Laboratory Task FreeRTOS

Mandatory (group of 2 students) - 1p towards final grade

This document describes how to get started with FreeRTOS on STM32L4-Nucleo.

You will need:

- STM32L4 Nucleo
- STM32CubeIDE installed on your computer
- UM1884: Description of STM32L4/L4+ HAL and low-layer drivers
- UM1722 User manual, Developing Applications on STM32Cube with RTOS

The following documents can be useful sources of information when working with the lab.
Nicolas Melot, Study of an operating system: FreeRTOS

FreeRTOS documentation and manual:
https://www.freertos.org/Documentation/RTOS_book.html

You can also find information on www.freertos.org specifically in the API references.

The Device Configuration Tool STM32CubeMX which is part of STM32CubeIDE is used to generate the initialization code for the development board and FreeRTOS. This is done in the same way as in the PingPong lab.

Start a new project for STM32L476 Nucleo-64. Be sure to select the development board (Board Selector) and not the microcontroller. In the menu “Clock Configuration” you don’t need to change the settings but verify that Sysclock is set to 80MHz.

Also check that there is a clock connected to the peripherals and that nothing is marked in red.

In the tab Middleware, enable FreeRTOS CMSIS_V2.

Under Project Manager / Code Generator you can configure to generate the initialization code for peripherals as pairs of .c/.h-files.
Make sure to select ‘*Keep User Code when regenerating*’.

It is appropriate to select a time base other than SysTick under the Pinout tab in SYS, for example TIM1.

Generate the Code:

Project -> Generate Code

You now have a project that can be compiled and executed on the development board. However, there will be no output visible as the program will be in a while loop forever.

From the code generation we received an executable skeleton which we will need to populate with our application code.

Test if you can execute the program on your board.

Some tips:

- If you place a breakpoint inside the while-loop after `osKernelStart();` you can see if the RTOS has started correctly. Since the execution should never reach the while-loop.
- If the RTOS is not starting you can try to increase the value of TOTAL_HEAP_SIZE in the FreeRTOS config parameters in CubeMX or the value of `configTOTAL_HEAP_SIZE` in the file FreeRTOSConfig.h

Many of the configuration parameters can either be changed in the CubeMX perspective of STM32CubeIDE or in the header file FreeRTOSConfig.h directly. It is recommended to change the configuration parameters in the CubeMX perspective as the file FreeRTOSConfig.h is overwritten when code is generated from CubeMX.

Now it is time to have a closer look at the code that is generated by CubeMX.
We start by looking at `main()` which looks as follows:

```
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();
  /* USER CODE BEGIN 2 */
  /* USER CODE END 2 */

  /* Call init function for freertos objects (in freertos.c) */
  MX_FREERTOS_Init();

  /* Start scheduler */
  osKernelStart();

  /* We should never get here as control is now taken by the scheduler */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}
```

As in the previous lab, comments mark where user-code can be added. First, all functions are called that initialize the HAL, clocks and GPIO. If you take a look at `MX_GPIO_Init` in the file `gpio.c` you will recognize the initialization code of the GPIO port and pins.

Open the file `freertos.c` here you can find the function `MX_FREERTOS_Init()`:

```
void MX_FREERTOS_Init(void); /* (MISRA C 2004 rule 8.1) */
/**
  * @brief  FreeRTOS initialization
  * @param  None
  * @retval None
  */
void MX_FREERTOS_Init(void) {
  /* USER CODE BEGIN Init */

  /* USER CODE END Init */
osKernelInitialize();

  /* USER CODE BEGIN RTOS_MUTEX */
  /* add mutexes, ... */
  /* USER CODE END RTOS_MUTEX */

  /* USER CODE BEGIN RTOS_SEMAPHORES */
  /* add semaphores, ... */
  /* USER CODE END RTOS_SEMAPHORES */

  /* USER CODE BEGIN RTOS_TIMERS */
  /* start timers, add new ones, ... */
  /* USER CODE END RTOS_TIMERS */

  /* USER CODE BEGIN RTOS_QUEUES */
  /* add queues, ... */
  /* USER CODE END RTOS_QUEUES */

  /* Create the thread(s) */
  /* definition and creation of defaultTask */
  const osThreadAttr_t defaultTask_attributes = {
    .name = "defaultTask",
    .priority = (osPriority_t) osPriorityNormal,
    .stack_size = 128
  };
  defaultTaskHandle = osThreadNew(StartDefaultTask, NULL, &defaultTask_attributes);

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

There must be at least one task configured before the operating system starts. In this case this task is called `StartDefauktTask`. You can find the declaration and definition further down in the same file `freertos.c`.

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

As you can see, this task executes an infinite loop and calls the delay function for 1ms every iteration.

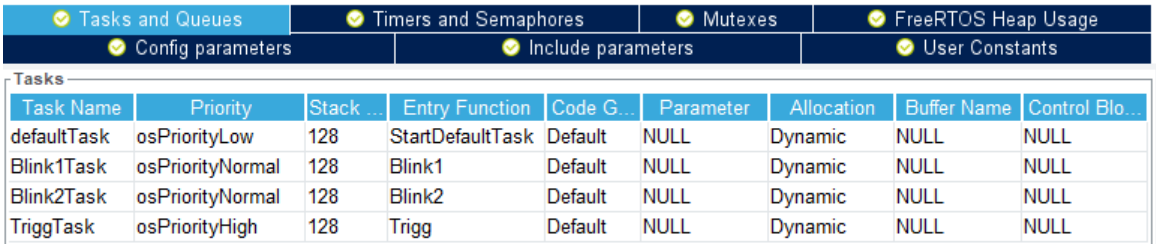
As task is declared to have a void return value and a parameter of type (void *).

A task should never return, this is why we need an infinite loop in the task code.

Task 1

The task will be discussed during the individual demonstration of the RTOS lab. You can add answers to the questions below directly to the lab manual, no separate report is needed.

We can use the CubeMX perspective in STM32CubeIDE to define the tasks we want to create. The code-generation then prepares the skeleton of those tasks for us.
Add three tasks: `Blink1Task`, `Blink2Task` and `TriggTask`.



Task Name	Priority	Stack ...	Entry Function	Code G...	Parameter	Allocation	Buffer Name	Control Blo...
defaultTask	osPriorityLow	128	StartDefaultTask	Default	NULL	Dynamic	NULL	NULL
Blink1Task	osPriorityNormal	128	Blink1	Default	NULL	Dynamic	NULL	NULL
Blink2Task	osPriorityNormal	128	Blink2	Default	NULL	Dynamic	NULL	NULL
TriggTask	osPriorityHigh	128	Trigg	Default	NULL	Dynamic	NULL	NULL

Generade the code!

The following code is generated in `freertos.c`:

```
  /* Create the thread(s) */
  /* definition and creation of defaultTask */
  const osThreadAttr_t defaultTask_attributes = {
    .name = "defaultTask",
    .priority = (osPriority_t) osPriorityLow,
    .stack_size = 128
  };
  defaultTaskHandle = osThreadNew(StartDefaultTask, NULL, &defaultTask_attributes);

  /* definition and creation of Blink1Task */
  const osThreadAttr_t Blink1Task_attributes = {
    .name = "Blink1Task",
    .priority = (osPriority_t) osPriorityNormal,
    .stack_size = 128
  };
  Blink1TaskHandle = osThreadNew(Blink1, NULL, &Blink1Task_attributes);

  /* definition and creation of Blink2Task */
  const osThreadAttr_t Blink2Task_attributes = {
    .name = "Blink2Task",
    .priority = (osPriority_t) osPriorityNormal,
    .stack_size = 128
  };
  Blink2TaskHandle = osThreadNew(Blink2, NULL, &Blink2Task_attributes);

  /* definition and creation of TriggTask */
  const osThreadAttr_t TriggTask_attributes = {
    .name = "TriggTask",
    .priority = (osPriority_t) osPriorityHigh,
    .stack_size = 128
  };
  TriggTaskHandle = osThreadNew(Trigg, NULL, &TriggTask_attributes);
```

Add the following two global variables to your file where it says: “Private variables”:

```
/* Private variables ---------------------------------------------------------*/
/* USER CODE BEGIN Variables */

uint8_t varBlink1 = 0;
uint8_t varBlink2 = 0;

/* USER CODE END Variables */
```

Add the following function prototype to the section for private function prototypes at the beginning of the file:

```
/* USER CODE BEGIN FunctionPrototypes */
void wait_cycles( uint32_t n );
/* USER CODE END FunctionPrototypes */
```

Add the implementation of the function to the end of the file at the place for application code:

```
/* USER CODE BEGIN Application */
void wait_cycles( uint32_t n ) {
    uint32_t l = n/3; //cycles per loop is 3
    asm volatile( "0:" "SUBS %[count], 1;" "BNE 0b;" :[count]"+r"(l) );
}
/* USER CODE END Application */
```

This function is used to simulate execution on the processor for a given amount of clock cycles and we will use it in our tasks to simulate execution.

Add code to `Blink1`, `Blink2` as shown below:

```
/* USER CODE BEGIN Header_Blink1 */
/**
 * @brief Function implementing the Blink1Task thread.
 * @param argument: Not used
 * @retval None
 */
/* USER CODE END Header_Blink1 */
void Blink1(void *argument)
{
        /* USER CODE BEGIN Blink1 */
        /* Infinite loop */
        for(;;)
        {
                varBlink1 = 1;
                wait_cycles(200000);
                varBlink1 = 0;
                vTaskDelay(10);
        }
        /* USER CODE END Blink1 */
}

/* USER CODE BEGIN Header_Blink2 */
/**
 * @brief Function implementing the Blink2Task thread.
 * @param argument: Not used
 * @retval None
 */
/* USER CODE END Header_Blink2 */
void Blink2(void *argument)
{
        /* USER CODE BEGIN Blink2 */
        /* Infinite loop */
        for(;;)
        {
                varBlink2 = 1;
                wait_cycles(400000);
                varBlink2 = 0;

                vTaskDelay(20);
        }
        /* USER CODE END Blink2 */
}

/* USER CODE BEGIN Header_Trigg */
```

```
/**
 * @brief Function implementing the TriggTask thread.
 * @param argument: Not used
 * @retval None
 */
/* USER CODE END Header_Trigg */
void Trigg(void *argument)
{
        /* USER CODE BEGIN Trigg */
        TickType_t xLastWakeTime;
        const TickType_t xPeriod = pdMS_TO_TICKS(200) ;  // ms to ticks
        // Initialise the xLastWakeTime variable with the current time.
        xLastWakeTime = xTaskGetTickCount();

        /* Infinite loop */
        for(;;)
        {
                vTaskDelayUntil( &xLastWakeTime, xPeriod );
                wait_cycles(10); //add a breakpoint in this line
        }
        /* USER CODE END Trigg */
}
```

For the function `vTaskDelay(10)` the parameter 10 tells how many ticks the execution is delayed. The granularity of the operating system tick for FreeRTOS tick can be set in CubeMX. The default value for `configTICK_RATE_HZ` is 1000, this means 1 ms for each operating system tick. In this lab, the tasks do no real work and we use the function `wait_cycles()` to loop for a given number of CPU cycles.

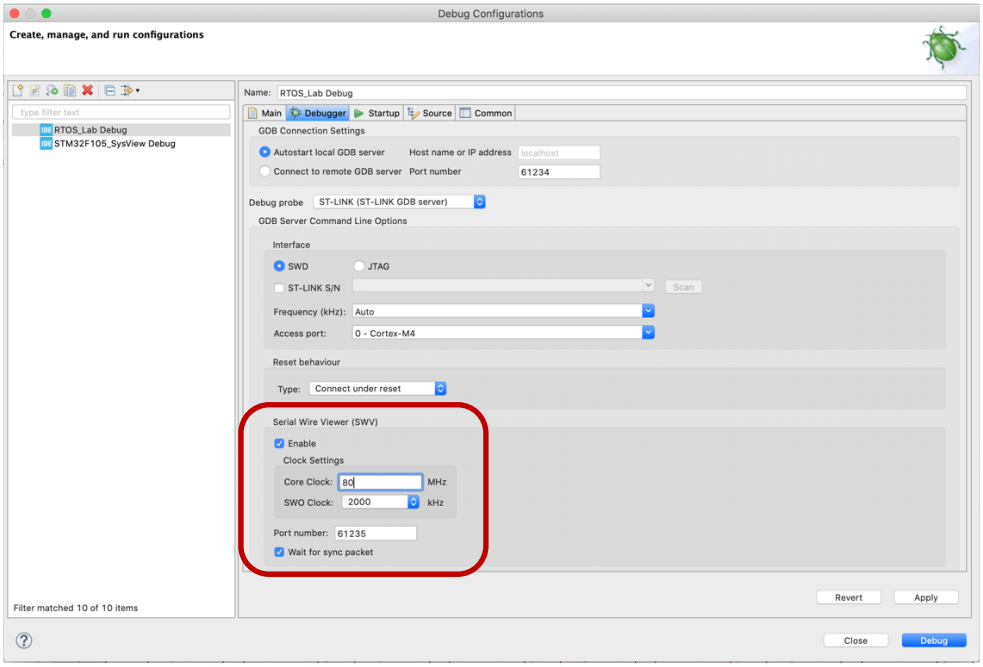
Set a breakpoint to the line `wait_cycles(10);` of the function `Trigg()`.

In the lecture video Module 4 that discusses debugging we saw how we can use the Serial Wire Viewer (SWV) to trace variables at runtime. This technique we will use to get an understanding how the tasks are executing

We will observe the two variables `varBlink1` and `varBlink2`.

For a how-to video on the usage of SWV please revisit the lecture video. Below the main points are outlined:

In the debug configuration, enable Serial Wire Viewer (SWV) and set the core clock to 80 MHz.



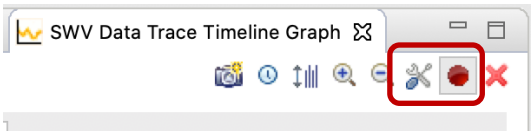
Start a debug session.

The execution will automatically be stopped at the first instruction of the `main()` function.

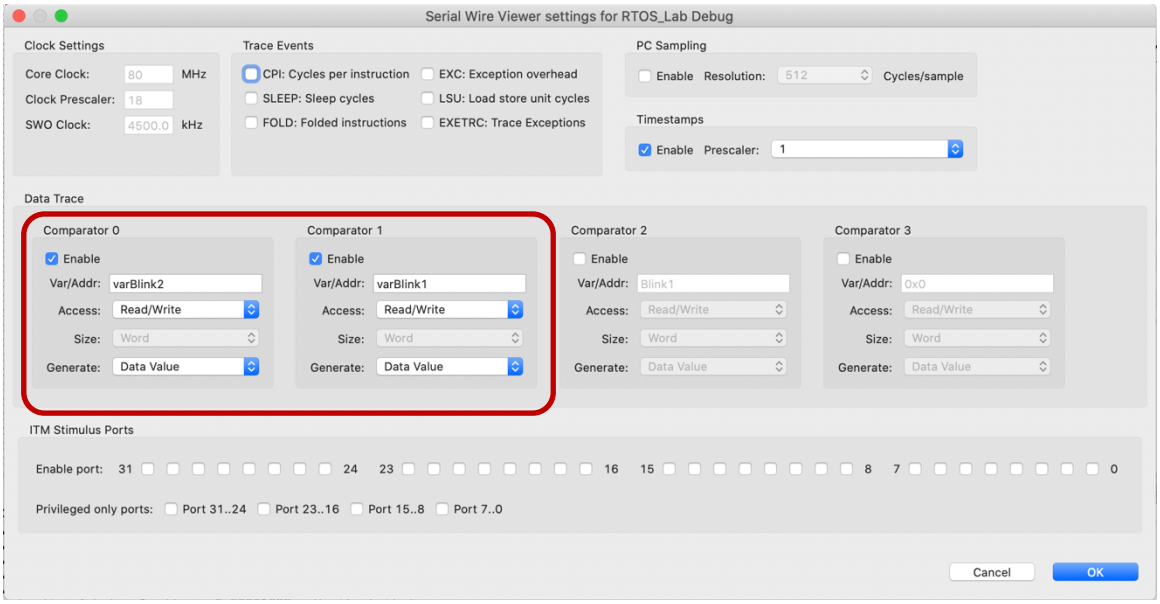
Now we can configure the tracing of the two variables.

For Open the SWV Data Trace Timeline Graph view, you can find it under “Window -> Show View -> SWV -> SWV Data Trace Timeline Graph”.

In the SWV Data Trace Timeline Graph view, open the configuration (by clicking on the tools symbol):



Here we need to configure the two variables we want to observe as shown below:

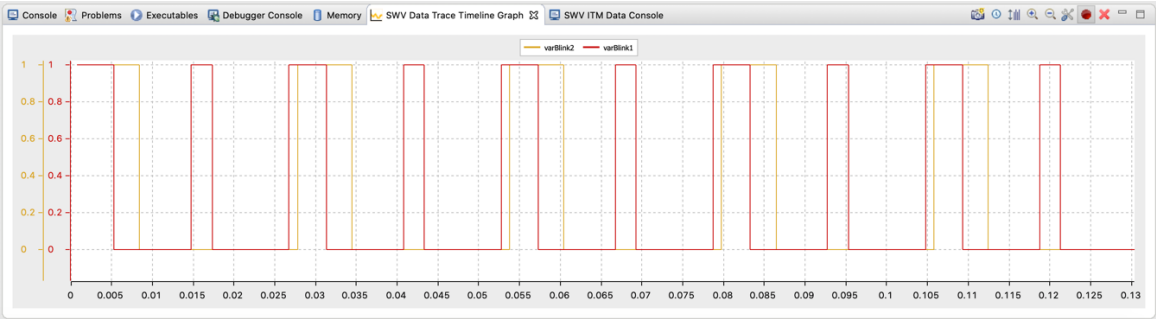


Click “OK” and start the recording by clicking the red “record” button in the SWV Data Trace Timeline Graph View.

When we resume the execution during the debug session the state of the two variables will be recorded and displayed in the view.

Resume the execution. Once the breakpoint in `Trigg()` is reached, we have recorded all the data we need for now.

Adjust the view so you can see the two traces in a good way (you need to zoom in and scroll to the beginning of the trace).
You should see a similar view than below:



Study the recorded trace. How often does a context switch occur between tasks?

Comment: Never because they have the same priority

How long does it take to complete a task from the time it starts? With what period do the tasks run?

Blink1

Finished after: Max 0.0045 Min 0.003

Period: Psf = 0.0125 and Pfs = 0.014

Blink2

Finished after: 0.0065

Period: 0.026

As you can see, the period depends on when a task has been completed because the delay is added after the execution has been completed. If we are to have fixed period times, we must specify a delay relative to the time when a task starts.

We can use `vTaskDelayUntil()` for this purpose.

Change Blink1 as follows:

```
void Blink1(void *argument)
{
        /* USER CODE BEGIN Blink1 */
        TickType_t xLastWakeTime;
        const TickType_t xPeriod = pdMS_TO_TICKS(10) ;  // ms to ticks
        // Initialise the xLastWakeTime variable with the current time.
        xLastWakeTime = xTaskGetTickCount();

        /* Infinite loop */
        for(;;)
        {
                varBlink1 = 1;
                waitCycle(200000);
                varBlink1 = 0;
                vTaskDelayUntil( &xLastWakeTime, xPeriod );
        }
        /* USER CODE END Blink1 */
}
```

Apply the same change to Blink2 but with a period of 20 ms.

Assume that Blink1 has a release time of 0 ms, 10 ms, 20 ms etc.

Assume that Blink2 has a release time of 0 ms, 20ms, etc.

The deadline of each task is equal to its period.

Change priorities for tasks and measure times from the release time, ie. when ready to run, for the first 20 milliseconds. Blink1 has time to run twice.

Prio Blink1 **>** Prio Blink2: Blink 1 finished after `0.0025` and `0.0025` and Blink2 after `0.005`

Prio Blink1 **=** Prio Blink2: Blink 1 finished after `0.0045` and `0.0025` and Blink2 after `0.0065`

Prio Blink1 **<** Prio Blink2: Blink 1 finished after `0.0025` and `0.0025` and Blink2 after `0.005`

Based on your measurements calculate the hyperperiod and the utilization:

For Prio Blink1 < Prio Blink2

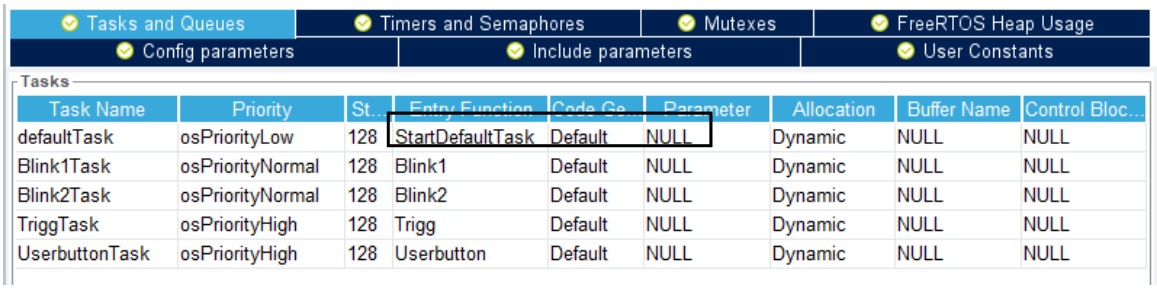
Hyperperiod: `LCM: (P1 & P2) = (0.01 & 0.02) = 0.2`

Utilization (%) `Used time / Whole period` `(0.0025+0.0025+0.005)/(0.026-0.006) = 50%`

How much idle time (unused time) is there? `half of Hyperperiod` `0.02/2 = 0.01`

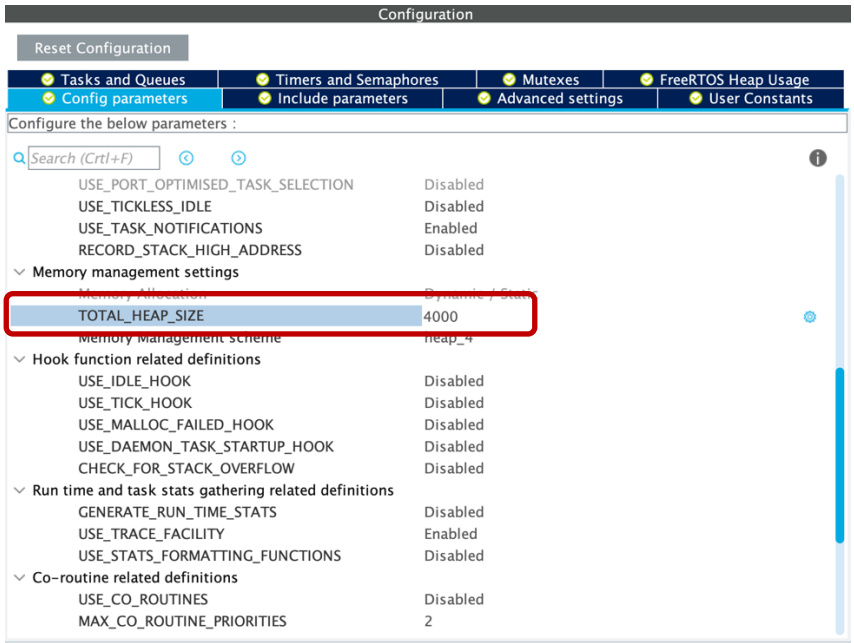
Can both tasks `Blink1` and `Blink2` meet their deadlines? `Yes! period 10ms and 20ms is larger than execution time for the tasks`

Add an additional task, `Userbutton`:



Task Name	Priority	St...	Entry Function	Code Ge...	Parameter	Allocation	Buffer Name	Control Bloc...
defaultTask	osPriorityLow	128	StartDefaultTask	Default	NULL	Dynamic	NULL	NULL
Blink1Task	osPriorityNormal	128	Blink1	Default	NULL	Dynamic	NULL	NULL
Blink2Task	osPriorityNormal	128	Blink2	Default	NULL	Dynamic	NULL	NULL
TriggTask	osPriorityHigh	128	Trigg	Default	NULL	Dynamic	NULL	NULL
UserbuttonTask	osPriorityHigh	128	Userbutton	Default	NULL	Dynamic	NULL	NULL

After the task is added a warning appears as the configured heap for FreeRTOS is not large enough. In the “Config Parameters” section you can increase the heap size.



Add the implementation for the function `UserButton()` as shown below:

```
/* USER CODE BEGIN Header_Userbutton */
/**
 * @brief Function implementing the UserbuttonTask thread.
 * @param argument: Not used
 * @retval None
 */
/* USER CODE END Header_Userbutton */
void Userbutton(void *argument)
{
        /* USER CODE BEGIN Userbutton */
        TickType_t xLastWakeTime;
        const TickType_t xPeriod = pdMS_TO_TICKS(20);
        // Initialise the xLastWakeTime variable with the current time.
        xLastWakeTime = xTaskGetTickCount();

        /* Infinite loop */
        for(;;)
        {
                if (HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin)==GPIO_PIN_RESET)
                {
                        varBlink3 = 1;
                        wait_cycles(250000);
                        varBlink3 = 0;          }
                vTaskDelayUntil( &xLastWakeTime, xPeriod );
        }
        /* USER CODE END Userbutton */
}
```

Add `varBlink3` to the declaration of `varBlink1` and `varBlink2`.

The execution of `UserButton()` starts once you press the user button, which is the blue button on the Nucleo-board. Before you start to execute the program, give an estimate if all tasks meet their deadlines.

Compute how much time `UserButton()` requires once you press the button:

0.003 from graph or wait_cycles/clock frequency = 250000/8*10^6 = 0.003125

Do the tasks `Blink1`, `Blink2`, and `UserButton` meet their deadlines? Yes, its a short execution time compared to period

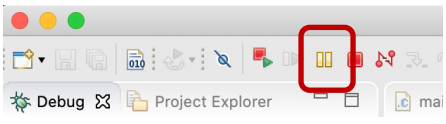
Now we are executing the program on the microcontroller!

Add `varBlink3` to the serial wire viewer settings, the same way as we did it for `varBlink1` and `varBlink2`.

Remove the breakpoint from the function `Trigg()`.

Start recording of the trace and let the program run.

Press the user button for a bit and after break the execution by clicking on the break symbol in the IDE:



Now visit the SWV Data Trace Timeline Graph view. You will see that the task was executing while you pressed the user button. Zoom into the section where all 3 tasks were executing and evaluate:

Do all tasks `Blink1()`, `Blink2()` and `UserButton()` meet their deadline? Yes all instructions fits in its period

meet Deadline = if execution time <= Period

Period = 10ms
Execution time from graph: Blink1 = 4.5ms ja!

Period = 20ms
Execution time from graph: Blink1 = 7ms ja!

Period = 20ms
Execution time from graph: Blink1 = 3ms ja!

Task 2

Rewrite `Blink1()` that it toggles the green LED on the Nucleo-Board once every 100ms. Remove `Blink2()` or deactivate it in some way.

Rewrite `UserButton()` that it blocks blinking in `Blink1()` when the button is pressed and allows blinking in `Blink1()` when the button is not pressed.

Use operating system mechanisms to realize blocking of Blink1 while the button is pressed. Use some type of semaphore or mutex, see the lecture video and/or the FreeRTOS manual for more details.

Individual Lab Demonstration:

For the individual demo of the RTOS lab be prepared to discuss the measurement results of Task 1 and explain / show how you realized the functionality of Task 2.