# MANUAL FOR LAB B1

ECE 298 – F2024 – Rev1.4

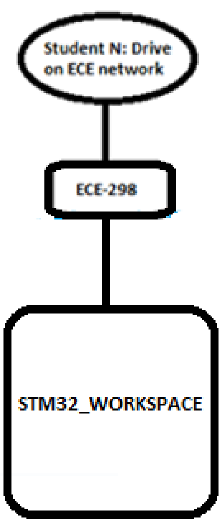
## Introduction

In your embedded system, you will be using a Nucleo STM32 MCU from STMicro. You will need to add interfaces that connect specific MCU pins to your Sensors, Actuators etc. The MCU pins on the Nucelo board can provide powerful functions, like General-Purpose I/O, I2C, SPI, Analog inputs, etc., but not necessarily all at once. You must be selective so that there is the best possibility of providing all the required functionality to operate those interfaces.

This manual will walk you through hardware setups and writing code in STM32CubeIDE platform that you can run on your MCU with your project. Becoming familiar with how this IDE tool is used for your MCU, lets you debug the hardware, software, and their interactions before any costly real-world prototypes are produced.

## Create an STM32 Development Workspace on the N: Drive

Go to the N: Drive and create a folder called ECE298. Then inside that folder, create a subfolder called STM32_Wrokspace (no spaces in the names please). This workspace will contain all your STM32 projects.




### Configuring the Environment for the STM32CubeIDE utility

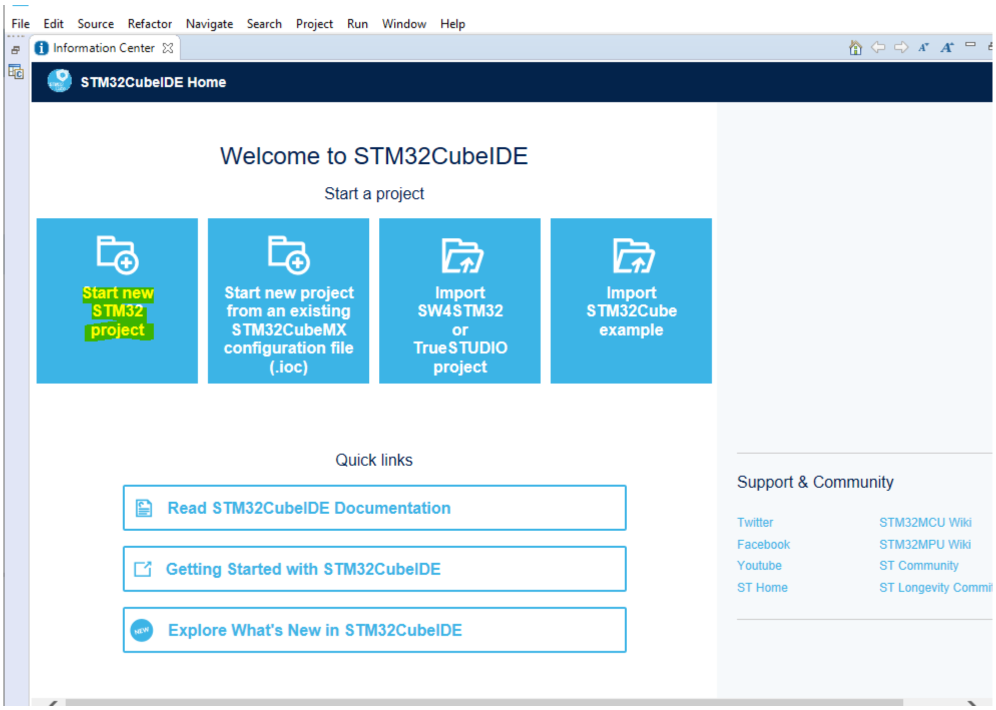
The STM32cubeIDE tool will help you select MCU pins for specific functions.

Go to the Start Menu and run STM32CubeIDE v1.16.0.

**NOTE: ALL OF THE FOLLOWING SCREENSHOTS ARE FROM THE ST MICRO STM32CubeIDE uitlity**

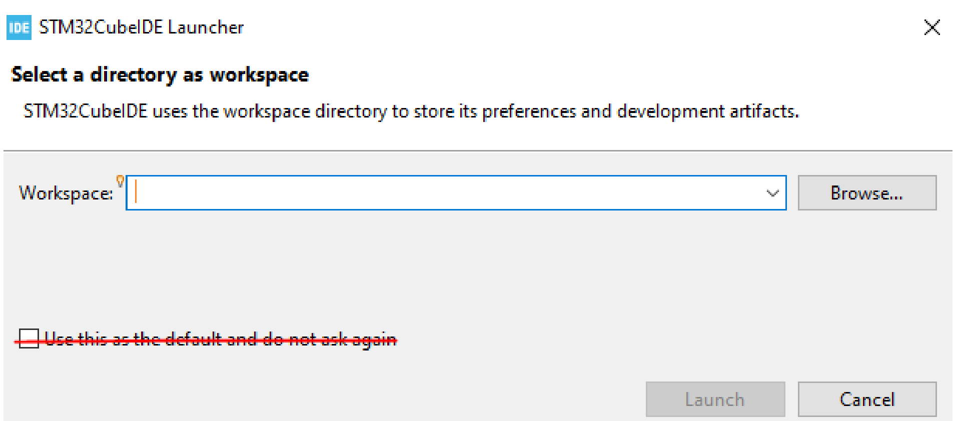
## Starting a Nucleo Project

A new popup window will appear:




Click on "Start new STM32 project".

You will see the following popup window:

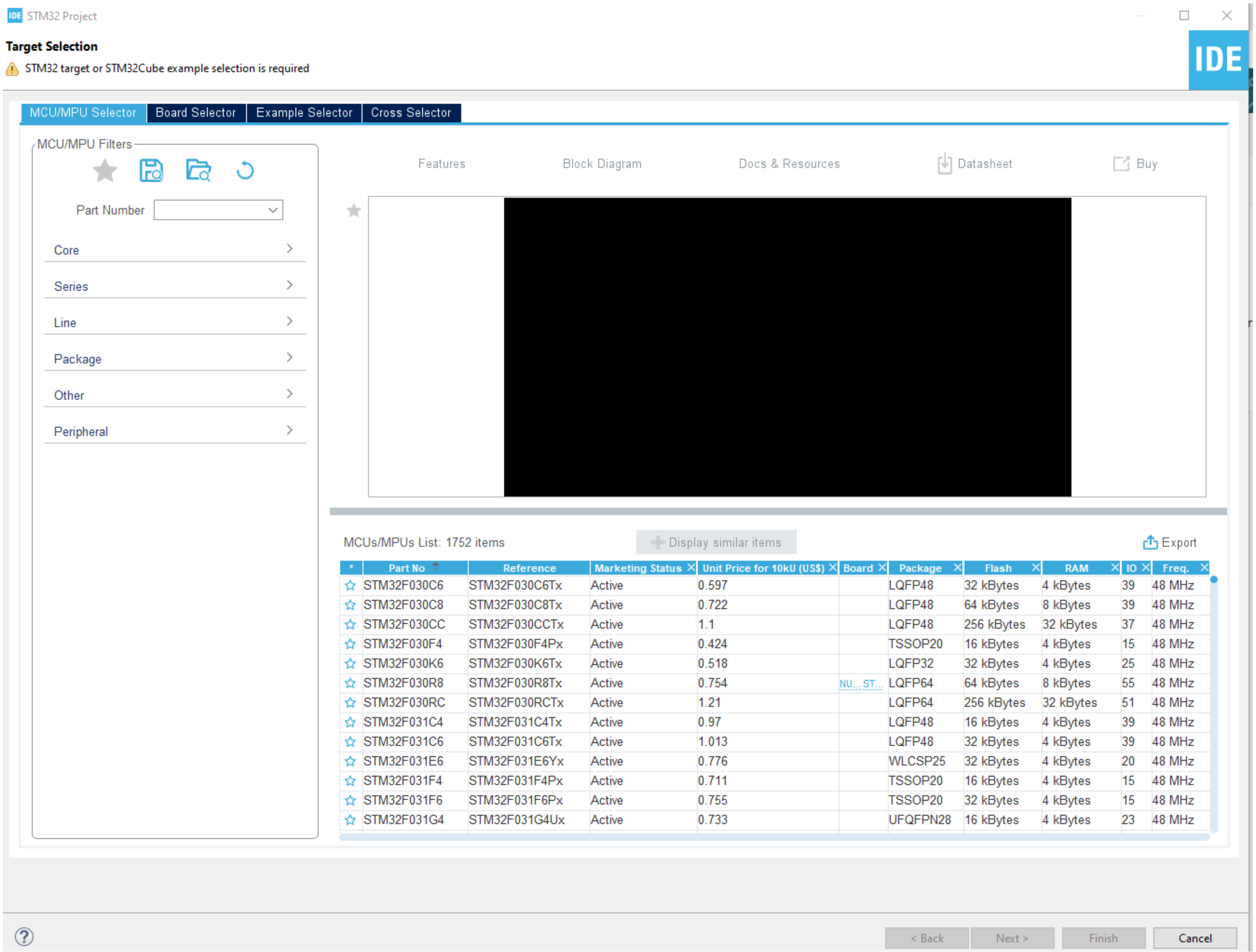



Click on the "Browse…" down-arrow button "V" beside the workspace field and browse to your

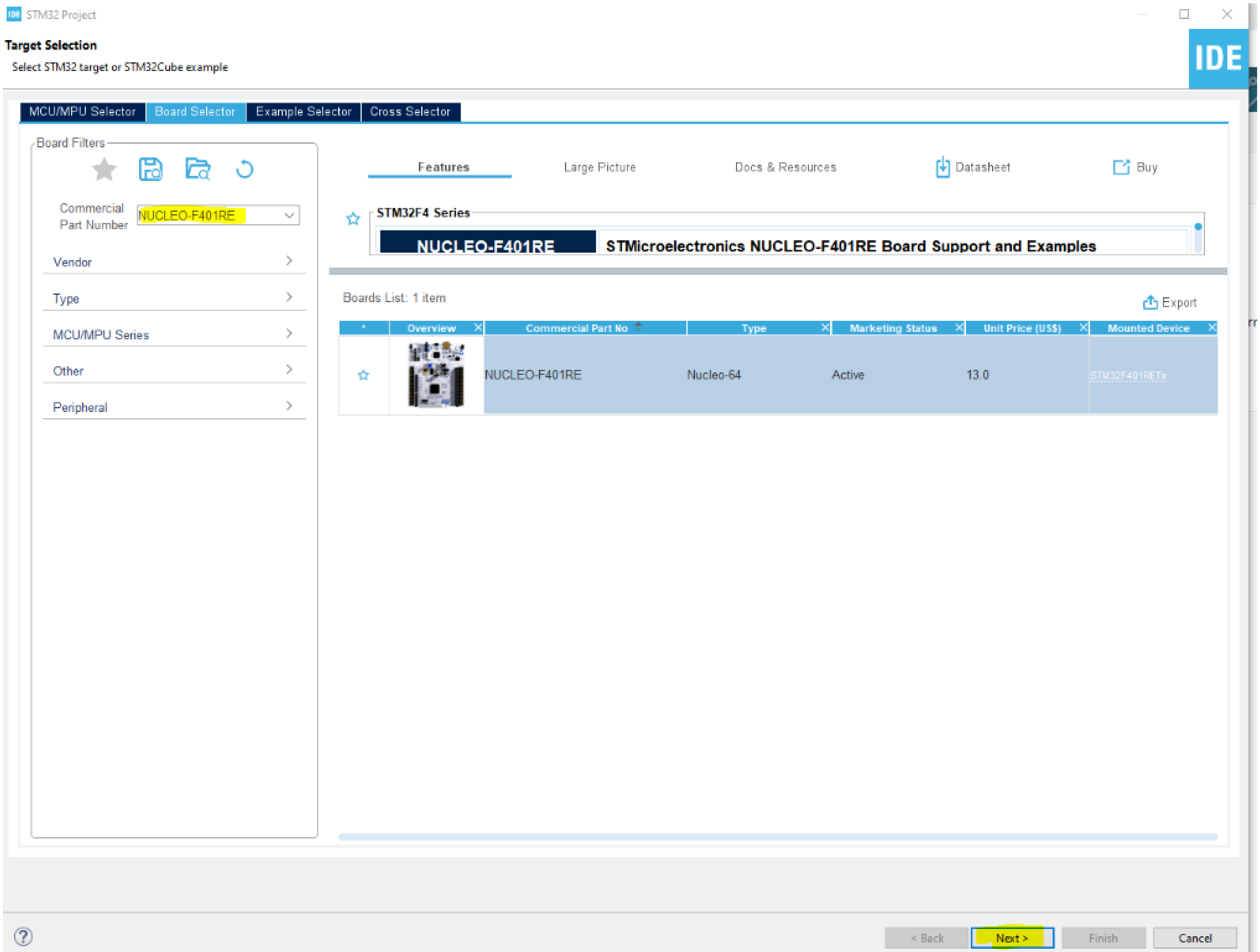
N-Drive ECE-298/STM32_Workspace folder. After that click the Launch button.

# FIRST EXERCISE: Blink Test Using MCU General-Purpose I/O pins

A new popup window appears that requests the MCU for the IDE project:



For this course, choose the Board Selector tab. Then a new window appears:



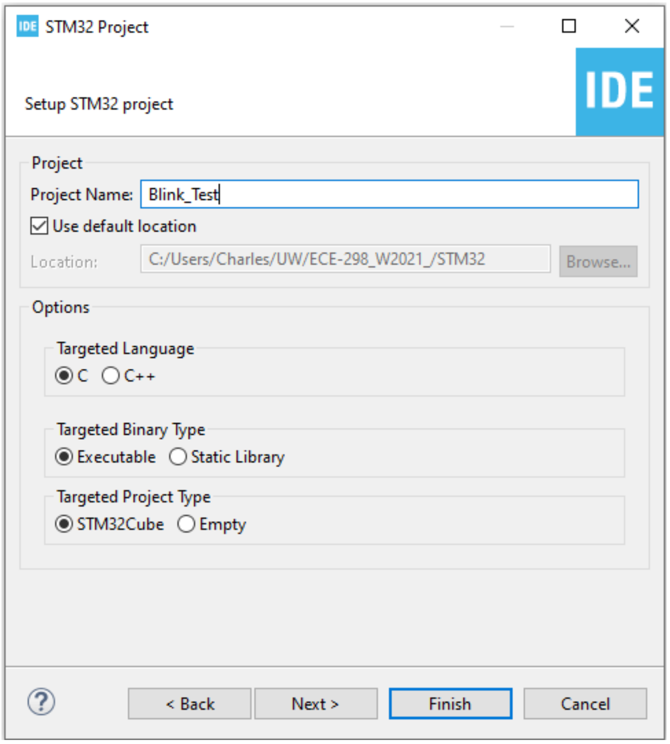

Depending on the Nucleo Dev board that was purchased from the W Store, you may have a Nucleo-F401RE or a Nucleo-F411RE type of board. Either of these boards is very suitable for this course.

Type (NUCLEO-F401RE or NUCLEO-F411RE) value in the TYPE field of the board you are using inside the "Commercial Part Number" field in the Board Filters area.

Click the "Next" button at the bottom.

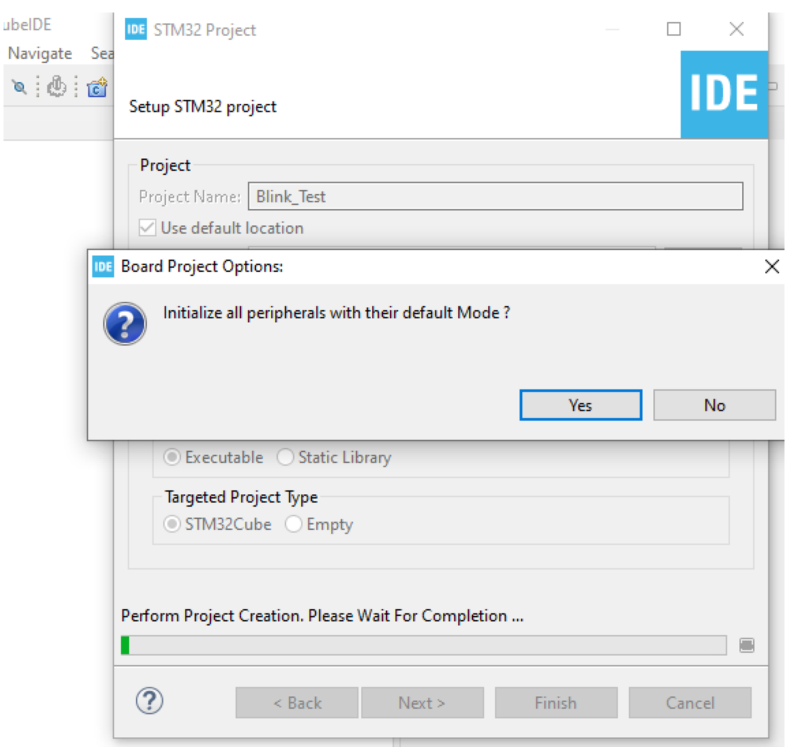
## Naming the Project:

A new project based on your board configuration will be defined. Name this project “Blink_Test”: (no spaces please)




Leave “Targeted Language”, “Targeted Binary Type”, and “Targeted Project Type” defaults as-is.

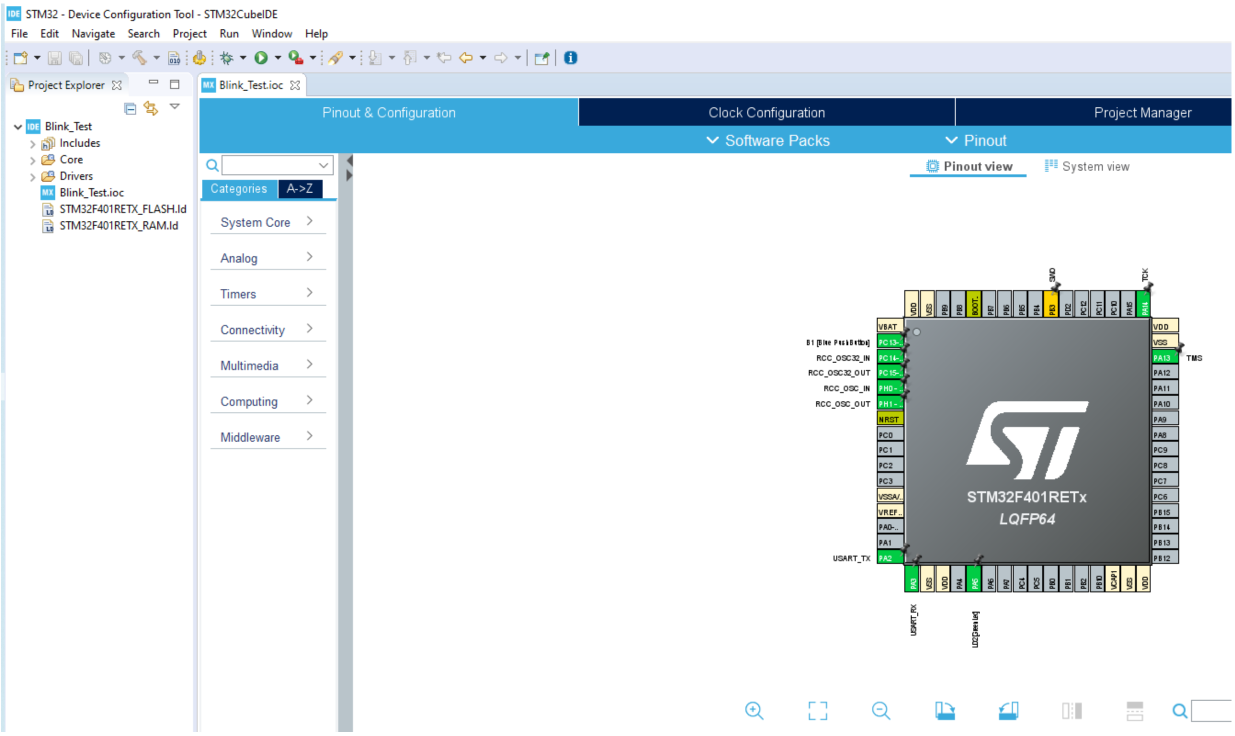
Click “Finish”.




Then a request comes to Initiaalize all peripherals with their Default mode….. CLICK YES.. This sets up the resources that are reserved for the Nucleo board operations with the host computer (running STM32CubeIDE).

THIS IS NECESSARY FOR ALL PROJECTS IN THIS COURSE.

The MCU Configuration Window becomes visible. It shows the pinout of the MCU on the Nucleo-F401/411RE development board:




A closer look at the pinout is below:

The **green** and **yellow** ones are reserved for the NUCLEO board.

We will not be retargeting any of those reserved pins for ECE 298 Embedded Project interfaces.

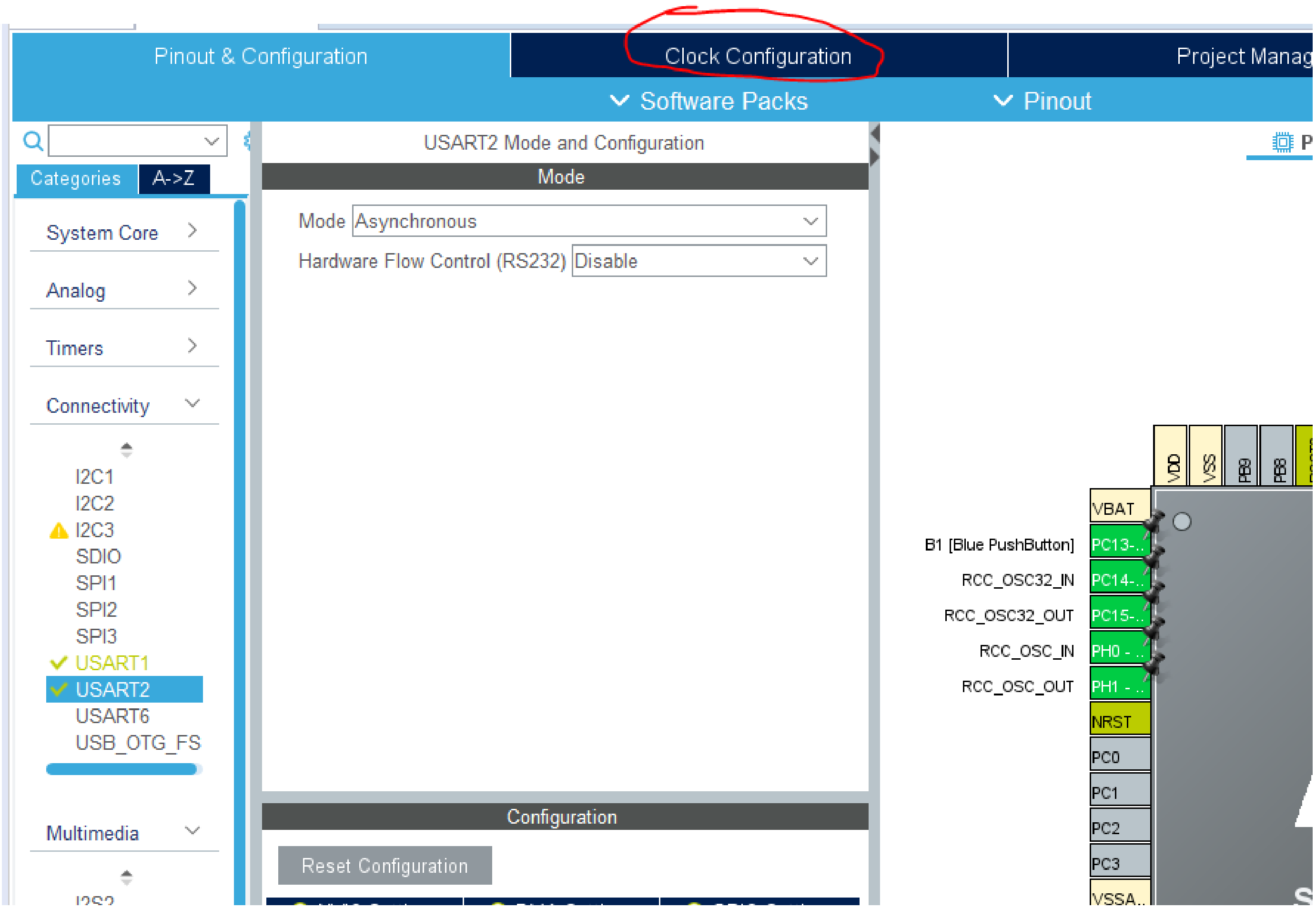
Note that GPIO pin LD2 <u>(PA5)</u> in the diagram (at the bottom). It is dedicated already on the reserved group for the “LD2” LED on the Nucleo development board. It is meant for USERS to use. This pin is already configured as a GPIO OUTPUT pin.

We will use LD2 as part of the first project (Blink_Test). We don’t have to set that up.

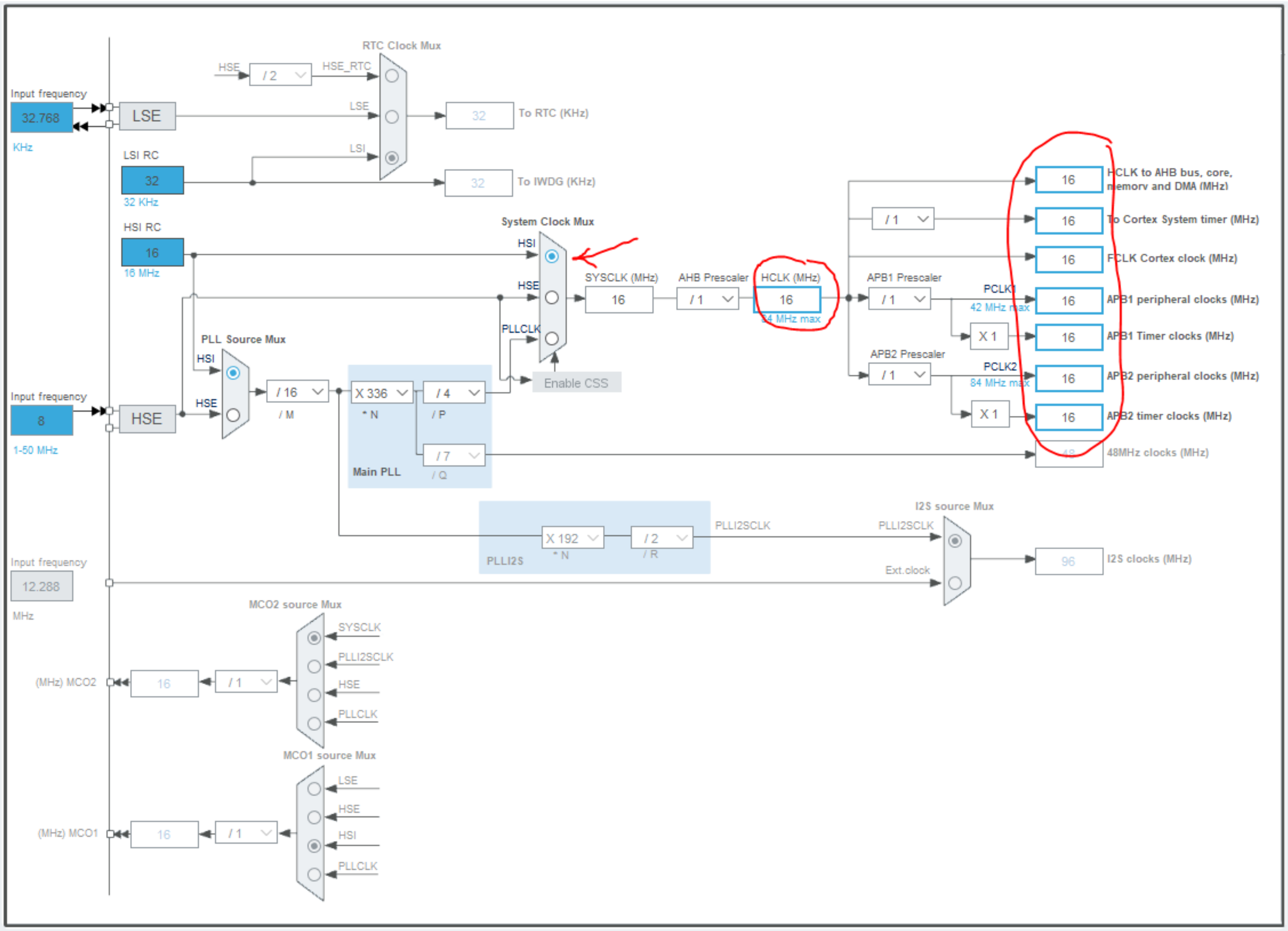
To set things up initially FOR ANY NUCLEO PROJECT in the STM32CubeIDE let’s start with the Clock tree that we will be using for this course.

## Clock Configuration

Choose the Clock Configuration Tab (circled in red below):



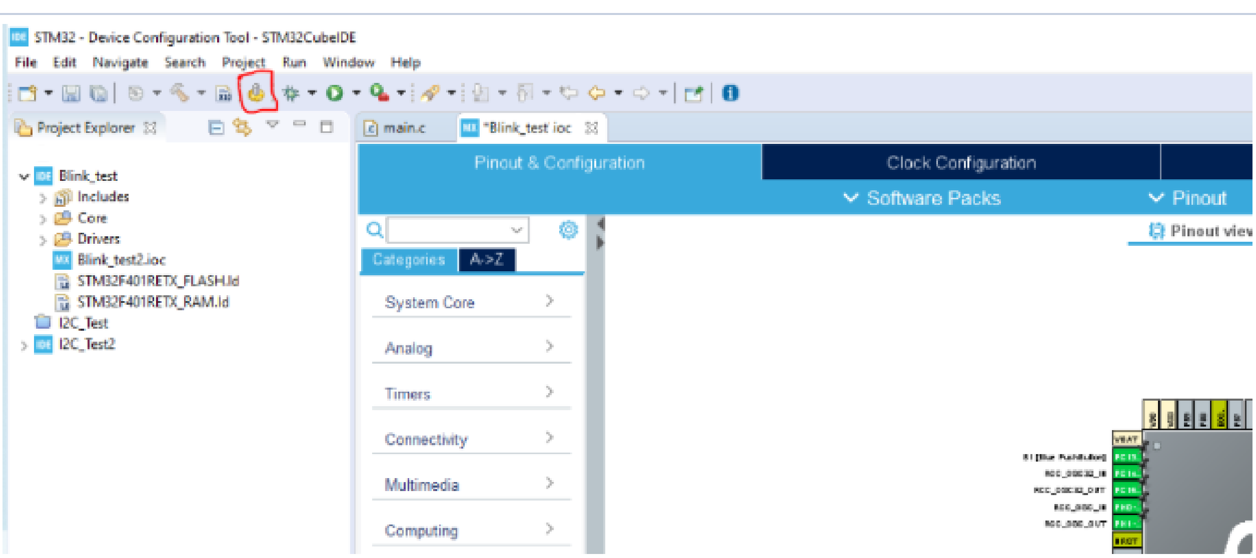

There is **one** clock configuration that will be used for all projects with our MCU in this course. This restriction simplifies the wide range of possibilities, for our class, to something manageable. We will **not** be using the PLL circuits. The Core Clock (HCLK) and all peripherals will be using a common 16MHz clock source. Refer to the diagram below and make sure that all of the red-circled parameters are configured for 16 MHz.



These are the last details to configure for this simpler Blink_Test project. Now we move on to "AUTO-Generating" the code for the MCU peripheral options we have chosen.

## Auto-Generating Code is to initialize the GPIO's, Peripherals and Clocks in your "main.c" program

Code that configures the clocks and internal resources of the MCU must now be generated. Click the **device configuration** button (red square in the figure below). The button looks like a COG wheel or gear.



Click “Yes” on the popup that asks if you want to change the “perspective” for developing C/C++ code.

Open the main.c file under Core→Src in the left pane above. Your main.c file has the generated code template ready for writing your MCU code. The GPIO initialization is done for the configured GPIO pins (PA5 for LD2 Blink_Test). This process generated the “MX_GPIO_Init” routine below:

```c
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOH_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOA, LD2_Pin, GPIO_PIN_RESET);

  /*Configure GPIO pin : B1_Pin */
  GPIO_InitStruct.Pin = B1_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);

  /*Configure GPIO pins : LD2_Pin */
  GPIO_InitStruct.Pin = LD2_Pin ;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

}
```

Find following in the code in the main.c file:

```c

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}

```

WHILE LOOP CODE IS INSERTED HERE

NOT here !

This area is where you will run your Blink_Test code in an infinite loop. **Any code you write here is safe from being overwritten by updates from the configuration tool that may come later in the Lab.**

> NOTE: In this course, besides using regular “C” coding language, there will be use of Hardware_Abstraction_Layer (HAL) commands to access GPIO pins and MCU peripherals. This enables the user to add the functionality to these resources with MINIMAL code statements. The GPIO pins and peripherals have extensive multiplexing registers to address the multitude of functions. The HAL commands allow easier coding.

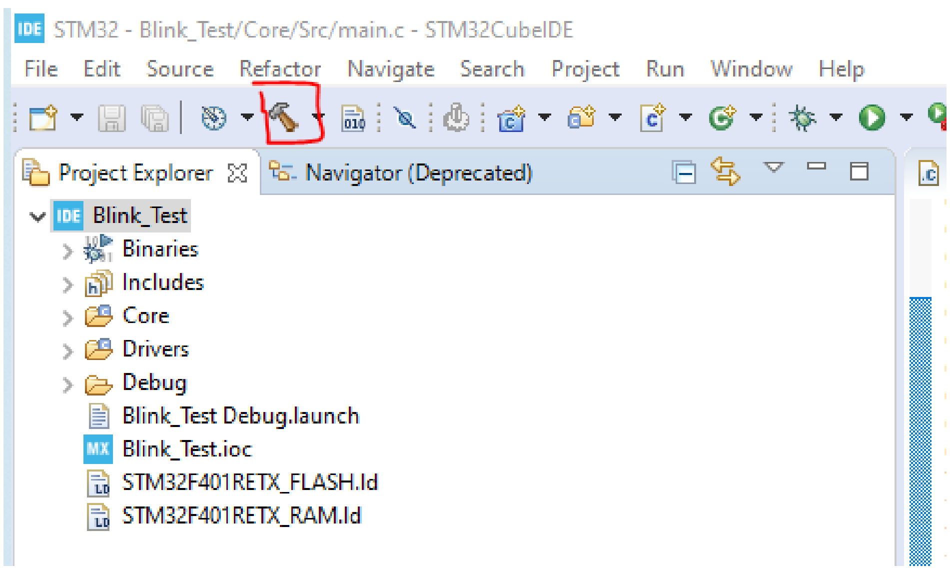
Enter the following HAL commands (Hardware Abstraction Layer) to toggle the LD2_pin:

```
while (1)
{
    HAL_GPIO_WritePin(GPIOA,LD2_Pin, GPIO_PIN_SET);
    HAL_Delay(250);
    HAL_GPIO_WritePin(GPIOA,LD2_Pin, GPIO_PIN_RESET);
    HAL_Delay(250);

  /* USER CODE END WHILE */
```

The parameter passed to HAL_Delay is a time delay in milliseconds. The total period of the blinking behaviour is LD2 Time ON plus LD2 Time OFF. The period id 500 msec. The frequency is 2 Hsz.

For this small program the amount of ON time = amount of OFF time. Thus the “Duty Cycle” is 50% because half of the period of the repeating signal is ON and half of it is OFF.,

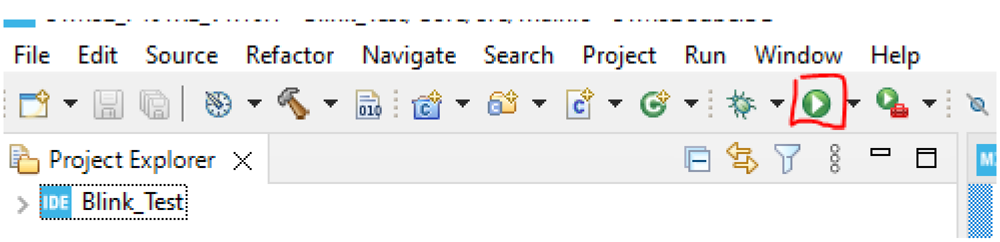
Now things are ready for compilation. Click on the “Hammer” icon:




If there are no compile errors (due to typos, etc.), your executable will be created.

Connect the Nucleo board to a Lab Computer USB port.

Download the executable file to the Nucleo Dev Board by going to the STM32CubeIDE and clicking the




After the download is completed, the LD2 LED should begin to do a 2Hz blink pattern. Note how the LD2 ON time is the same as the OFF time.

Here is a simple thing to try. Change the delay parameters in the program to 10 msec for ON time and 490 msec for OFF time.

```
while (1)
{
    HAL_GPIO_WritePin(GPIOA,LD2_Pin, GPIO_PIN_SET);
    HAL_Delay(10);
    HAL_GPIO_WritePin(GPIOA,LD2_Pin, GPIO_PIN_RESET);
    HAL_Delay(490);

  /* USER CODE END WHILE */
```

Compile the updated code and download it to the Nucleo board.

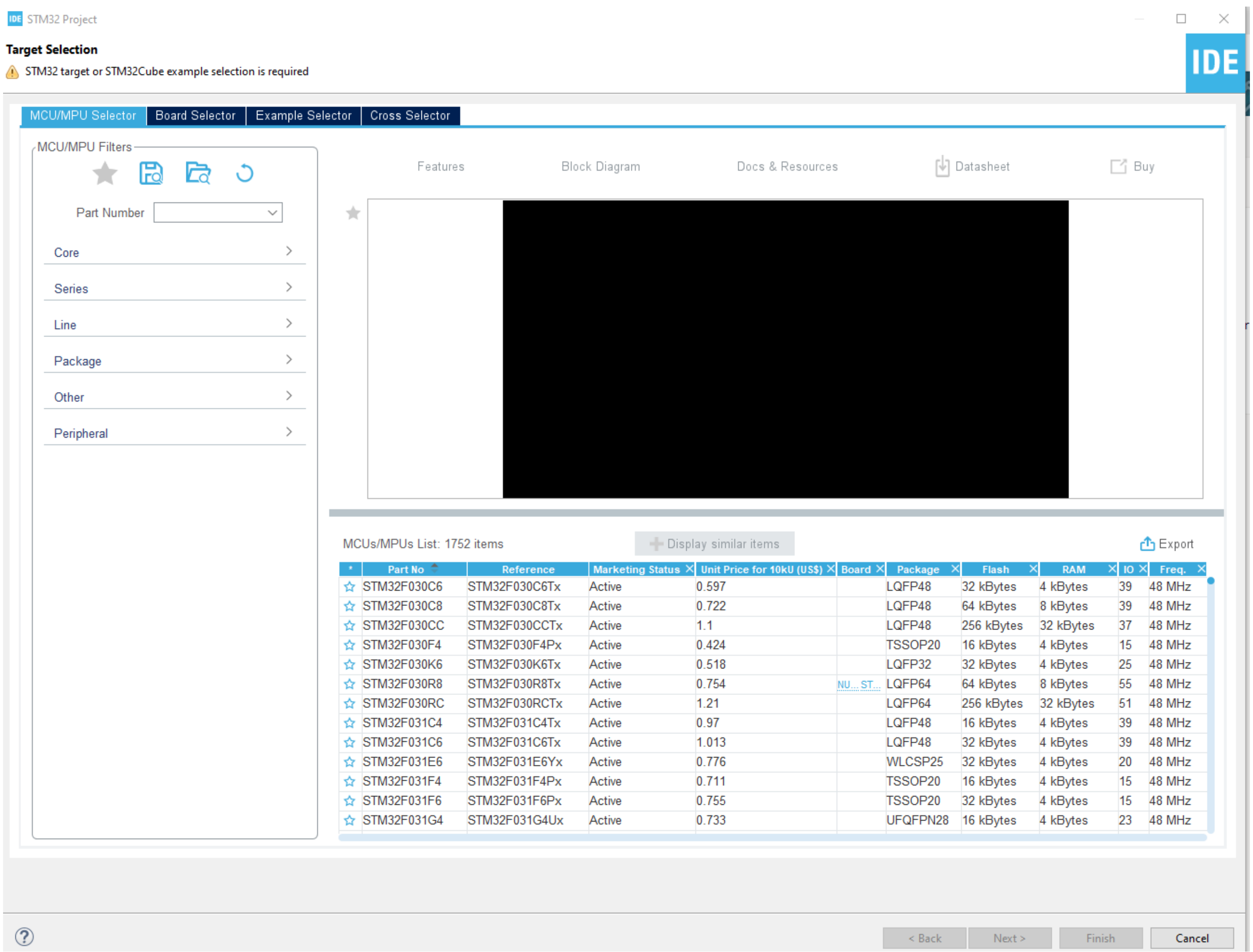
The LD2 will flash at the same 2 Hz rate but the amount of OFF time is significantly increased (ON time decreased).

For this version the Duty Cycle has been changed to a 10/500 ratio or 2%.

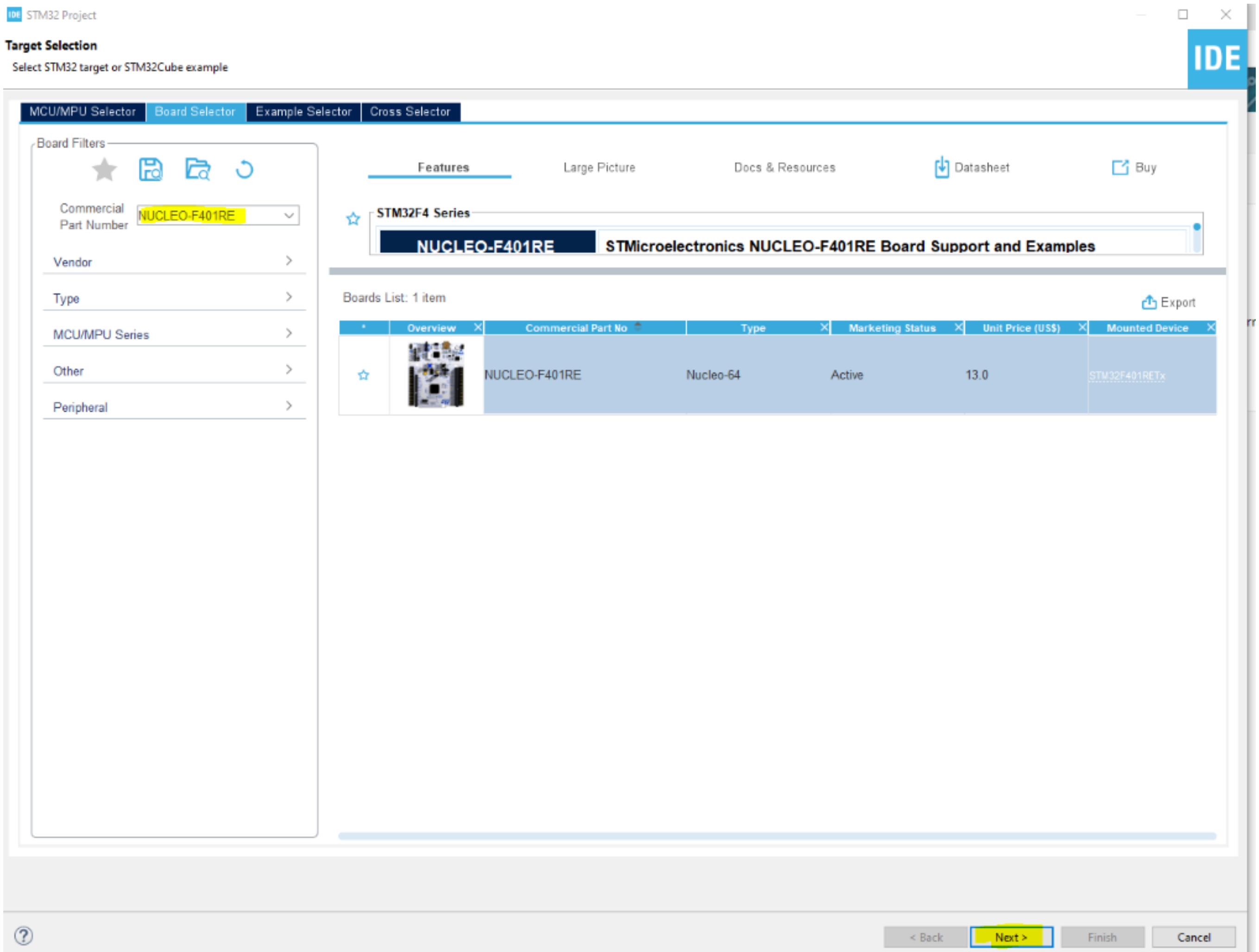
## SECOND EXERCISE: Timer_Blink Test – with a Timer Peripheral

In the STM32CubeIDE go to the FILE Tab and select NEW / STM32 Project.

A new popup window appears that requests the MCU for the IDE project:



For this course, choose the Board Selector tab. Then a new window appears:



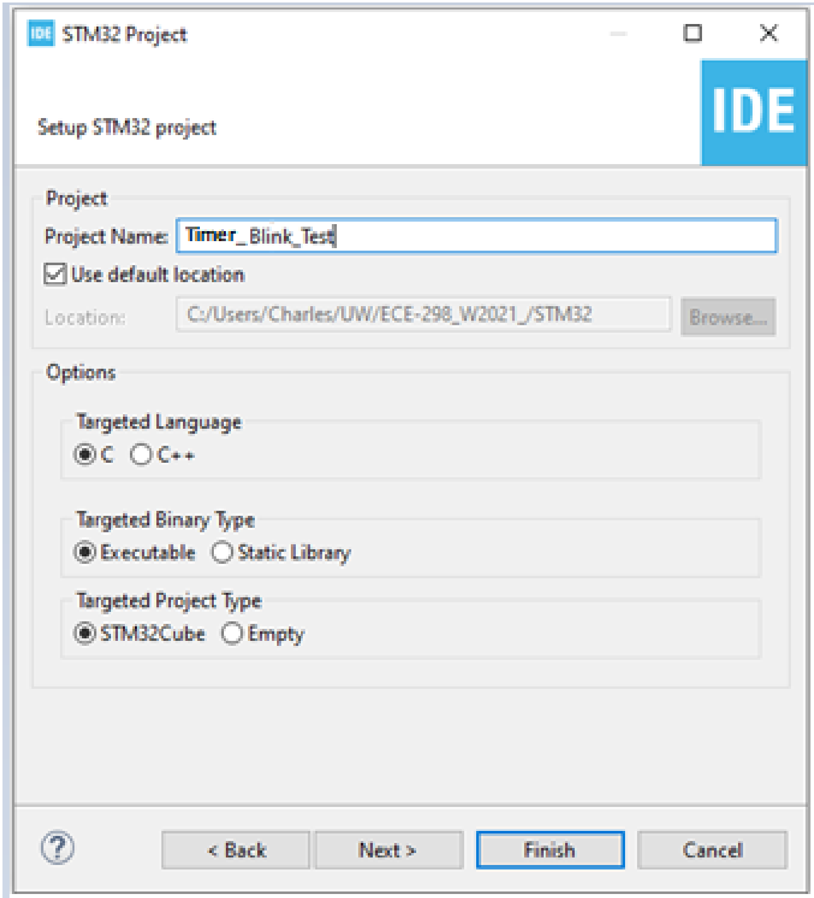

Depending on the Nucleo Dev board that was purchased from the W Store, you may have a Nucleo-F401RE or a Nucleo-F411RE type of board. Either of these boards is very suitable for this course. But the 401-based download files can't be used on 411-based Nucleo boards and vice-versa.

Type (NUCLEO-F401RE or NUCLEO-F411RE) value in the TYPE field of the board you are using inside the "Commercial Part Number" field in the Board Filters area.

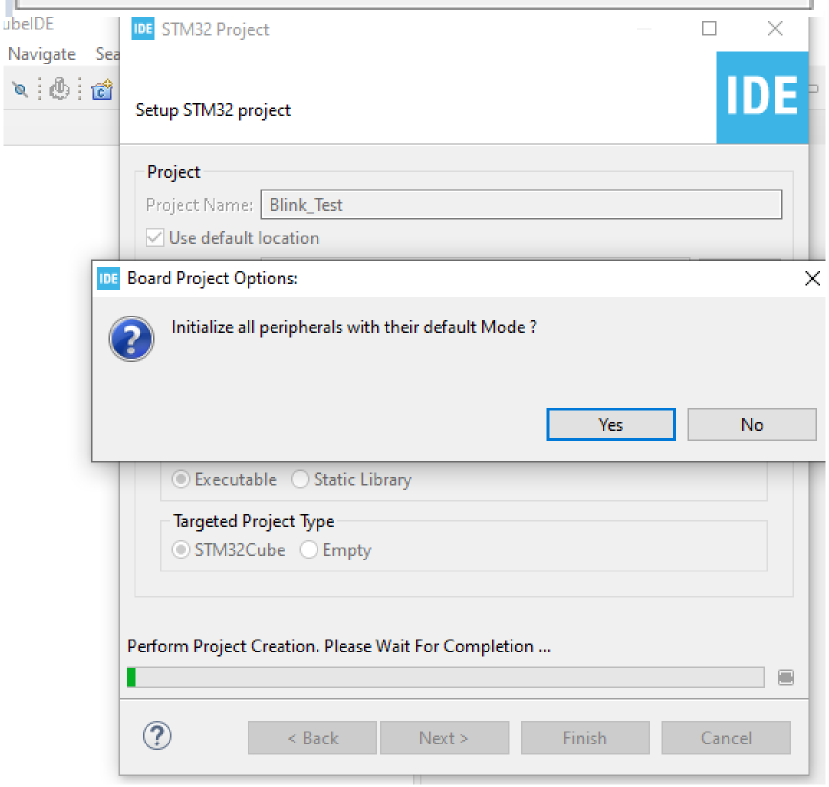
Click the "Next" button at the bottom.

## Naming the Project:

A new project based on your board configuration will be defined. Name this project “Timer_Blink_Test”: (again…no spaces please)



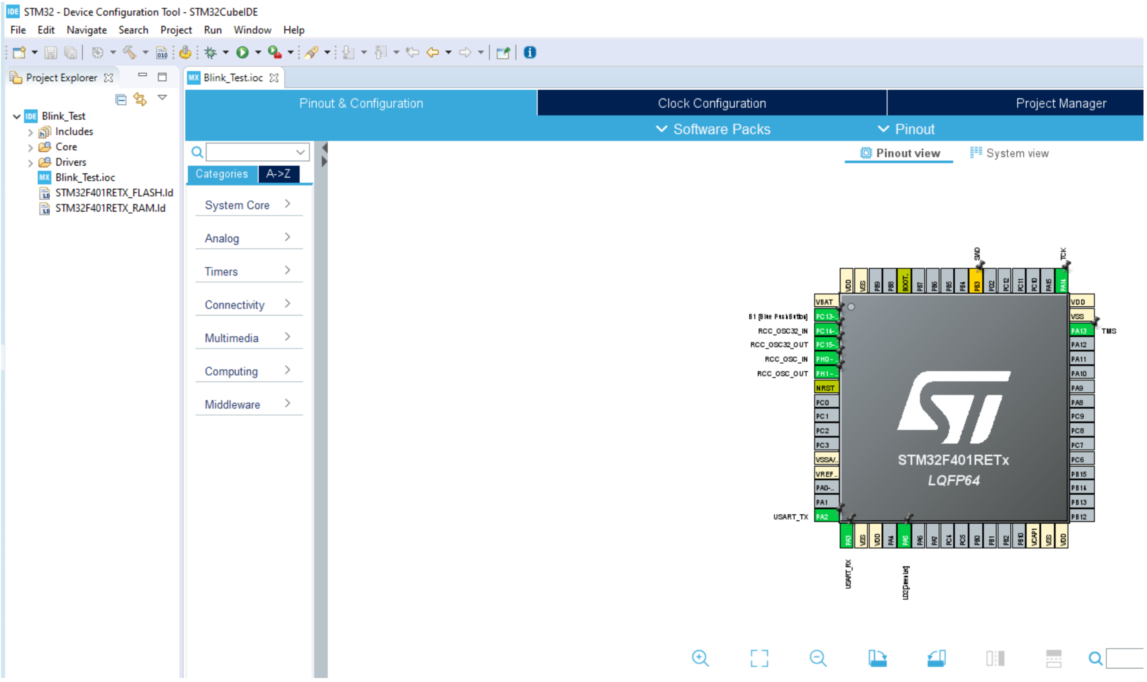

Click “Finish”.




Then a request comes to Initialize all peripherals with their Default mode….. CLICK YES.. This sets up the resources that are reserved for the Nucleo board operations with the host computer (running STM32CubeIDE).

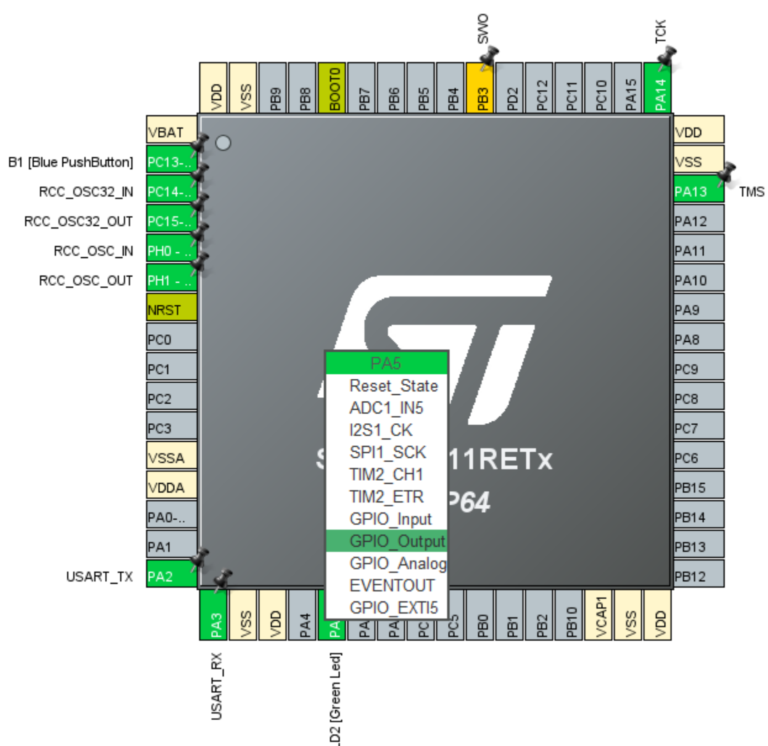
THIS IS NECESSARY FOR ALL PROJECTS IN THIS COURSE.

The MCU Configuration Window becomes visible. It shows the pinout of the MCU on the Nucleo-F401/411RE development board:




We want to use a Timer for this project, that can DRIVE the PA5 pin so that LD2 will blink under Timer control.

But we also need to change the connection to the PA5 pin to a Timer channel. Click on the PA5 (LD2) pin at the bottom and review the possible MCU Peripheral connections for this pin.

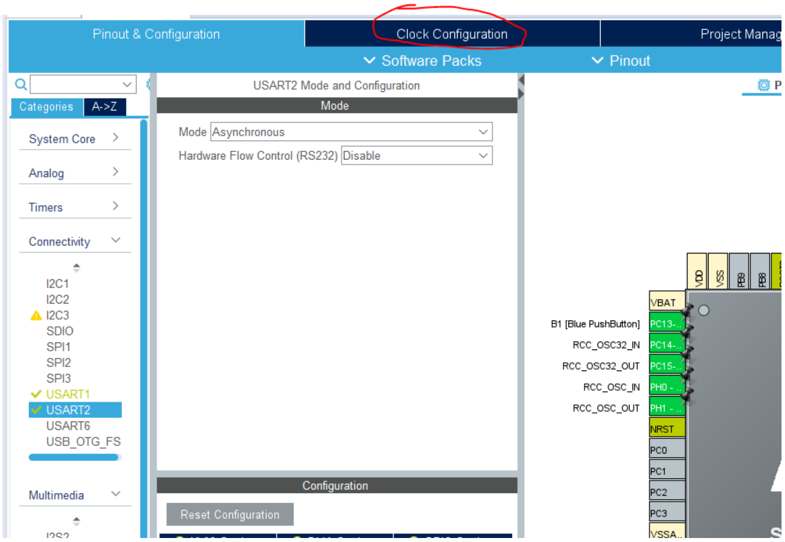


**Note that the TIMER2 Channel 1 can connect to this pin**.

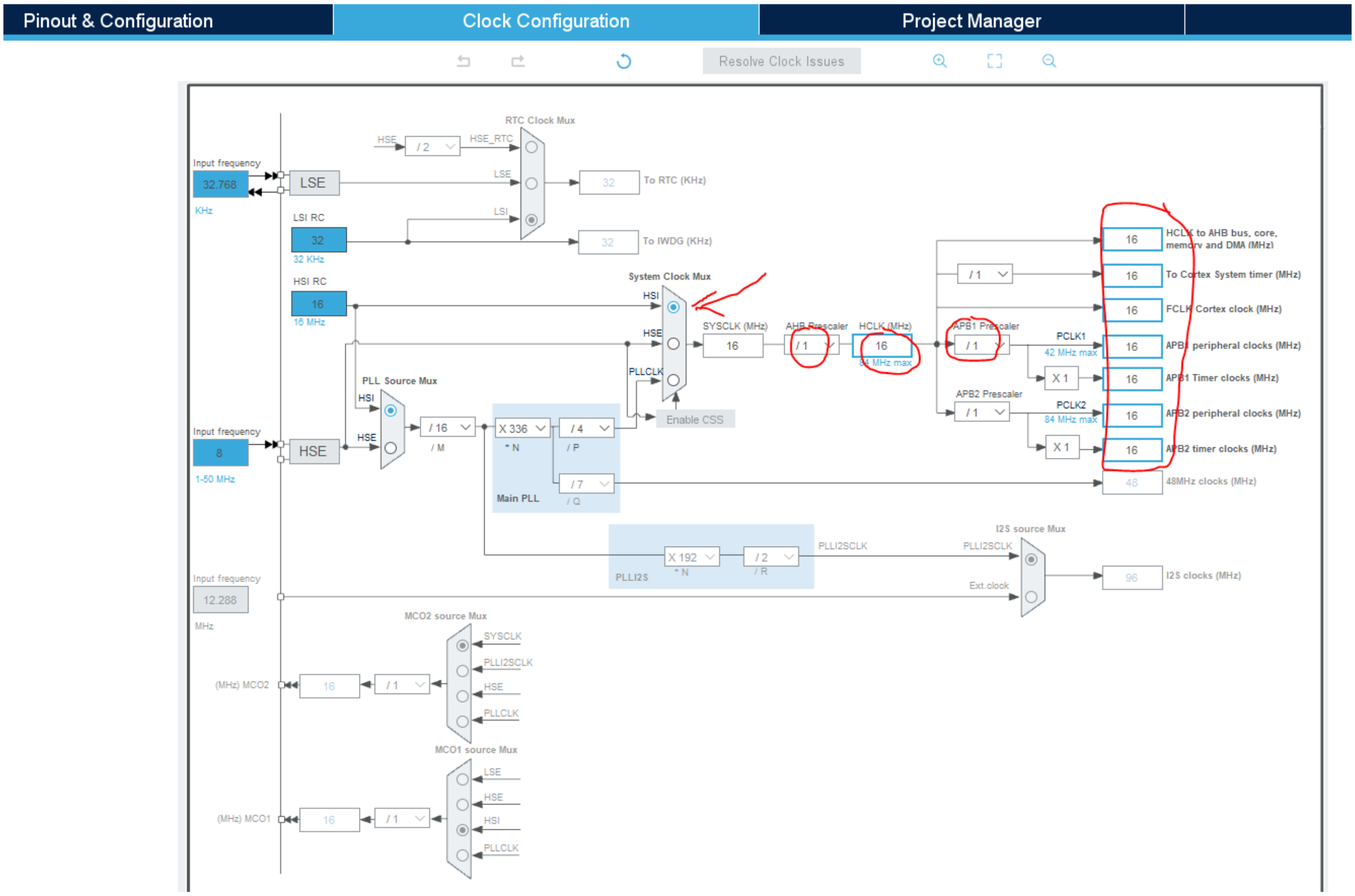
Let's set the Clock tree that we will be using for this NUCLEO PROJECT (Timer_Blink_test) like was done earlier.

## Clock Configuration

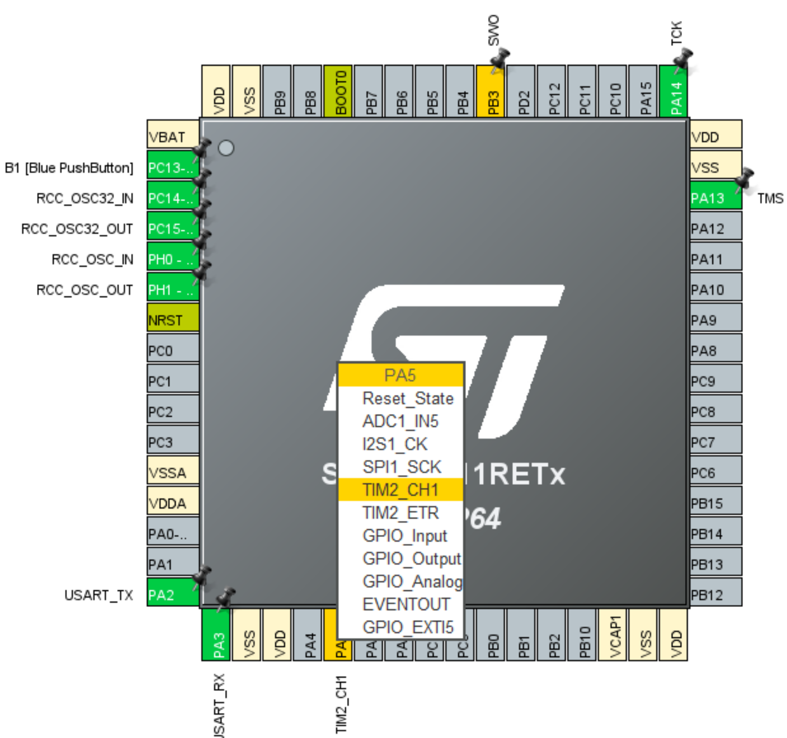
Choose the Clock Configuration Tab (circled in red below):



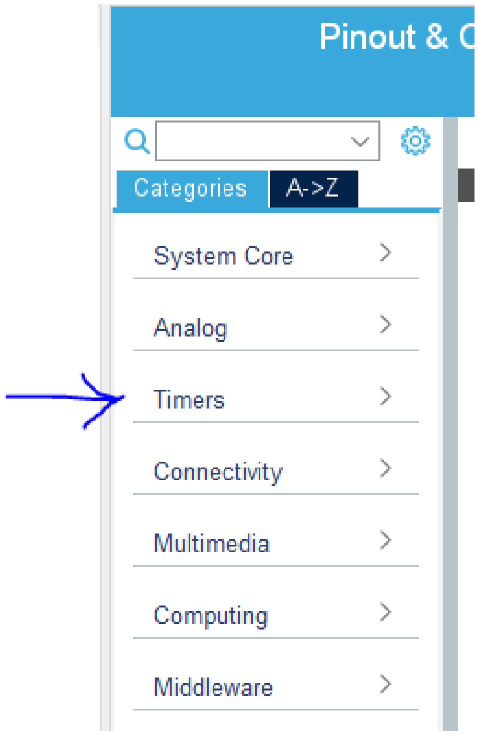

Like was done before, there is **one** clock configuration that will be used for all projects with our MCU in this course. This restriction simplifies the wide range of possibilities, for our class, to something manageable. We will **not** be using the PLL circuits. The Core Clock (HCLK) and all peripherals will be using a common 16MHz clock source. Refer to the diagram below and make sure that all of the red-circled parameters are configured for 16 MHz.

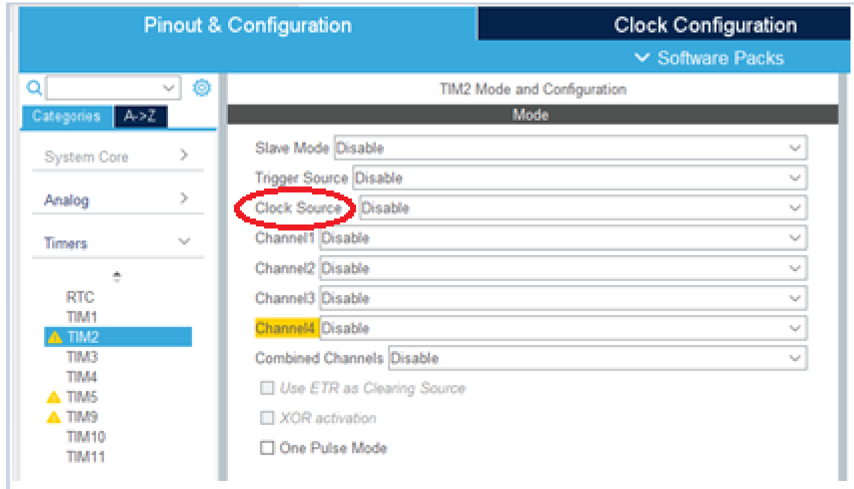



Returning to the pin PA5, the pin function is to be changed to the TIM2_CH1 connection.

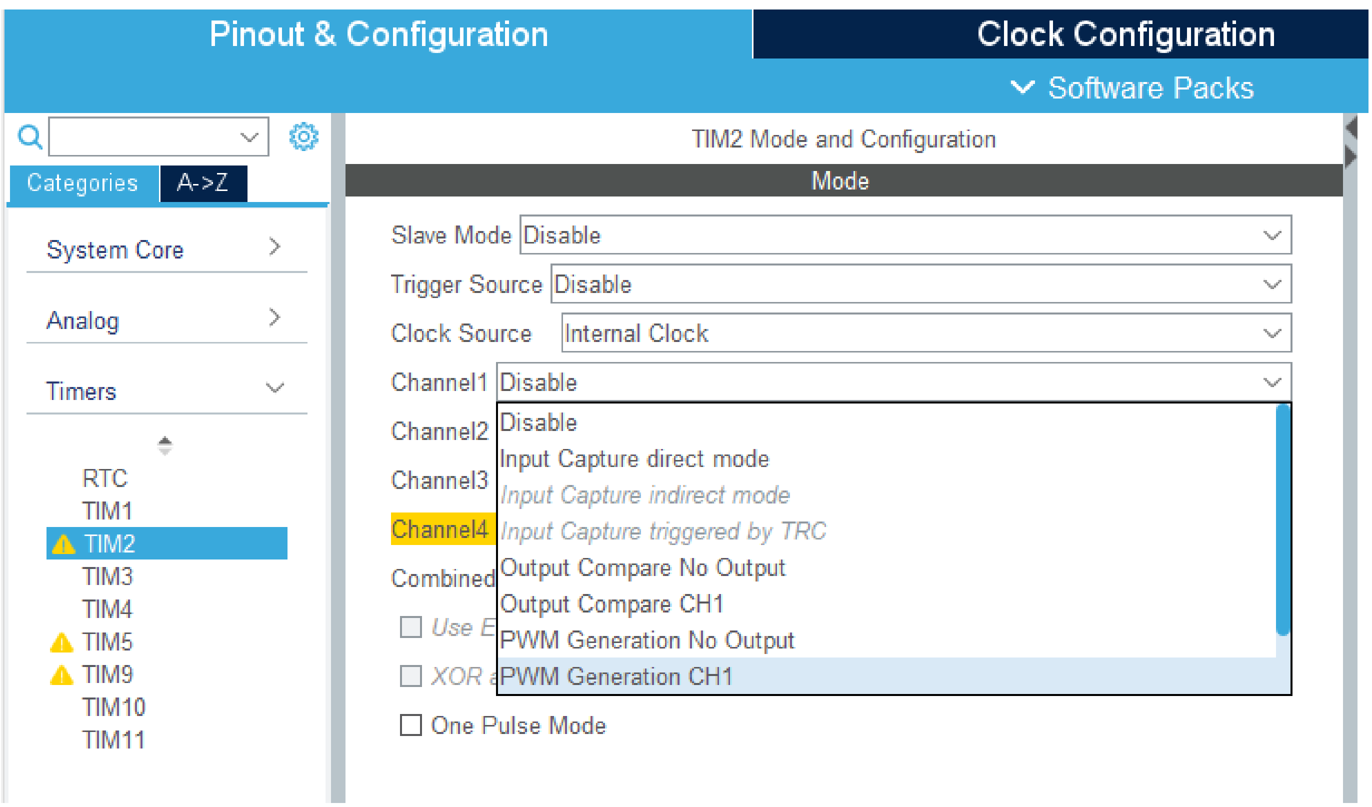


The PA5 pin turns YELLOW because the TIMER2 function is not set set up and coupled to PA5.



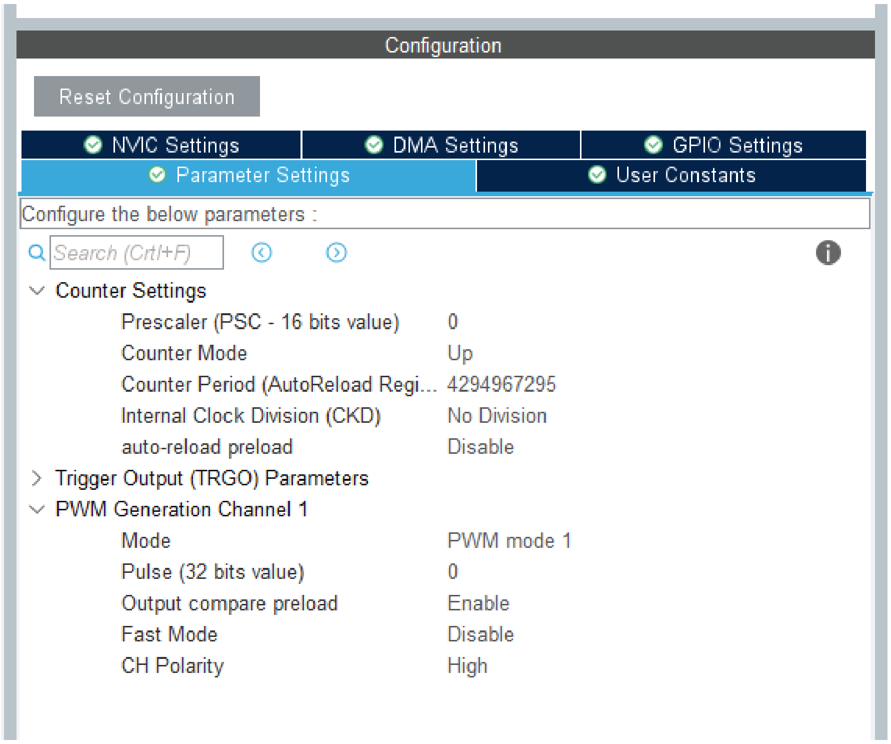




The TIMER 2 Peripheral comes up. It shows that the Channel 4 connection is NOT available (in Yellow) for use. But the other channels are usable. In the clock Source field select the Internal Clock option. This will then become a 16 MHz clock.



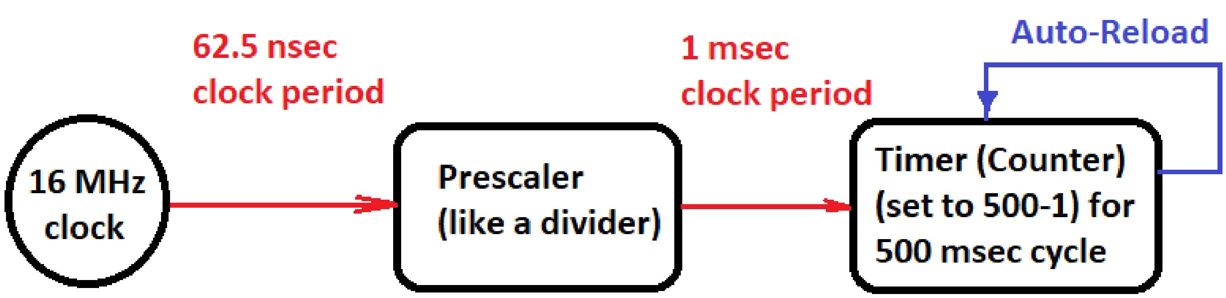

In the Channel 1 field select the PWM Generation CH1 option.

Now the Timer 2 Configuration options become visible.




The functionality that we are after now, is to create a regular repeating signal that has a 500 msec period with a specific constant pulse duration in each cycle. This will establish a Duty Cycle.

A block diagram will be useful at this point:




Recall that the Blink Test runs on a 500 msec period. It would be convenient if a 1 msec clock signal could be created and then a counter would count a number of 1 msec clocks up to 500 cycles.

Because digital counters always start with a value of zero for the first cycle, we can insert the count limit quantity as 500-1 (the counters will count between 0 and 499). After reaching the limit the counter starts over.

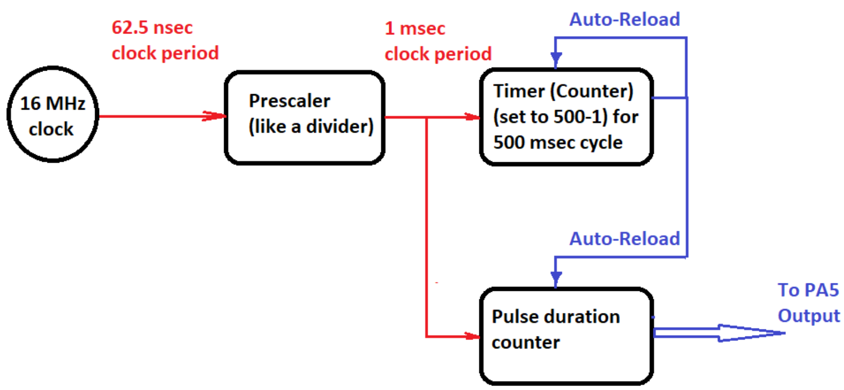
To get the 1 msec period into the counter we need a value for PRESCALING the 16 MHz clock before sending it to the counter clock input.

We want a 1 msec clock signal going to the Timer Counter. The 16 MHz clock has a 62.5 nanosecond period. Therefore, the Pre-scaler value should be set for 1 msec. Thus, the Pre-scaler field should be 16000-1 (counting from 0 and 15999)

The Counter will be set for an AUTO-RELOAD value of 500-1.

The Auto-reload preload field will be changed to ENABLED.

Another part of the Timer functionality is for controlling the Pulse width for each cycle.

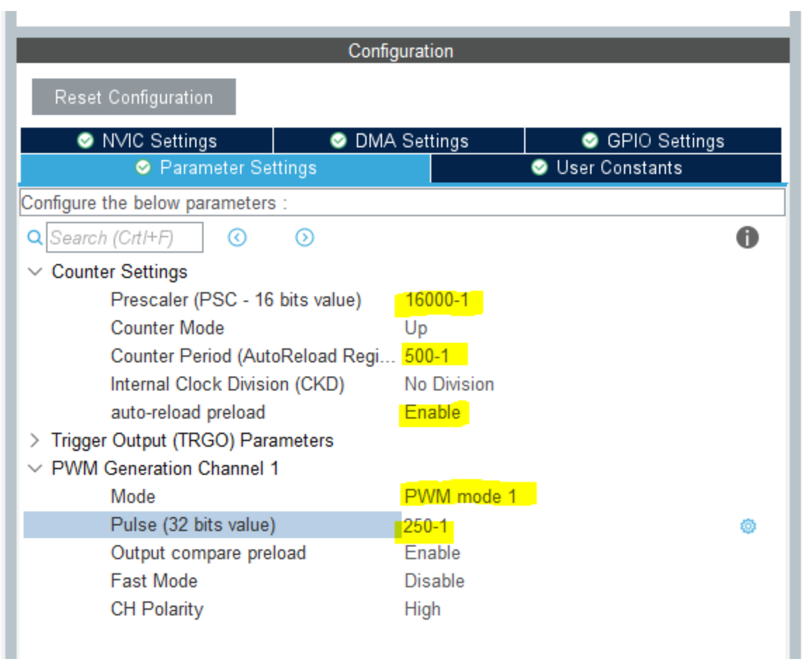



WHEN THE AUTO-RELOAD OCCURS THE TIMER COUNTER and the PULSE DURATION COUNTER RESET to ZERO. The PA5 is set to "HIGH" at this time and is reset to "LOW" when the Pulse Duration Counter reaches its count value.

Going to PWM Generation Channel 1, we can set the Mode to PWM_Mode 1

The Pulse value to 250-1. This will give us the 50% duty cycle of a 500 msec period for driving LD2 through the PA5 pin.

Thus, we have:




After these settings are made Click on the Configuration Button (Looks like a GEAR).

Now a new main.c file will be created for the Timer_Blink_Test project. It is opened for you and the auto-generated Initialization code for the MCU's Configured peripherals can be seen below.

```
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART2_UART_Init();
  MX_TIM2_Init();
  /* USER CODE BEGIN 2 */

  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */

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

Further in the code, the insertion of the AUTO-generated code for the Timer setup can be seen:

```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};
  TIM_OC_InitTypeDef sConfigOC = {0};

  /* USER CODE BEGIN TIM2_Init 1 */

  /* USER CODE END TIM2_Init 1 */
  htim2.Instance = TIM2;
  htim2.Init.Prescaler = 16000-1;
  htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim2.Init.Period = 500-1;
  htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
  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();
  }
  if (HAL_TIM_PWM_Init(&htim2) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  sConfigOC.OCMode = TIM_OCMODE_PWM1;
  sConfigOC.Pulse = 250-1;
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_1) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM2_Init 2 */

  /* USER CODE END TIM2_Init 2 */
  HAL_TIM_MspPostInit(&htim2);
```

So now the Timer Peripheral is set up. BUT we still must add the code in the main.c file, that tells the Timer hardware when to start etc. For that, we use another HAL command. We insert this in the code section that is just before the main while loop. Note the syntax.

Also note that in the “while (1) loop”, there is no C- coding necessary.

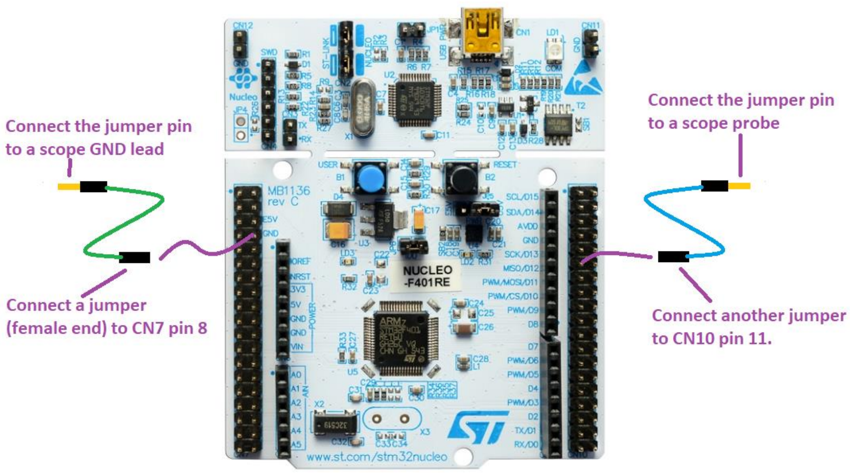
```c
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART2_UART_Init();
  MX_TIM2_Init();
  /* USER CODE BEGIN 2 */
  HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */

```

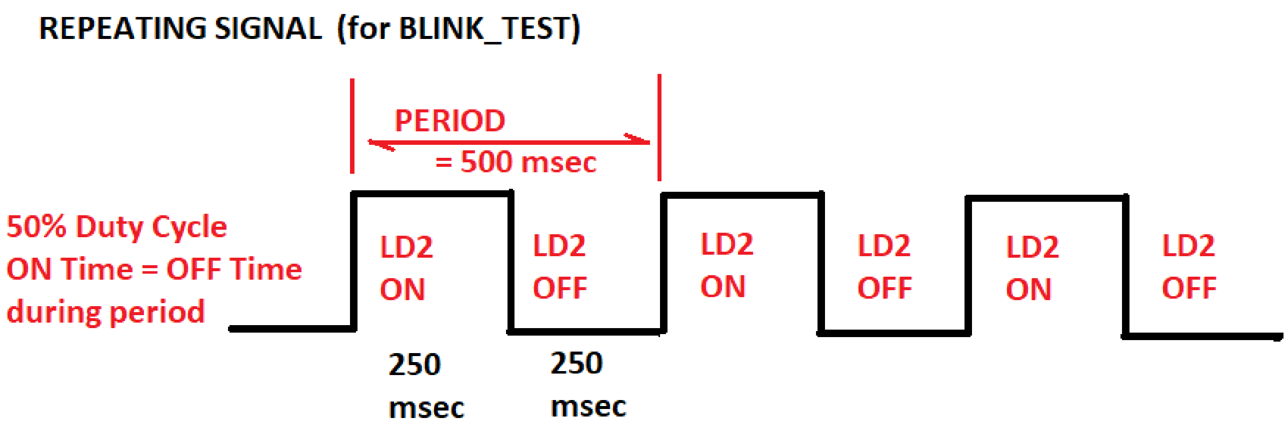
Save this main.c file now and perform a code compile by clicking on the “HAMMER” icon. Then download the executable file to the Nculeo Dev board (Green Arrow icon).

You will see on the Nucleo board that the same behaviour for the PA5 signal to LD2 can be accomplished by a Timer peripheral on the MCU device after the MCU has done the initialization. This frees up the MCU core to do other, more important things during is its processing time. This shows the power of using peripherals in an MCU device.

MANUAL FOR LAB B1 – Rev 1.4

**Using an oscilloscope**, observe the timing of the digital waveform that is driving LD2. Below is the hookup:




In each Timer peripheral, there are shadow registers that can be accessed by the MCU code while the Timer hardware is running. The shadow registers will have their content transferred to the Timer at the beginning of the next auto-reload cycle.

There are three shadow registers that can be accessed. These are:

1) Pre-scaler (controls the divider value for scaling the system clock to the Timer counter clock input)
2) ARR (Auto-Reload Register controls the Timer Counter period)
3) CCR1 (Capture Control Register for PWM Channel 1 controls the Pulse width)

For the Timer that is being used for this project (Timer2), the Timer2 Shadow registers can be accessed respectively, in the main.c file by using the following commands:

TIM2-> PSC = 16000-1; 🡐 sets a counter clock period to 1 msec

TIM2-> AAR = 500-1; 🡐 sets a counter maximum number of cycles to 500 (0 through to 499)

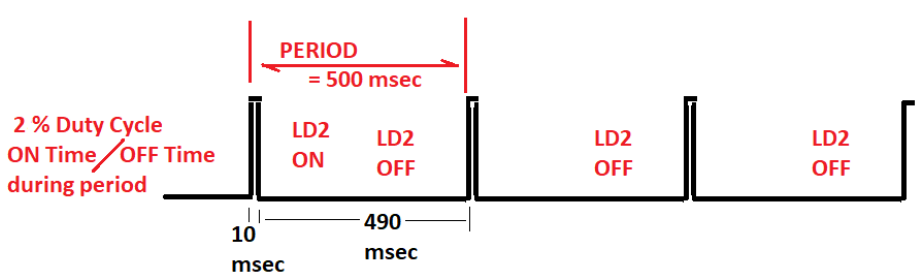
TIM2->CCR1 = 250-1;🡐 sets the output pulse width (within the period (of 500 cycles) to be 250 cycles.

```c
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART2_UART_Init();
  MX_TIM2_Init();
  /* USER CODE BEGIN 2 */
  HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
  TIM2->PSC = 16000-1;
  TIM2->ARR = 500-1;
  TIM2->CCR1 = 10-1;
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */

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

Thus, with the Timer being initially configured, the operating behaviour can be changed by using the Shadow registers in the main.c file.

Now here is something to try shown in the example above. Changing the CCR1 register value to 10-1 and recompiling /downloading to the Nucleo Dev board, you will be able to see a difference in the LED behaviour **and on the oscilloscope**.

**REPEATING SIGNAL (for BLINK_TEST)**



## THIRD EXERCISE: Pulse Width Modulation Blink Test – using a Timer Peripheral

In the STM32CubeIDE go to the FILE Tab and select NEW / STM32 Project. Name the project as PWM_Blink_Test.

Go through the steps as before (Selecting the Nucleo Board type; Initializing all Nucleo peripherals with their default Mode; Setting the PA5 pin for TIM2 Ch1; Setting all clocks to 16 MHz; Selecting the Timers category and choosing Timer2; Setting up the Timer 2 peripheral for internal clock; Choosing the Timer 2 CH1 for PWM mode; setting some initial peripheral register values as before; Run the configuration auto-coding (GEAR icon); Add the HAL Command (HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1) to the main.c file etc.

Now the MCU can be made to change the duty cycle over time. Do this by adding some varying PWM control by doing the following:

Set the Timer2 PSC register to 0 (for no scaling). Set the Timer2 ARR register to 60000 (for a very long blink period)

```
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*

  /* Reset of all peripherals, Initializes the Flash interface and the Systick.
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART2_UART_Init();
  MX_TIM2_Init();
  /* USER CODE BEGIN 2 */
  HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
  TIM2->PSC = 0;
  TIM2->ARR = 60000;
  TIM2->CCR1 = 0;
  int TIM2_Ch1_DCVAL = 0;

  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
      while(TIM2_Ch1_DCVAL < 60000) {
          TIM2_Ch1_DCVAL += (50);
          TIM2->CCR1 = TIM2_Ch1_DCVAL;
          HAL_Delay(1);
      }
      while(TIM2_Ch1_DCVAL > 0) {
          TIM2_Ch1_DCVAL -= (50);
          TIM2->CCR1 = TIM2_Ch1_DCVAL;
          HAL_Delay(1);
      }

    /* USER CODE END WHILE */
```

Add an integer variable called Tim2_Ch1_DCVAL and set it initially to 0. Add an increment value of 50 to be added every 1 msec. Have the Tim2_Ch1_DCVAL compared against a limit of 60000. When that is reached, decrement the Tim2_Ch1_DCVAL by 50 every 1 msec until the Tim2_Ch1_DCVAL reaches 0. Then repeat again.

On the Nucleo board, note how the behaviour of the LD2 changes. The brightness is proportional to the duty cycle.

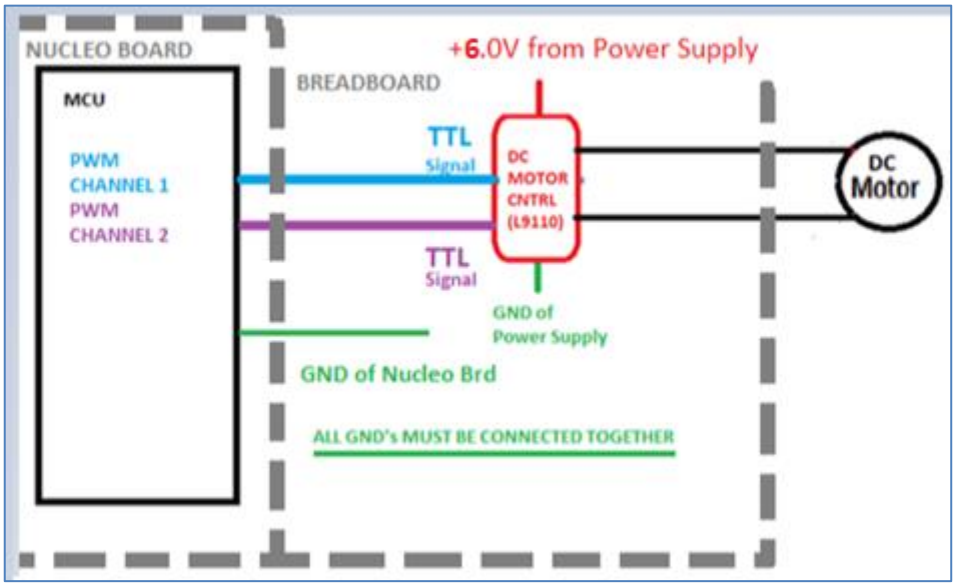
**You must also observe the behaviour on the oscilloscope so that you can see what happens with the signal in time.**

## FOURTH EXERCISE: ADDING A PWM-CONTROLLED BRUSHED-DC MOTOR

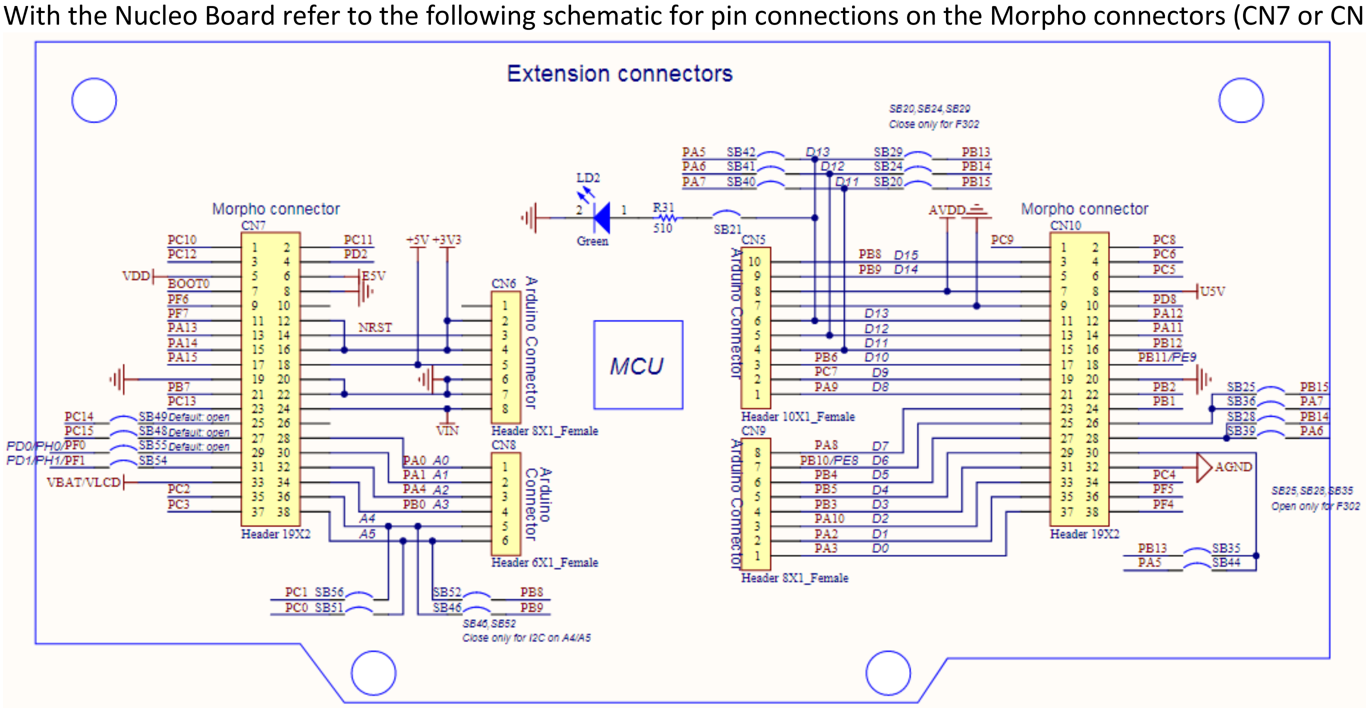
After getting some preliminary experience with creating a PWM signal, it is appropriate to move on to developing a PWM-Controlled external device. The first device to be controlled will be a Brushed-DC Motor and a PWM Motor Controller.

**You Will be making the following circuit:**

**THE L9110 (DC Motor Controller from your kit) will be used with the DC Brushed Motor:** This Motor Controller and DC Motor MUST be energized from a separate +6.0V power source (the bench power supply). The MCU PWM-controlled L9110 will regulate the power going to the DC Motor via PWM inputs from the Nucleo PWM channel pins.




With the Nucleo Board refer to the following schematic for pin connections on the Morpho connectors (CN7 or CN10)



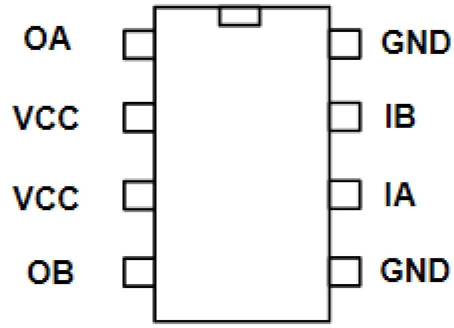

On the breadboard, place the L9110 (DC Motor Controller from your Reservoir System kit)) will be used to interface to this motor. The controller accepts a TTL-level PWM input. Look up the connection details for the L9110 in its datasheet.

There is one detail in the datasheet to clarify. To make the motor move in one direction, the input IB will be set to ZERO and a PWM signal is sent to the IA input. To make it go in the other direction, the input IA will be set to ZERO and a PWM signal to IB input.

Pin definitions:

| No. | Symbol | Function |
|---|---|---|
| 1 | OA | A road output pin |
| 2 | VCC | Supply Voltage |
| 3 | VCC | Supply Voltage |
| 4 | OB | B output pin |
| 5 | GND | Ground |
| 6 | IA | A road input pin |
| 7 | IB | B input pin |
| 8 | GND | Ground |

Device pins Figure




FOR THIS DEVICE, BOTH VCC PINS WILL BE COONECTED TO THE BENCH POWER SUPPLY +6.0V POWER.

On the breadboard, connect the Nucleo Morpho connector pin to which the MCU pin **PA6** is connected (use the schematic on Learn for the Nucleo Morpho connectors – CN10 – pin 13) to the L9110 IA input This is the TIM3 CH1 connection.

To the output pin for TIM3 Ch3 PWM output connected on the Morpho connector, connect the L9110 IB pin. Connect the motor wires to the L9110 OA and OB pins. From the bench power supply, connect GND to the GND pins and +6.0V to the L9110 VCC pins. Make sure that you also connect the Power Supply GND to the Nucleo board GND (like CN7 pin 8 or pin 19).

## For the Nucleo MCU with the STM32CubeIDE, Use a Timer to Create PWM for driving the Brushed DC Motor

Another project can be created to try out the control of a Brushed (ie: commutating) DC Motor.

1) Create a NEW STM32 project as before etc.
2) Go to the project.ioc file to establish which pins and peripherals you wish to use.
3) Set the clock tree for 16 MHz for all clocks
4) Use another Timer (say Timer3) and configure it to use internal clocking and CH1 & CH3 as PWM generators.
5) In the Parameter Settings: Set the Pre-scaler to 16-1; Counter Period to 2000-1; Auto-reload to ENABLE; Set the initial CH1 Pulse Width to 1200-1. Set the initial CH3 Pulse Width to 0. With these settings the PWM CH1 will have a period of 2000 usec and a default pulse width of 1200 usec. The CH3 PWM output will initially just be 0.

Choose pin **PA6** as a GPIO output for this PWM CH1 signal. Choose an appropriate second output pin for the CH3 PWM signal.

After running the Configurator (yellow gear) to generate the Auto-code, enter code into the main.c file (User Code Section 2) like the following:

```
MX_TIM3_Init();
/* USER CODE BEGIN 2 */
HAL_TIM_Base_Init(&htim3);
HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_1);
HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_3);

/* USER CODE END 2 */
```

You can use a similar setup as in the PWM examples above to increase the PWM (and thereby increase the DC Motor speed) until an upper limit is reached. Then gradually decease the PWM to a lower limit.

Add another loop to change the direction of rotation of the DC Motor. SET first Timer CHANNEL to create a PWM signal but set the second Timer Channel to a PWM of ZERO.

You should be able to increase the motor speed, then decrease the motor sped and then repeat the operation but with the motor running in the opposite direction.

PLEASE KEEP IN MIND:

Some devices have a LOT of mechanical inertia. Motors take a longer time to react to stimulus.

For the above examples of increasing/decreasing the Pulse widths for your PWM control of the motor, Allow a HAL DELAY of 5000 (5000msec) between loop passes. This will give the motor time (5 seconds) to change speed between pulse width increments/decrements.

## FIFTH EXERCISE: Using a UART/USART Peripheral

UARTs and USARTs are very similar and the MCU has peripherals that can implement either kind of function.

These peripherals are used to communicate messages SERIALLY with a Host computer or another UART/USART-based device.

The USART uses two wires for the data (TxD , RxD) for communications in each direction. For the USART another wire is added in each direction. This is the communications clock.

The UART is a similar serial communications device but only two wires are used (TxD, RxD) only (no clocks). UART stands for Universal Asynchronous Receiver Transmitter.

Because no clocks are used for the UART version it runs slower than the USART. But the trade-off is that it saves two pins when compared to the USART.

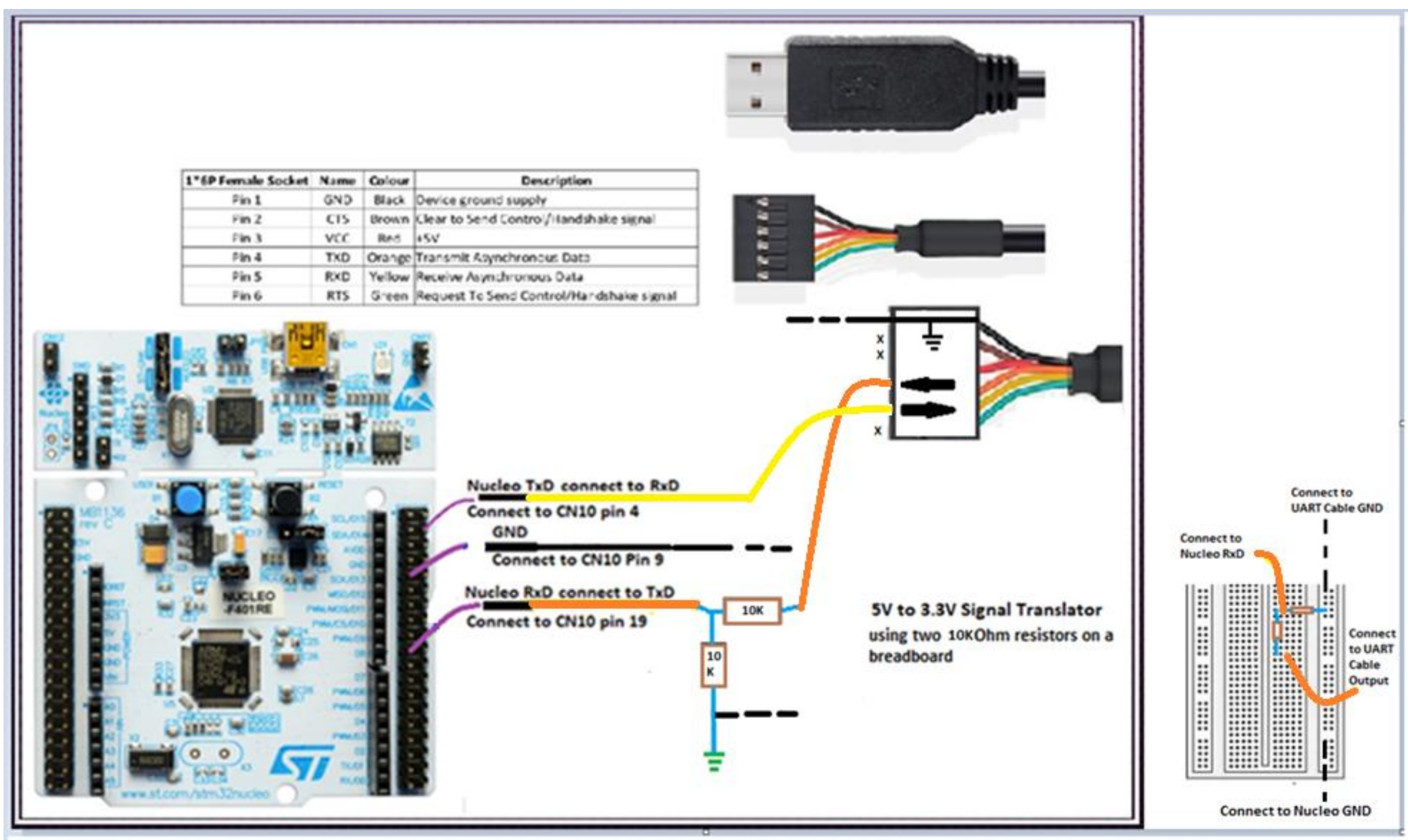
**ON YOUR BREADBOARD**:

A UART/USB data link connection must be made with the Lab Computer. A UART/USB Adapter cable is used for this. This is supplied at the computer workstation. The software driver for this function is loaded on the Lab Computer already.

The output from the USB/UART Cable is 5 volts. This is too high for the MCU input pin which operates with 3.3V signal levels. More about this topic will be discussed in Lab B2.

To translate the 5V signal to a 3.3V signal a small breadboard circuit will be used initially. Using your own breadboard, you may install two **10K** Ohm resistors to make a voltage divider.

The jumper wires end must connect to the Nucleo board for GND and the signals (Tx,Rx) must be connected to the Rx, Tx pins used for USART6.



MANUAL FOR LAB B1 – Rev 1.4

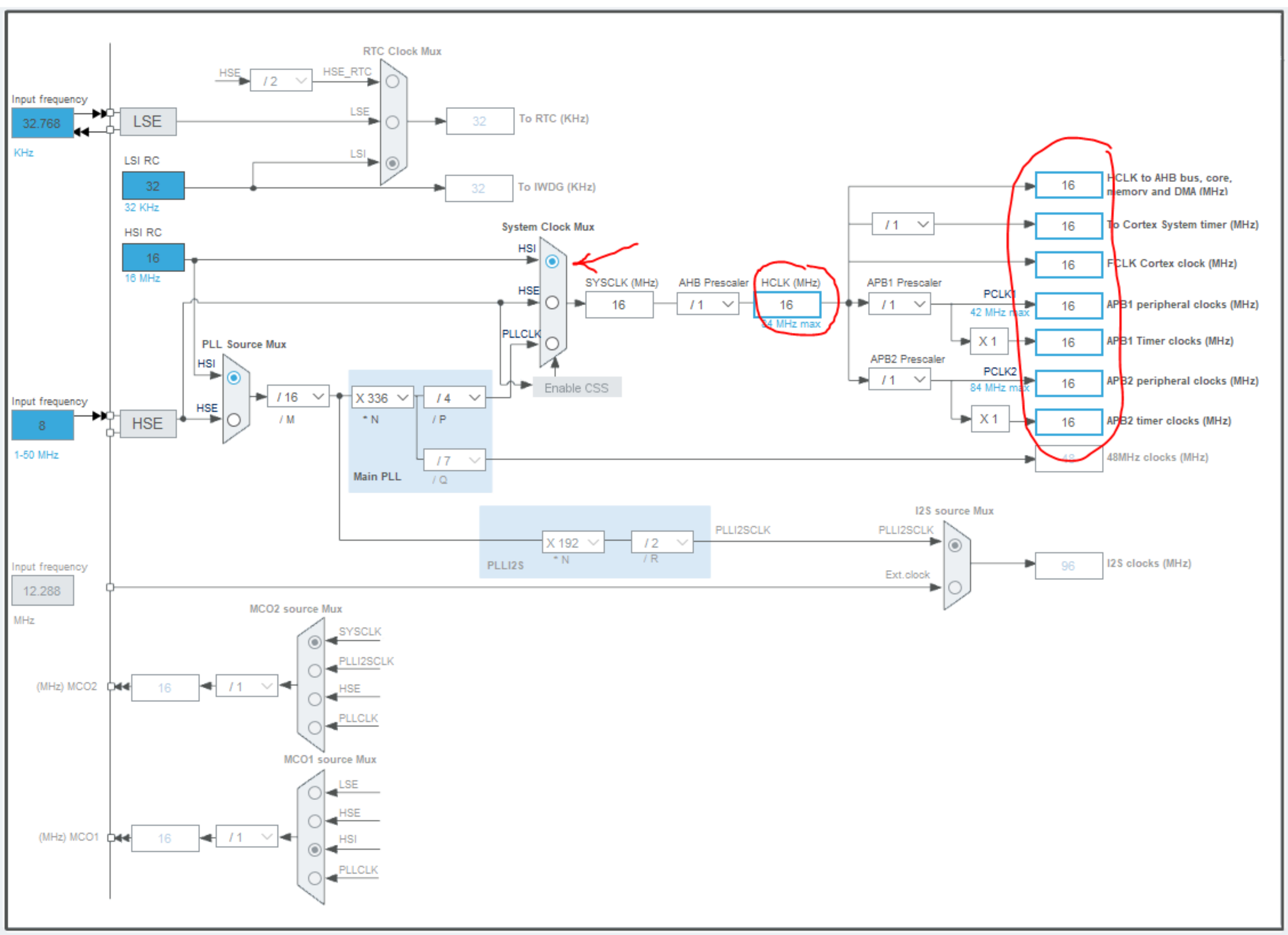
The USB/UART Cable at Pin 5 must connect to the FIRST resistor wire lead on the breadboard.

The SECOND resistor (see below) must have its second wire lead connected to GND. This GND point must also then be connected to the USB/UART cable at Pin 1. The GND must also be connected to the Nucleo board at CN10 Pin 9. This arrangement gets all of the GROUND connections together for the Nucleo, the USB/UART Cable and the voltage- divider circuit.

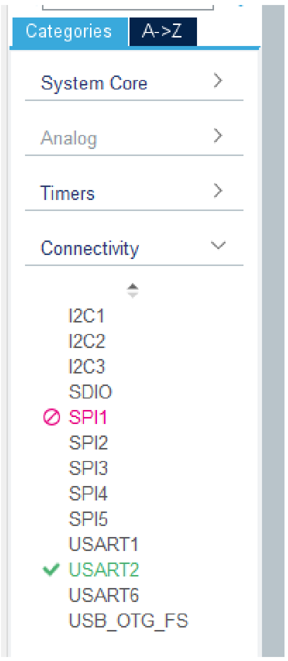
The connection between the FIRST and SECOND resistors must connect to the Nucleo board at CN10 Pin19.

The output from the Nucleo board (at CN10 Pin4) can connect DIRECTLY to the USB/UART cable at Pin 4.

1) Create a NEW STM32 project as before etc. Name the project as UART_Test.
2) Go to the project.ioc file to establish which pins and peripherals you wish to use.
3) Set the clock tree for 16 MHz for all clocks.

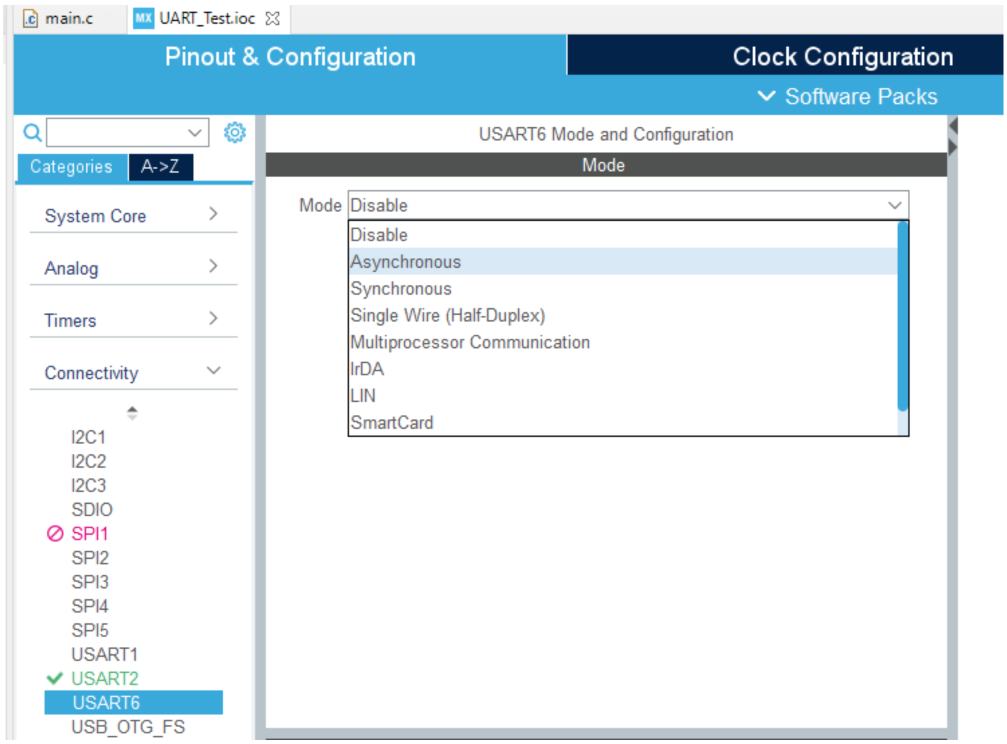


Then select the Connectivity Category.




With the default Peripherals initialized to their default values, the interfaces for the Nucleo board are reserved. SPI1 is not available. And USART2 is already in use.

Select the **USART6** peripheral and then the Asynchronous Mode option. This will make the peripheral operate like a UART instead of a USART.

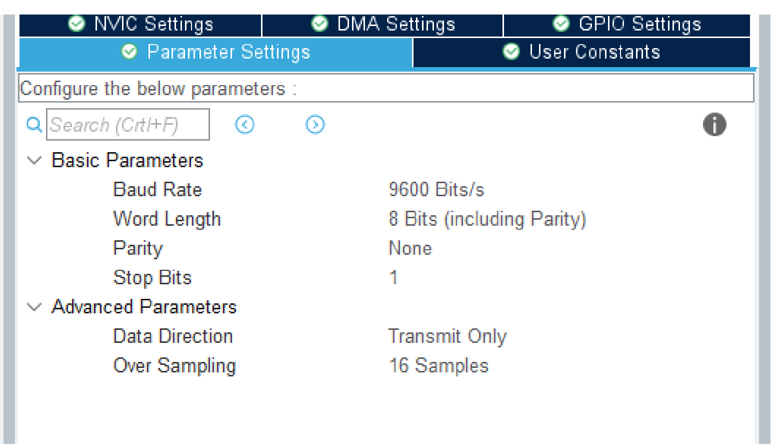



Then a Configuration window for the UART peripheral appears.

<u>Part1: Transmit Only Operation:</u>

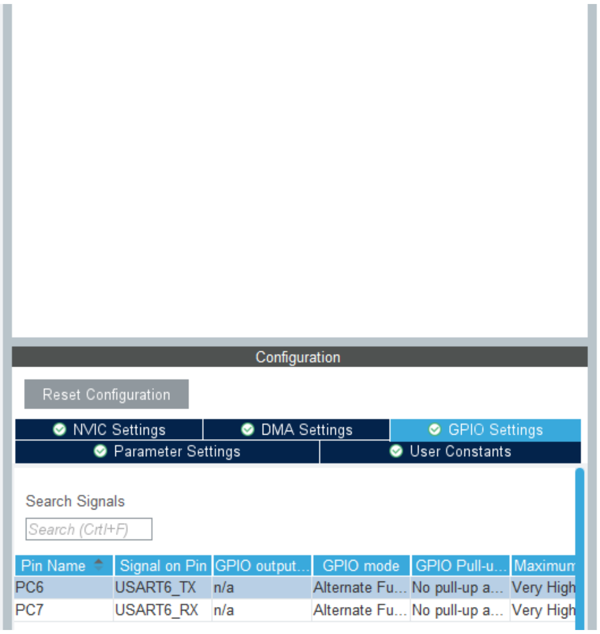
For this first test we will start experimenting **only with a TRANSMITTING** serial communications function.

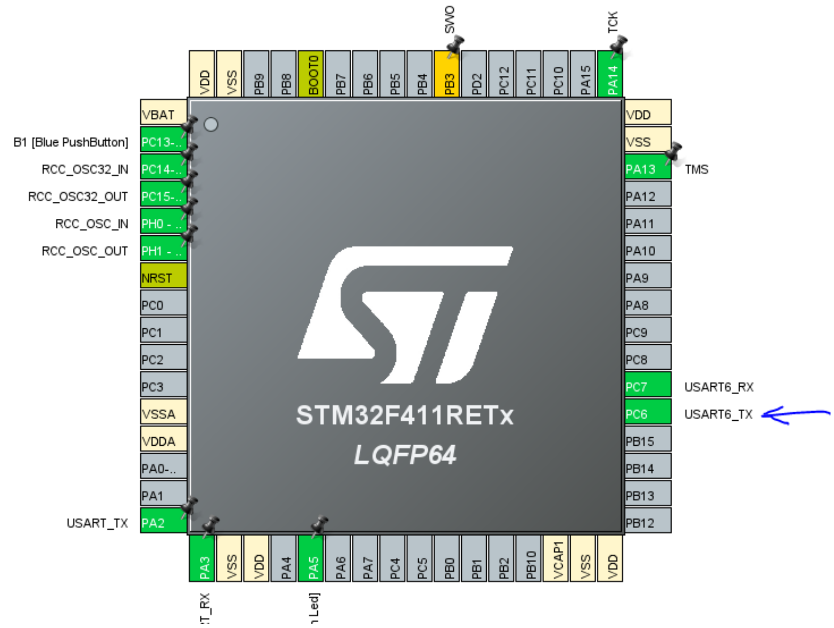
Set the Parameters to the following:




1) Baud rate (bits per second) to 9600.
2) Word Length to 8 bits (including parity).
3) Parity NONE
4) STOP BITS 1
5) Data Direction to **Transmit Only**
6) Over Sampling to 16 samples.

Click on the GPIO Settings Tab of the Configuration Window and then you can see that for this peripheral, TxD is on MCU pin PC6 and Rxd is on MCU pin PC7. Select the PC6 row and you will see the pin flashing on the MCU diagram.




Run the Configuration tool (the gear icon in the Menu bar). A new main.c file is then created and opened for the project. Take note of the following auto-generated Initialization code that was created for the configured peripheral.

```c
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART2_UART_Init();
  MX_USART6_UART_Init();
  /* USER CODE BEGIN 2 */

  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */

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

```c
/**
  * @brief USART6 Initialization Function
  * @param None
  * @retval None
  */
static void MX_USART6_UART_Init(void)
{

  /* USER CODE BEGIN USART6_Init 0 */

  /* USER CODE END USART6_Init 0 */

  /* USER CODE BEGIN USART6_Init 1 */

  /* USER CODE END USART6_Init 1 */
  huart6.Instance = USART6;
  huart6.Init.BaudRate = 9600;
  huart6.Init.WordLength = UART_WORDLENGTH_8B;
  huart6.Init.StopBits = UART_STOPBITS_1;
  huart6.Init.Parity = UART_PARITY_NONE;
  huart6.Init.Mode = UART_MODE_TX;
  huart6.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart6.Init.OverSampling = UART_OVERSAMPLING_16;
  if (HAL_UART_Init(&huart6) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART6_Init 2 */

  /* USER CODE END USART6_Init 2 */

}
```

To output C-BASED messages to the lab computer, the STRING function is used. To use the string function and the STDIO function we need to add two INCLUDE statements in the **USER CODE BEGIN Include** section.

#include <string.h>

#include <stdio.h>

In the main.c file we need to create a memory buffer that will hold a the message content before it is sent out.

```
int main(void)
{
  /* USER CODE BEGIN 1 */
    uint8_t txd_msg_buffer[64] = {0};
  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART2_UART_Init();
  MX_USART6_UART_Init();
  /* USER CODE BEGIN 2 */
  sprintf((char*)txd_msg_buffer, "HELLO\r\n");
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
   HAL_UART_Transmit(&huart6, txd_msg_buffer, strlen((char*)txd_msg_buffer), 1000);
  /* USER CODE END WHILE */
   /* USER CODE BEGIN 3 */
   }
   /* USER CODE END 3 */
}
```

Add a variable **array** of 8bit integers into the User Code Begin 1 section:

Uint8_t txd_msg_buffer[64] = {0};

To load the txd_msg_buffer we will use the sprintf command:

Sprintf((char*)txd_msg_buffer, “HELLO\r\n”);

This can be placed in the USER CODE BEGIN 2 Code section.

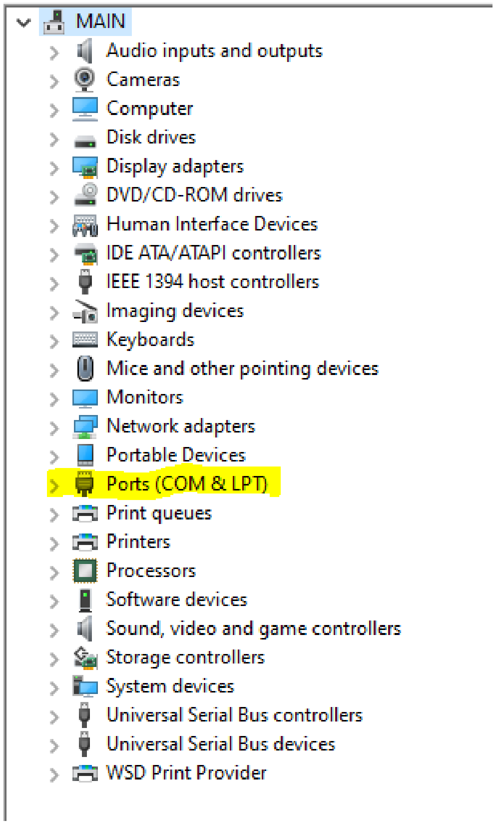
Then to transmit message out the txd_msg_buffer is sent to the UART peripheral by a HAL command: HAL_UART_Transmit(&huart6,txd_msg_buffer, strlen((char*)txd_msg_buffer), 1000);

We can make the message be repeatedly transmitted by inserting the HAL_UART command above in the main WHILE LOOP.

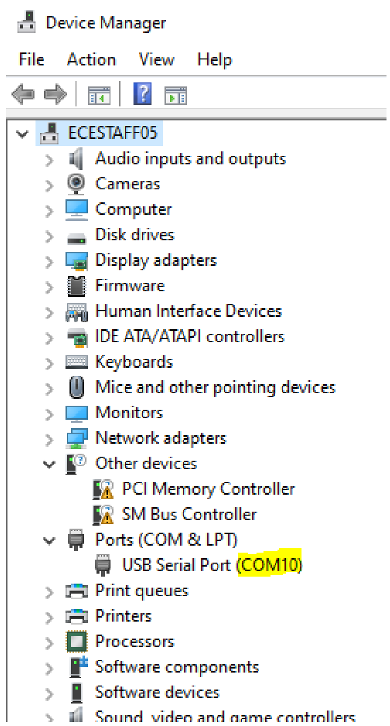
Compile and download the executable file to the Nucleo Dev board.

**Lab Computer Connection**:

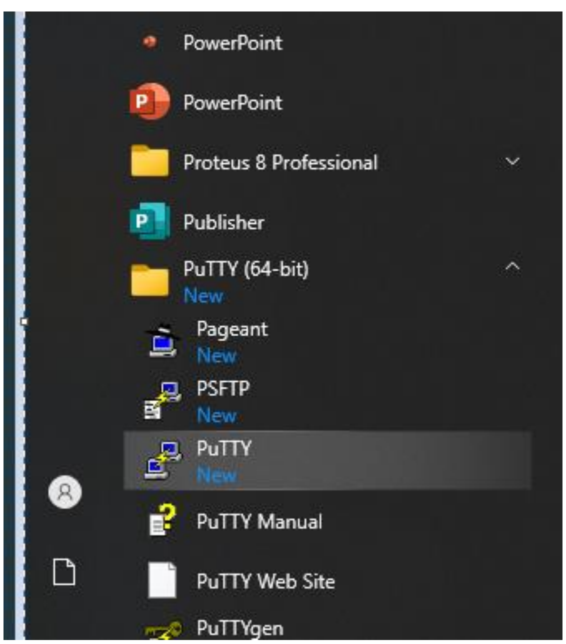
On the Lab Computer search for the "Device Manager" Control Panel utility. When the window comes up search for the Ports(COM & LPT) category.




Expand it and look for the Prolific USB-to-Serial Comm Port and NOTE THE COM number (COM10 shown here)

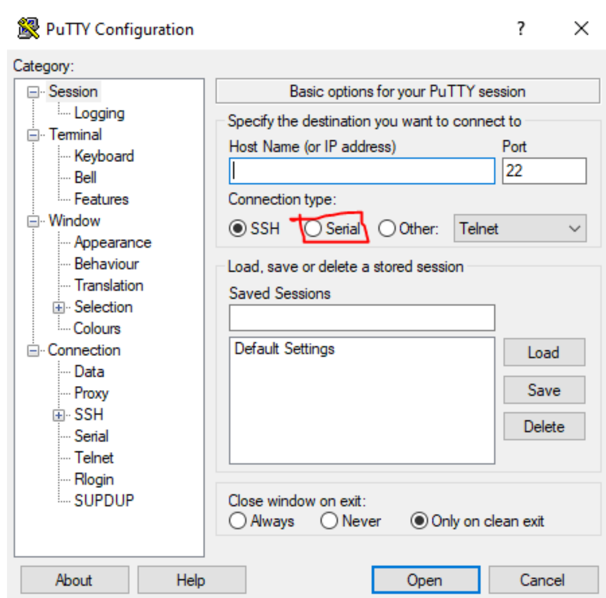



This information will now be used to set up a terminal program called PUTTY.

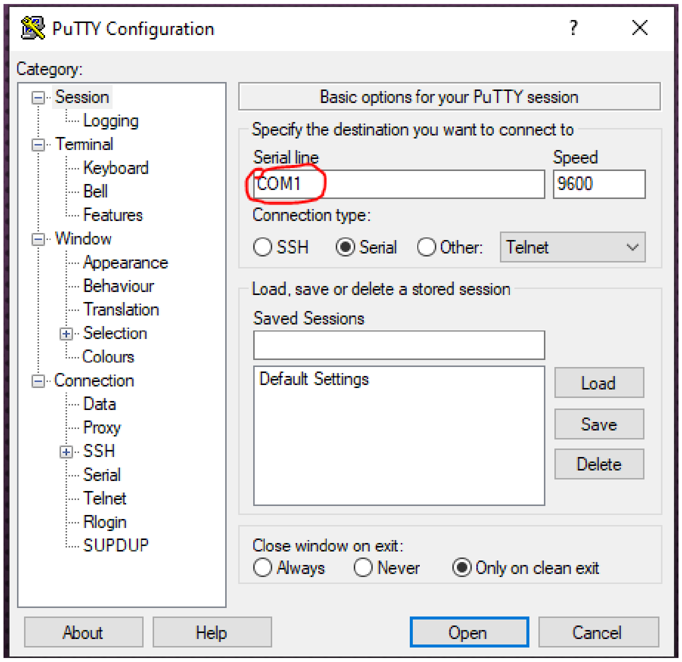



Go to the PUTTY utility on the Lab Computer START Menu:

Com

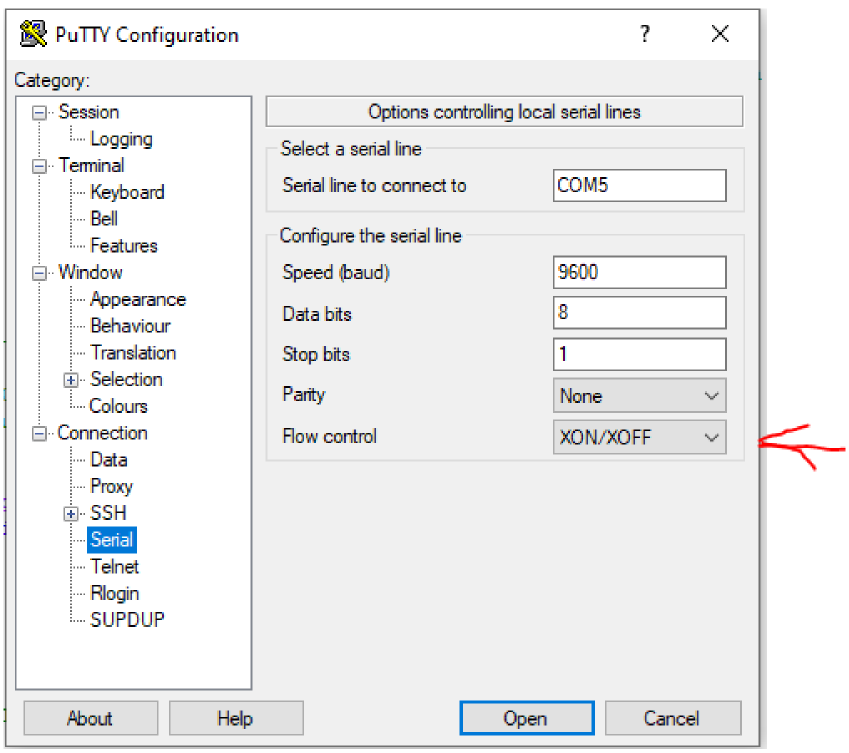



Select the PUTTY Mode for SERIAL



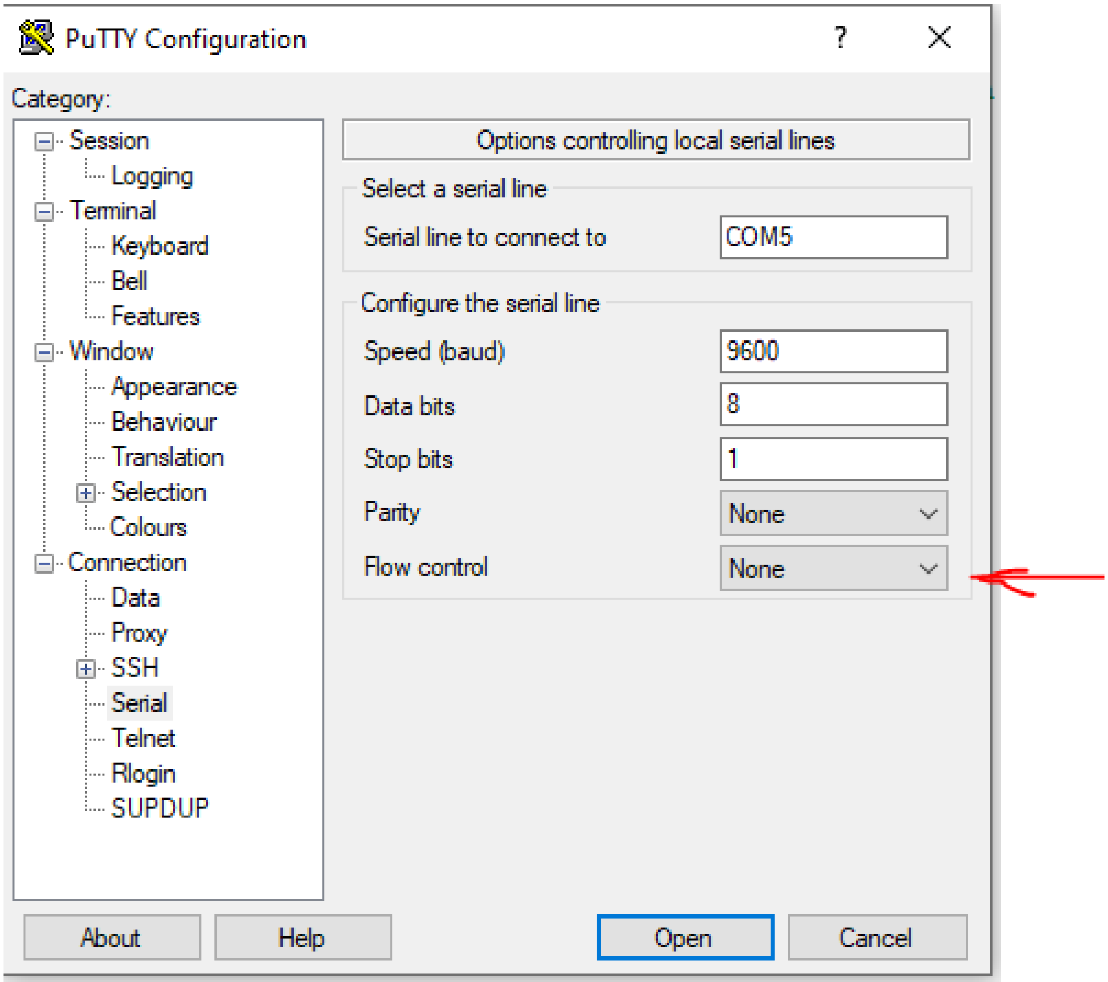

Set the Com Port to match the one found in the Device Manager Control Panel

Then go to the Connections category and select the Serial option




<u>**Change the FLOW CONTROL (default is XON/XOFF) TO NONE**</u>.

Click OPEN.




The Putty Window then opens and you can see information between the Nucleo Board and the Lab Computer.


But remember that this communications link that we have so far **is ONLY for outputting to the Lab Computer**.

## Transmit/Receive Operation with INTERRUPTS:

If we wish to also accept input from a serial link, there has to be a means for processing input characters. These input characters could arrive at any non-specified time so, an alternative capability must be used to process these unpredictable inputs. We turn to INTERRUPT PROCESSING for those kinds of events.

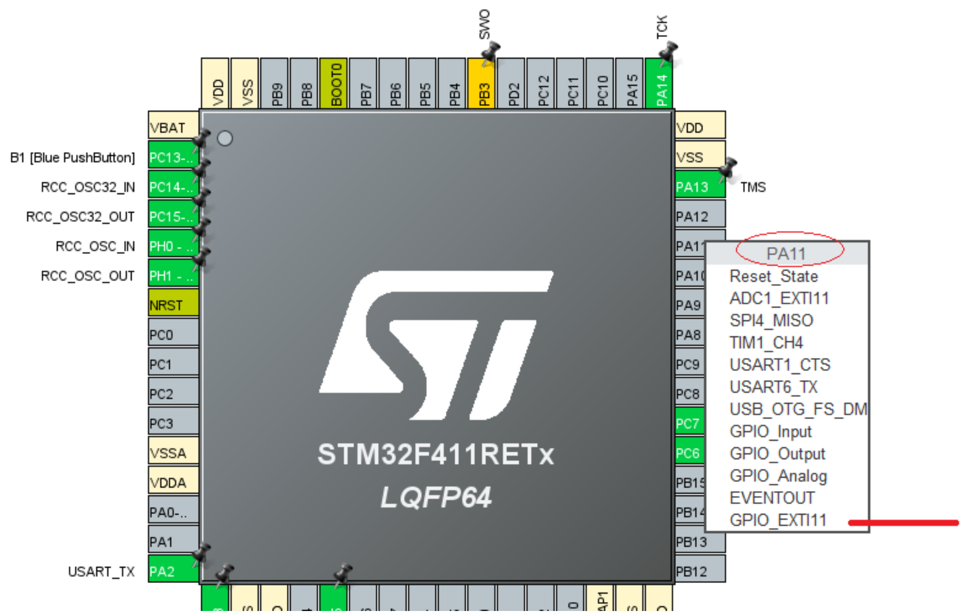
### A Brief Discussion on the Employment of Interrupts in a Nucleo Project

Interrupts are an extremely useful and efficient way to process events as they occur in a system's operation.

If an interrupt scheme is employed, then an MCU core can be relieved of the "burden" of polling for the event occurrence. Polling will always give you a FASTER <u>response</u> for processing an event than an Interrupt approach. <u>But</u> POLLING has an expensive processing cost in terms of possibly wasting MCU horsepower.

An INTERRUPT approach has its own trade-offs as well. The time required (termed as "context-switching") to store all MCU processing registers in memory so that the MCU can return to what it was doing before the interrupt event occurred can be significant. This context-switching can make the interrupt servicing take a little longer than the POLLING approach.

Interrupts can be generated directly from a GPIO Input (EXTI) in the STM32CubeIDE Configuration tool. Below the MCU Pins are shown with the EXTIxx option being a choice.




But MCU Peripherals can also be used to generate interrupts to the MCU Interrupt Controller.

For a second part of the UART testing we will be experimenting **with a BIDIRECTIONAL** serial communications function.

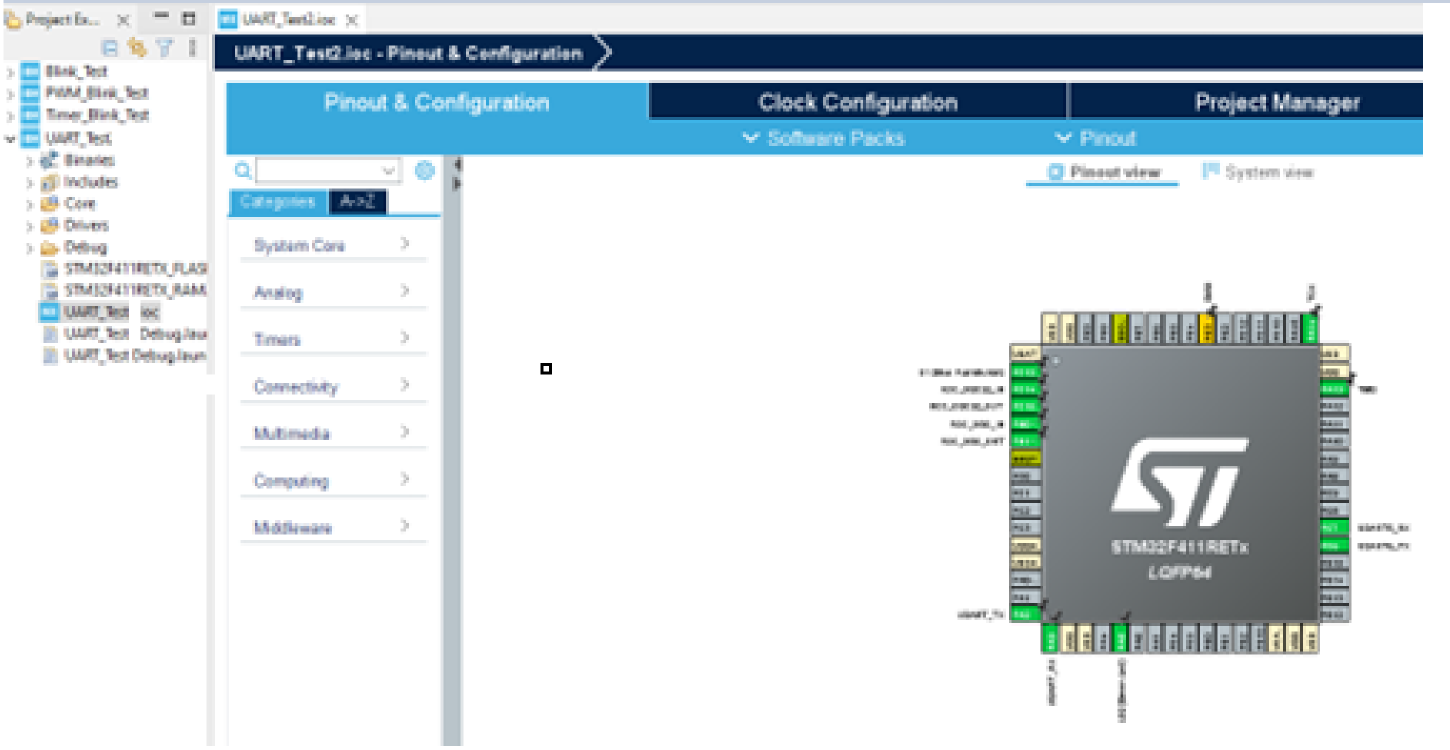
### Adding an Input Interrupt for a Serial Link:

Return to the UART_Test project.

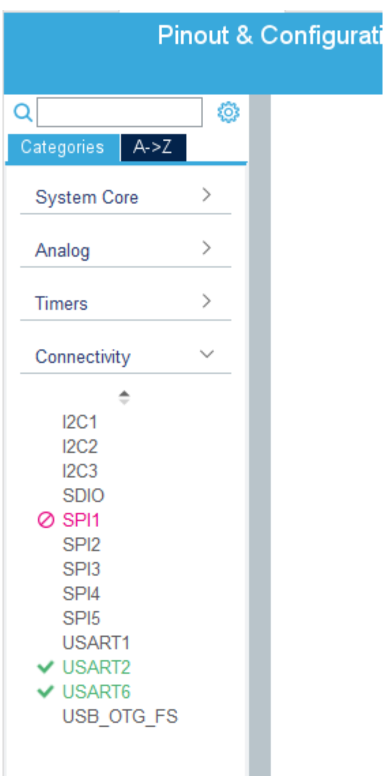
Expand the UART_Test project by clicking on the ">".

Then go to the Configurator Tool for the project by double-clicking on the UART_Test.ioc entry.

The Configurator Tool then shows how the MCU was configured last time.




In this project revision, select the Connectivity Category.




**USART2 is already in use because it is reserved for the Nucleo communications for the IDE operations.**

USART6 was used from the previous project in the lab so that will be used again.

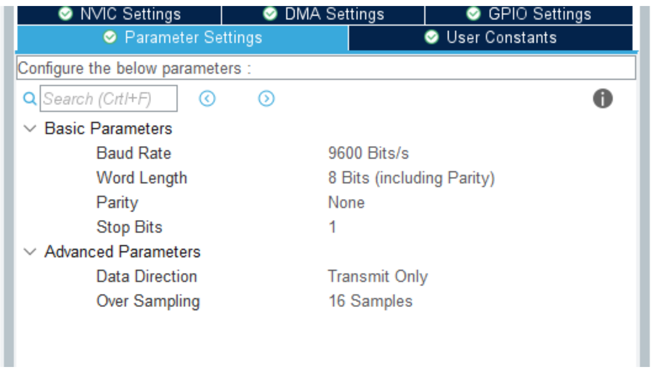
Select the USART6 peripheral and the Asynchronous Mode option will be visible as before in the USART6 Mode pane.

MANUAL FOR LAB B1 – Rev 1.4

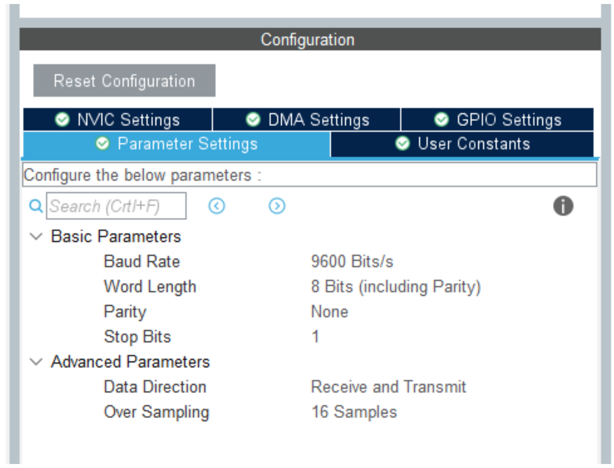
In the Configuration Pane select the Parameters Settings tab.

The Baud Rate will be 9600 as before.

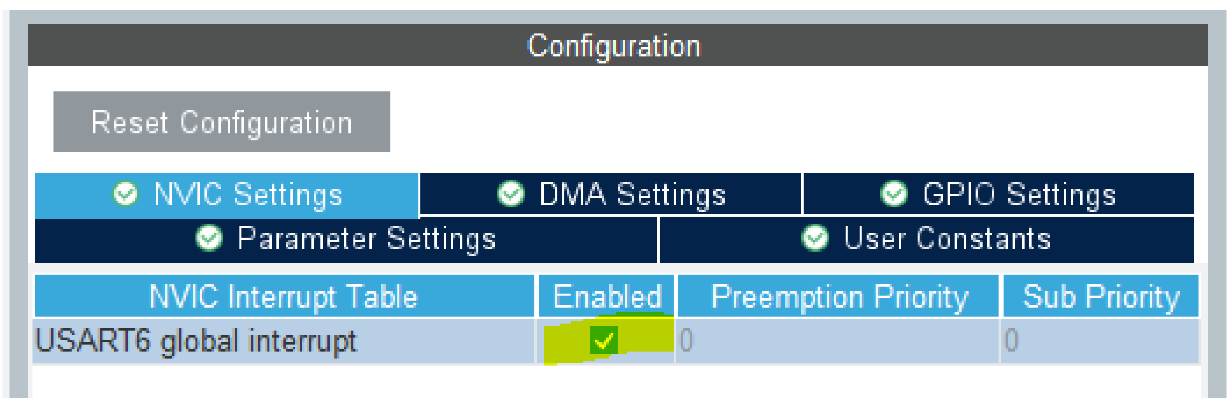
Change the Advanced Parameters DATA DIRECTION field **from TRANSMIT ONLY** (previously set for UART_Test.ioc) to the new setting **RECEIVE AND TRANSMIT**.




→




Then choose the NVIC Settings tab and click on the USART6 Global Interrupt check box (to enable an interrupt to be generated when the UART receives a character). The USART6 interrupt will be used to alert the MCU that a character has arrived at the UART Receiver port.




Run the Configuration tool (the GEAR icon in the Menu bar). An update to the main.c file is done and is opened for the project.

Take note of the following Initialization code that was just created and compare it with the generated code from the last project last time for the configured peripheral. The Mode has been changed from UART_Mode_TX to UART_Mode_TX_RX.

```
/**
  * @brief USART6 Initialization Function
  * @param None
  * @retval None
  */
static void MX_USART6_UART_Init(void)
{

  /* USER CODE BEGIN USART6_Init 0 */

  /* USER CODE END USART6_Init 0 */

  /* USER CODE BEGIN USART6_Init 1 */

  /* USER CODE END USART6_Init 1 */
  huart6.Instance = USART6;
  huart6.Init.BaudRate = 9600;
  huart6.Init.WordLength = UART_WORDLENGTH_8B;
  huart6.Init.StopBits = UART_STOPBITS_1;
  huart6.Init.Parity = UART_PARITY_NONE;
  huart6.Init.Mode = UART_MODE_TX;
  huart6.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart6.Init.OverSampling = UART_OVERSAMPLING_16;
  if (HAL_UART_Init(&huart6) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART6_Init 2 */

  /* USER CODE END USART6_Init 2 */

}
```

```
static void MX_USART6_UART_Init(void)
{

  /* USER CODE BEGIN USART6_Init 0 */

  /* USER CODE END USART6_Init 0 */

  /* USER CODE BEGIN USART6_Init 1 */

  /* USER CODE END USART6_Init 1 */
  huart6.Instance = USART6;
  huart6.Init.BaudRate = 9600;
  huart6.Init.WordLength = UART_WORDLENGTH_8B;
  huart6.Init.StopBits = UART_STOPBITS_1;
  huart6.Init.Parity = UART_PARITY_NONE;
  huart6.Init.Mode = UART_MODE_TX_RX;
  huart6.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart6.Init.OverSampling = UART_OVERSAMPLING_16;
  if (HAL_UART_Init(&huart6) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART6_Init 2 */

  /* USER CODE END USART6_Init 2 */
```

Recall from the last lab, that to use string functions and STDIO functions we need to have two INCLUDE statements in the USER CODE BEGIN Include section.

```
#include <string.h>
```

```
#include <stdio.h>
```

For the UART Receiver, some additional memory will be required to store received data. Because this variable is to be accessed by both the "main" routine and by an interrupt routine, the variable declaration must be declared outside of the "main" section. The statement uint8_t byte is to be placed in the Private User Code area.

```
/* USER CODE BEGIN PD */

/* Single byte to store UART input */
uint8_t byte;

/* USER CODE END PD */
```

Also a variable for indicating if an interrupt process is completed, is required (rcv_intpt_flag) in the USER CODE 0 area. This flag will be RESET in the MAIN routine and will be SET at the end of each Interrupt routine.

```
/* USER CODE BEGIN 0 */
uint8_t rcv_intpt_flag = 0;

/* USER CODE END 0 */
```

In the previous lab In the main.c file in UART_Test, a memory buffer was created that can be used tol hold a message for transmission. For this, a variable array of 8-bit integers into the User Code Begin 1 section:

Uint8_t txd_msg_buffer[64] = {0};

Recall, that to load the txd_msg_buffer with a message, the sprintf command is used. This time the message will be:

Sprintf((char*)txd_msg_buffer, “INPUT A CHRACTER:”);

Then to transmit the message, the txd_msg_buffer is sent to the UART peripheral by a HAL command: HAL_UART_Transmit(&huart6,txd_msg_buffer, strlen((char*)txd_msg_buffer), 1000); sent repeatedly. See below.

```
int main(void)
{
  /* USER CODE BEGIN 1 */
    uint8_t txd_msg_buffer[64] = {0}; //buffer space for output messages on UART
  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART2_UART_Init();
  MX_USART6_UART_Init();
  /* USER CODE BEGIN 2 */
  sprintf((char*)txd_msg_buffer, "\r\n INPUT A CHARACTER:");

  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
      HAL_UART_Transmit(&huart6, txd_msg_buffer, strlen((char*)txd_msg_buffer),1000);

    /* USER CODE END WHILE */

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

But for the revised UART_Test project, some new commands will be inserted to handle the reception of Terminal characters using an interrupt process.

```
/* USER CODE BEGIN 2 */
sprintf((char*)txd_msg_buffer, "\r\n INPUT A CHARACTER:");

//  HAL_UART_Transmit(&huart6, txd_msg_buffer, strlen((char*)txd_msg_buffer),1000);
//

/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
   rcv_intpt_flag = 00; // this flag is used to see if an receiver interrupt has occurred
   HAL_UART_Receive_IT(&huart6, &byte, 1); /* this enables the Uart Receiver to create an interrupt when
                                               a character arrives, it will be placed in "byte". */
   HAL_UART_Transmit(&huart6, txd_msg_buffer, strlen((char*)txd_msg_buffer),1000);
   while(rcv_intpt_flag == (00)) {}
  /* USER CODE END WHILE */

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

In the above screenshot, firstly an Interrupt-related flag (rcv_intpt_flag) is RESET to 00. Then, the UART6 Receiver is enabled to generate an Interrupt if a character arrives (received). The command for that is: HAL_UART_Receive_IT(&huart6,&byte,1);

Then the message, "\r\n INPUT A CHARACTER:" is sent out to the terminal.

THEN, the MCU is made to wait until the rcv_intpt_flag rcv_intpt_flag gets SET (i.e.: NOT EQUAL to 00) by an interrupt service routine ("Callback call").

When a character arrives from the Terminal, an interrupt occurs. The character is by the UART receiver and has been placed in memory at the location of "byte". Then, the MCU makes a "Callback" call that is associated with that interrupt.

NOTE: The MCU has a few, fixed reference names for Callback routines associated with the UART Receiver Interrupt function call from the Interrupt Controller. The Function Call we will use is **HAL_UART_RxCpltCallback.**

Let's move to the code where the specific Interrupt Callback processing is located. When the UART receives an input character, the MCU places the input character into memory at "byte" as mentioned above. Then it calls the routine (to be entered by you) shown below. Note that it is placed in USER CODE SECTION 4.

```
/* USER CODE BEGIN 4 */
/* This callback is called by the HAL_UART_IRQHandler when the given number of bytes are received */
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
  if (huart->Instance == USART6)
  {
    /* Transmit one byte with 100 ms timeout. Transmit the character that was received into the variable called "byte
    HAL_UART_Transmit(&huart6, &byte, 1, 100);
    rcv_intpt_flag = 1;
  }
}

/* USER CODE END 4 */
```

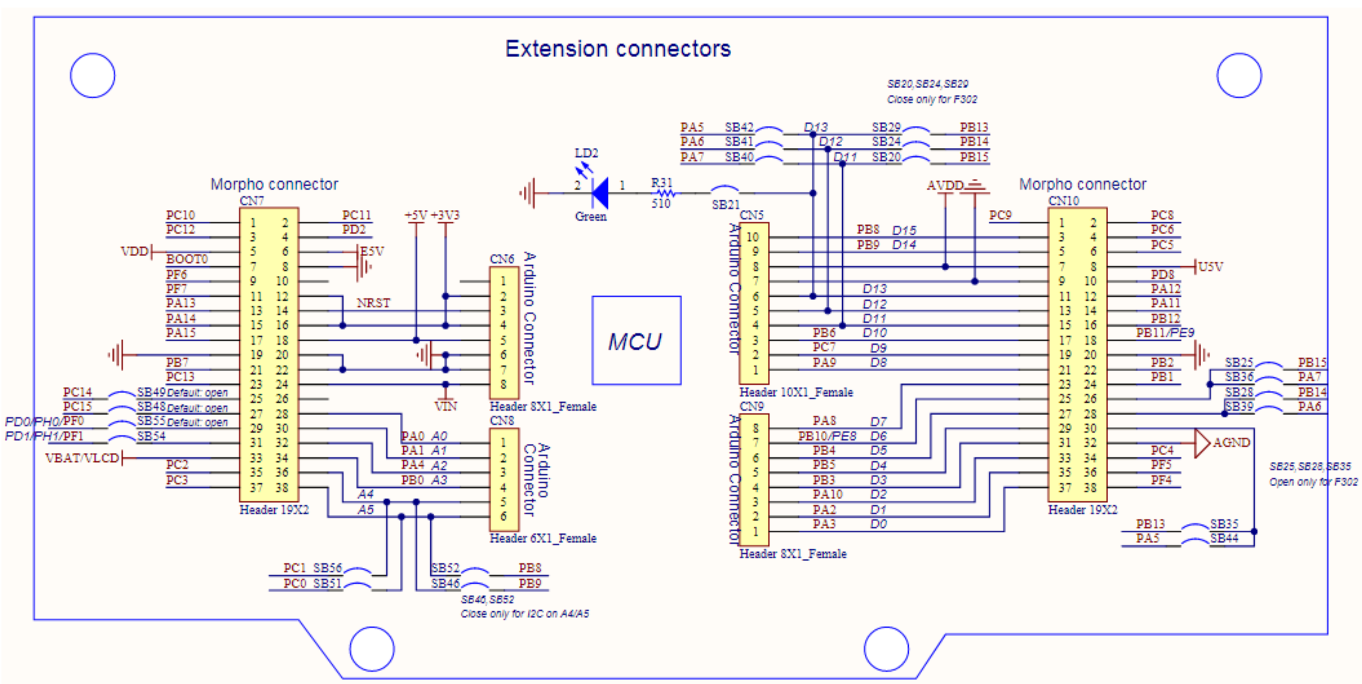
Here, the Callback code "echoes" the character back to the UART Transmitter (and the Terminal) and then it SETS the rcv_intpt_flag before a return to the mainline.

Then, in the mainline, the MCU is allowed to exit the “while (rcv_intpt_flag ==00) {} waiting loop, and the output message to the Terminal is sent again and the loop processing repeats.

Compile the main.c file and if no errors are present, then download it to the Nucleo board.

After downloading to the Nucleo, run Putty again with the new capability. A screenshot is shown below.


In the STM32CubeIDE, look for the GPIO pins used for the UART1 TxD and Rxd Pins. Then locate those signal net names on the Nucleo Schematic (see below).



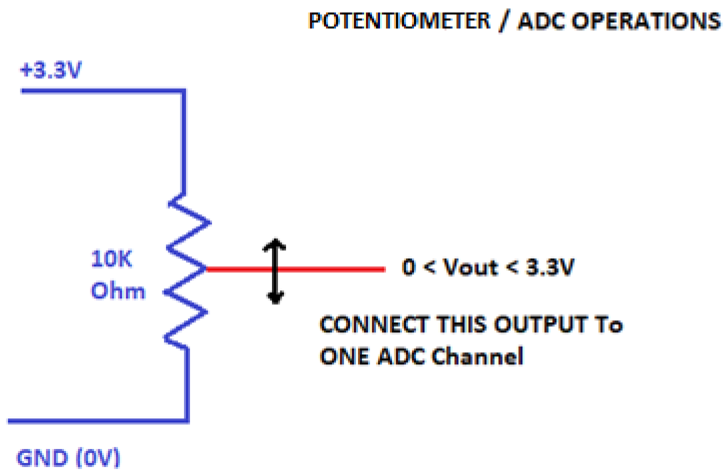
## SIXTH EXERCISE: Using an ADC Peripheral to Convert an Analog Input:

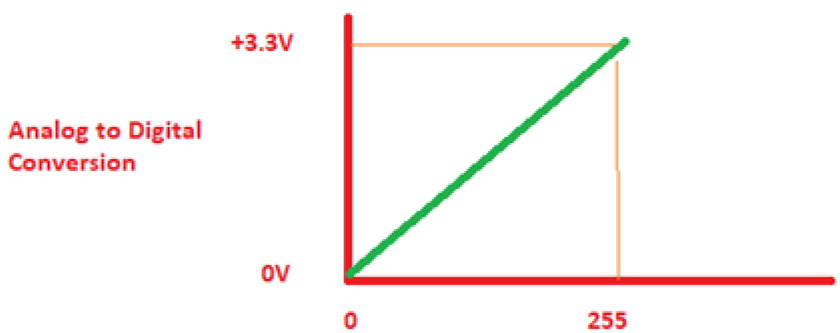
Next, we will move on to employing the single internal ADC peripheral. There are up to 16 input channels to which this peripheral can connect**, but only ONE AT A TIME**. For each of these input Analog channels, ANALOG signal levels must be limited to values between 0V and 3.3V. The analog voltage levels will be converted to 8-bit digital values over time.

**ON YOUR GREADBOARD**:

You will need the 10K Ohm Potentiometer from your kit, on a breadboard with connections to the Nucleo board for this next step.

**POTENTIOMETER**: This is a **USER INPUT device, and it** is to be placed on the Breadboard. The Potentiometer must be connected between +3.3V and GND on the outer two pins. The middle pin is the USER Input that won't need any voltage translation between it and the MCU ADC Input pins. The middle pin will have an analog voltage between +3.3V and 0V.




The MCU program for this device test must use the ADC with the following parameters. Middle pin analog values must be determined by the ADC Peripheral. Polling may be used to determine when each ADC conversion is completed. The X and Y values should be between 0 and 255 for each axis. These values must be displayed on another UART peripheral connection to the Lab Computer running a PUTTY session.

Begin an new STM32 project like in the previous fashion

MANUAL FOR LAB B1 – Rev 1.4

Board Selector: Nucleo-64, MCU Series: STM32F4,

Board:  Nucleo F401RE,

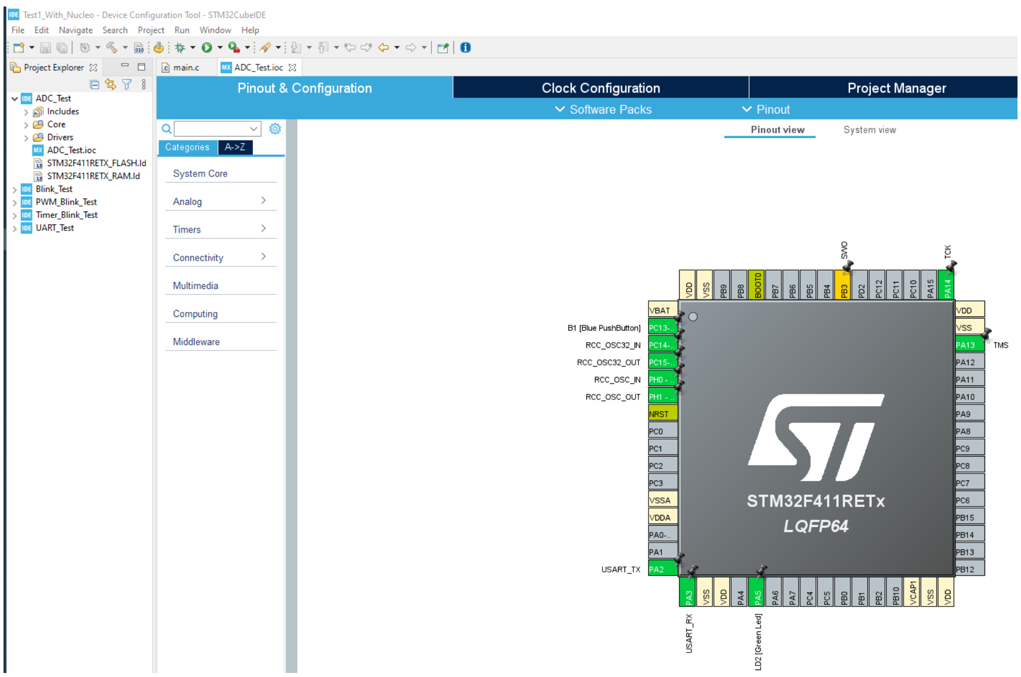
Project Name: ADC_Test and **remember to  Initialize all Peripherals with default Mode**.

For full disclosure, I will be starting with initial developments for using the ADC peripheral from a site on the web (Controllerstech.com) on how to develop the ADC for use with any of 16 input channels.

https://controllerstech.com/stm32-adc-multi-channel-without-dma/

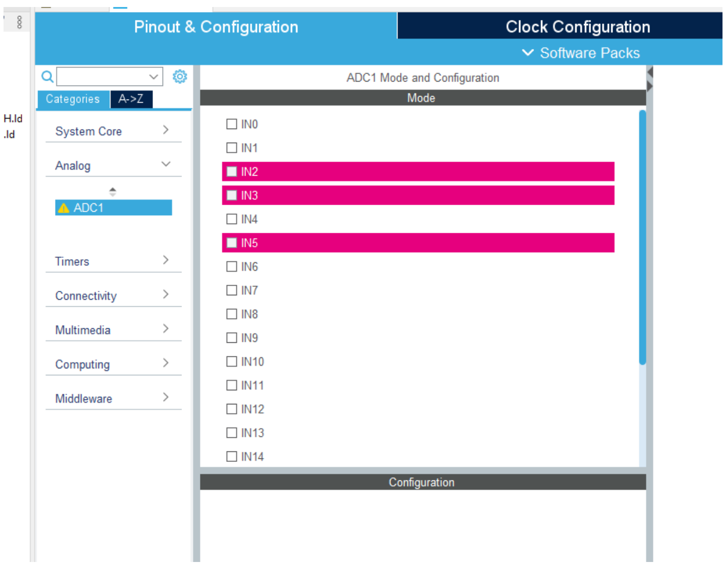
The code will be further simplified to include a function call that can be used to select any of those 16 ADC input channels. You may reduce the range of input selections if you wish, but just be aware of the provided syntax etc.

Going to the configuration Tool with the STM32CubeIIDE, there is the following startup screen after all of the reserved Nucleo peripherals are set to their default modes. Like was done before, set the Clock configuration for all clocks to 16 MHz.




The ADC peripheral is in the Analog Category. Click on the Analog > ADC 1 peripheral.

The ADC1 pane comes up and right away you will notice, because of the reserved pins for interfacing to the Nucleo board, that certain ADC input channels are NOT available (highlighted in magenta).




The GPIO pins that might be used by the ADC inputs for IN2, IN3 and IN5 are being used by other peripherals. This situation reveals that pin planning for using peripherals is required.

But for the the ADC_Test that is being developed here, another channel wil be used.

Choose **one input channels** in this project. For example, e.g.: choose IN9. This channel will be used under software control in the main.c file later.

Choosing a channel then opens up another configuration pane.

There is a limit for the ADC clocking operation which is a maximum of 14 MHz. But all of the projects that are being developed for the MCU in this course use a common 16MHz. So, the Clock Prescaler field must be set to a value of PCLK2 divided by 2. This means that the clock for the ADC will be 8 MHz.

Other Parameters:

ADC resolution:  8 bits.

Data Alignment: Right Alignhment

Scan Conversion Mode: Enabled

Continuous Conversion Mode: Disabled

Discontinuous Conversion Mode: Disabled

DMA Continuous Mode: Disabled

End of Conversion Selection:  EOC at the end of a Single Conversion

ADC_Regular ConversionMode:

Number of Conversions: 1

External Trigger Conversion Source: Regular Conversion Launched by software

External Trigger Conversion Edge: NONE

Rank:    1

Channel:  Channel 9 (let this be the default channel for now)

Sampling Time: 15 cycles

The ADC will be used with the above setup to run ONE ADC conversion. It will use 15 (8-MHz) clock cycles to sample the analog signal input. After it is completed, it will set an internal flag (EOC) to indicate that the conversion is done.

In the software in the main.c file the ADC will be polled to see when the EOC flag is asserted.

Run the Configuration utility (Gear looking ICON) to generate the auto-code for the ADC parameters.

After the Configuration tool run is completed, the following auto-code is placed in the main.c file for the ADC_Test project.

```
static void MX_ADC1_Init(void)
{

  /* USER CODE BEGIN ADC1_Init 0 */

  /* USER CODE END ADC1_Init 0 */

  ADC_ChannelConfTypeDef sConfig = {0};

  /* USER CODE BEGIN ADC1_Init 1 */

  /* USER CODE END ADC1_Init 1 */

  /** Configure the global features of the ADC (Clock, Resolution, Data Alignment and number of conversion)
  */
  hadc1.Instance = ADC1;
  hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV2;
  hadc1.Init.Resolution = ADC_RESOLUTION_8B;
  hadc1.Init.ScanConvMode = ENABLE;
  hadc1.Init.ContinuousConvMode = DISABLE;
  hadc1.Init.DiscontinuousConvMode = DISABLE;
  hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
  hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
  hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc1.Init.NbrOfConversion = 1;
  hadc1.Init.DMAContinuousRequests = DISABLE;
  hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
  if (HAL_ADC_Init(&hadc1) != HAL_OK)
  {
    Error_Handler();
  }

  /** Configure for the selected ADC regular channel its corresponding rank in the sequencer and its sample time.
  */
  sConfig.Channel = ADC_CHANNEL_9;
  sConfig.Rank = 1;
  sConfig.SamplingTime = ADC_SAMPLETIME_3CYCLES;
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN ADC1_Init 2 */
```

Notice in the highlighted portion at the bottom of the ADC auto-code, is a section that can be changed on a per-channel basis. The channel just shows an entry for using the channel 9 input.

If the ADC is setup to do the SAME kind of conversion (8-bit resolution, 15-cycle sampling etc. for a number of ADC channel inputs then all that is required to make the ADC use a different channel is the "sConfig.Channel = ADC_CHANNEL_x" instruction.

A function call routine can be created to change the value for this field depending on the ADC channel that you want to use. PLEASE BEAR IN MIND THAT IF AN ADC CHANNEL IS NOT AVAILABLE (Because another peripheral is using that pin) THEN AN EXIT TO AN ERROR HANDLER WILL BE INVOKED.

The Functional Call written below is "generic". It does not concern itself with which channel is available, so the user must be aware of which channel number being passed to the function call is valid.

The function call is ADC_Select_CH(int CH) (highlighted below). It consist of a C "switch" statement and it generates the text for the sConfig.Channel field for the ADC based on a Channel integer being sent to it.

The function call must be DECLARED in the USER CODE PFP as shown below:

```c
  */
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */

/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
ADC_HandleTypeDef hadc1;

UART_HandleTypeDef huart2;

/* USER CODE BEGIN PV */

/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);
static void MX_ADC1_Init(void);
/* USER CODE BEGIN PFP */
static void ADC_Select_CH(int CH);
/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
/* The following code snippet is adapted from the examples at controllerstech.com
 PLEASE BE SURE THAT YOU DON'T SELECT AN INACCESSABLE CHANNEL DUE TO A GPIO PIN BEING
 USED FOR A PURPOSE OTHER THAN FOR AN ADC CHANNEL */
void ADC_Select_CH(int CH)
{
	ADC_ChannelConfTypeDef sConfig = {0};

	switch(CH)
```

(continued below)

The function call itself must be placed in the **USER Code 0 section**. You may copy and paste the following:

```
void ADC_Select_CH(int CH)
{
	ADC_ChannelConfTypeDef sConfig = {0};

	switch(CH)
	{
	case 0:
	sConfig.Channel = ADC_CHANNEL_0;
	sConfig.Rank = 1;
	if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
	 {
	   Error_Handler();
	 }
	break;
	case 1:
	sConfig.Channel = ADC_CHANNEL_1;
	sConfig.Rank = 1;
	if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
	 {
	   Error_Handler();
	 }
	break;
	case 2:
	sConfig.Channel = ADC_CHANNEL_2;
	sConfig.Rank = 1;
	if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
	 {
	   Error_Handler();
	 }
	break;
	case 3:
	sConfig.Channel = ADC_CHANNEL_3;
	sConfig.Rank = 1;
	if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
	 {
	   Error_Handler();
	 }
	break;
	case 4:
	sConfig.Channel = ADC_CHANNEL_4;
	sConfig.Rank = 1;
	if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
	 {
	   Error_Handler();
	 }
	break;
	case 5:
	sConfig.Channel = ADC_CHANNEL_5;
	sConfig.Rank = 1;
	if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
	 {
	   Error_Handler();
	 }
	break;
	case 6:
	sConfig.Channel = ADC_CHANNEL_6;
```

```
        sConfig.Rank = 1;
        if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
         {
           Error_Handler();
         }
        break;
        case 7:
        sConfig.Channel = ADC_CHANNEL_7;
        sConfig.Rank = 1;
        if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
         {
           Error_Handler();
         }
        break;
        case 8:
        sConfig.Channel = ADC_CHANNEL_8;
        sConfig.Rank = 1;
              if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
         {
           Error_Handler();
         }
        break;
        case 9:
        sConfig.Channel = ADC_CHANNEL_9;
        sConfig.Rank = 1;
              if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
         {
           Error_Handler();
         }
        break;
        case 10:
        sConfig.Channel = ADC_CHANNEL_10;
        sConfig.Rank = 1;
        if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
         {
           Error_Handler();
         }
        break;
        case 11:
        sConfig.Channel = ADC_CHANNEL_11;
        sConfig.Rank = 1;
        if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
         {
           Error_Handler();
         }
        break;
        case 12:
        sConfig.Channel = ADC_CHANNEL_12;
        sConfig.Rank = 1;
        if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
         {
           Error_Handler();
         }
        break;
        case 13:
        sConfig.Channel = ADC_CHANNEL_13;
        sConfig.Rank = 1;
        if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
```

```
         {
           Error_Handler();
         }
        break;
        case 14:
        sConfig.Channel = ADC_CHANNEL_14;
        sConfig.Rank = 1;
        if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
         {
           Error_Handler();
         }
        break;
        case 15:
        sConfig.Channel = ADC_CHANNEL_15;
        sConfig.Rank = 1;
        if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
         {
           Error_Handler();
         }
        break;

        }

}
```

The above function call is just used to select any of 16 ADC input channels. **YOU WILL ONLY NEED ONE for your Embedded Project.** The RANK field is required for each sConfig change.

After function call is made, the ADC Conversion must be started. Then the ADC is polled for ADC completion. Finally, when the ADC process is finished, the value from the ADC is transferred to a variable.

So, with the above code installed in your main.c file, you can add some further HAL commands to access the ADC registers. The following code is just a generic example.

```
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {

      ADC_Select_CH(9);
      HAL_ADC_Start(&hadc1);
      HAL_ADC_PollForConversion(&hadc1, 1000);
      uint8_t ADC_CH9 = HAL_ADC_GetValue(&hadc1);
      HAL_ADC_Stop(&hadc1);

    /* USER CODE END WHILE */
```

Set up a UART for connection (e.g. UART6) to a Lab Computer with a Putty app session running on it like was done earlier. You can report the digitized values of the analog input levels. An example (using an arbitrary ADC channel), of how to build a message with a variable value included in it, is shown below:

```c
while (1)
{
    ADC_Select_CH(14);
    HAL_ADC_Start(&hadc1);
    HAL_ADC_PollForConversion(&hadc1, 1000);
    uint8_t ADC_CH14 = HAL_ADC_GetValue(&hadc1);
    HAL_ADC_Stop(&hadc1);
    /* PRINT OUT THE 8 bit digitized value of the external analog voltage
    sprintf((char*)txd_msg_buffer, "DIGITIZED ANALOG VALUE FOR CH14:  %d \r \n", ADC_CH14);
    HAL_UART_Transmit(&huart6, txd_msg_buffer, strlen((char*)txd_msg_buffer),1000);
    HAL_Delay(1000);
```

Below Is a PUTTY session running in a continuous software loop with the potentiometer and the Nucleo board.

Remember that USART Channel 6 connects to the workstation USB port.

Within the Nucleo MCU, each ADC measurement is set for an 8-bit resolution (256 levels). When the potentiometer is at its center position the Digitized value should be close to half of the 256-level scale for the channel.

For the channel the256 digital value represents 3.3V, the mid-range value of 127 is about 1.65V, and 0 is 0V etc.

MID-RANGE VALUE:

```
DIGITIZED ANALOG VALUE FOR CH9:  127
DIGITIZED ANALOG VALUE FOR CH9:  127
DIGITIZED ANALOG VALUE FOR CH9:  127
DIGITIZED ANALOG VALUE FOR CH9:  127
DIGITIZED ANALOG VALUE FOR CH9:  127
```

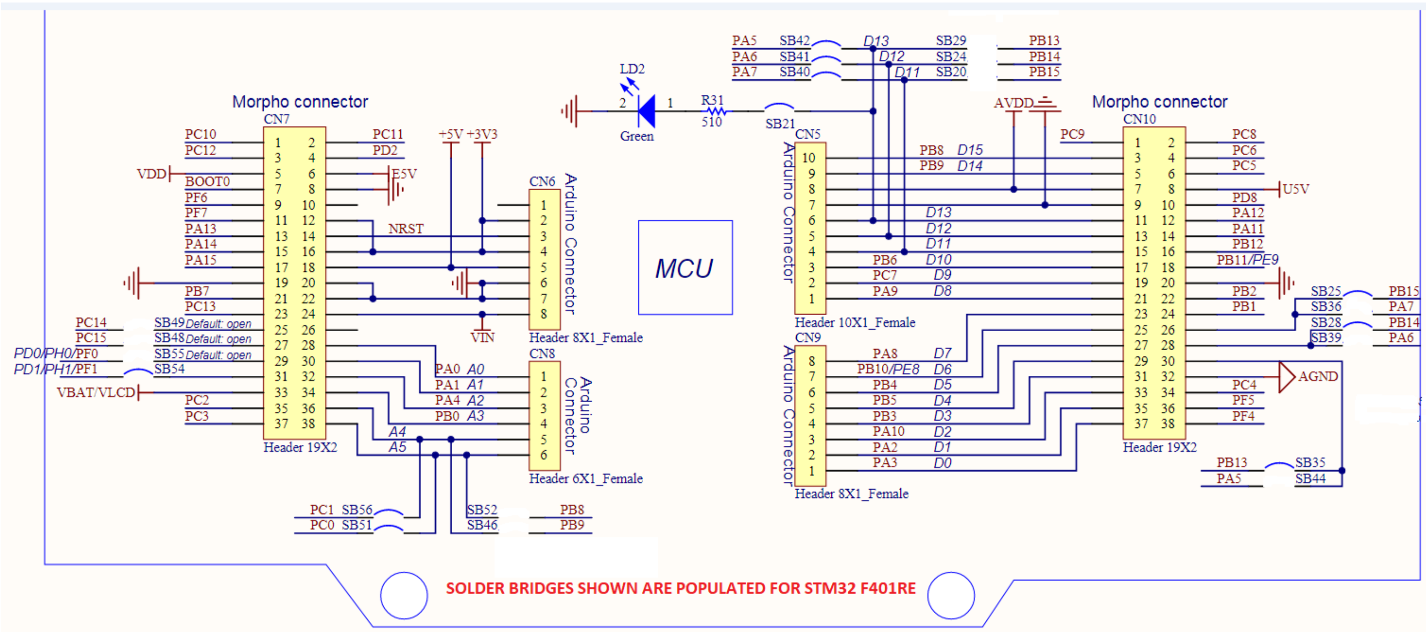
MINIMUM VALUE:

```
DIGITIZED ANALOG VALUE FOR CH9: 0
DIGITIZED ANALOG VALUE FOR CH9: 0
DIGITIZED ANALOG VALUE FOR CH9: 0
```

MAXIMUM VALUE:

```
DIGITIZED ANALOG VALUE FOR CH9:  255
DIGITIZED ANALOG VALUE FOR CH9:  255
DIGITIZED ANALOG VALUE FOR CH9:  255
DIGITIZED ANALOG VALUE FOR CH9:  255
```

Validate your Potentiometer operation control by observing the channel values with the above steps for the potentiometer positions etc.


Morpho connector
CN7
Header 19X2
Morpho connector
CN10
Header 19X2
CN6
Arduino Connector
Header 8X1_Female
CN8
Arduino Connector
Header 6X1_Female
CN5
Arduino Connector
Header 10X1_Female
CN9
Arduino Connector
Header 8X1_Female
MCU
LD2
Green
R31
510
SB21
NRST
VIN
AVDD
U5V
E5V
AGND
VDD
BOOT0
VBAT/VLCD
SOLDER BRIDGES SHOWN ARE POPULATED FOR STM32 F401RE