Processor Architecture and Technology
2019

Laboratory 2
Real-Time Operating System on Microcontrollers

Lecturer: MSc. Fabio Arnez

March 13, 2019

Contents

List of Figures

List of Tables

1 Introduction

1.1 The need of a Real-Time Kernel

There are many well established techniques for writing good embedded software without the use of a kernel, and, if the system being developed is simple, then these techniques might provide the most appropriate solution. In more complex cases, it is likely that using a kernel would be preferable, but where the crossover point occurs will always be subjective. As already described, task prioritization can help ensure an application meets its processing deadlines, but a kernel can bring other less obvious benefits, too. Some of these are listed very briefly below:

- **Abstracting away timing information**
  Using a kernel allows software to be completely event-driven, so no processing time is wasted by polling for events that have not occurred. Code executes only when there is something that must be done.
- **Maintainability/Extensibility** Abstracting away timing details results in fewer interdependencies between modules and allows the software to evolve in a controlled and predictable way. Also, the kernel is responsible for timing, so application performance is less susceptible to changes in the underlying hardware.
- **Modularity** Tasks are independent modules, each of which should have a well-defined purpose.
- **Team development** Tasks should also have well-defined interfaces, allowing easier development by teams.
- **Easier testing** If tasks are well-defined independent modules with clean interfaces, they can be tested in isolation.
- **Code reuse** Greater modularity and fewer interdependencies can result in code that can be re-used with less effort.
- **Improved efficiency** Using a kernel allows software to be completely event-driven, so no processing time is wasted by polling for events that have not occurred. Code executes only when there is something that must be done.
- **Idle time** The Idle task is created automatically when the kernel is started. It executes whenever there are no application tasks wishing to execute. The idle task can be used to measure spare processing capacity, to perform background checks, or simply to place the processor into a low-power mode.
- **Flexible interrupt handling** Interrupt handlers can be kept very short by deferring most of the required processing to handler tasks. Section 3.2 demonstrates this technique.
- **Mixed processing requirements** Simple design patterns can achieve a mix of periodic, continuous, and event-driven processing within an application. In addition, hard and soft real-time requirements can be met by selecting appropriate task and interrupt priorities.
- **Easier control over peripherals** Gatekeeper tasks can be used to serialize access to peripherals.

Figure 1 show sthe CMSIS Organization for RTOS support.

1.2 Multitasking on Small Embedded Systems

Microcontrollers (MCUs) that contain an ARM Cortex-M3 core are available from many manufacturers and are ideally suited to deeply embedded real-time applications. Typically, applications of this type include a mix of both hard and soft real-time requirements.

Soft real-time requirements are those that state a time deadline—but breaching the deadline would not render the system useless. For example, responding to keystrokes too slowly may make a system seem annoyingly unresponsive without actually making it unusable.

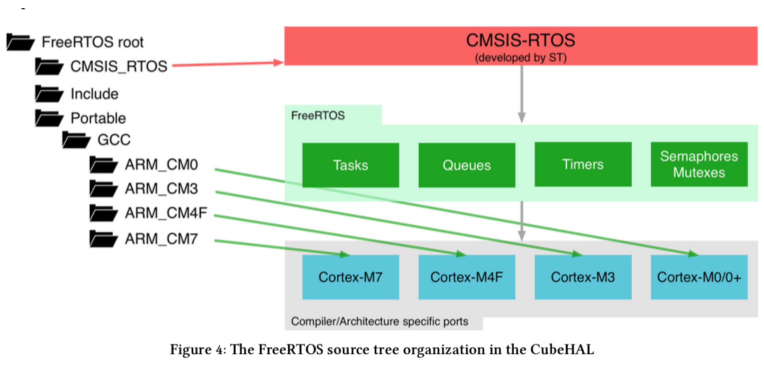


Figure 1: CMSIS RTOS Organization

Hard real-time requirements are those that state a time deadline—and breaching the deadline would result in absolute failure of the system. For example, a driver's airbag would be useless if it responded to crash sensor inputs too slowly.

FreeRTOS is a real-time kernel (or real-time scheduler) on top of which Cortex-M3 microcontroller applications can be built to meet their hard real-time requirements. It allows Cortex-M3 microcontroller applications to be organized as a collection of independent threads of execution. As most Cortex-M3 microcontroller have only one core, in reality only a single thread can be executing at any one time.

The kernel decides which thread should be executing by examining the priority assigned to each thread by the application designer. In the simplest case, the application designer could assign higher priorities to threads that implement hard real-time requirements, and lower priorities to threads that implement soft real-time requirements. This would ensure that hard real-time threads are always executed ahead of soft real-time threads, but priority assignment decisions are not always that simplistic. On the other hand, is important to note that each task is represented by a memory segment containing the Thread Control Block (TCB), which is nothing more than a descriptor containing all relevant information related to the task execution just "a moment"9 before it is preempted (the stack pointer, the program counter, CPU registers and other few things), plus the stack itself, that is activation records of those routines currently invoked on the thread stack. Figure 2 shows the RTOS task execution sequence and Figure 3 shows OS Memory Organization for Tasks.

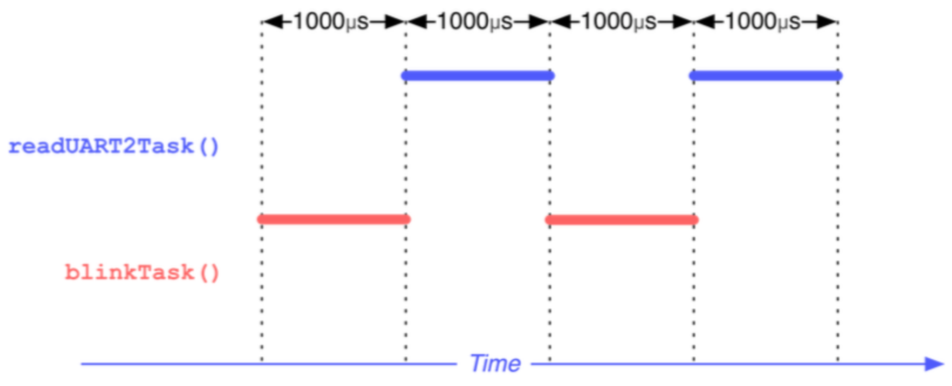


Figure 2: RTOS Task Execution

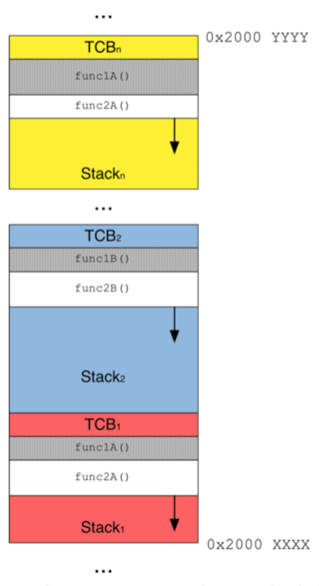


Figure 3: OS Memory Organization for Tasks

1.3 The ARM Cortex-MX Port of FreeRTOS

The Cortex-M3 port includes all the standard FreeRTOS features:

- Pre-emptive or co-operative operation
- Very flexible task priority operation
- Queues
- Binary semaphores
- Counting semaphores
- Recursive semaphores
- Mutexes
- Tick and Idle hook functions
- Stack overflow checking
- Trace hook macros
- Optional commercial licensing and support

2 Lab Materials

The following material list is required for the current laboratory session:

- STM32 Nucleo Board or Discovery Board
- STM32 toolchain and development environments
- CubeMX importer
- STM32CubeMX
- LEDs, resistances, prototyping board

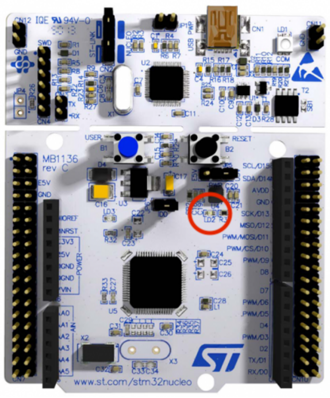


Figure 4: ARM Cortex ST Nucleo-F401RE Board: LED2 Location

3 Exercise 0: Blink 4 LEDs at Different Frequencies

The goal of this exercise is to blink four LEDs using the knowledge acquired in the previous lab session. Figure 5 shows a wiring diagram example for this task

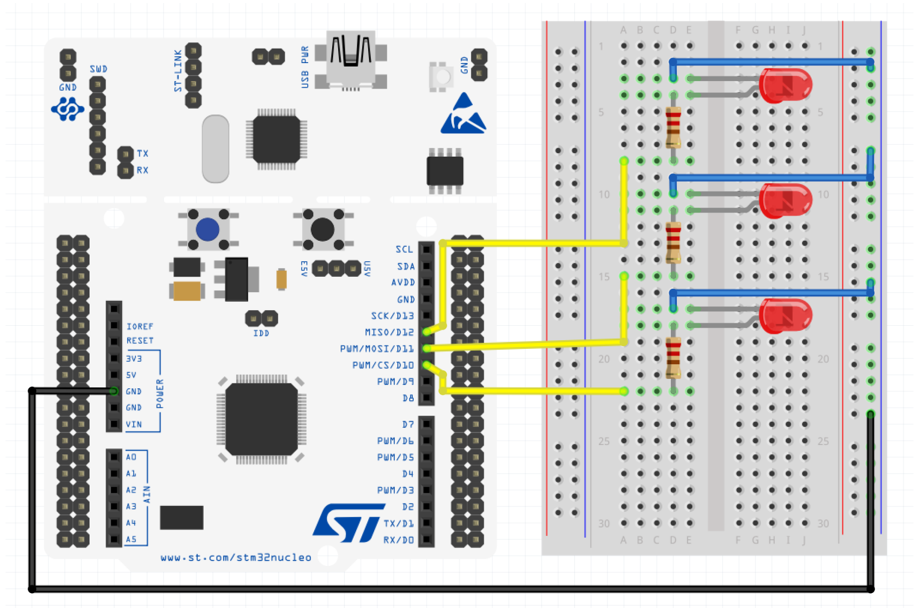


Figure 5: Task 0 Wiring Diagram Example

3.1 Exercise Tasks

- Implement the code using your current knowledge
- Describe the approach used and the code implemented
- Describe which will be the major problems if doing several task with different processing times and priorities using the current approach (e.g. system using: GPS frames reception from UART interface, IMU data reception through I2C interface, GUI using a Touchscreen for plotting location and IMU data and TCP/IP Stack for data communication to a cloud)

4 Exercise 1: Blink 4 LEDs with RTOS

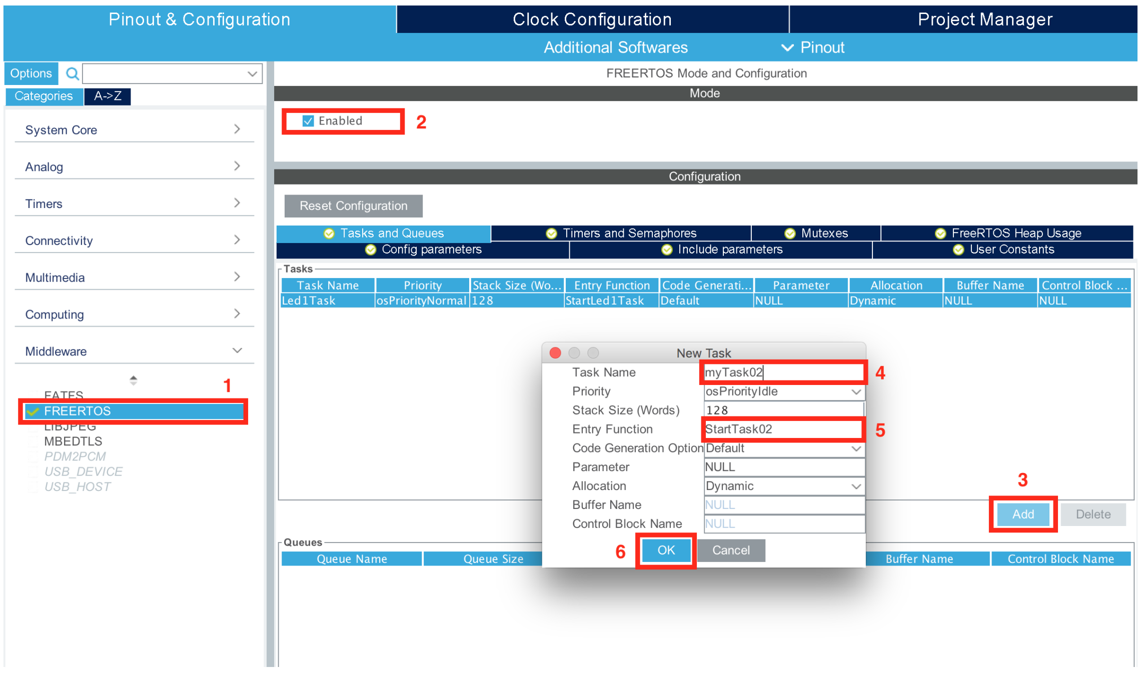
The goal of this exercise is to blink four LEDs using a Real-Time Operating System. In order to use a RTOS, we need include FreeRTOS during project configuration in STM32CubeMX.

4.1 STM32CubeMX and Eclipse Configuration for RTOS Support

To configure the project, please refer to the video on Section 6 for STM32CubeMX configuration. If using board with ARM Cortex-M4 or Cortex-M7 cores, please refer to ***Section 23.2.1.3 How to Enable FPU Support in Cortex-M4F and Cortex-M7 Cores*** from ***Mastering STM32*** book.

A summary of the steps is described below:

1. Create a new STM32CubeMX project
2. Add FreeRTOS to the project as described below in Figure 6



Tasks

Task Name	Priority	Stack Size (Wo...	Entry Function	Code Generati...	Parameter	Allocation	Buffer Name	Control Block ...
Led1Task	osPriorityNormal	128	StartLed1Task	Default	NULL	Dynamic	NULL	NULL

New Task

Task Name	myTask02
Priority	osPriorityIdle
Stack Size (Words)	128
Entry Function	StartTask02
Code Generation Option	Default
Parameter	NULL
Allocation	Dynamic
Buffer Name	NULL
Control Block Name	NULL

Figure 6: STM32CubeMX Project Configuration for FreeRTOS Support

3. Add tasks, queues, timers, semaphores, mutexes at your will
4. If using board with ARM Cortex-M4 or Cortex-M7 core, you need to **enable** the **FPU** support by doing the following modifications in the Eclipse Project:
   Project Settings->C/C++ Build->Settings->Target Processor section and select the entry **FP instructions (hard)** in the **Float ABI** field, and for the **FPU Type** field select **fpv4-sp-d16** if you have a Cortex-M4F based STM32 MCU. Finally rebuild the project.

4.2 Exercise Tasks

- Toggle 4 LEDS, each one at one of the following periods: $T = \{500, 1000, 1500, 2000\}$ $[ms]$ using a FreeRTOS Real-Time Operating System
- Set-up a project for both *STM32CubeMX* and *Eclipse*, to include *FreeRTOS* support.
- Implement 4 threads/tasks, one task/thread for each lead toggling at a specific period.
- Explain your code, the RTOS functions used and the relationship (in terms of function calls) from *CMSIS-RTOS* abstraction layer and *FreeRTOS* layer.

4.3 Useful Code

```
/* USER CODE BEGIN Header_StartTask2 */
/**
* @brief Function implementing the TaskX thread.
* @param argument: Not used
* @retval None
*/
/* USER CODE END Header_StartTaskX */
void StartTaskX(void const * argument)
{
    /* USER CODE BEGIN StartTaskX */
    /* Infinite loop */
    for(;;)
    {
        //Write Thread/Task Code HERE!
    }
/* USER CODE END StartTaskX */
}
```

Listing 1: Task/Thread Function

```
/* @brief   Wait for Timeout (Time Delay)
* @param   millisec      time delay value
* @retval  status code that indicates the execution status of the function.
*/
osStatus osDelay (uint32_t millisec)
```

Listing 2: Millisecond Delay Function

5 Exercise 2: Resource Sharing Between Tasks/Threads

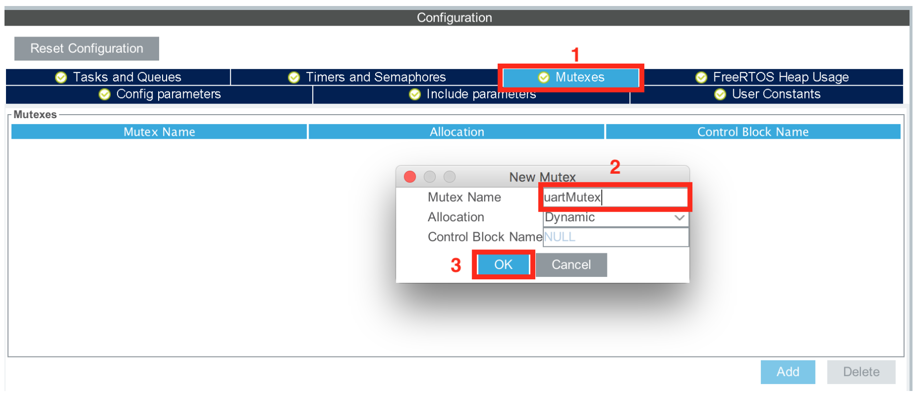
The goal of this exercise is to share a resource (in this case a UART interface) between two tasks/treads to print messages in console terminal in your PC (PuTTY, CoolTerm, Arduino Console, etc.). In order to share the UART interface efficiently, you must add a **Mutex** while in the RTOS configuration, in the STM32CubeMX Project.

5.1 STM32CubeMX and Eclipse Configuration for RTOS Support

To configure the project, please refer to the video on Section 6 for STM32CubeMX configuration. If using board with ARM Cortex-M4 or Cortex-M7 cores, please refer to ***Section 23.2.1.3 How to Enable FPU Support in Cortex-M4F and Cortex-M7 Cores*** from ***Mastering STM32*** book.

A summary of the steps is described below:

1. Add FreeRTOS Mutex to the project as described below in Figure 7



Mutex Name: uartMutex
Allocation: Dynamic
Control Block Name: NULL

Figure 7: STM32CubeMX Project FreeRTOS Configuration: Add Mutex

2. Add mutexes at your will according to the number of shared resources.

3. If using board with ARM Cortex-M4 or Cortex-M7 core, you need to **enable** the **FPU** support by doing the following modifications in the Eclipse Project:
Project Settings->C/C++ Build->Settings->Target Processor section and select the entry **FP instructions (hard)** in the **Float ABI** field, and for the **FPU Type** field select **fpv4-sp-d16** if you have a Cortex-M4F based STM32 MCU. Finally rebuild the project.

5.2 Exercise Tasks

- Create two tasks/Threads.
- Each tasks should Initially the same time delay (e.g. 1000 ms)
- Print the following message from both tasks:
"Hello from task X!", where X is the task number
- Explain the problem in this scenario.
- Add a Mutex to your project and to fix the previous problem and describe the new program execution

6 Useful Links

- STM32-Nucleo - Keil 5 IDE with CubeMX: Tutorial 9 - FreeRTOS