Programming industrial embedded systems
Laboratory exercise 3
FreeRTOS on STM32F4DISCOVERY

January 4, 2022

Contents

1 Required hardware, software and additional literature

For this laboratory exercise, you will need the following:
hardware:

- STM32F4DISCOVERY development kit,
- USB-mini cable for power supply and programming.

software:

- cross compiler for STM32F4,
- ST-link for flashing,
- STM32CubeMX,
- STM32CubeIDE or *GNU Make* environment,
- FreeRTOS sources (available for download from here).

additional literature and materials:

- materials and manuals from FreeRTOS web page [6], [4],
- PIES lecture notes on FreeRTOS.

Additional materials for this lab exercise are packed in the archive *PIES_2020_Lab3_Pack.zip*, available from FER web page, under the section *Laboratory exercises.*

2 Building a minimum FreeRTOS project for STM32F4 microcontroller

Before integrating FreeRTOS functionality in our project, we need to establish a minimum working project as described in the first lab exercise (chapter 3). We shall simply use the working Blinky example (with prebuilt STM32F4 HAL library) as a base that we shall extend to include the FreeRTOS functionality.

The following section will demonstrate how to configure all the resources required for a FreeRTOS application through STM32CubeMX. If you want to add FreeRTOS sources to your project build manually, follow the steps described in Appendix A and then continue to section 3.

2.1 Generating FreeRTOS code in STM32CubeMX

In STM32CubeMX, load the GPIO LED project from the first lab exercise. To open *FreeRTOS Mode and Configuration* window, open the *Middleware* panel and select *FreeRTOS*. In the *Mode* panel, choose *CMSIS_V1* interface.

FreeRTOS is customised using a configuration file `FreeRTOSConfig.h`. Every FreeRTOS application must have a `FreeRTOSConfig.h` header file in its pre-processor include path. This file is used for configuring application-specific parameters [3].

These parameters (as well as FreeRTOS objects, such as memory management settings, tasks, timers and semaphores) can be configured through *Configuration* panel in STM32CubeMX.

In this exercise we shall use the simplest memory management algorithm *Heap1*. Go to *Memory management settings* and select *heap_1* under *Memory management scheme*. This setting will include file `heap_1.c` in the source code.

Under *Kernel settings*, set the `TICK_RATE_HZ` parameter to 100. This setting means that the time slice for each task is 10ms. It is recommended to keep task scheduler tick frequency at 1kHz or less.

Other settings can be left at their default values and the user is encouraged to study how each setting will affect the system.

The next step is to configure clock parameters. In this exercise, we will use HSE oscillator as a source for system clock. We will set the system clock frequency to 8 MHz. In *Clock Configuration* window, set the input frequency to 8 MHz and select *HSE* under *System Clock Mux*, as shown in Fig. 1.

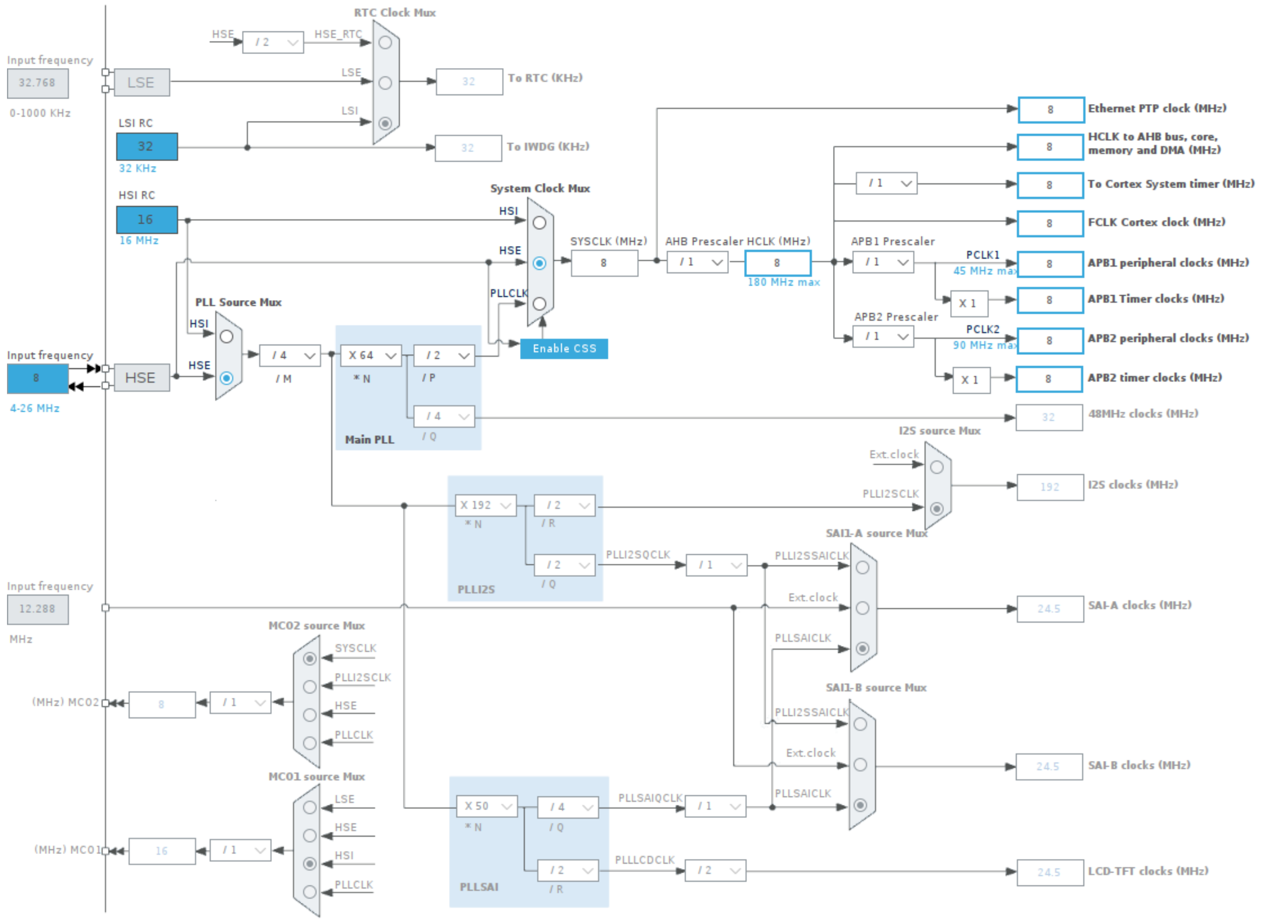

Figure 1: Settings in the Clock Configuration window.

Click on *Generate code*.

2.2 Organization of FreeRTOS source code

FreeRTOS source code is generated under your project folder, e.g. `Lab_GPIO/Middlewares/Third_Party/FreeRTOS`. The project source tree should look like this:

```
./
    Core/
    Drivers/
    Middlewares/
        Third_Party/
            FreeRTOS/
                ...
    Lab_GPIO.ioc
    Makefile
    startup_stm32f407xx.s
    STM32F407VTx_FLASH.ld
```

FreeRTOS kernel resides in `FreeRTOS/Source` folder of the source tree. The architecture dependent code is contained in the `FreeRTOS/Source/portable` subdirectory.

For more information, read the documents *FreeRTOS - FreeRTOS Source Code Directory Structure* [5] (description of FreeRTOS source code organization) and *FreeRTOS - Creating a new RTOS project* [1] (description of the minimum set of files that must be included in target project).

2.3 Configuring FreeRTOS

The next step is to make sure that all the settings are configured properly in the `FreeRTOSConfig.h` file. Please read the document *FreeRTOS Configuration header* [2] before proceeding.

In this exercise, we do not use idle and tick hooks (callbacks). These functions are simply excluded from the build by putting zero value at appropriate defines:

```
#define configUSE_IDLE_HOOK                      0
#define configUSE_TICK_HOOK                      0
```

If the user wants to use these functions, they can be added to `main.c`, for example:

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"

uint64_t u64Ticks=0;      // Counts OS ticks ( default = 1000 Hz )
uint64_t u64IdleTicks=0;    // Counts when the OS has no task to execute.

// This FreeRTOS callback function gets called once per tick.
void vApplicationTickHook( void ) {
  ++u64IdleTicks;
}

// This FreeRTOS callback function gets called when no other task is ready to execute.
void vApplicationIdleHook( void ) {
  ++u64IdleTicks;
}
```

One of the most important things during the FreeRTOS setup is to establish the right timings. Therefore, it is essential before going any further to check whether system clock rate is defined correctly, as all timings depend on that parameter (`configCPU_CLOCK_HZ`).

According to the settings we configured in STM32CubeMX, the parameter `configCPU_CLOCK_HZ` is defined in the following way (`FreeRTOSConfig.h`):

```
#define configCPU_CLOCK_HZ                       ( SystemCoreClock )
```

The `SystemCoreClock` variable is defined in the file `Core/Src/system_stm32f4xx.c`. In this case, we are using HSE as a source for the system clock (see System Clock Mux on Fig. 1). The value of HSE clock is defined by a preprocessing symbol in the file `Core/Inc/stm32f4xx_hal_conf.h`:

```
 #if !defined  (HSE_VALUE)
  #define HSE_VALUE    ((uint32_t)16000000U)
  /*!< Value of the External oscillator in Hz */
 #endif /* HSE_VALUE */
```

The actual CPU frequency in our example corresponds to HSE frequency and equals 16 MHz.

The RTOS task scheduler tick frequency is set in `Core/Inc/FreeRTOSConfig.h` file:

```
#define configTICK_RATE_HZ                       ((TickType_t)100)
```

At this point, we are ready to integrate the FreeRTOS functionality to our Blinky project.

3 Blinky with FreeRTOS

Assignment: The goal of this part of lab exercise is to demonstrate how to include FreeRTOS functionality in your program (for STM32F4DISCOVERY board) and how to make the simpliest application with a single task doing some periodic activity.

Make the *Blinky* program based on the code from chapter 3 in 1st lab exercise. Instead of using the *while* loop in the main program and *for* loops for implementing some undefined time delay, launch a single RTOS task that will use the function `vTaskDelay()` to toggle the LED state every second.

Guidelines:

3.1 Adding FreeRTOS functionality to the main function

To include FreeRTOS functionality in the `main.c` (or any other module that uses FreeRTOS) it is necessary to include some header files first:

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"
```

In this solution we shall use FreeRTOS task to periodically toggle LED state:

```
void vTask1(void *pvParameters)
{
    while(1) {
        gpio_led_state(LED5_RED_ID, 1);          // turn off
        gpio_led_state(LED6_BLUE_ID, 1);         // turn off
        vTaskDelay(1000 / portTICK_RATE_MS);     // delay 1s
        gpio_led_state(LED5_RED_ID, 0);         // turn on
        gpio_led_state(LED6_BLUE_ID, 0);        // turn on
        vTaskDelay(1000 / portTICK_RATE_MS);    // delay 1s
    }
}
```

Task function does not return value and takes a single (`void*`) argument. Tasks are typically implemented as infinite loops (with blocking), doing some repetitive job or reacting to events. In this case our task will toggle LED state each second.

The function call `vTaskDelay(ticks)` will cause the current task to block for *ticks* number of task scheduler ticks. Tick duration is determined by a parameter `configTICK_RATE_HZ` but we do not use this frequency directly in API function parameters. Instead, the recommended pattern is to use `portTICK_RATE_MS` macro that will automatically return number of milliseconds derived from `configTICK_RATE_HZ` parameter. In order to make the code independent of `configTICK_RATE_HZ` setting, we can easily write:

```
vTaskDelay(DELAY_MS / portTICK_RATE_MS);
```

where `DELAY_MS` is a desired delay in milliseconds for some function call. Delay should be independent of `configTICK_RATE_HZ` changes, what is accomplished by a construct above.

All we have to do now is to create and run `vTask1()` from the main function:

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();

    xTaskCreate(vTask1,                 // pointer to task function
        (const signed char*)"TASK1",    // task name
        configMINIMAL_STACK_SIZE,       // stack size
        NULL,                           // task input parameters -not used
        1,                              // task priority
        NULL);                          // task handle-not used

    vTaskStartScheduler();              // run task scheduler

    while(1);
}
```

At this point we have our *Blinky* program up and running, toggling the LEDs at the rate of one second.

4 Multitasking blinky

Assignment: The goal of this part of exercise is to learn how to create, run and delete tasks in multitasking FreeRTOS environment.

Guidelines:

4.1 Configuring push button GPIO input

Create a new project with FreeRTOS support following the guidelines in chapter section 2. In this part of lab exercise we shall use on-board push button to simulate external events. Push button is wired to PA0 GPIO port pin (see the STM32F4DISCOVERY board manual [7]).

We will configure the push button to trigger an external interrupt whenever it is pressed. In STM32CubeMX *Pinout* window, initialize pin PA0 as `GPIO_EXTI0`. Under *System Core - NVIC* panel, enable `GPIO_EXTI0` interrupt.

This part of the exercise needs to support task deletion. In order to be able to free memory occupied by deleted task TCB, we must use at least `heap_2.c` memory management algorithm. Select `heap_2` under *FreeRTOS - Configuration - Memory management scheme.*

Make sure that `vTaskDelete()` is enabled under *FreeRTOS - Configuration - Include parameters.*

Initialize the GPIO LED pins as described in the 1st Laboratory exercise. Click on *Generate Code.*

In order to avoid hard-coding the constants in the code, we will add the following definitions to `gpio.h`:

```
#define USER_BUTTON_PIN                GPIO_PIN_0
#define USER_BUTTON_GPIO_PORT          GPIOA
#define USER_BUTTON_GPIO_CLK           RCC_AHB1Periph_GPIOA
```

To keep track whether the push button was pressed, we will use a global variable:

```
uint8_t buttonPressed = 0;
```

We also need to write the `GPIO_EXTI0` interrupt callback function. Add the following code to `main.c`:

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if(GPIO_Pin == USER_BUTTON_PIN){
        HAL_NVIC_DisableIRQ(EXTI0_IRQn);
        buttonPressed = 1;
        HAL_NVIC_EnableIRQ(EXTI0_IRQn);
    }
    else{
        __NOP();
    }
}
```

4.2 Simple multitasking Blinky example

Our example will consist of four tasks, each task blinking a single on-board LEDwith repetition of 500ms:

Task	LED	Handler function
TASK1	Green LED	`void vTask1(void* pvParameters)`
TASK2	Orange LED	`void vTask2(void* pvParameters)`
TASK3	Red LED	`void vTask3(void* pvParameters)`
TASK4	Blue LED	`void vTask4(void* pvParameters)`

First we launch TASK1 that blinks green LED and periodically checks whether push button is pressed. If it is pressed, TASK1 launches the next task until all tasks are running. On the next push button press, the TASK1 will delete tasks TASK2, TASK3 and TASK4, waiting for next push button to launch TASK2 again. Task activity is signalled by blinking corresponding LED.

In the main function we have initialization code:

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();

    xTaskCreate(vTask1,                 // pointer to task function
        (const signed char*)"TASK1",  // task name
        configMINIMAL_STACK_SIZE,     // stack size
        NULL,                         // task input parameters -not used
        1,                            // task priority
        NULL);                        // task handle-not used

    vTaskStartScheduler();            // run task scheduler

    while(1);
}
```

The `MX_GPIO_Init()` function is described in Lab1 and it configures LED outputs using STM32F4 HAL API functions. We create TASK1 and start task scheduler. We want to keep track of task handlers in order to be able to delete tasks later. Therefore, at the beginning of `main.c`, we must define tasks handlers as global variables:

```
xTaskHandle xTaskHandle1, xTaskHandle2, xTaskHandle3, xTaskHandle4;
```

It is also important to provide forward definitions of task functions to be independent of their positional occurence in main.c source:

```
// forward declarations of task functions
void vTask1(void *pvParameters);
void vTask2(void *pvParameters);
void vTask3(void *pvParameters);
void vTask4(void *pvParameters);
```

These forward declarations should be placed either at the beginning of `main.c` module or in the `main.h` header file.

Now we shall write the code for TASK1:

```
void vTask1(void *pvParameters)
{
    uint32_t tasksCount, led_state;
    tasksCount = 1;                             // we have only one task at this point
    led_state  = 1;
    while(1) {
        vTaskDelay(500 / portTICK_RATE_MS);       // LED blinking frequency
        gpio_led_state(LED4_GREEN_ID, led_state); // green LED
        led_state = (led_state == 1) ? 0 : 1;
        if (buttonPressed) {
            buttonPressed = 0;
            tasksCount++;                         // key pressed - add new task
            if (tasksCount == 5) {                // kill all tasks except this
                vTaskDelete(xTaskHandle2);
                vTaskDelete(xTaskHandle3);
                vTaskDelete(xTaskHandle4);
                gpio_led_state(LED3_ORANGE_ID, 0);  // turn off all leds
                gpio_led_state(LED5_RED_ID, 0);
                gpio_led_state(LED6_BLUE_ID, 0);

                tasksCount = 1;
            } else {
                switch(tasksCount) {
                    case 2:
                        xTaskCreate(vTask2, "TASK2", configMINIMAL_STACK_SIZE,
                            NULL, 1, &xTaskHandle2);
                        break;
                    case 3:
                        xTaskCreate(vTask3, "TASK3", configMINIMAL_STACK_SIZE,
                            NULL, 1, &xTaskHandle3);
                        break;
                    case 4:
                        xTaskCreate(vTask4, "TASK4", configMINIMAL_STACK_SIZE,
                            NULL, 1, &xTaskHandle4);
                        break;
                }
            }
        }
    }
}
```

The code in TASK1 does the following:

- it runs an infinite loop and blinks green LED state every 500ms,
- every 500 ms it checks the state of the push button,
- if the push button is pressed at the moment of check, the task will increment the task counter variable `tasksCount`; if `tasksCount` is less than 5, the new task is created with the same priority as the current task; if `tasksCount` is equal to 5, all tasks except TASK1 are deleted.

The code for TASK2 is very simple:

```
void vTask2(void *pvParameters)
{
    uint32_t led_state;
    led_state = 1;
    while(1) {
        vTaskDelay(500 / portTICK_RATE_MS);             //LED blinking frequency
        gpio_led_state(LED3_ORANGE_ID, led_state);      // orange LED
        led_state = (led_state == 1) ? 0 : 1;
    }
}
```

TASK2 simply blinks orange LED every 500 ms. It is easy to write the same code for TASK3 and TASK4, with the only difference in the LED they will blink. Moreover, it is possible to write a single task function that will blink LED based on `pvParametersvalue`. Consider how to implement both solutions.

5 Activity synchronization with binary semaphores

Assignment:

The goal of this part of lab exercise is to show the basic usage of binary semaphores for inter-task activity synchronization.

Guidelines:

Create a new project with FreeRTOS support following the guidelines in chapter 3. Add support for push button following the instructions in chapter 4.

The system must have two tasks, TASK1 with lower priority (1) and TASK2 with higher priority (2). TASK1 blinks the green LED with repetition of 100 ms to signal that the task is active. TASK2 turns on red LED when it activates for a period of 2 seconds, and then turns it off before blocking itself. TASK2 will be initially blocked on a binary sempahore, waiting for TASK1 to release the sempahore on push button keypress event. TASK2 preempts TASK1 for 2 seconds and blocks on semphore waiting for a new keypress in TASK1 to activate again.

The main function should look like this:

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();

    xBinarySemaphore = xSemaphoreCreateBinary();
    if( xBinarySemaphore != NULL ) {
        xTaskCreate( vTaskLow, NULL, configMINIMAL_STACK_SIZE, NULL, 1, NULL );
        xTaskCreate( vTaskHigh, NULL, configMINIMAL_STACK_SIZE, NULL, 2, NULL );
        vTaskStartScheduler();
    }

    while(1);
}
```

First we initialize GPIO pins for LEDs and push button input. Then we create binary sempahore with the function call:

`xBinarySemaphore = xSemaphoreCreateBinary();`

It is necessary to declare semaphore handle as a global variable in `main.c`:

`xSemaphoreHandle xBinarySemaphore;`

Data type `xSemaphoreHandle` is used for all types of FreeRTOS semaphores (binary, counting and mutex) which differ only by a function that creates them (`xSemaphoreCreateBinary` in case of binary semaphore). Since a semaphore is a dynamic object, we must check whether the memory for semaphore could be reserved.

The newly created binary semaphore is initially in ”unavailable” state and the task will block if it tries to take it with function:

`xSemaphoreTake(xBinarySemaphore, portMAX_DELAY);`

Note: An older and deprecated function for creating binary semaphores `vSemaphoreCreateBinary()` can be invoked in a slightly different manner:

`vSemaphoreCreateBinary(xBinarySemaphore);`

(notice prefix v instead of x!). The principal difference to `xSemaphoreCreateBinary()` function is that binary semaphore created by `vSemaphoreCreateBinary()` is initially “available”, what is exactly the opposite behaviour comparing to `xSemaphoreCreateBinary()` function! The function `vSemaphoreCreateBinary()` is left in FreeRTOS solely for backward compatibility and it is not recommended for use in new projects.

Then we create tasks TASK1 and TASK2 with priorities 1 and 2, respectively, with appropriate handler functions (`vTaskLow` for TASK1 and `vTaskHigh` for TASK2).

Task scheduler will first execute higher priority task TASK2. The TASK2 handler function `vTaskHigh` looks like this:

```
void vTaskHigh( void *pvParameters )
{
    portTickType t1, t2;

    gpio_led_state( LED5_RED_ID, 0 );
    while(1) {
        xSemaphoreTake( xBinarySemaphore, portMAX_DELAY );
        gpio_led_state( LED5_RED_ID, 1 );
        t1 = xTaskGetTickCount();
        while(1) {
            t2 = xTaskGetTickCount();
            if( ( ( t2 - t1 ) * portTICK_RATE_MS ) >= 2000 ) {
               break;
            }
        }
        gpio_led_state( LED5_RED_ID, 0 );
    }
}
```

At the beginning of task execution we ensure that the red LED is turned off. Then we run an infinite while loop, trying first to acquire binary `semaphorexBinarySemaphore`:

`xSemaphoreTake(xBinarySemaphore, portMAX_DELAY);`

The parameter `portMAX_DELAY` denotes that we wait for an indefinitely long time period for binary semaphore to become available. Initially, it is unavailable because we have already taken it in the `main()` function during the semaphore creation process. The semaphore will be released by lower priority TASK1 on a button keypress event, as it will be shown later.

Once the semaphore is available, the TASK2 automatically preempts TASK1 because it has higher priority and signal it by turning on the red LED:

`gpio_led_state(LED5_RED_ID, 1);`

Next, we want to simulate some job that lasts for 2 seconds. Although we could use the function `vTaskDelay()` to suspend task execution for desired time period, it is not the behaviour we want to demonstrate here. Instead, we use "software-defined" *non-blocking* delay, implemented by polling number of RTOS task scheduler ticks. The function `xTaskGetTickCount` will return tick count at any moment and the time difference in milliseconds between two time instants `t1` and `t2` can be easily calculated using the formula:

`if (((t2 -t1) * portTICK_RATE_MS) >= 2000) break;`

After two seconds of non-blocking execution, the program exits the inner while loop in TASK2 and immediately turns off the red LED, before blocking on `xBinarySemaphore` (it cannot take binary semaphore twice until the TASK1 releases it). During the execution of inner loop, TASK1 is blocked because it has lower priority than TASK2 and there are no blocking API function calls in the inner loop.

Note: This example shows how to implement non-blocking delay by using the API function `xTaskGetTickCount` to simulate some useful work, but in practice in most cases it is desirable to use `vTaskDelayblocking` delay to yield processor to lower priority tasks waiting for their execution to resume. The code for TASK1 function looks like this:

```
void vTaskLow( void *pvParameters )
{
    uint32_t led_state, value;

    led_state = 1;
    while(1) {
        gpio_led_state( LED4_GREEN_ID, led_state );
        led_state = ( led_state == 1 ) ? 0 : 1;
        if( buttonPressed ) {
            buttonPressed = 0;
            xSemaphoreGive( xBinarySemaphore );
        }
        vTaskDelay( 100 / portTICK_RATE_MS );
    }
}
```

As the TASK2 blocks immediately on unavailable binary semaphore, TASK1 will be executed for indefinitely long (until keypress) after the system reset. TASK1 runs in an infinite loop and periodically blinks the green LED with repetition of 100ms. It also checks every 100ms whether the push button is pressed. If keypress event is detected, it gives the binary semaphore `xBinarySemaphore` which immediatelly unblocks higher priority TASK2 and preempts TASK1 execution. Green LED will stop blinking for 2 seconds and red LED will turn on for 2 seconds, signaling the TASK2 activitiy. TASK1 will resume its execution at the moment when TASK2 blocks again on `xBinarySemaphore`, after 2 seconds of non-blocking delay.

6 Lab outcomes

Students must individually accomplish the following outcomes:

- working STM32 projects which demonstrate the following functionalities:
  - *Blinky* with FreeRTOS,
  - Multitasking *Blinky* example,
  - Activity synchronization with binary semaphores.
- extend the project described in subsection 4.2 in a way that a binary semaphore is used for synchronizing access to global variable `buttonPressed`.

A Including FreeRTOS sources to STM32 project

The following text will describe how to manually integrate FreeRTOS into an existing STM32 project, configure all the required settings and establish a working FreeRTOS project.

The source tree of your working Blinky example should look like this:

```
./
    Core/
    Drivers/
    Lab_GPIO.ioc
    Makefile
    startup_stm32f407xx.s
    STM32F407VTx_FLASH.ld
```

Download the FreeRTOS distribution from FreeRTOS webpage and extract the files from the archive to some location. Read the documents *FreeRTOS - FreeRTOS Source Code Directory Structure* [5] (description of FreeRTOS source code organization) and *FreeRTOS - Creating a new RTOS project* [1] (description of the minimum set of files that must be included in target project). We shall use only the kernel distribution, without additional FreeRTOS+ components.

The demo projects for variuos platforms and development kits are located in the `FreeRTOSDemo` folder. There is no ready to use official port for STM32F4 Discovery board, but the closest match would be in the demo project `FreeRTOS/Demo/CORTEX_M4F_STM32F407ZG-SK`.

Make a new empty folder named FreeRTOS in the source tree of the Blinky project:

```
./
    Core/
    Drivers/
    FreeRTOS/
    Lab_GPIO.ioc
    Makefile
    startup_stm32f407xx.s
    STM32F407VTx_FLASH.ld
```

Copy the content of `FreeRTOS/Source` and `FreeRTOS/Source/include` in your project `FreeRTOS` folder. Also make two empty folders:

– `FreeRTOS/portable/ARM_CM4F`

– `FreeRTOS/portable/MemMang`

Copy the content from FreeRTOS distribution source tree folder `FreeRTOS/Source/portable/GCC/ARM_CM4F` to project folder `FreeRTOS/portable/ARM_CM4F`.

Copy file `heap_1.c` from FreeRTOS distribution source tree folder `FreeRTOS/Source/portable/MemMang` to project folder `FreeRTOS/portable/MemMang`. In this example we shall use the simplest memory management algorithm *Heap1* implemented in `heap_1.c`.

Your source tree should now look like this:

```
./
    Core/
    Drivers/
    FreeRTOS/
      event_groups.c
      list.c
      queue.c
      tasks.c
      timers.c
      include/
        croutine.h
        event_groups.h
        FreeRTOS.h
        FreeRTOSConfig.h
        list.h
        mpu_prototypes.h
        ...
      portable/
        ARM_CM4F/
          portmacro.h
          port.c
        MemMang/
          heap_1.c
    Lab_GPIO.ioc
    Makefile
    startup_stm32f407xx.s
    STM32F407VTx_FLASH.ld
```

The file marked in red, `FreeRTOSConfig.h` requires special attention. This file configures local FreeRTOS project build and it is not included in FreeRTOS distribution `Source/include` folder but in `Demo/xx/` folder (in this case CORTEX M4F STM32F407ZG-SK).

This file must be copied to include folder of the project build and `#defines` should be set as explained later in text.

At this point we have all necessary files included in our Blinky project source tree. However, these files are not included in the project build and the next step is to make necessary adjustments of the project environment in order to properly build the project with all needed FreeRTOS source files.

If you are using STM32CubeIDE, go to *Project - Properties - C/C++ General - Paths and symbols.* Add the folders which contain FreeRTOS headers to *Include* list. Then go to *Source Location* and add the folders which contain FreeRTOS source files.

If you are using *GNU Make* environment without the IDE, make the following adjustments in the *Makefile*:

```
...
C_INCLUDES =  \
-ICore/Inc \
-IDrivers/STM32F4xx_HAL_Driver/Inc \
-IDrivers/STM32F4xx_HAL_Driver/Inc/Legacy \
-IDrivers/CMSIS/Device/ST/STM32F4xx/Include \
-IDrivers/CMSIS/Include \
-IFreeRTOS/include \
-IFreeRTOS/portable/ARM_CM4F

...

C_SOURCES =  \
Core/Src/main.c \
Core/Src/stm32f4xx_it.c \
Core/Src/stm32f4xx_hal_msp.c \
...
FreeRTOS/event_groups.c \
FreeRTOS/list.c \
FreeRTOS/queue.c \
FreeRTOS/tasks.c \
FreeRTOS/timers.c \
FreeRTOS/portable/MemMang/heap_1.c \
FreeRTOS/portable/ARM_CM4F/port.c
```

At this point we have included all source files and headers in the project. The project now should successfully compile, but some compile-time errors can occur.

For example, if the following compiler error occurs:

```
FreeRTOS/include/FreeRTOSConfig.h:53:33: error:
'SystemCoreClock' undeclared (first use in this function)
```

it means that the compiler cannot find a definition for `SystemCoreClock`. This variable is defined in the file `Core/Src/system_stm32f4xx.c`. We will fix this error by declaring the `SystemCoreClock` with extern keyword in `FreeRTOSConfig.h`:

```
extern uint32_t SystemCoreClock
```

Next, after all C sources are properly compiled, the following linker error may occur:

```
FreeRTOS/tasks.o: In function ‘xTaskIncrementTick’:
./FreeRTOS/tasks.c:2798: undefined reference to ‘vApplicationTickHook’
./FreeRTOS/tasks.c:2815: undefined reference to ‘vApplicationTickHook’
FreeRTOS/tasks.o: In function ‘vTaskSwitchContext’:
./FreeRTOS/tasks.c:2988: undefined reference to ‘vApplicationStackOverflowHook’
FreeRTOS/tasks.o: In function ‘prvIdleTask’:
./FreeRTOS/tasks.c:3394: undefined reference to ‘vApplicationIdleHook’
FreeRTOS/portable/heap_1.o: In function ‘pvPortMalloc’:
./FreeRTOS/portable/heap_1.c:112: undefined reference to ‘vApplicationMallocFailedHook’
```

The reason for that are the following defines in `FreeRTOSConfig.h`:

```
#define configUSE_IDLE_HOOK             1
#define configUSE_TICK_HOOK             1
#define configCHECK_FOR_STACK_OVERFLOW  2
#define configMALLOC_FAILED_HOOK        1
```

that require the idle and tick hooks (with exact function names `vApplicationIdleHook()` and `vApplicationTickHook()`) to be implemented somewhere. Also, `heap_1.c` requires function hook `vApplicationMallocFailedHook()` for malloc fail to be defined and the same applies for `vApplicationStackOverflowHook()` that is executed when stack error occurs. If these functions are not used, they can be simply excluded from the build by putting zero value at appropriate defines:

```
#define configUSE_IDLE_HOOK             0
#define configUSE_TICK_HOOK             0
#define configCHECK_FOR_STACK_OVERFLOW  0
#define configMALLOC_FAILED_HOOK        0
```

The linker will also report multiple definition errors for `SysTick_Handler()`, `PendSV_Handler()` and `SVC_Handler()`. These functions are defined by default in the `Core/Src/stm32f4xx_it.c` file. If we take a look at `FreeRTOSConfig.h` file, we will notice that the FreeRTOS interrupt handlers are mapped to the standard CMSIS handlers:

```
#define vPortSVCHandler SVC_Handler
#define xPortPendSVHandler PendSV_Handler
#define xPortSysTickHandler SysTick_Handler
```

Since the standard CMSIS handlers are already defined in the `Core/Src/stm32f4xx_it.c` file, this mapping will cause a multiple definition of the handlers. Take a look at the definitions of these handlers in `Core/Src/stm32f4xx_it.c`. Since the handlers for `SVC_Handler()` and `PendSV_Handler()` are not implemented, we can remove the definitions from `Core/Src/stm32f4xx_it.c` file and remove the corresponding `#defines` from `FreeRTOSConfig.h`. The `SysTick_Handler()` function contains a call to `HAL_IncTick()` and therefore we cannot remove the handler. We can solve this problem by adding a call to `xPortSysTickHandler()` to the `SysTick_Handler()` function:

```
void SysTick_Handler(void)
{
    HAL_IncTick();
    xPortSysTickHandler();
}
```

Now the mapping from `FreeRTOSConfig.h` file can be removed. We also need to add a declaration of `xPortSysTickHandler()` to `Core/Inc/stm32f4xx_it.h`:

```c
void xPortSysTickHandler(void);
```

Now the project should build without compile and linker errors and the project is ready for adding the FreeRTOS functionality.

References

[1] *Creating a New FreeRTOS Project.* URL: `https://www.freertos.org/Creating-a-new-FreeRTOS-project.html`.

[2] *FreeRTOS Configuration Header.* URL: `https://www.freertos.org/a00110.html`.

[3] *FreeRTOS Kernel Developer Docs.* URL: `https://www.freertos.org/features.html`.

[4] *FreeRTOS Reference Manual.* URL: `https://www.freertos.org/fr-content-src/uploads/2018/07/FreeRTOS_Reference_Manual_V10.0.0.pdf`.

[5] *FreeRTOS Source Code Directory Structure.* URL: `https://www.freertos.org/a00017.html`.

[6] *Mastering the FreeRTOS Real Time Kernel-A Hands-On Tutorial Guide.* URL: `https://www.freertos.org/fr-content-src/uploads/2018/07/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf`.

[7] *UM1472 - Discovery kit for STM32F407.* URL: `https://www.st.com/resource/en/data_brief/stm32f4discovery.pdf`.