TRAINING MANUAL

EE-310

DIGITAL SYSTEM ENGINEERING

Unit 9

Getting Started With RTOS

Time allocation: 3 Hours

Objectives

The aim of this module is to get immersed into embedded programming on a real hardware. To complete the basic workflow, simple applications are developed, implemented, and demonstrated in an Embedded System work environment. Experiment(s) in this module are conducted using **Real-Time OS (RTOS)** to demonstrate some of the most common practical applications.

Resources

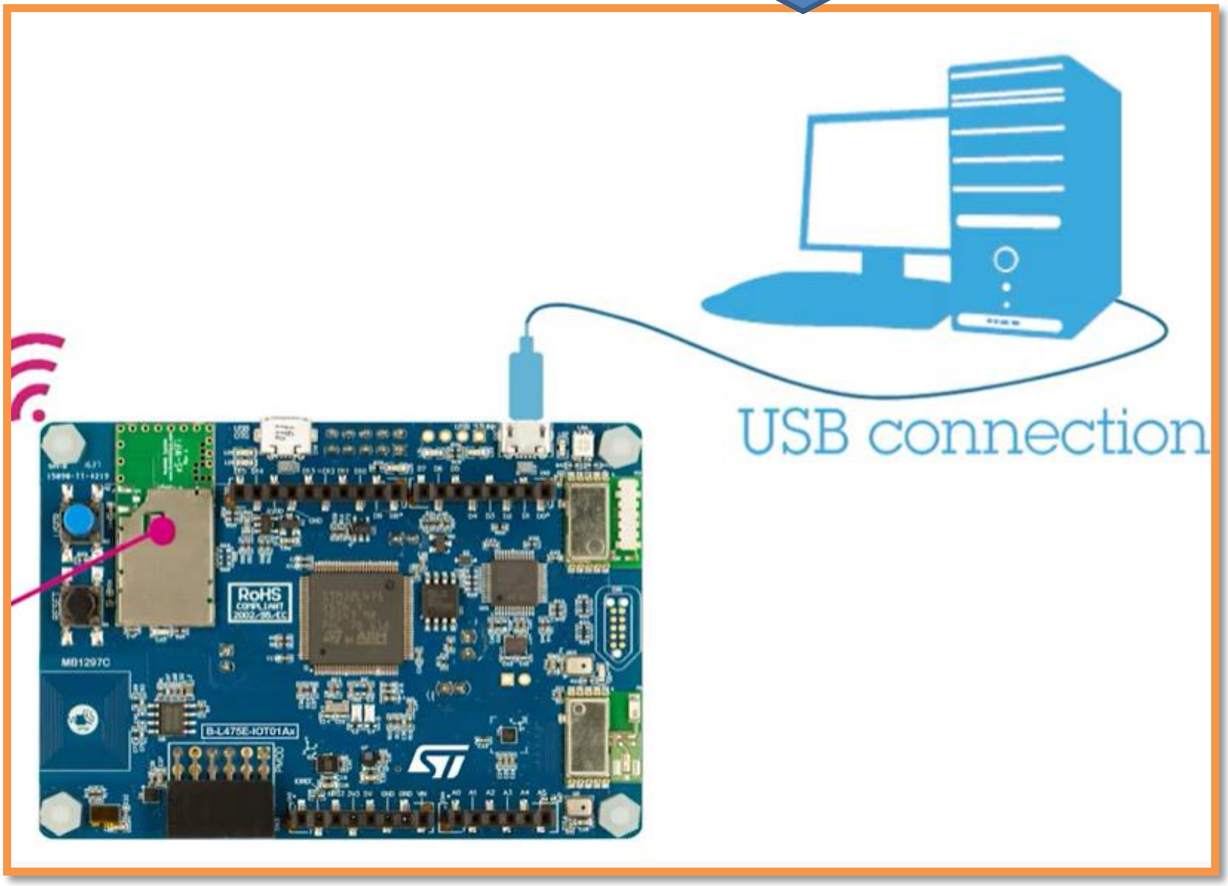
- Desktop PC / Laptop
- Software development Tools
- Embedded Kit (ARM Cortex Series)
- Jumper Wires / Breadboard / LEDs, Switches

Topics to be covered:

1. Getting Started a Tutorial Project
2. ARM Cortex M4 I/O Programming
3. GPIO (General Purpose I/O) Programming and Interfacing
4. Reading Switches and Displaying the same on LEDs
5. Standard Application(s) Interfacing and Programming
6. Realization of FreeRTOS (Real-Time Operating System)
7. Internet-of-Things (**IOT**) Application(s) Interfacing and Programming

Embedded System Setup

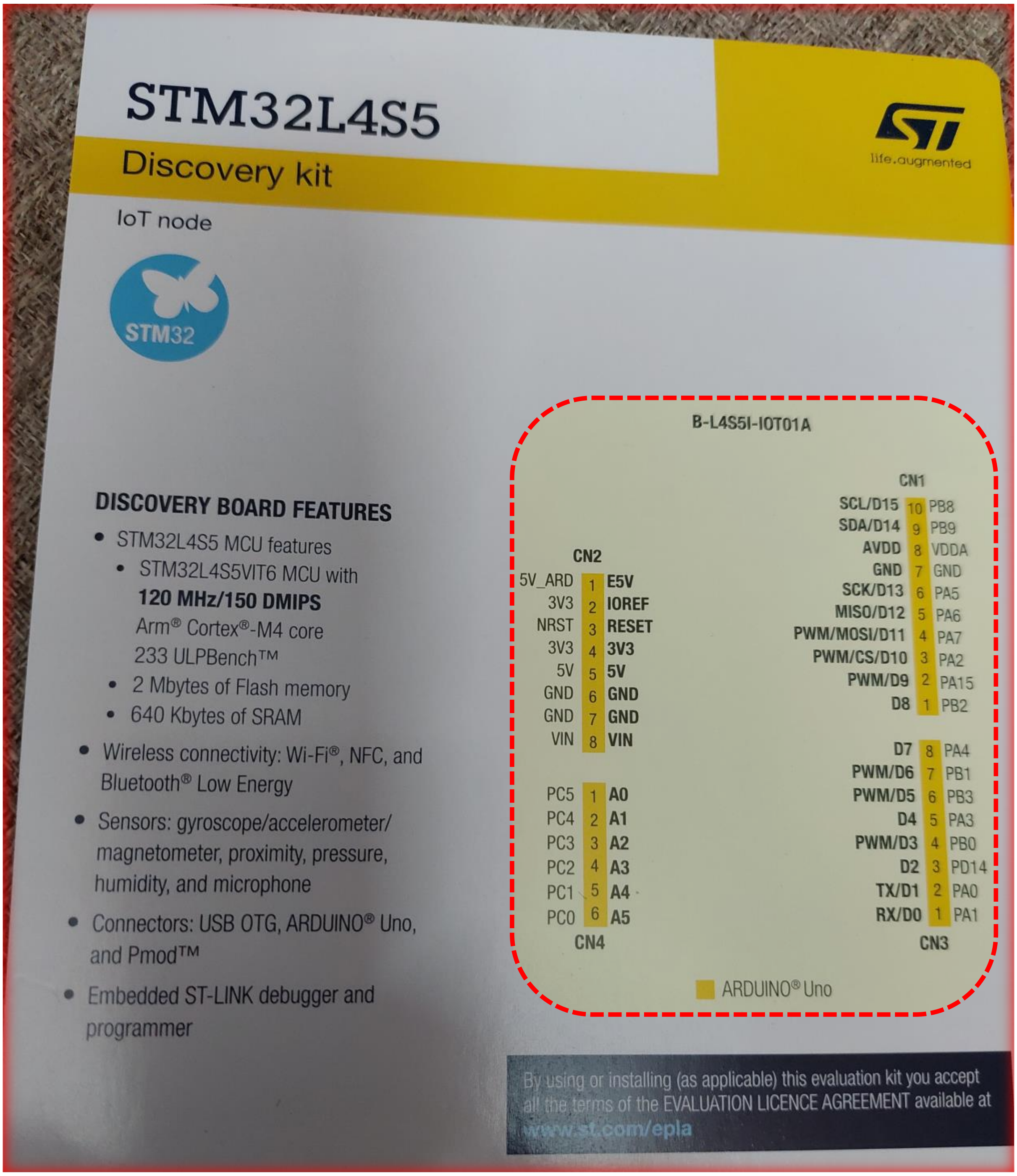
STM32 (ARM Cortex M4) Starter Kit - Development and Education Board



(STM32 µController)

Document: Datasheet (stm32l4s5) and Reference manual (stm32l4s5)

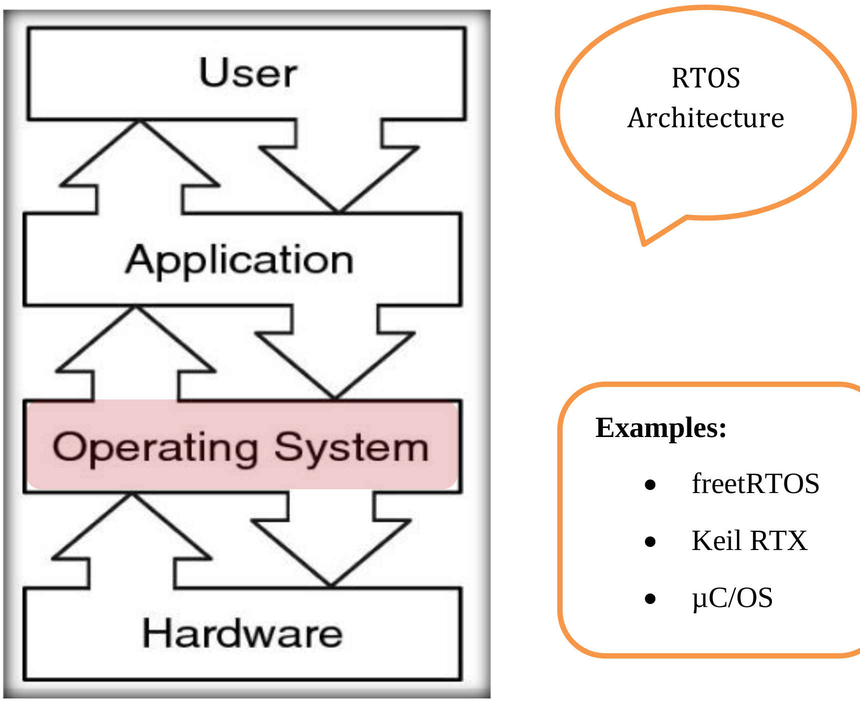
Expansion Connector



Getting Started With Embedded RTOS (freeRTOS)

What is an RTOS and Multitasking?

A RTOS is a real-time operating system which manages **software and hardware resources** on a computing system and provides services to application software which are **not** possible with bare metal.



Examples:

- freetRTOS
- Keil RTX
- µC/OS

A RTOS is basically a software component that rapidly switches between tasks, giving the impression that multiple programs are being executed at the same time on a single processing core.

In actual fact the processing core can only execute one program at any one time, and what the RTOS is actually doing is rapidly switching between individual programming threads (or Tasks) to give the impression that multiple programs are executing simultaneously.

When switching between **Tasks** the RTOS has to choose the most appropriate task to load next. There are several scheduling algorithms available. However, to provide a responsive system most RTOS use a pre-emptive scheduling algorithm.

In a pre-emptive system each Task is given an individual priority value. The faster the required response, the higher the priority level assigned. When working in pre-emptive mode, the task chosen to execute is the highest priority task that is able to execute. This results in a highly responsive system.

While selecting a RTOS, one of the most important considerations is what type of response is desired – Is a hard real time response required? This means that there are precisely defined deadlines that, if not met, will cause the system to fail. Alternatively, would a non-deterministic, soft real time response be appropriate? In which case there are no guarantees as to when each task will complete.

The choice of RTOS can greatly affect the development of the design.

By selecting an appropriate RTOS the developer gains:

- A Task based design that enhances modularity, simplifies testing and encourages code reuse;
- An environment that makes it easier for engineering teams to develop together;
- Abstraction of timing behaviour from functional behaviour, which should result in smaller code size and more efficient use of available resources.

Peripheral support, memory usage and real-time capability are key features that govern the suitability of the RTOS. Using the wrong RTOS, particularly one that does not provide sufficient real time capability, will severely compromise the design and viability of the final product.

The RTOS needs to be of high quality and easy to use. Developing embedded projects is difficult and time consuming – the developer does not want to be struggling with RTOS related problems as well. The RTOS must be a trusted component that the developer can rely on, supported by in-depth training and good, responsive support. FreeRTOS could be one of best choices amongst so many in the field.

What is FreeRTOS?

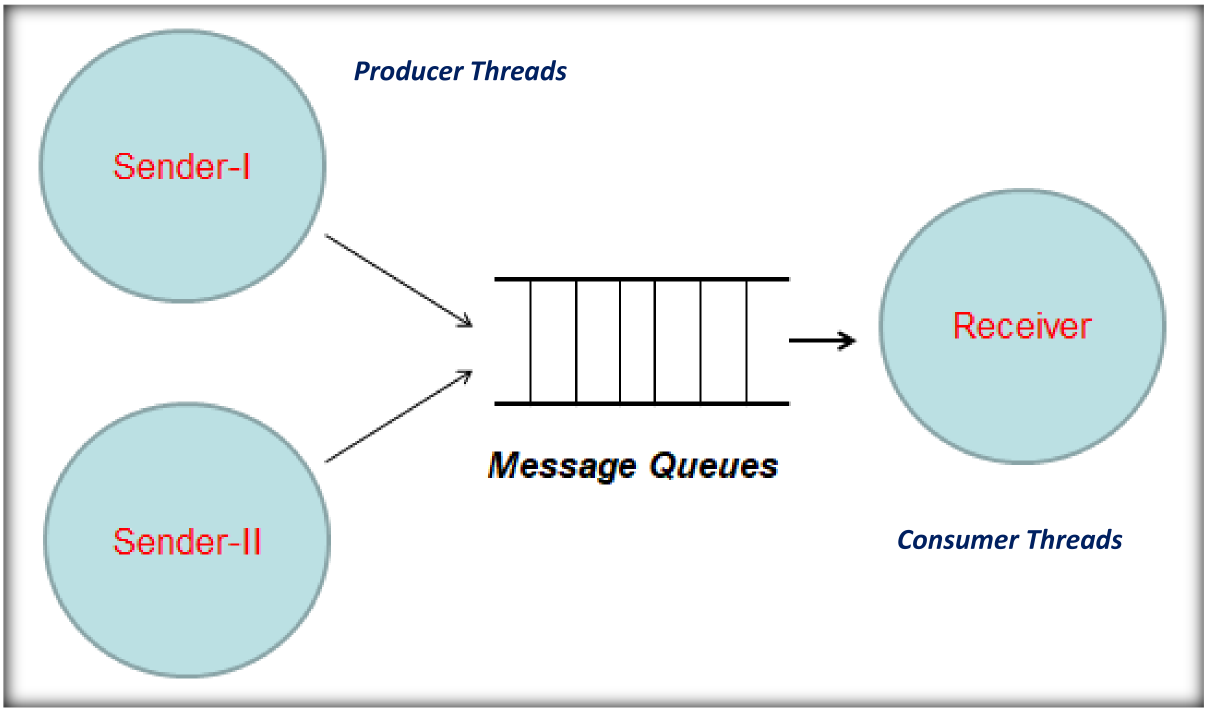
FreeRTOS is a class of RTOS that is designed to be small enough to run on a microcontroller (μC). A microcontroller is a small and resource constrained processor that incorporates, on a single chip, the processor itself, read only memory (ROM / Flash) to hold the program to be executed, and the random access memory (RAM) needed by the programs it executes. Typically the program is executed from the read only memory. One of the main attractions in freeRTOS is its free of cost licensing model.

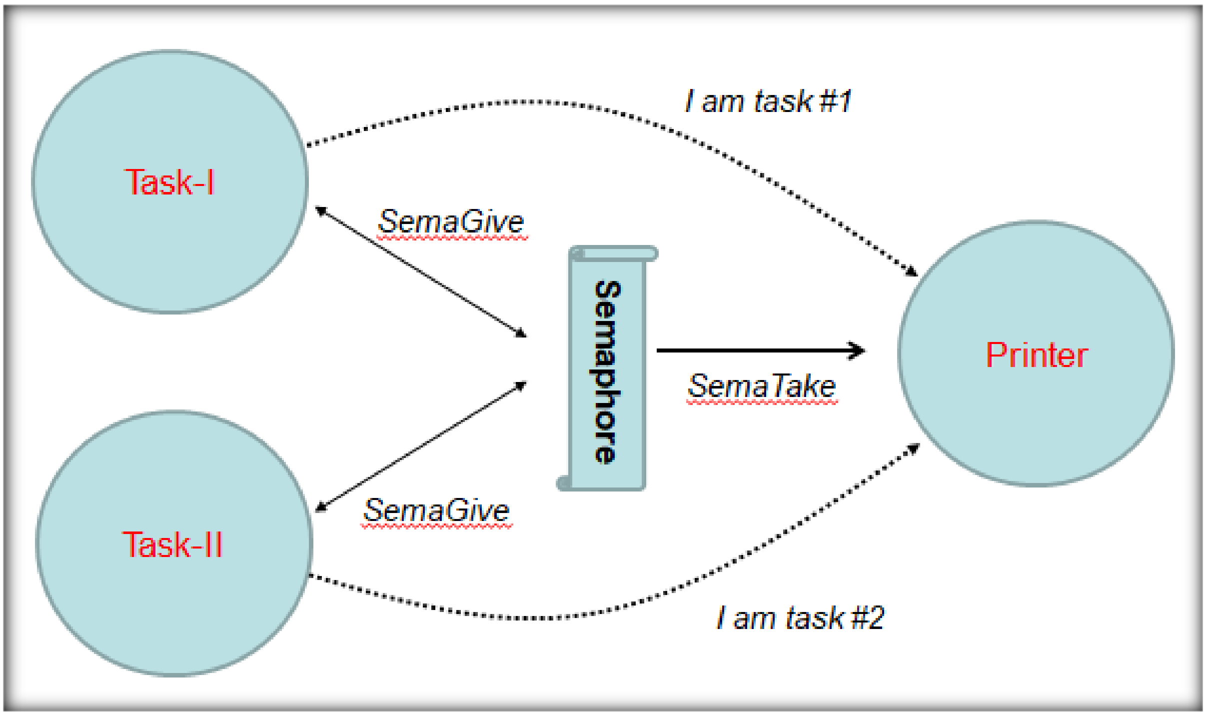
Microcontrollers are a central piece of the embedded systems that normally have a very specific job to do. The size constraints, and dedicated end application nature, rarely warrant the use of a full package implementation.

Applications - few to mention:

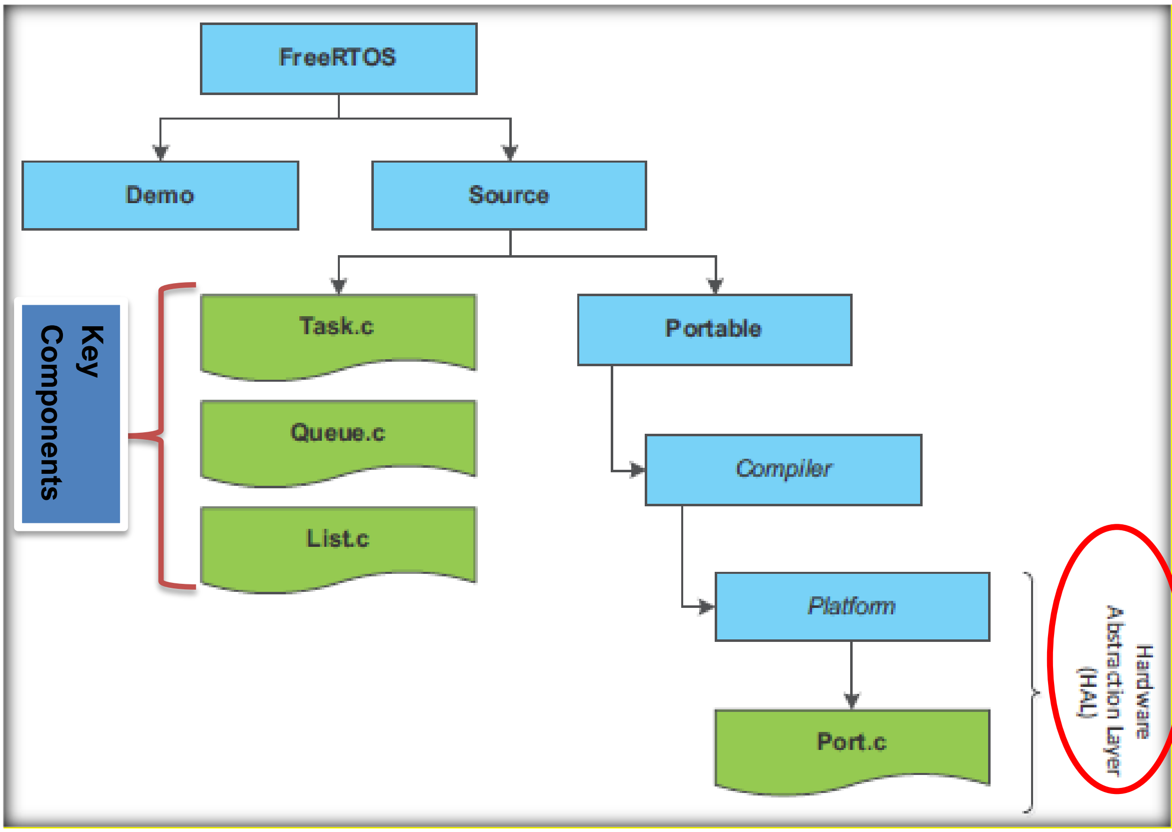
Command and control systems, heart pacemaker, industrial automation, and modern robotics systems

Key Features - Tasks Synchronization through Semaphores / Queues





FreeRTOS architecture



FreeRTOS configuration

FreeRTOSConfig.h

```
#ifdef __NVIC_PRIO_BITS
        /* __BVIC_PRIO_BITS will be specified when CMSIS is being used. */
        #define configPRIO_BITS                 __NVIC_PRIO_BITS
#else
        #define configPRIO_BITS                4         /* 15 priority levels */
#endif

/* The lowest interrupt priority that can be used in a call to a "set priority"
function. */
#define configLIBRARY_LOWEST_INTERRUPT_PRIORITY      0xf
```

Reference:

https://www.st.com/resource/en/user_manual/dm00105262-developing-applications-on-stm32cube-with-rtos-stmicroelectronics.pdf

FreeRTOS APIs

APIs Categories	API
Task Creation	– xTaskCreate – vTaskDelete
Task Control	– vTaskDelay – vTaskDelayUntil – uxTaskPriorityGet – vTaskPrioritySet – vTaskSuspend – vTaskResume – xTaskResumeFromISR – vTaskSetApplicationTag – xTaskCallApplicationTaskHook
Task Utilities	– xTaskGetCurrentTaskHandle – xTaskGetSchedulerState – uxTaskGetNumberOfTasks – vTaskList – vTaskStartTrace – ulTaskEndTrace – vTaskGetRunTimeStats
Kernel Control	– vTaskStartScheduler – vTaskEndScheduler – vTaskSuspendAll – xTaskResumeAll
Queue Management	– xQueueCreate – xQueueSend – xQueueReceive – xQueuePeek – xQueueSendFromISR – xQueueSendToBackFromISR – xQueueSendToFrontFromISR – xQueueReceiveFromISR – vQueueAddToRegistry – vQueueUnregisterQueue
Semaphores	– vSemaphoreCreateBinary – vSemaphoreCreateCounting – xSemaphoreCreateMutex – xSemaphoreTake – xSemaphoreGive – xSemaphoreGiveFromISR

Task-0

This task demonstrates:

- Simple working of a **freeRTOS** on **STM32L4S5** device

Objective

- Learn how to set-up Real-Time OS
- Create applications to start the **freeRTOS**
- Generate code in STM32Cube Tools using **CMSIS** functions

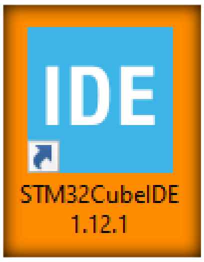
On the target board,

You will use GPIOs (**LEDs**) and/or USART (**Tera-Term**) to demonstrate the working of RTOS.

Procedure

Launch “**STM32CubeIDE**” Development Tool

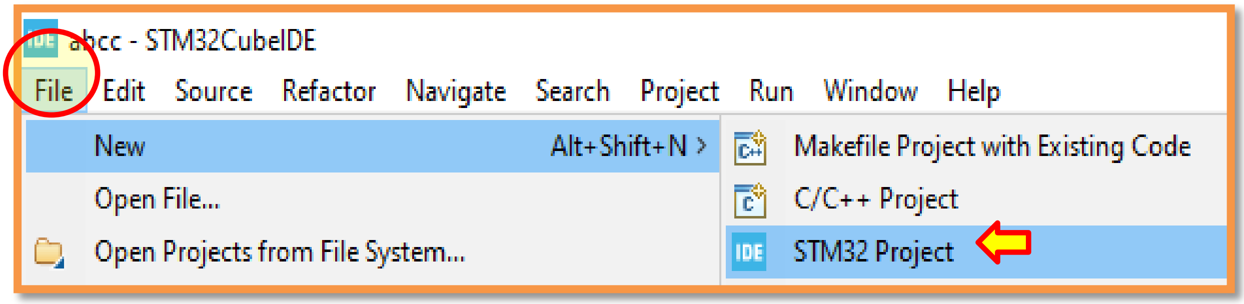
Double Click the Icon



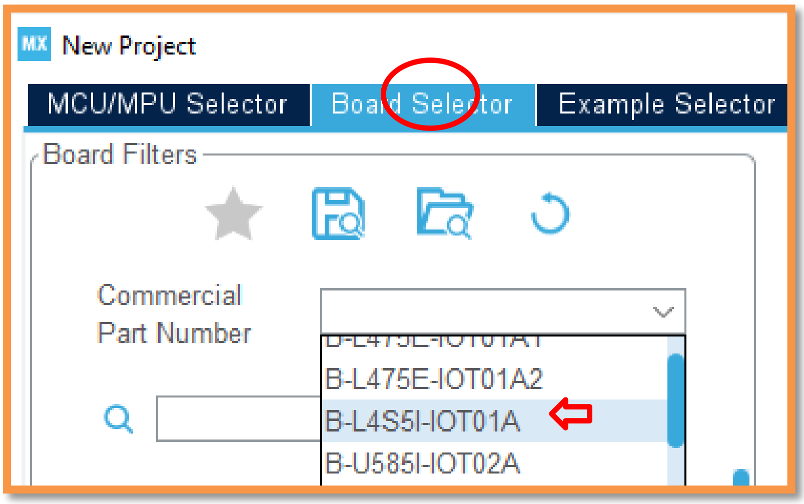
Open a new project

File → New Project

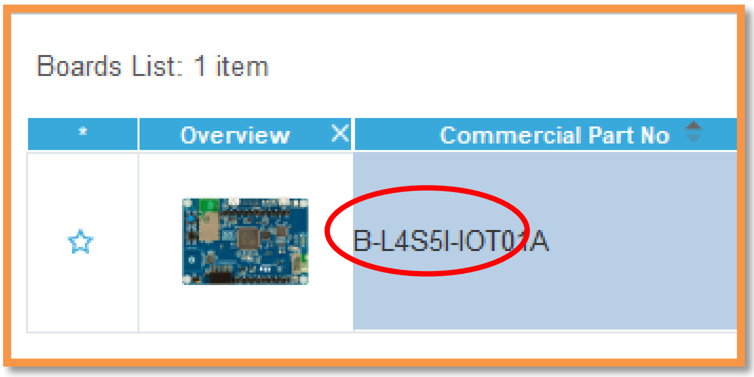
➔ STM32 Project



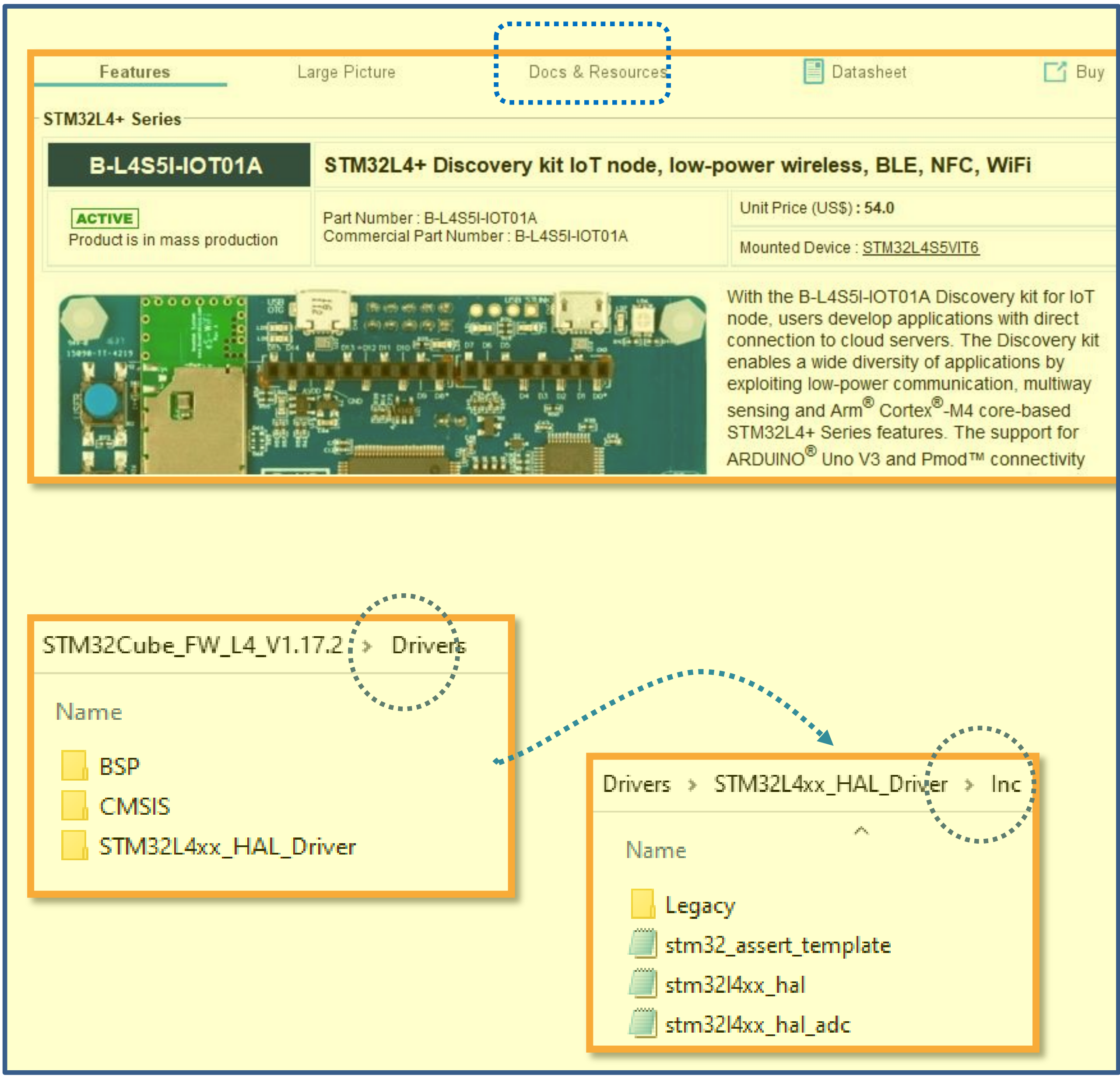
Select “**Hardware Platform**”



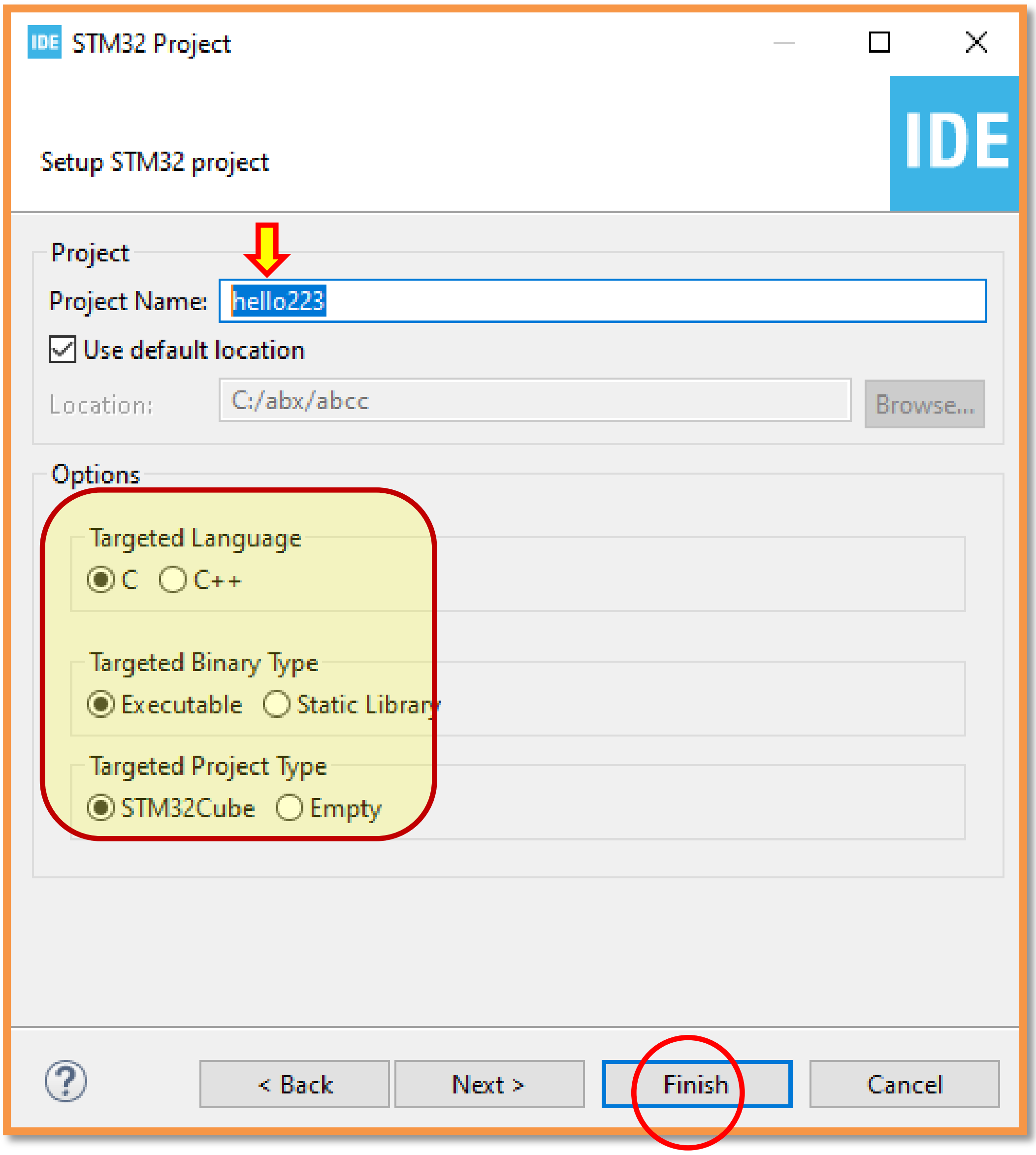
Select the specific “h/w board”, (if there are multiple options)

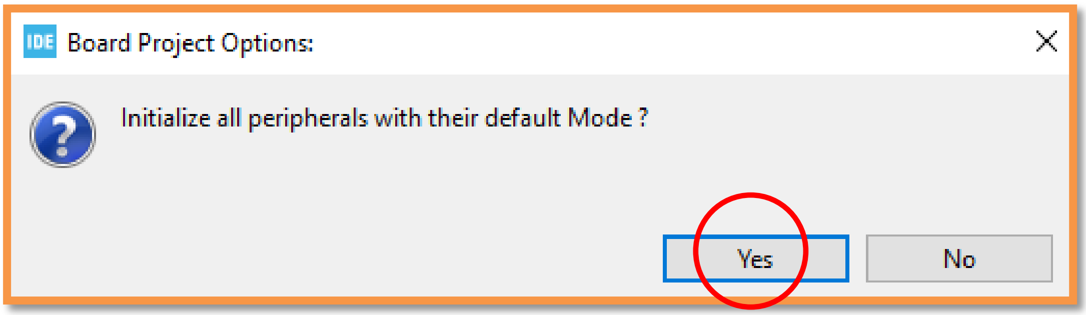


<Next>



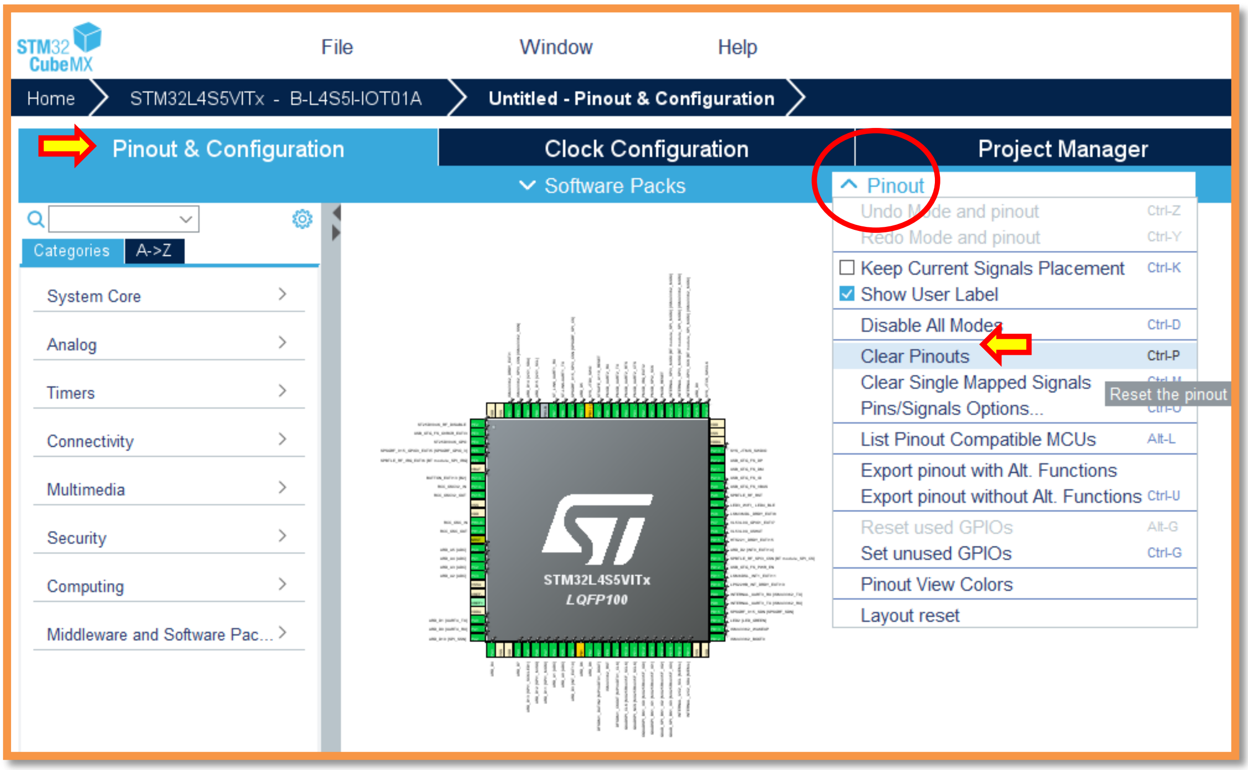
Type in "Project name"

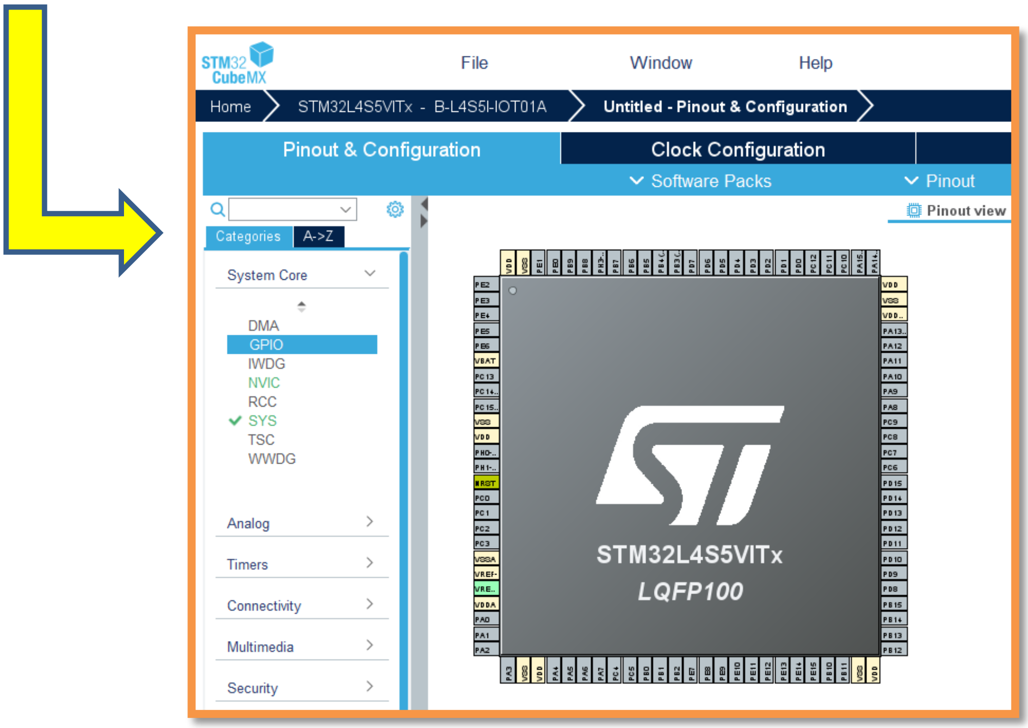




Reset “default Pinout”

- **Select** “Pinout & Configuration”,
- **Click** “Pinout”
- **Right Click**, “Clear Pinouts”





<u>H/W Configuration</u>

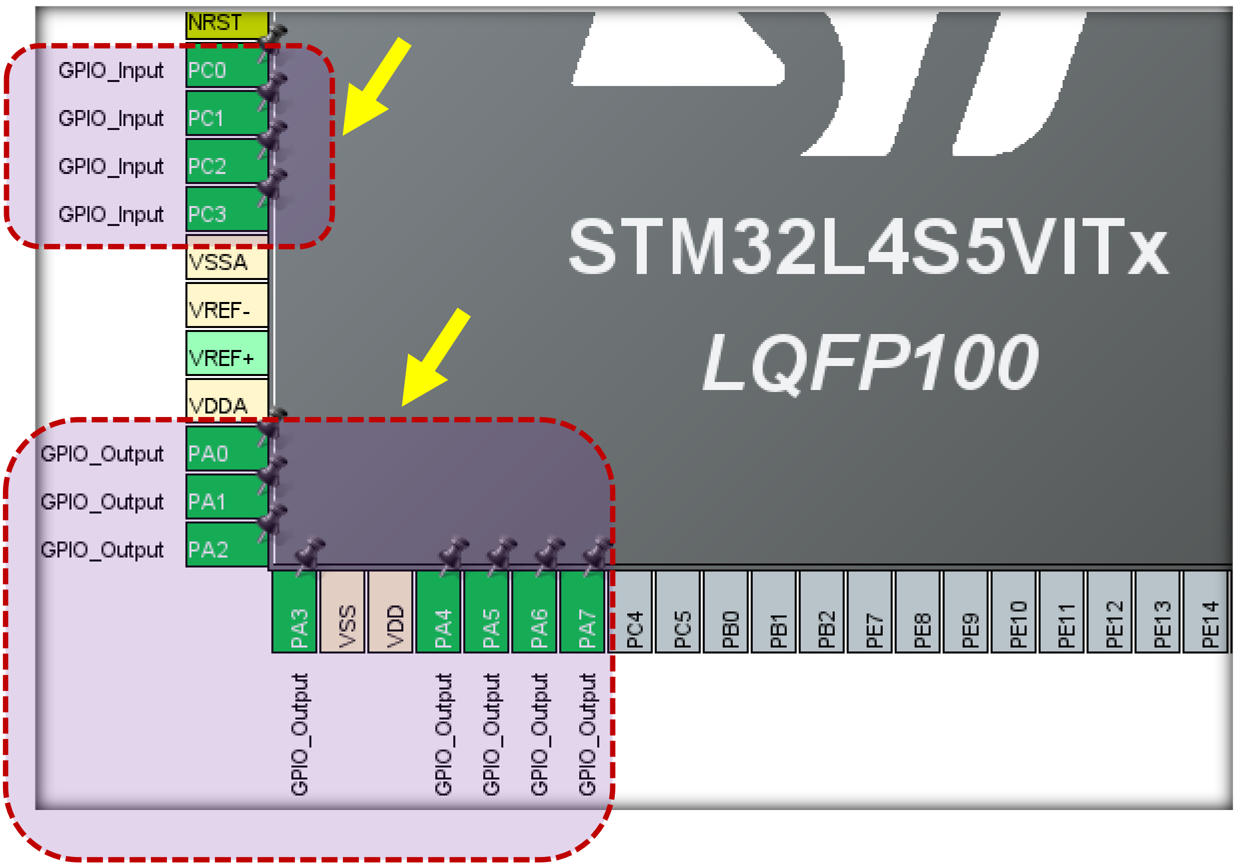
To demonstrate working of the given exercise,

Configure GPIO ports

Use external Base-board to access Switches and LEDs – Try, C&C custom board

> **Right Click** on **PA0-PA7, PB14**
> **Select**, **GPIO_Output**

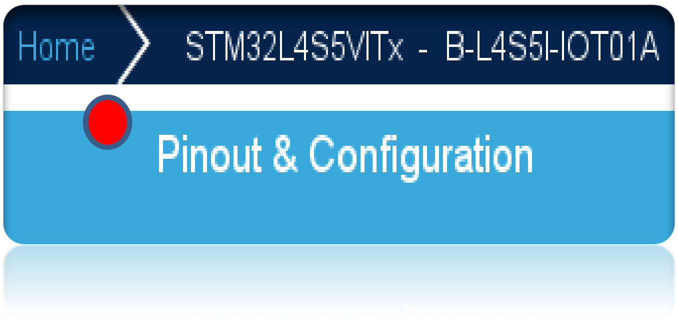
> **Right Click** on **PC0- PC3, PC13**
> **Select**, **GPIO_Input // With Pull-Up**



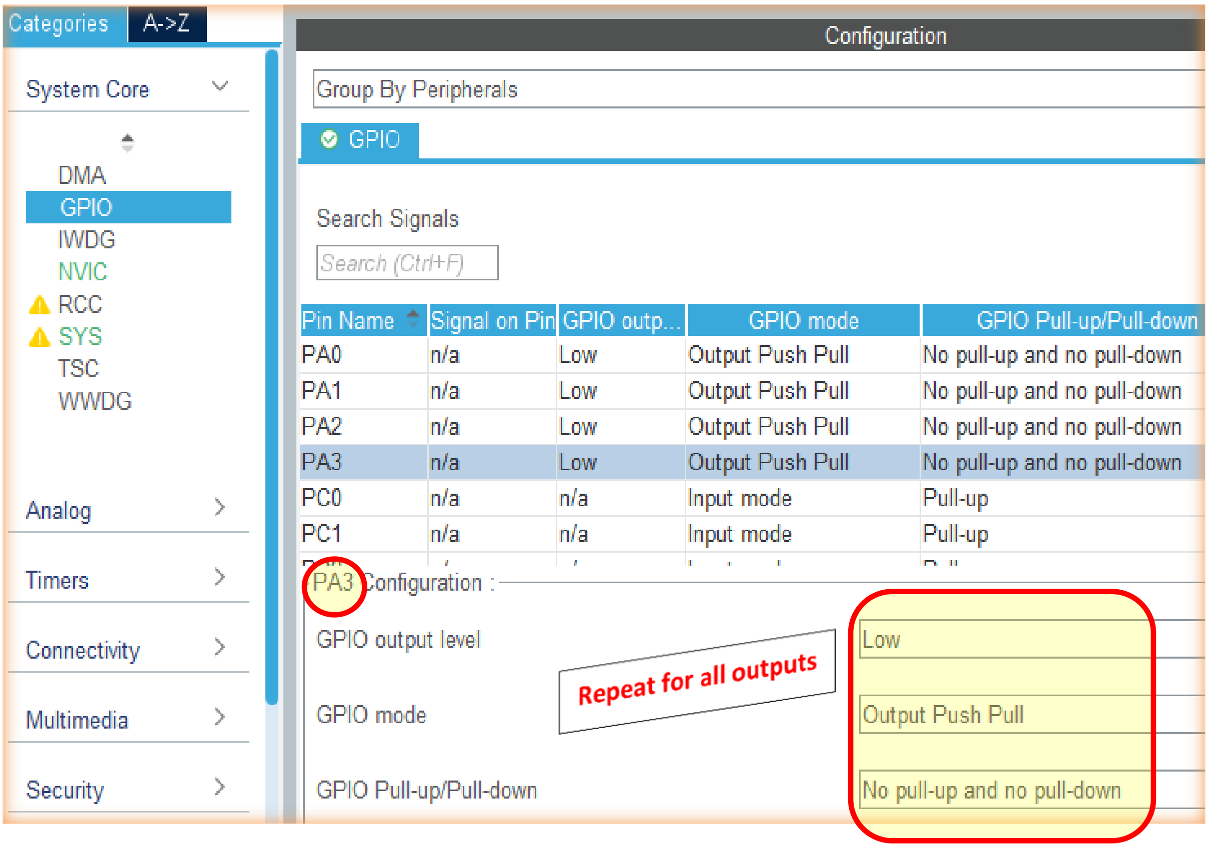
➔ **This enables:**

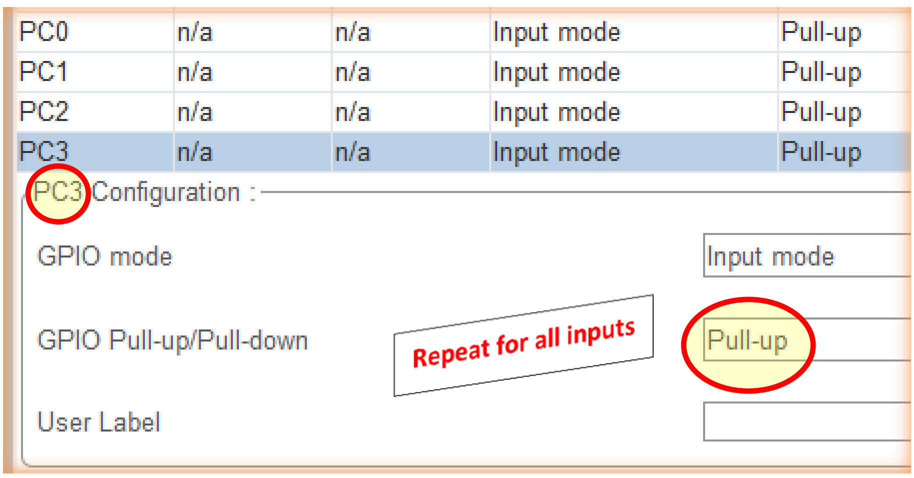
- Clock for port(s), and
- Sets the direction of the port as an input or output

Click "Pinout & Configuration"

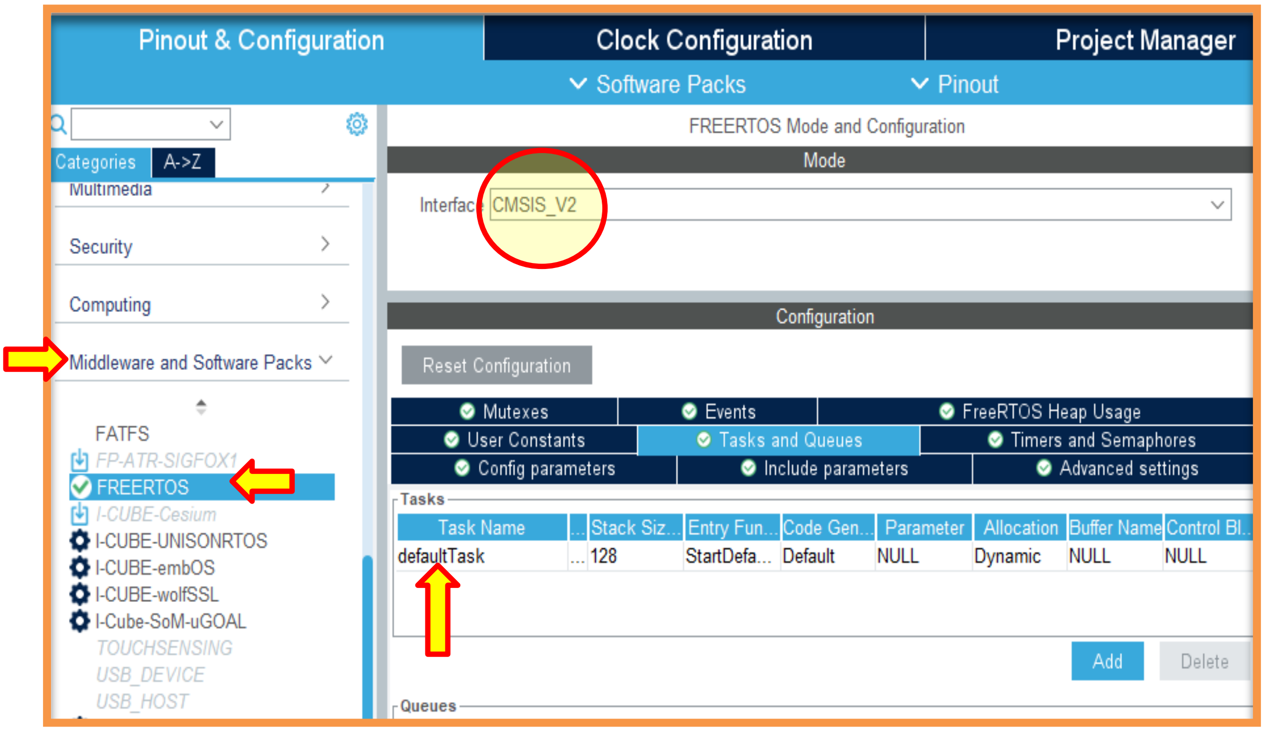


Configure GPIO (**mode**)

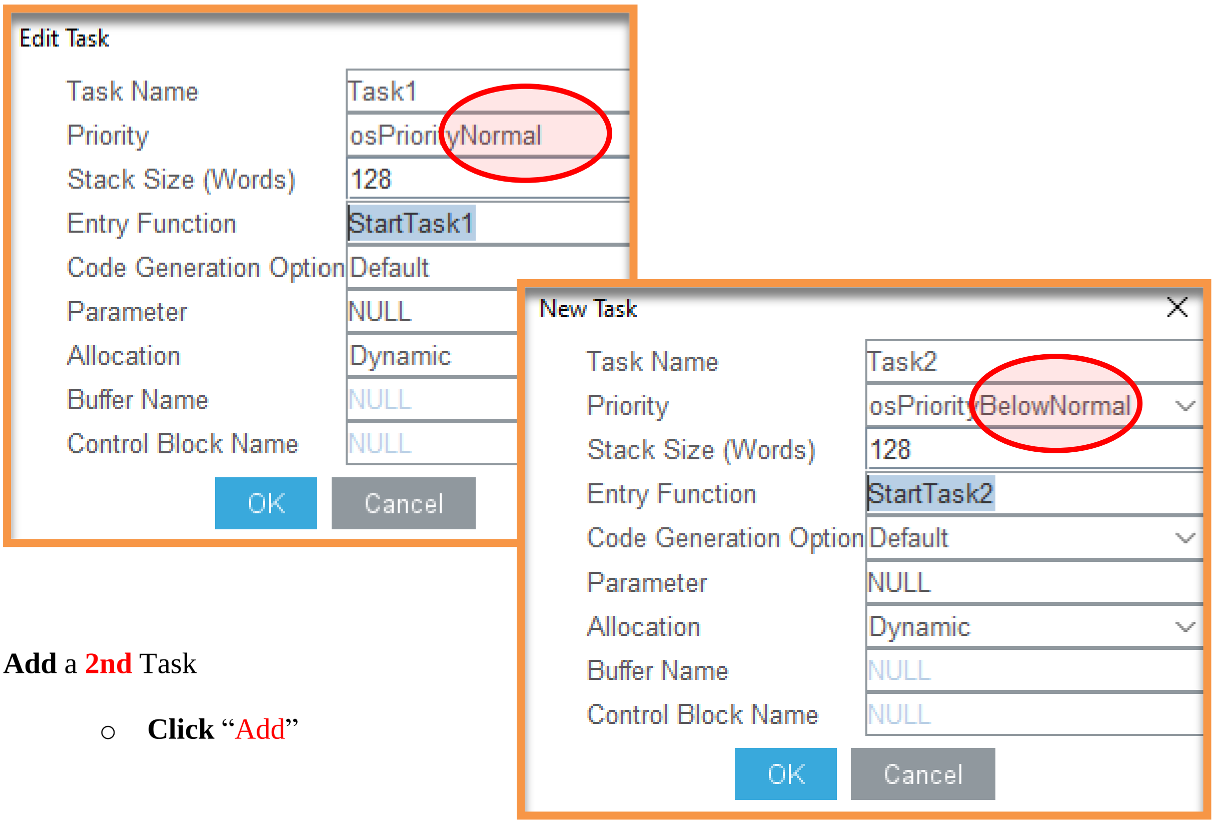




Configure FreeRTOS



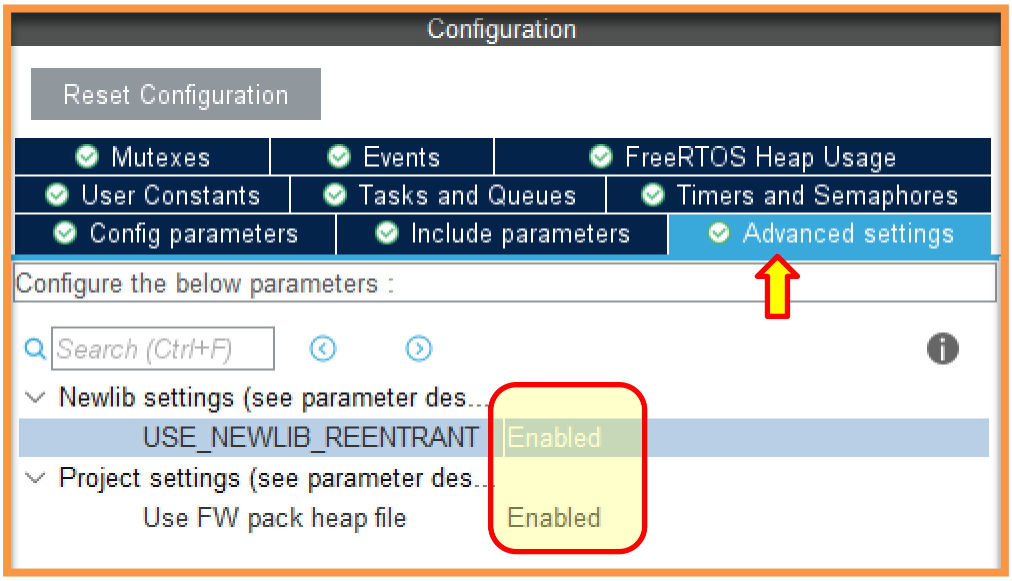
Configure / Edit default Tasks



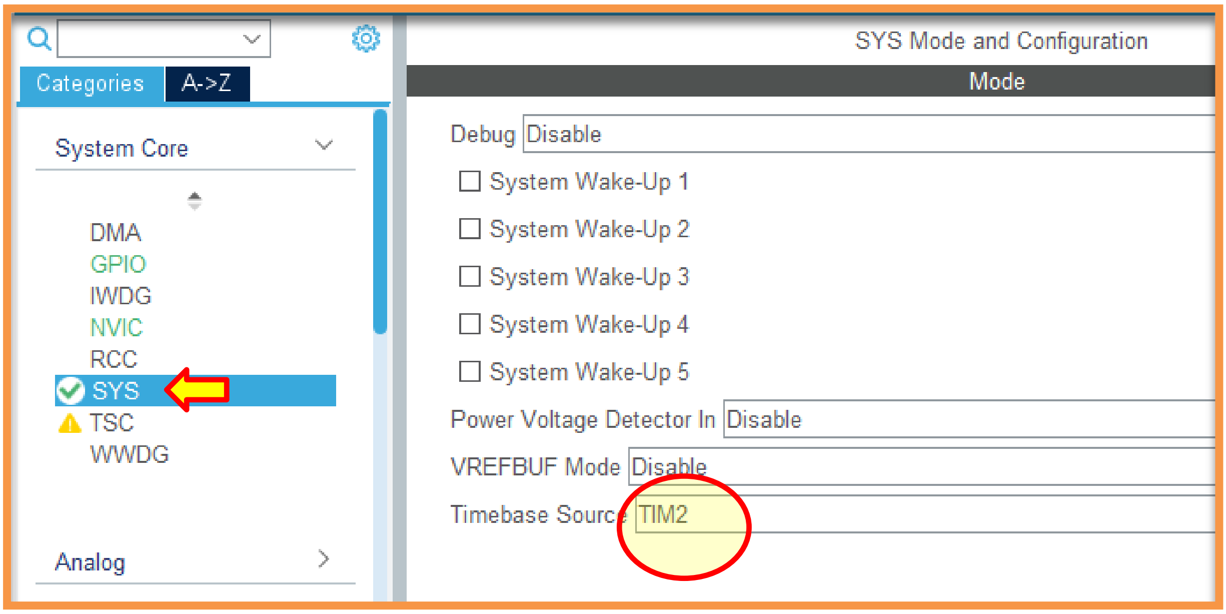
Add a **2nd** Task

- **Click** "Add"

Optional – Enable library (new version)



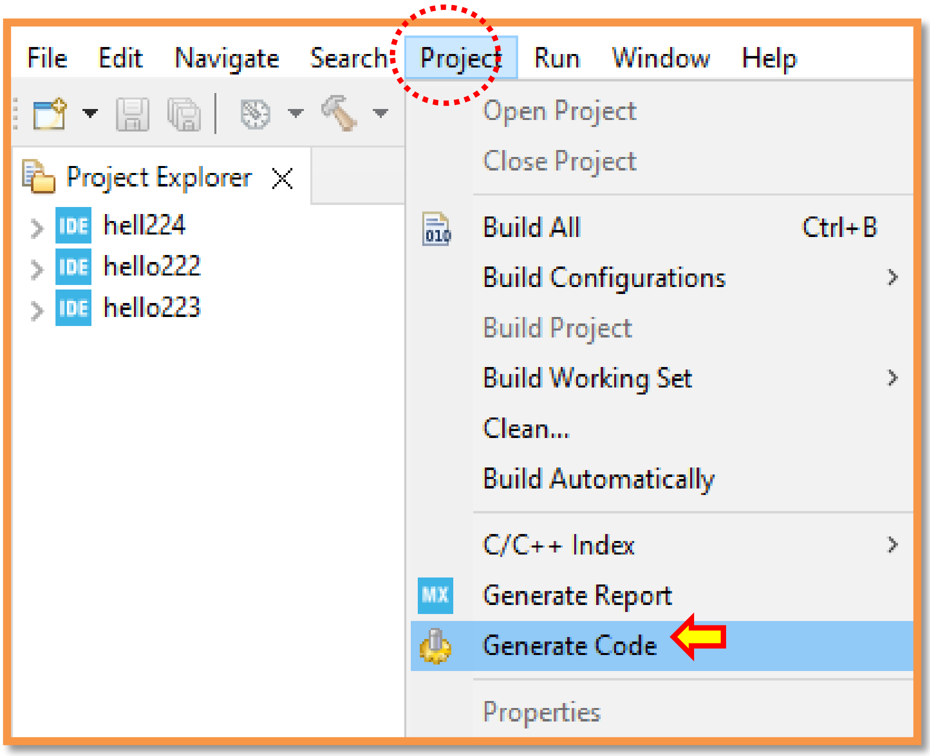
Choose Timer as the HAL Timebase Source (Instead of Systick)

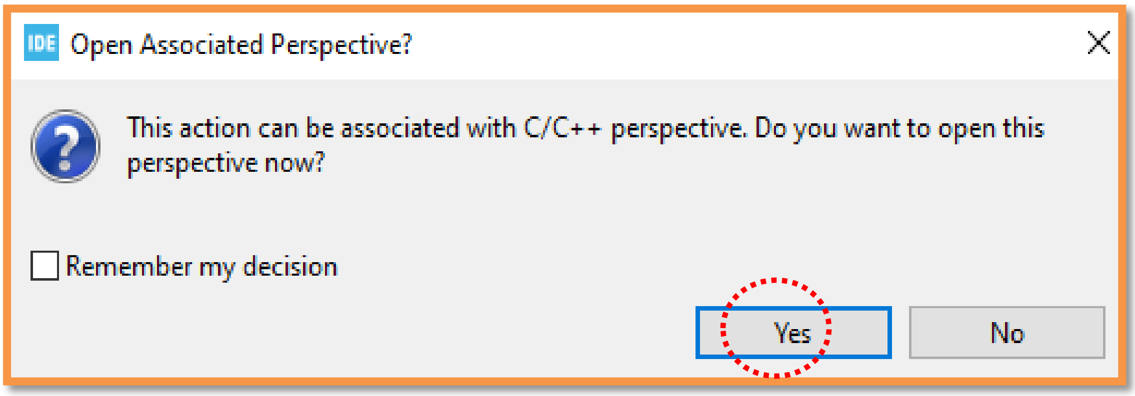


NOTE:

The **SysTick** is a special timer in most ARM processors that's generally reserved for operating system purposes. By default, SysTick will be used for things like HAL_Delay() and HAL_GetTick(). As a result, the STM32 HAL framework gives SysTick a very high priority. However, **FreeRTOS** needs SysTick for its scheduler, and it requires SysTick to be a much lower priority. **Therefore**, a quick work around is to use a **Timer** as a Time-base source in the cases of freeRTOS.

Generate Code

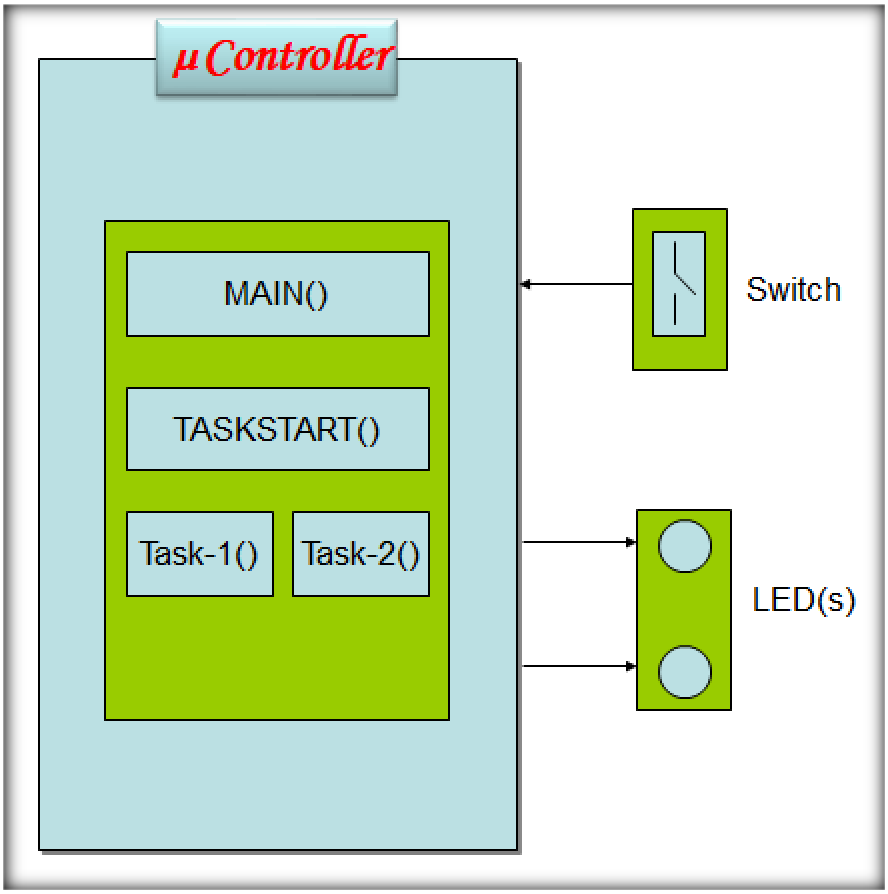




Task-1

This task demonstrates how to:

- Configure **GPIO** ports
- Create multiple Tasks in RTOS (**FreeRTOS**)
- Toggles a set of LEDs (**PA5 & PB14**) of PORTA & PORTB through **Tasks- 1 & 2**



- **Sample** Code for this task is given next

Sample Code to update "main.c"

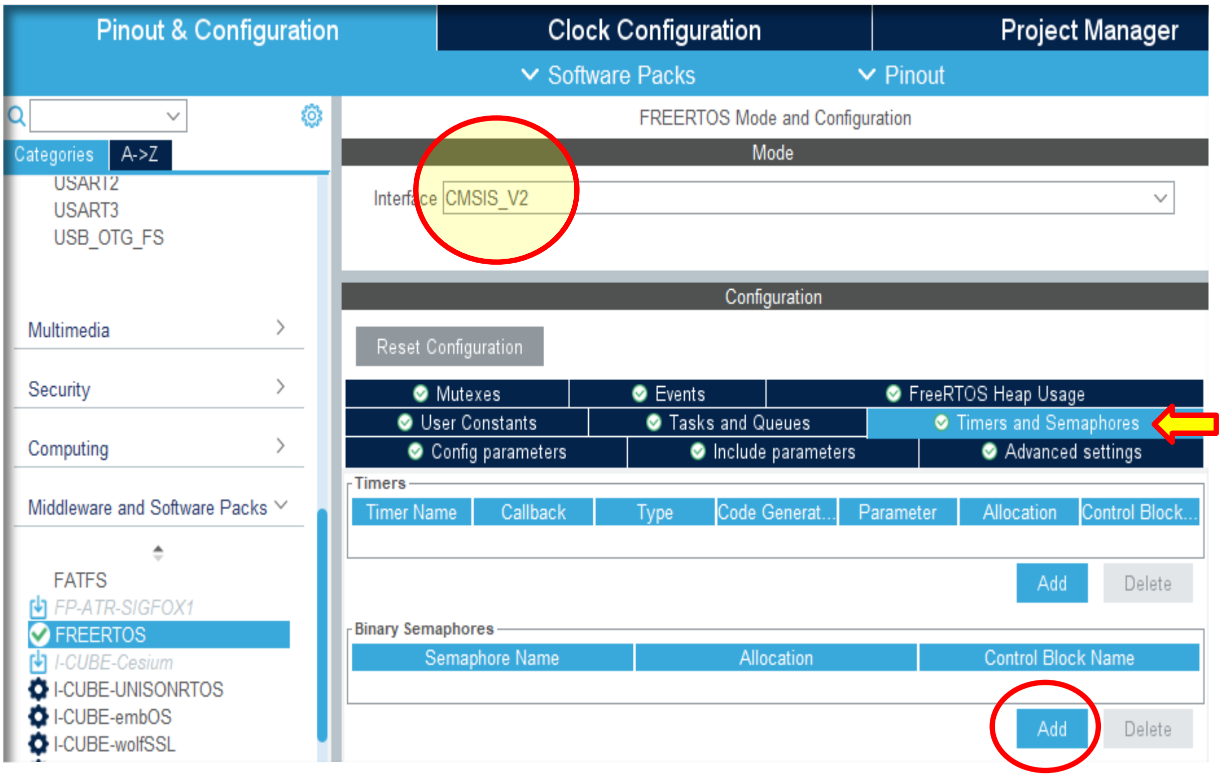
```
void StartTask1(void *argument)
{
 /* USER CODE BEGIN 5 */
 /* Infinite loop */
 for(;;)
 {
        GPIOB->ODR ^= (0x1 << 14); //PB14 ON
        osDelay(100); // vTaskDelay(100);
 }
 /* USER CODE END 5 */
}
```

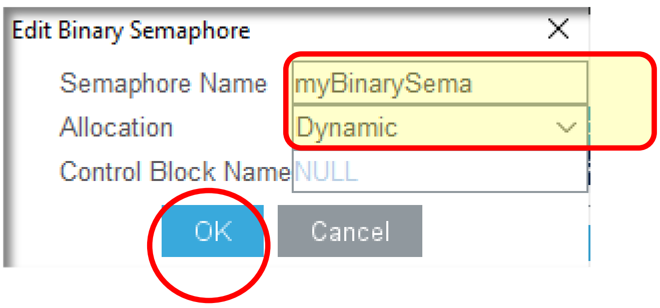
```
void StartTask2(void *argument)
{
 /* USER CODE BEGIN StartTask2 */
 /* Infinite loop */
 for(;;)
 {
         GPIOA->ODR ^= (0x1 << 5); //PA5 ON
         osDelay(50);
 }
 /* USER CODE END StartTask2 */
}
```

- **Build** "Project"
- **Flash** "binary code" on the h/w
- **Reset h/w board (**By pressing switch/button on the board**)**
- **Monitor** "LEDS" toggling on the h/w board

Task-2

Add "Semaphore" feature (Continuation to previous task)





Click "Generate Code"

STM32CubeIDE

Sample Code to update "main.c"

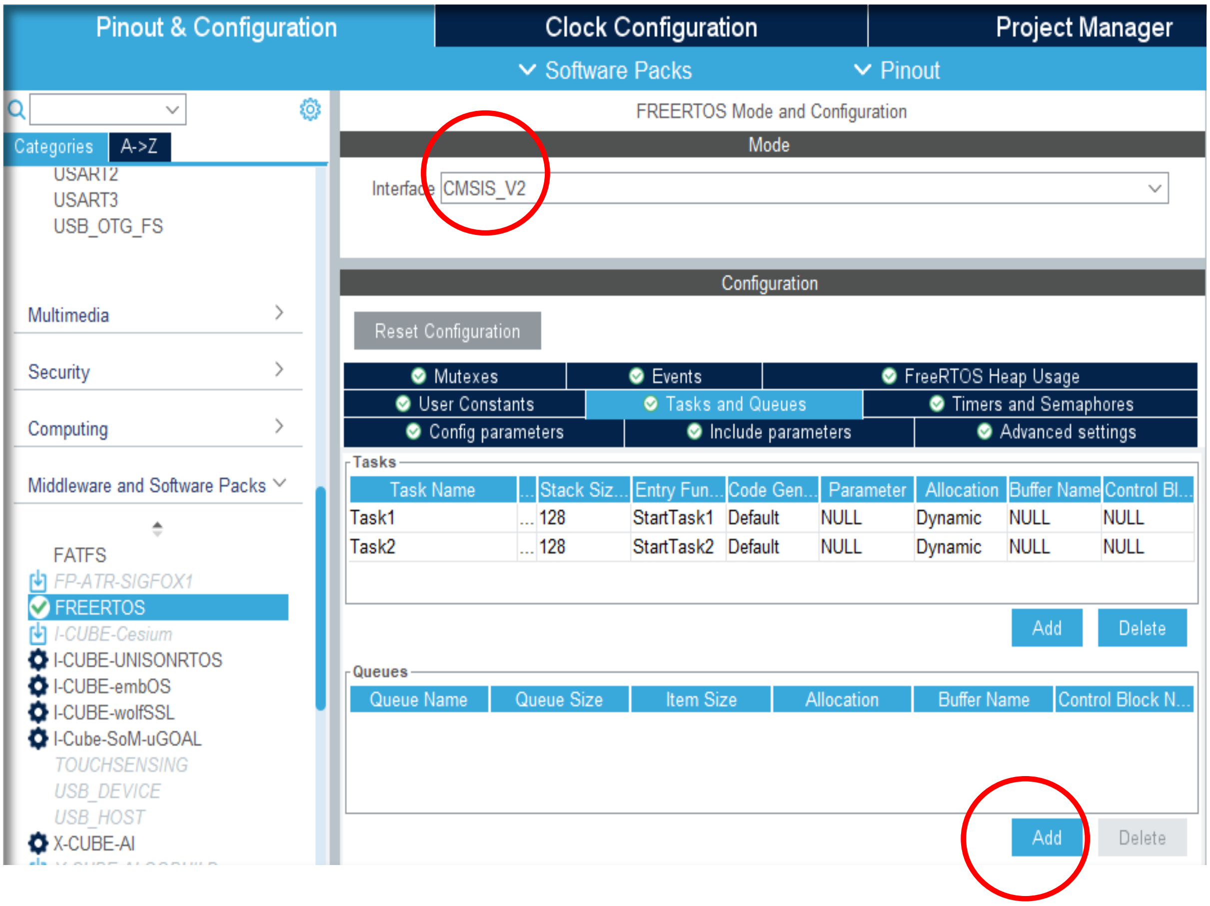
```
void StartTask1(void *argument)
{
 /* USER CODE BEGIN 5 */
 /* Infinite loop */
 for(;;)
 {
	osSemaphoreRelease(myBinarySemaHandle);

	GPIOB->ODR ^= (0x1 << 14); //PB14 ON
	osDelay(100);

 }
 /* USER CODE END 5 */
}
```

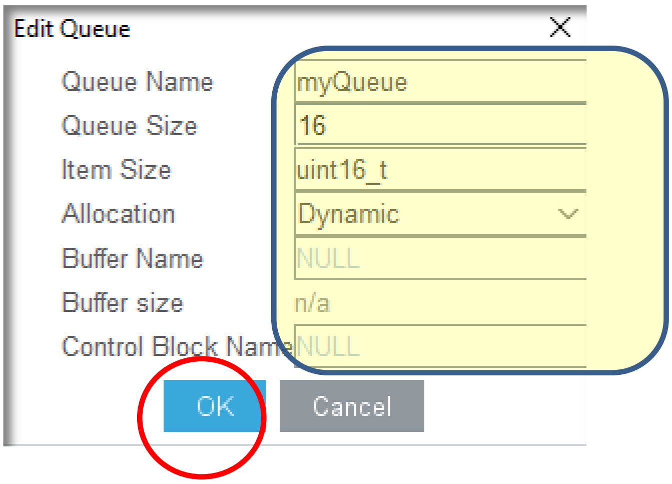
```
void StartTask2(void *argument)
{
 /* USER CODE BEGIN StartTask2 */
 /* Infinite loop */
 for(;;)
 {
	osSemaphoreAcquire(myBinarySemaHandle, osWaitForever);

	GPIOA->ODR ^= (0x1 << 5); //PA5 ON

 }
 /* USER CODE END StartTask2 */
}
```

- **Build** "Project"
- **Flash** "binary code" on the h/w
- **Reset h/w board (**By pressing switch/button on the board**)**
- **Monitor** "LEDS" toggling on the h/w board

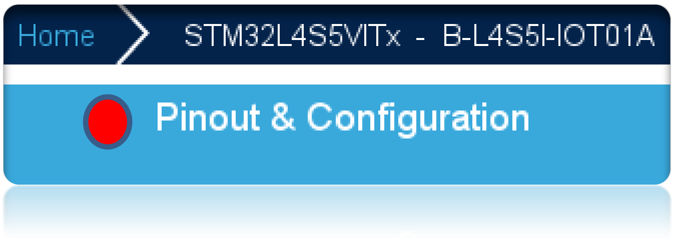
Task-3

Add “Queue” feature (Continuation to previous task)

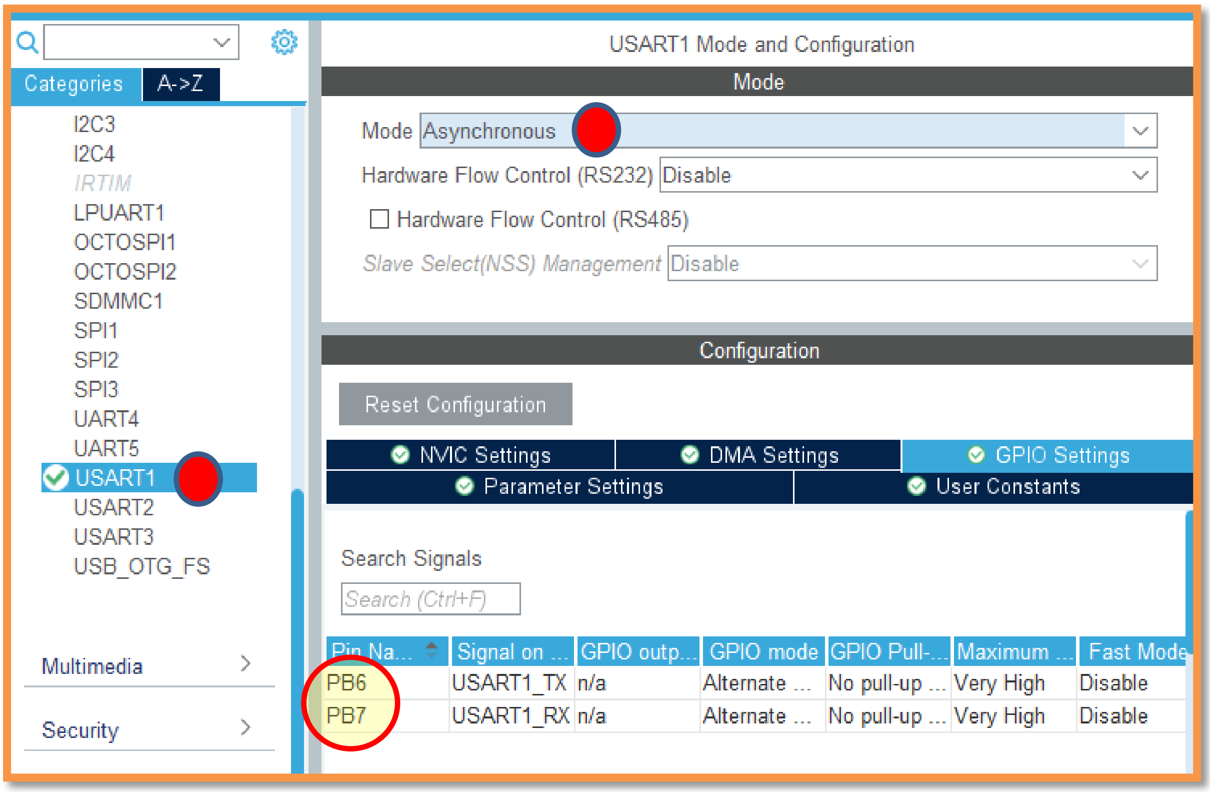


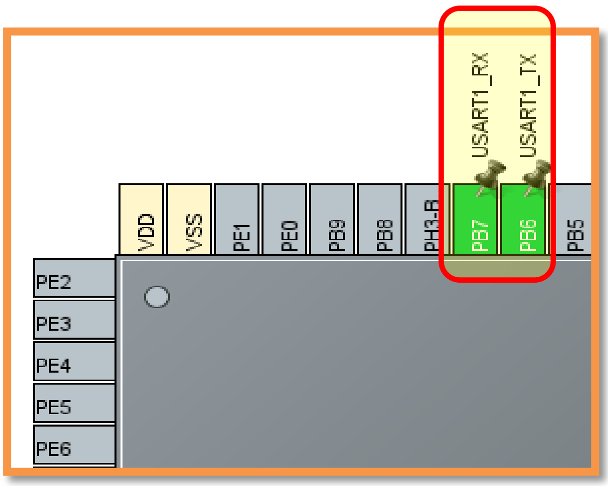


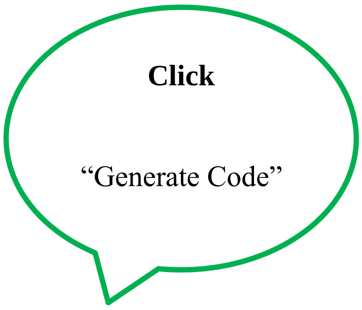
Add “UART1” feature



Configure USART Pins







Sample Code to update "main.c"

```
/* USER CODE BEGIN 0 */

#include <stdio.h>

typedef struct {      // object data type
            uint8_t Idx;
            uint8_t Buf[5];
} MSGQUEUE_OBJ_t;

/* USER CODE END 0 */
```

```
// Update queue size in "main.c"

/* creation of myQueue */

 myQueueHandle = osMessageQueueNew (16, sizeof( MSGQUEUE_OBJ_t ), &myQueue_attributes);
```

```
void StartTask1(void *argument)
{

      MSGQUEUE_OBJ_t msg [16] = {

                        {'A', {1,2,3,4,5} },
                        {'B', {6,7,8,9,10} },
                        {'C', {11,12,13,14,15} }
      }; // = {0};

      uint8_t i=0;
      osStatus_t status;

  while (1) {
            GPIOA->ODR ^= (0x1 << 5); //PA5 ON
            vTaskDelay( 100);

            do {
                status = osMessageQueuePut(myQueueHandle, &msg[i], 0U, 0U);

            } while ( status != osOK );

            i++;
            i= i & 0xf; // restricted to 16 messages

            osThreadYield();    // Suspend thread for a system tick
  }

}
```

```
void StartTask2(void *argument)
{

	char str_tmp[100] = "";		// To display formatted messages

	MSGQUEUE_OBJ_t msg;
	osStatus_t status;

	while (1) {
		status = osMessageQueueGet(myQueueHandle, &msg, NULL, 0U);   // wait for message

		if (status == osOK) {
			GPIOA->ODR ^= (0x1 << 5); //PA5 ON
			osDelay(50);

			snprintf(str_tmp,100," %c %d \n\r", msg.Idx, msg.Buf[2]);
			HAL_UART_Transmit(&huart1,( uint8_t * )str_tmp,sizeof(str_tmp),1000);
		}

	}

}
```

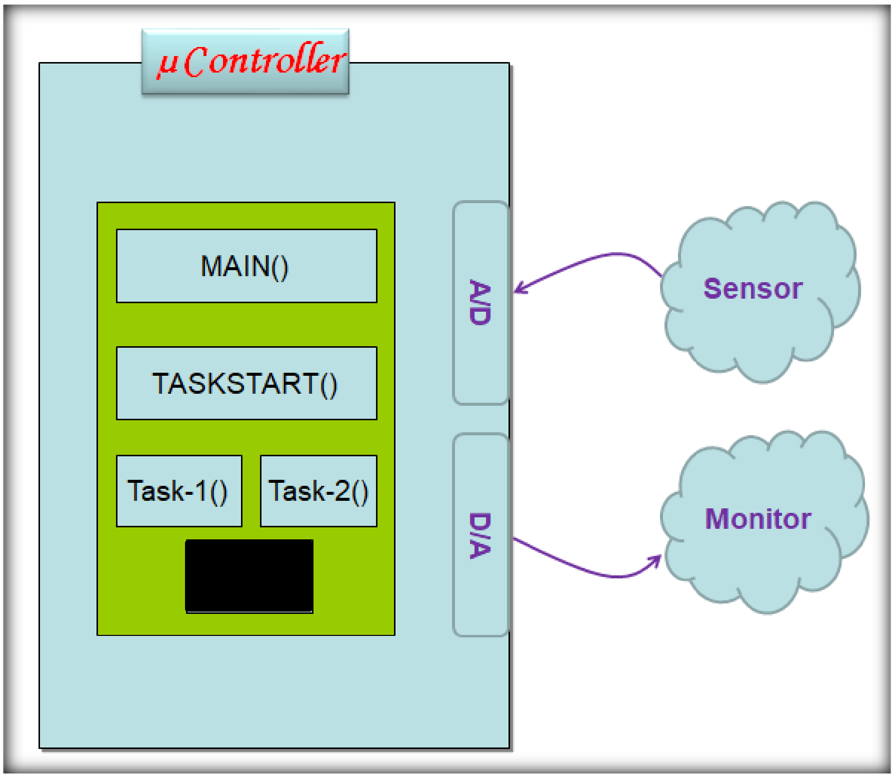
Realize Code

- **Build** "Project"
- **Flash** "binary code" on the h/w
- **Reset h/w board (**By pressing switch/button on the board**)**
- **Monitor** h/w board and "Tera-Term" Console window for the messages

Exercise

This task is to demonstrate how to:

- Configure GPIO ports
- Create multiple Tasks in RTOS (Free RTOS)
- Generate A/D data (Task 1)
- Generate D/A data (Task 2)



Construct a C program to demonstrate and verify the design behaviour.

Reference(s):

https://www.keil.com/pack/doc/CMSIS/RTOS2/html/group__CMSIS__RTOS__SemaphoreMgmt.html

Review Questions

Q1.

Q2

Q3
