BN	Section	Student ID	اسم الطالب
33	**3**	**9210899**	**Magdy Ahmed Abbas Abdelhamid**

1 System Design

1.1 Overview

This project aims to apply the knowledge of embedded programming and gain practical experience with Real-Time Operating System (RTOS) concepts. The project utilizes FreeRTOS on an emulation board provided via Eclipse CDT Embedded. It involves the implementation of four tasks that communicate through a fixed-size queue. Three sender tasks, with varying priorities, periodically send messages containing the current system time to the queue. If the queue is full, the sending operation fails and a counter for blocked messages is incremented. The receiver task periodically checks the queue for received messages, increments the counter for received messages, and sleeps again if no message is found. The sleep/wake control for the tasks is managed through timers, and their callback functions release dedicated semaphores to unblock the corresponding tasks. The project also includes a reset function that prints statistics, clears the counters and queue, configures the sender timer period, and handles the termination of the system when all timer values have been used. The receiver timer callback function triggers the reset function after receiving 1000 messages.

1.2 Design Implementation

The design and implementation of the code can be summarized as follows:

1. **Tasks:**
- There are four tasks: ‘TransmitterTask_1’, ‘TransmitterTask_2’, ‘TransmitterTask_3’, and ‘ReceiverTask’.
- Each task runs an infinite loop, performing specific actions based on its dedicated semaphore.
- ‘TransmitterTask_1’, ‘TransmitterTask_2’, and ‘TransmitterTask_3’ wait for their respective transmit semaphores and send messages to the message queue when the semaphore is obtained.
- ‘ReceiverTask’ waits for the receive semaphore and receives messages from the message queue.

2. **Queue:**
- The code creates a message queue named ‘MessageQueue’ with a Size of 3 or 10 depends on the required.

3. **Semaphores:**
- There are four semaphores: ‘TransmitSemaphore_1’, ‘TransmitSemaphore_2’, ‘TransmitSemaphore_3’, and ‘ReceiverSemaphore’.
- The transmit semaphores are used to control the execution of the transmitter tasks.
- The receiver semaphore is used to control the execution of the receiver task.

4. **Timers:**
- There are four timers: ‘xTransmitter_1_TimerHandle’, ‘xTransmitter_2_TimerHandle’, ‘xTransmitter_3_TimerHandle’, and ‘xReceiverTimerHandle’.
- Each timer is associated with a specific transmitter task or the receiver task and triggers a callback function when the timer expires.
- The callback functions for the transmitter tasks update the respective timers’ periods, reset the timers, and release the associated transmit semaphore (In all iterations ‘Tsender’ is random).
- The callback function for the receiver task checks the number of received messages and calls the ‘ResetFunction’ if the condition is met (In all iterations ‘Treceiver’ is fixed at 100 msec).

5. **Variables and Constants:**
- There are several variables and constants used to store and manipulate message counts, time periods, and other parameters.

6. **Initialization and Main Execution:**
- The main function initializes the tasks, queues, semaphores, and timers.
- It creates the transmitter and receiver tasks using ‘xTaskCreate’ and assigns task handles.
- It creates the message queue using ‘xQueueCreate’ and assigns it to ‘MessageQueue’.
- It creates the semaphores using ‘xSemaphoreCreateBinary’ and assigns them to respective variables.
- It creates the timers using ‘xTimerCreate’.
- The ‘MainReset’ function is called to initialize/reset various counters and random periods.
- The FreeRTOS scheduler started with ‘vTaskStartScheduler’.

7. **Reset and Statistics:**

- The ‘ResetFunction’ is responsible for printing statistics, resetting counters, and updating periods.
- It checks if the number of received messages exceeds a threshold (1000) and calls ‘ResetFunction’ to display statistics and reset the system.
- If the reset counter reaches $(0-5)$ (6 iterations), the scheduler is stopped, and the program ends and print ‘Game Over’.

1.3 Code Snippets

To produce a random duration within a specified range, we utilize the rand function, which generates a random number. The first array holds the values $\{50, 80, 110, 140, 170, 200\}$ and the second holds the values $\{150, 200, 250, 300, 350, 400\}$ expressing in msec the timer lower and upper bounds for a uniform distribution.
By applying the following formula, we can generate the desired duration: $T = \text{Rand}()\%(\text{Upper} - \text{Lower} + 1) + \text{Lower}$
This duration represents the interval for the 3 sender tasks, after which the callback function is triggered.

```
void UpdateRandomPeriods(void) {
Tsender1 = (rand()%(UpperBound[Reset] − LowerBound[Reset] + 1) + LowerBound[Reset]);
Tsender2 = (rand()%(UpperBound[Reset] − LowerBound[Reset] + 1) + LowerBound[Reset]);
Tsender3 = (rand()%(UpperBound[Reset] − LowerBound[Reset] + 1) + LowerBound[Reset]); }
```

The Following code snippets demonstrates the sender timer callback function, illustrating the process of generating random numbers and calculating the average sender period, while also showcasing the unblocking mechanism.

```
void TransmitterTask_1(void * p_T1_Parameters) {
  while (1) {
    if (xSemaphoreTake(TransmitSemaphore_1, portMAX_DELAY) == 1) {
      // Semaphore successfully acquired
      sprintf(Message, "Time is %lu\n", xTaskGetTickCount());
      if (xQueueSend(MessageQueue, &Message, 0) == pdPASS) {
        // Message sent successfully
        Transmitter_1_SentMessages + +;        } else {
        // Failed to send message
        Transmitter_1_BlockedMessages + +;        }     }  } }
```

```
static void Transmitter_1_TimerCallback(TimerHandle_t xTimer_T1_Started) {
  // Generate a random value for Tsender1 within the specified range
  Tsender1 = (rand() % (UpperBound[Reset] − LowerBound[Reset] + 1) + LowerBound[Reset]);
  // Change the period of the transmitter timer to Tsender1 milliseconds
xTimerChangePeriod(xTransmitter_1_TimerHandle, pdMS_TO_TICKS(Tsender1), 0);
  // Reset the transmitter timer to start counting from zero
  xTimerReset(xTransmitter_1_TimerHandle, 0) ;
  // Release the Transmitter semaphore to allow other tasks to access the transmitter
xSemaphoreGive(TransmitSemaphore_1); }
```

1.4 Program Flow

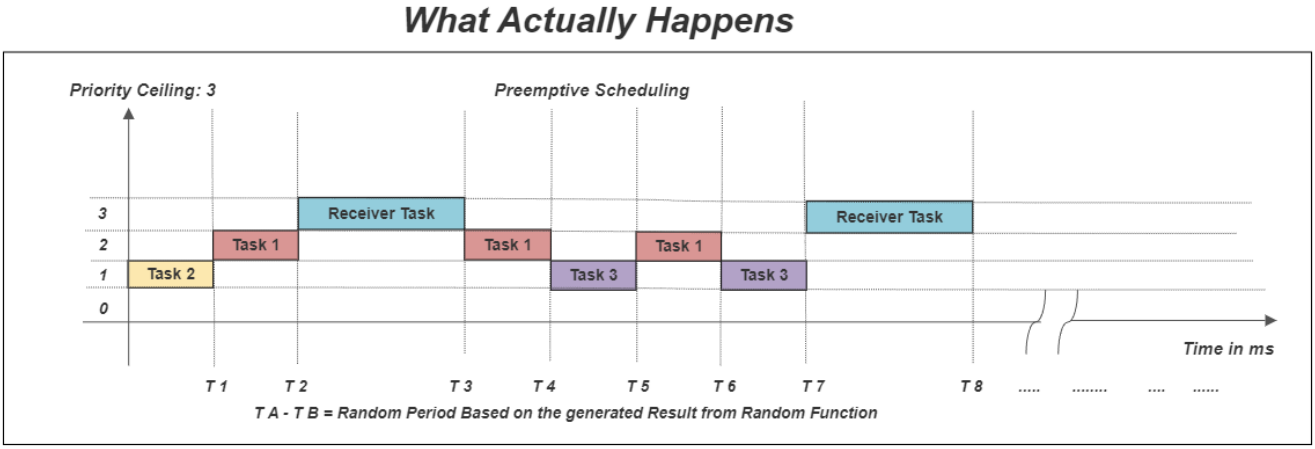

Figure 1: Tasks Priorities Vs Time.

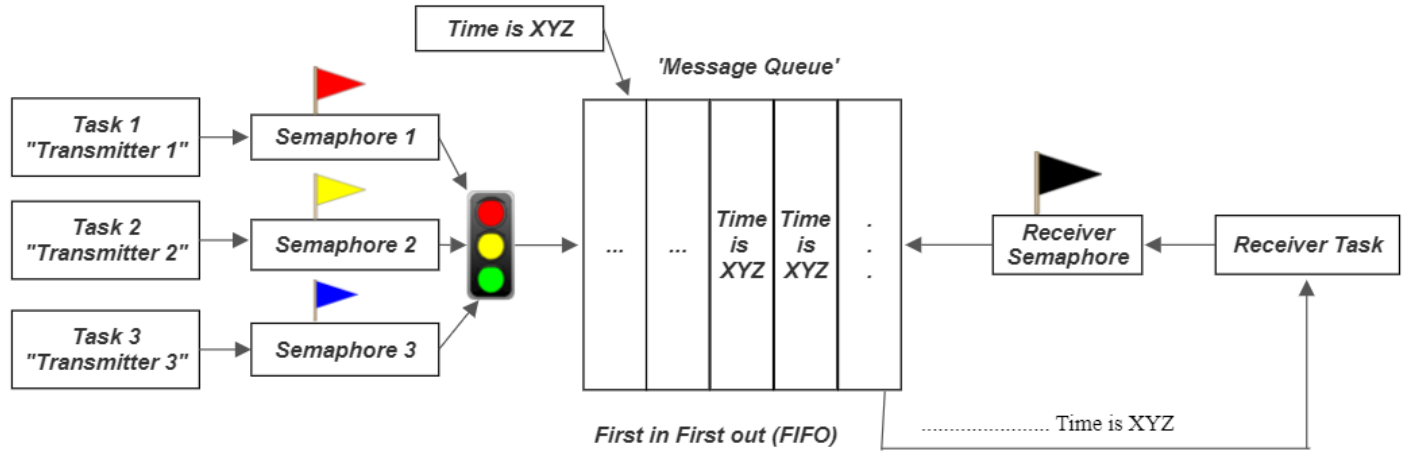

Figure 2: Data Structure Design.

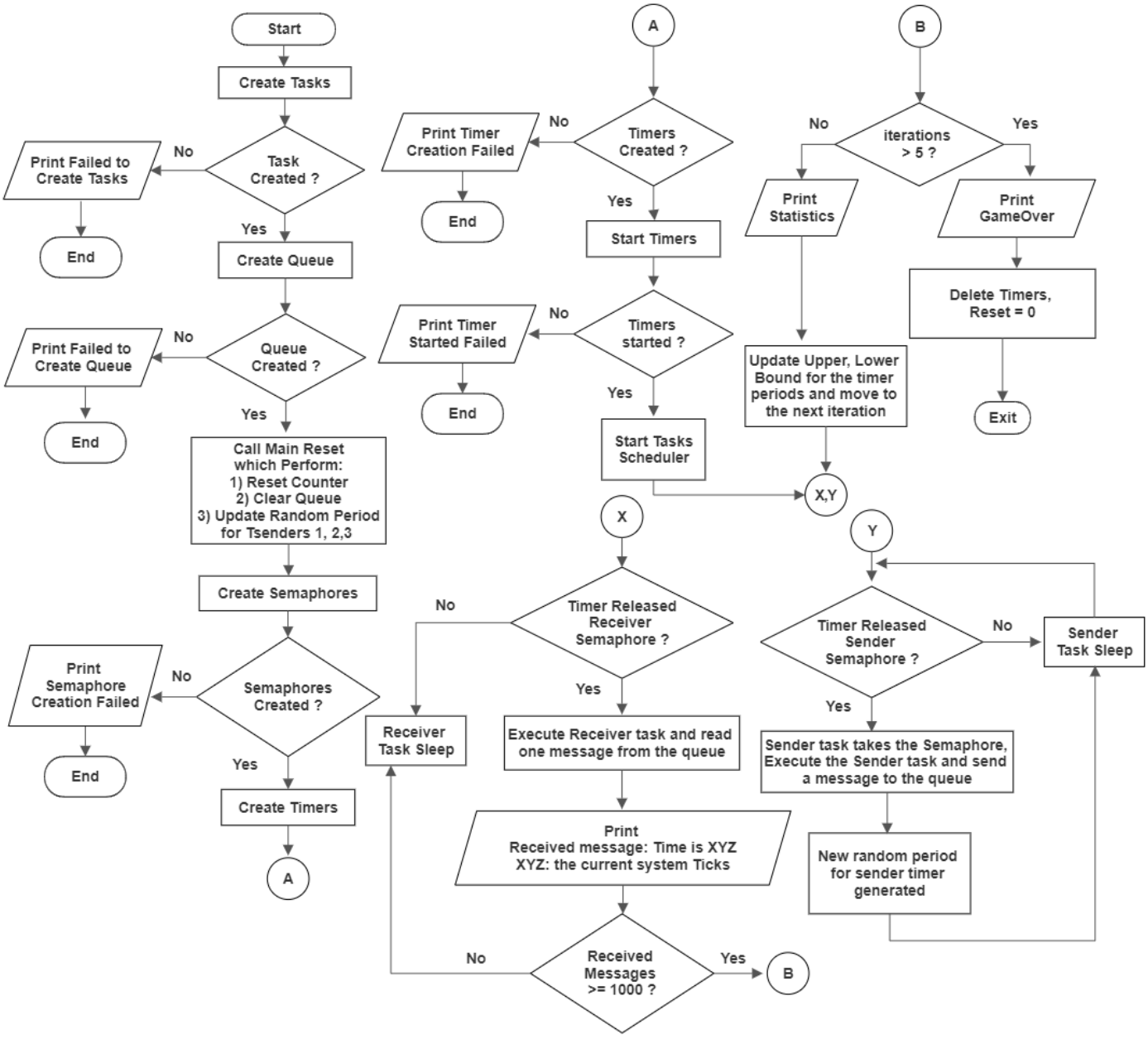

Figure 3: Flowchart Showing Program Flow.

2 Results and Discussion

2.1 Results

We executed the program for 6 iterations Twice. In the first run, we used a queue of Size 3, while in the second run, we used a queue with of Size 10, and with Average Sender period $= \frac{\text{upper} + \text{lower}}{2}$, With each run on the first, it displays messages to set the ticks time with the 'Received message: time is XYZ' After analyzing the program's output, we obtained the following outcomes:

2.1.1 Queue of Size 3

Table 1: Number of Sent, Blocked Messages for the **3** Senders, Total number of received, Sent, Blocked messages.

Statistics of Iterations	Sent Messages Sender 1	Blocked Messages Sender 1	Sent Messages Sender 2	Blocked Messages Sender 2	Sent Messages Sender 3	Blocked Messages Sender 3	Total # of Sent Messages	Total # of Blocked Messages	Total # of Received Message	Avg. Sender Period
0	336	674	329	688	338	672	1003	2034	1000	100
1	348	365	312	407	343	377	1003	1149	1000	140
2	355	197	331	222	317	237	1003	656	1000	180
3	332	122	337	114	334	116	1003	352	1000	220
4	341	42	325	63	337	47	1003	152	1000	260
5	334	5	329	7	339	1	1002	13	1000	300

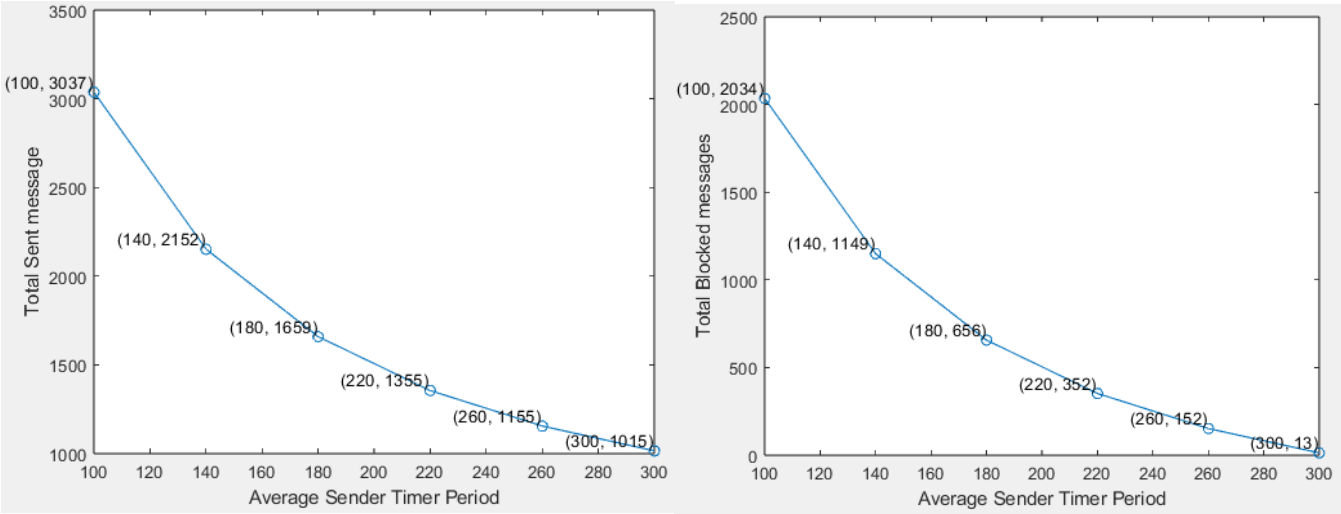

Figure 4: Total (Sent = Transmitted + Blocked) & Blocked messages VS the Average Sender Period for a queue of size 3.

Note:

Explaining the gap between the number of sent and received messages in the running period will be in the Discussion and Conclusion Section.

2.1.2 Queue of Size 10

Table **2**: Number of Sent, Blocked Messages for the **3** Senders, Total number of received, Sent, Blocked messages.

Statistics of Iterations	Sent Messages Sender 1	Blocked Messages Sender 1	Sent Messages Sender 2	Blocked Messages Sender 2	Sent Messages Sender 3	Blocked Messages Sender 3	Total # of Sent Messages	Total # of Blocked Messages	Total # of Received Message	Avg. Sender Period
0	361	638	322	693	327	696	1010	2027	1000	100
1	344	373	331	383	335	386	1010	1142	1000	140
2	349	202	323	229	338	218	1010	649	1000	180
3	337	114	333	118	340	113	1010	345	1000	220
4	338	46	347	40	325	58	1010	144	1000	260
5	335	0	336	0	332	0	1003	0	1000	300

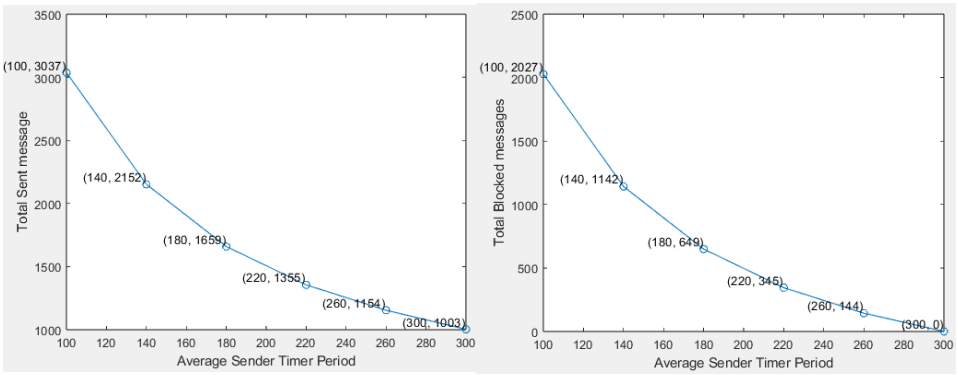

Figure 5: Total (Sent = Transmitted + Blocked) & Blocked messages VS the Average Sender Period for a queue of size 10.

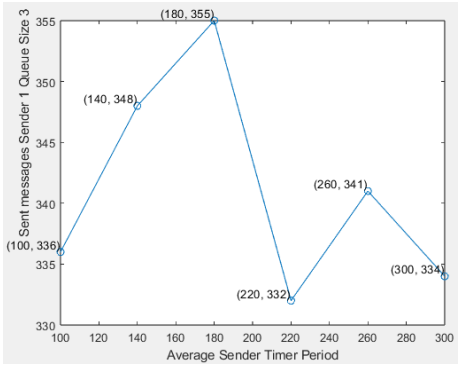

Figure 6: Sender 1 Sent msg. Queue 3

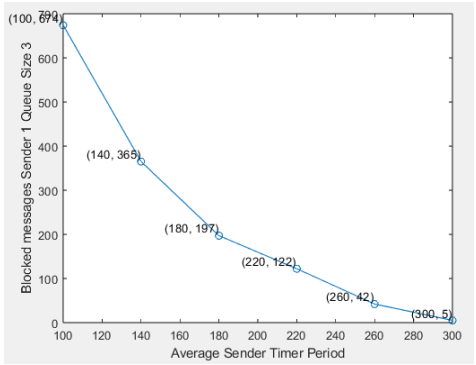

Figure 7: Sender 1 Blocked msg. Queue 3

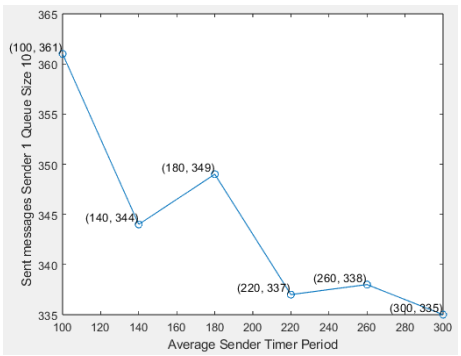

Figure 8: Sender 1 Sent msg. Queue 10

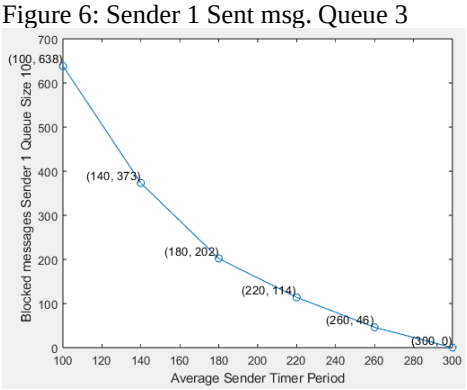

Figure 9: Sender 1 Blocked msg. Queue 10

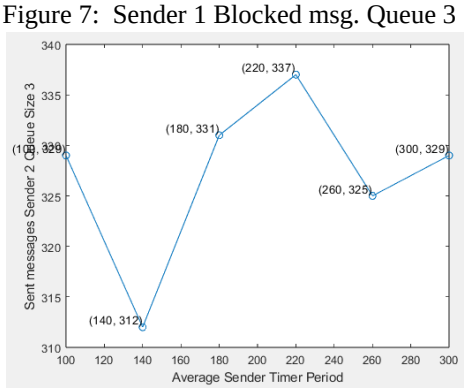

Figure 10: Sender 2 Sent msg. Queue 3

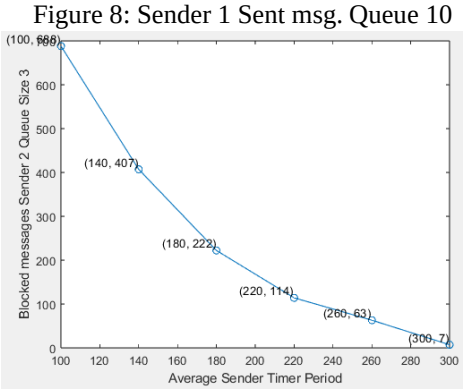

Figure 11: Sender 2 Blocked msg. Queue 3

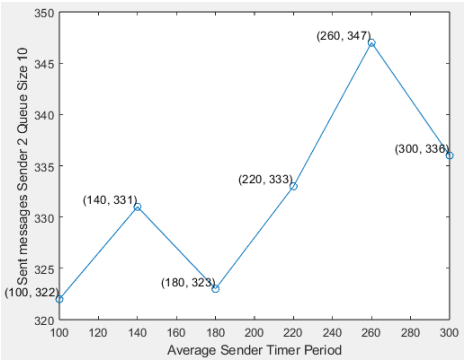

Figure 12: Sender 2 Sent msg. Queue 10

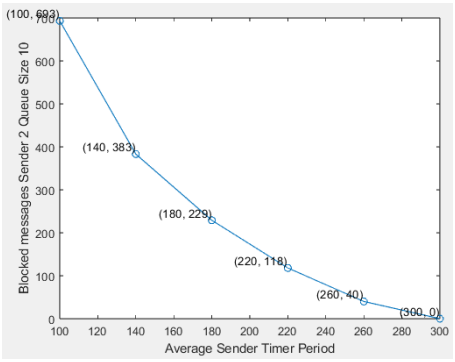

Figure 13: Sender 2 Blocked msg. Queue 10

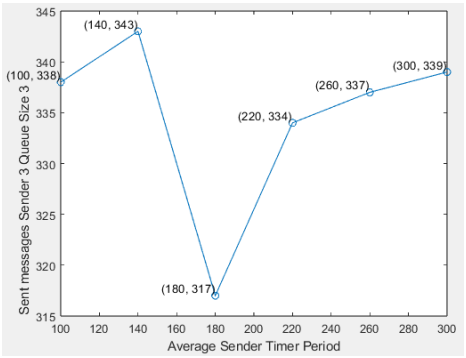

Figure 14: Sender 3 Sent msg. Queue 3

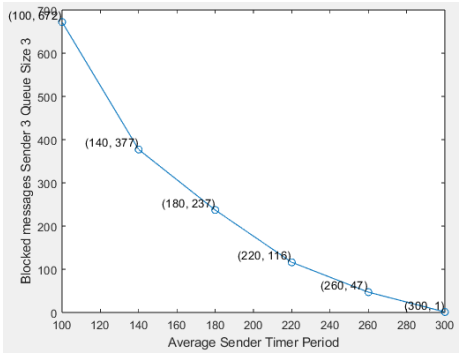

Figure 15: Sender 3 Blocked msg. Queue 3

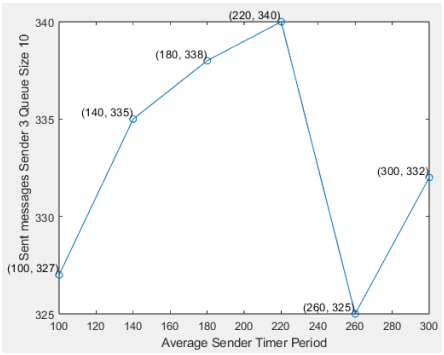

Figure 16: Sender 3 Sent msg. Queue 10

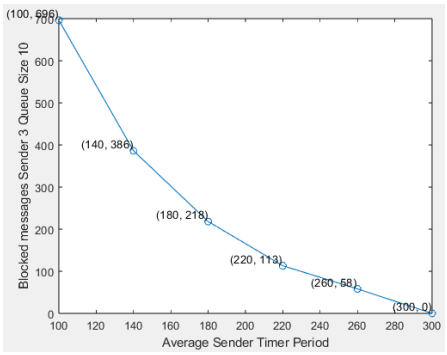

Figure 17: Sender 3 Blocked msg. Queue 10

2.2 Observations

Based on the data presented above, the following observations can be made:

1. The number of sent messages exceeds the number of received messages either by the size of the queue or by a slight margin as shown in Tables 1 & 2 Columns 8 & 11 and Figures 4 & 5.
2. As the average sender period increases, the number of blocked messages decreases as shown in all Tables and all figures.
3. Increasing the queue size results in a decrease in the overall number of blocked messages and also accelerates the process of reaching zero blocked messages.

2.3 Discussion and Conclusions

The receiver extracts messages from the queue at a constant rate (every 100ms), while the sender transmits messages at a variable rate spanning a wide range. If the sender's transmission interval, 'Tsender', is shorter than the receiver's interval, 'Treceiver' in the initial iteration, it indicates that the queue fills up faster than the receiver can handle. Conversely, if 'Tsender' is greater than 'Treceiver', it implies that the receiver can empty the queue before it becomes full. Therefore, the number of blocked messages depends on the receiver's ability to manage the queue, which is determined by 'Tsender'. As 'Tsender' increases, the number of blocked messages decreases, explaining Observation number 2.

Observation number 3, which pertains to the increase in queue size to 10, follows logically from the previous conclusion. The number of blocked messages is fundamentally influenced by the receiver's capacity to handle the queue. When the queue size increases, senders can add more messages to the queue without filling it completely, giving the receiver more time to process the messages before the queue reaches its maximum capacity. Consequently, enlarging the queue size leads to a reduction in the average number of blocked messages. Combined with an increase in 'Tsender', this results in a faster attainment of zero blocked messages compared to when the queue size was 3, as both these factors aid the receiver in keeping pace with the queue.

Observation number 1, regarding the Gap between the sent and received messages, can be explained by the fact that the receiver can only read one message upon waking up. For instance, suppose we have an empty queue of size n, and the receiver has already processed 999 messages. In the meantime, senders could have sent n messages before encountering a blockage due to the queue being full. When the receiver awakens and processes the 1000^{th} message, there may still be up to $n - 1$ messages remaining in the queue, which accounts for the discrepancy between the number of sent and received messages.

In simpler terms, there is a difference between the total number of messages sent and the number of messages received. This occurs because there are three senders and only one receiver. When the queue size is 3 and the delay is 100 milliseconds, the queue becomes full. As a result, when the receiver becomes available, it can only receive one message out of the three sent. After another 100 milliseconds, there may be attempts from two or three senders to send messages, but only one sender can successfully send while the others perceive the queue as full, resulting in blocked messages. When the queue size increases to 10, the gap between the total number of sent messages and received messages will slightly widen.