Timers and their modes

Embedded devices perform some activities on a time basis. Timers are hardware modules that assist CPU to not only generate periodically events, but also do several features in which some are very important to work with external peripheral like Pulse Width Modulation, Pulse Capture. Timer is one of advanced peripheral that offer a wide range of customizations.

#arm #stm32 #timer #pwm #capture

Last update: 2021-06-14 14:07:22

Table of Content

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 a “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. A timer can have additional pre-load register, therefore, timer will count from 0 to the pre-load value, and then go back to count again from 0.

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

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

Main groups of Timers:

- **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 pre-load 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 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. STM32Cube HAL Usage

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 to be 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 1: Blink LED with Timer

This is a basic usage of a timer. Application will enable a basic timer with a pre-scaler and a pre-load value and let the timer run. Then the timer will keep counting and fire up an interrupt to application to do something.

Requirements:

- Use a Timer to toggle an LED every 250ms

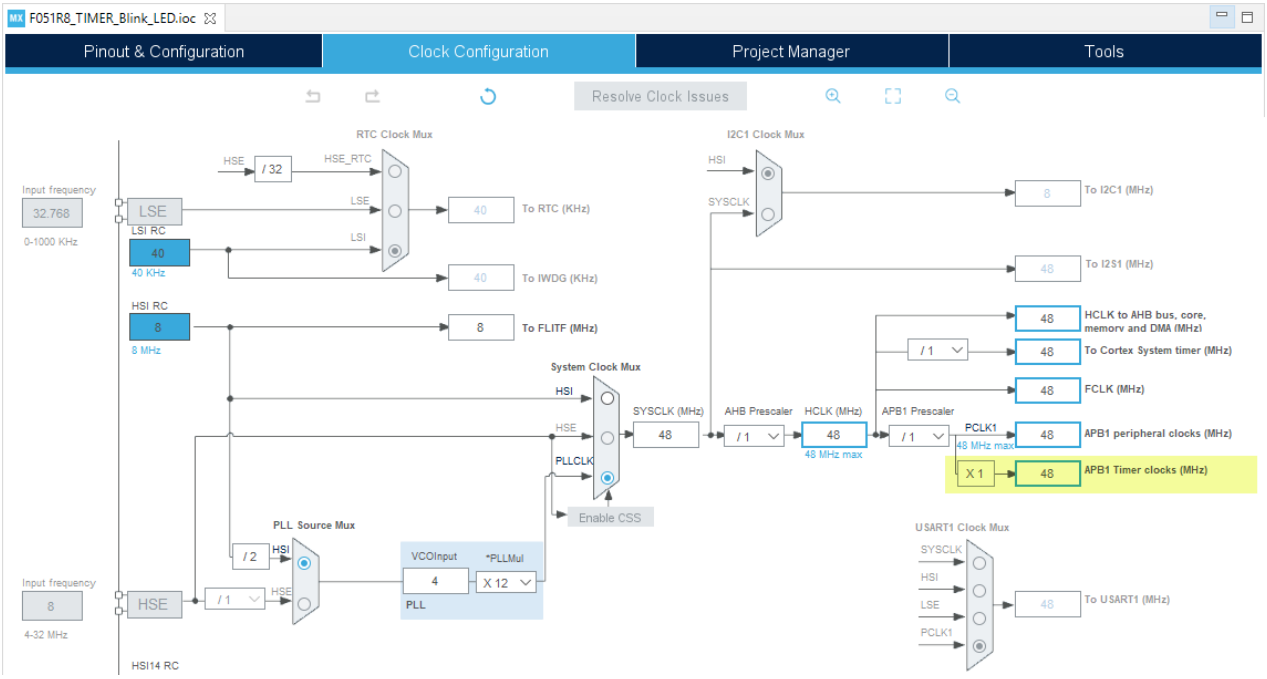
Target board:

Any board as a basic timer and a GPIO are always available.

4.1. Create a new project

This tutorial will use the STM32F0 Discovery board which features an STM32F051R8 MCU. Starting a new project with below starting configs:

- System Clock set to 48 MHz, which also drives the Timers on the APB bus.
- Set **PC9** as the output for the on-board Blue LED



Setup clock for Timers

4.2. Enable a Timer

For this simple lab, a basic timer will be used. Look at the Datasheet of the MCU to know how many timers are in the MCU and their types, and read the Reference Manual to get details about a specific timer. Note that a specific timer TIMx is the same in all STM32 MCUs to guarantee the compatibility and portability of the timer on different target.

> **STM32F051xx Timers**

The STM32F051xx devices include up to six general-purpose timers, one basic timer and an advanced control timer.

Timer type	Timer	Counter resolution	Counter type	Prescaler factor	DMA request generation	Capture/compare channels	Complementary outputs
Advanced control	TIM1	16-bit	Up, down, up/down	integer from 1 to 65536	Yes	4	3
General purpose	TIM2	32-bit	Up, down, up/down	integer from 1 to 65536	Yes	4	-
	TIM3	16-bit	Up, down, up/down	integer from 1 to 65536	Yes	4	-
	TIM14	16-bit	Up	integer from 1 to 65536	No	1	-
	TIM15	16-bit	Up	integer from 1 to 65536	Yes	2	1
	TIM16 TIM17	16-bit	Up	integer from 1 to 65536	Yes	1	1
Basic	TIM6	16-bit	Up	integer from 1 to 65536	Yes	-	-

Timer features in STM32F051xx

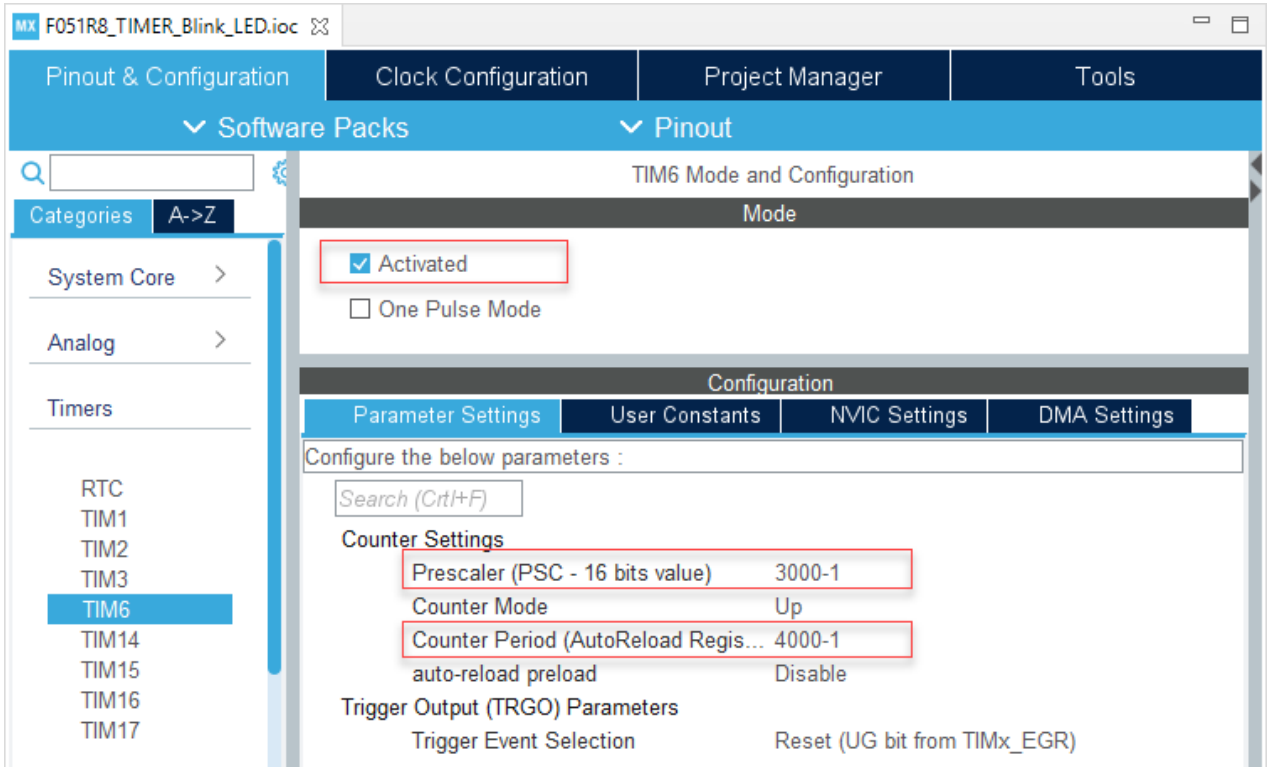
The basic timer TIM6 is mainly used for DAC trigger generation. It can also be used as a generic 16-bit time base.

> **TIM6/TIM7 main features**

- 16-bit auto-reload up-counter
- 16-bit programmable pre-scaler used to divide (also “on the fly”) the counter clock frequency
- Synchronization circuit to trigger the DAC
- Interrupt/DMA generation on the update event: counter overflow

In the CubeMX, activate TIM6 module and then calculate the pre-scaler and auto reload register values:

- Toggle every 250ms, meaning the rate is 4 Hz
- Timer frequency is 48 MHz, so it should be reduced 12.000.000 times. Choose any combination of pre-scaler and auto-reload whose multiplication result is 12.000.000. Because these values are counted from zero, therefore, the filled number should be decreased by one.
- One Pulse mode will make timer run once, do not select this to put timer in repeat mode.
- Enable the Interrupt for this timer



Setting a basic timer

> **Do NOT use Polling mode on Timer**
>
> Polling mode on Timer is just keep reading the timer's counter to compare with a given counter. However, please note that the timer is an independent asynchronous peripheral which may run at higher frequency of the CPU core.
>
> This line of code `if(__HAL_TIM_GET_COUNTER(&tim) == value){...}` does not guarantee that the CPU accesses to the counter register exactly at the same time the timer reaches the given value.

4.3. Generated code

After code generation, there is a function `MX_TIM6_Init()` to initialize the activated timers.

```
static void MX_TIM6_Init(void) {
    TIM_MasterConfigTypeDef sMasterConfig = {0};

    htim6.Instance = TIM6;
    htim6.Init.Prescaler = 3000-1;
    htim6.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim6.Init.Period = 4000-1;
    htim6.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;

    if (HAL_TIM_Base_Init(&htim6) != HAL_OK)
    {
        Error_Handler();
    }
```

```
    sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    if (HAL_TIMEx_MasterConfigSynchronization(&htim6, &sMasterConfig)
        != HAL_OK)
    {
        Error_Handler();
    }
}
```

The clock and interrupt of the selected timer is configured in `HAL_TIM_Base_MspInit()` function:

```
void HAL_TIM_Base_MspInit(TIM_HandleTypeDef* htim_base) {
    __HAL_RCC_TIM6_CLK_ENABLE();
    HAL_NVIC_SetPriority(TIM6_DAC_IRQn, 0, 0);
    HAL_NVIC_EnableIRQ(TIM6_DAC_IRQn);
}
```

Because the timer has enabled its interrupt, HAL calls to `HAL_TIM_IRQHandler()` in the interrupt handler, then finally informs to application via different callback. In this lab, application only need to know when a full period is done through the callback of `HAL_TIM_PeriodElapsedCallback()` and toggle the LED.

4.4. User code

This simple lab only needs to override the callback when a full period elapses.

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
  HAL_GPIO_TogglePin(LED_GPIO_Port, LED_Pin);
}
```

And then start the timer in Interrupt mode in the main application:

```
int main() {
    HAL_TIM_Base_Start_IT(&htim6);
    while (1){...}
}
```

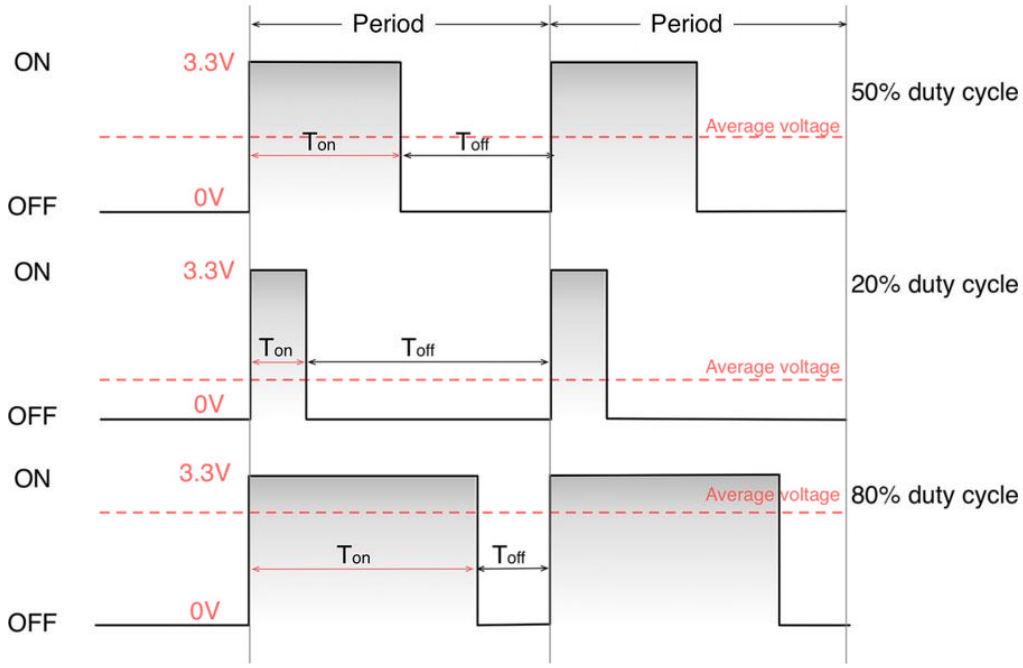
5. Lab 2: PWM on LED brightness

5.1. PWM 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}/Period.

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:



PWM output and its average voltage

- 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:

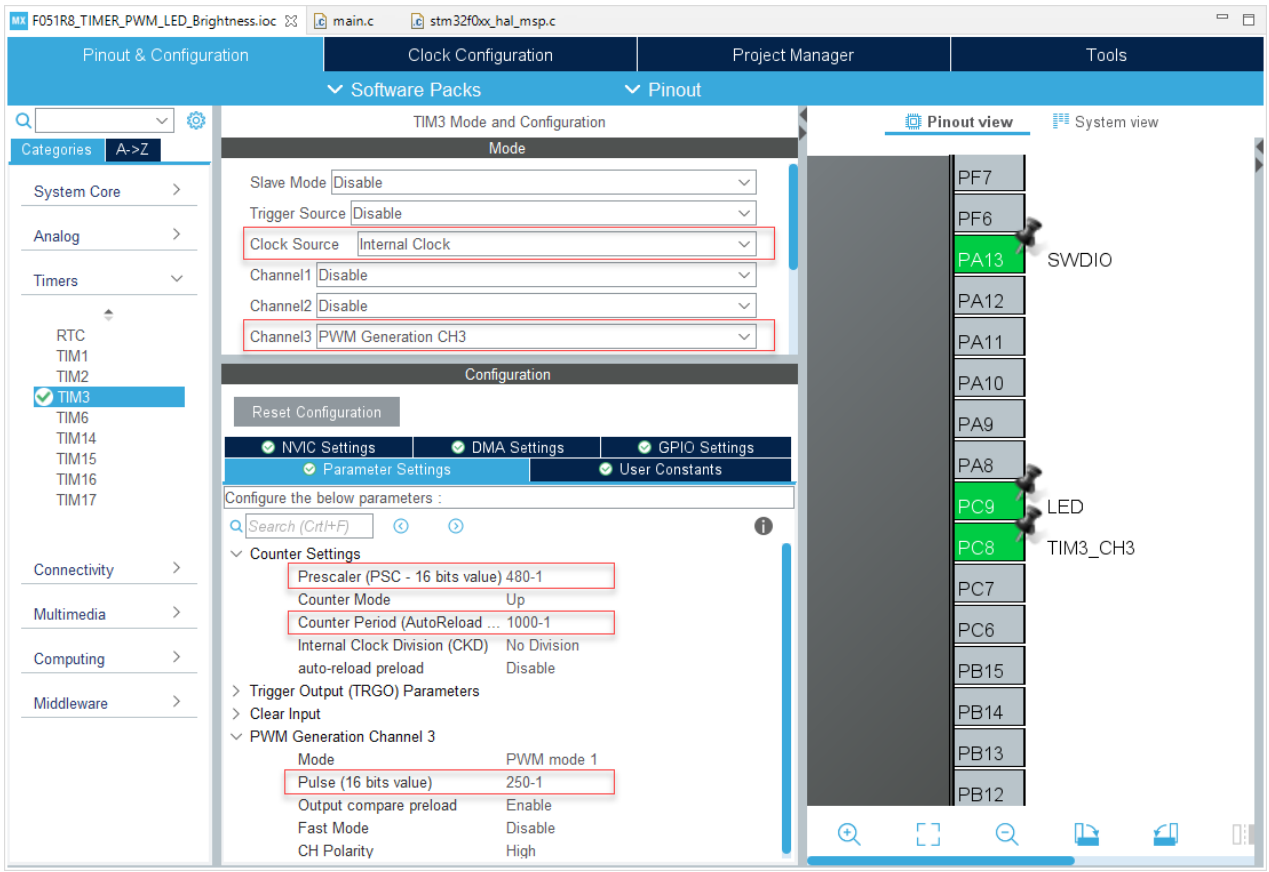
- **PWM mode 1**: in up-counting, the channel is active as long as *Counter* < *Period*, else inactive. In down-counting, the channel is inactive as long as *Counter* > *Period*, else active.
- **PWM mode 2**: in up-counting, channel is inactive as long as *Counter* < *Period*, else active. In down-counting,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

5.2. Drive the brightness

This lab will generate changeable-duty PWM signal on pin **PC8** which connected to the blue LED on the STM32F0 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.

5.3. Setup PWM on a Timer

The Blue LED on **PC8** is connected to **Timer 3 - Channel 3 Output**. Therefore, select the Alternate Function of **PC8** as **TIM3_CH3** firstly. After that, when configuring the **TIM3**, select **Channel 3** as *PWM Generation CH3*.



Setup PWM on the Timer 3 Channel 3

Next step is configure the PWM frequency and duty:

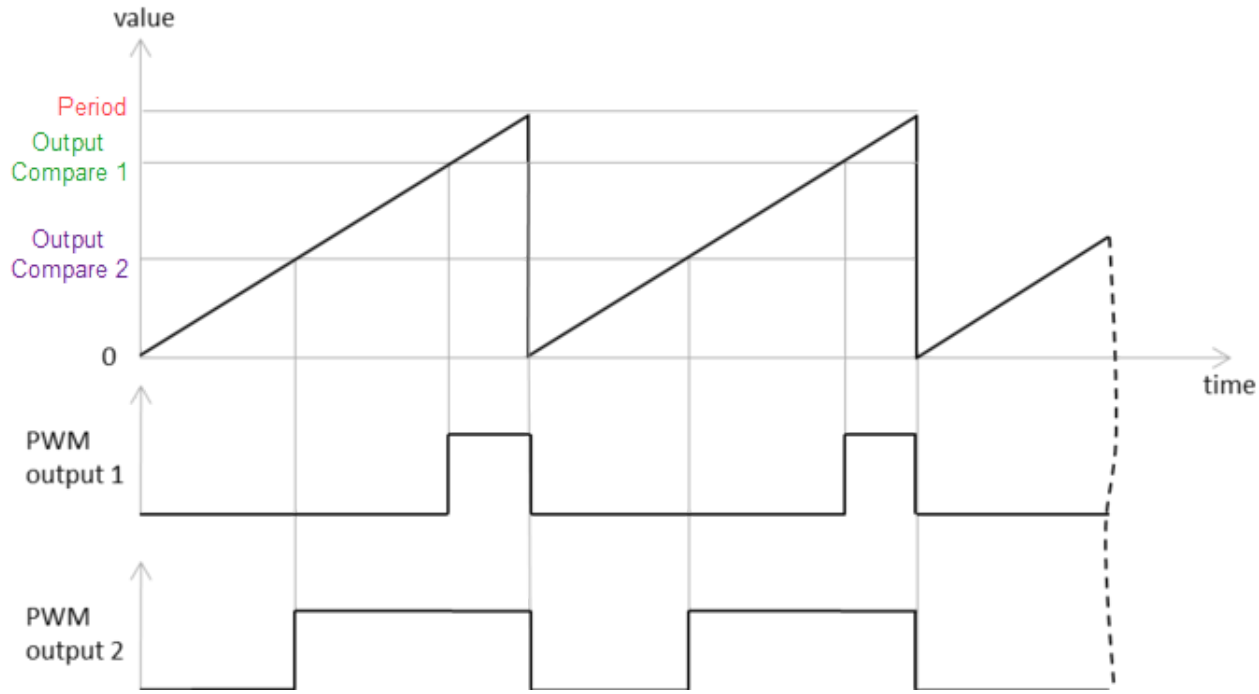
- Use the internal clock source, which is 48 MHz in this lab
- PWM Frequency = 100 Hz, let's use `Pre-scaler` = 480-1, and `Counter period` = 1000-1.
- PWM Duty: the `Counter period` is set to 1000, the if duty is 25%, the `Pulse counter` must be set at 250-1

5.4. Setup delay on a Timer

Do the same thing as it's done in the previous lab to setup an 100 Hz interrupt on a basic timer TIM6. The period is now only 10 ms

5.5. Generated code

Let's see how TIM3 is configured in the generated function `MX_TIM3_Init()` which setups the timer's params and setup PWM mode. Note that PWM is Output Compare register to mark the point where pulse is inverted.



PWM output based on Output Compare and Period registers

```
static void MX_TIM3_Init(void) {
  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};
  TIM_OC_InitTypeDef sConfigOC = {0};
  /* Setup base */
  htim3.Instance = TIM3;
  htim3.Init.Prescaler = 480-1;
  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();
  }
  /* Setup clock source */
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig) != HAL_OK) {
    Error_Handler();
  }
  /* Init PWM */
  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();
  }
  /* Setup PWM params*/
  sConfigOC.OCMode = TIM_OCMODE_PWM1;
  sConfigOC.Pulse = 250-1;
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  if (HAL_TIM_PWM_ConfigChannel(&htim3, &sConfigOC, TIM_CHANNEL_3)
        != HAL_OK) {
    Error_Handler();
  }
  /* Setup GPIO for PWM Output */
  HAL_TIM_MspPostInit(&htim3);
}
```

5.6. User code

The `Counter period` (aka. `Auto Reload`) is stored in the register `ARR` of the timer. And the duty of the PWM is set in the `Compare Control` register named `CRx`.

In the main application, save the PWM duty and use it to control PWM duty later. All timers must be started exclusively.

```
uint16_t dutyCycle = 0;
int main() {
    dutyCycle = __HAL_TIM_GET_AUTORELOAD(&htim3);
    HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_3);
    HAL_TIM_Base_Start_IT(&htim6);
    while (1) {
        HAL_Delay(1000);
    }
}
```

To change the duty of the output PWM inside the 100 Hz basic timer's interrupt, override the callback and set the Output Compare value with changed duty.

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
  if(dutyCycle >= __HAL_TIM_GET_AUTORELOAD(&htim3)) {
    dutyInc = -1;
  } else if(dutyCycle == 0) {
    dutyInc = +1;
  }
  dutyCycle += dutyInc;
  __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_3, dutyCycle);
}
```

PWM at 100Hz with changing duty - video

LED brightness with changing PWM

6. Other modes

Other modes of timers will be covered in some examples in up coming posts.