

Lab 6: Embedded Applications using FreeRTOS

I. Objectives

1. Getting familiar with implementing embedded software using RTOS for STM32 microcontrollers
2. Getting familiar with FreeRTOS Tasks and scheduling
3. Getting familiar with FreeRTOS communication, resource sharing and synchronization services

II. Introduction

An Embedded Operating System like FreeRTOS is nothing but software that provides multitasking facilities. The FreeRTOS kernel is ideally suited to deeply embedded real-time applications that use microcontrollers or small microprocessors. This type of application normally includes a mix of both hard and soft real-time requirements. FreeRTOS allows to run multiple tasks and has a simple scheduler to switch between tasks. Here are some of the FreeRTOS services:

- Priority-based multitasking capability
- Queues to communicate between multiple tasks
- Semaphores to manage resource sharing and synchronization between multiple tasks

II.1 Tasks

Designing an embedded application with RTOS involves dividing the application into set of concurrent tasks (threads) and assigning them priorities. A task is a small piece of code (a C function) which competes with other tasks to get the CPU resources. Every task has an entry point and runs continuously within an infinite loop. **A freeRTOS task will not have an exit point but can be deleted once its job is done.**

A task has its own private stack memory and shares the heap memory with other tasks. A task can be in one of four different states

- **Running:** The task which is executing currently is said to be in running state. It owns the CPU.
- **Ready:** The task which is neither suspended nor blocked but still not executing will be in ready state. It's not in running state because either a high priority or equal priority task is executing.
- **Blocked:** A task will go in blocked state whenever it is waiting for an event to happen. The event can be completing a delay period or availability of a resource. The blocked tasks are not available for scheduling.
- **Suspended:** When `vTaskSuspend()` is called, the task goes in suspended state. It can be resumed by calling `vTaskResume()`. The suspended tasks are also not available for scheduling.

Here is a simple task and a small program that uses it. Note that `vTaskStartScheduler()` never returns and FreeRTOS will begin servicing the tasks at this point. Also, note that before you create a task, you need to know what stack memory is. The stack size of a task depends on the memory consumed by its local variables and function call depth.

```
void hello_world_task(void* p)
{
    while(1) {
        puts("Hello World!");
        vTaskDelay(1000);
    }
}
```

```

int main() {
    xTaskCreate(hello_world_task, (signed char*)"task_name", STACK_BYTES(2048), 0, 1, 0);
    vTaskStartScheduler();
    return -1;
}

```

`vTaskDelay()` is used instead of busy waiting loop for implementing a delay. `vTaskDelay()` is actually smart enough to put the task to sleep and wake it up precisely when the timeout is done. `vTaskDelay()` will delay by the defined amount, and if you wanted a precise periodic processing of one second, this function might not work as you'd expect. In the example above, even if you wanted the message to be displayed once per 1000ms, your actual rate would be anywhere from 1005 to 1050ms because `puts()` function's processing time can vary. In this example, if we switch over to `vTaskDelayUntil()`, we will be updating sensors exactly once per 1000ms. More can be found @ <https://www.freertos.org/vtaskdelayuntil.html>

II.2 Interprocess Communication using Message Queues

You can communicate between tasks by using Queues. The code below demonstrates how to do that.

```

QueueHandle_t qh = 0;      // Global Queue Handle

void task_tx(void* p)
{
    int myInt = 0;
    while(1)
    {
        myInt++;
        if( xQueueSendToBack(qh, &myInt, 500) != pdPASS ) {
            puts("Failed to send item to queue within 500ms");
        }
        vTaskDelay(1000);
    }
}

void task_rx(void* p)
{
    int myInt = 0;
    while(1)
    {
        if( xQueueReceive(qh, &myInt, 1000) != pdPASS ) {
            puts("Failed to receive item within 1000 ms");
        }
        else {
            printf("Received: %u\n", myInt);
        }
    }
}

int main()
{
    qh = xQueueCreate(1, sizeof(int));
    xTaskCreate(task_tx, (signed char*)"t1", STACK_BYTES(2048), 0, 1, 0);
    xTaskCreate(task_rx, (signed char*)"t2", STACK_BYTES(2048), 0, 1, 0);
    vTaskStartScheduler();
    return -1;
}

```

Interrupt handlers are pieces of code run by the microcontroller and therefore are not handled by FreeRTOS. This can potentially create problems with memory access since the operating system cannot handle these context changes. This is a reason why several functions exists in two versions: one for regular tasks and another is intended to interrupts handler. This is the case of queue management functions like `xQueueReceive()` and `xQueueReceiveFromISR()`. For this reason, it is necessary to make interrupts handlers' execution as short as possible.

II.3 Semaphores and Critical Sections

Semaphores are meant to limit access to shared resources and to synchronize between tasks. There are several types of semaphores:

Critical Section

A Critical Section is a mutual exclusion technique, used to ensure data consistency is maintained at all times when accessing a shared resource between tasks, or between tasks and interrupts.

The goal is to ensure that, once a task starts to access a shared resource, the same task has exclusive access to the resource until the resource has been returned to a consistent state.

A task can start a critical section with `taskENTER_CRITICAL()` and stop it using `taskEXIT_CRITICAL()`

```
void vTask1( void * pvParameters )
{
    for( ; ; )
    {
        /* Perform some functionality here. */
        /* Call taskENTER_CRITICAL() to create a critical section. */
        taskENTER_CRITICAL();

        /* Execute the code that requires the critical section here. */

        /* exit the critical section */
        taskEXIT_CRITICAL();
    }
}
```

`taskENTER_CRITICAL()` and `taskEXIT_CRITICAL()` must not be called from an interrupt service routine. `taskENTER_CRITICAL_FROM_ISR()` is an interrupt version of `taskENTER_CRITICAL()`, and `taskEXIT_CRITICAL_FROM_ISR()` is an interrupt version of `taskEXIT_CRITICAL()`.

Mutex

A mutex is used to guard a resource (e.g., I2C bus which can be used only by one task at a time). Mutex provides mutual exclusion with priority inversion mechanism. Mutex will only allow one task to get past `xSemaphoreTake()` operation and other tasks will be blocked if they call this function at the same time.

```
// In main(), initialize your Mutex:
SemaphoreHandle_t spi_bus_lock = xSemaphoreCreateMutex();

void task_one()
{
    while(1) {
        if(xSemaphoreTake(spi_bus_lock, 1000)) {
            // Use Guarded Resource
        }
    }
}
```

```

        // Give Semaphore back:
        xSemaphoreGive(spi_bus_lock);
    }
}

void task_two()
{
    while(1) {
        if(xSemaphoreTake(spi_bus_lock, 1000)) {
            // Use Guarded Resource

            // Give Semaphore back:
            xSemaphoreGive(spi_bus_lock);
        }
    }
}

```

In the code above, only ONE task will enter its `xSemaphoreTake()` branch. If both tasks execute the statement at the same time, one will get the mutex, the other task will sleep until the mutex is returned by the task that was able to obtain it in the first place.

Binary Semaphore

Binary semaphore can also be used like a mutex, but binary semaphore doesn't provide priority inversion mechanism and also can be normally discarded and not returned. Binary semaphores are better suited for helper tasks for interrupts. For example, if you have an interrupt and you don't want to do a lot of processing inside the interrupt, you can use a helper task. To accomplish this, you can perform a semaphore give operation inside the interrupt, and a dedicated task will sleep or block on `xSemaphoreTake()` operation.

```

// Somewhere in main() :
SemaphoreHandle_t event_signal;
vSemaphoreCreateBinary( event_signal ); // Create the semaphore
xSemaphoreTake(event_signal, 0);      // Take semaphore after creating it.

void System_Interrupt()
{
    xSemaphoreGiveFromISR(event_signal, NULL);
}

void system_interrupt_task()
{
    while(1) {
        if(xSemaphoreTake(event_signal, 9999999)) {
            // Process the interrupt
        }
    }
}

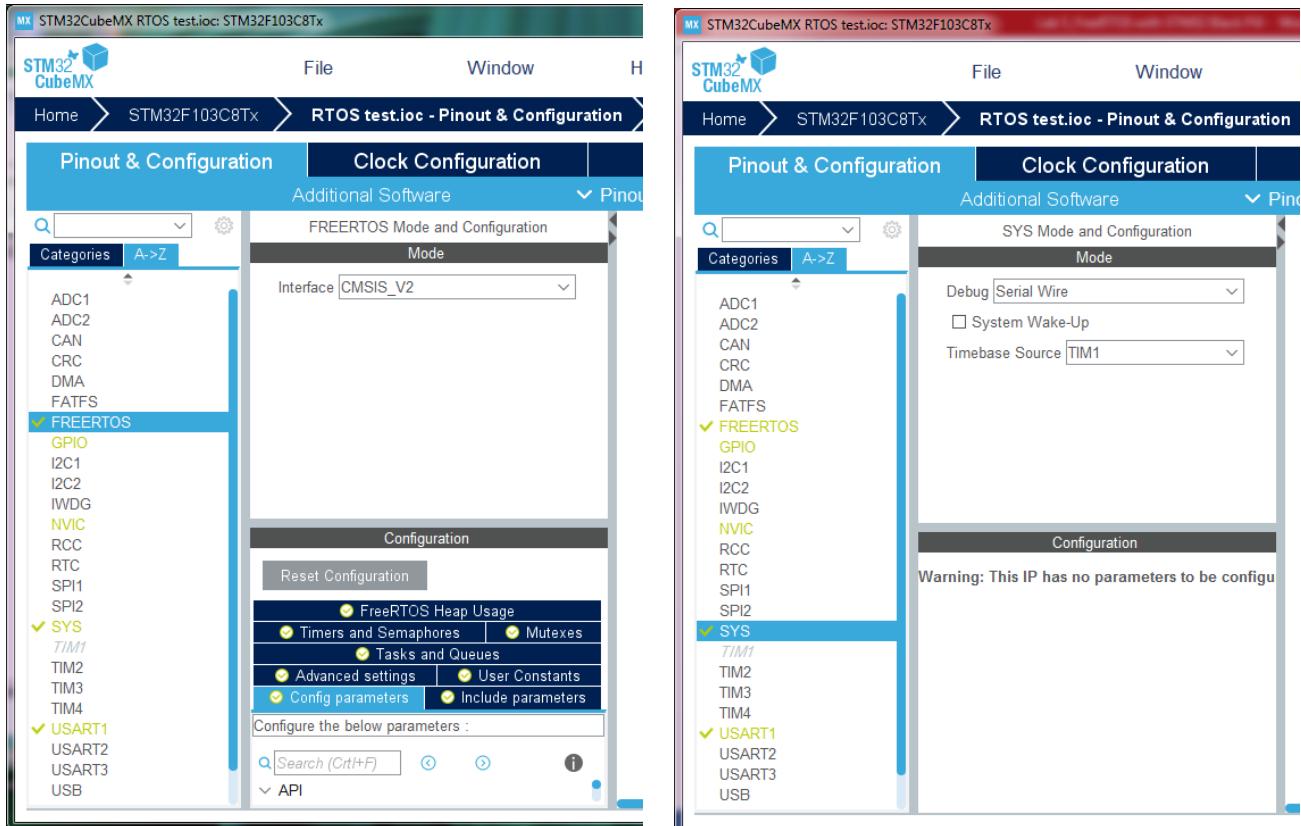
```

Another way to use binary semaphore is to wake up one task from another task by giving the semaphore. The semaphore will essentially act like a signal sent from one task to another (synchronization).

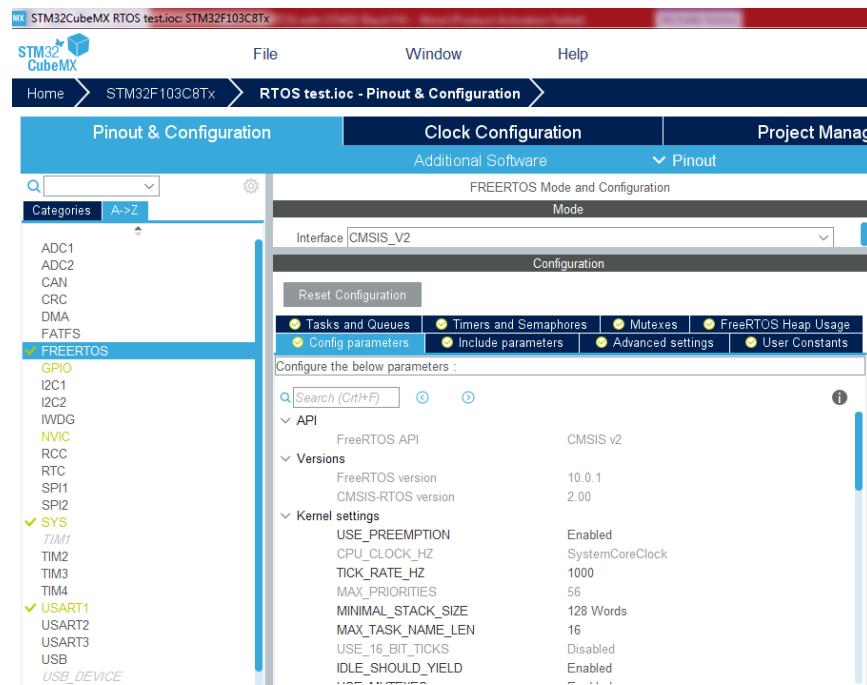
The function `xSemaphoreGiveFromISR()` is the variant of `xSemaphoreGive()` that must be used within an interrupt handler.

II.4 FreeRTOS with CubeMX

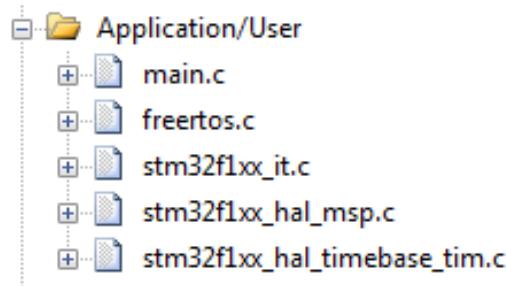
After creating a new project using CubeMX as we did in Experiment 5, enable FreeRTOS and TIM1 as a time base for HAL as shown by the following screenshots.



FreeRTOS configuration dialog can be used to configure FreeRTOS as well as creating tasks, queues, semaphores, ... as shown below:



Once, the project is created and opened by Keil µVision, you may edit the `main.c` file to implement your application. The ISRs for the peripherals enabled IRQs can be found in `stm3214xx_it.c` file (don't forget to configure the NVIC while creating the project by CubeMX).



III. References:

- FreeRTOS Tutorial: http://socialledge.com/sjsu/index.php/FreeRTOS_Tutorial
- FreeRTOS APIs: <https://www.freertos.org/a00106.html>
- FreeRTOS_Reference_Manual_V10.0.0.pdf
- STM UM1722: Developing RTOS Applications on STM32Cube using CMSIS-RTOS APIs.pdf

IV. Experiment:

Create an application that sends and receives messages to/from a PC using USB-UART bridge module/cable on UART. The application gets a simple expression (digit +/- digit\r) from the PC (typed by the user using a terminal emulator), evaluates it then sends the results back to the PC. The application makes use of the following:

- i. **UART ISR:** receives characters from the PC using UART interrupt handler and places them into a queue (*RQ*).
- ii. **Task 1 (let's call it Task1_Evaluate):** Reads characters from *RQ* queue, recognizes the expression and evaluates it. The result is placed into a queue (*SQ*)
- iii. **Task 2 (let's call it Task2_UARTSend):** Reads results from *SQ* queue and sends them to the PC using UART.

Work through your code step by step along with running and debugging following every step. Start by UART ISR, then Task 1, then finally Task 2.

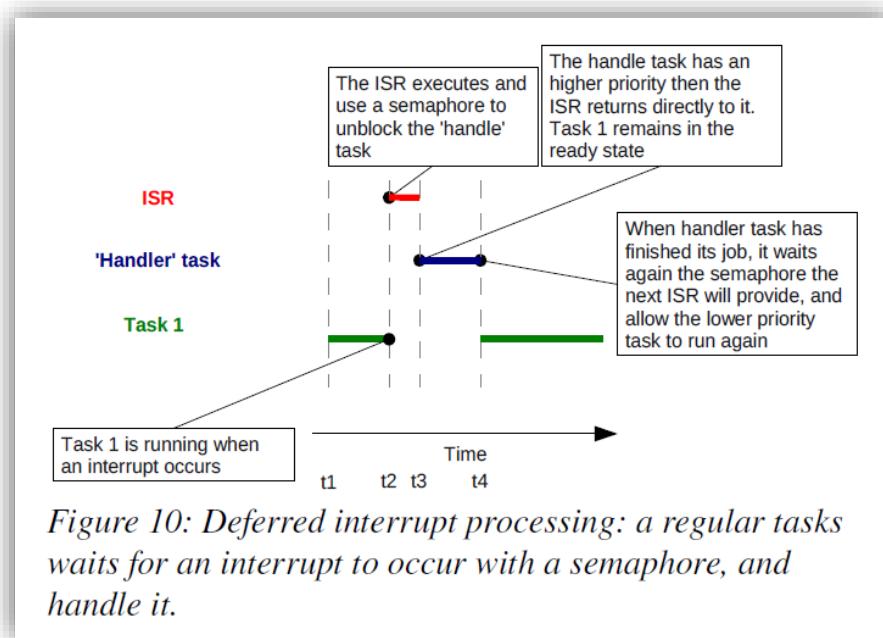
Notes:

- For UART communication, only use the micro USB interface using UART2 if you tested it before and you know its RX is working. Otherwise please use the USB-To-TTL module with its TX,RX,Gnd connected to the nucleo board pins.
- Always power up the nucleo board from the PC using the USB cable. So you should have two working USB slots on your PC, one to connect the nucleo (for power up and program download and debugging), and one to connect the USB-To-TTL (for TX and RX to Teraterm).
- On CubeMX under UART configuration, enable the UART global interrupt with a preemption priority 5 or higher (Preemption priority of interrupts should be >=5 if their handlers call system functions)

- The UART is a **shared resource between UART ISR and Task 2**, so it must be protected.
- The function call `HAL_UART_Receive_IT(&huart2, &rxdata, 1);` should be added in `main()` to enable UART interrupts and start receiving data via interrupt, and it must be placed before `osKernelStart()` function.
- For the UART handler use `HAL_UART_RxCpltCallback` function to process the received data same as was done in Lab 4 Exp 4.

Lab Report [10 pts] (Individual submission)

1. [2 pts] Provide your C code of the experiment conducted in the lab along with screenshots of at least 3 mathematical operations on TeraTerm.
2. [4 pts] Modify the lab experiment to create a new task, let's call it **Task3_UARTReceive**. UART ISR should be as short as possible and not intended for long task executions. So you should only let it unblock the new Handler task **Task3_UARTReceive** that is able to handle the interrupt routine intended tasks. This requires using a **binary semaphore to synchronize tasks**.



Note: For using the binary semaphore, the UART interrupt handling needs to be disabled before giving the semaphore, then re-enabled at the end.

Provide your code along with a small sized video of the running application.

3. [4 pts] Develop an application that sets the DS3231 RTC clock and continuously displays the current time and room temperature over the UART interface using FreeRTOS. The OS should run two concurrent tasks. Task1 continuously displays the current time over the UART interface and task2 continuously displays the room temperature over the same interface.
Provide your code along with a small sized video of the running application.