BoardOS - Project Group Report

1 IDE and Project Configuration

1.1 IDE Installation

The first step to setup the work environment is to install **STM32CubeIDE**, the IDE for the STM32 Nucleo Board, released by the board producer STMicroelectronics. The software supports by all popular operating systems, like Windows, Linux and MacOS, but 64-bit version only. The setup installation can be downloaded from the official site at this link, choosing the right installer for your operating system. After agreeing on the Terms and condition the download starts automatically and a zip file will be obtained that contains an ".exe" file (for Windows).

1.2 Create a new project

After installing and launching the IDE it is possible to create the first Project inside the Stm32CubeIDE. To create a project those steps needs to be followed:

1. Go to *File*, *New* and then select ***STM32 Project***;
2. On Target Selection, click on ***Board Selector*** tab and write down "NUCLEO-F446RE"on *Commercial Part Number* field. Select the board from the list and click ***Next***;
3. Write a **Project Name**, change the **Project Location** if necessary, and click ***Finish***.

The project is now created and it will automatically open the "ioc" file. Now, by default, STM32Cube Ide adds some default configuration like the UART peripherals configuration and the Clock Configuration.

1.3 How to use FreeRTOS microkernel

FreeRTOS is a market-leading real-time operating system (RTOS) for microcontrollers and small microprocessors. In order to use it, it must be activated through the IDE graphical interface.

After creating the project in STM32CubeIDE, automatically the .ioc file opens the "Pinout & Configuration" tab. Inside this tab, by selecting the section **Middleware and Software Packs**, is possible to create a FreeRTOS project by clicking on **FREERTOS**. The opened tab will have, by default, the interface mode disabled and, to configure correctly FreeRTOS, is necessary to switch that field from disabled to CMSIS_V2. Now, the Configuration tab is enabled and it is possible to change different parameters like timers and semaphores, some kernel settings, and other useful parameters, and finally it is be possible to create Tasks and Queues.

2 Resource Sharing and Inter-Thread Communication

In most applications, threads need to communicate with each other or access shared resources together. There are many ways to exchange data between threads, for example using shared data, polling loops and message passing. The *CMSIS-RTOS API* provides various ways for exchanging messages between threads, improving inter-thread communication and resource sharing. The following methods are some of function available to the user:

- **Resource Sharing**: *Binary Semaphore*
- **Inter-Thread Communication**: *Message Queue* and *Mail Queue*

2.1 Binary Semaphore

Semaphores are used to synchronize tasks with other events in the system. Waiting for semaphore is equal to `wait()` procedure, task is in a blocked state not taking CPU time. In FreeRTOS implementation semaphores are based on queue mechanisms.

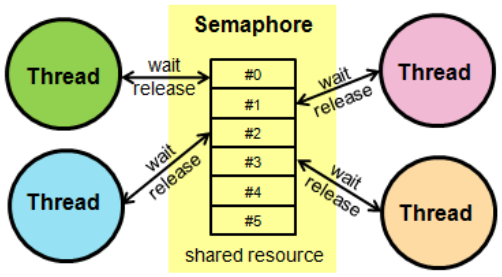

Figure 1: CMSIS-RTOS Semaphore

There are some types of semaphores in FreeRTOS, like binary, counting or mutex. The **Binary Semaphore** is a simple on/off mechanism, that uses this function:

```
osSemaphoreId binarySemaphoreHandle; // Semaphore Declaration
osSemaphoreWait(binarySemaphoreHandle, osWaitForever); // Acquire a token from the semaphore container
osSemaphoreRelease(binarySemaphoreHandle); // Send the token back to the semaphore container
```

With `osSemaphoreWait()` function, we wait until a Semaphore token becomes available.
With `osSemaphoreRelease()` function, we release a Semaphore token from a thread.

2.2 Message Queue

A Message Queue is the basic communication model between threads. The data is passed from one thread to another in a FIFO-like operation. Using Message Queue functions, you can send, receive, or wait for messages.

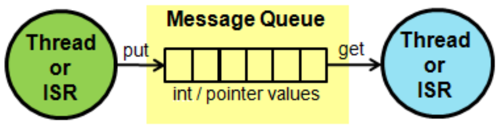

Figure 2: CMSIS-RTOS Message Queue

The queue is made by integers or pointers 32-bit long. The length of queue is declared during creation phase and is defined as a number of items which will be send via queue.

```
osMessageQId         osMessageCreate (const osMessageQDef_t *queue_def, osThreadId thread_id)
osStatus             osMessagePut (osMessageQId queue_id, uint32_t info, uint32_t millisec)
osEvent              osMessageGet (osMessageQId queue_id, uint32_t millisec)
```

Those are the functions provided by CMSIS-RTOS for the Message Queue management. `osMessageCreate` is used to create and initialize a Message Queue, `osMessagePut` put a Message to a Queue and `osMessageGet` get a Message or wait for a Message from a Queue.

2.3 Mail Queue

A Mail Queue operates similarly to a Message Queue, but instead of transferring a 32-bit value, the data that is being transferred consists of memory blocks that need to be allocated, before putting data in, and free-up, after taking data out.

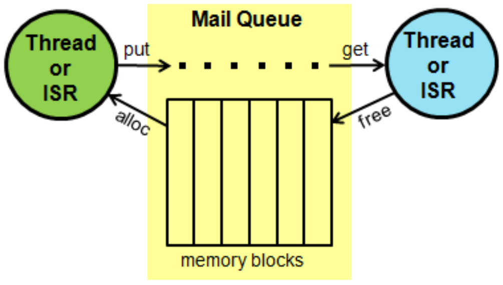

Figure 3: CMSIS-RTOS Mail Queue

The Mail Queue uses a *memory pool* to create formatted and fixed-size memory blocks and passes pointers to these blocks in a Message Queue. This allows the data to stay in an allocated memory block while only a pointer is moved between the separate threads, and it's an advantage to Message Queues because there are no big data transfers.

```
typedef struct{
    uint8_t buffer[256];
} MailStruct;
osMailQDef(mailQueue, 16, MailStruct); // Create a Mail Queue Definition passing
```

This code shows the declaration a Mail data structure used for inter-thread communication, with a buffer array of 256 elements. The `osMailQDef` function is used to define the attributes of a Mail Queue, like the name, the maximum number of messages and the data type of a single message element.

```
MailStruct *p = osMailAlloc(mailHandle, osWaitForever); // Allocate a memory block from a mail
p = &mailData; // Link the pointer to the address of the buffer we want to transfer
osMailPut(mailHandle, p); // Put a mail to a queue
```

In those code lines, extracted from the *Sender Thread*, `osMailAlloc` is used to allocate a memory block from the buffer filled with the mail information we want to transfer, while `osMailPut` put the memory block specified into a Mail Queue.

```
osEvent event = osMailGet(mailHandle, osWaitForever); // Get a mail from a queue
MailStruct *p = (MailStruct*) event.value.p; // p->buffer[i]
osMailFree(mailHandle, p); // Free a memory block from a mail
```

In the *Receiver Thread*, `osMailGet` is used to suspend the execution of the current RUNNING thread until a mail arrives, then returning the mail information. Instead, `osMailFree` free the memory block read by the receiving function.