TIM1&TIM8 advanced control timer

Also, TIM2 to TIM5 and TIM9 to TIM14: general purpose timers

TIM2 to TIM5 main features
General-purpose TIMx timer features include:

- 16-bit (TIM3 and TIM4) or 32-bit (TIM2 and TIM5) up, down, up/down auto-reload counter.
- 16-bit programmable prescaler used to divide (also “on the fly”) the counter clock frequency by any factor between 1 and 65536.
- Up to 4 independent channels for:
    - – Input capture
    - – Output compare
    - – PWM generation (Edge- and Center-aligned modes)
    - – One-pulse mode output
- Synchronization circuit to control the timer with external signals and to interconnect several timers.
- Interrupt/DMA generation on the following events:
    - – Update: counter overflow/underflow, counter initialization (by software or internal/external trigger)
    - – Trigger event (counter start, stop, initialization or count by internal/external trigger)
    - – Input capture
    - – Output compare
- Supports incremental (quadrature) encoder and hall-sensor circuitry for positioning purposes
- Trigger input for external clock or cycle-by-cycle current management

The time-base unit includes:

- Counter Register (TIMx_CNT)
- Prescaler Register (TIMx_PSC):
- Auto-Reload Register (TIMx_ARR)

The prescaler can divide the counter clock frequency by any factor between 1 and 65536. It is based on a 16-bit counter controlled through a 16-bit/32-bit register (in the TIMx_PSC register). It can be changed on the fly as this control register is buffered. The new prescaler ratio is considered at the next update event.

Clock selection
The counter clock can be provided by the following clock sources:

- Internal clock (CK_INT)
- External clock mode1: external input pin (TIx)
- External clock mode2: external trigger input (ETR) available on TIM2, TIM3 and TIM4 only.
- Internal trigger inputs (ITRx): using one timer as prescaler for another timer, for example, Timer can be configured to act as a prescaler for Timer 2. Refer to Using one timer as prescaler for another for more details.

Key Differences:

1. **Complexity and Functionality:**
   - **Advanced Control Timers:** Advanced features such as dead-time, break input, and support for motor control.
   - **General Purpose Timers:** Basic features sufficient for general timing and counting operations.
2. **Applications:**
   - **Advanced Control Timers:** Ideal for motor control applications, power conversion systems, and applications requiring sophisticated PWM.
   - **General Purpose Timers:** Suitable for standard timing operations, handling simple PWM signals, and event counting.
3. **Number of Channels:**
   - **Advanced Control Timers:** Up to three or four PWM channels with additional functionalities.
   - **General Purpose Timers:** Up to four channels, but without the advanced features for motor control.

HAL:
Functions for programming the basics: HAL_TIM_Base_XXX

```
typedef struct {
  TIM_TypeDef *Instance;  /* Timer Register base addr */
  TIM_Base_InitTypeDef Init;  /* TIM Time Base params */
  HAL_TIM_ActiveChannel Channel;
  DMA_HandleTypeDef *hdma[7];  /* DMA handlers */
  HAL_LockTypeDef Lock;  /* Locking object */
  __IO HAL_TIM_StateTypeDef State;  /* timer state  */
} TIM_HandleTypeDef;

typedef struct {
uint32_t Prescaler; /* Prescaler value (PSC) */
uint32_t CounterMode; /*Counter mode (up, down,cent.)*/
uint32_t Period; /* Value of Auto-reload reg (ARR) */
uint32_t ClockDivision; /* clock division */
uint32_t RepetitionCounter; /* Repetition counter val.*/
} TIM_Base_InitTypeDef;
```

To **modify** a parameter:

- Htim6.Init.Prescaler = ...;
- HAL_TIM_Base_Init(&htim6);

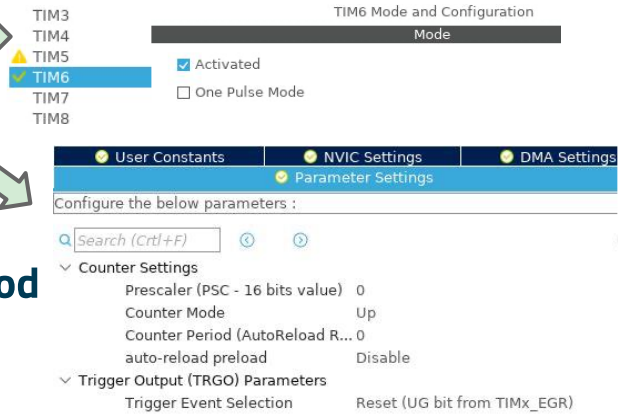
Prescaler : divides the timer clock by a factor ranging from 1 up to 2^16-1 = 65535 (16-bit even though the field is uint32_t!)
CounterMode : defines the counting direction of the timer: TIM_COUNTERMODE_UP (default)
Period : maximum value for the timer counter before it restarts counting again (ARR register)

- 16-bit timers: from 0x1 to 0xFFFF (65535)
- 32-bit timers: from 0x1 to 0xFFFF FFFF (57813)

		APB1	on page 931
0x4000 3C00 - 0x4000 3FFF	SPI3 / I2S3		Section 26.7.10: SPI register map on page 896
0x4000 3800 - 0x4000 3BFF	SPI2 / I2S2		
0x4000 3000 - 0x4000 33FF	IWDG		Section 20.4.5: IWDG register map on page 645
0x4000 2C00 - 0x4000 2FFF	WWDG		Section 21.6.4: WWDG register map on page 652
0x4000 2800 - 0x4000 2BFF	RTC & BKP Registers		Section 22.6.21: RTC register map on page 690
0x4000 2000 - 0x4000 23FF	TIM14		Section 18.5.12: TIM10/11/13/14 register map on page 626
0x4000 1C00 - 0x4000 1FFF	TIM13		
0x4000 1800 - 0x4000 1BFF	TIM12		
0x4000 1400 - 0x4000 17FF	TIM7		Section 19.4.9: TIM6&TIM7 register map on page 639
0x4000 1000 - 0x4000 13FF	TIM6		
0x4000 0C00 - 0x4000 0FFF	TIM5		Section 17.4.21: TIMx register map on page 579
0x4000 0800 - 0x4000 0BFF	TIM4		
0x4000 0400 - 0x4000 07FF	TIM3		
0x4000 0000 - 0x4000 03FF	TIM2		

APB1 clock's maximum frequency is 45MHz

First locate the timer you want to enable
Then click on "activate"
Once the timer is activated it can be configured using the configuration panel
To set the period configure **prescaler** and **Counter Period**
CubeMX reports the resolution of the timer in the "counter period" tooltip



In the NVIC settings panel click on **"Enable"** checkbox to enable the timer to generate interrupts
Once the code is generated the code to be executed when the timer generates an update event should be implemented in the callback:
HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) that is called in the ISR, in the *stm32xx_hal_tim.c*:

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
/* Need to check which timer generated the interrupt! */
  if(htim->Instance == TIM6) {
    HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
  } else if (htim->Instance == TIM7) {
    /* Do something else due to TIM7 interrupt */
}}
```

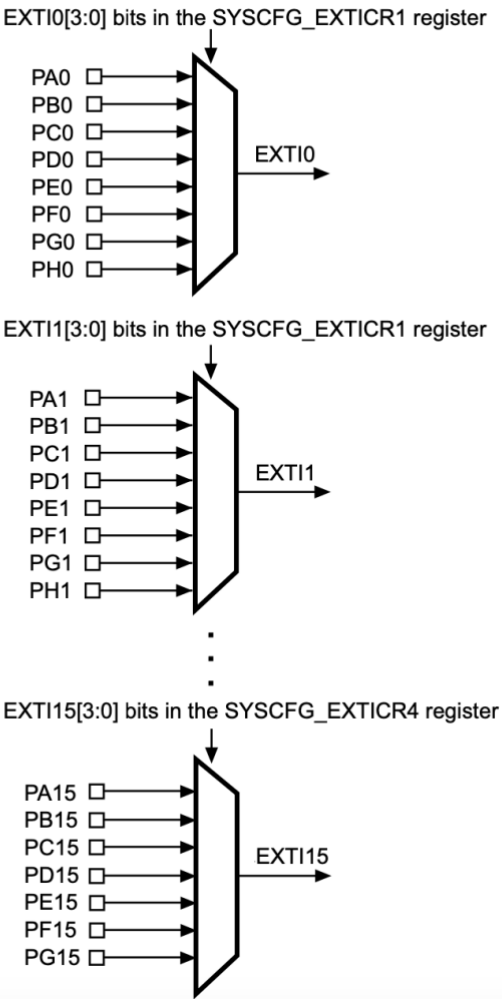
(Callbacks are functions that are called from within the interrupt handler, typically provided by the HAL (Hardware Abstraction Layer) library in STM32. Callbacks allow you to write your application-specific code without modifying the HAL code.)

TIM6_IRQHandler(void) of the corresponding IRQ, in the *stm32f4xx_it.c* (An interrupt handler, also known as an IRQ handler, is a function that is executed in response to an interrupt. In STM32, each interrupt source is associated with a specific handler function. These functions

usually have a specific name format like XXX_IRQHandler where XXX is the name of the peripheral or interrupt source.)

Once the timer has been configured, the timer can be started using, in the *main.c* before the *while(1)*: HAL_TIM_Base_Start_IT(TIM_HandleTypeDef *hdl) if the timer is configured in interrupt mode

Figure 31. External interrupt/event GPIO mapping



1.

```
void EXTI0_IRQHandler(void) {
HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_0);  // Chiama la funzione HAL per gestire l'interrupt
}
```

2.

```
void HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin) {
// Codice per gestire l'evento EXTI...
```

3.

```
// Chiama la callback associata all'interrupt GPIO
HAL_GPIO_EXTI_Callback(GPIO_Pin);
}
```

4.

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
    if (GPIO_Pin == GPIO_PIN_0) {
// Codice personalizzato da eseguire quando si verifica l'interrupt su GPIO_PIN_0
    // Ad esempio, cambiare lo stato di un LED
    HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_0);
    }
}
```

It is possible to configure at runtime the parameters of a timer using some macros:

```
void set_basic_timer_params(TIM_HandleTypeDef *hdl, uint32_t prescaler, uint32_t period)
{
  __HAL_TIM_SET_PRESCALER(hdl, prescaler);
  __HAL_TIM_SET_AUTORELOAD(hdl, period);
  hdl->Instance->EGR = TIM_EGR_UG;
}
```

PWM:

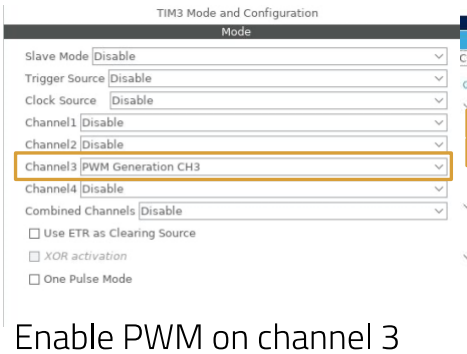
In this case, we don't need to use it in interrupt mode

Duty cycle can be selected by changing the "pulse" value stored in the CCR register

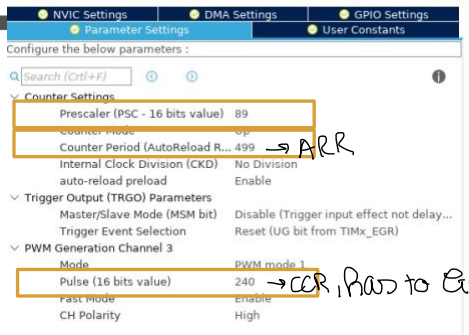
$$f_{CLK\_CNT} = f_{CLK}/(PSC+1)$$

$$CP = \textbf{Period} = \frac{(1+ARR) \times (PSC+1)}{f_{CLK}} \qquad \delta = \frac{T_{ON}}{T} = \frac{CCR \cdot T_0}{(ARR+1)\,T_0}$$

$$CP = \frac{(1+ARR)}{f_{CLK\_CNT}} = (1+ARR)\,T_{CNT}$$



Enable PWM on channel 3



Configure the channel to have a given period and duty cycle

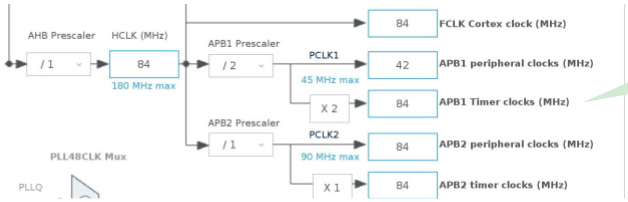
Start the PWM with, in the user code begin 2 after the initialization of the peripheral,

HAL_TIM_PWM_Start(cc, uint32_t Channel)

&htim2, TIM_CHANNEL_1, … , TIM_CHANNEL_4

Declare ccr as a uint32_t in the user code begin 1 in the *main.c*
Compute the value of the CCR register to impose such a DC in the *while(1)*:
ccr = (uint16_t)(duty_cycle*(float)(1+ARR))
▪ Assign this value to the CCR register: __HAL_TIM_SET_COMPARE(&htim, TIM_CHANNEL_X, ccr);
▪ hdl->Instance->EGR = TIM_EGR_UG;

Set timer TIM2 (GP timer) in PWM mode on channel 2
We want to set a PWM frequency of f_{PWM}=**1KHz**
First let's check what is the source frequency of TIM2. The manual says TIM2 is connected to APB1 timer bus (also check the `#define TIM2_BASE` in stm32f446xx.h)
Locate APB1 timer clocks in clock configuration in CubeMX



Here it is!
84MHz

Set PSC and ARR accordingly: $f_{\mathrm{PWM}} = \frac{f_{\mathrm{APB1}}}{(\mathrm{ARR}+1)(\mathrm{PSC}+1)}$

We readily obtain: $(\mathrm{ARR}+1)(\mathrm{PSC}+1) = \frac{f_{\mathrm{APB1}}}{f_{\mathrm{PWM}}}$

$= \frac{84 \cdot 10^6\,\mathrm{Hz}}{1 \cdot 10^3\,\mathrm{Hz}} = 84000 > 2^{16} - 1$

Select the lowest possible PSC such that ARR can be the highest and (ARR+1)(PSC+1) is as close as possible to 84000 (in this case PSC = 1, ARR = 41999)
→ the best resolution possible

HAL_DELAY(...) to change the pwm after some time in the main