Timers

rev1.0 24/03/2020

GOAL

Blink the on-board LEDs using timers interrupts

PREREQUISITES

Software needed:

- STM32IDE

Hardware used in this example:

- **NUCLEO-F446ZE**

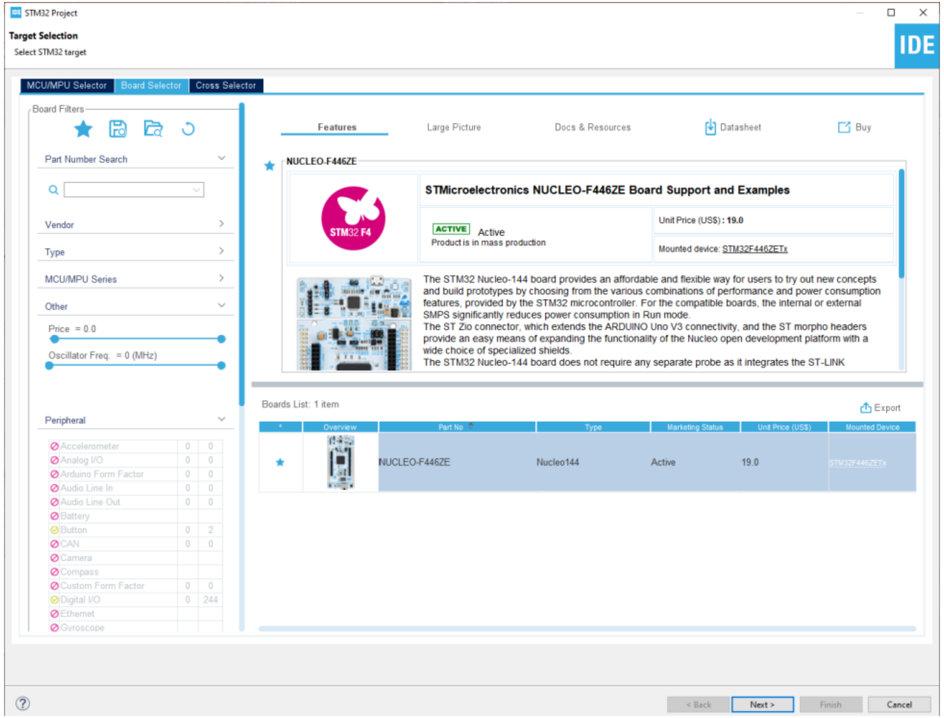
What is a Timer?

- The timer is a tool that allows the timing of some actions.
- They are very useful for having time references: it is important to note that they keep the date and time reference but only time intervals
- They can generate interrupts at regular time intervals
- Within the microcontroller there are several.

Start a new project

From the stm32IDE software click on File -> New -> STM32 Project.

Select your board or your uC and click *next.*



Target Selection

Select STM32 target

Price = 0.0

Oscillator Freq. = 0 (MHz)

Accelerometer	0	0
Analog I/O	0	0
Arduino Form Factor	0	0
Audio Line In	0	0
Audio Line Out	0	0
Battery		
Button	0	2
CAN	0	0
Camera		
Compass		
Custom Form Factor	0	0
Digital I/O	0	244
Ethernet		
Gyroscope		

NUCLEO-F446ZE

STMicroelectronics NUCLEO-F446ZE Board Support and Examples

Active
Product is in mass production

Unit Price (US$) : **19.0**

Mounted device: STM32F446ZETx

The STM32 Nucleo-144 board provides an affordable and flexible way for users to try out new concepts and build prototypes by choosing from the various combinations of performance and power consumption features, provided by the STM32 microcontroller. For the compatible boards, the internal or external SMPS significantly reduces power consumption in Run mode.
The ST Zio connector, which extends the ARDUINO Uno V3 connectivity, and the ST morpho headers provide an easy means of expanding the functionality of the Nucleo open development platform with a wide choice of specialized shields.
The STM32 Nucleo-144 board does not require any separate probe as it integrates the ST-LINK

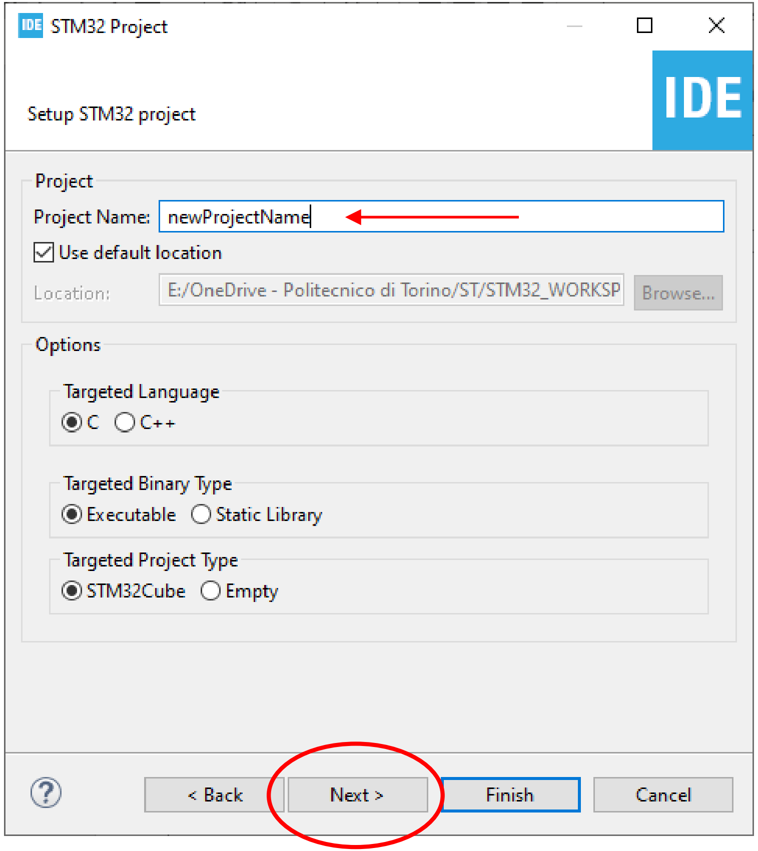
Boards List: 1 item

*	Overview	Part No	Type	Marketing Status	Unit Price (US$)	Mounted Device
★		NUCLEO-F446ZE	Nucleo144	Active	19.0	STM32F446ZETx

Start a new project

Type the name of your project and click next.

By default the project will be created in the workspace folder.



STM32 Project

Setup STM32 project

Project

Project Name: newProjectName

☑ Use default location

Location: E:/OneDrive - Politecnico di Torino/ST/STM32_WORKSP

Options

Targeted Language
● C ○ C++

Targeted Binary Type
● Executable ○ Static Library

Targeted Project Type
● STM32Cube ○ Empty

Start a new project

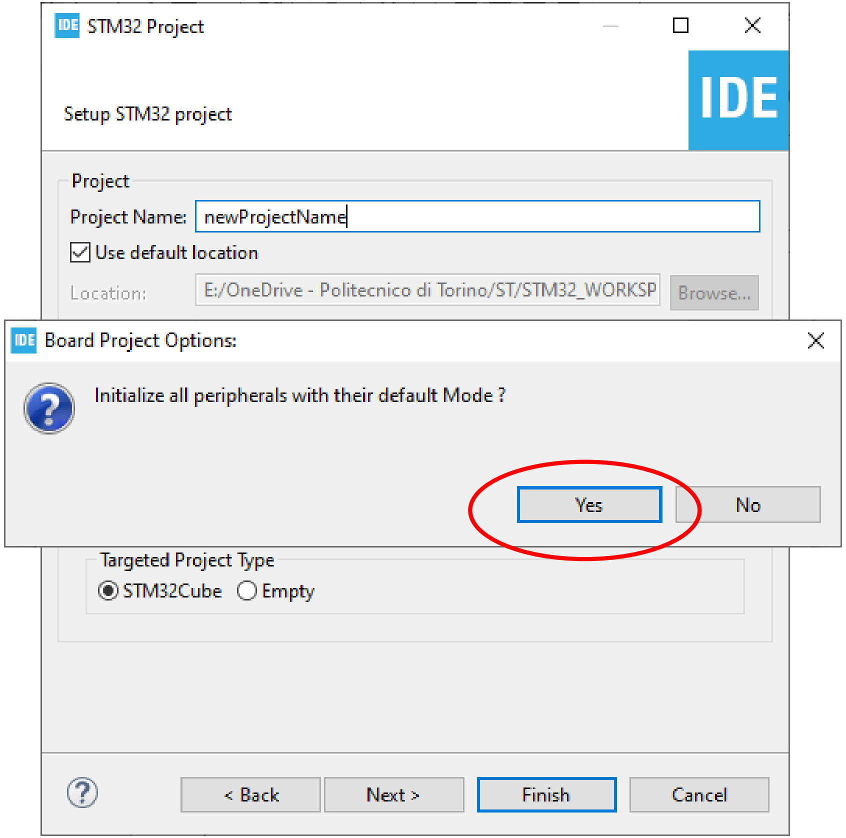
Type the name of your project and click next.

By default the project will be created in the *workspace* folder.

The *STM32IDE* has the option to initialize all the peripheral with their ***default*** mode:

Clicking *Yes* the *USART3*, all the *LEDs* and the blue *UserButton* will be configured as default.

Click *Yes*.



STM32 Project

Setup STM32 project

Project

Project Name: newProjectName

☑ Use default location

Location: E:/OneDrive - Politecnico di Torino/ST/STM32_WORKSP

Board Project Options:

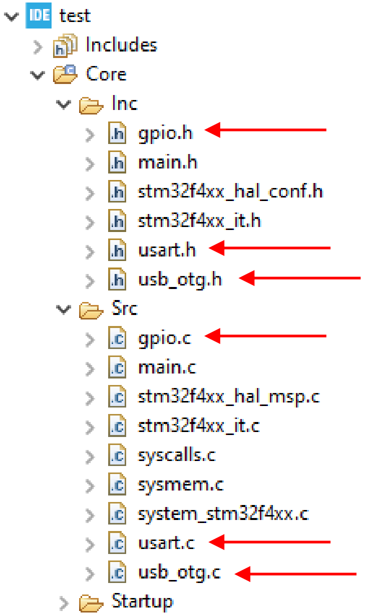
Initialize all peripherals with their default Mode ?

Targeted Project Type

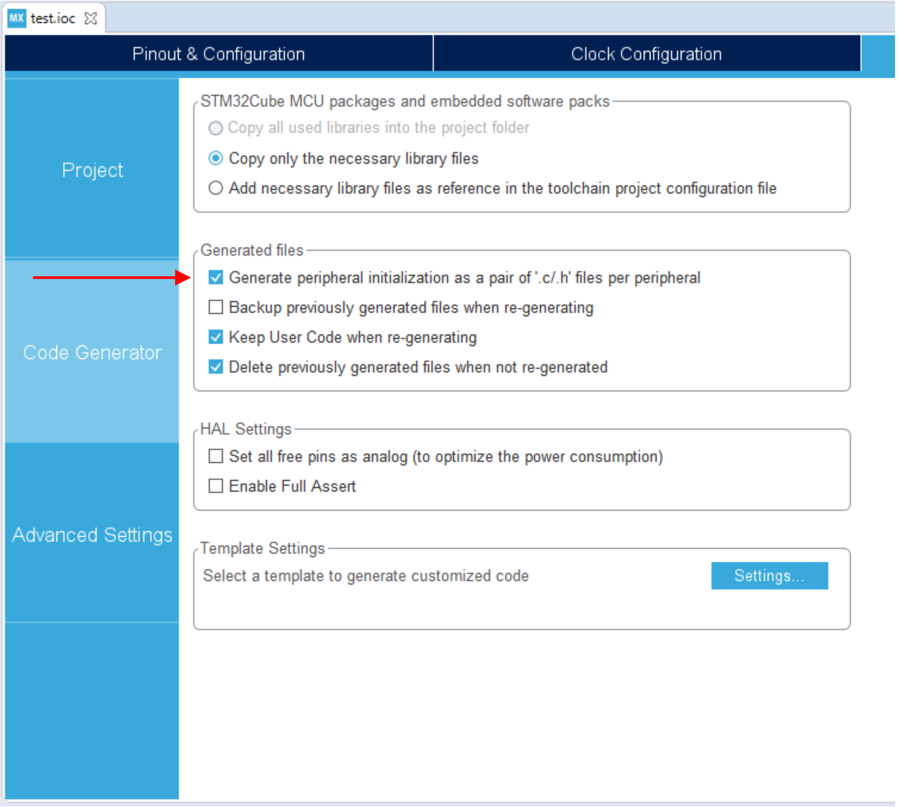
● STM32Cube ○ Empty

Project Manager

In the Code Generator Tab check the ***Generate peripheral initialization […]*** box: each periperhal will have a disting *periph.c* and *periph.h* files.



```
test
  Includes
  Core
    Inc
      gpio.h
      main.h
      stm32f4xx_hal_conf.h
      stm32f4xx_it.h
      usart.h
      usb_otg.h
    Src
      gpio.c
      main.c
      stm32f4xx_hal_msp.c
      stm32f4xx_it.c
      syscalls.c
      sysmem.c
      system_stm32f4xx.c
      usart.c
      usb_otg.c
    Startup
```



STM32Cube MCU packages and embedded software packs

○ Copy all used libraries into the project folder

● Copy only the necessary library files

○ Add necessary library files as reference in the toolchain project configuration file

Generated files

☑ Generate peripheral initialization as a pair of '.c/.h' files per peripheral

□ Backup previously generated files when re-generating

☑ Keep User Code when re-generating

☑ Delete previously generated files when not re-generated

HAL Settings

□ Set all free pins as analog (to optimize the power consumption)

□ Enable Full Assert

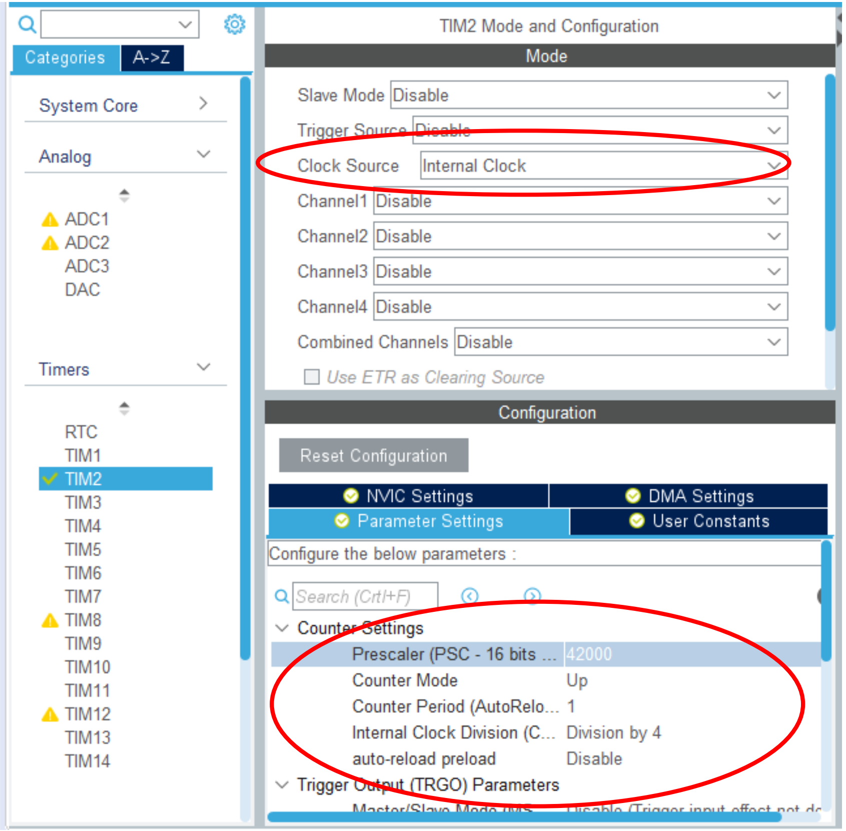
Template Settings

Select a template to generate customized code

Timer Parameters

In this example we will use TIM2, the parameters to be set are as follows:

- *Clock Source*: is the clock reference of the timer
- *Prescaler (PSC):* is a divider of the timer activation frequency
- *Counter Period:* timer activation period



TIM2 Mode and Configuration

Mode

Slave Mode Disable

Trigger Source Disable

Clock Source Internal Clock

Channel1 Disable

Channel2 Disable

Channel3 Disable

Channel4 Disable

Combined Channels Disable

□ Use ETR as Clearing Source

Configuration

Configure the below parameters :

Counter Settings

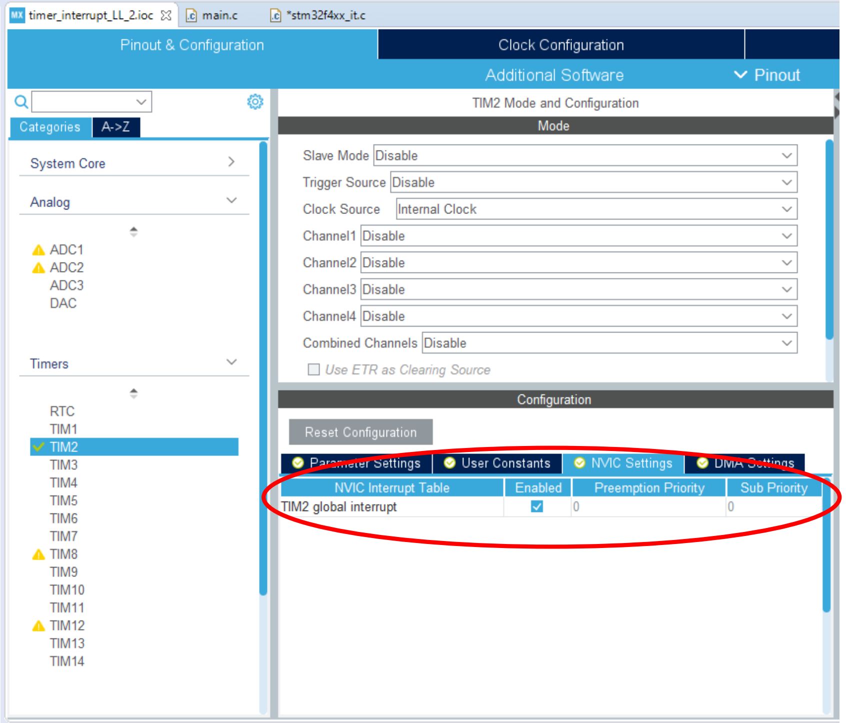
Prescaler (PSC - 16 bits ...	42000
Counter Mode	Up
Counter Period (AutoRelo...	1
Internal Clock Division (C...	Division by 4
auto-reload preload	Disable

Trigger Output (TRGO) Parameters

Master/Slave Mode (MS... Disable (Trigger input effect not d...

NVIC

- **NVIC** is fundamental for the generation of the interrupt to enable it and set its priority.
- Under the *NVIC tab* it is therefore necessary to tick the box for interrupt management

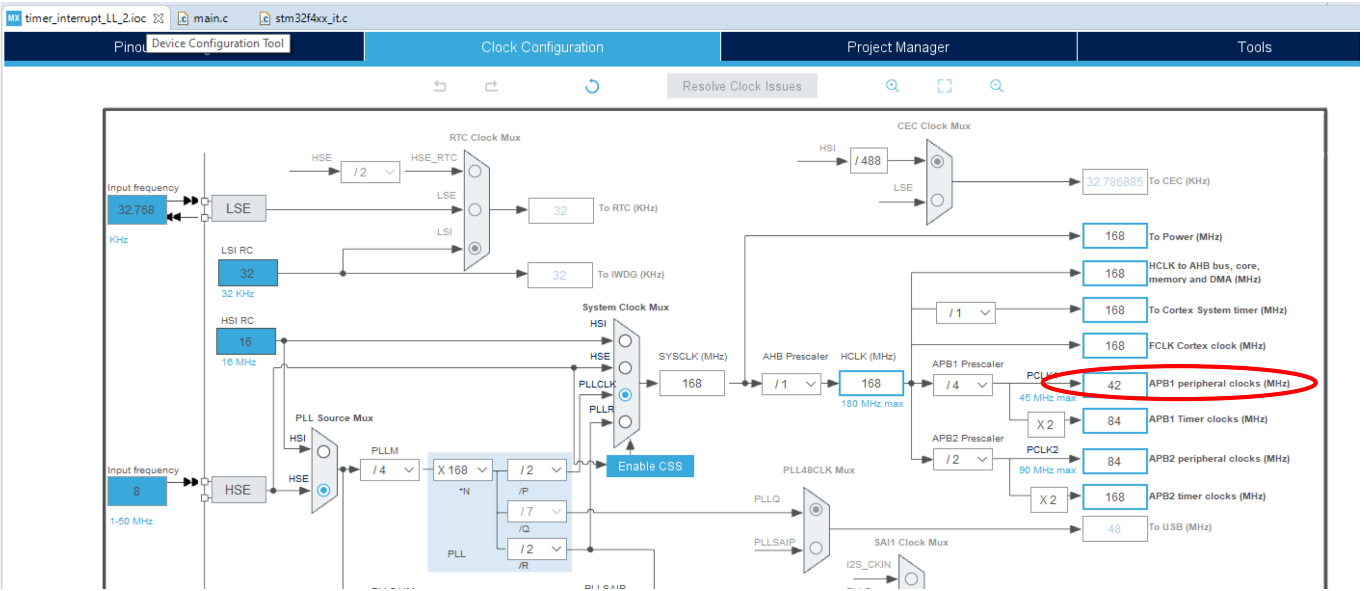


Slave Mode Disable
Trigger Source Disable
Clock Source Internal Clock
Channel1 Disable
Channel2 Disable
Channel3 Disable
Channel4 Disable
Combined Channels Disable
☐ Use ETR as Clearing Source

NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
TIM2 global interrupt	☑	0	0

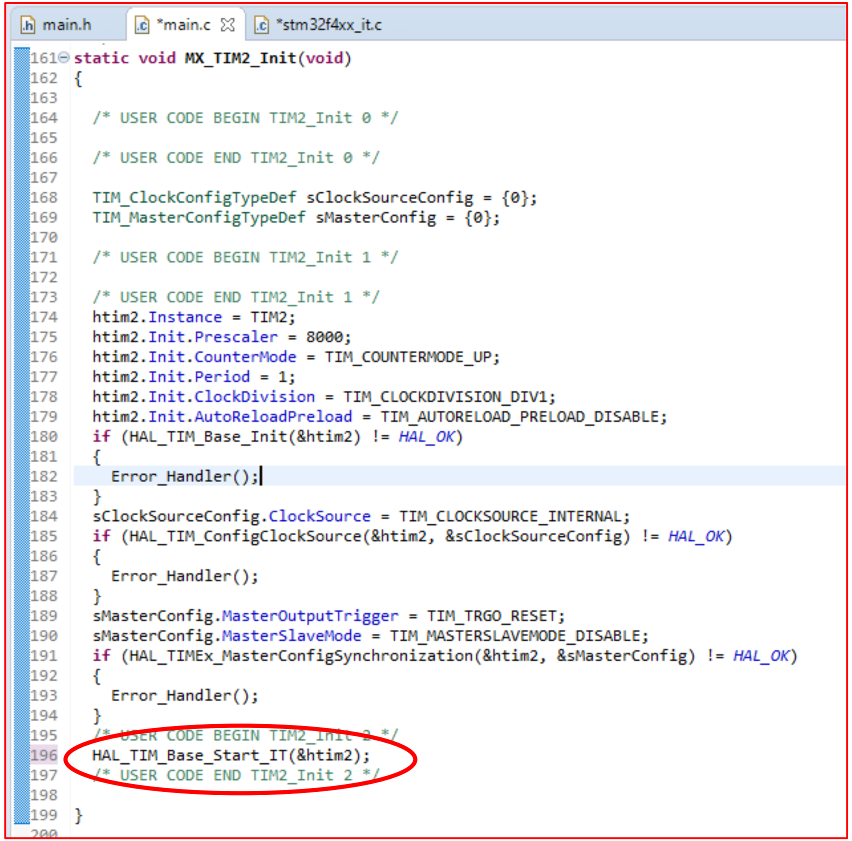
Prescaler

In our example the **Prescaler** has been set to *42000* as the **APB1** peripheral clock, to which **TIM2** also refers, is *42MHz*: in this way, having the Counter Period set to 1, the timer will be called every millisecond (42MHz / 42000 = 1000). At this point we just have to generate the code



Timer initialization

Before managing the interrupt, it must be enabled: for this reason, in the initialization of the timer it is necessary to use the ***HAL_TIM_Base_Start_IT()*** function. Otherwise no interrupt will be generated.

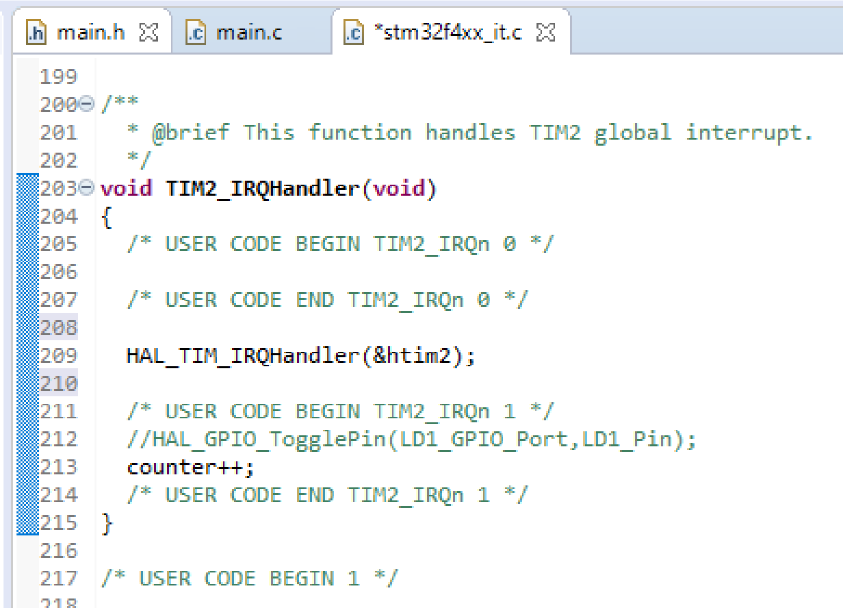


```c
static void MX_TIM2_Init(void)
{

  /* USER CODE BEGIN TIM2_Init 0 */

  /* USER CODE END TIM2_Init 0 */

  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};

  /* USER CODE BEGIN TIM2_Init 1 */

  /* USER CODE END TIM2_Init 1 */
  htim2.Instance = TIM2;
  htim2.Init.Prescaler = 8000;
  htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim2.Init.Period = 1;
  htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
  {
    Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM2_Init 2 */
  HAL_TIM_Base_Start_IT(&htim2);
  /* USER CODE END TIM2_Init 2 */

}
```

Handler

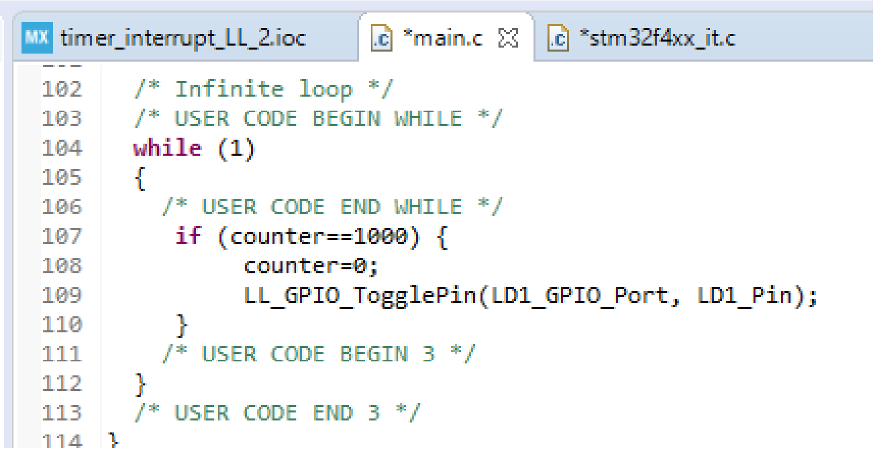
- The first step is to check what happens as soon as the interrupt is generated: we then go to the Handler management within the ***stm32f4xx_it.c*** file.

- Just as in the previous cases, *CubeMX* manages the handler, it is necessary to add the Callback function of the timer or if necessary, you can write the management of the interrupt within the Handler, as in this case.



```

/**
  * @brief This function handles TIM2 global interrupt.
  */
void TIM2_IRQHandler(void)
{
  /* USER CODE BEGIN TIM2_IRQn 0 */

  /* USER CODE END TIM2_IRQn 0 */

  HAL_TIM_IRQHandler(&htim2);

  /* USER CODE BEGIN TIM2_IRQn 1 */
  //HAL_GPIO_TogglePin(LD1_GPIO_Port,LD1_Pin);
  counter++;
  /* USER CODE END TIM2_IRQn 1 */
}

/* USER CODE BEGIN 1 */
```

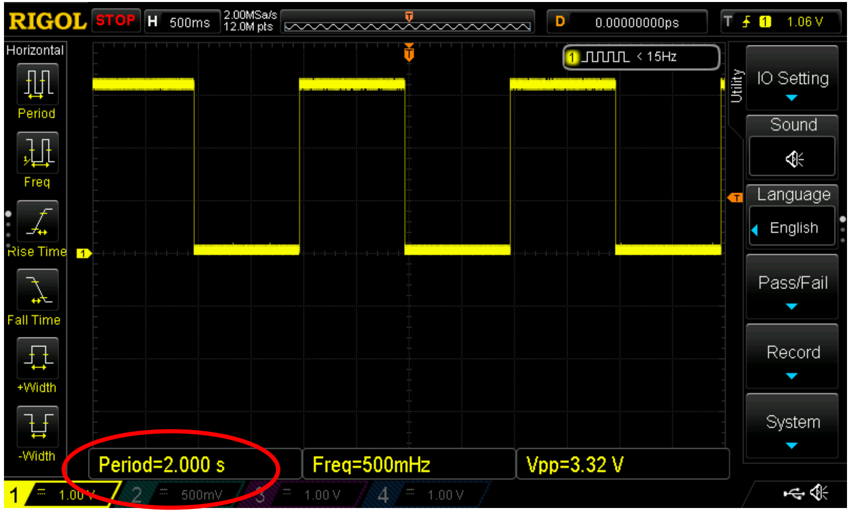
Using the counter variable, it is possible to "count" how many interrupts have been generated: remember that we have set the timer parameters in order to generate an interrupt every millisecond. Therefore only 1 second will pass after 1000 interrupts.



```
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */
     if (counter==1000) {
          counter=0;
          LL_GPIO_TogglePin(LD1_GPIO_Port, LD1_Pin);
     }
    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}
```

Results

We load the program on the board and check that everything is working properly: Through the use of the oscilloscope we can note that exactly every second the LED changes its state, so we have a period of 2 seconds as we expected.



Period=2.000 s

Freq=500mHz

Vpp=3.32 V