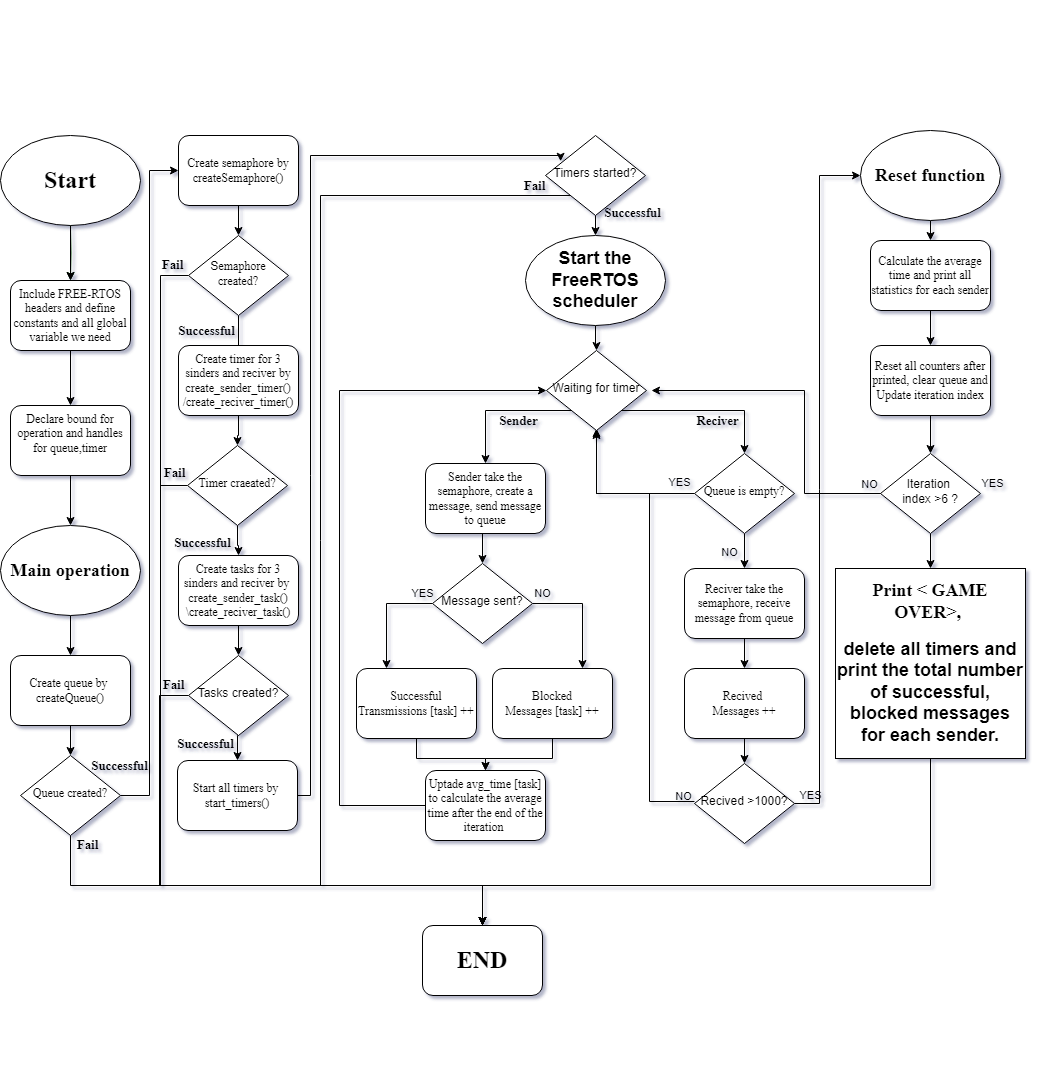
|  |  |  |  |
| --- | --- | --- | --- |
| BN | Section | Student ID | اسم الطالب |
| **32** | **2** | **9220457** | **عبدالرحمن احمد محمد عبداللطيف** |
| **26** | **1** | **9220145** | **اسلام فتحي محمد سليمان** |

# System Design

Here’s a flow chart of the system of RTOS Communicating Tasks:

Flow chart 1: System Design

**Explanation:**

This flowchart details the steps of a program using the Free-RTOS operating system, involving creating semaphores, timers, tasks, and handling message queues. **Here's a step-by-step explanation:**

**Start:**

Include the necessary **Free-RTOS** header files. Define constants and declare the global variables needed for the program, define bounds and other required.

**Main Operation:**

* Call the createQueue() function to create the message queue, the createsemaphore() function to create the necessary semaphores, the create\_sender\_timer() to create timers for sender tasks, and create\_reciver\_timer() to create a timer for the receiver task, the create\_sender\_task() to create tasks for sending operations and create\_reciver\_task() to create the task for receiving operations, start\_timers() to start all the created timers.
* After each of these functions we check if the semaphores were successfully created, if not, the program ends with a failure message.
* If all steps are done correctly, the Free-RTOS scheduler begins

Code 1: Create Semaphore Code

**void createsemaphore(void)**

**{for (int i = 0; i < 3; i++)**

**{ xSenderSemaphores[i] = xSemaphoreCreateBinary();**

**if (xSenderSemaphores[i] == NULL)**

**{trace\_printf("Failed to create sender semaphore %d\n", i);**

**exit(0); }}**

**xReceiverSemaphore = xSemaphoreCreateBinary();**

**if (xReceiverSemaphore == NULL)**

**{ trace\_printf("Failed to create receiver semaphore\n");**

**exit(0);}}**

**Free-RTOS Scheduler:**

* The program enters a waiting state until timers trigger tasks. The sender task takes a semaphore when its timer triggers. The sender creates a message and attempts to send it to the queue.
* Check if the message was successfully sent to the queue. If successful, increment the sender's "successful transmissions" counter. If not, increment the sender's "blocked messages" counter. Update the average time for the sender task for statistical purposes.
* The receiver task takes a semaphore when its timer triggers. The receiver attempts to receive a message from the queue. Check if the queue is empty. If empty, wait until the next timer trigger. If there are messages, the receiver processes them. Check if the number of received messages has reached 1000. If fewer than 1000, return to waiting. If 1000 or more, call the reset function.

Code 2: Receiver Task Code

void ReceiverTask(void \*parameters)

{while (1)

{// Wait for the semaphore to be released by the timer

xSemaphoreTake(xReceiverSemaphore, portMAX\_DELAY);

// Check the queue for messages

if (xQueueReceive(testQueue, receivedMessage, 0) == pdPASS)

{receivedMessages++;

trace\_printf("Received: %s\n", receivedMessage);

// If 1000 messages received, call the reset function

if (receivedMessages >= 1000)

{vResetFunction();}}}}

Code 3: Start Timer

void start\_timers(void)

{for (int i = 0; i < 3; i++)

{//start timer of each sender with the random period

timerStarted = xTimerStart(xSenderTimers[i], 0);

if (timerStarted != pdPASS)

{//check if the timer started successful

trace\_printf("Handle timer start failure\n");

exit(0);}}

//start timer of receiver task with fixed period

timerStarted = xTimerStart(xReceiverTimer, 0);

if (timerStarted != pdPASS)

{trace\_printf("Handle timer start failure\n");

exit(0);}}

**Callback Functions:**

* **vSenderTimerCallback:** Called when a sender timer expires. It releases the corresponding sender task semaphore, allowing the sender task to attempt sending a message to the queue.
* **vReceiverTimerCallback:** Called when the receiver timer expires. It releases the receiver semaphore, allowing the receiver task to check for and process incoming messages from the queue.

void vSenderTimerCallback(TimerHandle\_t xTimer)

{//store the number of task in taskId

int taskId = (int)(intptr\_t)pvTimerGetTimerID(xTimer);

// Release the semaphore associated with this timer

xSemaphoreGive(xSenderSemaphores[taskId]);}

void vReceiverTimerCallback(TimerHandle\_t xTimer)

{// Release the receiver semaphore

xSemaphoreGive(xReceiverSemaphore);}

Code 4: Sender and Receiver Call-Back Function

**Reset Function:**

* Calculate the average transmission time for each sender task and print the statistics for each sender task.
* Reset all counters for the next iteration. Clear the message queue and update the iteration index.
* Check if the iteration index has exceeded 6. If not, return to waiting. If yes, proceed to print "GAME OVER". Print "GAME OVER", delete all timers, and print the total number of successful and blocked messages for each sender task.

**End:** The program ends after completing the final reset function for the last iteration.

# Results and Discussion:

In this program, we choose senders 1 and 2 with the same priority and sender 3 is a higher priority than 1 and 2.

Sender tasks have random timer periods depending on the upper and lower periods.

The **receiver task** has higher priority than the sender tasks, and has a fixed period of **‘100 msec’.**

when sender 3 timer relace the semaphore then the sender 3 task operate immediately.

## Table of statistics:

Table 1: Average Time and Total Messages for Each Sender



\*Avg: Average time for each sender Task1: Sender1 Task2: Sender2 Task3: Sender3

Table 2: Sent and Blocked Messages



\*S: Total sent messages for each sender B: Total blocked messages for each sender

**The gap** between the number of sent and received is that the sender tasks generate messages at a higher rate than the receiver task can process them, causing sender tasks to be blocked until space is available in the queue again. so that the blocked message increases compared to a successful message, we can solve this problem by making the sender tasks and the receiver task operate at the same rate.

## Graphs of the results:

### Queue Size 3:

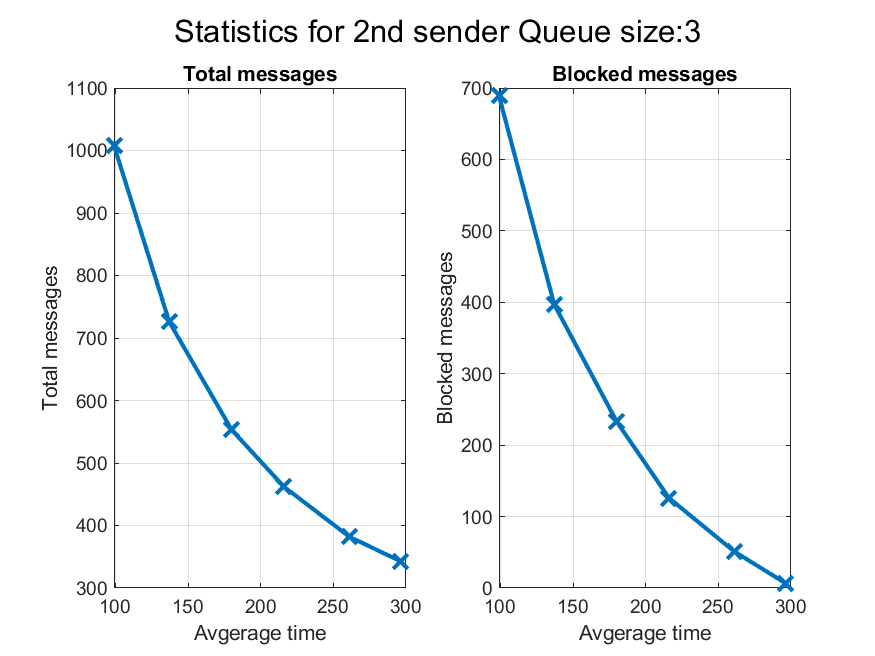
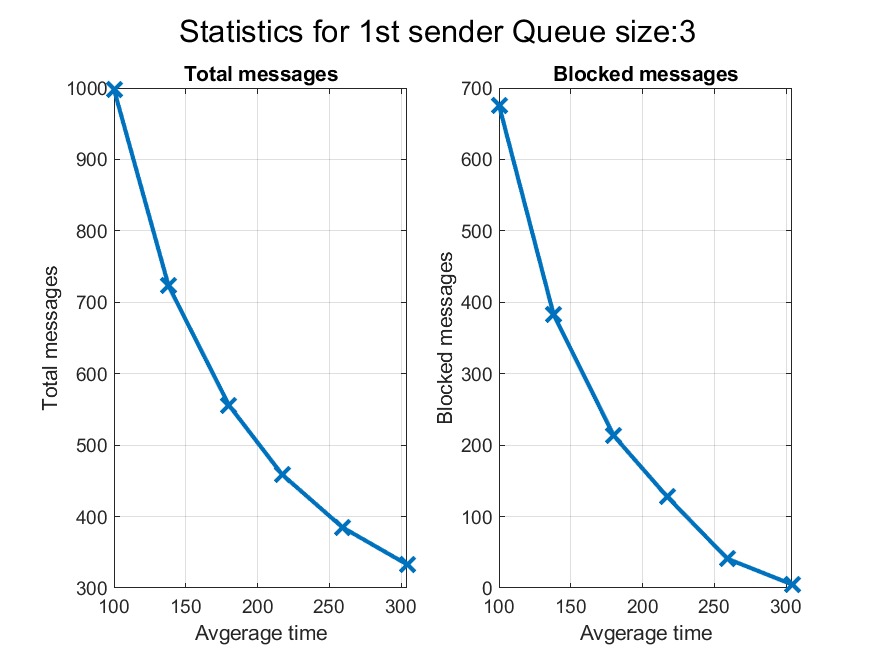


Figure 1: 1st Sender

Figure 2: 2nd Sender

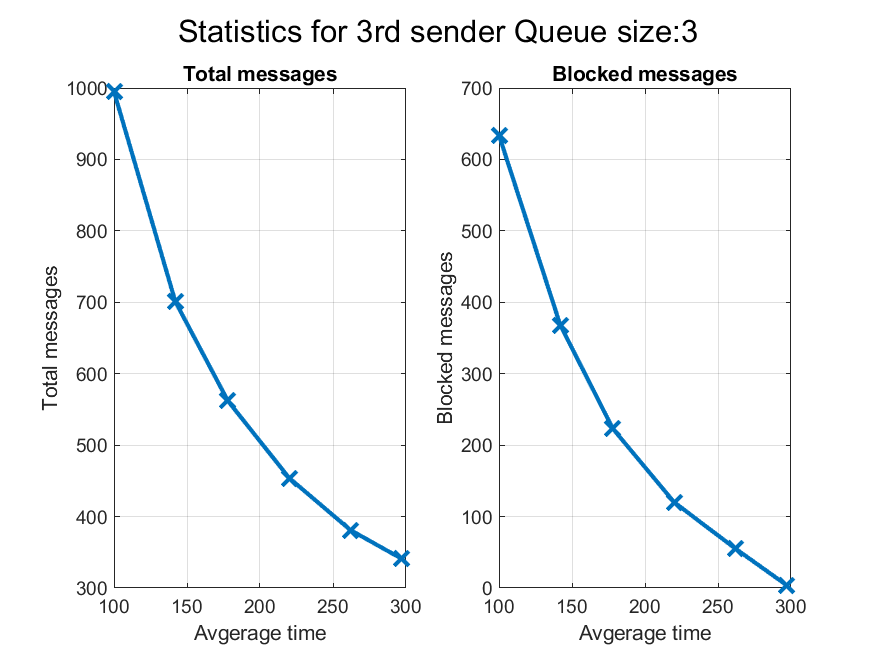
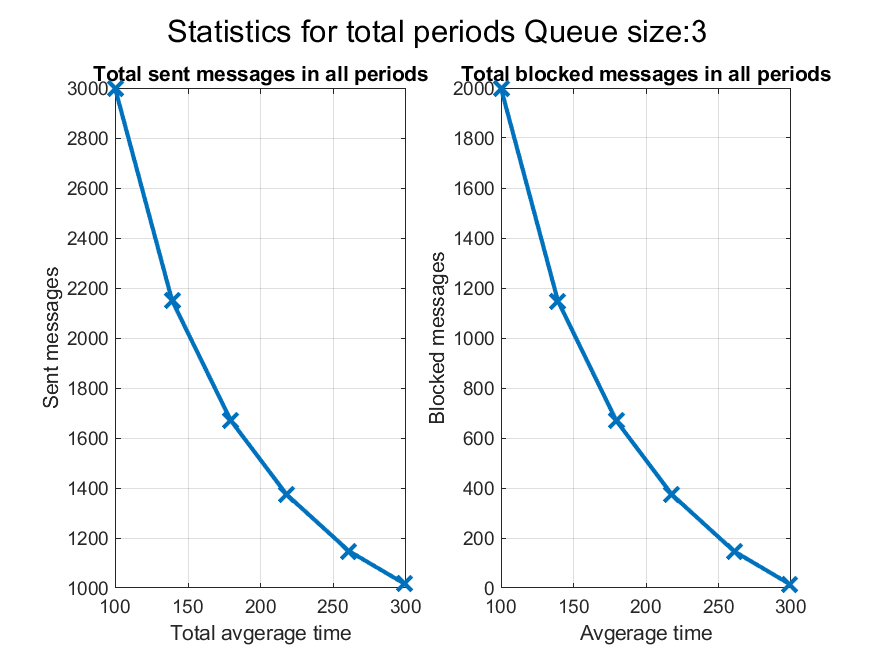


Figure 3: 3rd Sender

Figure 4: All Senders

### Queue Size 10:

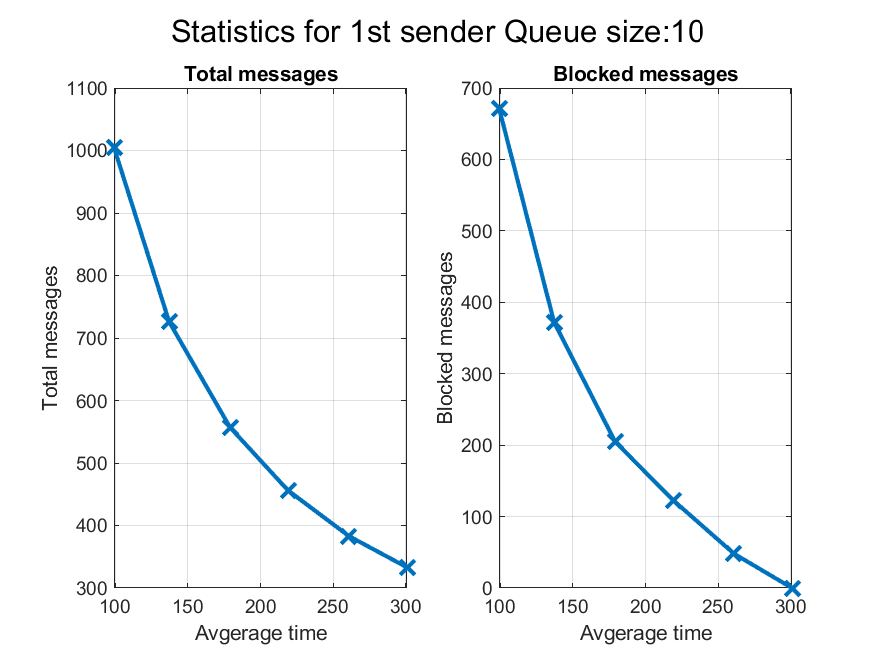


Figure 6: 2nd Sender

Figure 5: 1st Sender

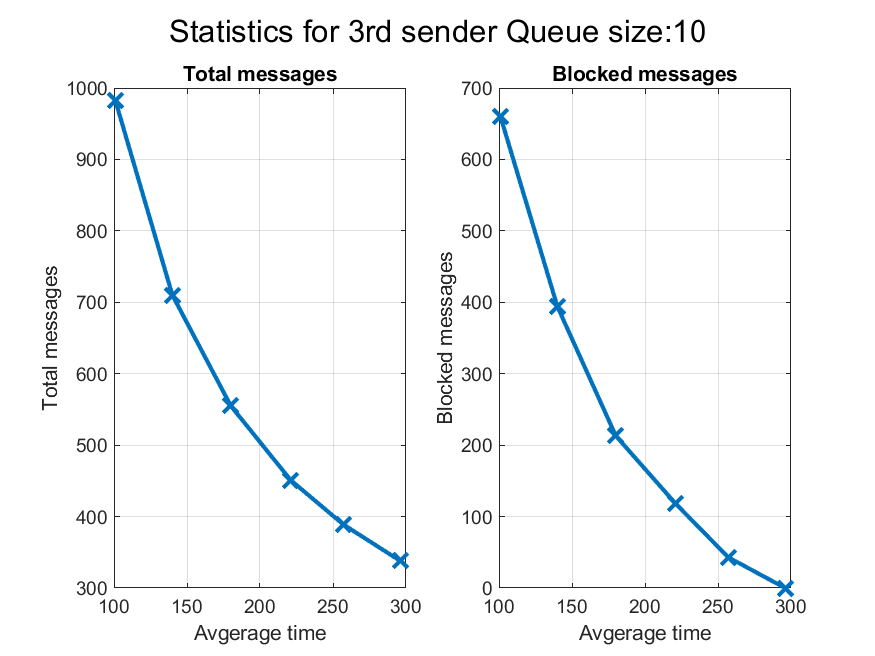
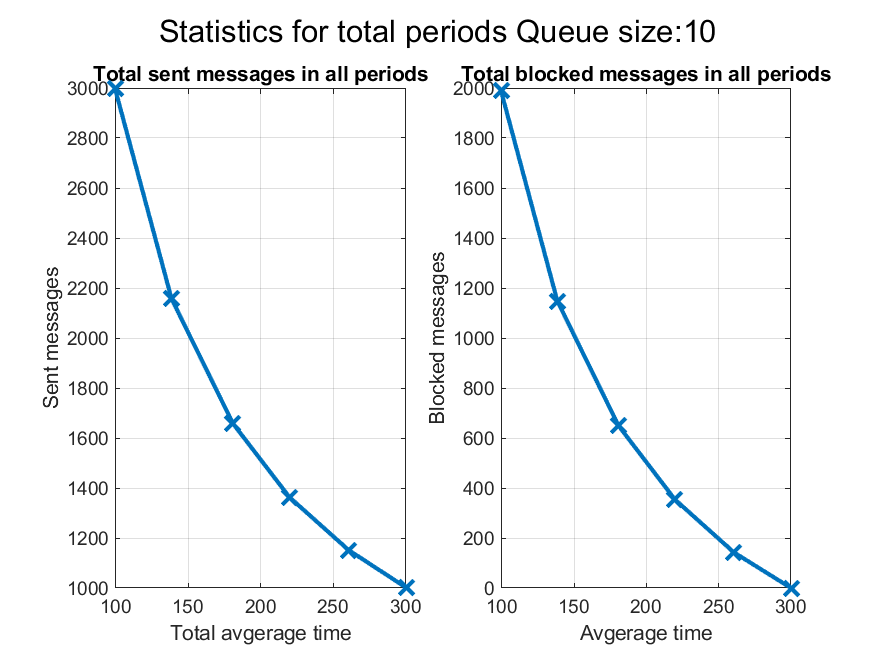


Figure 8: All Senders

Figure 7: 3rd Sender

## From these results and plots, we conclude that:

* With a larger queue size, there is more space to store messages from the sender tasks, this allows sender tasks to transmit their messages more often without being blocked so the blocked message will decrease and successful messages will increase.
* While a larger queue size reduces the blocked messages, it may introduce a delay in processing messages, as messages may wait longer in the queue before being processed by the receiver task especially if the sender tasks generate messages at a rate faster than the receiver task can process them.