
# Exercise Manual
# For

# SC2104/CE3002
# Sensors, Interfacing and Digital Control

## Practical Exercise #1:
## Familiarization with
## STM32CubeIDE
## and
## STM32F4 Board

**Venue: SCSE Hardware Lab**

**COMPUTER ENGINEERING COURSE**

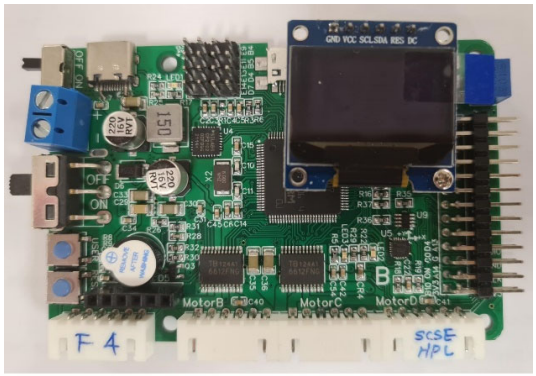
**SCHOOL OF COMPUTER SCIENCE AND ENGINEERING**
**NANYANG TECHNOLOGICAL UNIVERSITY**

## Learning Objectives

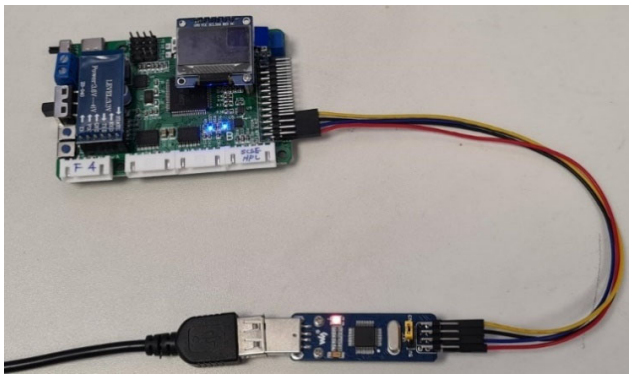
These exercises are to introduce students to the STM32F4 board and its software development platform STM32CubeIDE. The student will learn how to use the development platform to configure the STM32F4 board, and how to code application programs to access the hardware features provided on the board.

## Equipment and accessories required

i) One desktop computer installed with STM32CubeIDE and PuTTY.
ii) One STM32F4 board
iii) One ST-LINK SWD board
iv) One HC-SR04 Ultrasonic Sensor



STM32F4 board



ST-Link for downloading and debugging code

---

## Introduction

You will go through a sequence of tasks designed to let you be familiarized with using the STM32CubeIDE platform to develop programs for the STM32F4 processor based systems. The applications that you develop in this practical exercise #1 will be used in subsequent lab exercises, while the procedures learnt in here will not be reiterate in the subsequent lab exercises (we will assume that you are already familiar with it after this lab).

## 1. STM32CubeIDE and STM32F4 Board

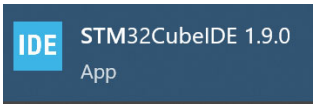
STM32CubeIDE is a C (and C++) development platform that supports GUI based configuration wizards to automatic generate C initialization for configured peripherals, code compilation and debugging for STM32 based microcontrollers and microprocessors.

STM32 CubeIDE

This IDE is based on the Eclipse/CDT framework and GCC toolchain for code development, and the computer in the lab should already be installed with the latest version of the STMCubeIDE.

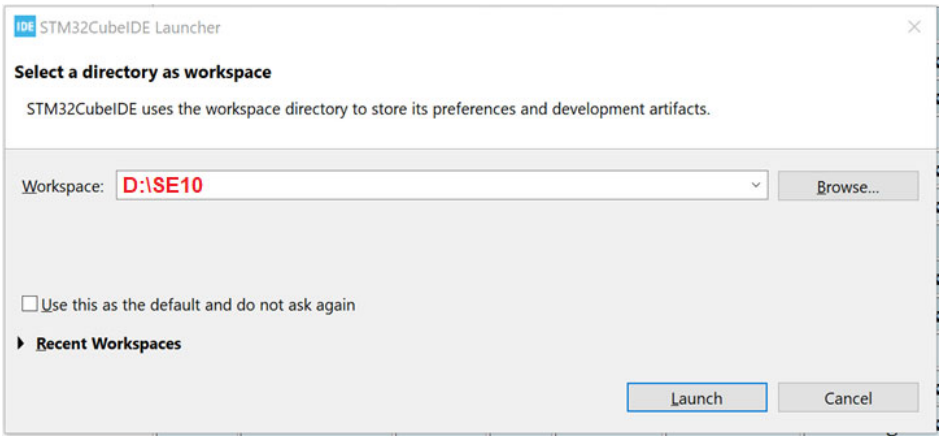
## 1.1. Setting up STM32CubeIDE

Launch the STM32CubeIDE on the computer



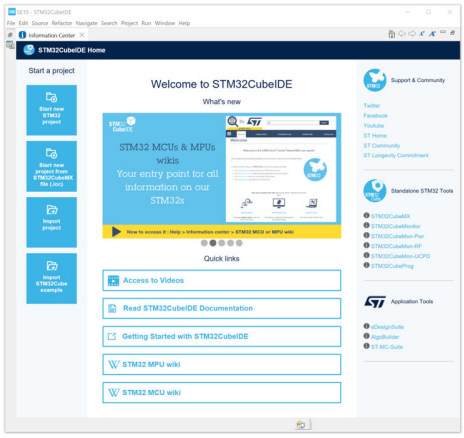

- You will be asked to create a sub-directory for the Workspace to be used.

  When doing the SC2104 lab exercises in SCSE labs using the lab's computer, you should choose Drive D for the Workspace* and create a subdirectory (based on your lab group number, e.g., D:\SE10).

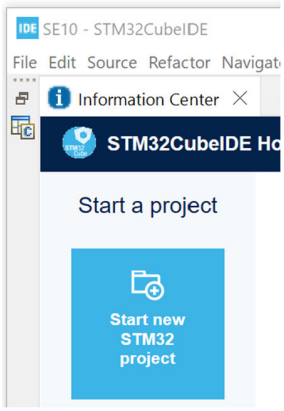



  * Note that as the lab computers bootup through Virtualization, all data you store on Drive D will be deleted after you logout of the computer. As such, you should bring a USB thumbdrive to save a copy of your code at the end of the lab session which you can use for subsequent lab exercises. Alternatively, you can choose your USB thumbdrive to be the Workspace (instead of using Drive D), and all your code will then be saved to it automatically.

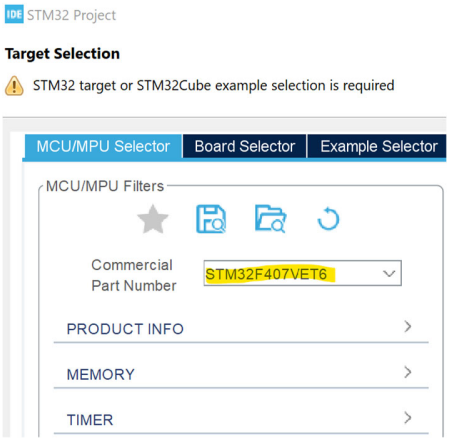
- You will see the IDE startup interface as follows



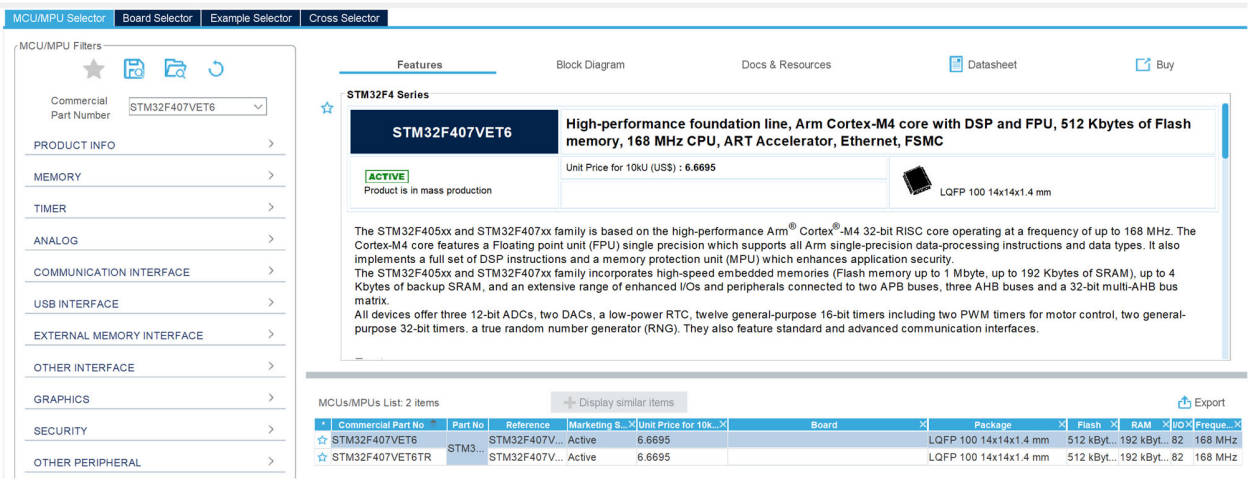
- Select "Start new STM32 project" on the left panel.



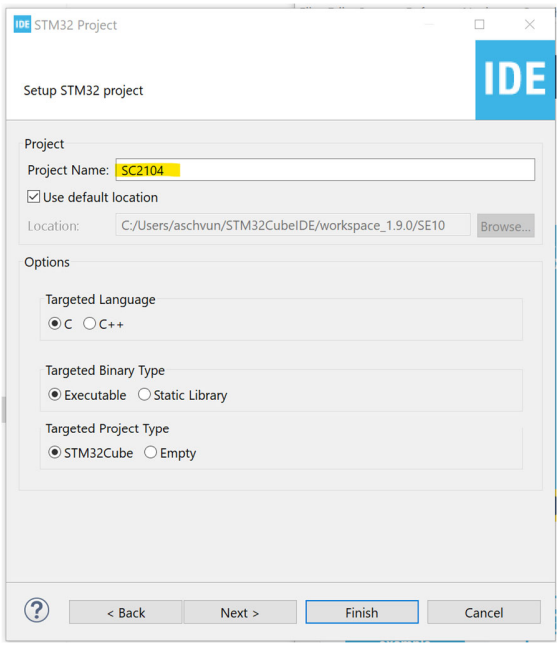

- Key in the microcontroller's Part Number used in the STM32F4 board: STM32F407VET6




- The IDE will list the corresponding part available in its library, which you can then select.

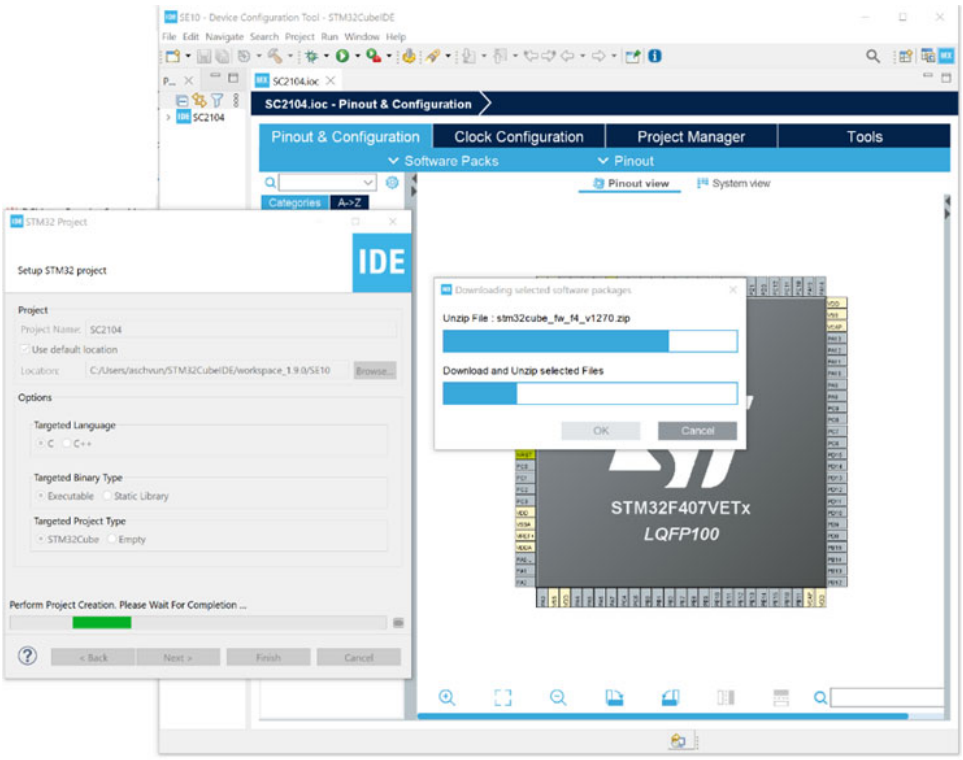


- Click “Next” and enter the Project Name (e.g., ‘SC2104’, or better - ‘SC2104-Lab1’).



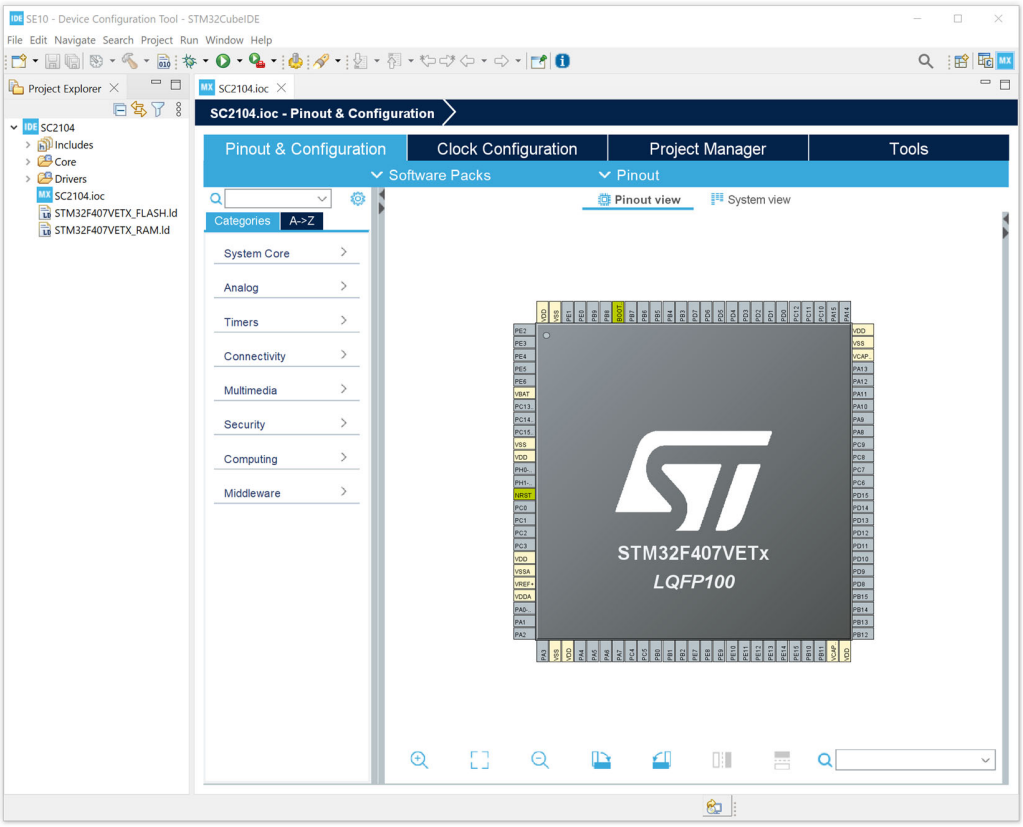

- Click “Finish”.

The IDE will then do the necessary setup and download the appropriate files related to the selected microcontroller (which may take a while for a new project).

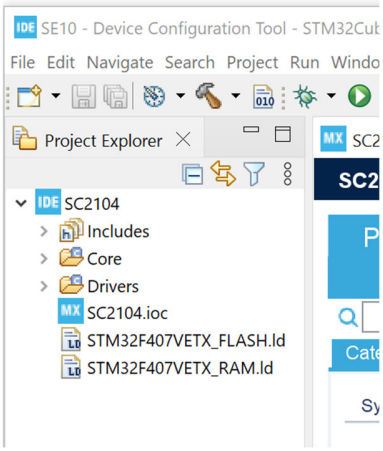


## 1.2. Microcontroller Configuration

Once the setup of the microcontroller by the IDE is completed, you will be presented with the screen showing a GUI of the microcontroller selected.

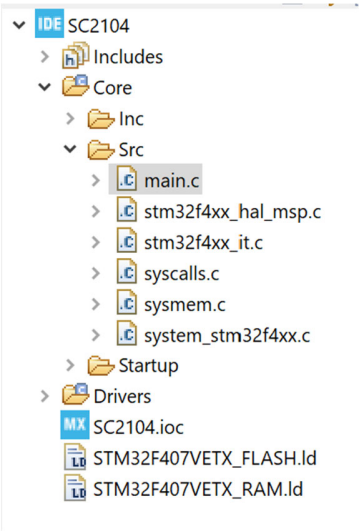



You will also see some files on its left panel, which have been setup by the IDE corresponds to the microcontroller selected.



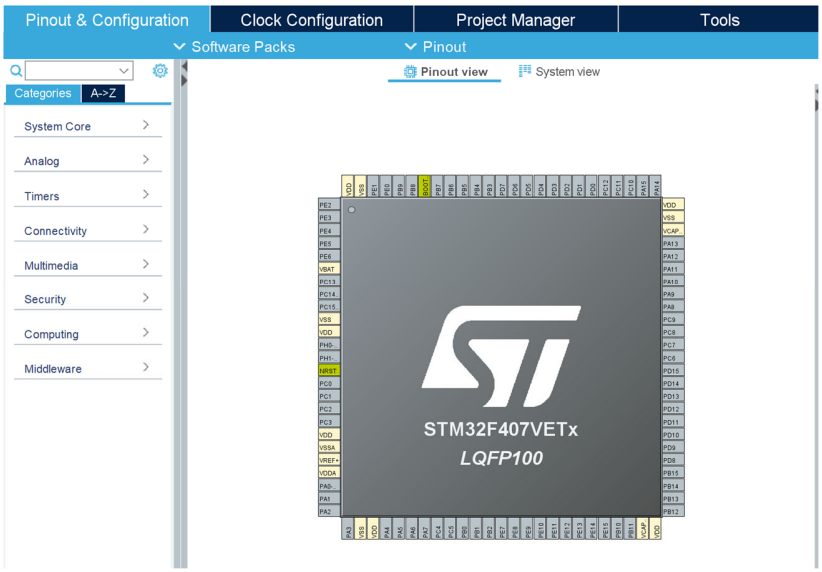

Take note of the file that has extension "ioc" (which contains the GUI of the selected microcontroller seen above). You will use this file to configure the microcontroller (e.g., I/O pins) as needed.

There are also some c source files generated during the setup, including a "main.c" which contain the necessary system related code for initialization and start-up. More code will be added to these files after you configure the microcontroller. (This is also the file that you will later add your own application code).




## 1.3 Configuration the I/O interface

To configure the microcontroller, select the ioc file which will display the following GUI screen.

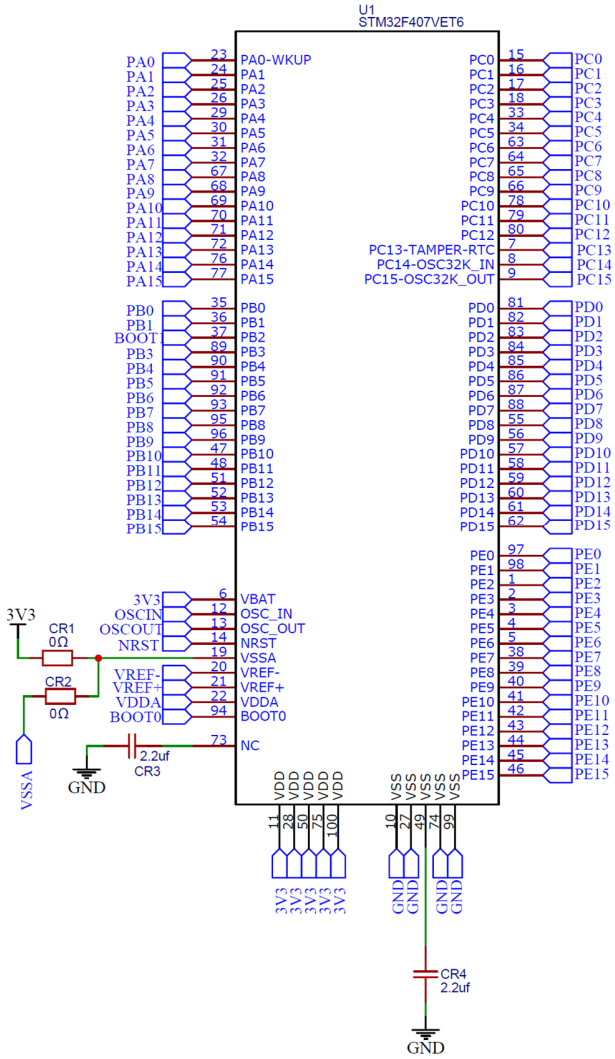



In this exercise, you will first configure the relevant I/O interfaces for the following operations:

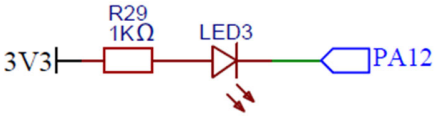
a. A LED that will be blinked by your program
b. A buzzer that will control by your program
c. A Serial Wire Debugger (SWD) to download your code to the microcontroller

To do so, you will need to refer to the schematics of the STM32F4 board that you are using. You then select the corresponding I/O pins of the microcontroller and configure them accordingly.

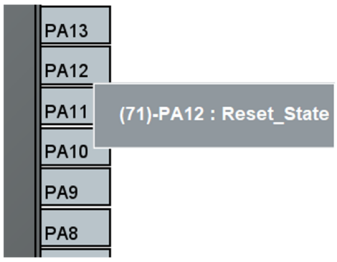
First, the following shows the overall I/O pins of the microcontroller on the circuit board.



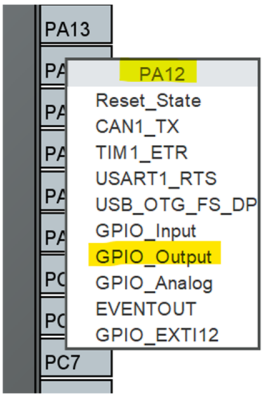

a. To begin, you will configure the LED3 on the board that is controlled through PA12 (meaning pin 12 of Port A) as shown below.



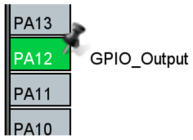

Select PA12 of the microcontroller in the IDE



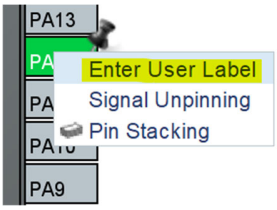
Click on the pin, and a list of options will be displayed.

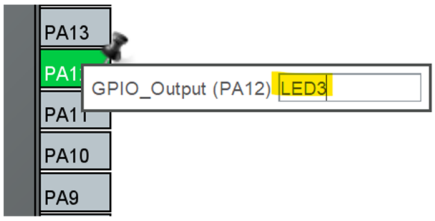


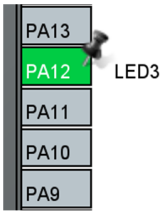

Select the configuration that you need for pin PA12, which in this case, you will select it to be an output pin: GPIO_Output



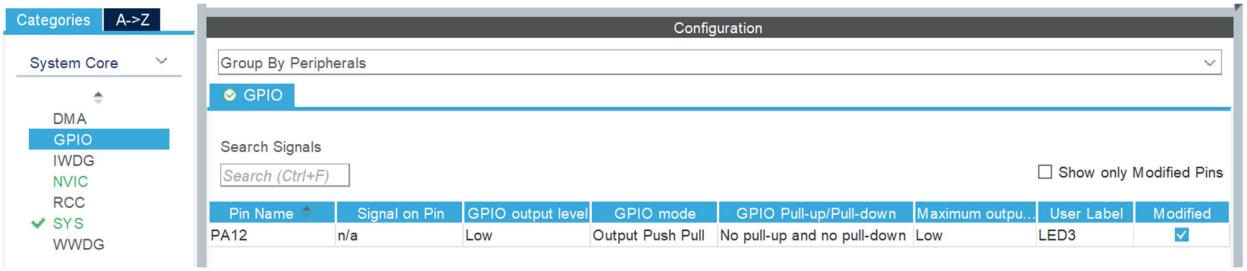

You can change the label of the pin to make it more descriptive by right clicking on the pin. E.g., change it to “LED3”.



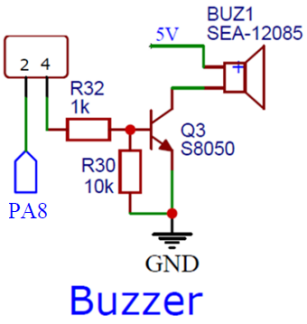



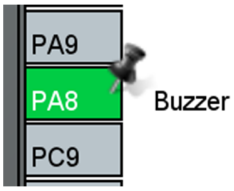

You can see more detail about PA12 that you have setup through the “System Core → GPIO” tab on the left panel . (E.g., the type of output this pin is configured with.)



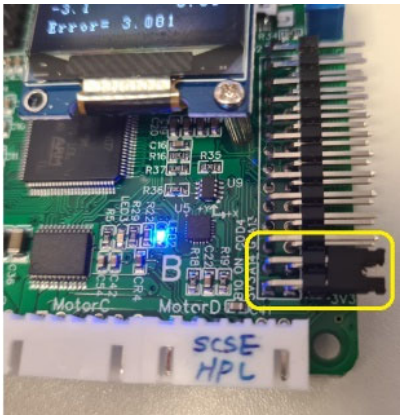

This completes the configuration setup of PA12 to drive the LED3 device. But before you proceed to generate the initialization code for this I/O pin, you will setup two other interfaces that are to be used in this exercise.

b. Using the same procedure, setup the buzzer that is connected to PA8 as shown below.

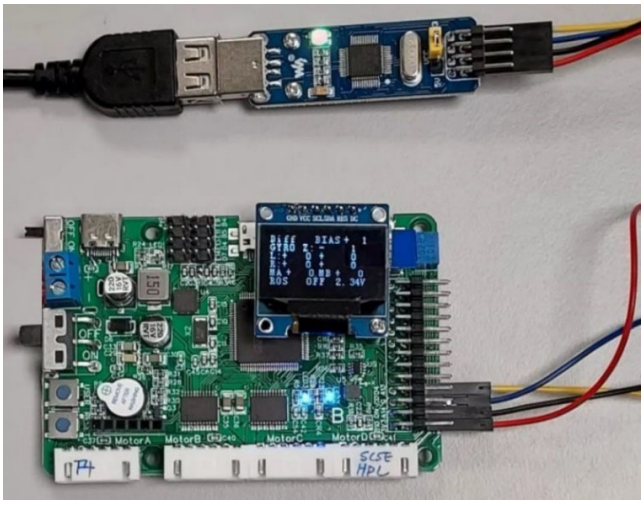



Note that a header connector is needed between pin 2 and pin 4 of the side header pins (see below photo) for the buzzer to be driven by PA8 signal to produce the audio sound.

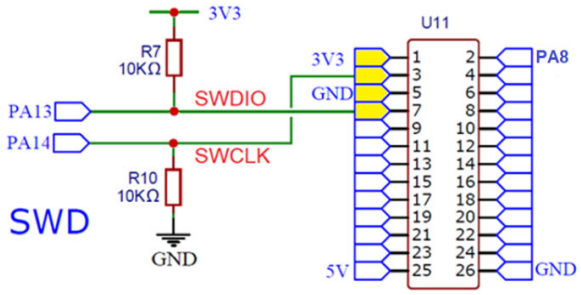


c. You will also be using a Serial Wire Debug (SWD) ST-Link board to download your code onto the board. It also enables you to debug your program - such as to set breakpoints in your code and single step your code and inspect the values of the variables used in your program.



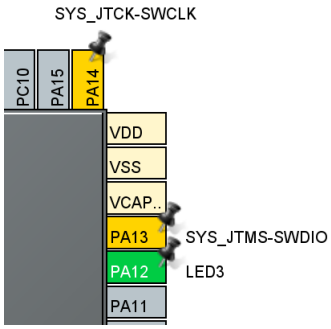
Note the order of the wires' connections from the debugger to the STM32F4 board.

The SWD interface consists of two wires (excluding the power and ground wires): SWCLK and SWDIO.



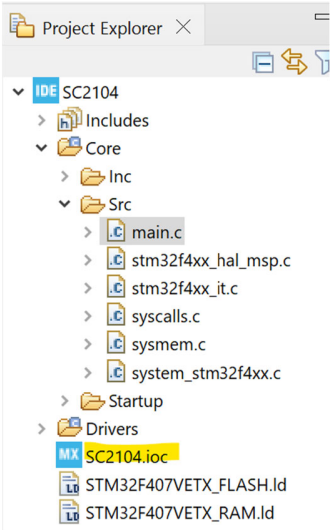

On the STM32F4 circuit board, these two signals are to be generated by PA13 and PA14 as indicated in the above schematic.

Furthermore, PA13 and PA14 have built-in SWD support functions which you can select accordingly as shown below.

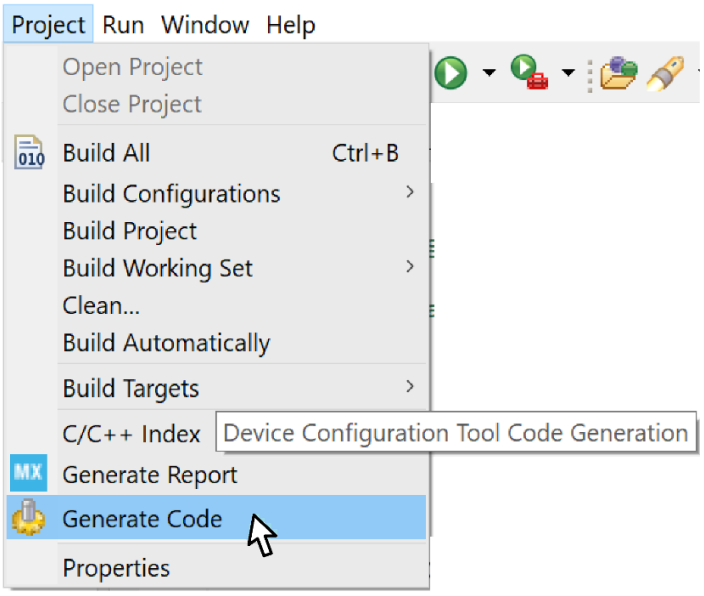



## 1.3 Generating the initialization code for the configured I/O pins

At this stage, what you have configured are stored in the ioc file. The next step is to let the IDE generates the corresponding C initialization code correspond to the configured hardware. These autogenerated code will be added to the **main.c** file (which you can further add your own code to implement specific applications.)



To generate the initialization code for the configured hardware, click on the "Project" Tab and choose "Generate Code".




(Alternatively, just execute "File → Save" using the "File" tab.)

Once the code generation process is completed, you will notice new initialization code and functions are added somewhere inside the main.c file, such as the following that corresponds to the configurations of the two pins that you have specified above.

```
/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOA_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOA, Buzzer_Pin|LED3_Pin, GPIO_PIN_RESET);

  /*Configure GPIO pins : Buzzer_Pin LED3_Pin */
  GPIO_InitStruct.Pin = Buzzer_Pin|LED3_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);

}
```

You can now add your own application code to the main.c file to operate the interfaces.

## 2. Writing your application code

Click and open the main.c file, and look for the following lines of code within its "main" function.

```
  /* Private user code ---------------------------------------------------------*/
  /* USER CODE BEGIN 0 */

  /* USER CODE END 0 */

  /**
    * @brief  The application entry point.
    * @retval int
    */
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();
  /* 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 */
}
```

Notice that there are various comments indicating where user code should be added. Code that are added by you within the indicated "USER CODE BEGIN - END" sections will be preserved during future code generation by the IDE (i.e., when you need to re-configure the I/O pins through the ioc file, which you will be doing later). User code that are not within the "USER CODE" sections will be deleted when new initialization code is generated through the ioc file.

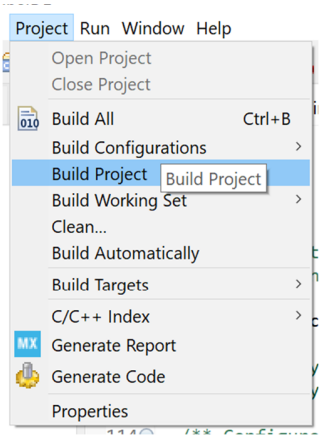
The STM32CubeIDE platform also provides various Hardware Abstraction Layer (HAL) set of ANSI-C MISRA-C 2004 compliant APIs that can be used by upper layer software (such as the user application – see Appendix IV) to access the underlying hardware. (Google for "STM32F4 HAL Documentation" for complete list of the HAL APIs.)

The following shows the few lines of HAL instructions that can be used to blink the LED3 that you configured earlier. (It should be obvious how these lines of code drive the LED.)

```
91   /* Infinite loop */
92   /* USER CODE BEGIN WHILE */
93   while (1)
94   {  HAL_GPIO_WritePin(GPIOA, GPIO_PIN_12, GPIO_PIN_SET);
95      HAL_Delay(1000); //1000 msec
96      HAL_GPIO_WritePin(GPIOA, GPIO_PIN_12, GPIO_PIN_RESET);
97      HAL_Delay(1000);
98     /* USER CODE END WHILE */
99
100    /* USER CODE BEGIN 3 */
101  }
```

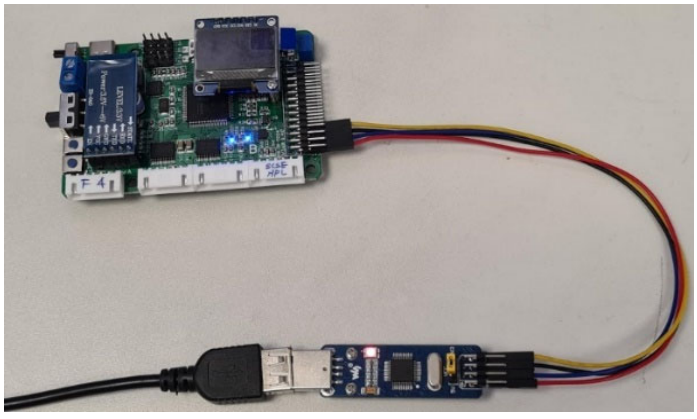
## 3. Executing your application program

To run the C program, you will need to first compile the program. Go to the "Project" tab and select "Project → Build Project" option in the IDE.

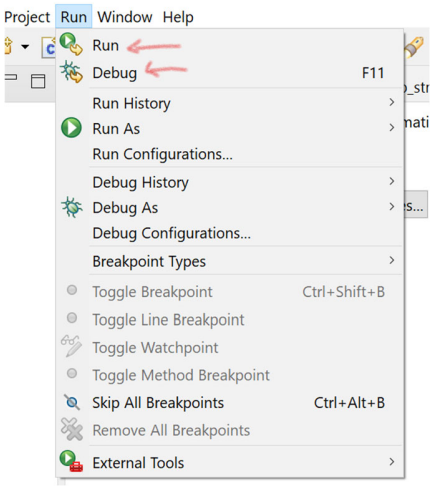



If there is no error, you can then download the code to the STM32F4 board by using the SWD ST-Link debugger board.

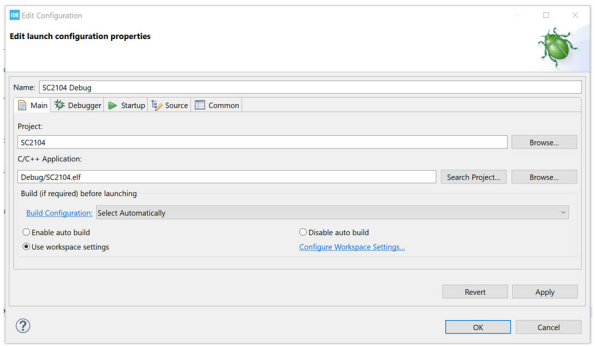
First connect the ST-Link board between the computer and the STM32F4 board. (Check that the order of the wires is connected accordingly as shown in the diagram below. With this setup, the ST-Link board is able to power up the STM32F4 board without the need of external power source.)

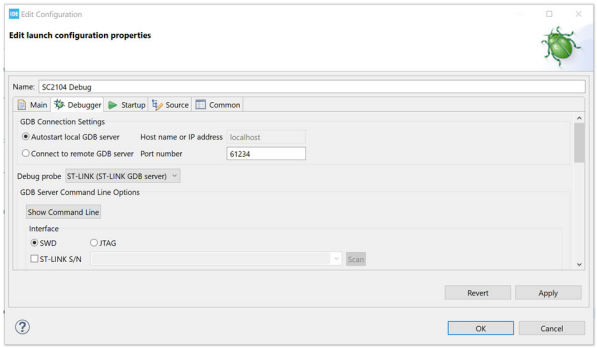


To download the code, on the "Run" tab, select either "Run → Run", or "Run → Debug".



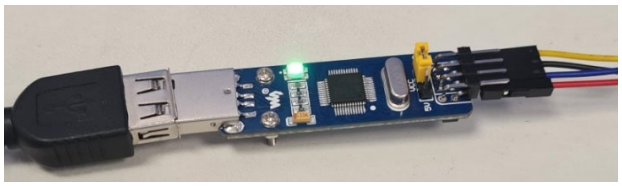

The first time you download the code, a pop-up message will appear, requesting you to confirm the ST-Link debug board setup.



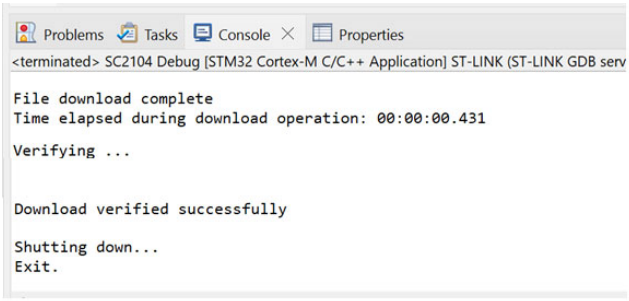




Typically, there is no change needed to be made with the default setup, and you can just proceed to download the code.

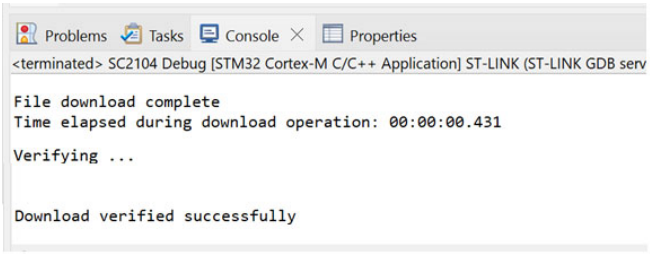
During the download, the LED on the ST-Link debugger board will blink between red and green color. (If not, unplug its USB cable on the PC and re-insert it to retry)



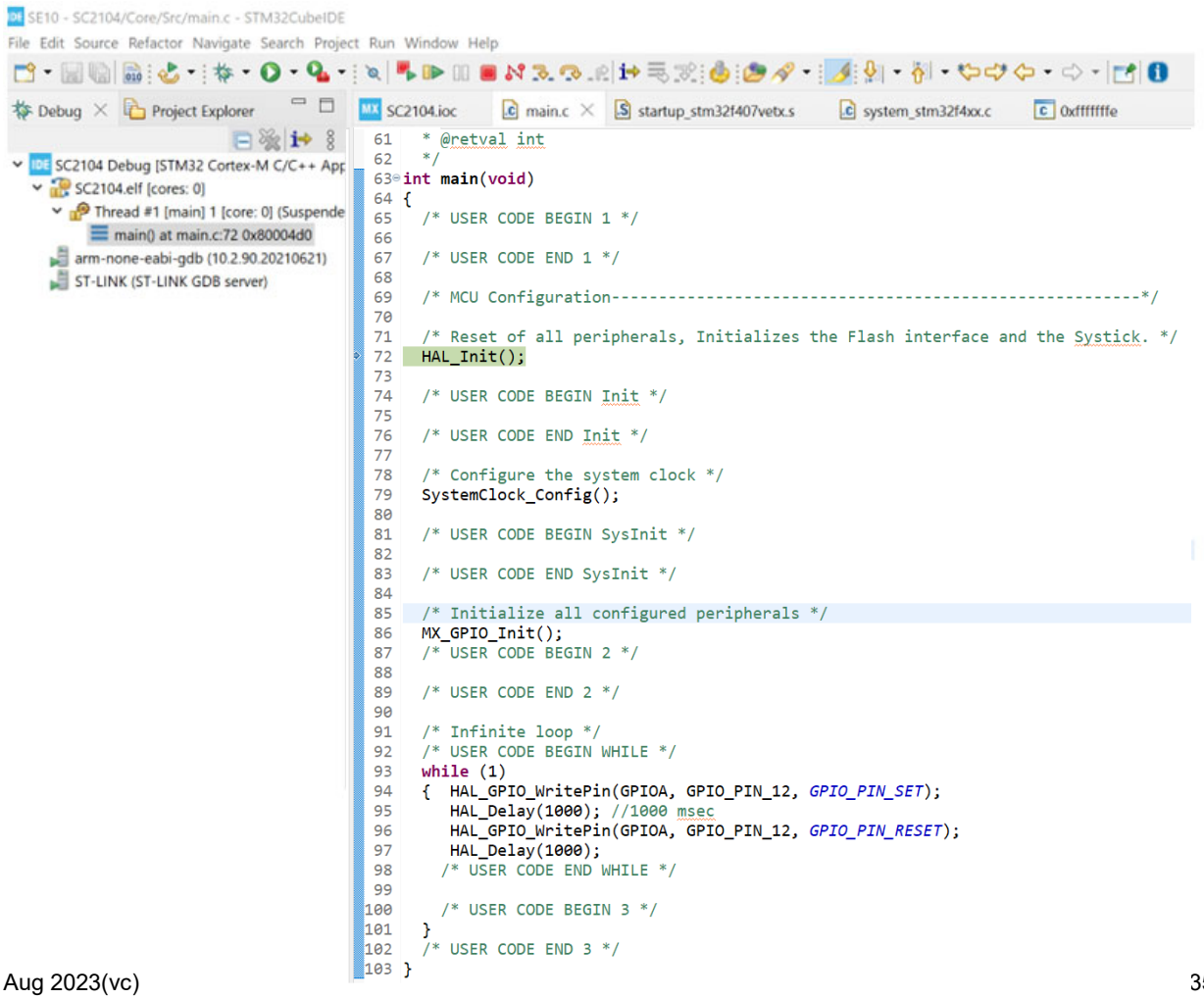
a. If you use "Run → Run", option, the debugger will exit once the program is downloaded successfully (with its LED remain red color). The STM32F4 board should immediately execute the program. (If not, you can unplug and plug back the ST-Link board to reset the STM32F4 board and make it start executing the program)



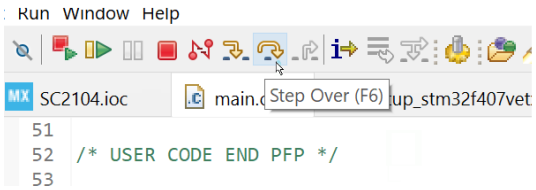

b. If you choose "Run → Debug", the debugger will pause the execution of the program after it is downloaded (with the LED on the ST-Link board continue to blink between red and green color)



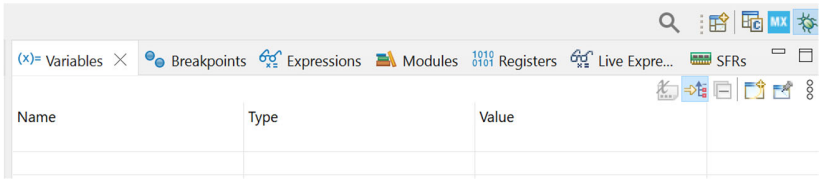

The following screenshot shows the program that is paused at the **`HAL_Init()`** statement when executing in Debug mode.



You can then proceed to single step the program by using "Step Over" function, and monitor the execution of the code line-by-line (and observe the contents of any Variables as the code is executed)

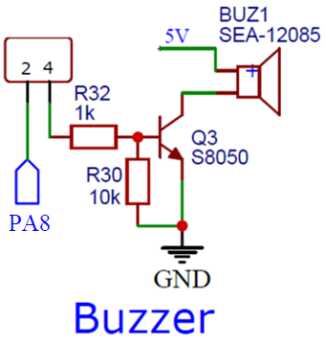


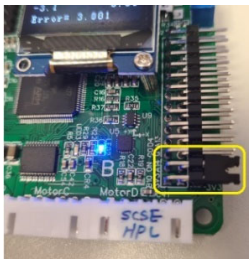

The panel on the right will allow you to select and observe the values of Variables etc. when you single step the program.




## 4. Practice #1 - Buzzer

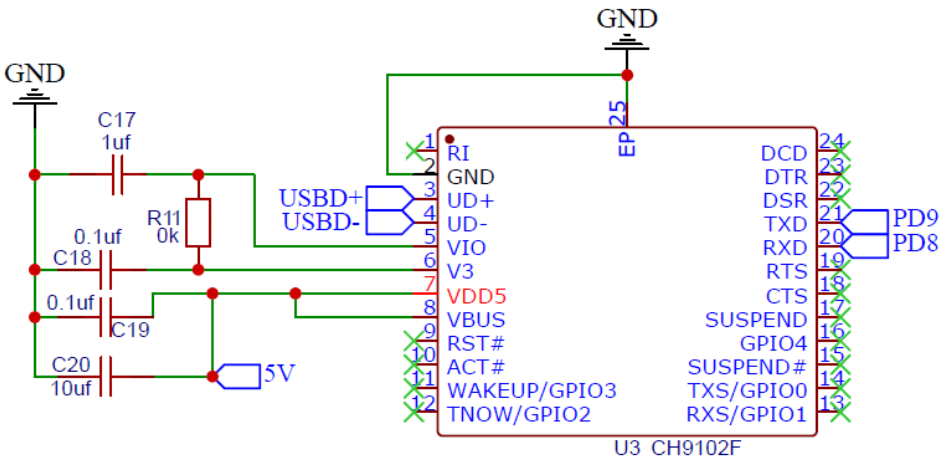
Add the code needed to operate the buzzer based on the configuration you have done earlier. The Buzzer will sound when it is turned on.



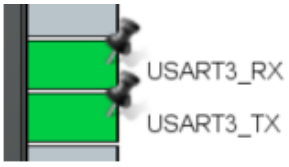
## 5. Practice #2 - Serial Port Communication

You will next develop an application to send data through the serial port (to the PC). This is to be done through **USART3** serial port based on the circuit shown below.



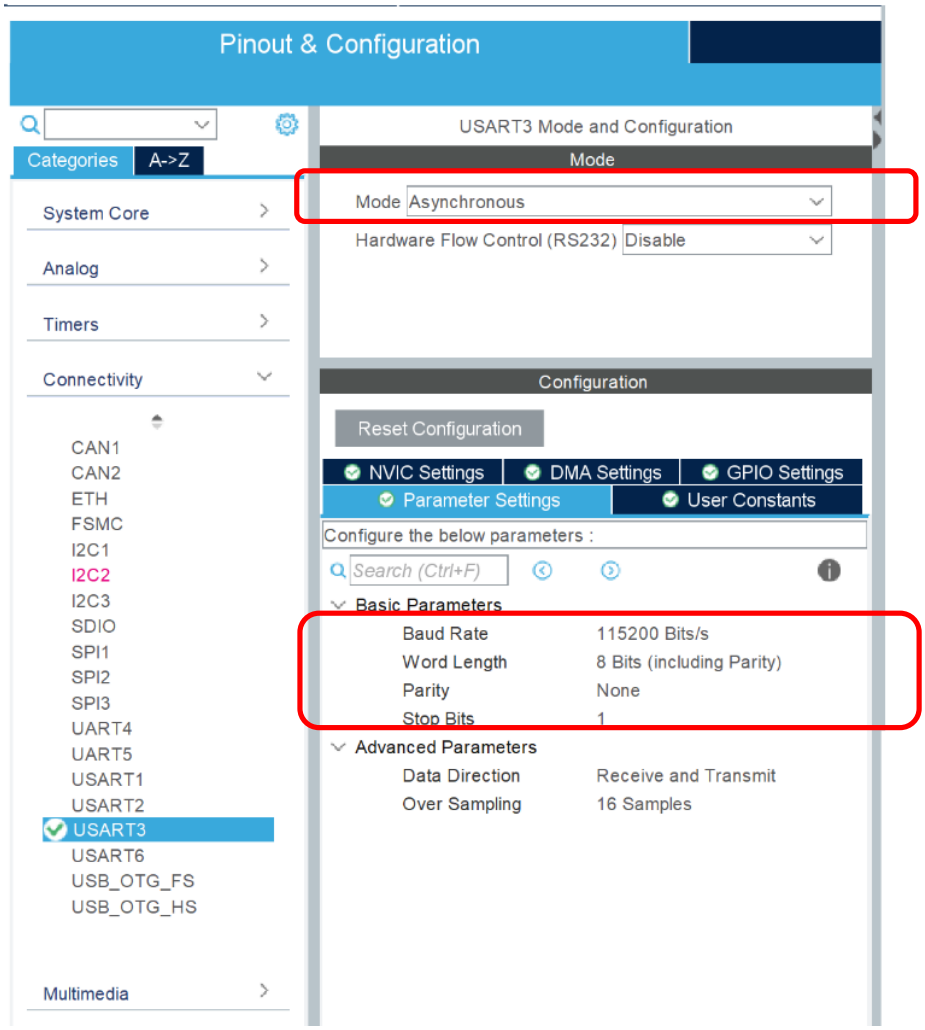
The USART3 (Universal Synchronous/Asynchronous Receiver/Transmitter) serial communication is interfaced through the two pins: *Pxxx?* and *Pxxx?* as shown in the above schematic.

To use this serial interface, you need to first configure these two pins. This is to be done through the ioc file as before.




Select these two pins of the microcontroller, and assign them the USART3's RX and TX functions (these serial port functions are built-in for these two pins).

In addition to setting up the RX and TX function to the pins, you also need to enable the USART3 serial port through the "Connectivity" entry on the left panel as shown below, by changing USART3's Mode to "Asynchronous" and configure the settings as shown below. (What do these parameters means?)




As before, do a "File → Save" to let the IDE auto-generates the code needed to activate the serial port.

Extra lines of code should now appear in the GPIO_Init function showing the configuration of the two pins for the serial port operation. Another function, MX_USART3_UART is also automatically generated which is used to initialize the serial port operation.

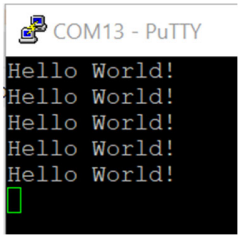
```
154 /**
155   * @brief USART3 Initialization Function
156   * @param None
157   * @retval None
158   */
159 static void MX_USART3_UART_Init(void)
160 {
161
162   /* USER CODE BEGIN USART3_Init 0 */
163
164   /* USER CODE END USART3_Init 0 */
165
166   /* USER CODE BEGIN USART3_Init 1 */
167
168   /* USER CODE END USART3_Init 1 */
169   huart3.Instance = USART3;
170   huart3.Init.BaudRate = 115200;
171   huart3.Init.WordLength = UART_WORDLENGTH_8B;
172   huart3.Init.StopBits = UART_STOPBITS_1;
173   huart3.Init.Parity = UART_PARITY_NONE;
174   huart3.Init.Mode = UART_MODE_TX_RX;
175   huart3.Init.HwFlowCtl = UART_HWCONTROL_NONE;
176   huart3.Init.OverSampling = UART_OVERSAMPLING_16;
177   if (HAL_UART_Init(&huart3) != HAL_OK)
178   {
179     Error_Handler();
180   }
181   /* USER CODE BEGIN USART3_Init 2 */
182
183   /* USER CODE END USART3_Init 2 */
184
185 }
```

### 5.1 Coding and executing the serial program

The following two lines of code show examples of how messages can be sent through the serial port from STM32F4 board. Include them in the appropriate places in your STM32F4 program.

```
70 int main(void)
71 {
72   /* USER CODE BEGIN 1 */
73      uint8_t sbuf[15] = "Hello World!\n\r";
74   /* USER CODE END 1 */
            :
      HAL_UART_Transmit(&huart3, sbuf, sizeof(sbuf), HAL_MAX_DELAY);
```

Connect the STM32F4 board USART3 serial port to the computer using a USB cable. (See Appendix II on how to find the COM port to be used for this.)

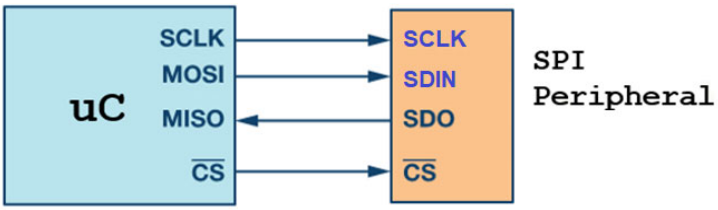



Run the PuTTY program on the computer to display the messages sent by your STM32F4 program. Alternatively, you can open a console window in STM32CubeIDE for this – see Appendix II.

## 6. Practice #3 – SPI interface to the OLED Display

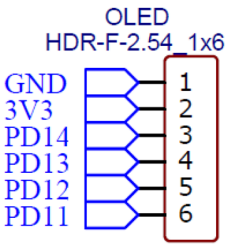
The on-board OLED Display is driven by a controller chip (SSD1306) to display various messages sent by the STM32F4 microcontroller using SPI based transferring protocol.

Standard SPI interface uses 3 signals, the SCLK (Clock), SDIN (Data in) and SDO (Data out), plus a CS# (Chip Select). (Refer Lecture notes for operation detail)




As the OLED Display only accepts and displays data sent by the STM32F4 microcontroller, and does not send data to the microcontroller, SDO is not used in the STM32F4 circuit board interface. Furthermore, the CS# signal is tied low on the OLED Display such that it is always enabled, ready to accept data and command from the microcontroller.

The OLED Display on the STM32F4 board is to be driven by PD11 to PD14 using the SPI signaling protocol.



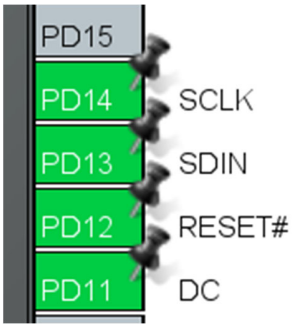


However, PD11 to PD14 do not have built-in SPI function. As such, in this exercise, you will need to code a function to perform the SPI signaling using the “bit banging” approach.

### 6.1 Configurating the interface pins

Firstly, configure the pins PE5-PE8 as general output to serve the following functions:

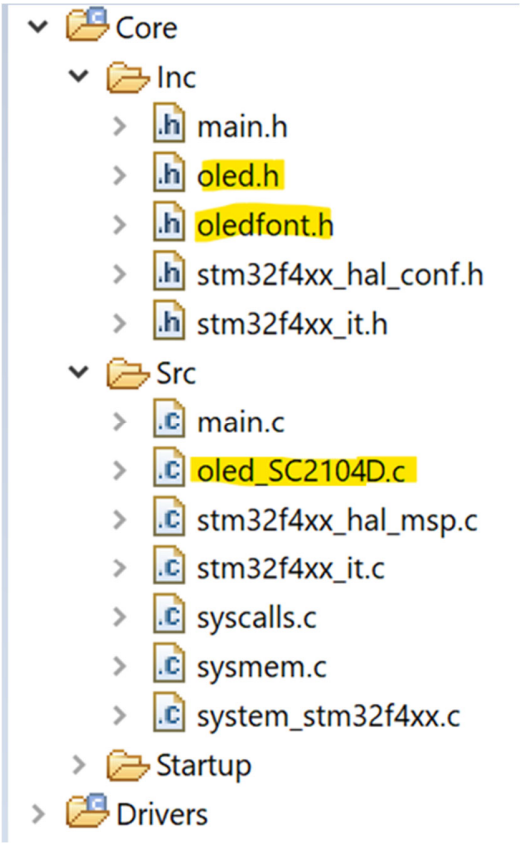
- PD14 = SCLK - the clock signal
- PD13 = SDIN - the data in signal
- PD12 = RESET# - reset signal, asserted to reset the OLED display
- PD11 = DC - signal to indicate whether the SDIN signal is Data (to be displayed) or Command# (to access the display’s built-in registers).



## 6.2 Library functions and sample program

For this OLED Display interface, library functions to display messages on the Display are provided, together with the necessary driver codes to operate the Display, except the function "`OLED_WR_Byte`" that emulate the SPI signaling which you will develop.

These OLED Display related functions and driver code are to be added to the project using the three files as highlighted below. (These three files are available in the NTULearn courses site, under the "Laboratory" folder).




The following shows the prototypes of the various library functions that are used to operate the OLED Display (these are declared in the "oled.h" file)

```
//OLED Control Functions
void OLED_WR_Byte(uint8_t dat,uint8_t cmd);  // to be implemented
void OLED_Display_On(void);
void OLED_Display_Off(void);
void OLED_Refresh_Gram(void);
void OLED_Init(void);
void OLED_Clear(void);
void OLED_DrawPoint(uint8_t x,uint8_t y,uint8_t t);
void OLED_ShowChar(uint8_t x,uint8_t y,uint8_t chr,uint8_t size,uint8_t mode);
void OLED_ShowNumber(uint8_t x,uint8_t y,uint32_t num,uint8_t len,uint8_t size);
void OLED_ShowString(uint8_t x,uint8_t y,const uint8_t *p);
void OLED_Set_Pos(unsigned char x, unsigned char y);
```

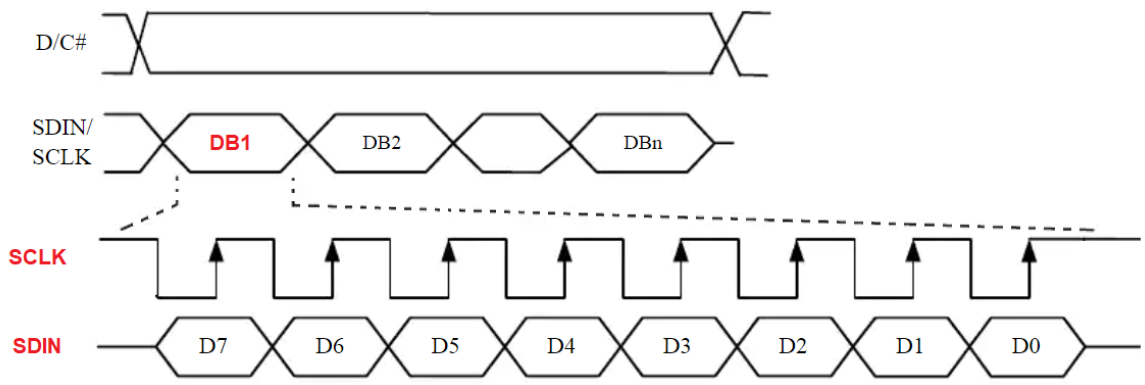
The followings show the sample code of a program (in main.c file) that are used to display messages on the OLED Display, using some of the library functions provided.

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

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include "oled.h"
/* USER CODE END Includes */

                :

int main(void)
{
  /* USER CODE BEGIN 1 */
     uint8_t sbuf[15] = "Hello World!\n\r";
     uint8_t *OLED_buf;
  /* USER CODE END 1 */

                :

/* USER CODE BEGIN 2 */
OLED_Init();
OLED_ShowString(10, 5, "SC2104/CE3002"); // show message on OLED display at line 10

OLED_buf = "Lab 1"; // anther way to show message through buffer
OLED_ShowString(40,30, OLED_buf); //another message at line 40
OLED_Refresh_Gram();
/* USER CODE END 2 */
```

## 6.3 The `OLED_WR_Byte()` function

The `OLED_WR_Byte()` in the oled_SC2104D.c is the key function called by all the OLED library APIs to generate the SPI signalling for sending the data to the OLED display.

The following timing diagram shows how the SCLK and the SDIN are to perform the SPI signalling to transfer the data.



Below is the skeleton code of the **`OLED_WR_Byte()`** function that you will need to complete in this exercise, to generate the two signals SCLK and SDIN as shown above.

```
/**********************************************************************
Function: Send the data/command to the OLED Display controller using SPI bit-banging
Input   : dat: data/command on SDIN pin
          DataCmd: data/command# on D/C# pin
               1 => sending data
               0 => sending command
Output  : none

**********************************************************************/

void OLED_WR_Byte(uint8_t dat,uint8_t DataCmd)
{
    uint8_t i;

    if(DataCmd == 1)          // Data write
      OLED_RS_Set();          // Set the D/C# line
    else                      // Command write
      OLED_RS_Clr();          // Clear the D/C# line

    for(i=0;i<8;i++)
    { // Complete the code below
            :
            :
            :
    }

    OLED_RS_Set();          // Keep the D/C# line high upon exit
}
```

You can call the following functions to toggle the two signals:

```
OLED_SCLK_Set();    // set SCLK
OLED_SCLK_Clr();    // clear SCLK
OLED_SDIN_Set();    // Set SDIN
OLED_SDIN_Clr();    // clear SDIN
```

## 6.4 Program development

(a) Download the following files from the NTULearn course site (under the "Laboratory → Practical Exercises" folder) and save it to a local directory on the computer.


(b) Import (see Appendix) all three files into your project.

(c) Complete the **`OLED_WR_Byte()`** function in the oled_SC2104D.c file as described in 6.3.

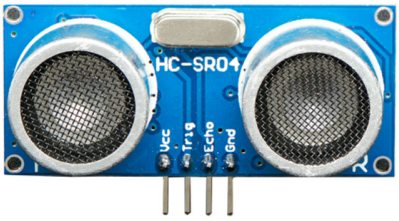
(d) Add the message displaying code shown in 6.2 to your **`main.c`**.

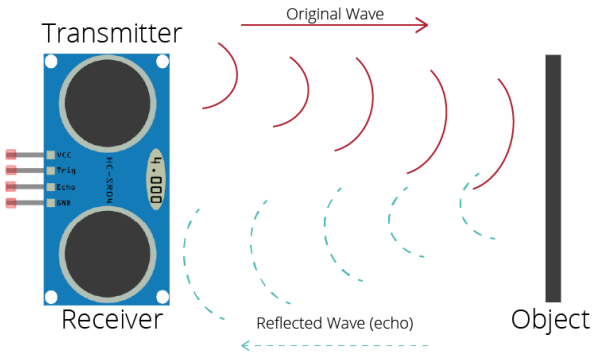
(e) Compile and download your program to the STM32F4 board to see that it is executing as expected.

If it does not work, this is the good opportunity for you to learn how to use the single stepping feature of the debugger to debug your program.

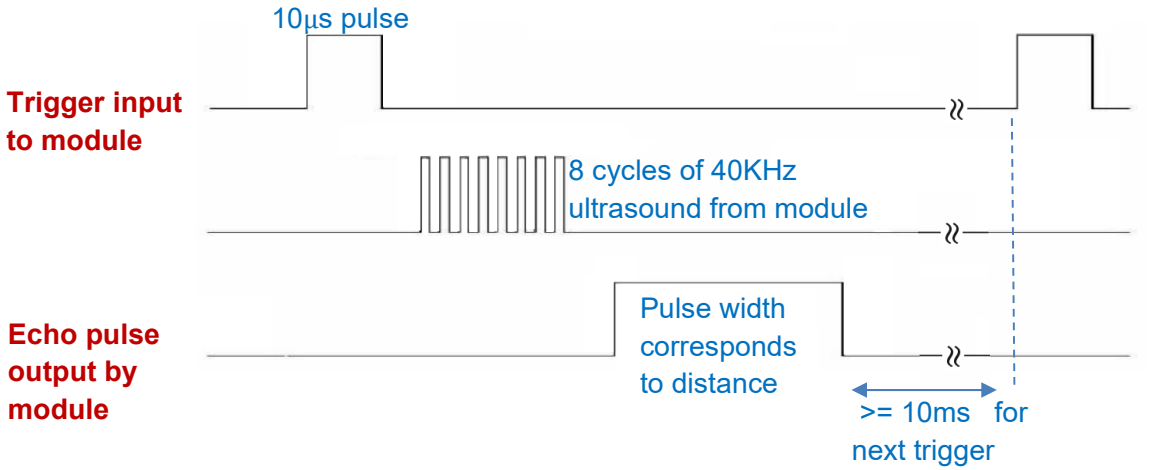
## 7. Practice #4 – Interfacing HC-SR04 Ultrasonic Distance Sensor

The HC-SR04 ranging module is an ultrasonic sensor that can be used to measure the distance of an object located in front of the sensor (within the range of 2cm to 4m).




HC-SR04 consists of two ultrasonic transducers. One acts as a transmitter that converts trigger signal into 40 KHz ultrasonic sound pulses. The other acts as a receiver that listens for the transmitted pulses being reflected (and output a pulse at the Echo pin).



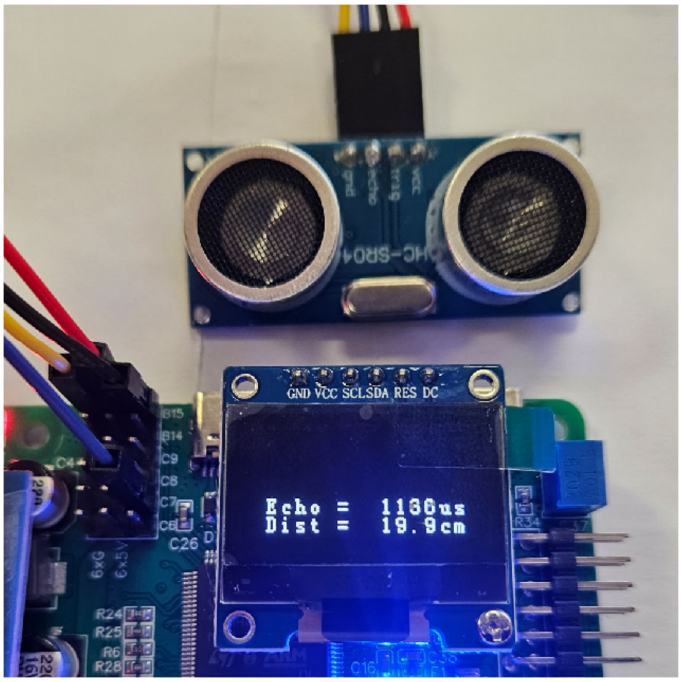

Upon receiving a 10µs pulse at the Trigger input, the module emits eight 40KHz sonic burst, and assert the Echo signal (to logic '1'). When the receiver receives the reflected pulses, it resets the Echo signal (to logic '0'). The pulse width at the Echo pin is hence proportional to the distance of the object in front of the module, which can be calculated by using the formula:

Distance = Echo pulse duration * Sound velocity / 2 ;

where the Sound velocity = 343 meters/sec (@ 20$^{O}$C).

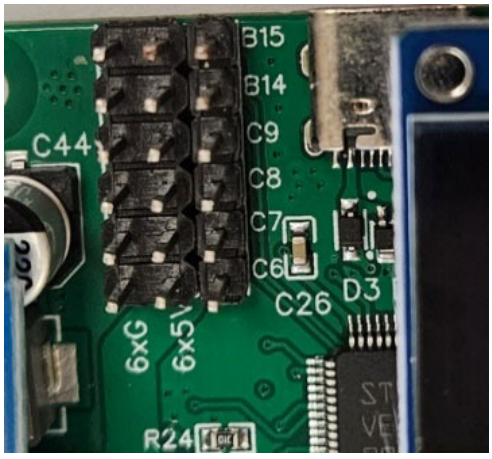
Echo output will be terminated(reset) after 36ms if there is no reflected signal received (e.g., no object detected, or object farther than 4 meters).

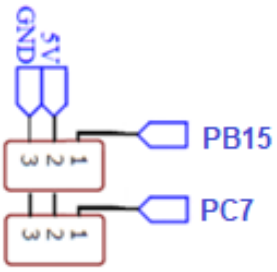
In this exercise, you are to interface a HC-SR04 device to the STM32F board and code a program that can detect the object and display the distance measured on the OLED display.




## 7.1 Interface configurations

The interface between the HC-SR04 device and the STM32F board can be done through the header pins that are conveniently available on the STM32F4 board as shown below.



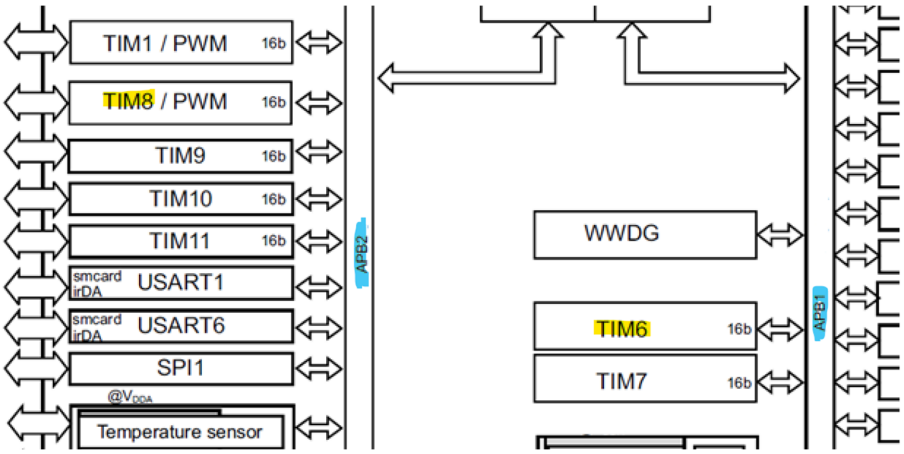

Connect the HC-SR04's Vcc, Gnd and Trig pins to the first row of the header (Gnd, 5V and B15), and the Echo pin to C7.

From the device operation's timing diagram, we can see that we need to be able to
i) generate a 10usec pulse and send it to the Trig pin
ii) measure the pulse width of the signal output by the Echo pin

We will achieve both by using Timers available in the microcontroller.

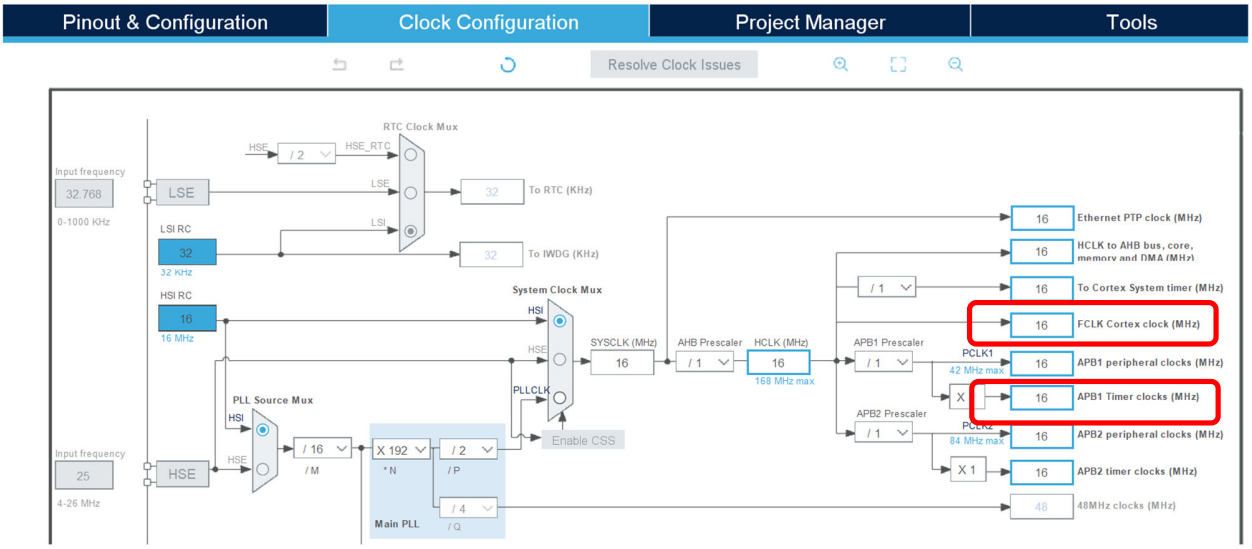
To generate a 10usec pulse, we will use the basic timer TIM6 and configure it to run at 1µsec resolution.

As can be seen in the following internal block diagram of the microcontroller, TIM6 is connected to the interface bus APB1 within the microcontroller.

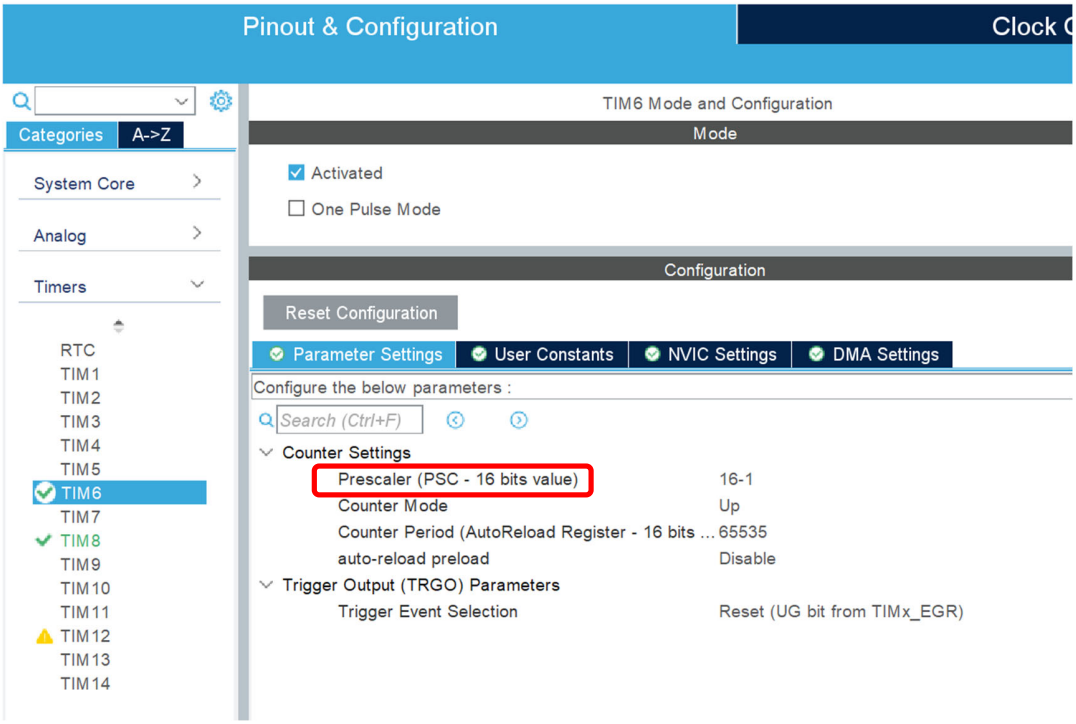



Depending on the frequency of the APB1 bus, we can set the timer to run at certain resolution with appropriate settings.

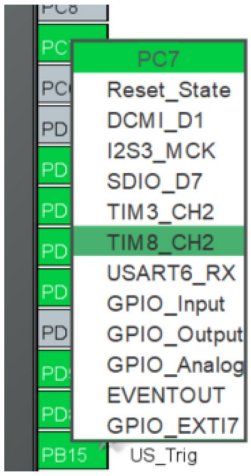
To find the frequency that APB1 bus is running: Open the ioc file, click on the “Clock Configuration”.




For APB1 that is running at 16MHz (what is the clock period?), Timer TIM6 can be set to run at 1µsec resolution by multiplying its clock period by 16 (why?) through its Prescaler setting (setting = 16-1 because timer starts running from 0).

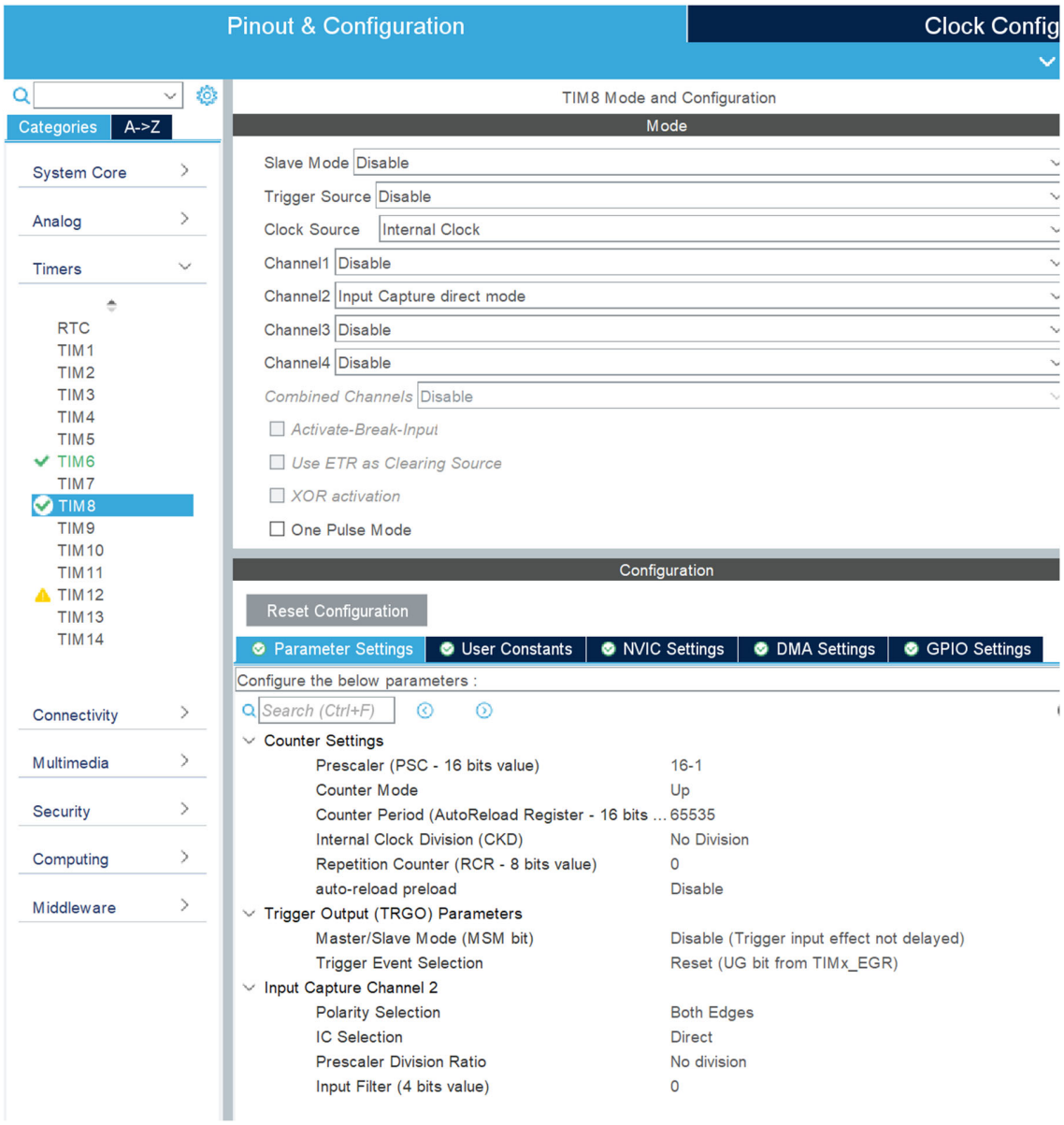


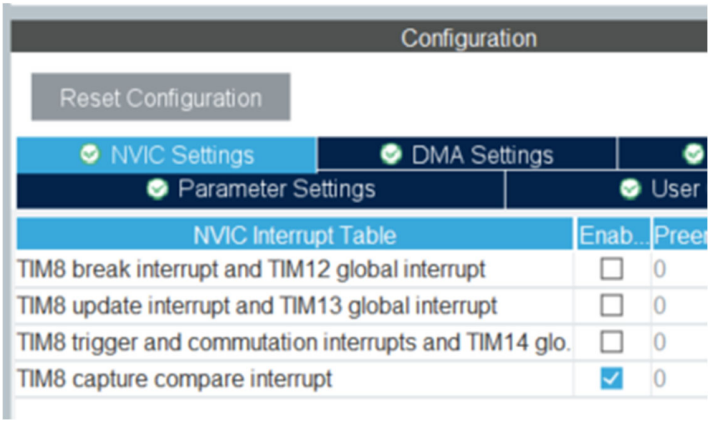

For the Echo pin, we will use PC7 which can be configured to operate with Channel 2 of timer TIM8 running in Input capture mode.




From the earlier Block Diagram and Clock Configuration diagrams, we can see that TIM8 is connected to APB2 (which is also running at 16MHz in the diagram). We will using this Internal Clock setting to configure it to run at 1µsec resolution

To detect both the rising edge and falling edge of the Echo signal, set the Polarity Selection to be "Both Edges" and enable its interrupt.


Pinout & Configuration
Clock Config
Categories
A->Z
System Core
Analog
Timers
RTC
TIM1
TIM2
TIM3
TIM4
TIM5
TIM6
TIM7
TIM8
TIM9
TIM10
TIM11
TIM12
TIM13
TIM14
Connectivity
Multimedia
Security
Computing
Middleware
TIM8 Mode and Configuration
Mode
Slave Mode Disable
Trigger Source Disable
Clock Source Internal Clock
Channel1 Disable
Channel2 Input Capture direct mode
Channel3 Disable
Channel4 Disable
Combined Channels Disable
Activate-Break-Input
Use ETR as Clearing Source
XOR activation
One Pulse Mode
Configuration
Reset Configuration
Parameter Settings
User Constants
NVIC Settings
DMA Settings
GPIO Settings
Configure the below parameters :
Search (Ctrl+F)
Counter Settings
Prescaler (PSC - 16 bits value) 16-1
Counter Mode Up
Counter Period (AutoReload Register - 16 bits ... 65535
Internal Clock Division (CKD) No Division
Repetition Counter (RCR - 8 bits value) 0
auto-reload preload Disable
Trigger Output (TRGO) Parameters
Master/Slave Mode (MSM bit) Disable (Trigger input effect not delayed)
Trigger Event Selection Reset (UG bit from TIMx_EGR)
Input Capture Channel 2
Polarity Selection Both Edges
IC Selection Direct
Prescaler Division Ratio No division
Input Filter (4 bits value) 0


Configuration
Reset Configuration
NVIC Settings
DMA Settings
Parameter Settings
User
NVIC Interrupt Table
Enab...
Preer
TIM8 break interrupt and TIM12 global interrupt 0
TIM8 update interrupt and TIM13 global interrupt 0
TIM8 trigger and commutation interrupts and TIM14 glo. 0
TIM8 capture compare interrupt 0

## 7.2 Program Logic

The following is the logic of the program that you can implement the application.

- Set the Trig signal (through PB15) low // just in case
- Wait for 50msec
- Set the Trig signal high
- Delay for 10usec
- Set the Trig signal low
- Wait for 50msec
- Read the Echo signal pulse width captured by the timer TIM8, which is processed by the Callback* Function `void HAL_TIM_IC_CaptureCallback()`
- Display the duration of the pulse width on the OLED display
- Calculate the distance and display it on the OLED display
- Repeat the above

*Callback Function: In microcontroller operation when an interrupt occurs, it causes the program to branch to the vector table, which in turn calls an interrupt service routine (ISR) to service the interrupt. This ISR typically will perform the necessary housekeeping task (e.g., save the necessary registers values, reset the interrupt etc.) and execute the code written by the programmer. However, this will require the programmer to dive into the detail of the low level interrupt mechanism in the microcontroller which may be rather complex.

The STM32CubeIDE 'shields' the programmer from these details. It generates an ISR to perform the necessary low level housekeeping tasks, and then call a function, the Callback function, which allow the programmer to focus on developing the code to handle the event that causes the interrupt. The Callback function will return to the ISR upon completing its execution, which in turn restore the state of the processor and enable the microcontroller to resume its normal operation.

*In case you wonder:* STM32CubeIDE libraries contain various predefined Callback functions but with empty code (i.e., they do not contain any code in their bodies and hence do nothing when called). These functions are declared as weak functions (using the __weak attribute). If you need to use such a function to implement your own version of Callback function, you only need to define it in your code (instead of overwriting the function provided in the library) and the linker will skip the weak version and use yours instead during compilation and linking.

## 7.3 Program development

Code a program based on the program logic as described in 7.2 above. The following are code snippets that you can use as reference to develop your program.

a) The following for reference is a function to implement a delay loop based on timer TIM6, which can be used to generate a 10µsec pulse.

```
void delay_us (uint16_t us)
{   // the following HAL functions can be used to produce a delay
    __HAL_TIM_SET_COUNTER(&htim6,xxx);  // set the counter value to xxx
    __HAL_TIM_GET_COUNTER(&htim6);      // read the counter
                :
}
```

b) Implement the following Callback function for timer TIM8 in your main.c file. Complete the code needed to calculate the pulse duration and store it in the global variable echo.

```
int echo = 0; // for storing the echo pulse duration
int tc1, tc2;

void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim){
  if(htim==&htim8){ // If the interrupt due to Timer TIM8:
     // (i) check whether interrupt is due to +ve edge or -ve edge
     // (ii) read the timer value and store it in tc1 (for +ve edge) or tc2 (for -ve edge)
     //      e.g. tc1=HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_2);
     // (iii)calaculate the duration of the pulse
     // (iv) store the value in the global variable echo
    }
}
```

c) The following are the code statements that are to be used to start the timers.

```
//start Timer
HAL_TIM_Base_Start(&htim6); // Timer 6 for delay_us routine
HAL_TIM_IC_Start_IT(&htim8, TIM_CHANNEL_2); // Timer 8 for capturing Echo pulse
```

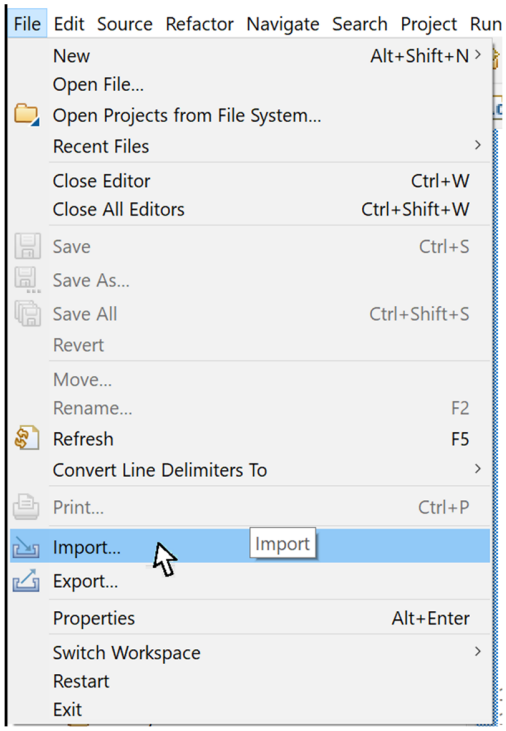
d) The following are the code statements to display the measured data on the OLED display.(See Appendix III on how to enable printing of floating point numbers for the **`sprintf()`** function used here)

```
char buf[15];     //buffer for displaying values on OLED display

sprintf(buf, "Echo = %5dus ", echo); // echo is in usec
OLED_ShowString(10, 40, &buf[0]); // display on line 40
OLED_Refresh_Gram();

sprintf(buf, "Dist = %5.1fcm ",  echo * xxxxx);  //calculate distance
OLED_ShowString(10, 50, &buf[0]); // diisplay on line 50
OLED_Refresh_Gram();
```

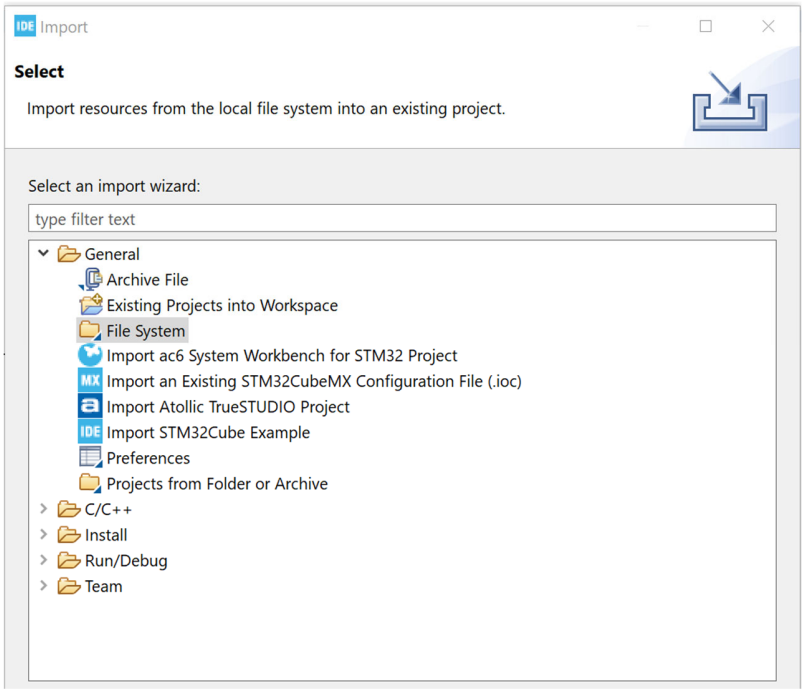
## Appendix I – Import of files into a project

File can be added to the project by using the "Import" function.

(a) Under the "File" tab, select the "Import" function



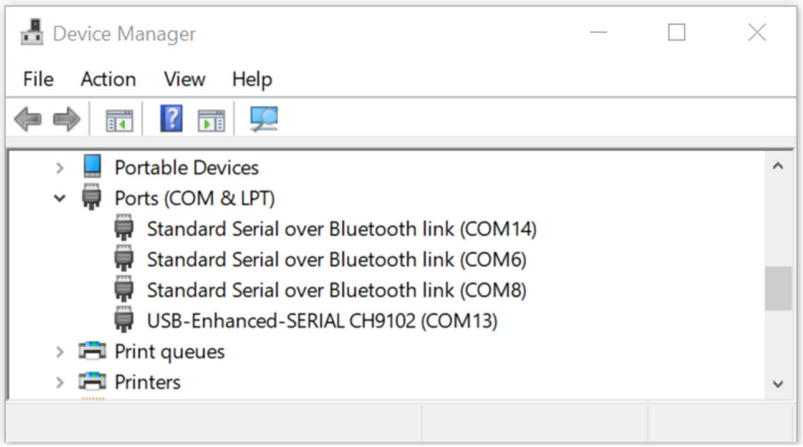

(b) Then select "General → File System" and follow the instructions to select the directory to import the file needed.



## Appendix II – Virtual COM and using Console in STM32Cube IDE

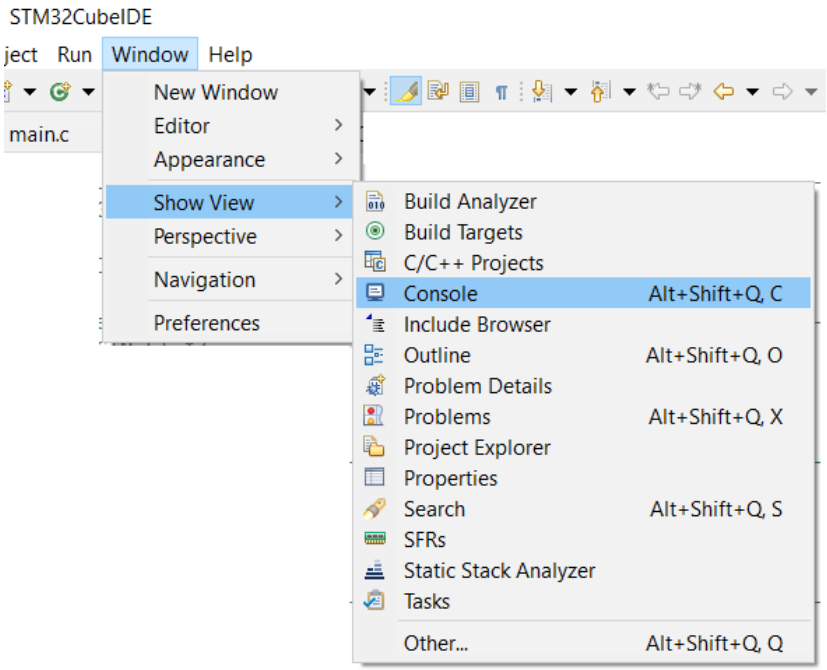
First, check the COM port that is activated by the OS to enable serial communication with the STM32F4 board once it is connected.

On the Windows PC, go to Device Manager and check the virtual COM Port number to be used (which one do you think it is?)

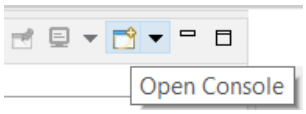


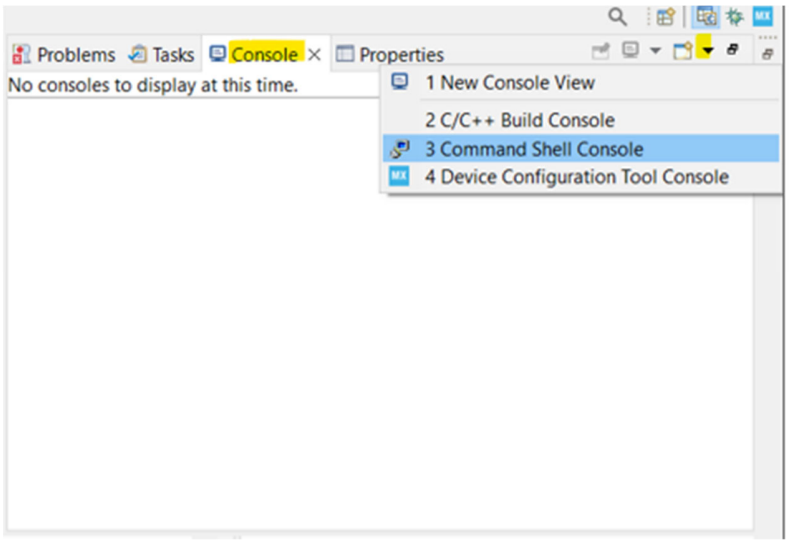

Instead of running Putty on the PC, you can open a console window directly in the STM32CubeIDE to display the message received through the serial port.

- Open a Console window in STM32CubeIDE: Window-> Show View->Console

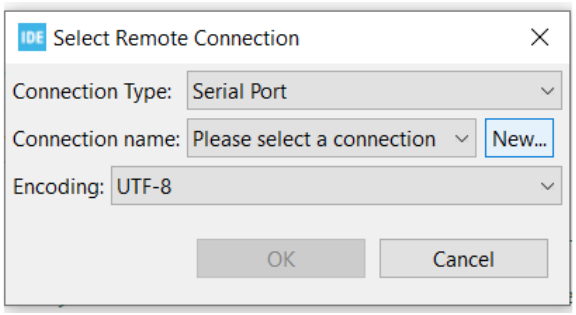


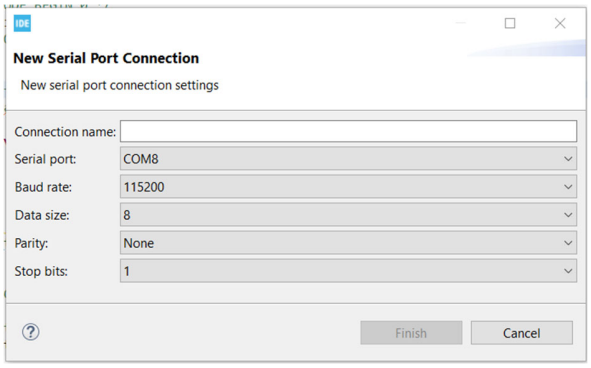
- In Console, click on the Open Console icon and select the Command Shell Console

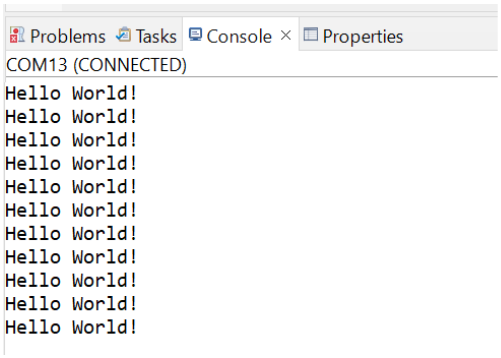



- Set the Remote Connection to Serial Port with UTF-8. Then Click on "New" to set the parameters for the serial port (using the Virtual COM port number indicated in the Device Manager).

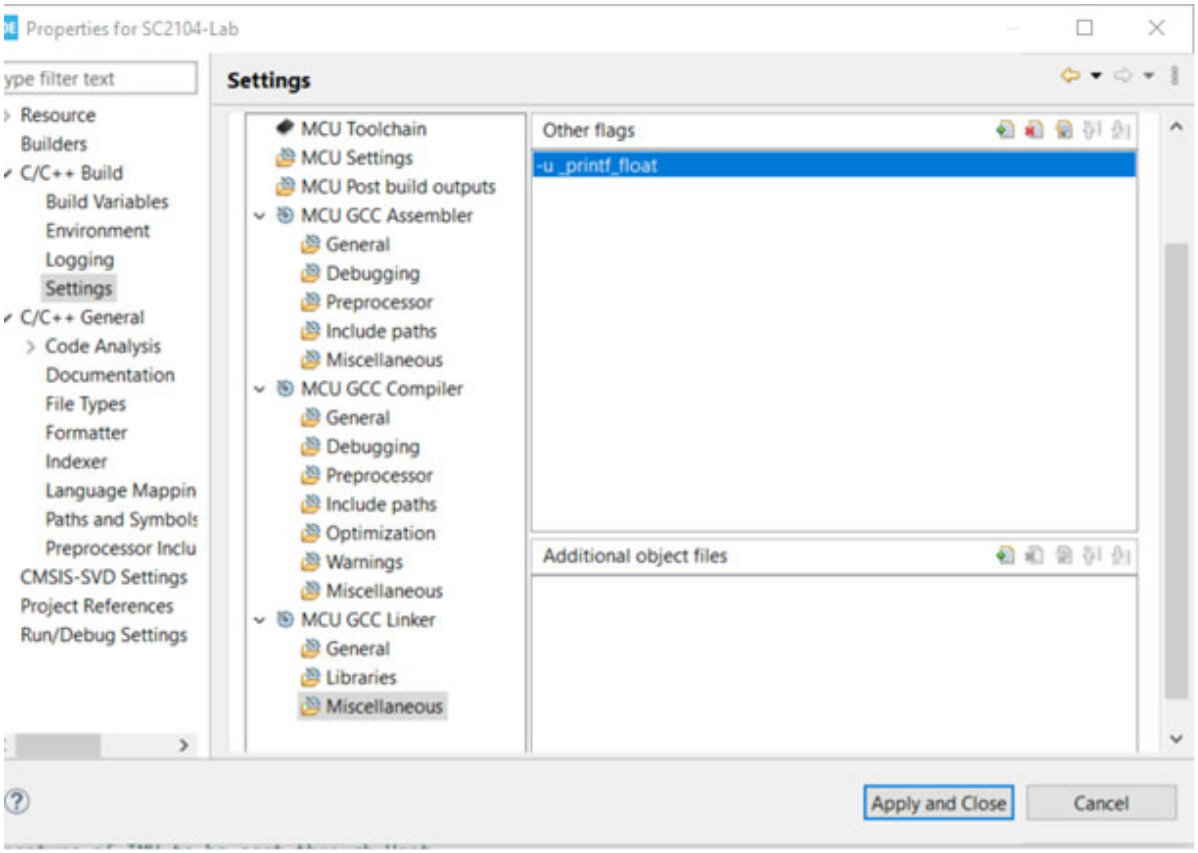


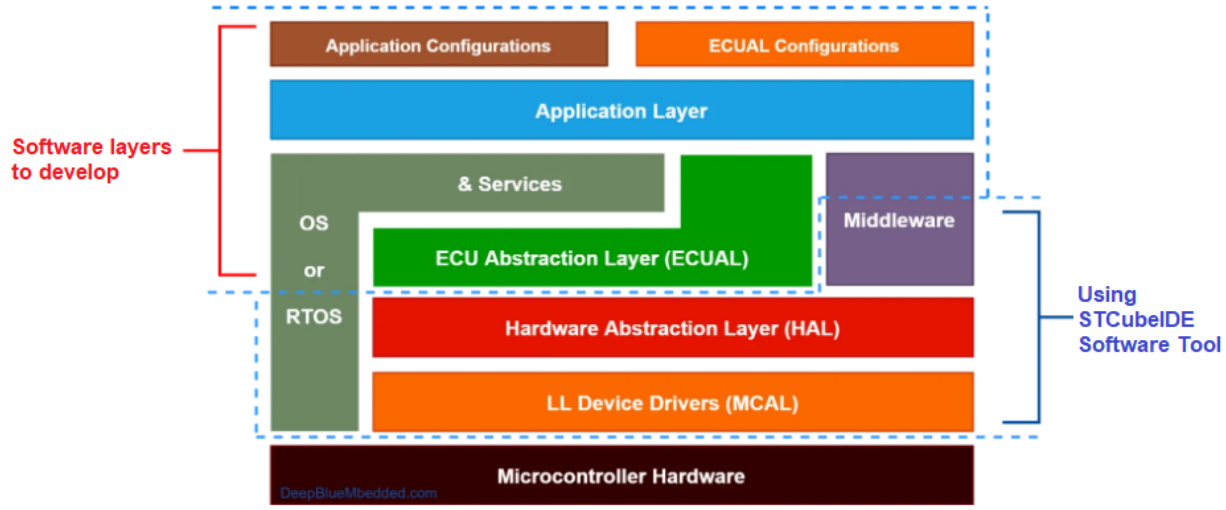



## Appendix III – Enabling the support to print floating point number

The floating-point support for printf is not enabled by default in STM32CubeIDE. To do so, we need to add the flag `-u _printf_float` to its linker option. (-u is to force linking and loading of library module that is undefined, in this case, `_printf_float` )



## Appendix IV - STM32 Software Development Overview




The STM32Cube Hardware Abstraction Layer (HAL) ensures maximized portability across the STM32 microcontrollers. But high-level APIs provided by the HAL may have extra more features than what you may actually need in some situations, may lead to using more memory space and executing some tasks a little bit slower due to the overhead of the functionalities embedded in the libraries.

You can use the LL drivers and optimize more at the register level in order to enhance memory utilization or the speed of execution. However, the application developed would not be easily portable across multiple targets.