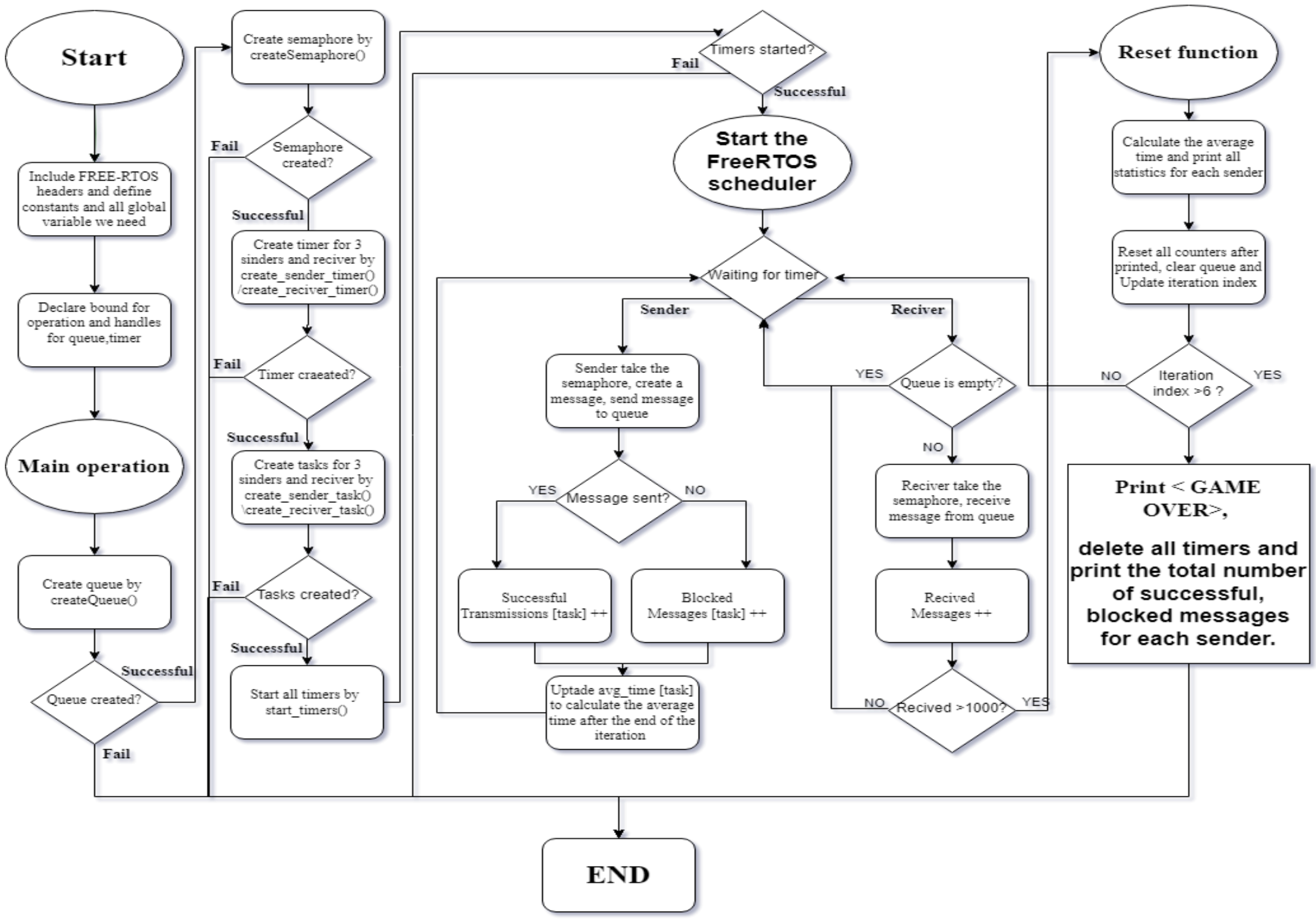
BN	Section	Student ID	اسم الطالب
32	**2**	**9220457**	**عبدالرحمن احمد محمد عبداللطيف**
26	**1**	**9220145**	**اسلام فتحي محمد سليمان**

1 System Design

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



Flow chart 1: System Design

<u>Explanation:</u>

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

```
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);}}
```

Code 1: Create Semaphore Code

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.

```
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 2: Receiver Task Code

```
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);}}
```

Code 3: Start Timer

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.

2 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.

2.1 Table of statistics:

Table 1: Average Time and Total Messages for Each Sender

QueueSize	Period	Avg1	Avg2	Avg3	Total average	Total sent messages		Blocked messages	
Size 3	1	100	99	100	99.66666667	Task1	3454	Task1	1446
	2	138	137	142	139				
	3	180	180	178	179.3333333	Task2	3474	Task2	1501
	4	217	216	220	217.6666667				
	5	259	261	262	260.6666667	Task3	3434	Task3	1404
	6	304	296	297	299				
Size 10	1	99	98	101	99.33333333	Task1	3460	Task1	1420
	2	137	138	140	138.3333333				
	3	179	183	180	180.6666667	Task2	3445	Task2	1435
	4	219	219	221	219.6666667				
	5	261	263	257	260.3333333	Task3	3425	Task3	1428
	6	301	303	296	300				

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

Table 2: Sent and Blocked Messages

QueueSize	Total messages	S1	S2	S3	Blocked messages	B1	B2	B3
Size 3	3001	998	1008	995	1999	676	689	634
	2150	723	726	701	1148	383	397	368
	1671	556	553	562	669	213	233	223
	1375	459	462	454	374	128	126	120
	1148	385	382	381	147	41	51	55
	1017	333	343	341	16	5	7	4
Size 10	2998	1005	1011	982	1989	672	657	660
	2157	726	721	710	1148	372	382	394
	1658	557	546	555	649	205	231	213
	1363	456	456	451	354	123	113	118
	1152	383	380	389	143	48	52	43
	1002	333	331	338	0	0	0	0

*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.

2.2 Graphs of the results:

2.2.1 Queue Size 3:

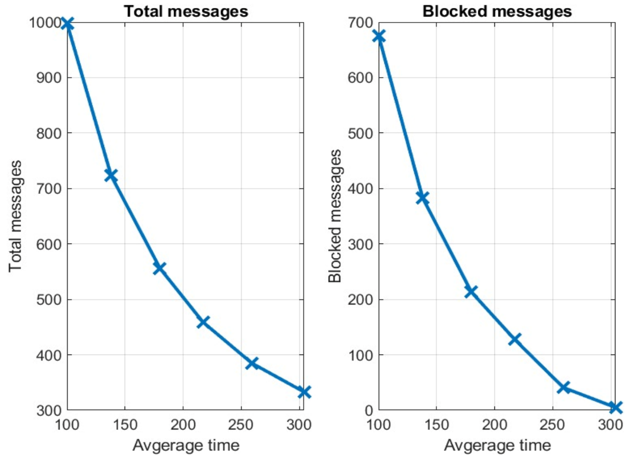


Figure 1: 1st Sender

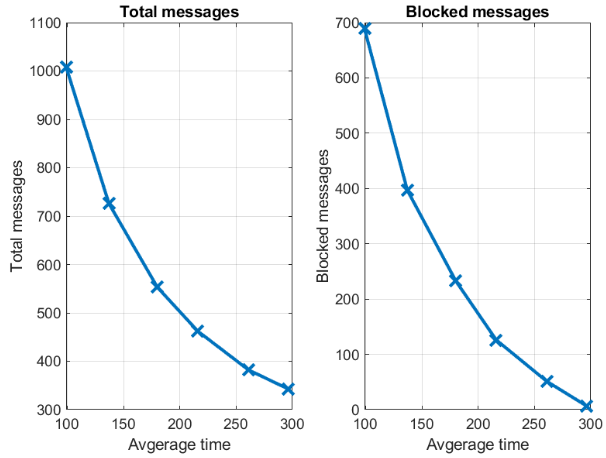


Figure 2: 2nd Sender

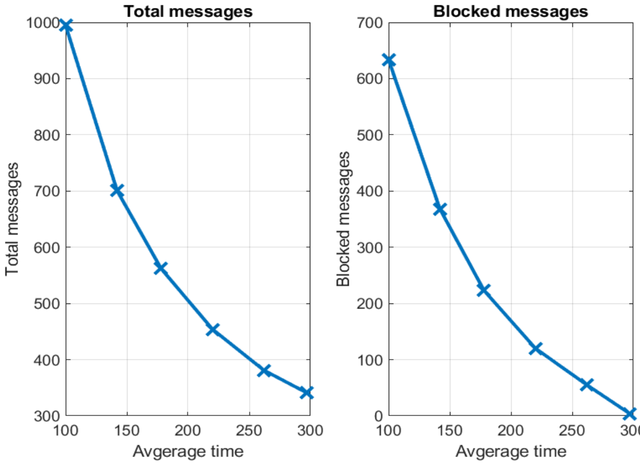


Figure 3: 3rd Sender

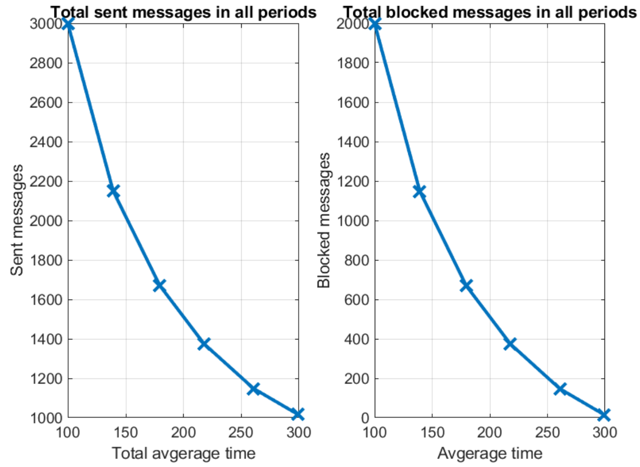


Figure 4: All Senders

2.2.2 Queue Size 10:

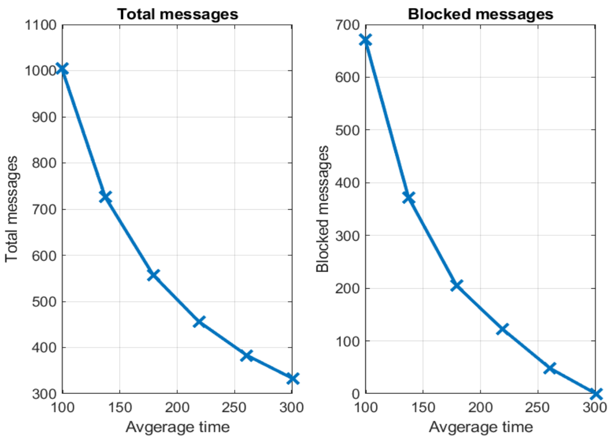


Figure 5: 1st Sender

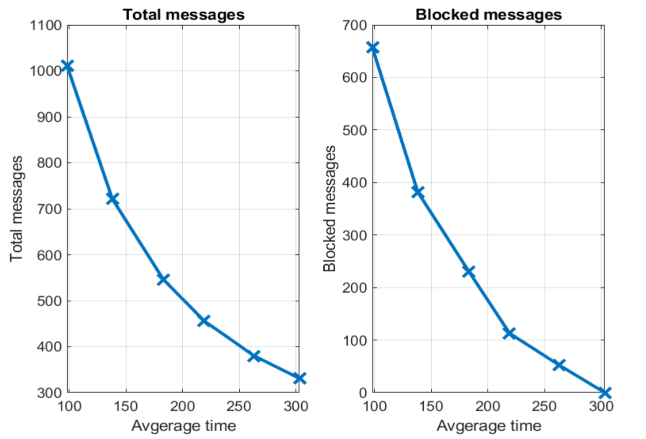


Figure 6: 2nd Sender

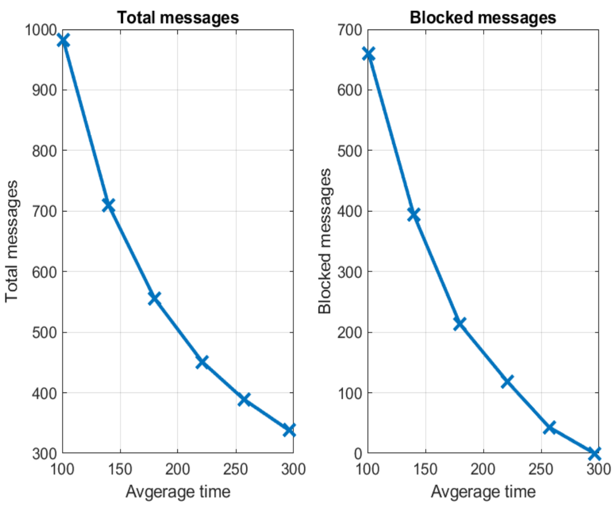


Figure 7: 3rd Sender

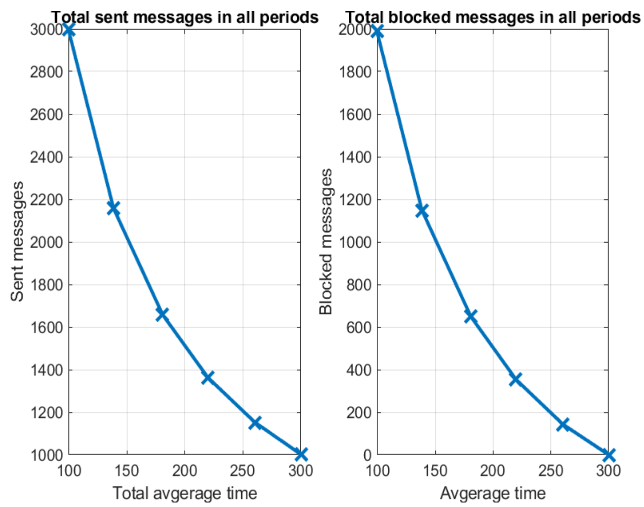


Figure 8: All Senders

2.3 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.