Lab 2: Introduction to Real-Time System Implementation Using uC/OS-III and FreeRTOS

Matthew Salazar (1233223), Chase Keeler (1236835), Waleed Elshowaya (1235545), Cameron Bauman (1236693)

ENGG*4420

Petros Spachos

1.0 Introduction

The objective of this lab was to implement basic task switching through the use of priorities, stack, queue, and semaphores in the uC/OS-III and FreeRTOS kernels. This producer-consumer program was created in the STM32CubeIDE, which acts as an all-in-one multi-OS development tool using C/C++. The STM32F429I-DISC1 evaluation board comes equipped with a graphical interface used to display custom user interfaces. Connecting the evaluation board to the computer using both connectors allowed for serial communication between the PC and evaluation board, and the ability to download code to the evaluation board. A diagram of this STM32 development board can be seen in Figure 1.

Real-time systems employ functions containing deadlines that must be met. To ensure this functionality works as expected, priorities can be used to execute ready tasks in an order of highest priority first. When a lower priority task is currently running, it monopolizes the processors' resources, by preempting this task, the higher priority task can temporarily execute and resume the lower priority task once it's completed.

Binary semaphores act as a way to understand which tasks are accessing the shared resource. As seen in the FreeRTOS example, tasks will wait until the currently executing task releases the semaphore before it can start it's execution. This allows for synchronization between tasks and ultimately avoids race conditions.

2.0 Background

Applying real-time system design principles discussed in lecture, this lab required students to implement key concepts using two real-time operating systems: uC/OS-III and FreeRTOS. The uC/OS-III portion aligned more closely with lecture material, as the functions and topics were based on the *uC/OS-III User Manual* by Jean J. Labrosse. Before beginning the lab demonstration, students were required to complete tutorials for both operating systems. For uC/OS-III, this involved learning how to define a task, assign it a priority level, and allocate a stack size. To verify functionality, the tutorial required students to output a message through the serial COM port indicating which task (either Task 1 or Task 2) was actively running.

For FreeRTOS, the tutorials introduced more advanced concepts discussed in class, including the implementation of message queues, binary and counting semaphores, and murex's. Similar to the uC/OS-III example, the fundamentals of task initialization were covered, but the exercises were extended so that Task 1 would add a string message to a queue and then temporarily pause itself using osDelay(). From a real-time systems perspective, this delay allows the scheduler to switch to the lower-priority task (Task 2), enabling it to read the queued message. Semaphores act as binary flags or counters to indicate when a specific piece of hardware or bus is unavailable, preventing tasks from accessing data that is currently in use. A mutex is a special type of binary semaphore designed specifically for mutual exclusion, only the thread that locks the resource can unlock it whereas a regular semaphore can be locked and unlocked by any thread or task.

Both FreeRTOS and uC/OS-III build on a microcontroller's existing kernel to add a layer of real-time system control, but there are differences between them. FreeRTOS is completely open source and requires no licensing, whereas uC/OS-III requires a license for commercial use. FreeRTOS is often preferred because

it is lightweight and easier to implement, whereas uC/OS-III is best for mission-critical or safety-certified embedded systems.

3.0 Methodology

In this lab, we set up the development board and utilized the STM32CubeIDE to create and test a real time application. As stated earlier, the main objective of this lab was to understand synchronization mechanisms and task management through implementing a producer-consumer problem.

To start the lab, we imported the ENGG*4420 uC/OS-III and FreeRTOS examples to our project into our IDE workspace. The FreeRTOS template came with fundamental kernal configuration files, necessary header files, task creation routines, and compatibility layer, which provided a consistent interface throughout our lab implementation. This allowed us to effectively define threads, queues, and semaphores.

We created our own producer and consumer tasks. The producer task generates and sends data through a message queue while the consumer tasks receive and processes data from the queue. The producer task must execute data production at a constant time rate and use the function *osMessagePut()* to send the structure pointer to the queue. The timing control is through the *osDelay()* function which executes periodic timing. This delay ensures that the producer task, or task 1, sends data at fixed periodic intervals.

In contrast, the consumer task uses *osMessageGet()* instead of *osMessagePut(),* and waits until new data is available. The blocking call makes sure that the consumer task is idle and efficient until needed, when data is received. These messages are displayed through the terminal window. We labeled these tasks as task 1 and task 2 for simplicity. These tasks are initialized with priorities and a stack size. The producer task is generally given higher priority as the data needed to be generated within a certain timeline. For the system design, we also defined 2 threads as per the lab document. Communication through the system is also handled through declared message queues.

Each of these tasks were initialized with specific priorities and stack sizes specific for this application. In this implementation, the producer task was declared with a higher priority than the consumer task which was given a normal priority. This allowed the producer task to have priority to generate time sensitive messages before timing out and allowing the consumer task to handle data as it received it became available. This demonstrated the fundamental behavior of FreeRTOS's preemptive scheduling to ensure higher priority tasks interrupt lower priority tasks when they are in a ready state, allowing students to strengthen their knowledge of real-time scheduling.

Once this was configured, two threads were defined as well as a message queue to contain both the message data and a source identifier. For this implementation the data being sent between threads was a "Hello" string. To further implement this real-time implementation, additional features were added including semaphores and mutexes. Like described earlier, the semaphore was used to signal one task from the other when resources were available for it to begin its functionality. Similarly, mutexes were used to control access to the shared resources between the two tasks.

To demonstrate the lab, messages from the consumer and producer tasks are alternated on the terminal to display successful communication and synchronization through the system. This was achieved by configuring the virtual COM port and setting up Tera Term with the correct baud rate, so it is synchronized with the data transfer rate from the development board. When running the program, the successful output of the producer and consumer tasks sending and receiving data can be seen in Figure 2.

4.0 Conclusion

In this lab, the group faced challenges when defining and declaring the task handles between the freertos.c file and the main c file. It caused errors and warnings that were generated due to mistakes and incompletion of code from earlier steps in the lab. We also had issues displaying both task shifts correctly in the terminal towards the end of the lab. This was due to incorrect usage of the delay function for task 2.

Challenges aside, this lab taught the group how real-time systems can be managed using semaphores and mutexes. It grew our understanding of preemptive scheduling and synchronizations specifically. This included implementing the stack size, queue times and choosing priorities.

In terms of technical skills, this lab strengthened our programming and debugging skills for real time applications. It helped us transfer the lecture information into a real application in the STM32CubeIDE, which is also a tool we were not familiar with prior to the lab completion. In a group lab environment, we also improved our teamwork and problem solving skills. To keep improving these skills, we will have to continue developing real time systems that are more challenging and different in goals and targets so that we learn to apply our experience through other real time concepts.

5.0 References

[1] R. Muresan and K. Dong “ENGG4420: Real-Time Systems Design – Lab Manual” Courselink, https://courselink.uoguelph.ca/d2l/le/content/972947/viewContent/4257371/View (accessed Oct. 20, 2025)

[2] R. Muresan and K. Dong “ENGG4420 Real Time System Design – Lab 2: RTOS basics” Courselink, https://courselink.uoguelph.ca/d2l/le/content/972947/viewContent/4263977/View (accessed Oct. 20 2025)

[3] K. Dong “ENGG4420 Real Time System Design – Lab 2 RTOS Introduction” Courselink, https://courselink.uoguelph.ca/d2l/le/content/972947/viewContent/4264009/View (accessed Oct. 20 2025)

6.0 Appendix

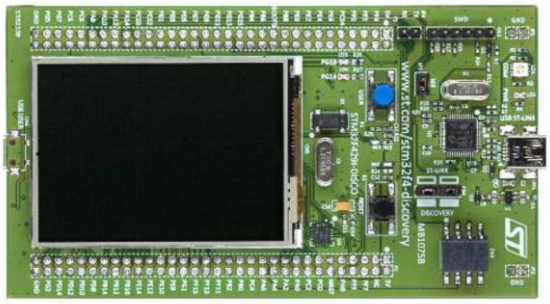

Figure 1: *STM32F429 Discovery Kit*

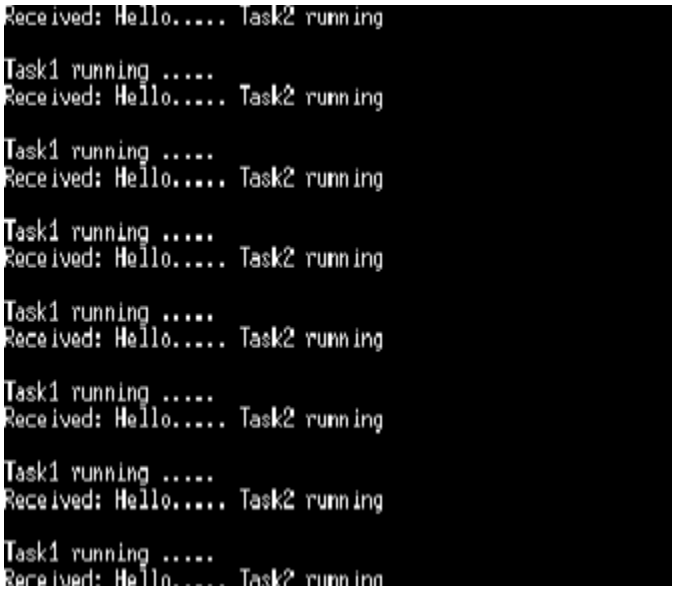

Figure 2: *Producer & Consumer Task Outputs*