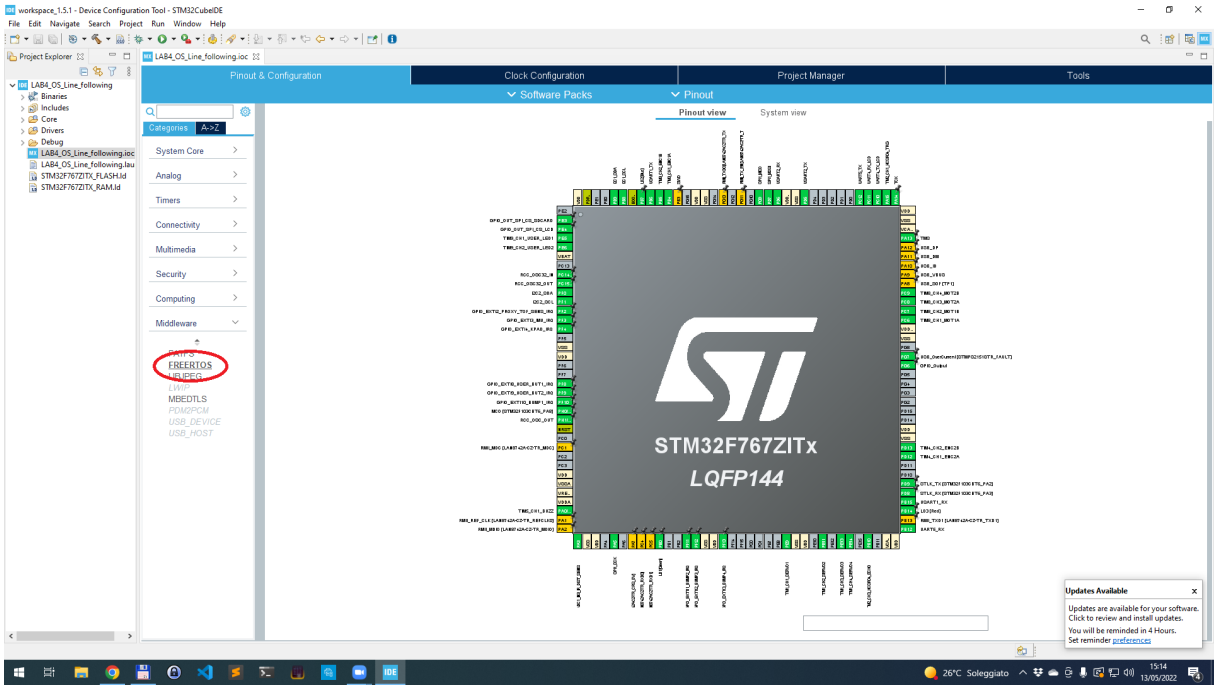
ERTC - Laboratory 5

1 Introduction

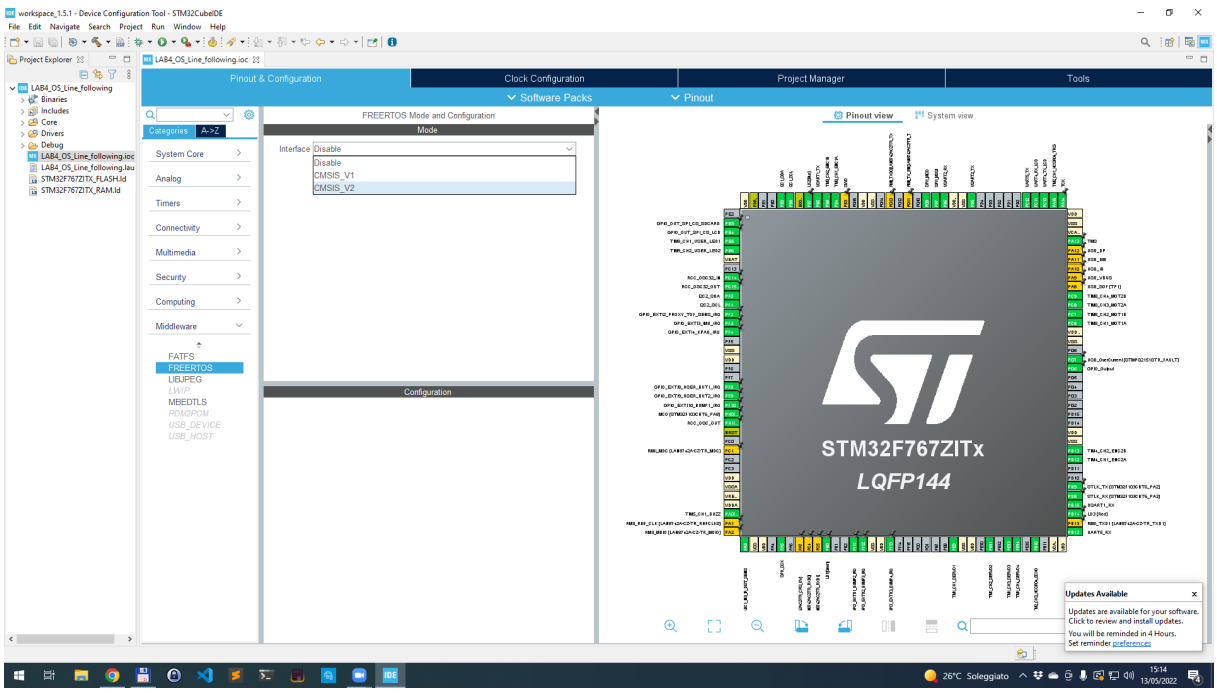
The goal of this document is to provide an insight into the use of FreeRTOS. We will modify the code from Lab 3 to use FreeRTOS.

2 Enable FreeRTOS

To enable FreeRTOS, open the `*.ioc` file, navigate to the *Middleware* tab and than click *FreeRTOS*.

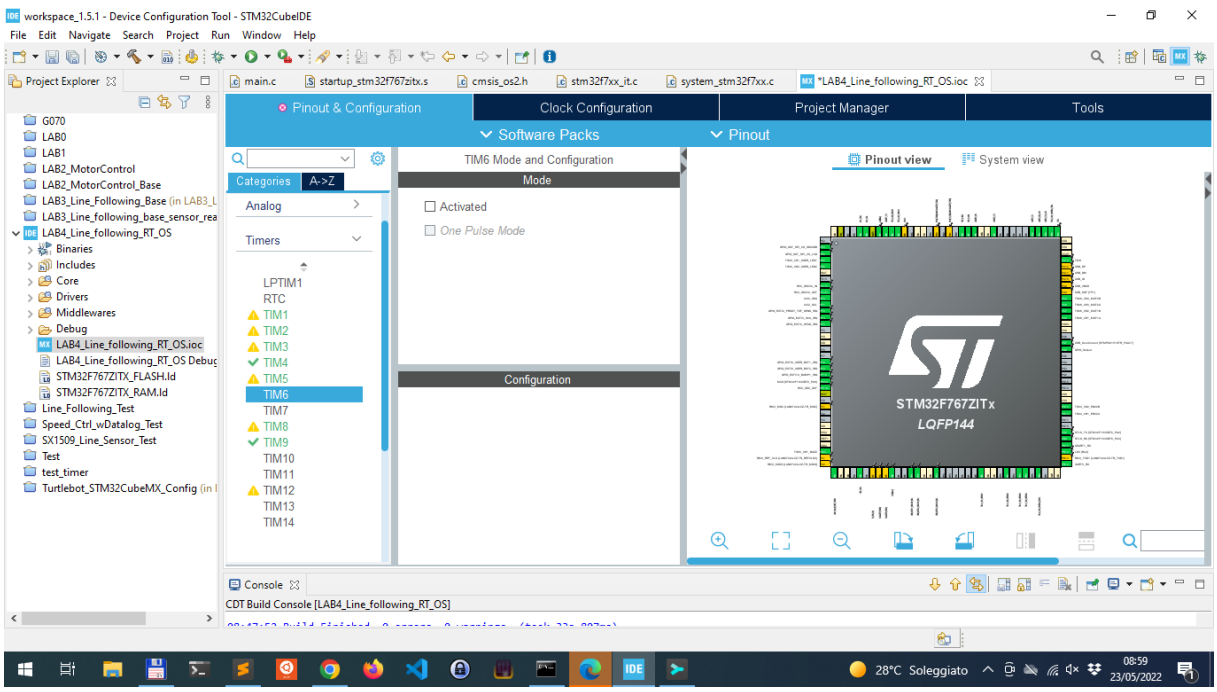


From the *interface* menu select *CMSIS_V2*. This will enable the FreeRTOS support.

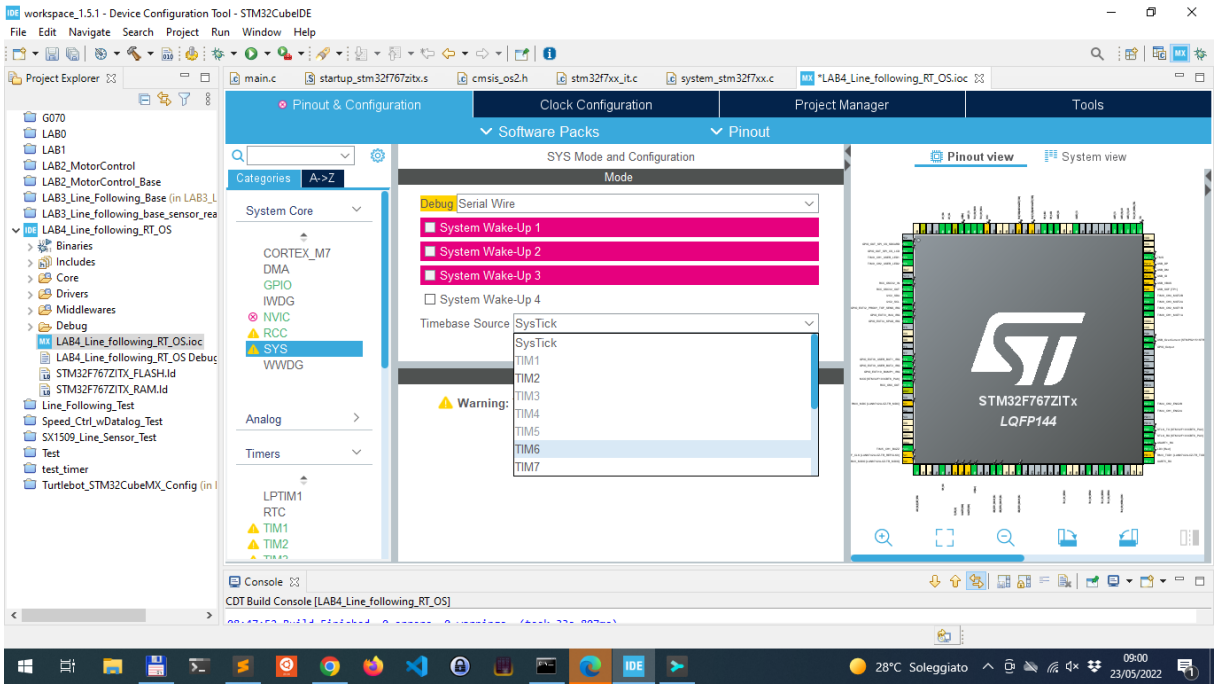


Leave all the other options as they are.

Go to *Timers->TIM6* and remove the tick from *Activated*



Go to *System Core-> SYS* and in *Timebase source* select *TIM6*



3 Create a task

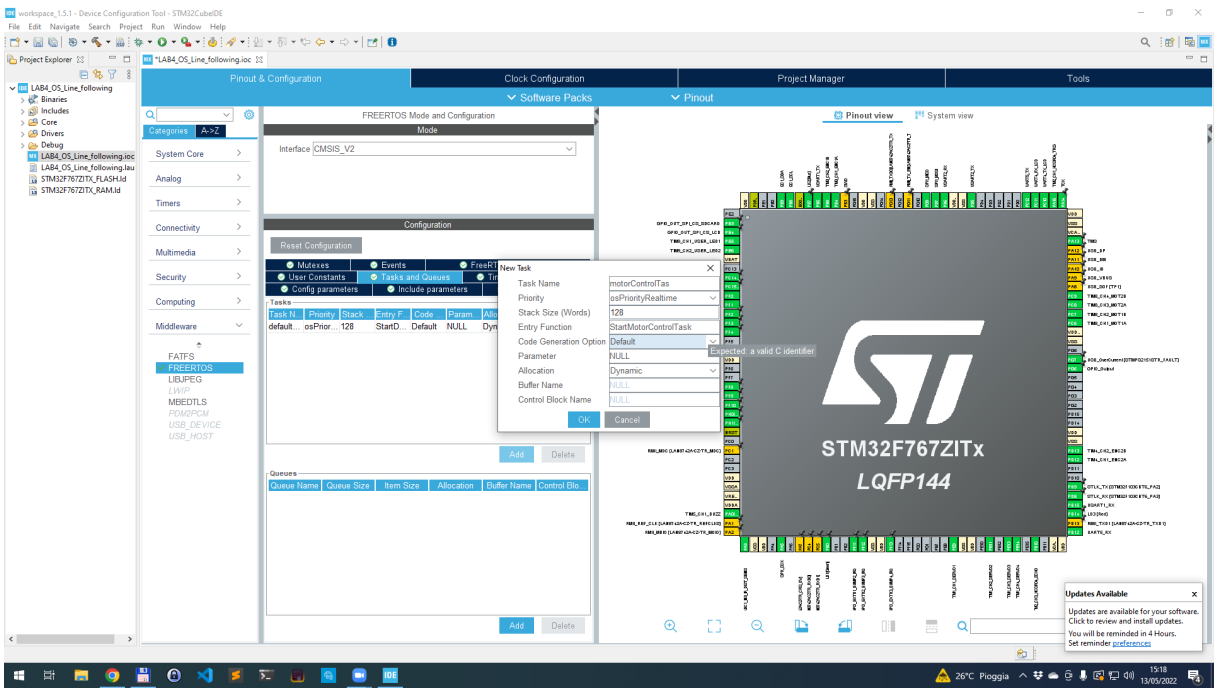
Click on the *Tasks and Queues* tab.

Than under the *Tasks* panel click on the *Add* button.

Here there are three important option to modify:

- The *Task Name* field, which is the name of the task.
- The *Task Priority* field, which is the priority of the task.
- The *Entry Function* field, which is the name of the function that will execute our task.

For now, leave the other fields as they are.



Create the following three tasks:

Task Name	*Task Priority*	*Entry Function*	*Description*
controlTask	osPriorityRealtime	StartControlTask	This task is responsible for controlling the motors.
lineTask	osPriorityRealtime	StartLineTask	This task is responsible for reading the line sensor.
commTask	osPriorityLow	StartCommTask	This task is responsible for communicating with the PC.

Save the file and generate the code.

After the code generation, you will have somewhere in the `main.c` those three functions:

```
1  /* USER CODE BEGIN Header_StartControlTask */
2  /**
3  * @brief Function implementing the controlTask thread.
4  * @param argument: Not used
5  * @retval None
```

```
 6  */
 7  /* USER CODE END Header_StartControlTask */
 8  void StartControlTask(void *argument)
 9  {
10    /* USER CODE BEGIN StartControlTask */
11    /* Infinite loop */
12    for(;;)
13    {
14      osDelay(1);
15    }
16    /* USER CODE END StartControlTask */
17  }
18
19  /* USER CODE BEGIN Header_StartLineTask */
20  /**
21  * @brief Function implementing the lineTask thread.
22  * @param argument: Not used
23  * @retval None
24  */
25  /* USER CODE END Header_StartLineTask */
26  void StartLineTask(void *argument)
27  {
28    /* USER CODE BEGIN StartLineTask */
29    /* Infinite loop */
30    for(;;)
31    {
32      osDelay(1);
33    }
34    /* USER CODE END StartLineTask */
35  }
36
37  /* USER CODE BEGIN Header_StartCommTask */
38  /**
39  * @brief Function implementing the commTask thread.
40  * @param argument: Not used
41  * @retval None
42  */
43  /* USER CODE END Header_StartCommTask */
44  void StartCommTask(void *argument)
45  {
46    /* USER CODE BEGIN StartCommTask */
47    /* Infinite loop */
48    for(;;)
49    {
50      osDelay(1);
51    }
52    /* USER CODE END StartCommTask */
53  }
```

Observation: In lecture 20 we said that is recommended to not use delays since they may lead to scheduling anomalies. However, in this case `osDelay`, which is different from `HAL_Delay`, is “special”.

Indeed, the `osDelay(1)` function is a FreeRTOS primitive that is used to suspend the execution of the task for a certain amount of time. This leave space for other tasks to run.

`osDelay` takes as parameter the duration in System Ticks (in our case one tick corresponds to 1 ms) in which the task must be suspended. Since it is duration, the periodicity of the task is given by the sum of the suspend duration and the time taken to execute the task. It is clear that this may lead to some problems to the control loop since the timing may not be accurate.

To avoid this problem you can use `osDelayUntil` which takes as parameter the **absolute** tick at which the task must be resumed. For example, using `osDelayUntil(10)`, the task will be resumed every 10 ticks independently of the time taken to execute the task.

4 Exercise 1

Change the `osDelay(1)` to `osDelayUntil(10)` for all three tasks. Take the code you wrote inside the TIM6 callback (ignore the `if (htim->Instance == TIM6)`) and split the code basing on the functionalities. In particular, everything regarding the motors should be in the `controlTask` function, everything regarding the line sensor should be in the `lineTask` function and everything regarding the communication with the PC should be in the `commTask` function.

Delete completely the `void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)` function.

If needed, move local variables to global ones.

You will end up with a code similar to the following:

```
1  /* USER CODE BEGIN Header_StartControlTask */
2  /**
3  * @brief Function implementing the controlTask thread.
4  * @param argument: Not used
5  * @retval None
6  */
7  /* USER CODE END Header_StartControlTask */
8  void StartControlTask(void *argument)
9  {
10   /* USER CODE BEGIN StartControlTask */
11   /* Infinite loop */
12   for(;;)
13   {
14
15     uint32_t TIM3_CurrentCount, TIM4_CurrentCount;
16     int32_t TIM3_DiffCount, TIM4_DiffCount;
17     static uint32_t TIM3_PreviousCount = 0, TIM4_PreviousCount = 0;
18
```

```
      .
      .
      .

      /*  read current values of encoders counters  */
      TIM3_CurrentCount = __HAL_TIM_GET_COUNTER(&htim3);
      TIM4_CurrentCount = __HAL_TIM_GET_COUNTER(&htim4);

      .
      .
      .

      /*  Prepare data packet */
      loggerdata.w1 = w1_meas_rpm;
      loggerdata.w2 = w2_meas_rpm;
      loggerdata.u1 = u1_sat_V;
      loggerdata.u2 = u2_sat_V;



      /*  update previous values of encoder counts  */
      TIM3_PreviousCount = TIM3_CurrentCount;
      TIM4_PreviousCount = TIM4_CurrentCount;


      osDelayUntil(10);
    }
    /* USER CODE END StartControlTask */
}

/* USER CODE BEGIN Header_StartLineTask */
/**
* @brief Function implementing the lineTask thread.
* @param argument: Not used
* @retval None
*/
/* USER CODE END Header_StartLineTask */
void StartLineTask(void *argument)
{
  /* USER CODE BEGIN StartLineTask */
  /* Infinite loop */
  for(;;)
  {
      status = HAL_I2C_Mem_Read(&hi2c1, SX1509_I2C_ADDR1 << 1,
          REG_DATA_B, 1, &line_sensor, 1, I2C_TIMEOUT);
      if( status != HAL_OK )
         error = 5;
    osDelayUntil(10);
  }
  /* USER CODE END StartLineTask */
}
```

```
69
70  /* USER CODE BEGIN Header_StartCommTask */
71  /**
72  * @brief Function implementing the commTask thread.
73  * @param argument: Not used
74  * @retval None
75  */
76  /* USER CODE END Header_StartCommTask */
77  void StartCommTask(void *argument)
78  {
79    /* USER CODE BEGIN StartCommTask */
80    /* Infinite loop */
81    for(;;)
82    {
83        ertc_dlog_update(&logger);
84        ertc_dlog_send(&logger, &loggerdata, sizeof(loggerdata));
85
86      osDelayUntil(10);
87    }
88    /* USER CODE END StartCommTask */
89  }
```

Test your code. Everything should be working as before. The main difference is that now your tasks are seamlessly handled by the operating system and not by a low level timer anymore.

5 Synchronization

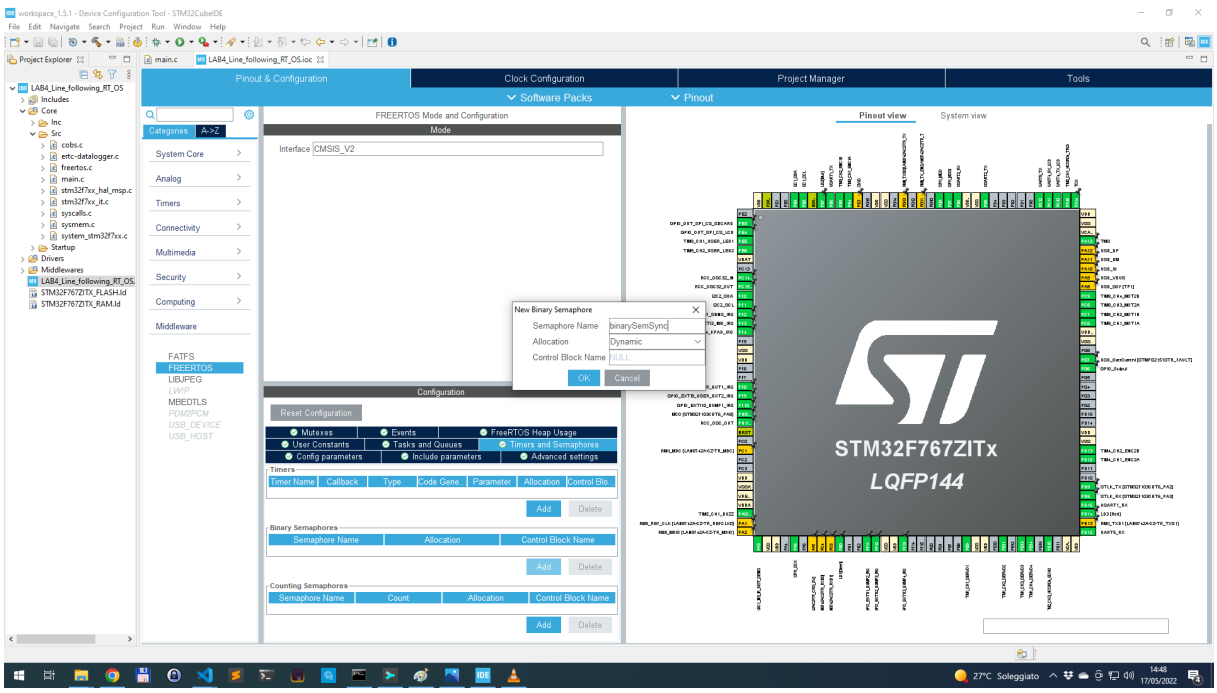
Suppose that we want to synchronize the execution of the controlTask and the lineTask. In other words, we want to make sure that the controlTask is executed only after the lineTask so when we received a measure from the sensor. To do this, we need to add a semaphore.

5.1 Binary semaphore

Open the `*.ioc` file, navigate to the *Middleware* tab and than click *FreeRTOS*.

Click on the *Timers and Semaphores* tab.

Than under the *Binary Semaphores* panel click on the *Add* button.



Call it `binarySemSync`.

Save and generate the code.

Somewhere in the main function you will find the following code

```
1  binarySemSyncHandle = osSemaphoreNew(1, 1, &binarySemSync_attributes);
```

Since we need to suspend the execution if *lineTask* has not been executed, change the initial value of the counter to 0. In other words you have to change the instruction above to

```
1  binarySemSyncHandle = osSemaphoreNew(1, 0, &binarySemSync_attributes);
```

6 Exercise 2

Place `osSemaphoreAcquire(binarySemSyncHandle, osWaitForever);` as a first line inside the for loop of the `StartControlTask` function. `osSemaphoreAcquire` is a FreeRTOS primitive that is used to suspend the execution of the task until the semaphore is available (during the lectures we called this *wait*).

```
1  void StartControlTask(void *argument)
2  {
3    /* USER CODE BEGIN StartControlTask */
4    /* Infinite loop */
5    for (;;)
```

```
6     {
7       osSemaphoreAcquire(binarySemSyncHandle, osWaitForever);
8       .
9       .
10      .
11    }
12  }
```

Try to run the code. Is it working? Are the motors spinning? If everything is correct, controlTask must be suspended and the motors must be stopped. **Discuss why**.

To allow the execution of the controlTask, we need to release the semaphore. To do this we need to use the function `osSemaphoreRelease(binarySemSyncHandle);`.

1. Place `osSemaphoreRelease(binarySemSyncHandle);` in a proper position inside the `lineTask` function.
2. Is the `osDelay(10)` inside the controlTask still need? If not, why? Every how many ms is the controlTask executed?

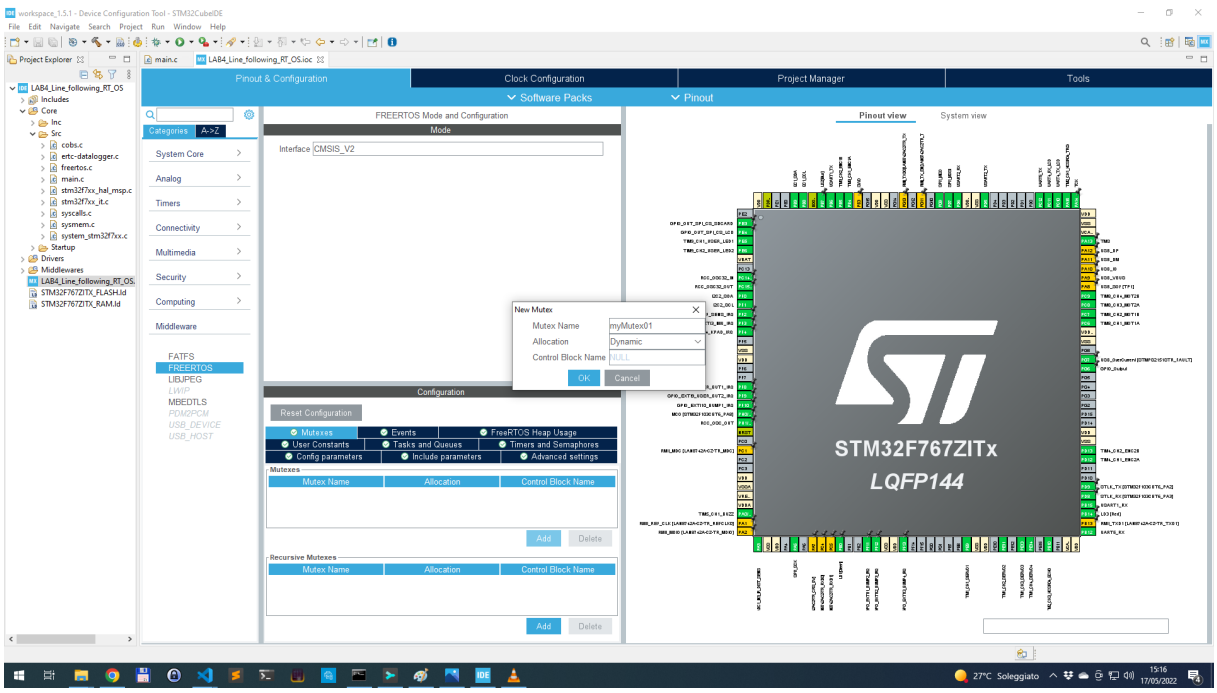
7 Mutex

Mutex is a synchronization primitive that can be used to protect shared resources from concurrent access. It allows to define a critical section of code that can be executed by only one task at a time.

To create a mutex, open the `*.ioc` file, navigate to the *Middleware* tab and than click *FreeRTOS*.

Click on the *Mutexes* tab.

Than under the *Mutex* panel click on the *Add* button. For example, call it *DataMutex*.



Save and generate the code.

Just to give you an example, in our code we can define a critical section as follows:

```
osMutexAcquire(DataMutexHandle, osWaitForever);
/* Enter Critical section */

loggerdata.w1 = w1_meas_rpm;
loggerdata.w2 = w2_meas_rpm;
loggerdata.u1 = u1_sat_V;
loggerdata.u2 = u2_sat_V;

/* Exit Critical section */
osMutexRelease(DataMutexHandle);
```

8 Exercise 3

Perform an assessment of your code. Try to identify possible critical sections and discuss why they are needed. Discuss about the consequences if the critical section is not protected by a mutex.

9 Esercize 4

Try to replace the binary semaphore of exercise 2 with a queue to pass data from the lineTask to the controlTask. Correctly configuring the `.ioc`, creating the queue, and selecting the right os primitives to call is on you.