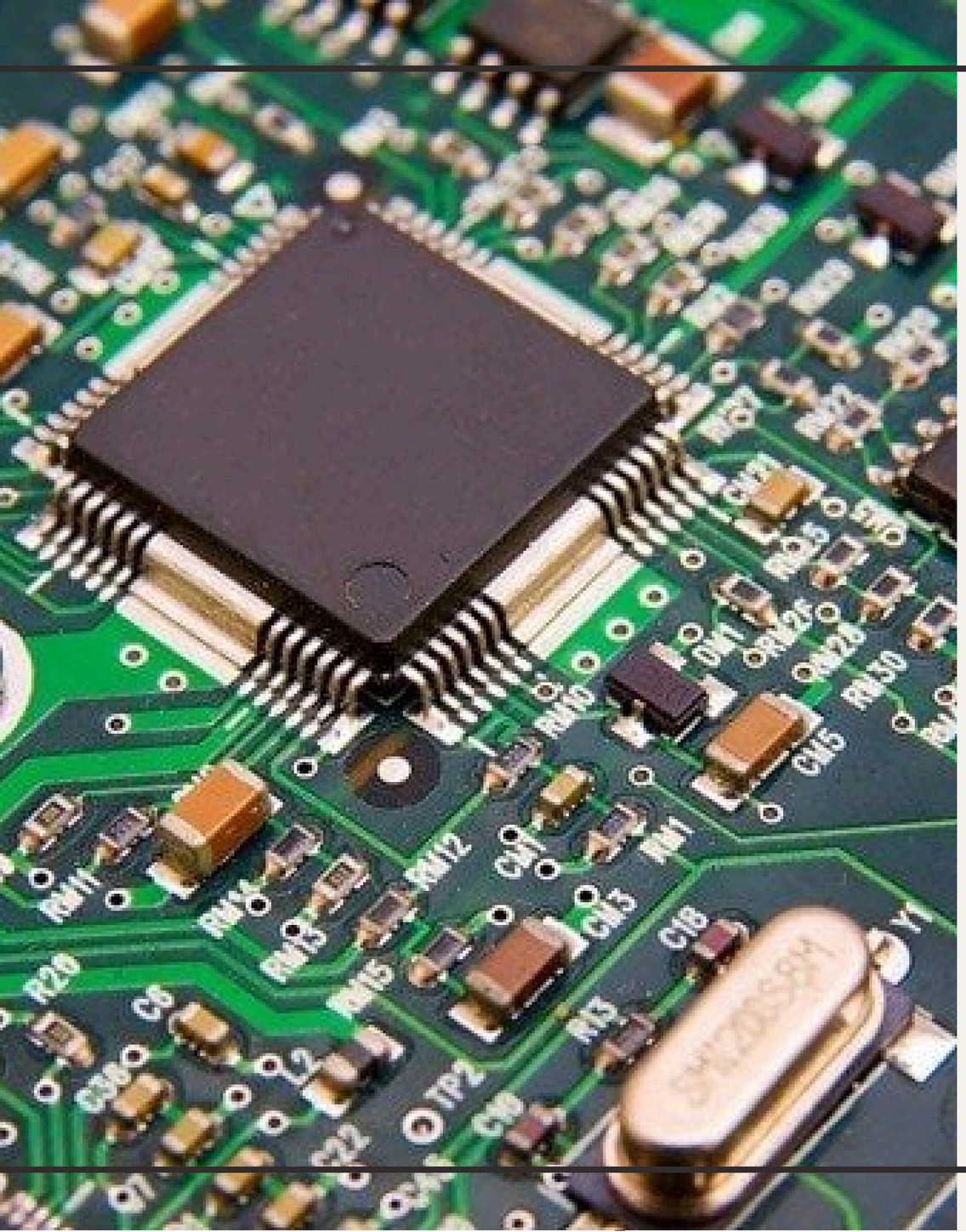
STM32 Microcontrollers Introduction and Guide

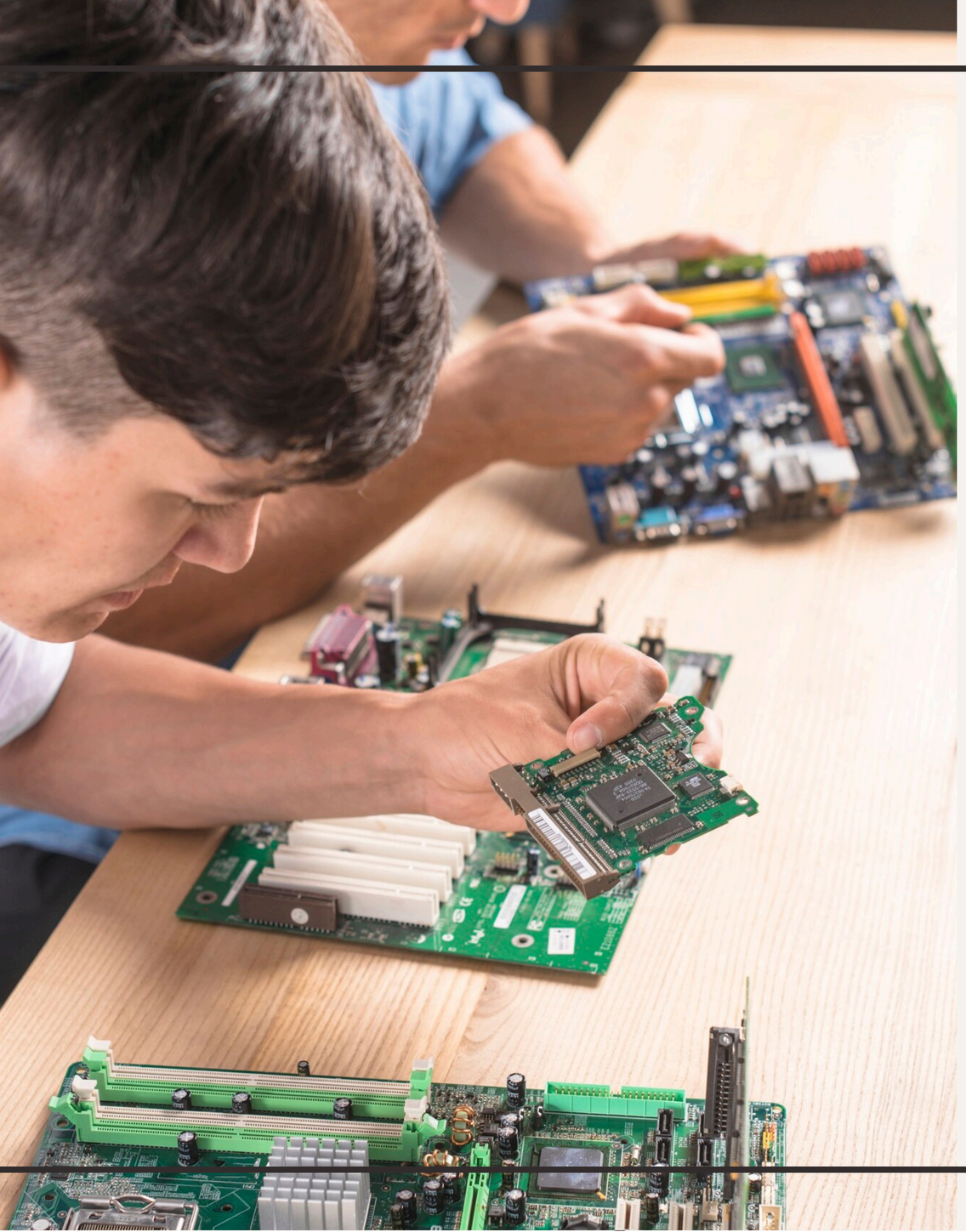
M Andi Abdillah | Barunastra ITSN

List of contents



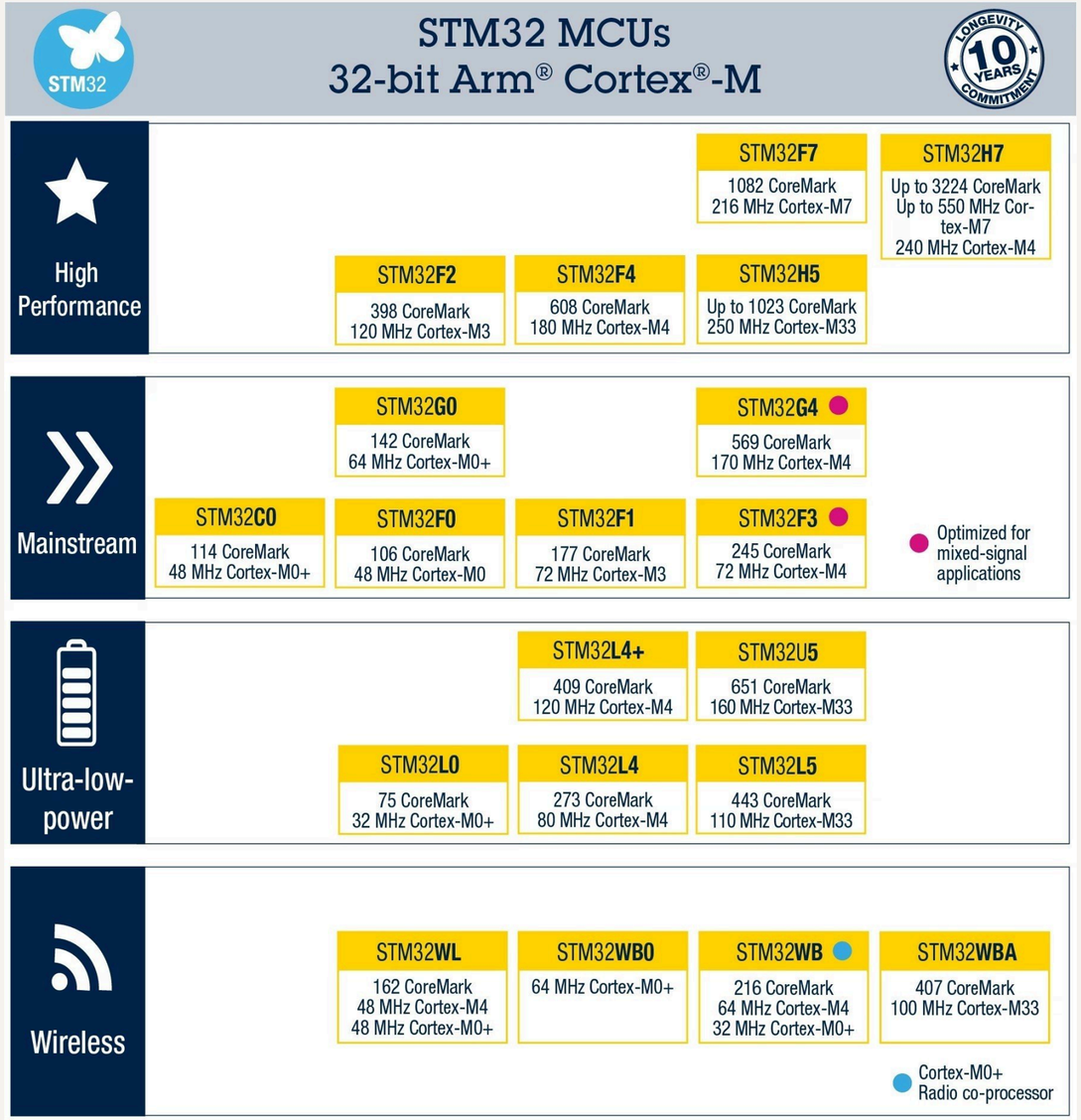
What is a Microcontroller?

A microcontroller unit(MCU) is a small computer on a single integrated circuit. A microcontroller contains one or more CPUs (processor cores) along with memory and programmable input/output peripherals. Program memory in the form of ferroelectric RAM, NOR flash or OTP ROM is also often included on chip, as well as a small amount of RAM. Currently, there are many chip companies that produce microcontrollers such as Atmel, Espressif, Microchip, Raspberry, STMicroelectronics, Texas Instrument, and more.



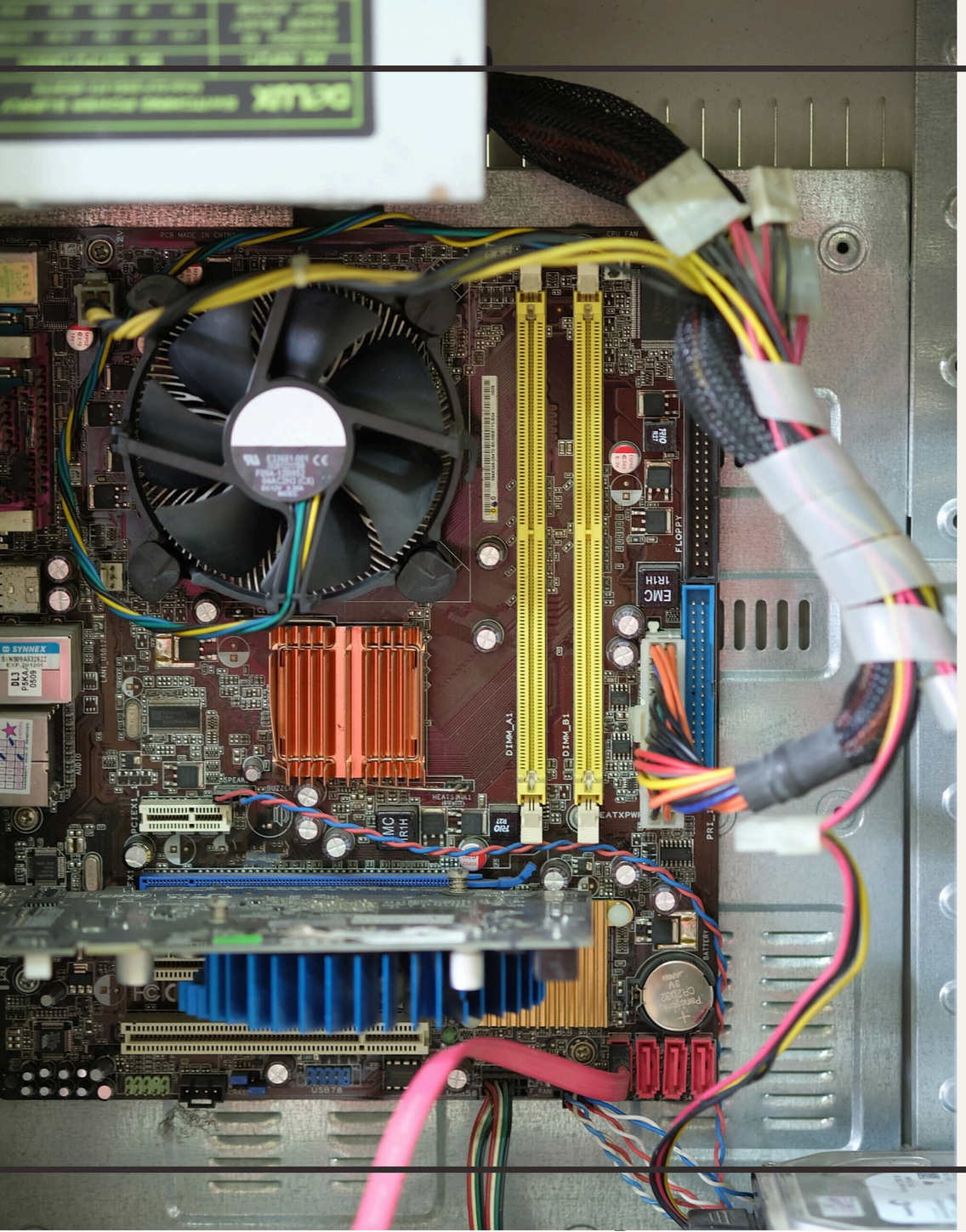
When is a Microcontrollers used

Microcontrollers are used in automatically controlled products and devices, such as automobile engine control systems, remote controls, office machines, power tools, toys and other embedded systems.



Why we use STM32?

STM32 is a family of 32-bit microcontrollers based on the Arm Cortex-M processor is designed to offer new degrees of freedom to MCU users. It offers products combining very high performance, real-time capabilities, digital signal processing, low-power / low-voltage operation, and connectivity, while maintaining full integration and ease of development.

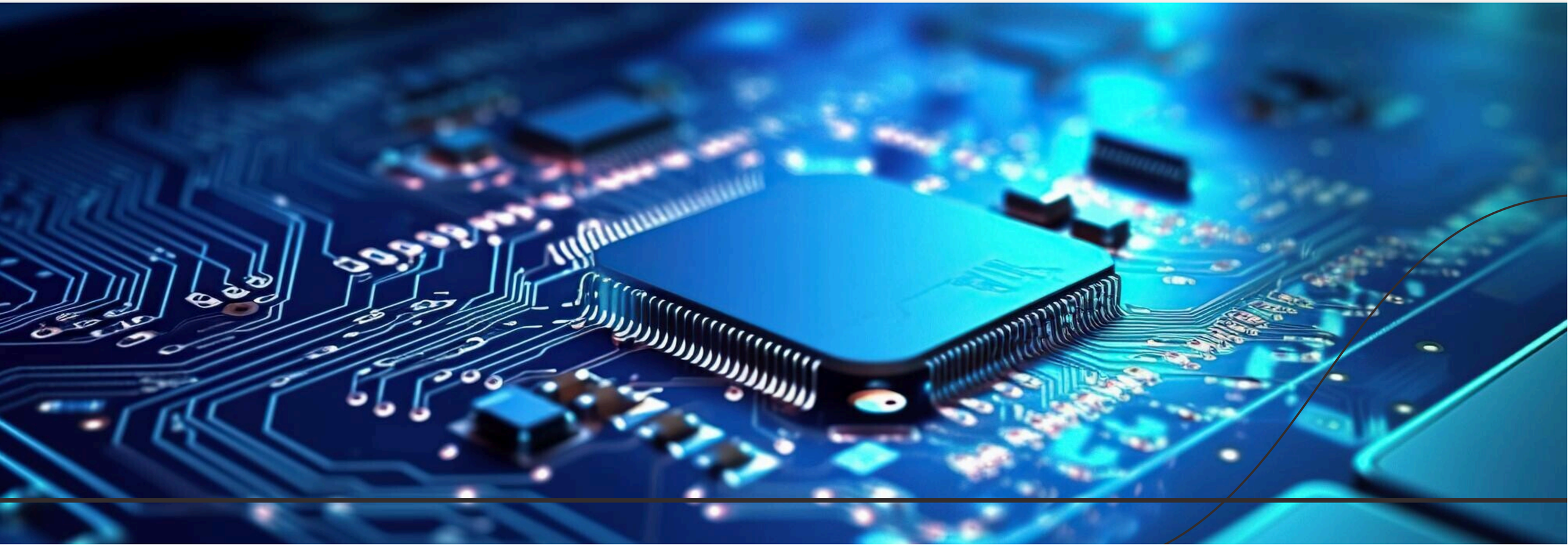


Peripheral Interfacing

- GPIO : General Purpose Input/Output
- ADC/DAC : Analog/Digital Converter
- USART/UART :Universal (Synchronous) Asynchronous Receiver-Transmitter
- CAN bus : Control Area Network bus
- I2C : Inter Integrated Circuit
- SPI : Serial Peripheral Interface
- USB : Universal Serial Bus
- Ethernet

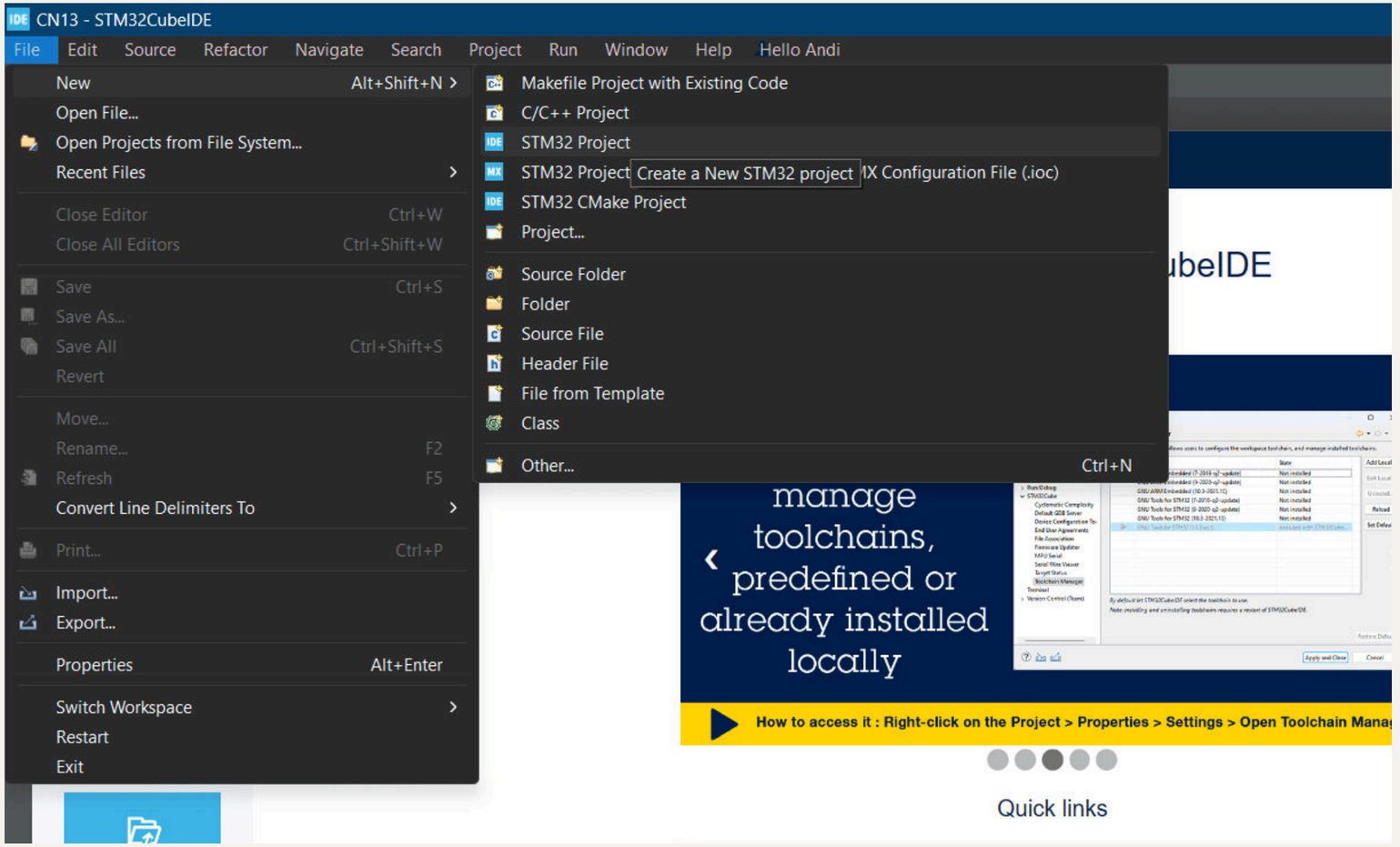
What You Need

- STM32CubeIDE
- C programming language



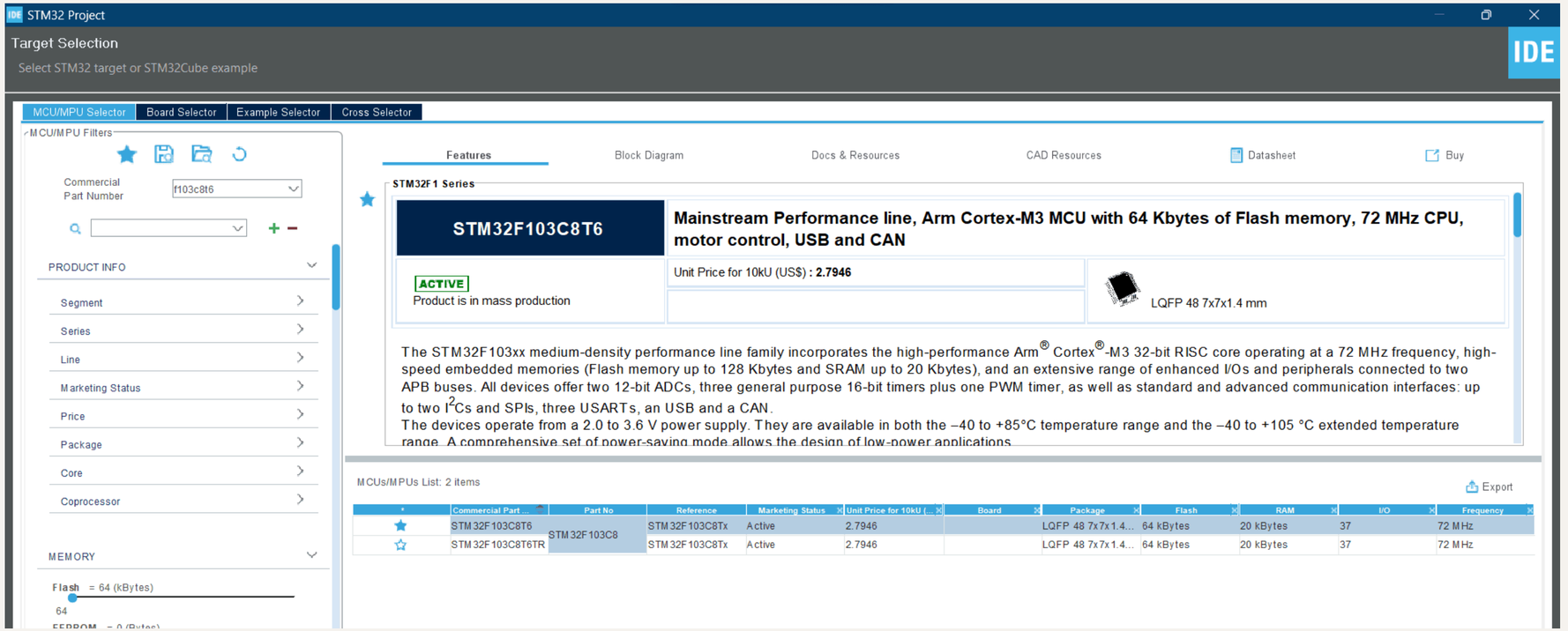
1. Create a New Project, debugging setup, and set RCC(Reset and Clock Control

select your workspace directory, go to File->new->STM32 project

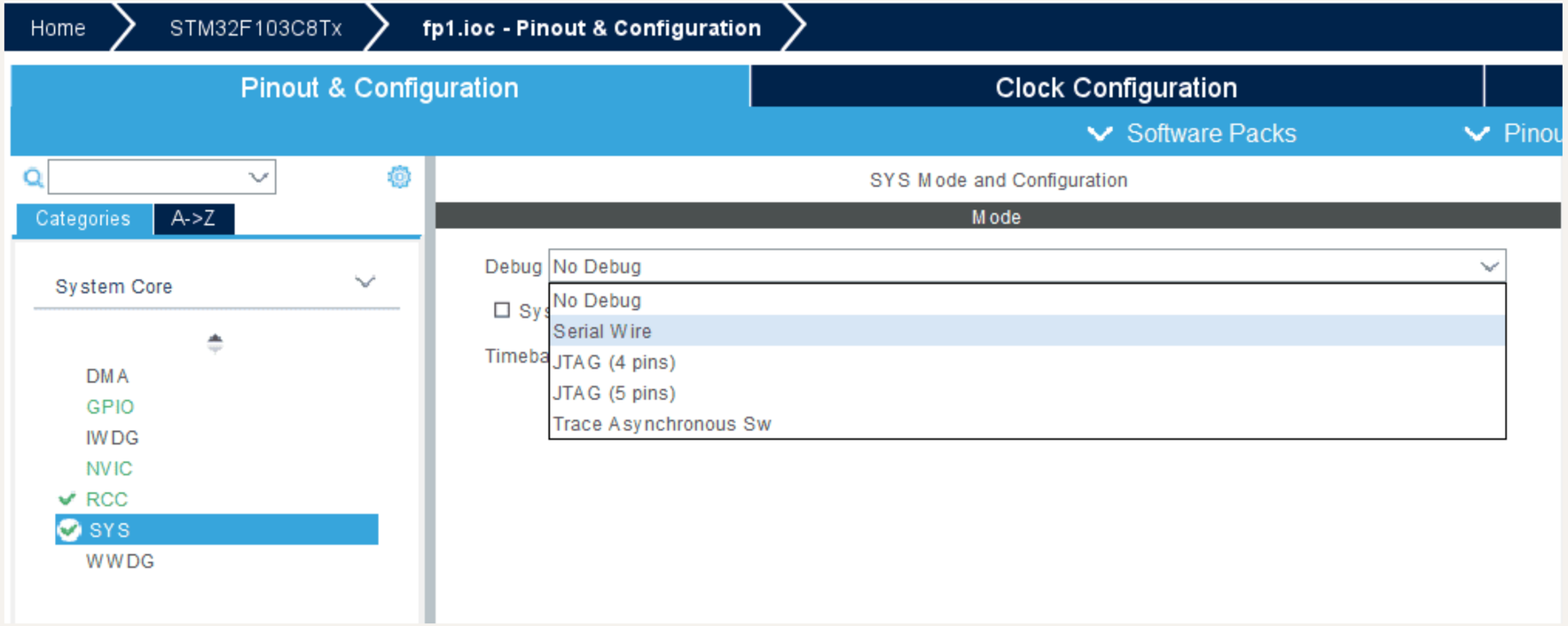


In this tutorial, we will use the STM32F103C8T6 or commonly know as Blue Pill which is cheap and easy to find in stores. You can find it in MCU/MPU selector. For nucleo or discovery board i suggest you to find it in board selector as it has the default configuration for that board.

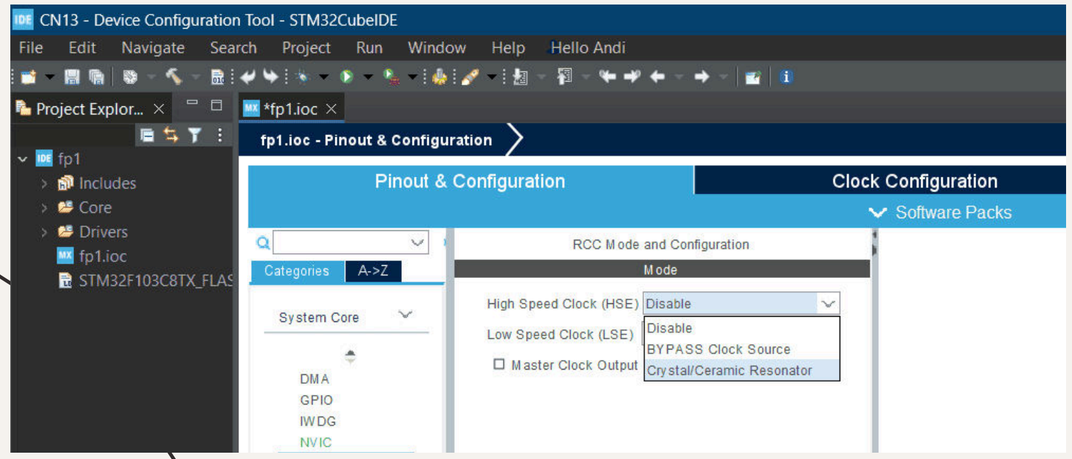
click next, give name to your project, leave everything else as it is and finish. you will be redirected to IOC (Input Output Configuration) setup.

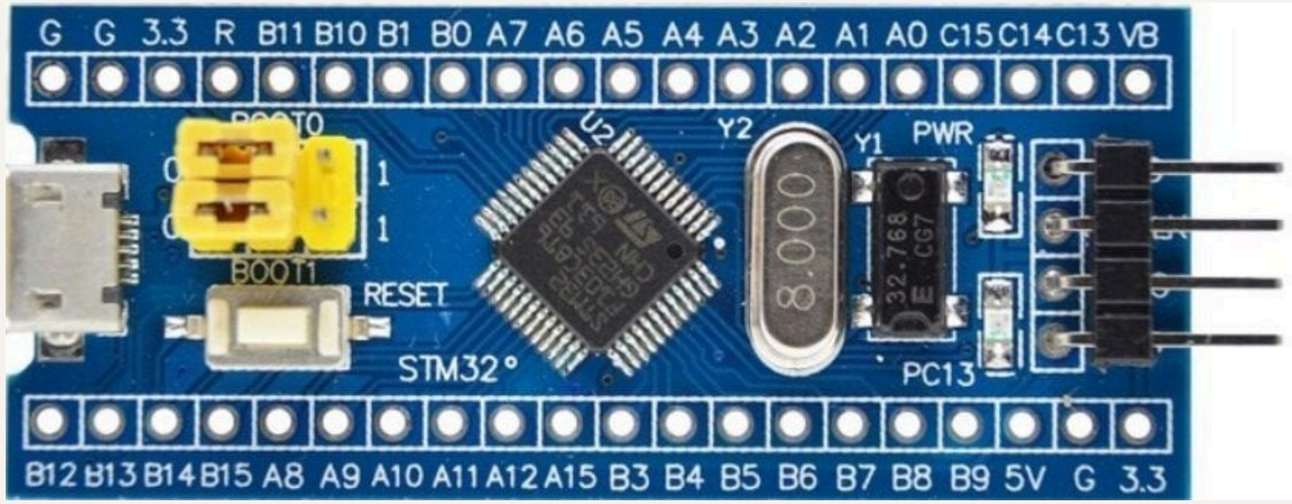


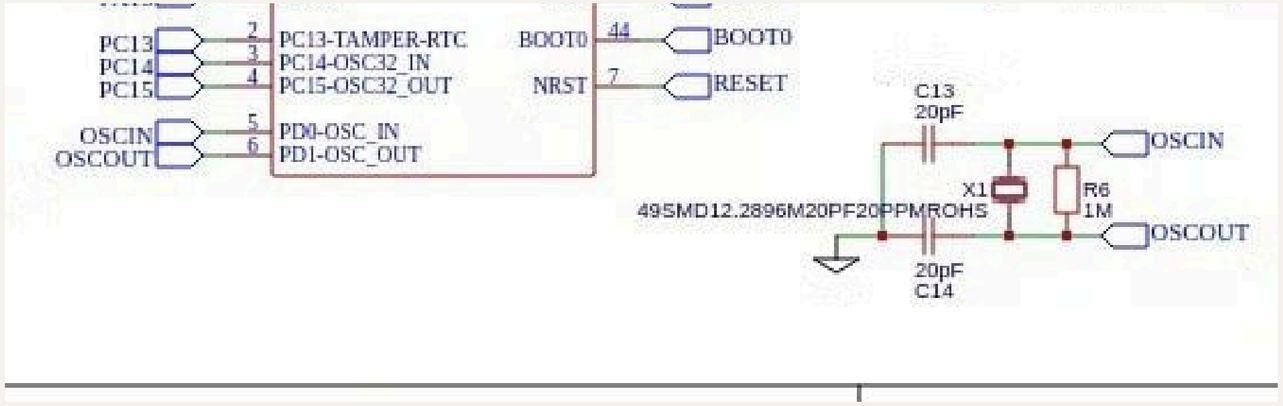
To program and debug the STM32, we must enable debug communication. Typically, stm32 use ST-LINK to program and debug it, which uses a SWD (Serial Wire Debugging) type connection. to enable it, go to System Core->SYS->Debug and select Serial Wire.



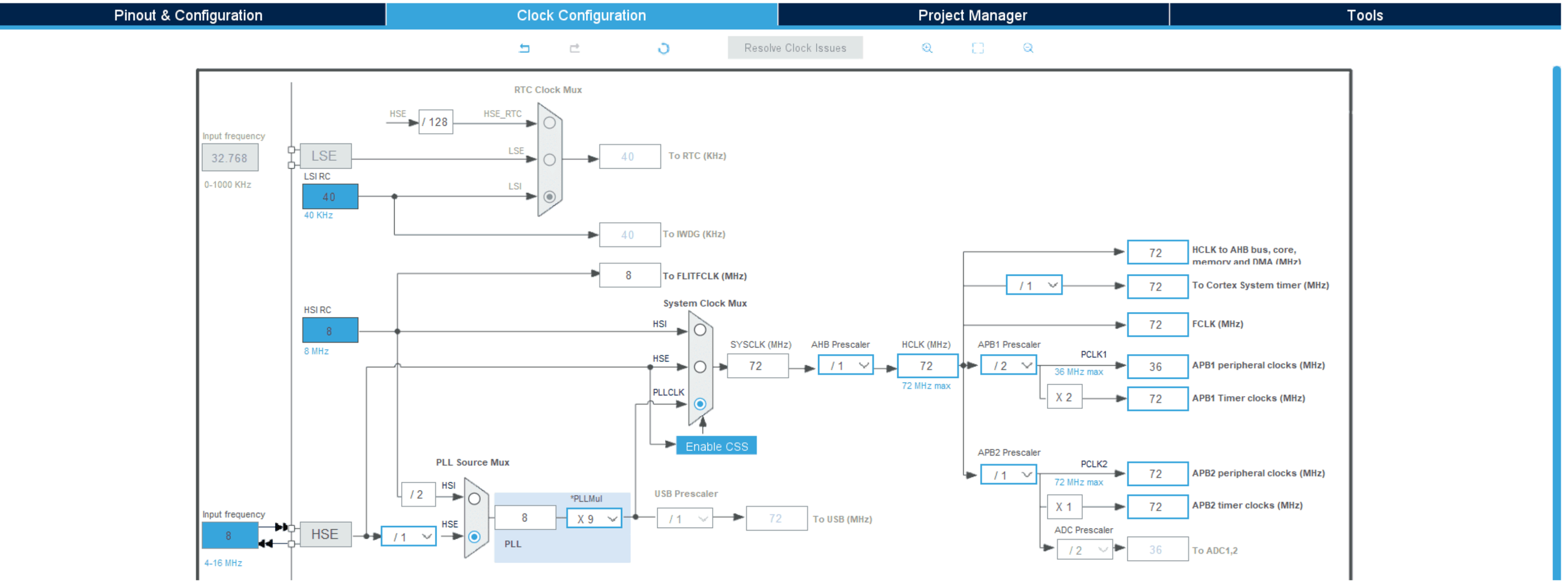
We will use an external crystal oscillator which is more precise than internal oscillator. to use it, go to System Core->RCC->High Speed Oscillators, select Crystal/Ceramic Oscillator. HSE/LSE configuration is depends with your board. In this case, the Blue Pill only have one ceramic oscillators which is connected at the HSE.





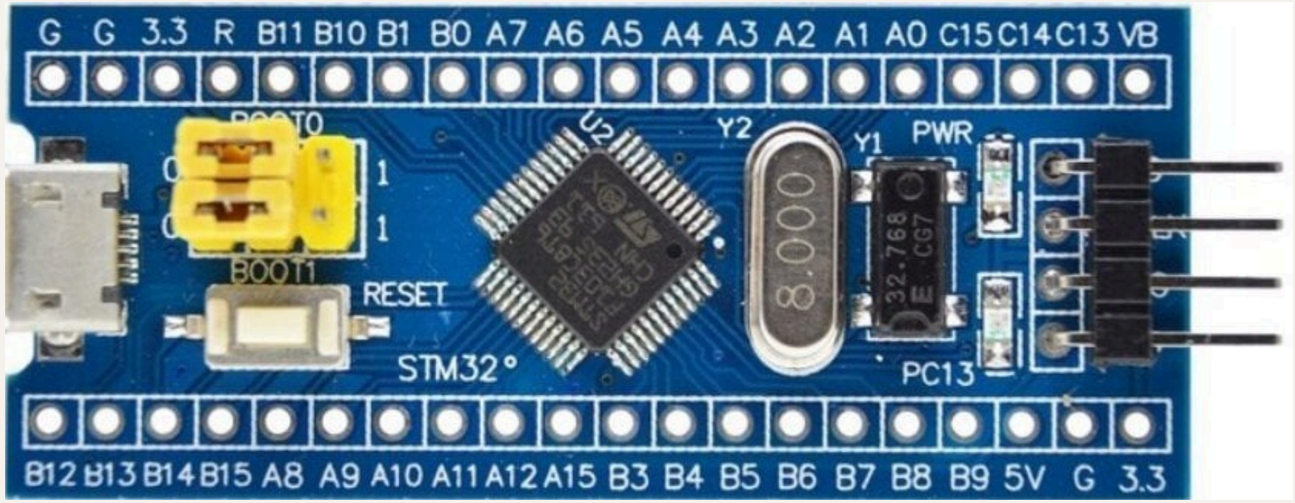


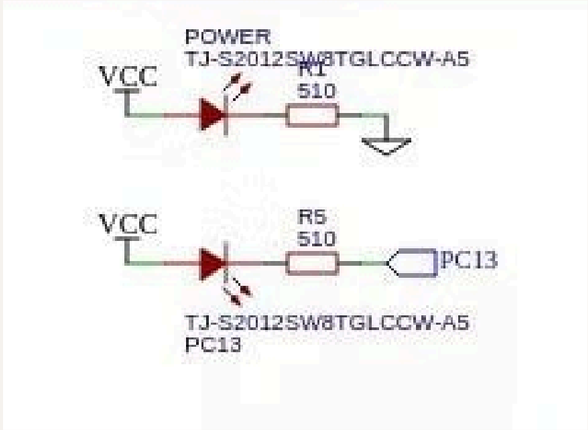
We will set the hardware and peripheral clocks to maximum clock speed. Switch tab to clock configuration, set the input frequency of the HSE to 8 because the crystal ceramic resonator that soldered in the blue pill is 8Mhz. Switch PLL (Phase Lock Loop) source mux to HSE and System clock mux to PLLCLK. Find the correct numbers for PLLMux and other prescaler to get the maximum clock speed and you're done setting RCC.



2. Digital Input And Output

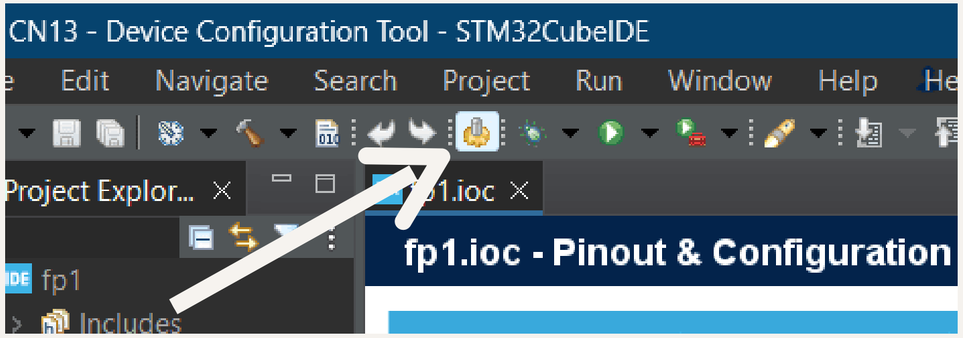
Bluepill have a user LED connected to pin PC13, we will try to turn it on

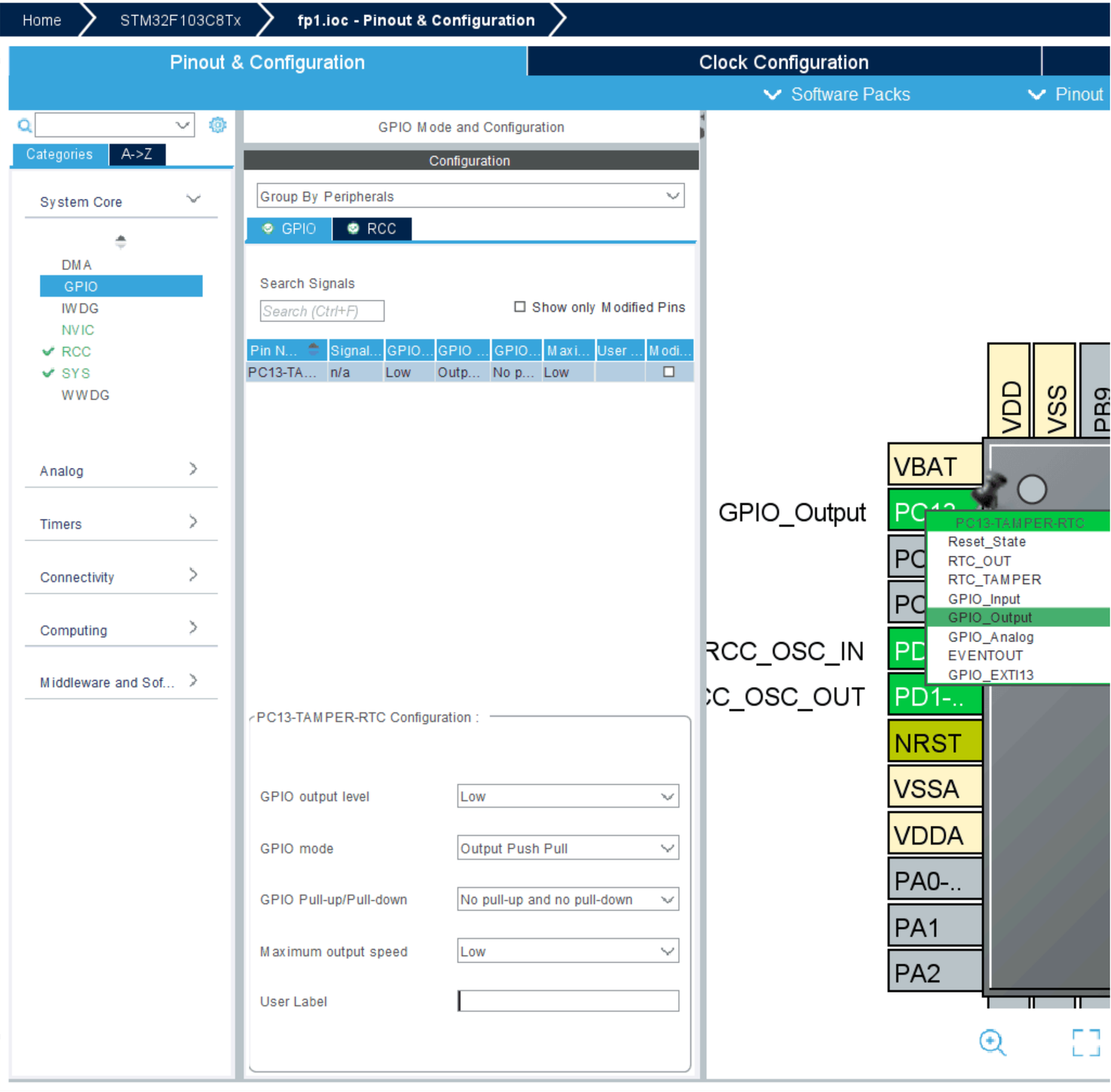




Go back to Pinout & Configuration tab, click on the PC13 pin of the MCU and select GPIO Output. you can use the same method to configure other pin. you can also label your GPIO pin from System Core-> GPIO

Generate code or save and notification to generate code will be displayed



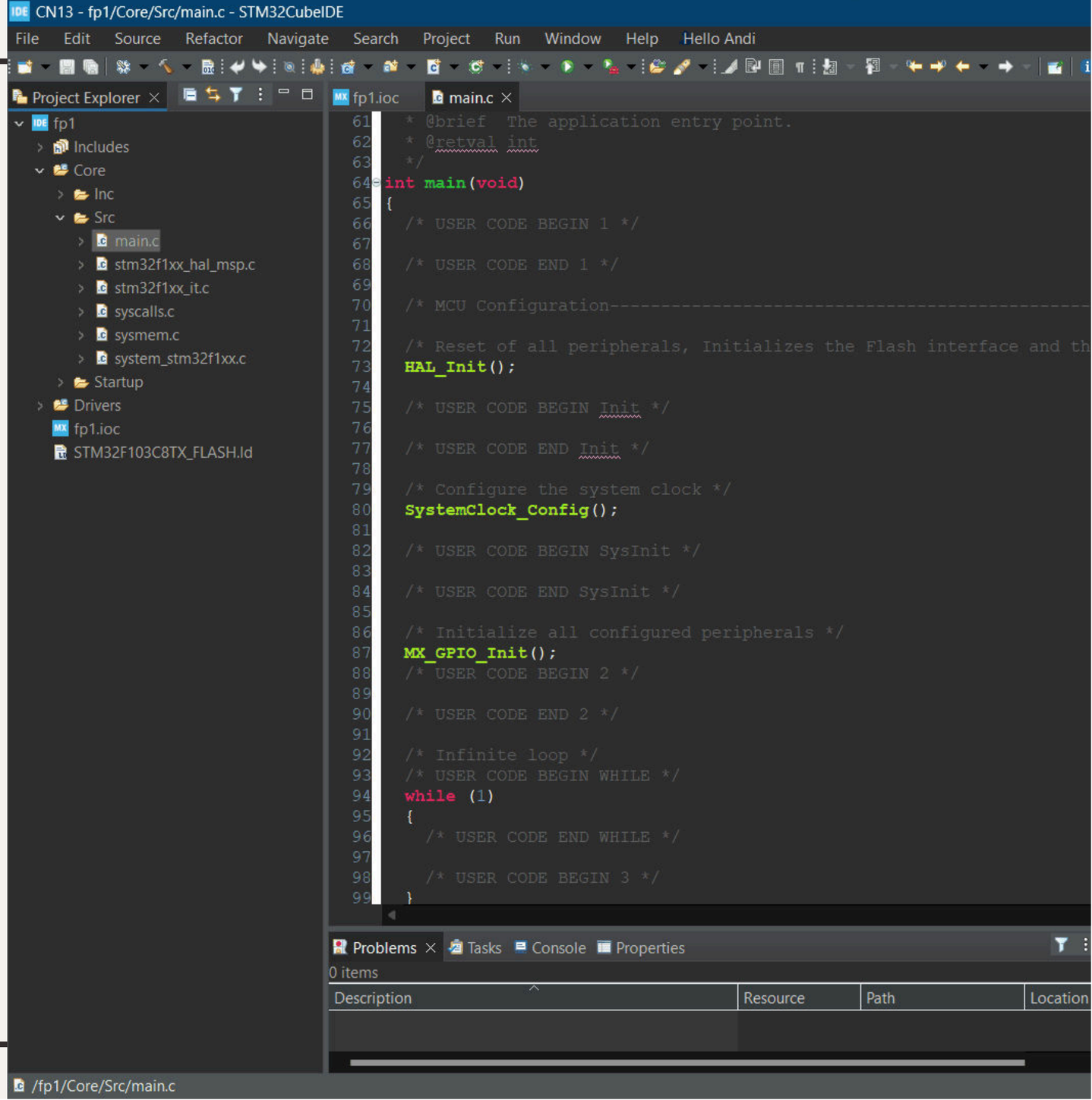


The source code is located in your STM32 Project name->Core->Src

The header code is located in your STM32 Project name->Core->Inc

You can add your own library in that directory.

make sure that if you modify generated code, please write it between user code begin comment and user code end comment.



```
HAL_GPIO_WritePin(GPIOx, GPIO_Pin, PinState);
HAL_GPIO_TogglePin(GPIOx, GPIO_Pin);
HAL_GPIO_ReadPin(GPIOx, GPIO_Pin);
```

Basic syntax for digital output is write pin and toggle pin. while for digital pin is read pin.

```
while (1)
{
    HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_SET); //1 //HIGH
    HAL_Delay(1000); //1000ms delay
    HAL_GPIO_WritePin(LED_GPIO_Port, LED_Pin, GPIO_PIN_RESET); //0 //LOW
    HAL_Delay(1000); //1000ms delay
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */
}
```

This is an example code for blinking LED. The first line is to make the PC13 pin output High, second line is for delay, and third line is to make "LED" label pin output Low.

```
while (1)
{
    uint8_t btn;
    btn = HAL_GPIO_ReadPin(BTN_GPIO_Port, BTN_Pin);
    HAL_GPIO_WritePin(LED_GPIO_PORT, LED_Pin, btn);
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */
}
```

This is an example code to turn on LED while button pressed. Always pull the GPIO input to prevent floating input. You can pull it internally from GPIO IOC pin configuration

3. Timer

A timer (sometimes referred to as a counter) is a special piece of hardware inside many microcontrollers. Their function is simple: they count (up or down, depending on the configuration, we'll assume up for now).

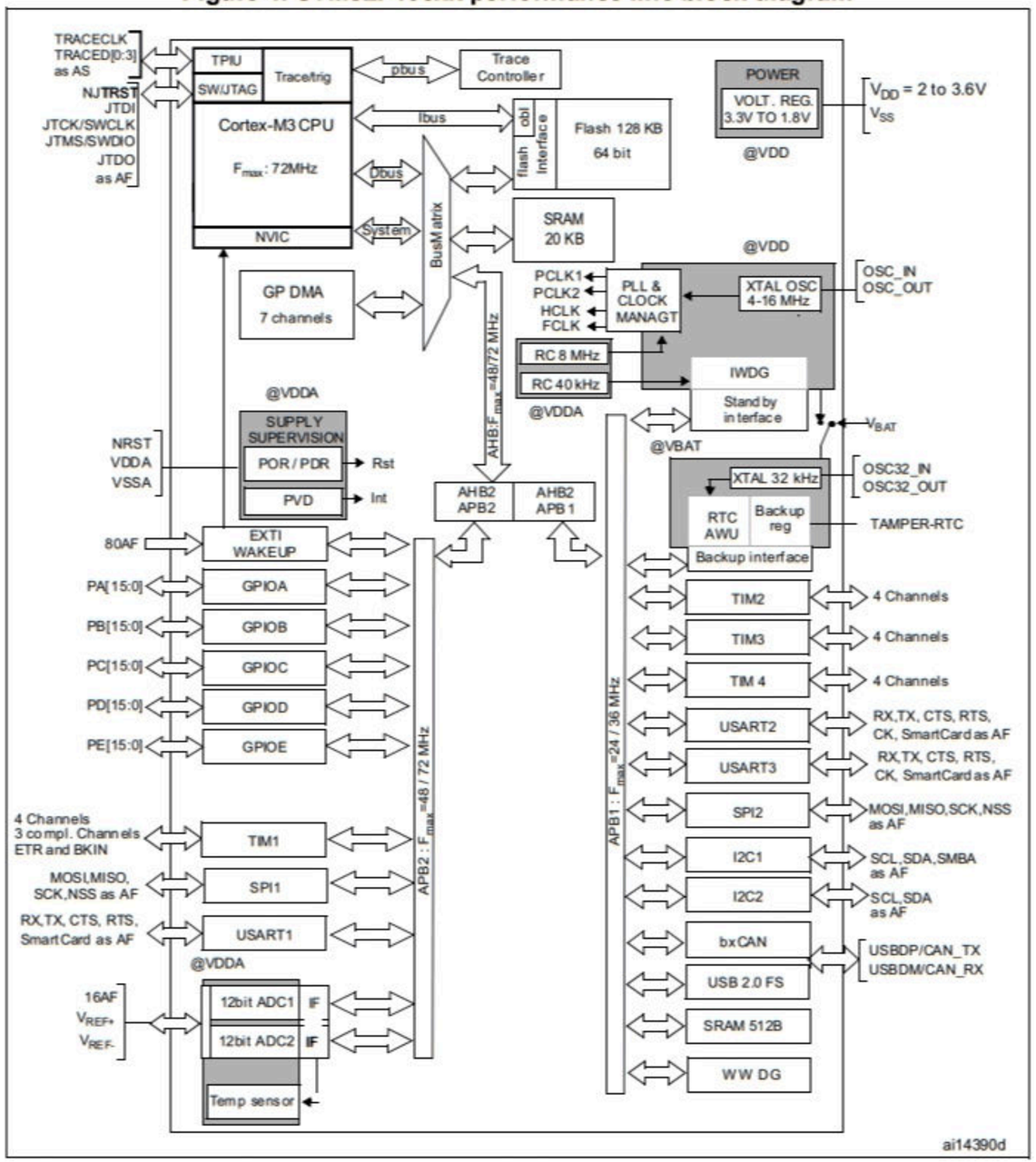
Here are some of the common hardware functions you'll see with timers:

- Output compare (OC): toggle a pin when a timer reaches a certain value.
- Input capture (IC): measure the number of counts of a timer between events on a pin.
- Pulse width modulation (PWM): toggle a pin when a timer reaches a certain value and on rollover. By adjusting the on versus off time (duty cycle), you can effectively control the amount of electrical power going to another device.

a. Basic Timer Interrupt

You should look the datasheet for a performance line block diagram that gives you details of where is the timers connect. as an example, we use TIM2 which is connected to APB1 clock. you can look APB2 clock from your clock configuration.

Figure 1. STM32F103xx performance line block diagram

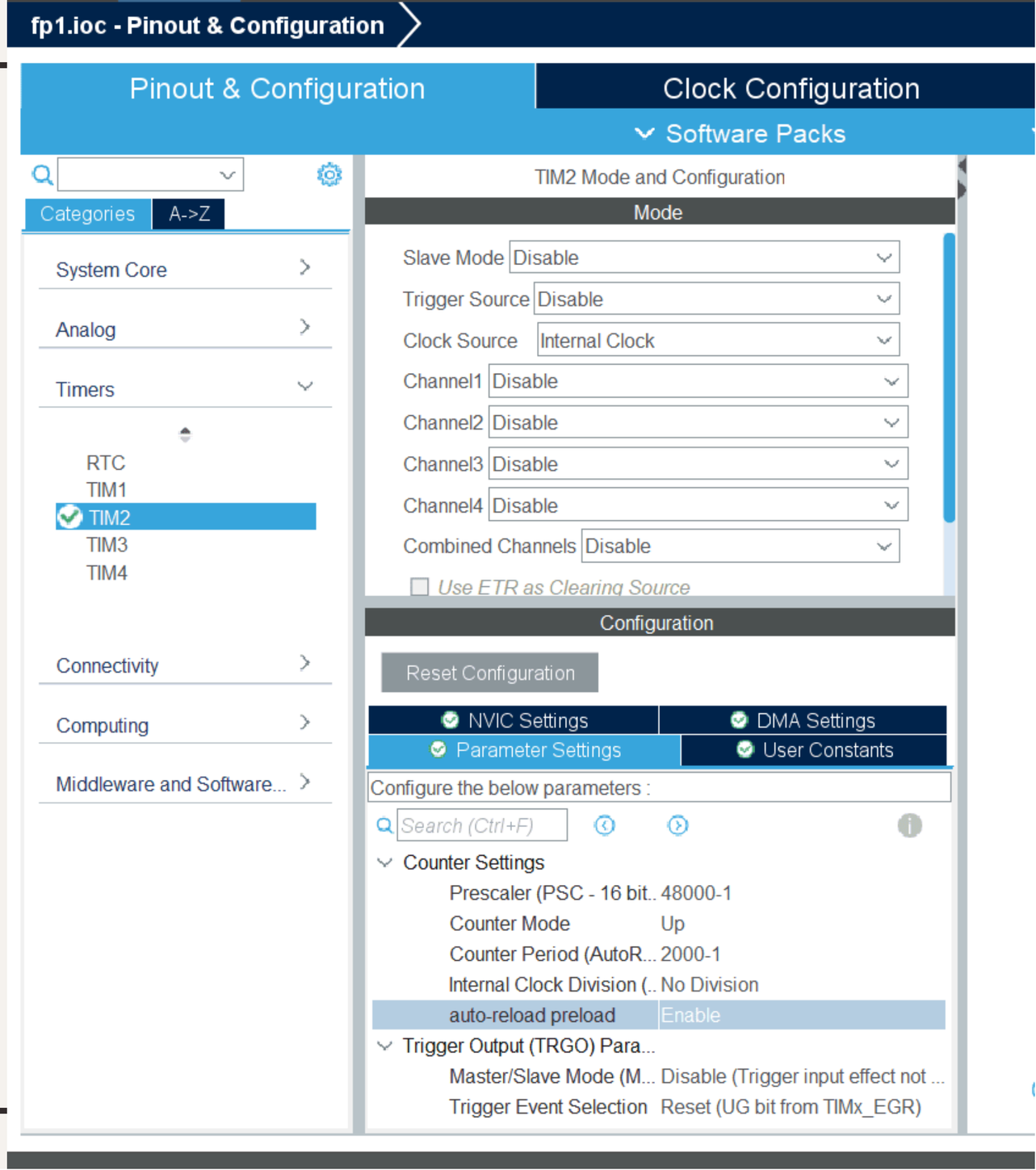


1. T_A = –40°C to +105°C (junction temperature up to 125°C).
2. AF = alternate function on I/O port pin.

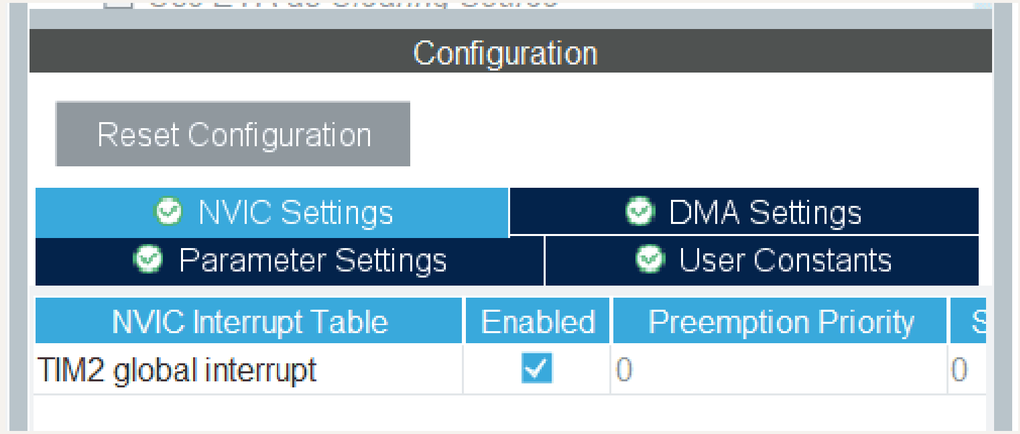
Now select TIM2 from Timers and switch clock source to internal clock. TIM2 is connected to APB1 which has a frequency of 72Mhz. The goal is to make 1 seconds periodic timer. We have prescaler and counter period here. This is very important to understand.

$$72.10^6 / 36.10^3 / 2.10^3 = 1Hz$$

After the timer is divided by prescaler, the frequency becomes 2Khz or 0.5ms per tick. if we have a counter period or counter limit of 2000 , we will have a time of 1 seconds.
We have to decrease the prescaler and counter period values by 1 because the counter starts from zero. Set counter mode to up and enable auto reload preload.



Go to NVIC (Nested Vectored Interrupt Controller) Setting tab and enable TIM2 Global Interrupt. Generate code and go to main source code.




```
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_TIM2_Init();
  /* USER CODE BEGIN 2 */
  HAL_TIM_Base_Start_IT(&htim2);
  /* USER CODE END 2 */
```

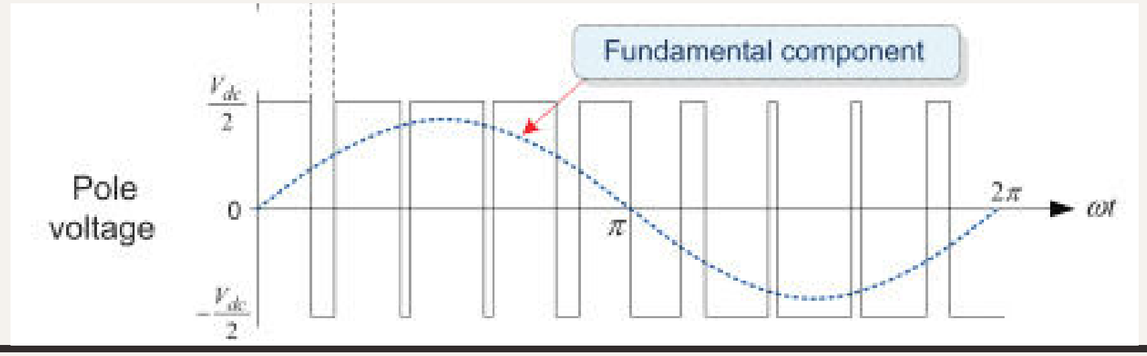
To start the timer as interrupt mode, you can write it after all initial peripheral (user code 2).

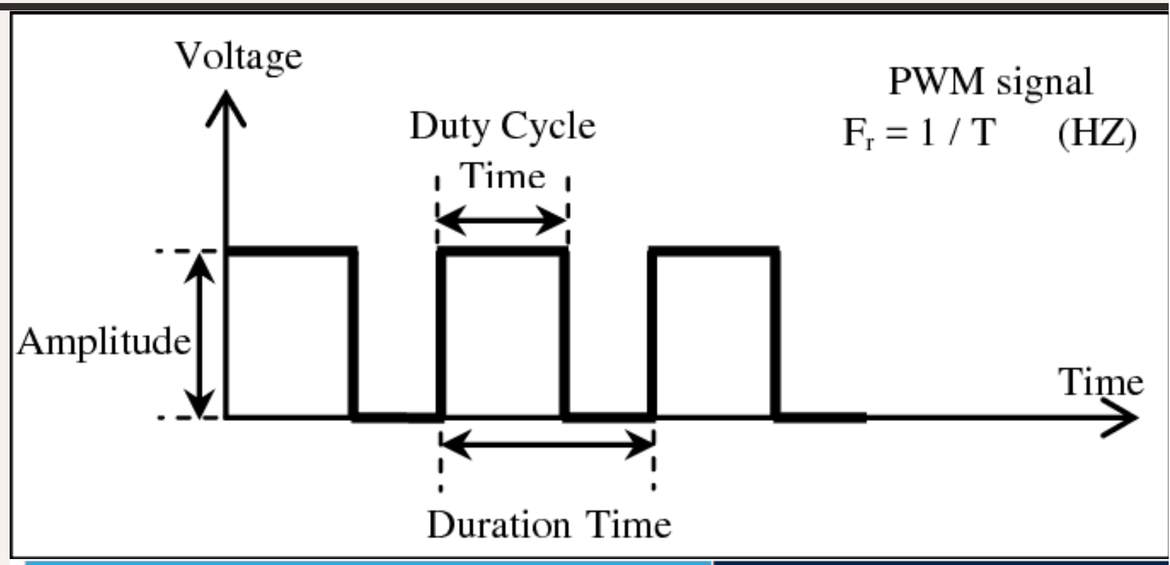
```
/* USER CODE BEGIN 4 */

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
    if(htim->Instance == TIM2){
        HAL_GPIO_TogglePin(LED_GPIO_Port, LED_Pin);
    }
}

/* USER CODE END 4 */
```

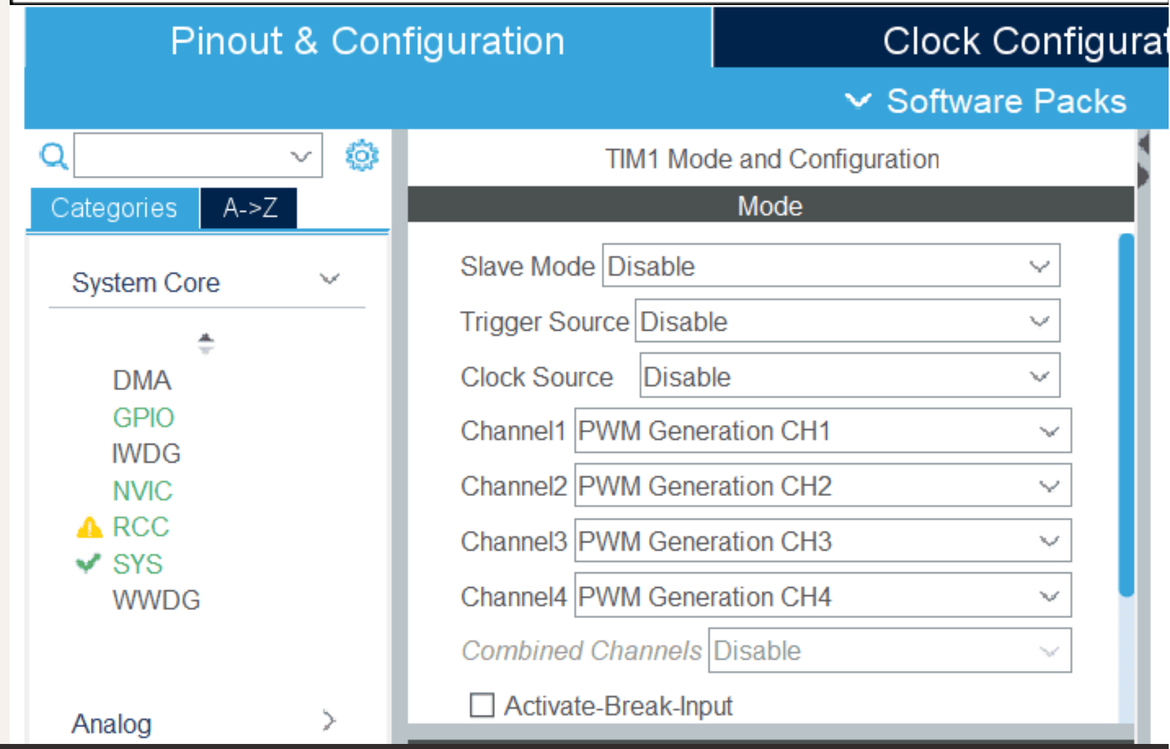
create this function that will be called when period is over. you can find the actual syntax in inc, stm32f1xx_hal_tim.h and you can blinking LED with non blocking code.

b. Pulse Width Modulation

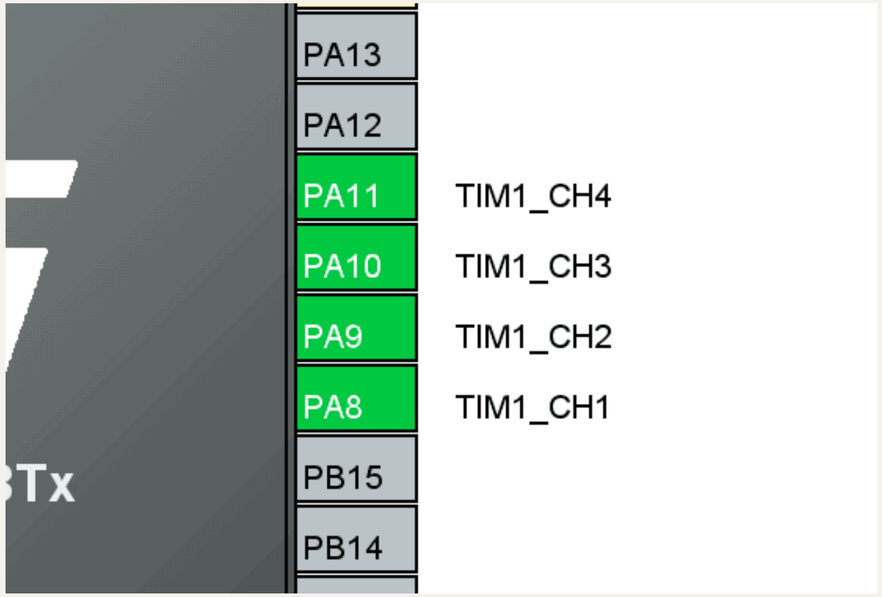
Pulse-width modulation (PWM) is a powerful technique for controlling analog circuits with a microcontroller's digital outputs. PWM is used in many applications, ranging from communications to power control and conversion. in STM32, you can have up to 4 channel per timer. each channel can control own pulse but the prescaler and the counter period is same for one timer. pwm is the same as the output compare, but pwm is periodic.







Set prescaler to 72 to get 1ms per tick and i made counter period to 256 because this is the maximum number for 1 byte. We want to control the brightness of the LED. Make each channel to produce 0%, 25%, 50%, and 100% of duty cycle. you can see some pins are automatically configured for each channel. You can connect it with LED.



Configure the below parameters :

Search (Ctrl+F)

Counter Settings

Prescaler (PSC - 1...	72-1
Counter Mode	Up
Counter Period (Aut...	256-1
Internal Clock Divisi...	No Division
Repetition Counter (..	0

Search (Ctrl+F)

PWM Generation Channel 1

Mode	PWM mode 1
Pulse (16 bits value)	0
Output compare preload	Enable
Fast Mode	Disable
CH Polarity	High
CH Idle State	Reset

PWM Generation Channel 2

Mode	PWM mode 1
Pulse (16 bits value)	64
Output compare preload	Enable
Fast Mode	Disable
CH Polarity	High
CH Idle State	Reset

PWM Generation Channel 3

Mode	PWM mode 1
Pulse (16 bits value)	128
Output compare preload	Enable
Fast Mode	Disable
CH Polarity	High

```
/* USER CODE BEGIN 2 */
HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_1);
HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_2);
HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_3);
HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_4);
```

You can start produce PWM signal each channel with this function.

```
/* USER CODE BEGIN 3 */
  //change pulse width of timer 1 channel 1 to 100
  __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_1, 100);
  //change pulse width of timer 1 channel 2 to 255
  TIM1->CCR2 = 255;
```

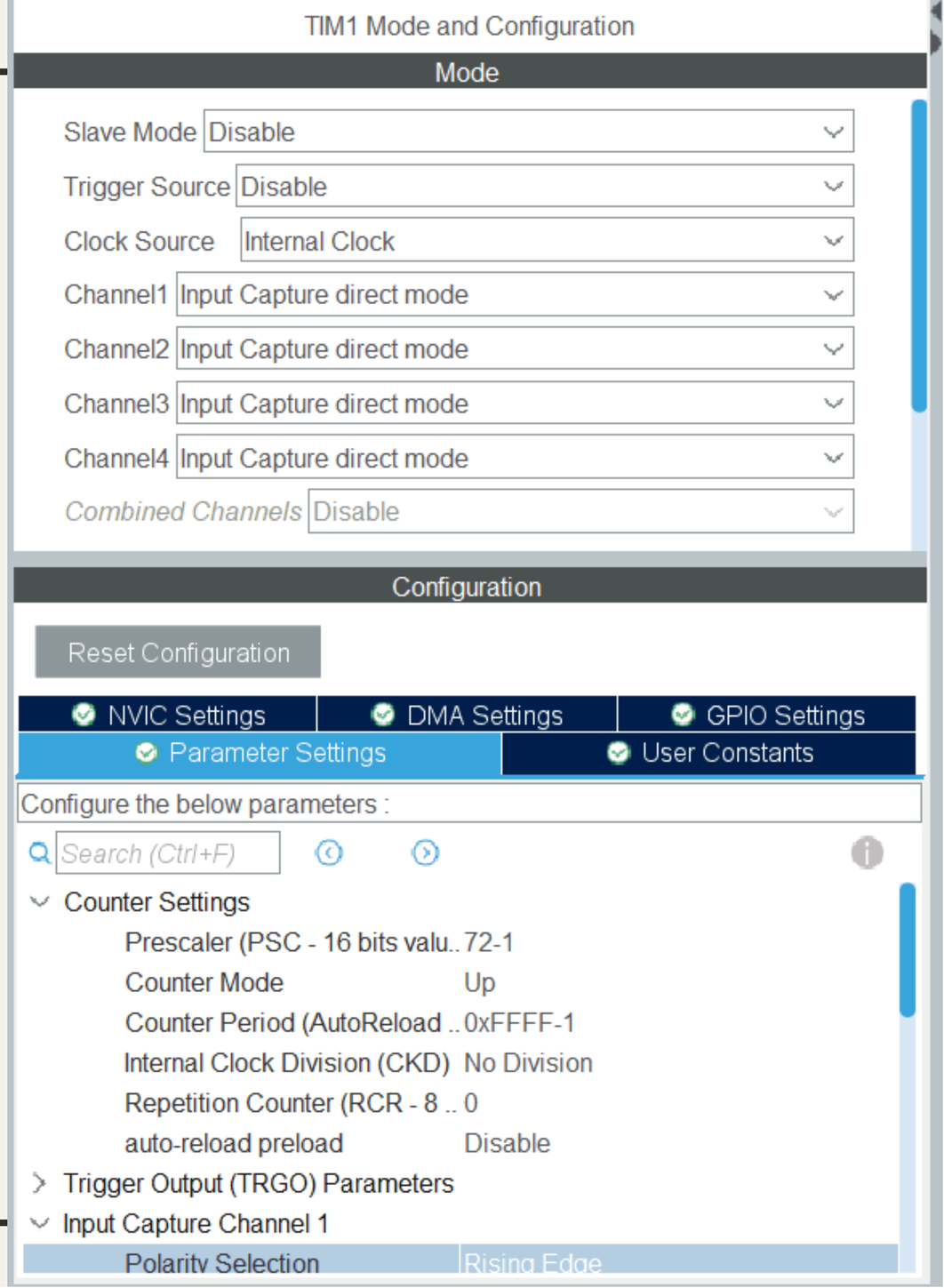
If you want to change the pulse length of a timer channel, you can call the function or write directly to the register.

```
/* USER CODE BEGIN 3 */
  //change timer 1 prescaler
  __HAL_TIM_SET_PRESCALER(&htim1, 10);
  TIM1->PSC = 10;
  //change timer 1 counter period
  __HAL_TIM_SET_AUTORELOAD(&htim1, 100);
  TIM1->ARR = 100;
```

You can also change the prescaler and counter period of the timer with this function.

c. Input Capture

Input capture is used to receive a PWM signal which can be used to measure the Frequency and width of the input signal. I set the prescaler to make 1 MHz timer and make counter period of its max value for 16 bit timer. The minimum frequency that the Timer can read is equal to (TIMx CLOCK/ARR). In polarity selection, you can select rising edge, falling edge, or both depends what do you want to capture.



For example, if you want to get the frequency, you can measure between two rising edges. When the first Rising edge occurs, the counter value is recorded. Another counter value is recorded after the second rising edge occurs. Now the difference between these 2 counter values is calculated. The Difference in the counter values will give us the frequency. If you want to get the pulse width, you can measure value between rising edge and falling edge. Callback function of input capture is

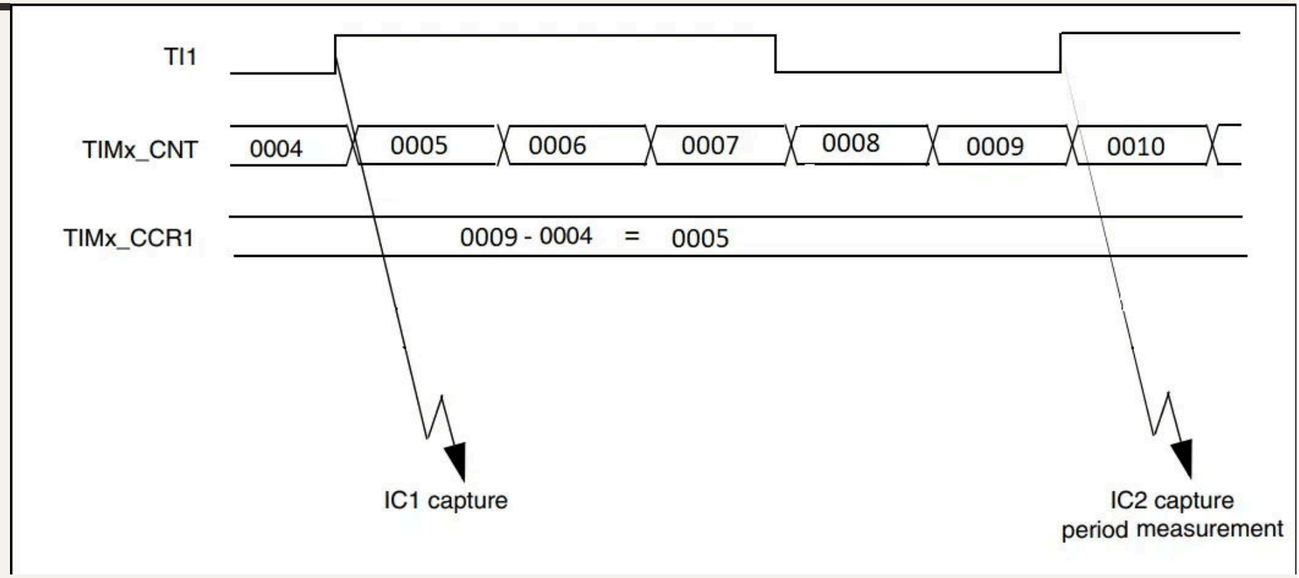
```
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)
```

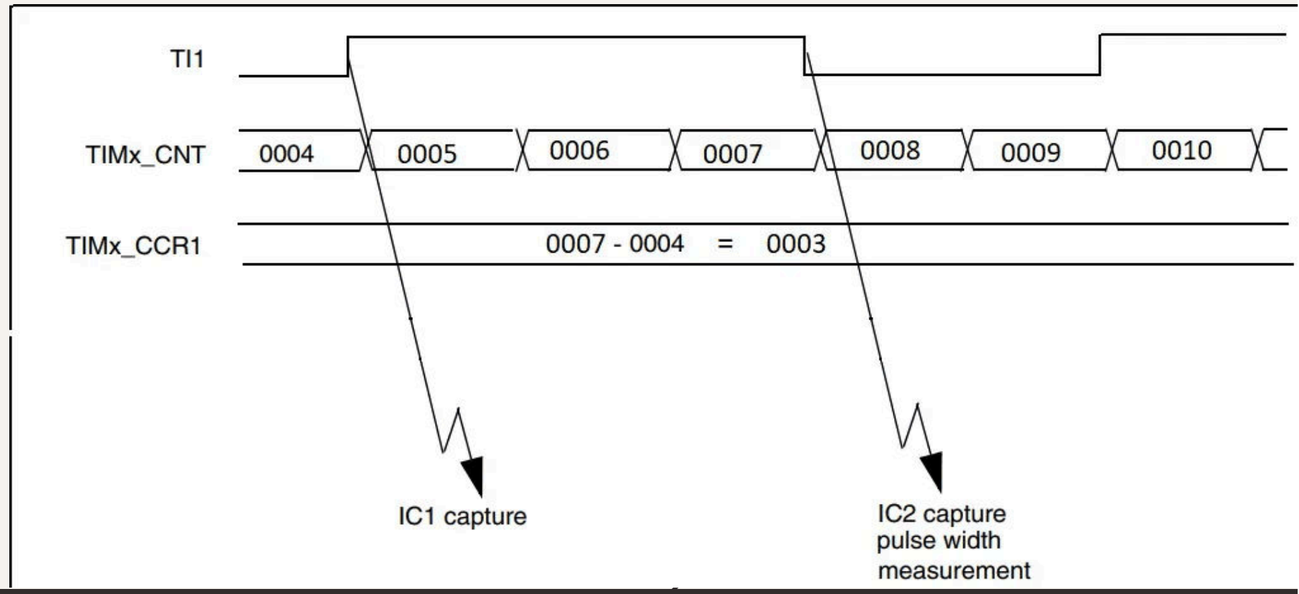
function to get current captured counter is

```
HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1);
```

function to reset counter is

```
//reset counter
  __HAL_TIM_SET_COUNTER(&htim1, 0);
TIM1->CNT = 0;
```



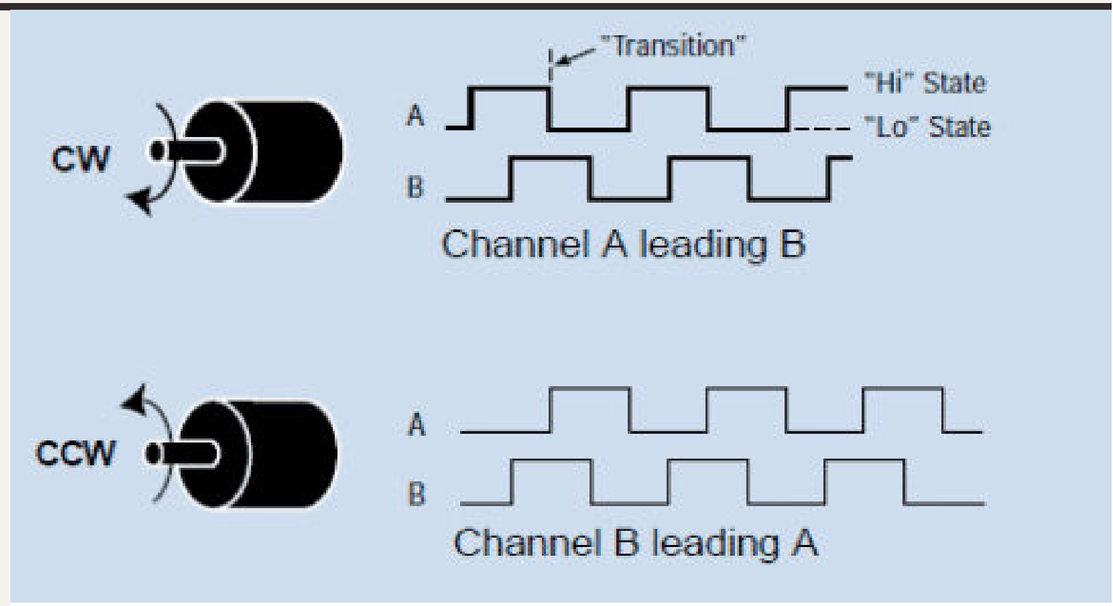


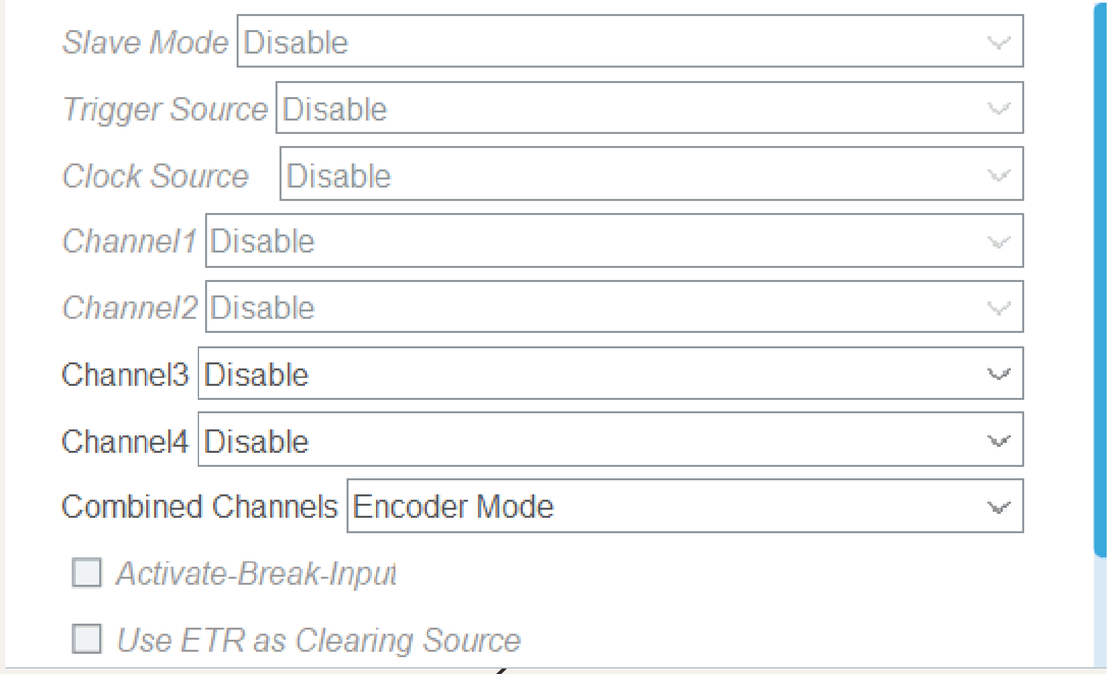
d. Encoder mode

An incremental encoder is a linear or rotary electromechanical device that has two output signals, A and B, which issue pulses when the device is moved. in timer configuration, select encoder mode in combined channel and 2 pin is automatically generated. just leave everything. to start measure encoder value, you can call this function.

```
/* USER CODE BEGIN 2 */
HAL_TIM_Encoder_Start(&htim1, TIM_CHANNEL_ALL);
```

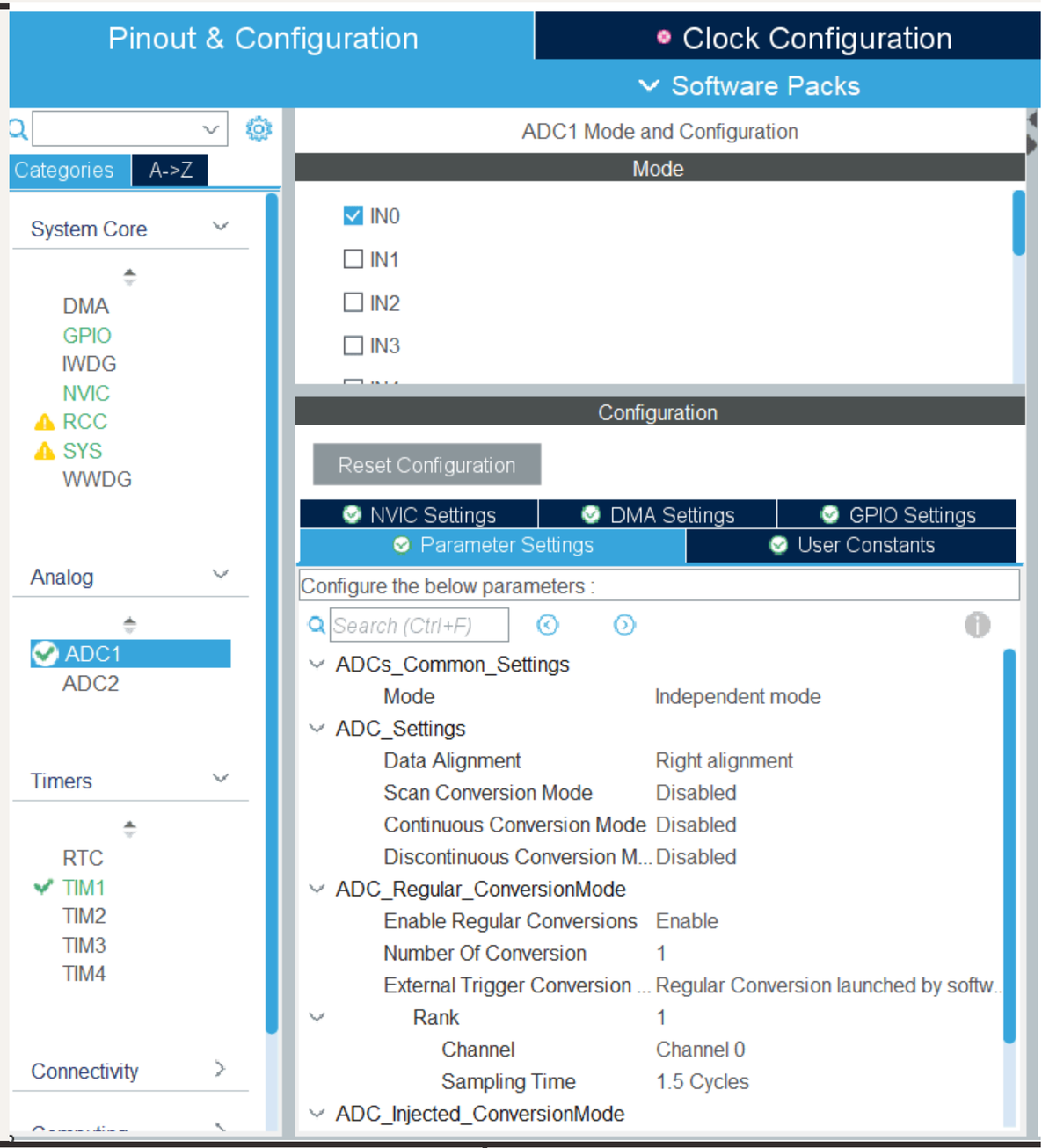
The timer counter value used for the encoder depends on the rotary encoder.





4. ADC

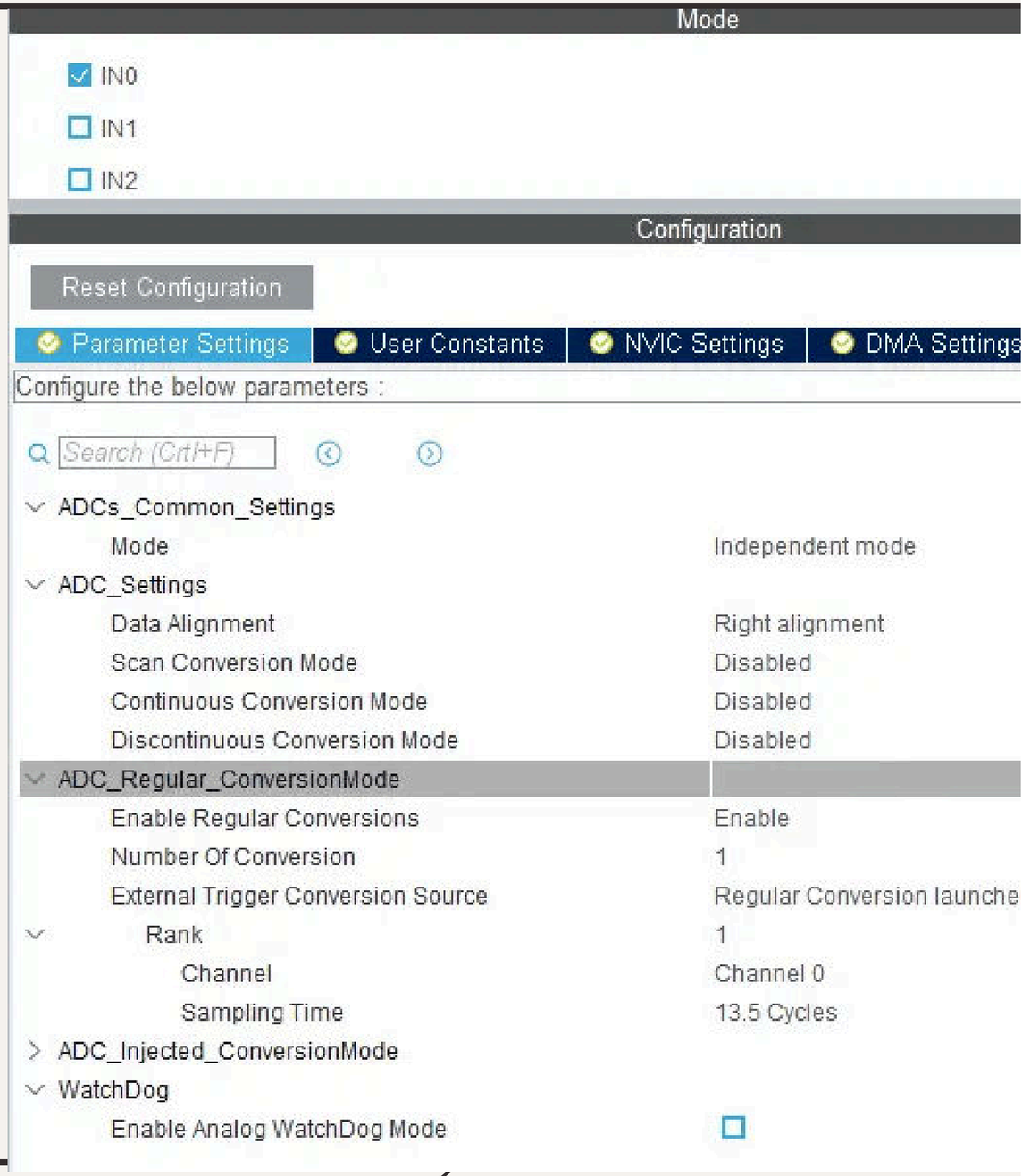
STM32 can measure adc values up to 16 bits, but the stm32 used in our series can measure a maximum of 12 bits. In Single conversion mode the ADC does one conversion and than stops. In continuous conversion mode ADC starts another conversion as soon as it finishes one. This method is more efficient if you want to convert continuously. Scan mode is used to scan a group of analog channels. This mode will be automatically selected if you are doing conversions for more than 1 channel. According to the datasheet, the total conversion time is calculated as follows: Tconv = Sampling time + 12.5 cycles Example: With an ADCCLK = 14 MHz and a sampling time of 1.5 cycles: Tconv = 1.5 + 12.5 = 14 cycles = (14cycles/14MHz)= 1 µs



a. Single channel using Poll for conversion

Here I am using only 1 channel and the continuous conversion is DISABLED. Also the sampling time is 13.5 cycles which is around 1 us, as the ADC clock is 12MHz. The code is as follows

```
/* USER CODE BEGIN 3 */
 HAL_ADC_Start(&hadc1);
 //poll for conversion with timeout 100ms
 HAL_ADC_PollForConversion(&hadc1, 100);
 //get adc value
 adc_val = HAL_ADC_GetValue(&hadc1);
 HAL_ADC_Stop(&hadc1);
```



b. Single channel using Interrupt

Poll for conversion uses blocking mode to monitor for the conversion and is not an efficient way to use ADC. Using Interrupt is an alternate way to do. First we need to enable continuous conversion mode otherwise after single conversion, ADC will stop and we have to restart it. Also make sure you enable the interrupt in the NVIC tab. First we have to start the ADC in the interrupt mode by using the function

```
HAL_ADC_Start_IT (&hadc1);
```

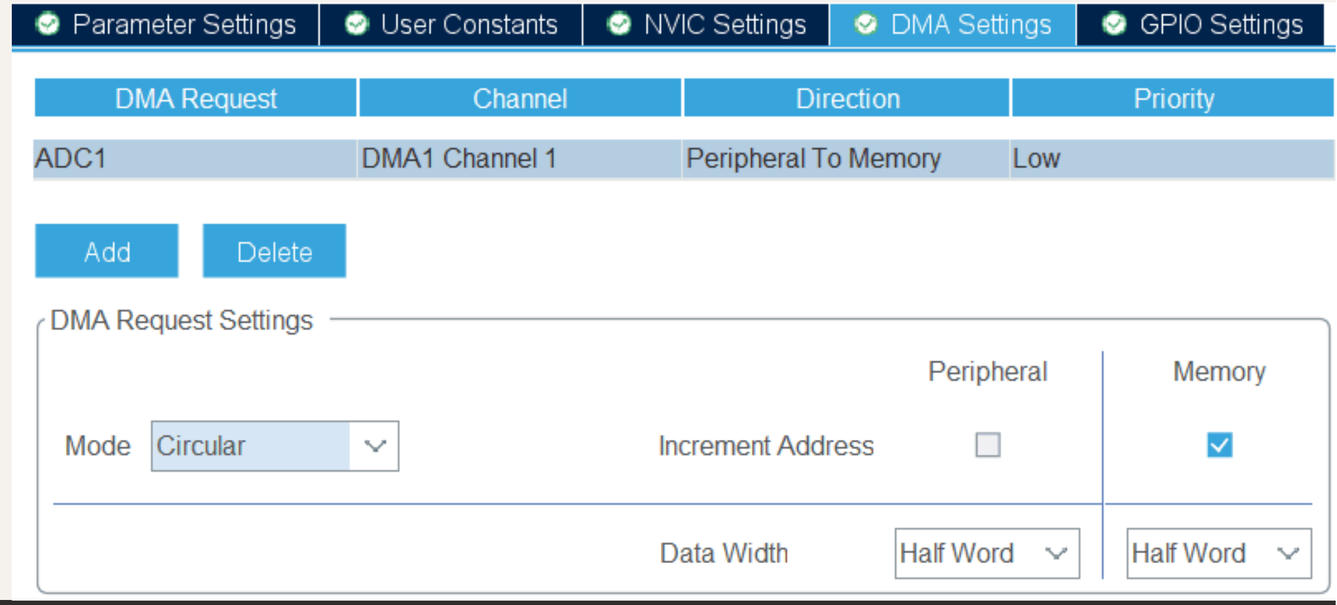
Now whenever the conversion is complete, a callback function is called and we are going to write the rest of the code inside it.

```
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
  adc_val = HAL_ADC_GetValue(&hadc1);
  /*If continuousconversion mode is DISABLED uncomment below*/
  //HAL_ADC_Start_IT (&hadc1);
}
```

c. Single channel using DMA (Direct Memory Access)

DMA method also works in a non-blocking mode. That means we can use the rest of the program while the DMA would be keep fetching the value in the background and when needed, we can get the value. In DMA method, whenever the conversion is complete, the ADC values are saved in the buffer, and we can read them anytime we want. The setup for the DMA is shown below.

- Circular DMA mode will ensure that the DMA will never stop even after the conversion is complete, the counter will be reloaded and the DMA will start again automatically.
- The Data width should selected as WORD (32 bit) or Half WORD (16 bit), as the resolution is 12 bit



To start ADC in DMA mode, we have to use the function below

```
HAL_ADC_Start_DMA (&hadc1, &buffer, 1);
```

This will start the ADC1 in DMA mode and the converted value will be stored in the buffer.
the callback function for ADC completed conversion is

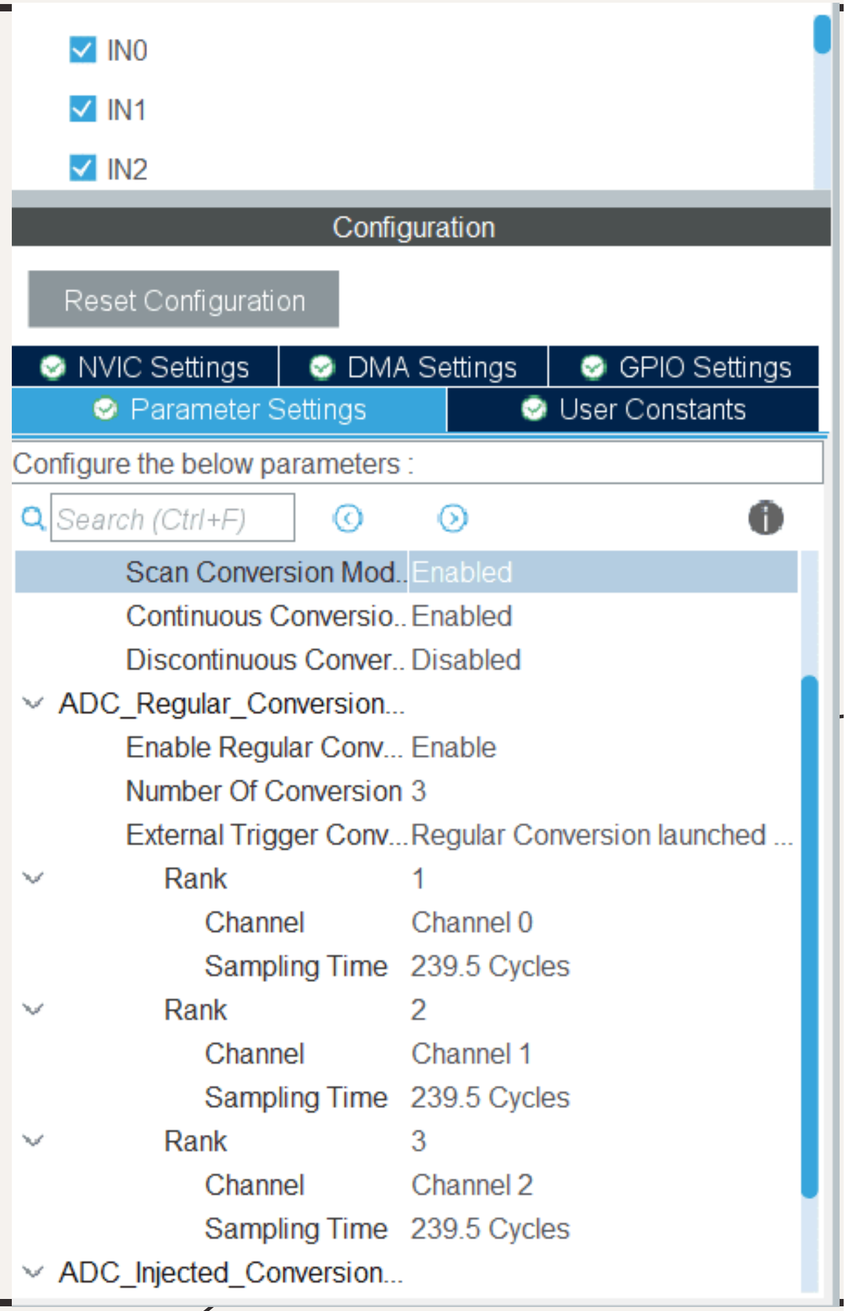
```
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
   adc_val = buffer;
}
```

DMA in Single Channel is Pointless, and it’s more useful in multi channel mode.

d. Multi channel using DMA

In order to read the multiple channels, we need to use DMA. The benefit of it is the conversion will take place in the background and we can perform some other operation with the controller and when we need the values, we can just read them easily.

Note that 3 channels are selected and the Continuous conversion mode is enabled. Also in the new CubeMx, you don’t need to worry about scan conversion mode. It will enable and disable by it’s own, based on if you are using multi channels or only one channel. The number of conversions changes depending on the total number of ADCs enabled. Also change each rank to whatever channel and sampling time you want



Make sure that the DMA is circular and data width is selected as WORD or Half WORD

```
uint32_t value[3];
HAL_ADC_Start_DMA(&hadc1, value, 3); // start adc in DMA mode
```

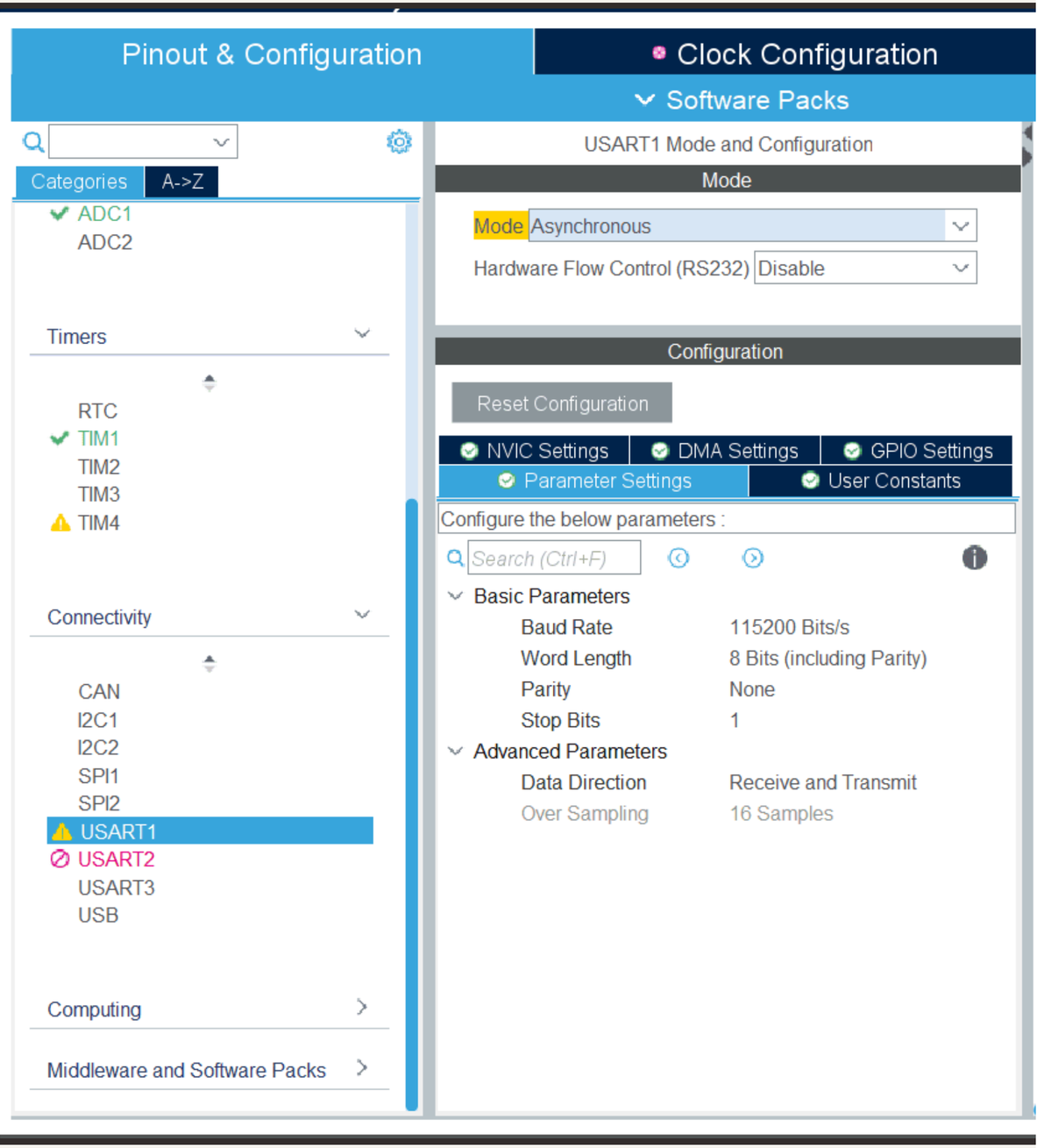
Starts the ADC in DMA mode and the converted data is stored in ‘value‘ buffer. ‘3’ is the length of data to be transferred from ADC peripheral to memory.

```
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc){
     //all channel conversion is completed
}
void HAL_ADC_ConvHalfCpltCallback(ADC_HandleTypeDef *hadc){
     //half channel conversion is completed
}
```

You can use completed or half completed callback if you create that function.

5. Serial Communication

We will cover how to Transmit data to UART (Universal Asynchronous Receiver Transmitter) in STM32. UART have two pins, TX and RX. TX is used for data transmissions and RX is used to receive data . The connection between two uart devices is always reversed. Go to connectivity and enable UART or USART as Asynchronuous mode.



a. Transmit using the poll method

The data is transmitted using blocking mode. For meaning, the CPU will block every other operation until the data transfer is complete. This method is good to use if you are only using UART and nothing else, otherwise all other operations will be affected.

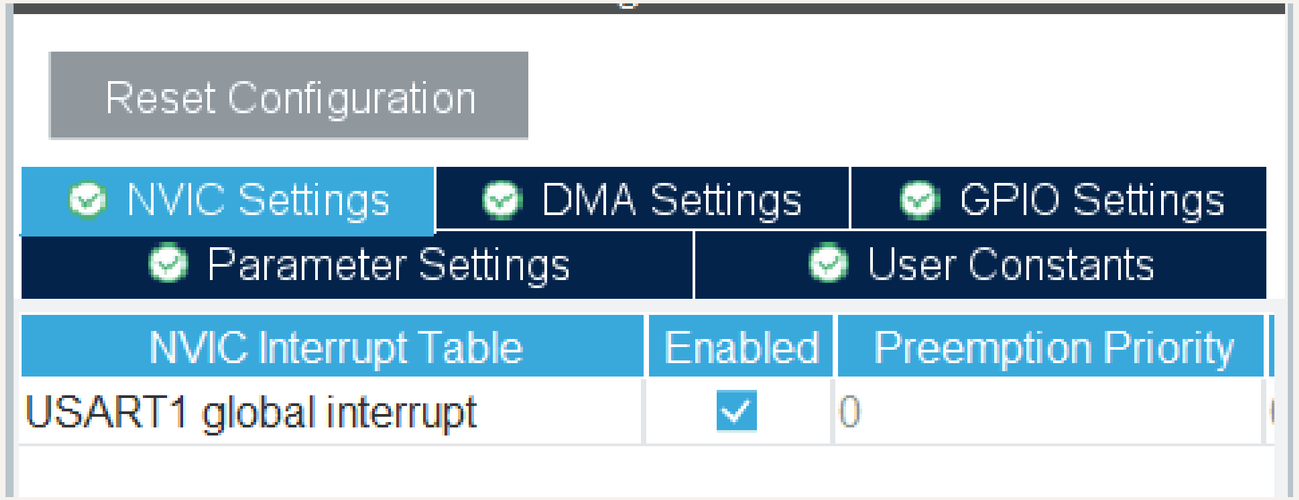
to transmit data, simply use this method

```
uint8_t data[] = "HELLO WORLD \r\n";
//transmit data to uart1 with 10ms timeout
HAL_UART_Transmit (&huart1, data, sizeof (data), 10);
```

buffer format is unsigned 8 bit integer that stores a string. This data will send via UART1 with a data size itself and a 10ms timeout. If you want to send larger data, you must increase the timeout value or the data will not sent completely

b. Transmit using the Interrupt method

In interrupt mode, Transmission takes place in non-blocking mode or in the background. So the rest of the processes works as they should. When the data transmission is complete, a Tx Complete Callback is called where we can write instructions like “what to do after the transfer is complete?”. In order to enable the Interrupt is in the NVIC Tab under the UART.



To transmit the data, you can use the function below

```
HAL_UART_Transmit_IT(&huart2, data, sizeof (data));
```

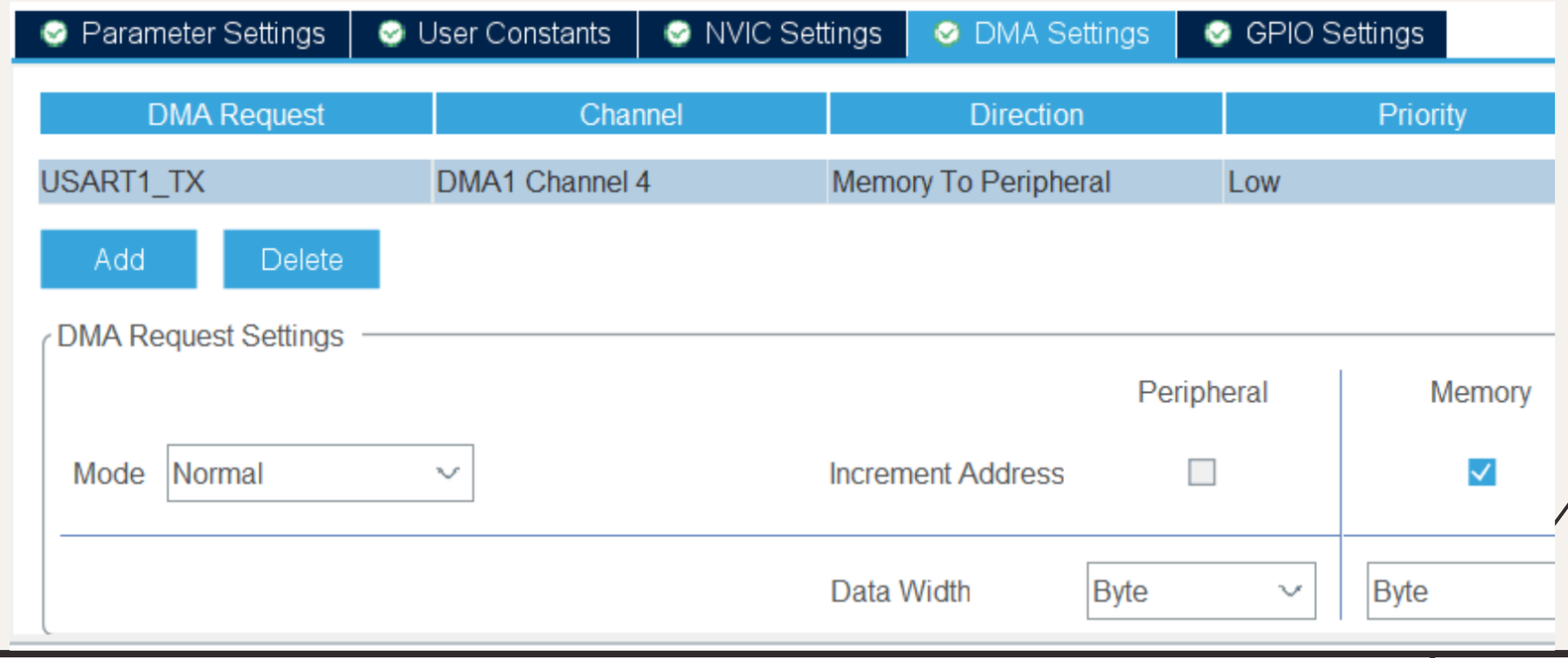
Transmit completed callback function

```
void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart)
{
   HAL_UART_Transmit_IT(&huart2, data, sizeof (data));
}
```

In the above code, HAL_UART_TxCpltCallback will be called when the data transmission is complete and as you can see inside this function, I am again starting a new data transmission.

c. Transmit using the DMA method

DMA also works somewhat same as interrupt, means that data transfer is in a non-blocking mode. In DMA, when half the data gets transferred, a half transfer complete interrupt gets triggered and when the data transfer completes, a transfer complete interrupt gets triggered. To Setup the DMA, we have to ADD the DMA in the DMA Tab under the UART.

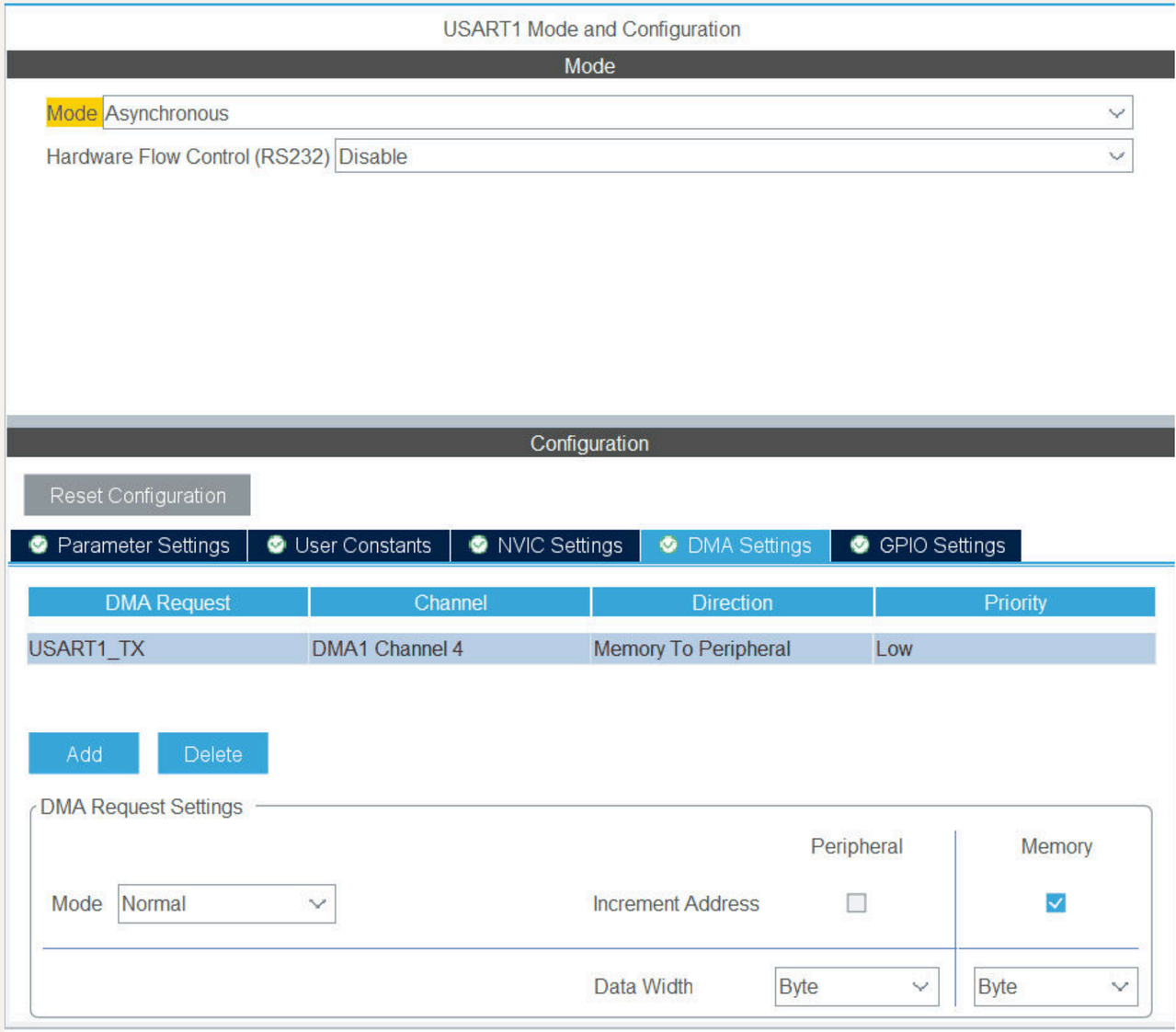


Here We are doing the Transmission, so UART1_Tx DMA is added. In the Circular mode, the DMA will keep transmitting the data. After Transmitting all the data, it will start automatically from the beginning. Data Width is selected as Byte, as we are sending characters, which takes only 1 byte in the memory. To start transmission, you can call this function.

```c
uint8_t data[] = "HELLO WORLD \r\n";
//start transmit data to uart1
HAL_UART_Transmit_DMA(&huart1, data, sizeof (data));
```

also DMA have a half and completed callback

```c
void HAL_UART_TxHalfCpltCallback(UART_HandleTypeDef *huart)
{
    //half transmit completed
}

void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart)
{
    //transmit completed
}
```



d. Receive using the poll method

The data is Received in blocking mode and the CPU will block every other operation until the data Reception is complete. This method is good to use if you are only using UART and nothing else, otherwise all other operations will be affected. To receive data using poll method simply use

```
// receive 4 bytes of data in 100ms timeout
HAL_UART_Receive (&huart2, Rx_data, 4, 100);
```

This will store the received data to Rx_data. Receive data using poll mode is so weak, we recommended you to use interrupt or DMA mode.

e. Receive using the Interrupt method

To Receive data using the Interrupt, activate interrupt in NVIC tab and we will use

```
uint8_t Rx_data[10];  //  creating a buffer of 10 bytes

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
  HAL_UART_Receive_IT(&huart2, Rx_data, 4);
}

HAL_UART_Receive_IT (&huart2, Rx_data, 4);
```

In the above code, receive complete interrupt will be called when the data reception is complete. As you can see inside this function, I am again starting a new data reception.This will result in continuous reception of data

f. Receive using the Interrupt method

To Receive the Data using the DMA, activate DMA as Rx mode, you can use normal or circular mode, so start receive data as DMA mode, we will do as follows

```
HAL_UART_Receive_DMA (&huart2, Rx_data, 4);  // Receive 4 Bytes of data
```

you can still use a half and completed receive callback function below

```
void HAL_UART_RxHalfCpltCallback(UART_HandleTypeDef *huart)
{
    //half receivve completed
}

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    //receive completed
}
```

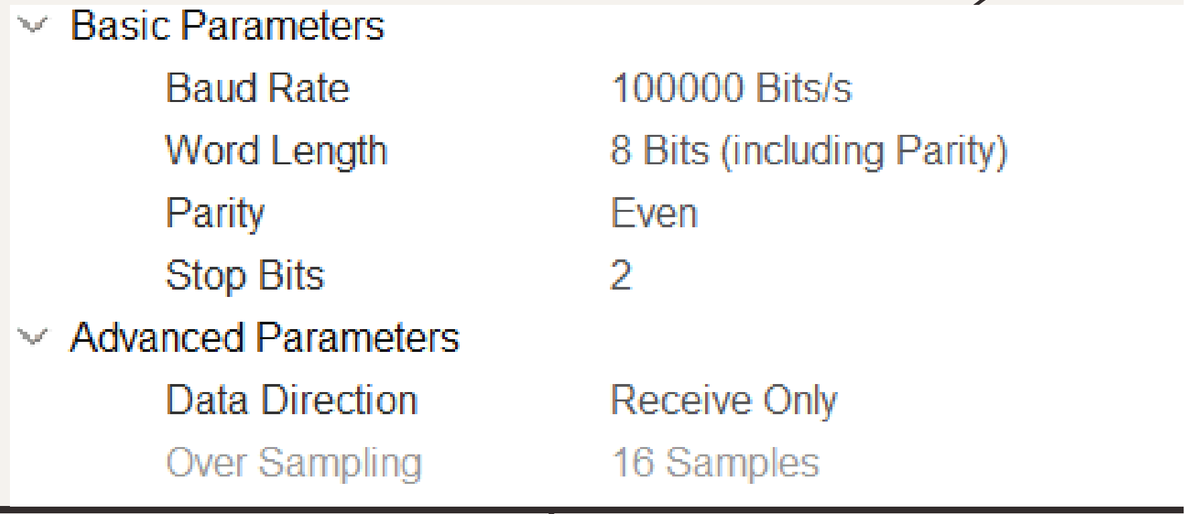
g. receive SBUS data

SBUS is a bus protocol for receivers to send commands to servos. Unlike PWM, SBUS uses a bus architecture where a single serial line can be connected with up to 16 servos with each receiving a unique command. The SBUS protocol uses an inverted serial logic with a baud rate of 100000, 8 data bits, even parity, and 2 stop bits. The SBUS packet is 25 bytes long consisting of:

- Byte[0] : SBUS header, 0x0F
- Byte[1 -22] : 16 channels, 11 bits each
- Byte[23] : Bit 7 : channel 17, Bit 6 : channel 18, Bit 5 : SBUS_SIGNAL_LOST (0x01), Bit 4 : SBUS_SIGNAL_FAILSAFE (0x03)
- Byte[24]: SBUS footer (0x00)

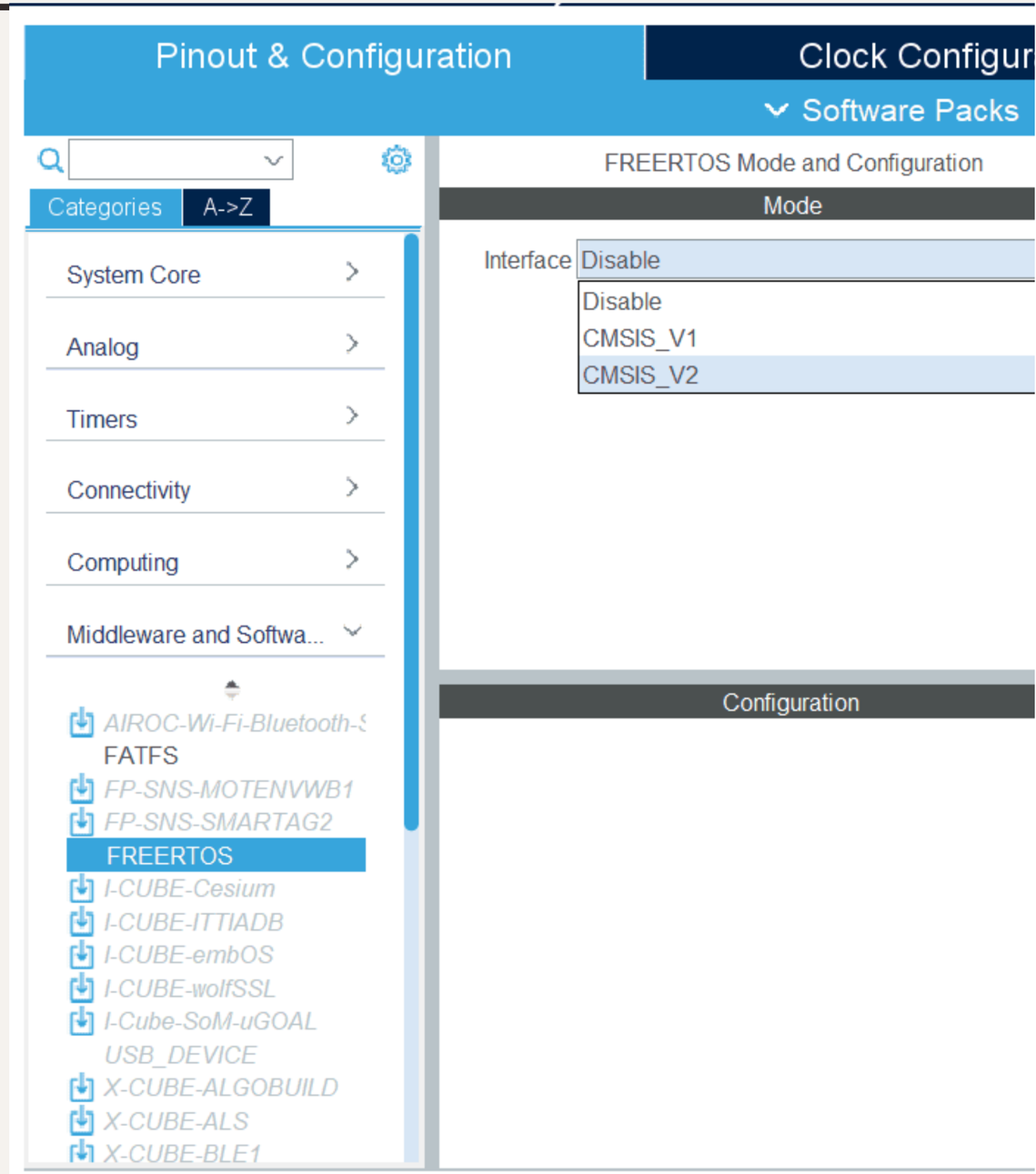
we suggest you to use DMA mode to get data quickly

```
if (buf[0] == 0x0F) {
        CH[0] = (buf[1] >> 0 | (buf[2] << 8)) & 0x07FF;
        CH[1] = (buf[2] >> 3 | (buf[3] << 5)) & 0x07FF;
```

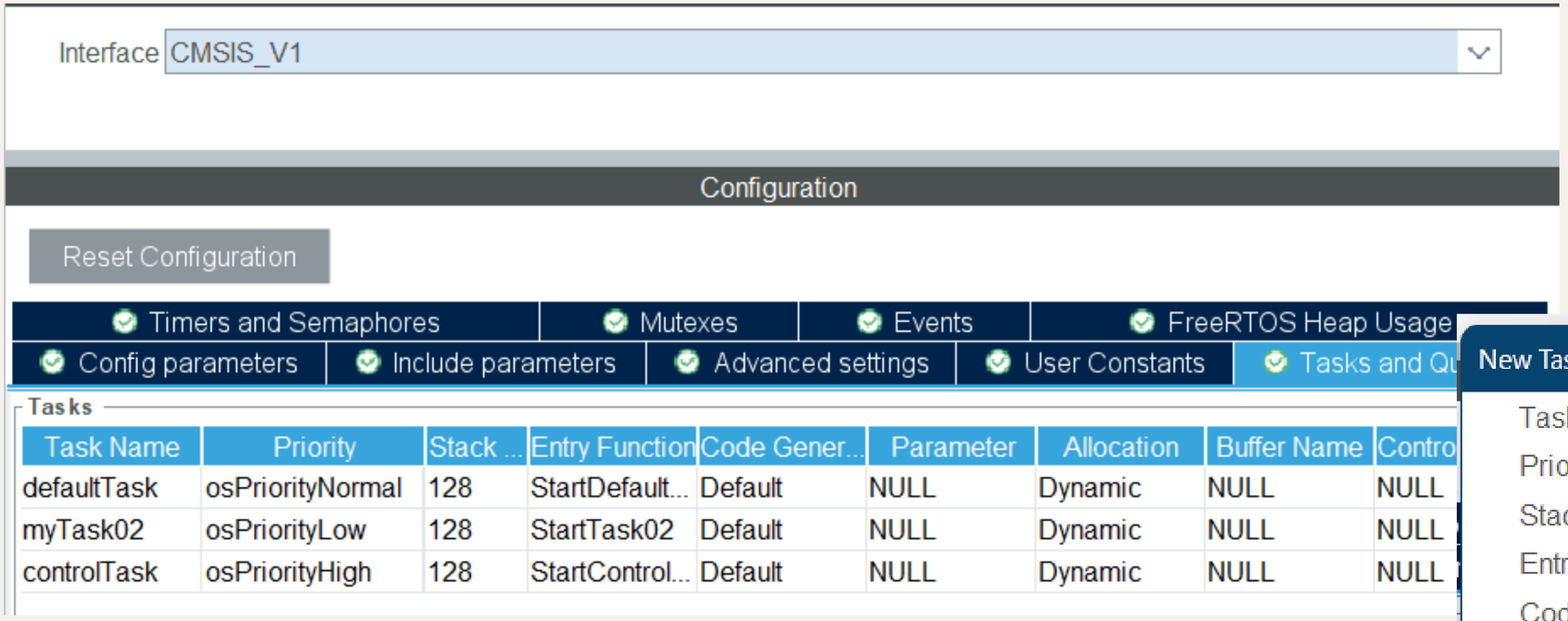


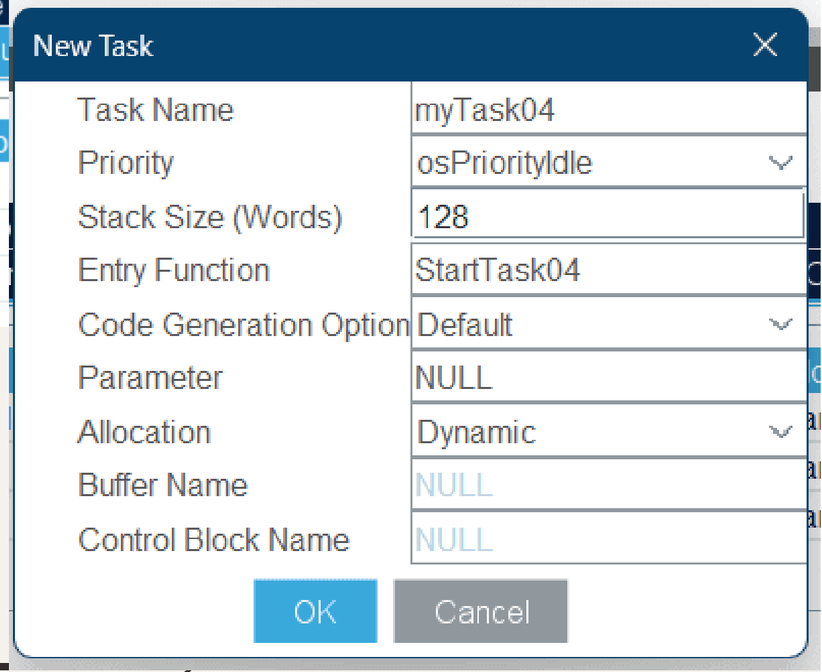
6. RTOS

RTOS stands for Real Time Operating System. And as the name suggests, it is capable of doing tasks, as an operating system does. The main purpose of an OS is to have the functionality, where we can use multiple tasks at the same time. Which obviously isn't possible in normal mode. You can enable RTOS in Middleware and Software Packs->FREERTOS. Here we have 2 version of CMSIS (Common Microcontroller Software Interface Standard). The first version has wider compatibility than the second one.

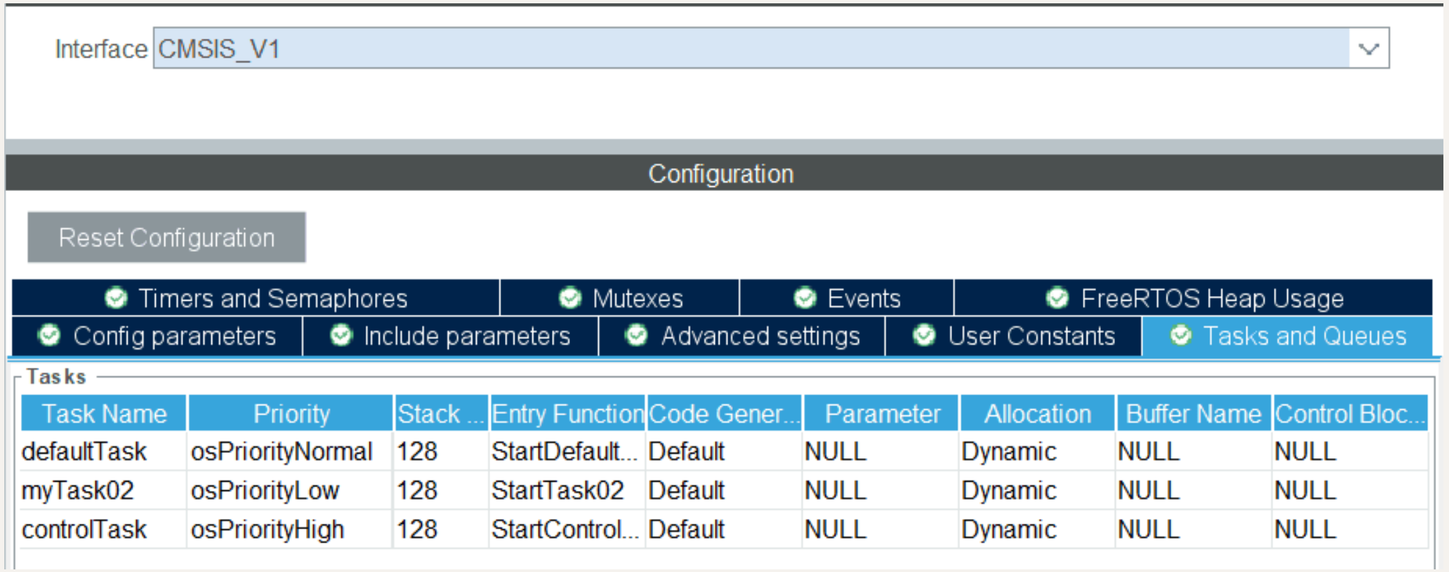


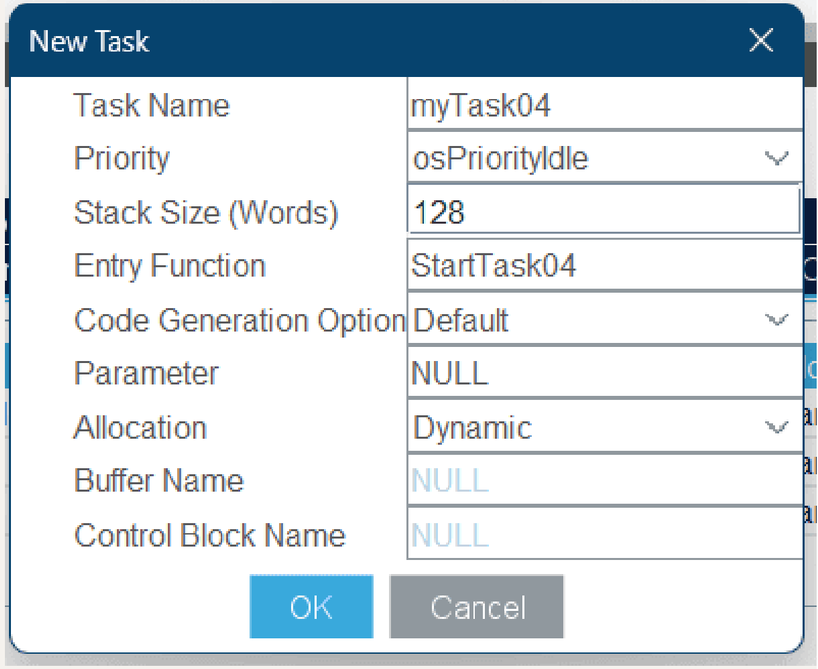
go to the ‘tasks and queues‘ tab and here you will see a default task created by default. you can add another task and just focus on task name, priority, and entry function.



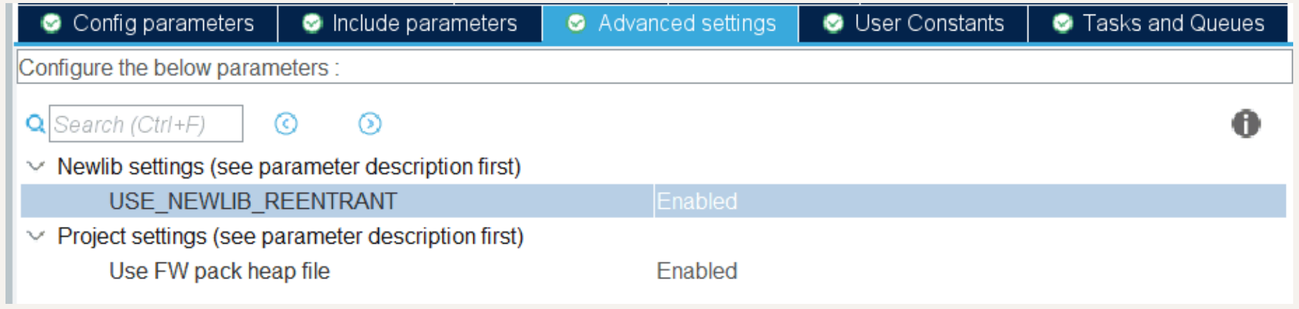


go to the ‘tasks and queues‘ tab and here you will see a default task created by default. you can add another task and just focus on task name, priority, and entry function.

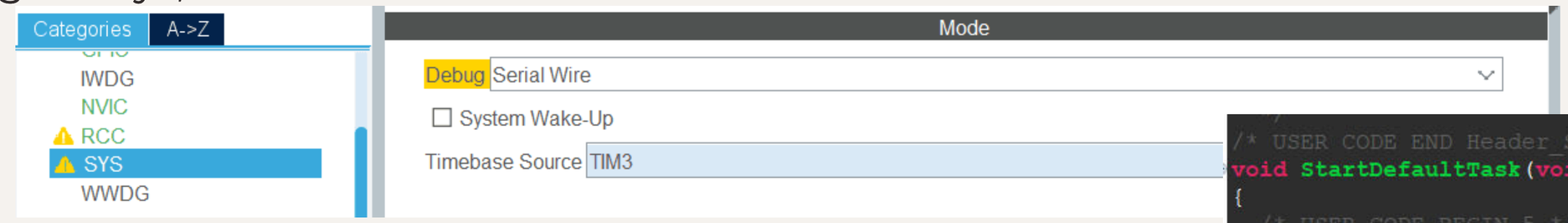




We need to enable the NEWLIB_REENTRANT in advanced settings tab



Also one important thing about using RTOS is that, we can't use systick as the time base. So go to sys, and choose some other timebase as shown below



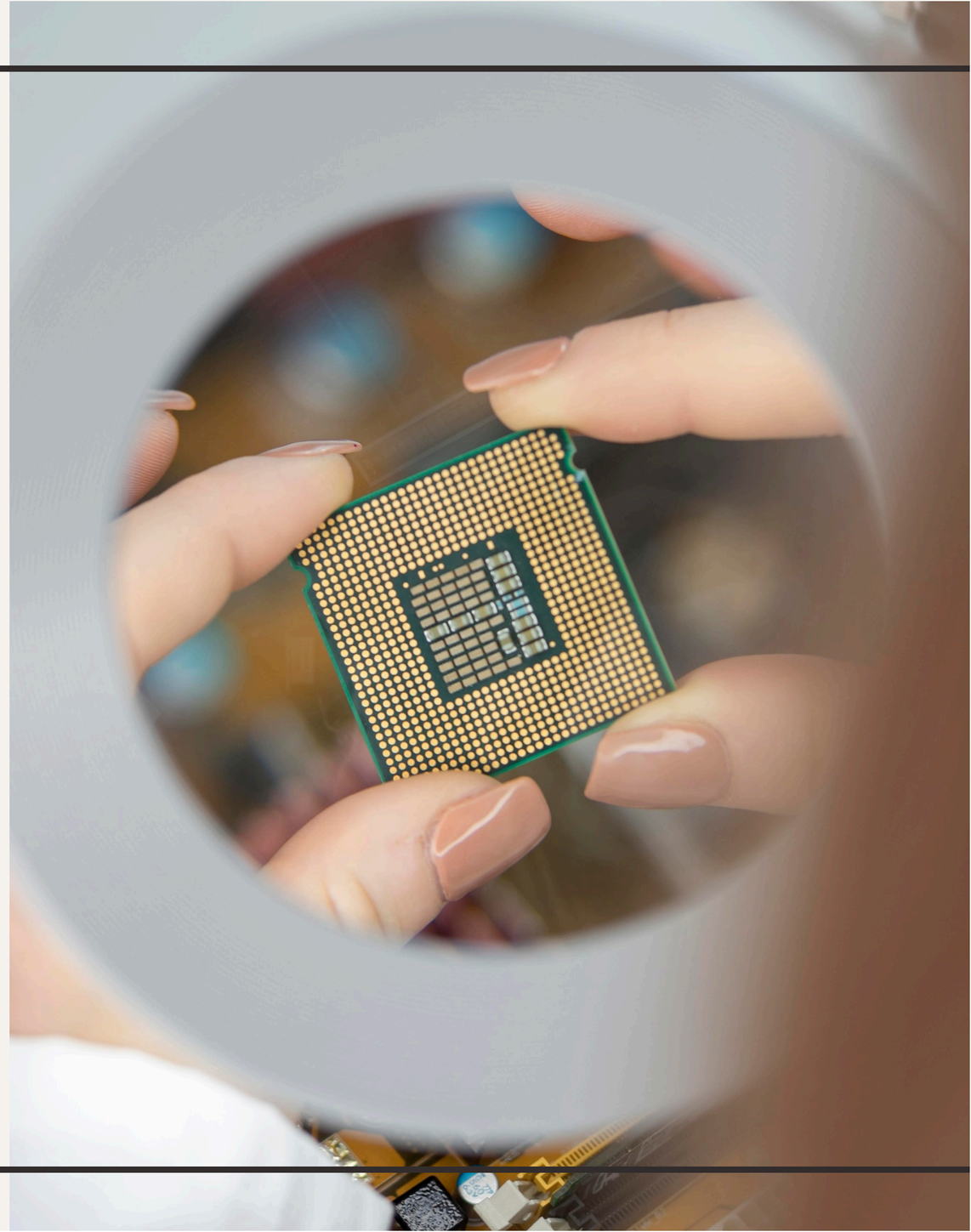
once the code is generated, you cannot write code in the main function after the kernel is started. You must write the initial code beforehand or write it in the created task.

```
/* Create the thread(s) */
/* definition and creation of defaultTask */
osThreadDef(defaultTask, StartDefaultTask, osPriorityNormal, 0, 128);
defaultTaskHandle = osThreadCreate(osThread(defaultTask), NULL);

/* definition and creation of myTask02 */
osThreadDef(myTask02, StartTask02, osPriorityLow, 0, 128);
myTask02Handle = osThreadCreate(osThread(myTask02), NULL);

/* definition and creation of controlTask */
osThreadDef(controlTask, StartControlTask, osPriorityHigh, 0, 128);
controlTaskHandle = osThreadCreate(osThread(controlTask), NULL);

/* USER CODE BEGIN RTOS_THREADS */
/* add threads, ... */
/* USER CODE END RTOS_THREADS */

/* Start scheduler */
osKernelStart();

/* We should never get here as control is now taken by the scheduler */
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
```

```
/* USER CODE END Header_StartDefaultTask */
void StartDefaultTask(void const * argument)
{
  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {
    osDelay(1);
  }
  /* USER CODE END 5 */
}

/* USER CODE BEGIN Header_StartTask02 */
/**
* @brief Function implementing the myTask02 thread.
* @param argument: Not used
* @retval None
*/
/* USER CODE END Header_StartTask02 */
void StartTask02(void const * argument)
{
  /* USER CODE BEGIN StartTask02 */
  /* Infinite loop */
  for(;;)
  {
    osDelay(1);
  }
  /* USER CODE END StartTask02 */
}
```

Conclusion

Congratulations on completing the basic guide to mastering **STM32** microcontrollers. Even though what is listed is only basic knowledge, a building cannot rise high without a foundation



Thanks!

andiabdillahcoc@gmail.com
+62 8133 593 5685
github.com/AndiArvy
@andi_abdllh