BN	Section	Student ID	اسم الطالب
28	**3**	**9220705**	**محمد عادل احمد السيد**

1 System Design

In system design's section we'll go through the workflow of the Code explaining each piece of code:

1- Initialization and Setup process:

Include standard libraries and the necessary headers for FreeRTOS.

2- Task, Semaphores and Timer Callbacks Declaration:

- Declare task functions (SenderTask1, SenderTask2, SenderTask3, ReceiverTask) as shown in figure 1.
- Declare xTimerSenders of TimerHandle_t datatype as a name for the timers which will be used with xTimerCreate function, the callback functions which are called when the timer of each task expires, declare xTimerSendersStarted variables of BaseType_t datatype to store the status of timer start operations typically returning pdPASS for success and pdFAIL for failure.

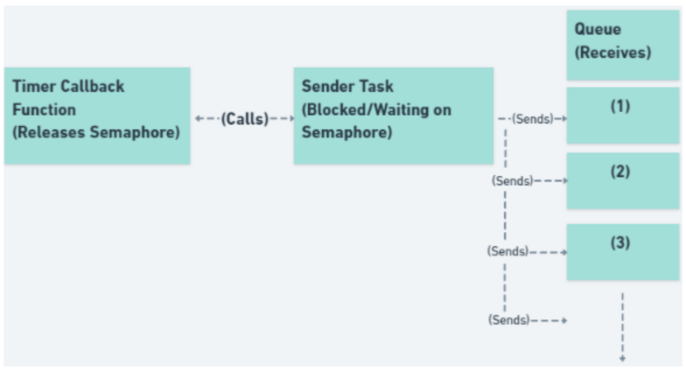

Timer Callback Function (Releases Semaphore)
(Calls)
Sender Task (Blocked/Waiting on Semaphore)
(Sends)
(Sends)
(Sends)
(Sends)
Queue (Receives)
(1)
(2)
(3)

Figure 1: A visual representation of the described process.

- Declare MutexSenders as a semaphore, used to control access to shared resource in each task, our shared queue called testQueue, create a binary semaphore (as semaphores initialization for sender and receiver tasks).

3- Main Function:

- Using xQueueCreate we created the queue of size 3&10 elements each of size char then check if the queue creation was successful by ensuring testQueue is not NULL so if the queue creation is successful, the reset () function is called, and tasks are created using xTaskCreate () as shown in figure 3 indicating the parameters used in this function, if the queue creation was failed an error message is printed, and the program exits peacefully.
- Create software timers as in figure 4 for each sender and the receiver, with initial random periods for senders and a fixed period for the receiver then start all created timers, start the scheduler if all timers are successfully started.

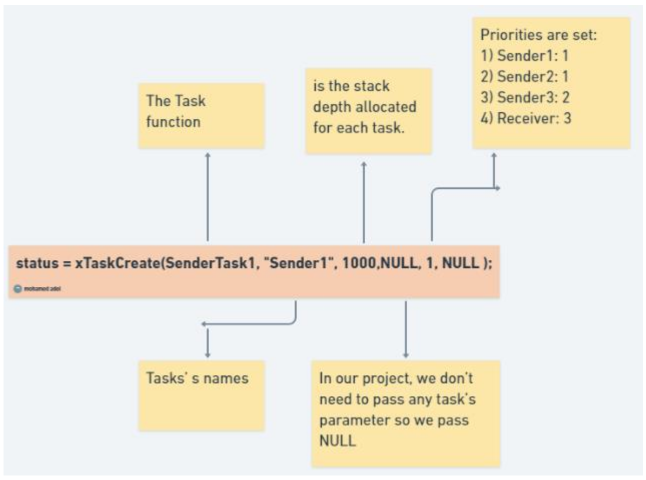

The Task function
is the stack depth allocated for each task.
Priorities are set:
1) Sender1: 1
2) Sender2: 1
3) Sender3: 2
4) Receiver: 3
status = xTaskCreate(SenderTask1, "Sender1", 1000,NULL, 1, NULL);
Tasks' s names
In our project, we don't need to pass any task's parameter so we pass NULL

Figure 3: Chosen parameters for xTaskCreate

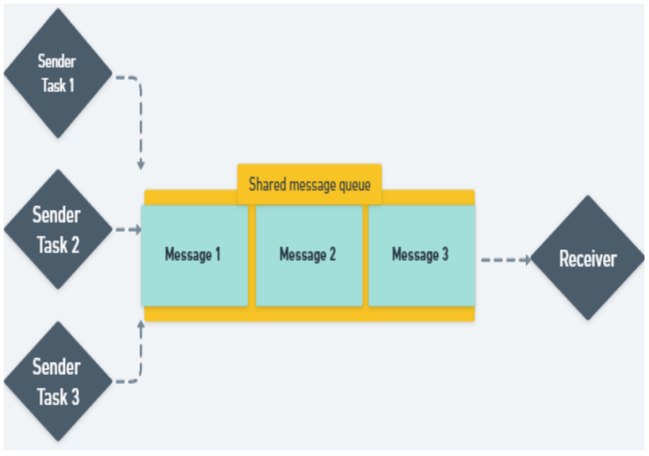


Figure 2: Communications tasks process

4- Program's functions

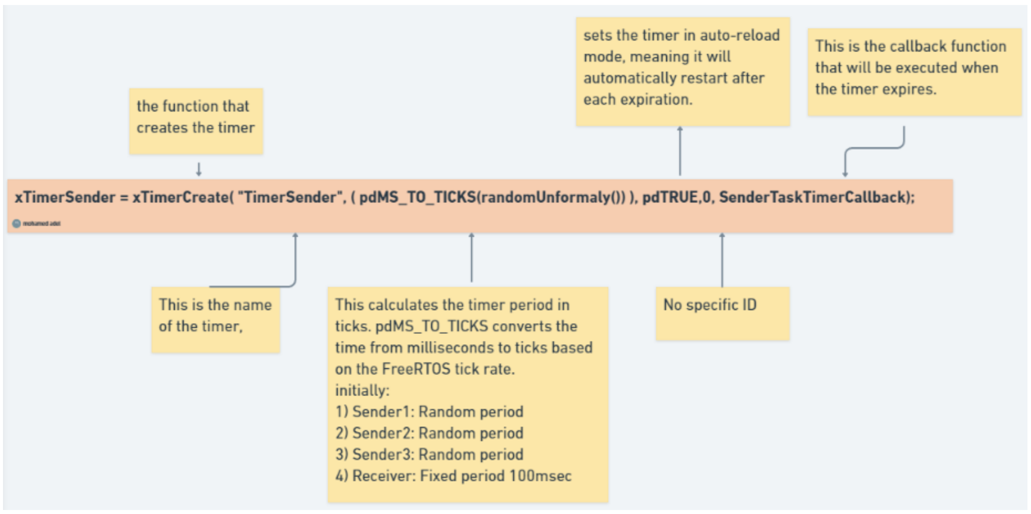
- As shown in code snippet 1 the function firstly releases the semaphore to act like a signal for the corresponding task. Once it's done, the function generates a random number from randomUnformaly function as in code snippet 1 then changes the timer period, incrementing the corresponding sum and its number to get total average period therefore the sender or receiver task are forced to be blocked for a period before they send & receive another message to/from the queue.

```
void SenderTask1TimerCallback(TimerHandle_t xTimerSender1){
  xSemaphoreGive(MutexSender1);
  int randomPeriod = randomUnformaly();
  SumSender1 += randomPeriod;
  PeriodsSender1++;
  TickType_t newPeriod = pdMS_TO_TICKS(randomPeriod);
  xTimerChangePeriod(xTimerSender1, newPeriod, 0);}
```

Code snippet 1: Sender Callback Function

```
int randomUnformaly(){
int lower_bounds[] = {50, 80, 110, 140, 170, 200};
int upper_bounds[]={150, 200, 250, 300, 350, 400};     int index = iterative-1;
int random_value = lower_bounds[index] + rand()%(upper_bounds[index]- lower_bounds[index] + 1);
return random_value;}
```

Code snippet 2: Random Uniformly Function



sets the timer in auto-reload mode, meaning it will automatically restart after each expiration.
This is the callback function that will be executed when the timer expires.
the function that creates the timer
This is the name of the timer,
This calculates the timer period in ticks. pdMS_TO_TICKS converts the time from milliseconds to ticks based on the FreeRTOS tick rate. initially: 1) Sender1: Random period 2) Sender2: Random period 3) Sender3: Random period 4) Receiver: Fixed period 100msec
No specific ID

```
xTimerSender = xTimerCreate( "TimerSender", ( pdMS_TO_TICKS(randomUnformaly()) ), pdTRUE,0, SenderTaskTimerCallback);
```

Figure 4: Chosen Parameters For xTimerCreate

- As shown in code snippet 3, the function corresponding to each sender so firstly declare a container MessageToSend to hold the message which will be sent to the queue testQueue, declare a xStatus, xTimeNow therefore get in the while loop in which the sender task waits for its semaphore, prepares a message with the current tick count, and attempts to send it to the queue. Success and failure counts are updated accordingly.

```
void SenderTask1(){
char MessageToSend[20];   TickType_t xTimeNow;          BaseType_t xStatus;
while(1){
  xSemaphoreTake(MutexSender1,portMAX_DELAY);      xTimeNow=xTaskGetTickCount();
  snprintf(MessageToSend,20,"Time is %u",xTimeNow);
  xStatus = xQueueSend(testQueue, &MessageToSend, 0 );
if (xStatus!=pdPASS) {
C1F++;}    else {C1S++;}}}
```

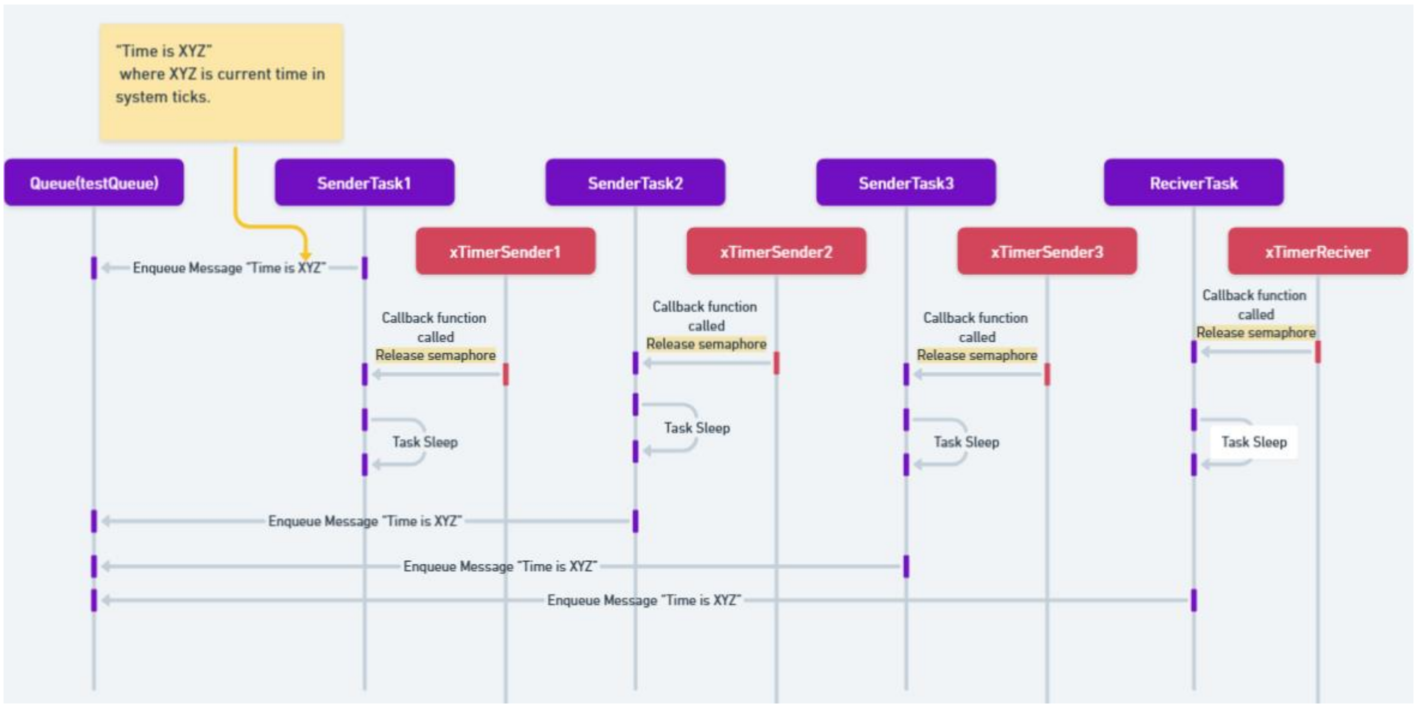
Code snippet 3: Sender Function

- As shown in code snippet 4, the receiver function waits for a signal from its callback function, if it receives it, it tries to read the message from the queue and store it in receivedMessage, if the message is received, increment the number of successfully received messages, total no of received messages, if not, free the memory and increment the number of blocked messages, check if reached 1000 message, if not, continue the process, else call the reset function to print the results, reset all counters, clear the queue and configure the values controlling the sender period but not to forget that if all values in the array are used, destroy the timers and print a message “Game Over” then stop execution.
- Therefore, we can say that our system structure is based on:
  - The reset function, three sender tasks of 2 have the same priority and the other higher and one receiver task.
  - Three sender timers and one receiver timer with callback functions.

```
void ReceiverTask(){
BaseType_t status;
```

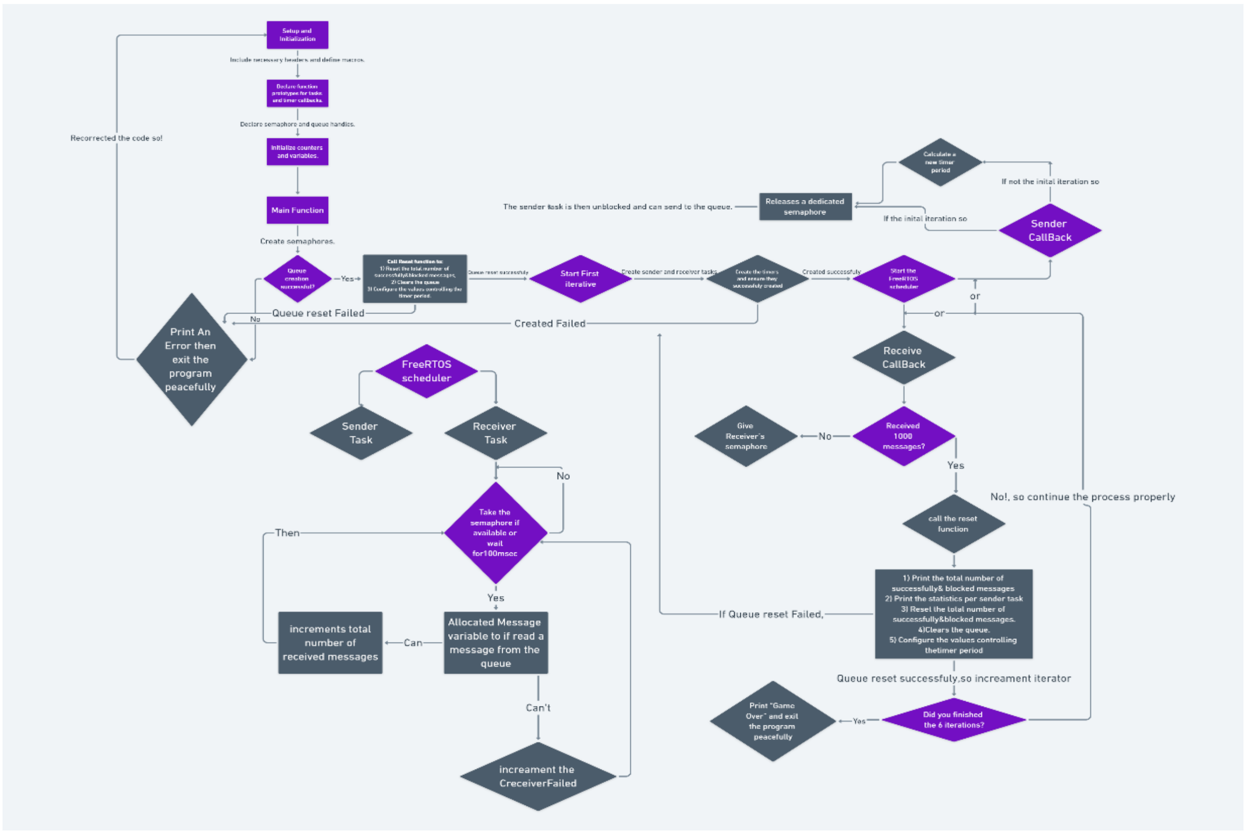
```
while(1){
xSemaphoreTake(MutexResiiver,portMAX_DELAY);     char *receivedMessage=(char *)malloc(15);
status = xQueueReceive( testQueue, &receivedMessage, 0 ); if( status == pdPASS ){
CreceiverSuccss++;    NoOfMessages++;
if(NoOfMessages==1000){ reset(); free(receivedMessage);}  else {CreceiverFailed++;}}}
```

Code snippet 4: Receiver Function



"Time is XYZ" where XYZ is current time in system ticks.
Queue(testQueue)
SenderTask1
SenderTask2
SenderTask3
ReciverTask
xTimerSender1
xTimerSender2
xTimerSender3
xTimerReciver
Enqueue Message "Time is XYZ"
Callback function called
Release semaphore
Task Sleep
Enqueue Message "Time is XYZ"
Enqueue Message "Time is XYZ"
Enqueue Message "Time is XYZ"

Figure 5: Message Sequence



Setup and Initialization
Include necessary headers and define macros.
Declare function prototypes for tasks and timer callbacks.
Declare semaphore and queue handles.
Initialize counters and variables.
Main Function
Create semaphores.
Recorrected the code so!
Queue creation successful?
Yes
No
Call Reset function to: 1) Reset the total number of successfully&blocked messages, 2) Clears the queue 3) Configure the values controlling the timer period.
Queue reset successfuly
Queue reset Failed
Print An Error then exit the program peacefully
Start First iterative
Create sender and receiver tasks
Create the timers and ensure they successfuly created
Created Failed
Created successfuly
Start the FreeRTOS scheduler
or
or
Sender CallBack
If not the inital iteration so
Calculate a new timer period
If the inital iteration so
Releases a dedicated semaphore
The sender task is then unblocked and can send to the queue.
Receive CallBack
Received 1000 messages?
No
Give Receiver's semaphore
Yes
call the reset function
No!, so continue the process properly
1) Print the total number of successfully& blocked messages 2) Print the statistics per sender task 3) Reset the total number of successfully&blocked messages. 4)Clears the queue. 5) Configure the values controlling thetimer period
If Queue reset Failed,
Queue reset successfuly,so increament iterator
Did you finished the 6 iterations?
Yes
Print "Game Over" and exit the program peacefully
FreeRTOS scheduler
Sender Task
Receiver Task
Take the semaphore if available or wait for100msec
No
Then
Yes
Allocated Message variable to if read a message from the queue
Can
increments total number of received messages
Can't
increament the CreceiverFailed

Figure 6: Flowchart of The Workflow

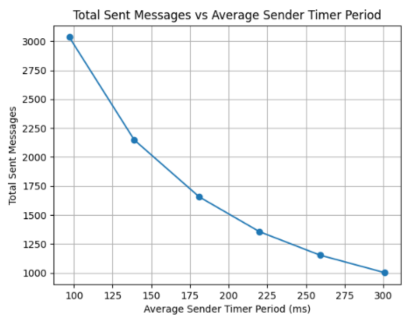
2 Results and Discussion

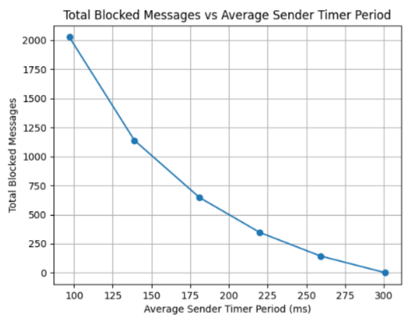
Firstly, we will present the results for queue sizes of 3 and 10 in the table 1. Subsequently, these results will be visualized using Python code, beginning with a queue of size 10 and then repeating the process for a queue of size 2.

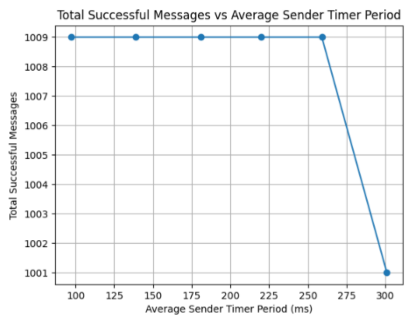
Table 1: The Results of the Queue of size 3 | size 10

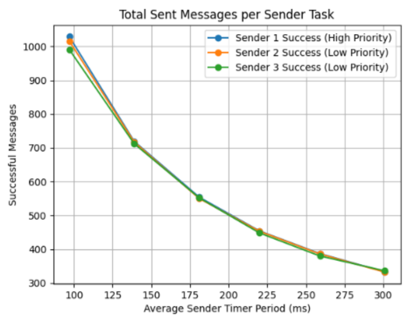
Iterations	Reset 1	Reset 2	Reset 3	Reset 4	Reset 5	Reset 6
successfully sent messages	1002|1009	1002|1009	1002|1009	1002|1009	1002|1009	1001|1001
blocked sent messages	2034|2027	1146|1139	654|647	353|346	153|144	10|0
Sender 1: successfully sent messages	348|350	346|349	336|336	318|344	328|328	331|333
Sender 1: failed sent	657|679	374|370	219|218	136|110	59|59	4|0
Sender 1: Average period	99|97	138|139	180|180	220|220	258|258	303|301
Sender 2: successfully sent	332|328	343|336	321|325	347|340	335|335	334|332
Sender 2: failed sent	688|688	379|381	232|225	104|113	49|51	4|0
Sender 2: Average period	98|98	138|139	180|181	222|220	260|259	301|303
Sender 3: successfully sent	322|331	313|324	345|348	337|325	339|346	336|336
Sender 3: failed sent	689|660	393|388	203|204	113|123	45|34	2|0
Sender 3: Average period	98|100	141|140	182|181	222|223	261|262	300|300

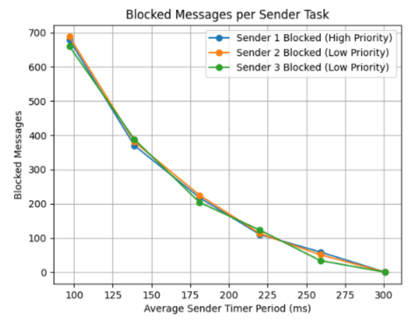
A. Queue of Size 10 Graphs:

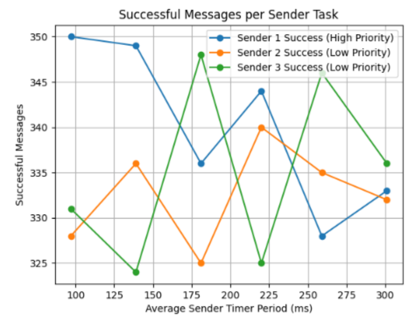


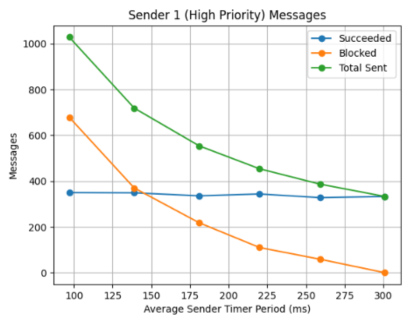


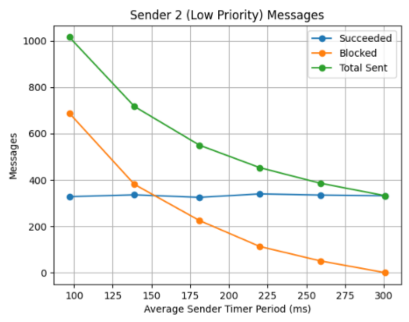


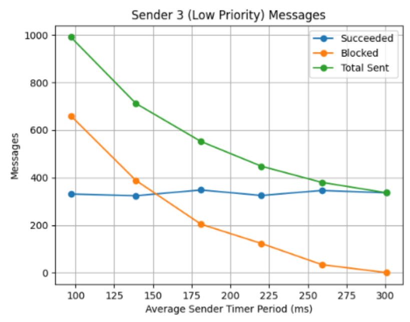




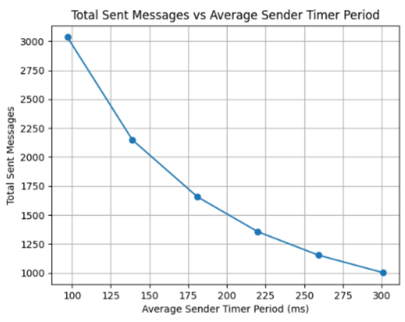


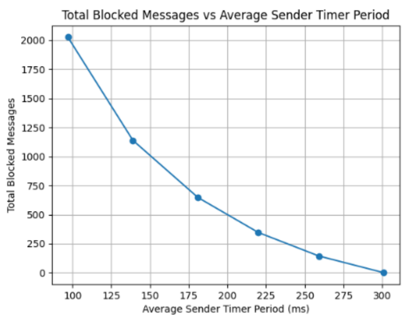


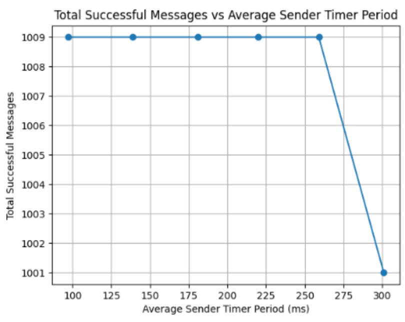


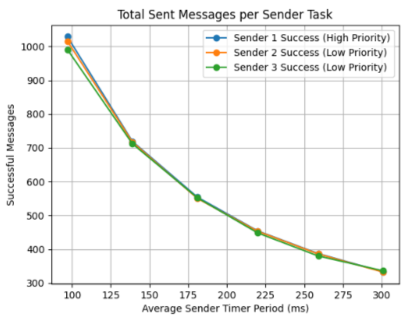


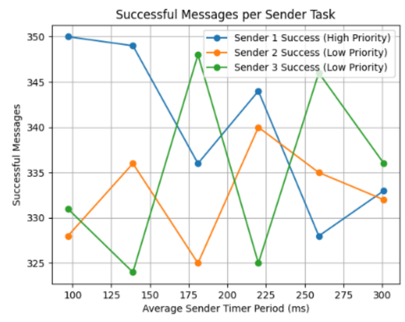
B. Queue of Size 3 Graphs:

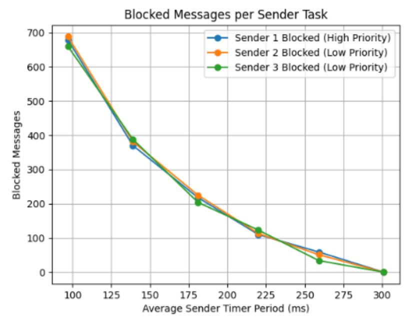


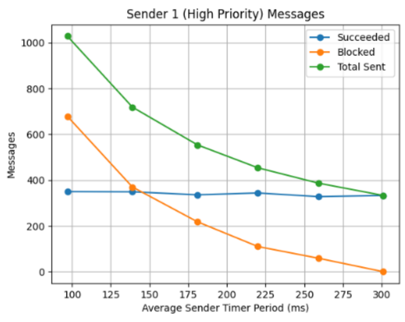


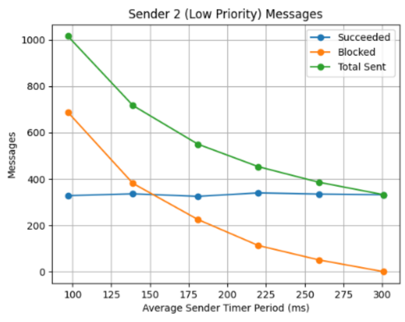


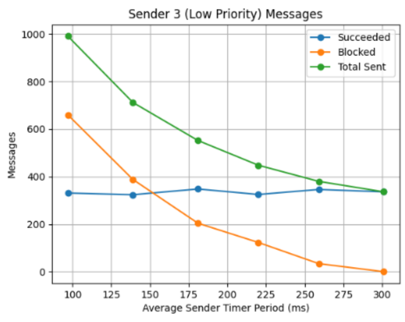












Explanation of the gap between the number of sent and received messages:

As shown in table 1 we notice that the gap at the operating random/fixed period is duo to:

- Queue Size Limitations, which meaning that when the queue is full, additional messages from the sender tasks are blocked until space becomes available meaning until the receiver reads the next message. This causes the sender tasks to fail to send some messages, which increases the count of blocked messages.
- Task Priorities, As shown in figures of "**Successful Messages per Sender Task**" which meaning that the priorities assigned to tasks affect how frequently they are scheduled to run. If receiver tasks have lower priority than sender tasks, the queue can fill up faster, leading to more blocked messages and vice versa.
- Timer Callbacks and Periods and semaphore handling, which meaning that how often each sender task is triggered to send a message. If the periods are too short and sender tasks are triggered too frequently, the queue may not have enough capacity to handle the influx of messages, leading to more blocked messages therefore the randomly generated periods for sender tasks can lead to unequal message generation rates, leading to variability in the number of successfully sent versus received messages.

As shown in Figures of "Total Blocked Messages vs Average Sender Time Period" the gap decreases if the sender timer period increases since it allows the sender task to send messages at a slower rate, reducing the possibility of the queue becoming full so we say that when the receiver' period is becoming smaller than the sender timer, the gap nearly reaches zero so we can say that the receiver task is receiving messages at a faster rate than the sender task, meaning that the possibility of the queue becoming full the lowest eliminating the gap.

Use a queue of size 3, then repeat for a queue of size 10. What happens when queue size increases?

As shown in table 1 and the figures with a queue size of 3 the total number of blocked messages was significantly higher compared to a queue size of 10, we noticed the reduction in blocked messages so we say that a larger queue can hold more messages before becoming full, this means sender tasks are less likely to encounter a full queue, leading to fewer blocked messages.