
  - 4.3. Generated code
  - 4.4. User code
5. Lab: Timer interrupt
  - 5.1. Add a timer
  - 5.2. Handle callback
  - 5.3. Start timer
6. Lab: PWM
  - 6.1. Duty
  - 6.2. Drive the brightness
  - 6.3. Setup PWM on Timer
  - 6.4. Generated code
  - 6.5. User code

## 1. Timer overview

A timer is a free-running counter with a counting frequency that is a fraction of its source clock. The counting speed can be reduced using a dedicated pre-scaler for each timer. Depending on the timer type, it can be clocked by the internal clock (which is derived from the bus where it is connected), by an external clock source or by another timer used as "master".

The  $F_{sys}$  is not the frequency that is incrementing the timer module, but it gets divided by the Pre-scaler, then it gets fed to the timer. Every clock cycle, the value of the timer is incremented by 1.

Let's see an example to calculate the timer period.

- $F_{sys} = 80 \text{ MHz}$
- Pre-scaler = 1:1024
- Timer gets incremented by 1 every  $1024 * 1/80000000 \text{ s} = 12.8 \text{ us}$
- If set overflow at full 16-bit (at 65535), and start counting from 0, it will generate a signal every  $12.8 \text{ us} * 65535 = 838848 \text{ us} = 838.848 \text{ ms}$

A timer can have additional pre-load register, therefore, timer will count from 0 to the pre-load value, or vice-versa.

When driven by an external input, such as external pulse or button, timer can be used to count the number of events happened, and measure the frequency.

STM32 timers can mainly grouped in categories:

- **Basic timers:** 16-bit timers used as time base generator; do not have output/input pins; used as a master of other timers or used to feed the DAC peripherals.
- **General purpose timers:** 16/32-bit timers with multiple purposes; have four-programmable input/output channels; used in any application for output compare (timing and delay generation), One-Pulse Mode, input capture (for external signal frequency measurement), sensor interface (encoder, hall sensor), etc. This type has sub-groups: 1-channel/2-channels, or 1-channel/2-channels with one complimentary output ( a dead time generator on one channel).
- **Advanced timers:** have more features than General purpose timers such as features related to motor control and digital power conversion applications: three complementary signals with dead time insertion, emergency shut-down input.
- **High resolution timer:** a timer allows generating digital signals with high-accuracy timings, such as PWM or phase-shifted pulses. It has Delay lines with closed loop control guarantee a very small resolution whatever the voltage, temperature or chip-to-chip manufacturing process deviation.

- **Low-power timers:** have a diversity of clock sources to run in low frequencies, or from external clock-like inputs, and have the capability to wake up the system from low-power modes.

## 2. Timer modes

An STM32 timer module can operate in any of the following modes, however, have to check the datasheet to figure out which modes are supported by which timers.

### Timer Mode

In timer mode, the timer module gets clocked from an internal clock source with a known frequency. Hence the clocking frequency is known, the overflow time can also be calculated and controlled by the preload register to get any arbitrarily chosen time interval. Each timer overflow, the timer signals the CPU with an interrupt that indicates the end of the specified time interval.

This mode of operation is usually used to get a specific operation done at each specific time interval, and to achieve timing & synchronization between various tasks and events in the system. It can also replace delays in various situations for better system response.

### Counter Mode

In counter mode, the timer module gets clocked from an external source (timer input pin). So the timer counts up or down on each rising or falling edge of the external input. This mode is really helpful in numerous situations when need to implement a digital counter without polling input pins or periodically reading a GPIO or continuously interrupt the CPU when hooking with an EXTI pin. If using another timer as an interval, this mode can be used to measure frequency.

### PWM Mode

In *Pulse-Width Modulation (PWM)* mode, the timer module is clocked from an internal clock source and produces a digital waveform on the output channel pin called the PWM signal. By using output compare registers (OCR), the incrementing timer's register value is constantly compared against this OCR register. When a match occurs the output pin state is flipped until the end of the period and the whole process is repeated.

### Advanced PWM Mode

The advanced PWM signal generation refers to the hardware ability to control more parameters and add some hardware circuitry to support extra features for the PWM signal generation. Which includes:

- The ability to produce a complementary PWM signal that is typically the same as the PWM on the main channel but logically inverted

- The ability to inject dead-time band in the PWM signal for motor driving applications to prevent shoot-through currents that result from PWM signals overlapping
- The ability to perform auto-shutdown for the PWM signal, it's also called "auto brake" which is an important feature for safety-critical applications
- The ability to phase-adjust the PWM signal

### **Output Compare Mode**

In output compare mode, a timer module controls an output waveform or indicates when a period of time has elapsed. When a match is detected between the output compare register (OCR) and the counter, the output compare function assigns the corresponding output pin to a programmable value.

### **One-Pulse Mode**

One-pulse mode (OPM) is a particular case of the previous modes. It allows the counter to be started in response to a stimulus and to generate a pulse with a programmable length after a programmable delay. Starting the counter can be controlled through the slave mode controller. Generating the waveform can be done in output compare mode or PWM mode.

### **Input Capture Mode**

In Input capture mode, the Capture/Compare Registers (TIMx\_CCRx) are used to latch the value of the counter after a transition detected by the corresponding ICx signal. When a capture occurs, the corresponding CCXIF flag (TIMx\_SR register) is set and an interrupt or a DMA request can be sent if they are enabled.

This mode is extremely important for external signal measurement or external event timing detection. The current value of the timer counts is captured when an external event occurs and an interrupt is fired.

### **Encoder Mode**

In the encoder interface mode, the timer module operates as a digital counter with two inputs. The counter is clocked by each valid transition on both input pins. The sequence of transitions of the two inputs is evaluated and generates count pulses as well as the direction signal. Depending on the sequence, the counter will count up or down.

### **Timer Gate Mode**

In timer gated mode, a timer module is also said to be working in "slave mode". Where it only counts as long as an external input pin is held high or low. This input pin is said to be the timer gate that allows the timer to count or not at all.

### **Timer DMA Burst Mode**

The STM32 timers, not all of them, have the capability to generate multiple DMA requests upon a single event. The main purpose is to be able to re-program part of the timer multiple times

without software overhead, but it can also be used to read several registers in a row, at regular intervals.

### Infrared Mode

An infrared interface (IRTIM) for remote control can be used with an infrared LED to perform remote control functions. It uses internal connections with TIM15 and TIM16 as shown in the diagram down below. To generate the infrared remote control signals, the IR interface must be enabled and TIM15 channel 1 (TIM15\_OC1) and TIM16 channel 1 (TIM16\_OC1) must be properly configured to generate correct waveforms. The infrared receiver can be implemented easily through a basic input capture mode.

## 3. HAL Software

The *Hardware Abstract Layer (HAL)* is designed so that it abstracts from the specific peripheral memory mapping. But, it also provides a general and more user-friendly way to configure the peripheral, without forcing the programmers to now how to configure its registers in detail.

*excerpt from [Description of STM32F0 HAL and low-layer drivers](#)*

### How to use TIM HAL

1. Initialize the TIM low level resources by implementing the following functions depending from feature used :
  - Time Base : `HAL_TIM_Base_MspInit()`
  - Input Capture : `HAL_TIM_IC_MspInit()`
  - Output Compare : `HAL_TIM_OC_MspInit()`
  - PWM generation : `HAL_TIM_PWM_MspInit()`
  - One-pulse mode output : `HAL_TIM_OnePulse_MspInit()`
  - Encoder mode output : `HAL_TIM_Encoder_MspInit()`
2. Initialize the TIM low level resources :
  - Use `__HAL_RCC_TIMx_CLK_ENABLE()` to enable the TIM interface clock
  - TIM pins configuration
    - Use `__HAL_RCC_GPIOx_CLK_ENABLE()` to enable the clock for the TIM GPIOs
    - Configure these TIM pins in Alternate function mode using `HAL_GPIO_Init()`
3. The external Clock can be configured, if needed (the default clock is the internal clock from the APBx), using the following function: `HAL_TIM_ConfigClockSource` , the clock configuration should be done before any start function.
4. Configure the TIM in the desired functioning mode using one of the Initialization function of this driver:

- `HAL_TIM_Base_Init` : to use the Timer to generate a simple time base
- `HAL_TIM_OC_Init` and `HAL_TIM_OC_ConfigChannel` : to use the Timer to generate an Output Compare signal.
- `HAL_TIM_PWM_Init` and `HAL_TIM_PWM_ConfigChannel` : to use the Timer to generate a PWM signal.
- `HAL_TIM_IC_Init` and `HAL_TIM_IC_ConfigChannel` : to use the Timer to measure an external signal.
- `HAL_TIM_OnePulse_Init` and `HAL_TIM_OnePulse_ConfigChannel` : to use the Timer in One Pulse Mode.
- `HAL_TIM_Encoder_Init` : to use the Timer Encoder Interface.

5. Activate the TIM peripheral using one of the start functions depending from the feature used:

- Time Base : `HAL_TIM_Base_Start()` , `HAL_TIM_Base_Start_DMA()` ,  
`HAL_TIM_Base_Start_IT()`
- Input Capture : `HAL_TIM_IC_Start()` , `HAL_TIM_IC_Start_DMA()` ,  
`HAL_TIM_IC_Start_IT()`
- Output Compare : `HAL_TIM_OC_Start()` , `HAL_TIM_OC_Start_DMA()` ,  
`HAL_TIM_OC_Start_IT()`
- PWM generation : `HAL_TIM_PWM_Start()` , `HAL_TIM_PWM_Start_DMA()` ,  
`HAL_TIM_PWM_Start_IT()`
- One-pulse mode output : `HAL_TIM_OnePulse_Start()` , `HAL_TIM_OnePulse_Start_IT()`
- Encoder mode output : `HAL_TIM_Encoder_Start()` , `HAL_TIM_Encoder_Start_DMA()` ,  
`HAL_TIM_Encoder_Start_IT()` .

6. The DMA Burst is managed with the two following functions:

`HAL_TIM_DMABurst_WriteStart()` and `HAL_TIM_DMABurst_ReadStart()`

## 4. Lab: Basic Timer

This project will use a basic timer (TIM16 or TIM17) to measure a pulse width on the Echo pin of the Ultrasonic distance sensor US-100.

Triggering the sensor to start operation is done by sending a short pulse to the TRIGGER pin and it should be anything wider than 5µs. It can be even a few milliseconds. The module sends an ultrasonic signal, eight pulses of 40kHz square wave from the transmitter; the echo is then picked up by the receiver and outputs a waveform with a time period proportional to the distance.

The echo response pulse corresponds to the time it takes for the ultrasonic sound to travel from the sensor to the object and back. Hence, the distance is computed as:

$$\text{Distance} = \text{Pulse Width} * \text{Speed of Sound} / 2 \text{ (m)}$$

The actual speed of sound depends on the several environment factors, with temperature having most pronounced effect:

$$\text{Speed of Sound} = 331.4 + 0.6T \text{ (m/s)}$$

US-100 has built-in temperature compensation, so the distance formula is reduced to:

$$\text{Distance} = \text{Pulse width} * 165.7 \text{ (m)}$$

The pulse width is calculated by the timer counter divided by the counting frequency:

$$\text{Pulse width} = \text{Timer counter} / \text{Frequency}$$

Therefore, the final equation is:

$$\begin{aligned} \text{Distance} &= \text{Timer counter} / \text{Frequency} * 165.7 \text{ (m)} \\ &= \text{Timer counter} * 165700 / \text{Frequency} \text{ (mm)} \end{aligned}$$

## 4.1. Configure IO

Start with a new project and enable UART for printing debug. Consider to use [Redirection](#) to use `printf()`.

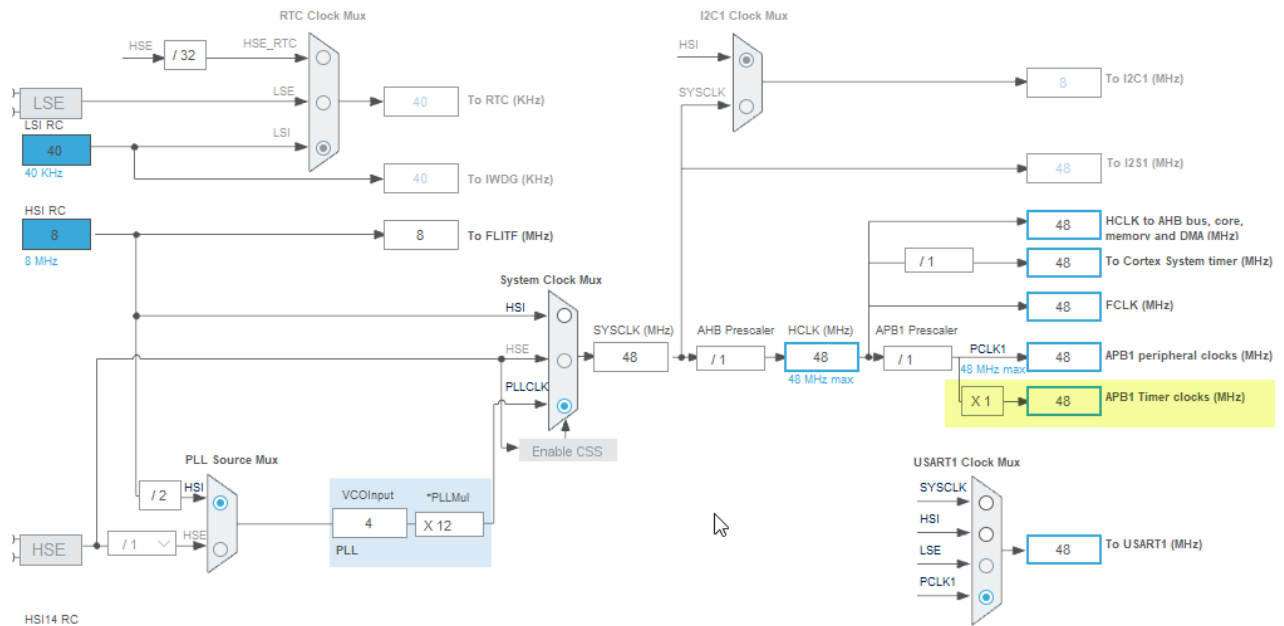
To connect with US-100 in trigger mode, use 2 GPIO pins as below:

MCU Pin	US-100 Pin
PA1 (Output)	Trigger/TX
PA2 (Input, EXTI2, Both edges)	Echo/RX

## 4.2. Enable Timer

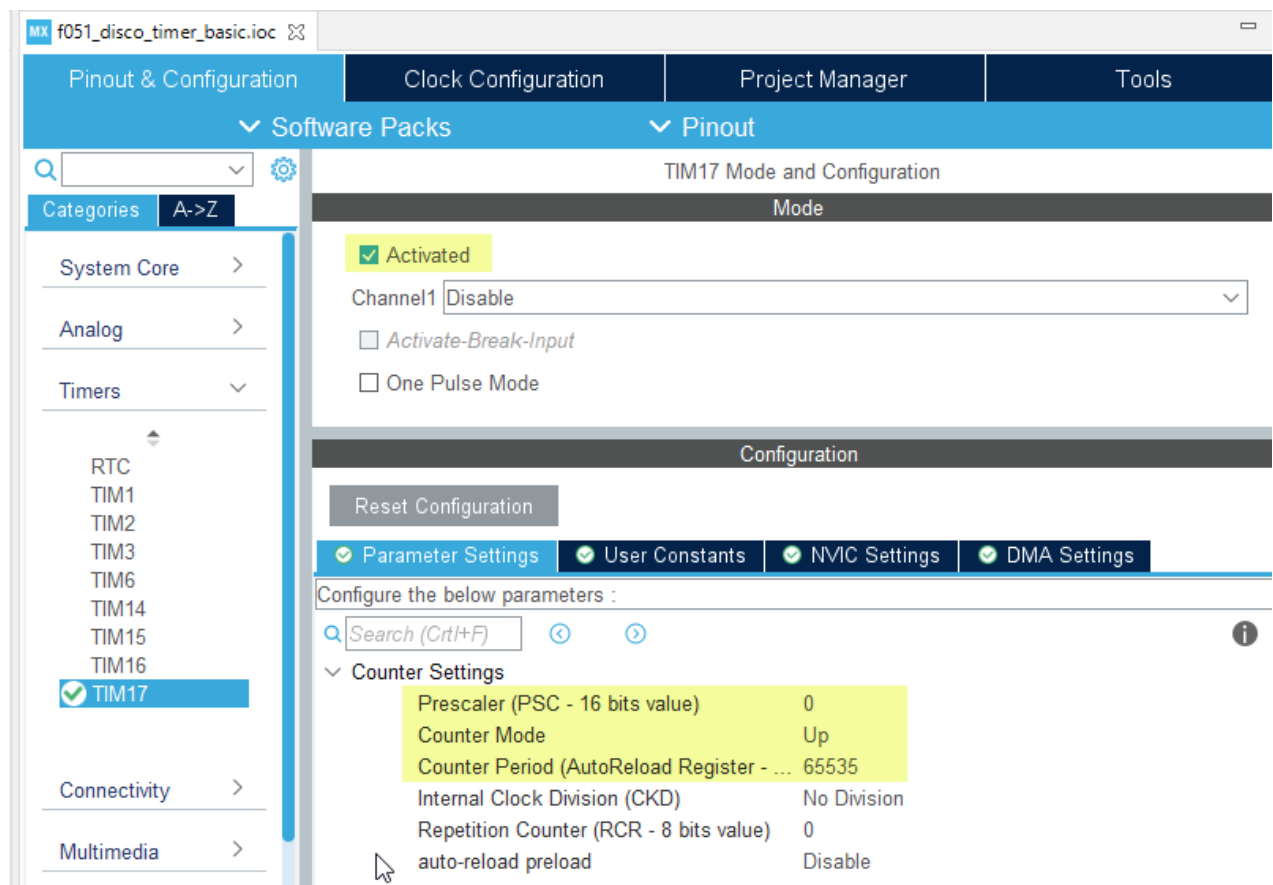
The timer will be used is TIM17, which is driven by APB1 Bus clock running at 48 MHz. The resolution will be  $1 / 48000000 = 20 \text{ us}$ .





Clock path to timers

Just activate the timer and use default settings for this time, as the period is set to 65535 which means the longest counting period is  $20 \text{ us} * 65535 = 1.3 \text{ s}$ .



Enable timer

### 4.3. Generated code

This first function is to init the TIM17 peripheral:

```
static void MX_TIM17_Init(void) {
    htim17.Instance = TIM17;
    htim17.Init.Prescaler = 0;
    htim17.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim17.Init.Period = 65535;
    htim17.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    htim17.Init.RepetitionCounter = 0;
    htim17.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
    if (HAL_TIM_Base_Init(&htim17) != HAL_OK) {
        Error_Handler();
    }
}
```

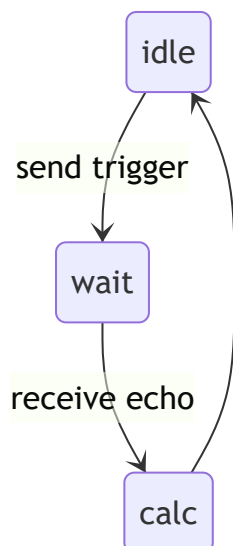
and then the interrupt handler is generated for PA2 input on EXTI2\_3 line:

```
void EXTI2_3_IRQHandler(void) {
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_2);
}
```

### 4.4. User code

#### States

The system will go around 3 states:



There are some variables to hold the system state, timer counter, and distance value.

```
enum {
    IDLE,
```

```

    WAIT,
    CALC
};
int state = IDLE;
int try = 0;
unsigned int counter = 0;
unsigned int distance = 0;

```

## Main loop

The mail loop checks the current state and do corresponding actions:

```

while (1) {
    if (state == IDLE) {
        __HAL_TIM_SET_COUNTER(&htim17, 0);

        HAL_GPIO_WritePin(TRIGGER_GPIO_Port, TRIGGER_Pin, GPIO_PIN_SET);
        HAL_Delay(1);
        HAL_GPIO_WritePin(TRIGGER_GPIO_Port, TRIGGER_Pin, GPIO_PIN_RESET);

        state = WAIT;
        try = 0;
    } else if (state == WAIT) {
        // do nothing
    } else if (state == CALC) {
        counter = __HAL_TIM_GET_COUNTER(&htim17);
        printf("T = %d ~ %d us\n", counter, counter*1000/48000);

        distance = counter * 1657 / 480000;
        // overflow if use * 165700 / 48000000
        printf("D = %d mm\n\r", distance);

        state = IDLE;
        try = 0;
    }

    HAL_Delay(100);
    if(++try > 10) {
        state = IDLE;
    }
}

```

### Overflow

If using wrong data type, overflow will happen if the value to be store is larger than the maximum value of that type.

## Handle ECHO pulse

The ECHO pulse is handled in the `HAL_GPIO_EXTI_Callback()` function called from the `EXTI2_3_IRQHandler()`. This function starts the timer when it detects a rasing edge, and stop the

timer when it gets a falling edge.

The interrupt latency would affect the measured time at which it starts measuring the time. However, it does add the same little bit of lagging for the 2<sup>nd</sup> edge as well so there is not much measurement error at all due to interrupt latency.

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if(GPIO_Pin == ECHO_Pin) {
        if(state == WAIT) {
            if(HAL_GPIO_ReadPin(ECHO_GPIO_Port, ECHO_Pin)) {
                HAL_TIM_Base_Start(&htim17);
            } else {
                HAL_TIM_Base_Stop(&htim17);
                state = CALC;
            }
        }
    }
}
```

The result is shown in a logic analyzer and a serial output as well.



*Output on Echo pin and calculated distance*

## 5. Lab: Timer interrupt

This project will setup a general-purpose timer to operate in timer mode. This timer will fire an event every 100ms to toggle a LED.

Let's start with the previous project.

### 5.1. Add a timer

To get 100ms while using 48 MHz clock, the Pre-scaler and Counter period value have to be set.

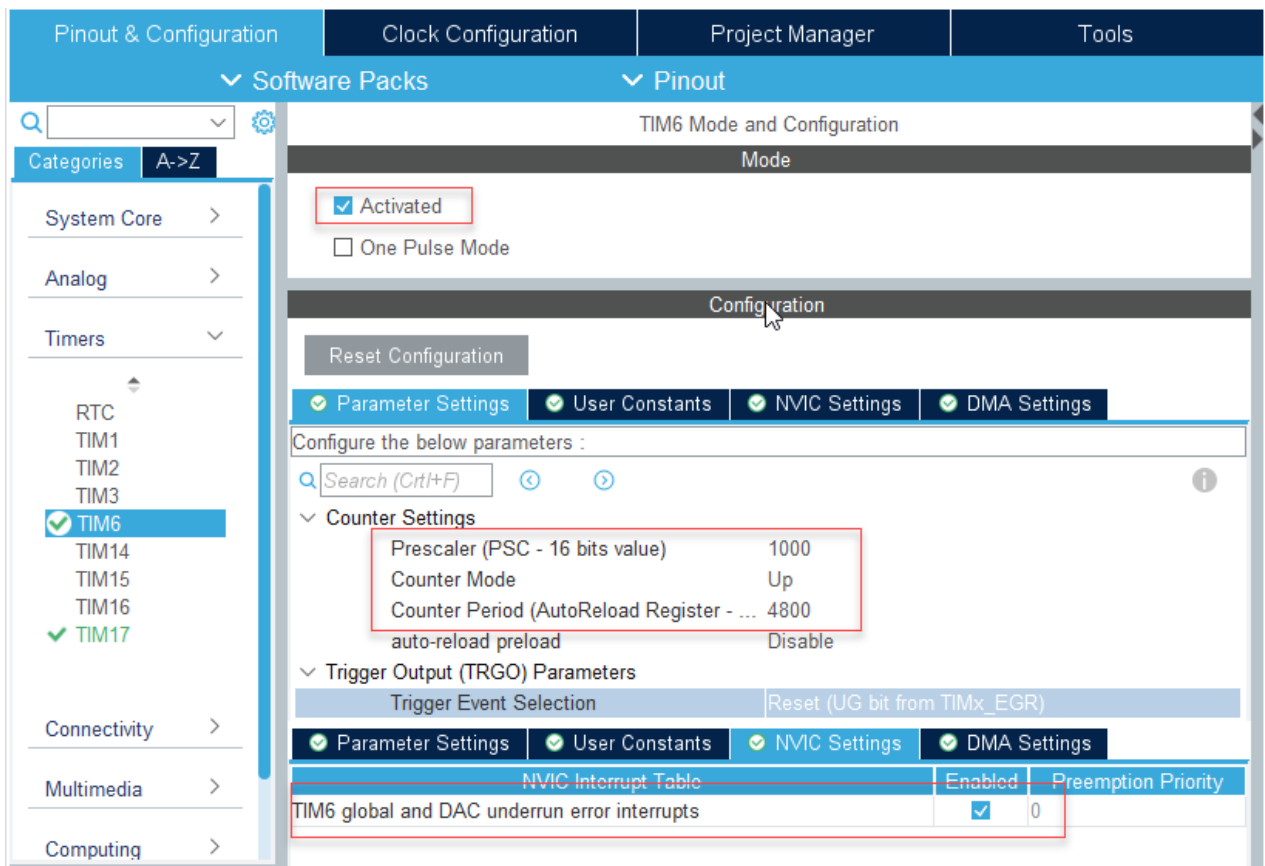
$$\begin{aligned} \text{Time out} &= \text{Counter} * \text{Tick period} \\ &= (\text{Counter Period} - 0) * (\text{Pre-scaler value}) * (1 / \text{Frequency}) \end{aligned}$$

which equals to:

$$\text{Counter Period} * \text{Pre-scaler value} = \text{Time out} * \text{Frequency}$$

In this project, it is simple to choose Prescaler = 1000, and Counter Period = 4800 to make 100 ms interval.

Next step is to enable the interrupt when the timer has counted from 0 to the counter period.



*Enable timer with prescaler, counter period, and interrupt*

## 5.2. Handle callback

Every time when the timer reaches to the counter period value, it fires an event `HAL_TIM_PeriodElapsedCallback()` to notify to system. In this function, toggle the LED:

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if (htim == &htim6) {
        HAL_GPIO_TogglePin(LD3_GPIO_Port, LD3_Pin);
    }
}
```

```

    }
}

```

## 5.3. Start timer

Finally, start the timer in the main function, before the while loop with Interrupt mode:

```

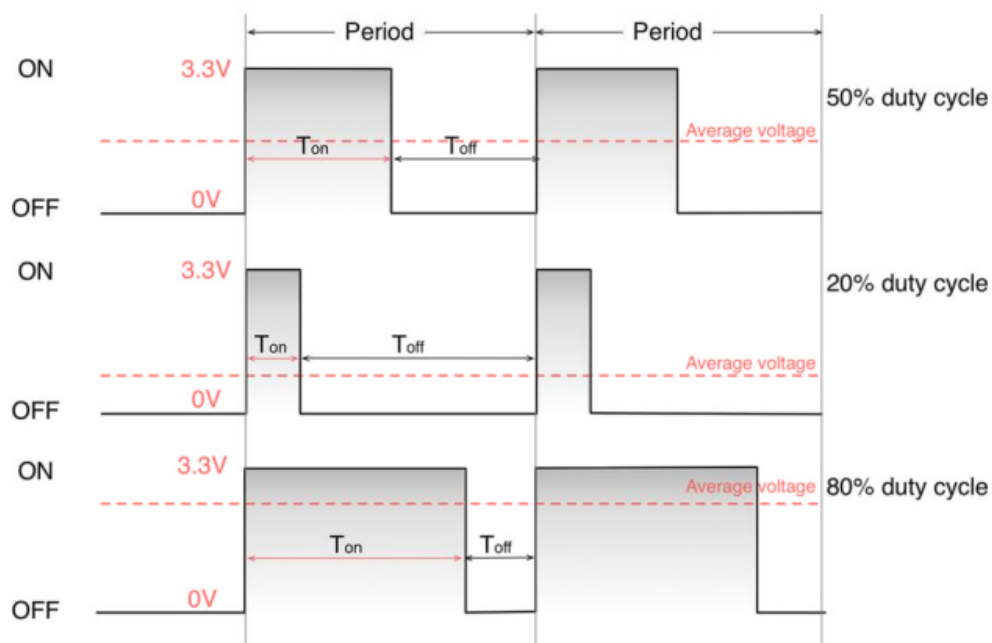
HAL_TIM_Base_Start_IT(&htim6);
while(1) {
    ...
}

```

## 6. Lab: PWM

### 6.1. Duty

The square waves has a common characteristic: they have a  $T_{ON}$  period equal to the  $T_{OFF}$  one. For this reason they are also said to have a 50% duty cycle. A duty cycle is the percentage of one period of time (for example, 1s) in which a signal is active. As a formula, a duty cycle is expressed as the  $T_{ON}/\text{Period}$ .



*Different duty cycles*

*Pulse-width modulation (PWM)* is a technique used to generate several pulses with different duty cycles in a given period of time at a given frequency. PWM has many applications in digital electronics, but all of them can be grouped in two main categories:

- control the output voltage (and hence the current);

- encoding (that is, modulate) a message (that is, a series of bytes in digital electronics) on a carrier wave (which runs at a given frequency).

Those two categories can be expanded in several practical usages of the PWM technique. If only focusing on the control of the output voltage, here are several applications:

- generation of an output voltage ranging from 0V up to VDD (that is, the maximum allowed voltage for an I/O, which in an STM32 is 3.3V);
  - dimming of LEDs;
  - motor control;
  - power conversion;
- generation of an output wave running at a given frequency (sine wave, triangle, square, and so on);
- sound output;

There are two PWM modes available:

- **PWM mode 1:** in up-counting, the channel is active as long as *Period* < *Pulse*, else inactive. In down-counting, the channel is inactive as long as *Period* > *Pulse*, else active.
- **PWM mode 2:** in up-counting, channel is inactive as long as *Period* < *Pulse*, else active. In down-counting, channel 1 is active as long as *Period* > *Pulse*, else inactive.

## 6.2. Drive the brightness

This project will generate changeable-duty PWM signal on PC8 which connected to the blue LED on the F0 Discovery board. If the duty goes up to 100%, the LED will have the highest brightness, and it becomes off when the duty goes down to 0%. The target application should slowly change the duty of the generated PWM.

## 6.3. Setup PWM on Timer

The Blue LED on PC8 is connected to Timer 3 - Channel 3 Output. Therefore, select the Alternate Function of PC8 first. After that, when configuring the TIM3, select Channel 3 as *PWM Generation CH3*.

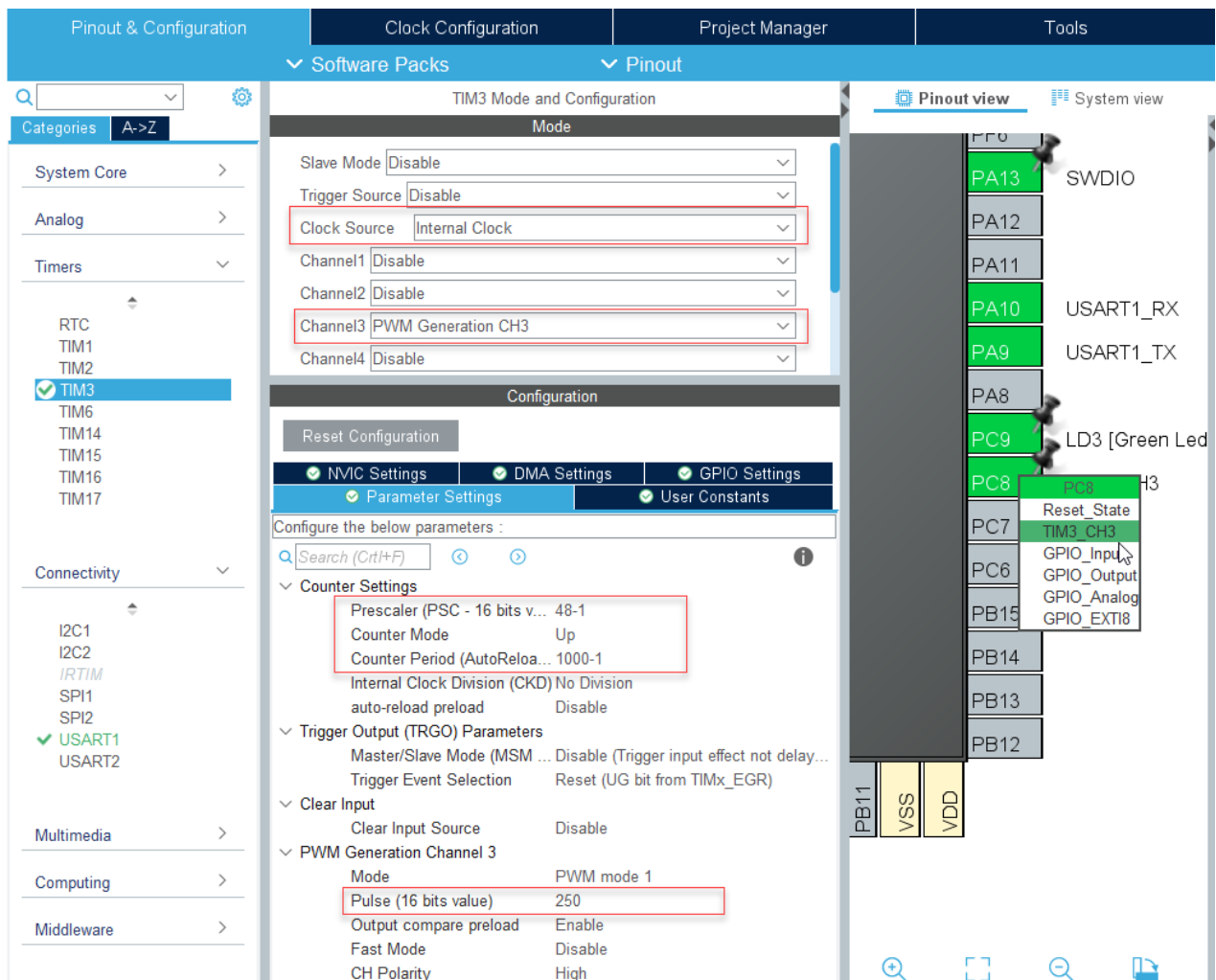
Next step is configure the PWM frequency and duty:

- PWM Frequency = 1000 Hz, it means:

```
Prescaler * Counter period = Timer input clock frequency / 1000
                          = 48000000 / 1000
                          = 48000
```

Let's use *Prescaler* = 48-1 (count from 0 to 47 to get 48 cycles), and *Counter period* = 1000-1.

- PWM Duty: the Counter period is set to 1000, the if duty is 25%, the *Pulse counter* must be set at 250



Setup PWM on TIM3 and its output on PC8

## 6.4. Generated code

The function `MX_TIM3_Init()` is generated to setup TIM3 at Channel 3 with PWM settings.

```
static void MX_TIM3_Init(void) {
    TIM_ClockConfigTypeDef sClockSourceConfig = {0};
    TIM_MasterConfigTypeDef sMasterConfig = {0};
    TIM_OC_InitTypeDef sConfigOC = {0};

    htim3.Instance = TIM3;
    htim3.Init.Prescaler = 48-1;
    htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim3.Init.Period = 1000-1;
```



```

htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
if (HAL_TIM_Base_Init(&htim3) != HAL_OK) {
    Error_Handler();
}

sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig) != HAL_OK) {
    Error_Handler();
}

if (HAL_TIM_PWM_Init(&htim3) != HAL_OK) {
    Error_Handler();
}

sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) != HAL_OK) {
    Error_Handler();
}

sConfigOC.OCMode = TIM_OCMode_PWM1;
sConfigOC.Pulse = 250;
sConfigOC.OCpolarity = TIM_OCPOLARITY_HIGH;
sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
if (HAL_TIM_PWM_ConfigChannel(&htim3, &sConfigOC, TIM_CHANNEL_3) != HAL_OK) {
    Error_Handler();
}
HAL_TIM_MspPostInit(&htim3);
}

```

## 6.5. User code

To dynamically change the duty of the PWM, there are variables holding the PWM Period and PWM pulse value to calculate the duty and set a new value.

```

uint16_t period = 0;
uint16_t duty = 0;

```

HAL provides some macros to access to register level directly, for example, below macros are used to get period and pulse value. Note that pulse value actually is the output compare value. Timer constantly compares the time counter value with output compare value to generate output level.

```


period = __HAL_TIM_GET_AUTORELOAD(&htim3) + 1;
duty = __HAL_TIM_GET_COMPARE(&htim3, TIM_CHANNEL_3);

```

Then start the timer in PWM mode on the channel 3, before going to the while loop. Inside the while loop, change the duty of PWM by calculating the pulse value and set it to the output

compare register:

```
HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_3);
while (1) {
    HAL_Delay(100);
    duty += 10;
    if(duty >= 100) {
        duty = 10;
    }
    __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_3, duty * period / 100);
}
```

 Changing the PWM duty too fast may lead to bad result, as the average voltage of each cycle is not clearly affect the output. In the above example, PWM duty is keep constant in 100ms before it is changed.