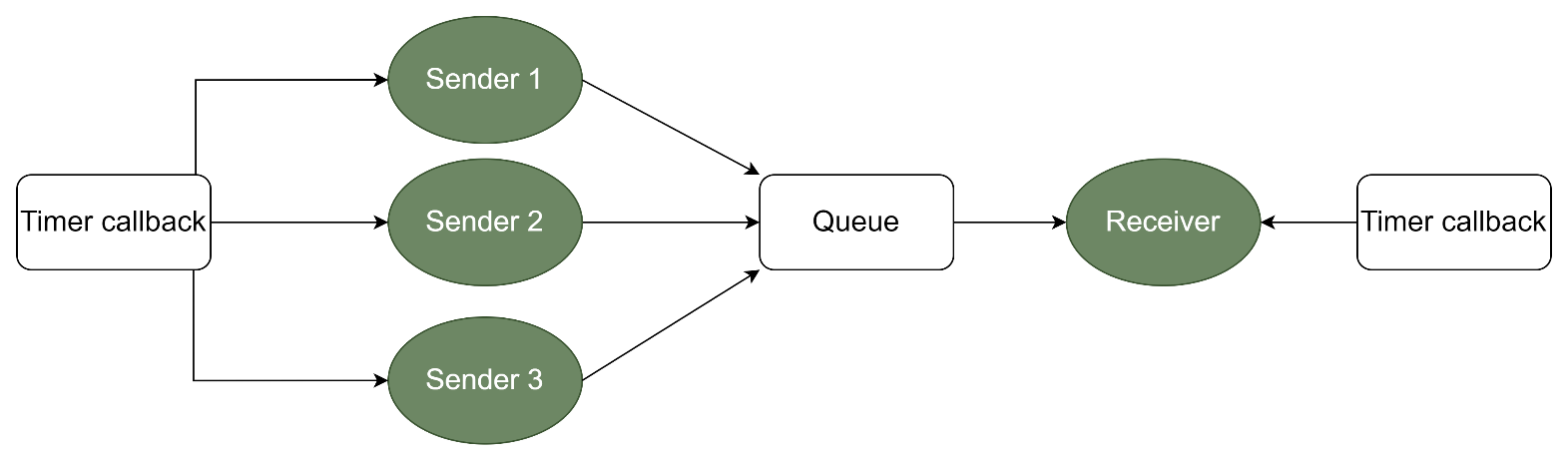
|  |  |  |  |
| --- | --- | --- | --- |
| BN | Section | Student ID | اسم الطالب |
| **11** | **1** | **9210053** | **احمد جمال اسماعيل ابراهيم نورالدين** |
| **18** | **1** | **9210105** | **احمد عبدالله عبدالحميد احمد السعدني** |

**1 System design**

Figure 1: System quick design view



* 1. **Main Structure and functions used with explanation and code snippets**

|  |
| --- |
| **void** Timers\_Destroy (TimerHandle\_t timer1, TimerHandle\_t timer2, TimerHandle\_t timer3, TimerHandle\_t timer4) {  xTimerDelete (timer1, 0);  xTimerDelete (timer2, 0);  xTimerDelete (timer3, 0);  xTimerDelete (timer4, 0);  } |

1.1.1 Timer destroy function

The Timers Destroy function takes four timer handles as input parameters and uses the RTOS API function xTimerDelete to delete these timers. The purpose of this function is to clean up and destroy the specified timers in an RTOS-based application.

|  |
| --- |
| **void** Sender\_1\_timerCallback (TimerHandle\_t xTimer) {  xSemaphoreGive (Sender\_1\_Semaphore);}  **void** Sender\_2\_timerCallback (TimerHandle\_t xTimer) {  xSemaphoreGive (Sender\_2\_Semaphore);}  **void** Sender\_3\_timerCallback (TimerHandle\_t xTimer) {  xSemaphoreGive (Sender\_3\_Semaphore);  } |

1.1.2Sender Callback function

The Sender timer Callback function gives the Sender Semaphore.

1.1.3Receiver callback function

|  |
| --- |
| **void** Receiver\_timerCallback (TimerHandle\_t xTimer) {  xSemaphoreGive (Receiver\_Semaphore);  } |

The Receiver Timer Callback function gives the Receiver Semaphore.

* + 1. sender function

|  |
| --- |
| **void** Sender\_1\_function (**void** \*ptr) {  **unsigned long** **int** current\_time\_in\_ticks;  **char** sent\_message [50];  **while** (1) {  xSemaphoreTake (Sender\_1\_Semaphore, portMAX\_DELAY);  current\_time\_in\_ticks = (**unsigned long int**) xTaskGetTickCount ();  sprintf (sent\_message, "Time is %lu", current\_time\_in\_ticks);  BaseType\_t isSent = xQueueSend (Queue, &sent\_message, 0);  **if** (isSent == pdPASS) {  transmitted\_messages [0] ++;}  **else**{  blocked\_messages [0] ++;}  TickType\_t random\_period = Random\_Period\_Generator ();  xTimerChangePeriod (Sender\_1\_Timer, random\_period, 0);  }  } |

The sender function is used to check if the semaphore is available or not, if it's available it sends a message to queue and check if sent it will increment the transmitted messages counter if not it will increment the blocked messages counter.The semaphore will be available after timer call back function calls. The task will be blocked when the queue is full so the receiver will not receive the data.

* + 1. Receiver function

|  |
| --- |
| **void** Receiverfunction (**void** \*ptr) {  **char** received\_message [50];  **while** (1) {  xSemaphoreTake (Receiver\_Semaphore, portMAX\_DELAY);  **if** (uxQueueMessagesWaiting (Queue) == 0) {}  **else** {  xQueueReceive (Queue, received\_message, 0);  printf ("%s \n", received\_message);  number\_of\_received\_message++;}  **if** (1000 <= number\_of\_received\_message) {  Reset\_Function ();}}} |

The receiver function checks if the semaphore is available or not, if it's available it receives message from queue and print it and increment the received messages counter.The semaphore will be available after timer call back function calls. The task will be blocked when the queue is empty for long time.

|  |
| --- |
| TickType\_t Random\_Period\_Generator () {  TickType\_t Tsender;  **int** random\_value = rand () %(100+20\*iteration\_number+1) + (50+30\*iteration\_number);  Tsender = pdMS\_TO\_TICKS (random\_value);  **return** Tsender;  } |

* + 1. Random function

The Random\_Period\_Generator function generates a random period duration based on the iteration\_number variable. It uses the rand () function, and a specific formula to calculate the random value within the upper and lower bounds given for the six iterations. The random value is then converted to the equivalent tick count and returned as a result.

|  |
| --- |
| **void** Reset\_Function () {  printf ("This is iteration %i \n", 1+iteration\_number);  **int** total\_blocked=blocked\_messages [0] +blocked\_messages [1] +blocked\_messages [2];  **int** total\_transmitted=transmitted\_messages[0]+transmitted\_messages[1]+transmitted\_messages[2];  printf ("Total number of successfully sent messages = %i \n", total\_transmitted);  printf ("Total number of blocked messages = %i \n", total\_blocked);  **for** (**int** counter = 0; counter < 3; counter++) {printf ("Sender Task %i:\n", (1 + counter));  printf ("Number of successfully sent messages = %i \n", transmitted\_messages[counter]);  printf ("Number of blocked messages = %i \n", blocked\_messages[counter]);}  iteration\_number++; **if** (6 == iteration\_number) {printf ("Game Over \n");  Timers\_Destroy (Sender\_1\_Timer, Sender\_2\_Timer, Sender\_3\_Timer, Receiver\_Timer);  vTaskEndScheduler (); exit (0);} number\_of\_received\_message = 0;  **for** (**int** counter = 0; counter < 3; counter++) {  blocked\_messages[counter] = 0;}  **for** (**int** counter = 0; counter < 3; counter++) {  transmitted\_messages[counter] = 0;}  xQueueReset (Queue);} |

* + 1. Reset function

The Reset Function is responsible for displaying statistics related to message transmission and blocking, resetting various counters and states, and controlling the termination of the program based on the number of iteration. It also includes the cleanup and destruction of timers using a custom Timers Destroy function.

* + 1. Main function

Within this function, we initiate the creation of a queue, tasks, semaphores, and timers. Subsequently, we invoke the initialization function to set up the system, and finally commence the task scheduler to begin execution.

* + 1. Queue

queue is a data structure we used to store the message sent from the three-sender task to be sent to the receiver task due to the absence of direct communication between individual tasks. the queue has a specific size that determines the number of messages stored in it.

* 1. **system flowchart design**

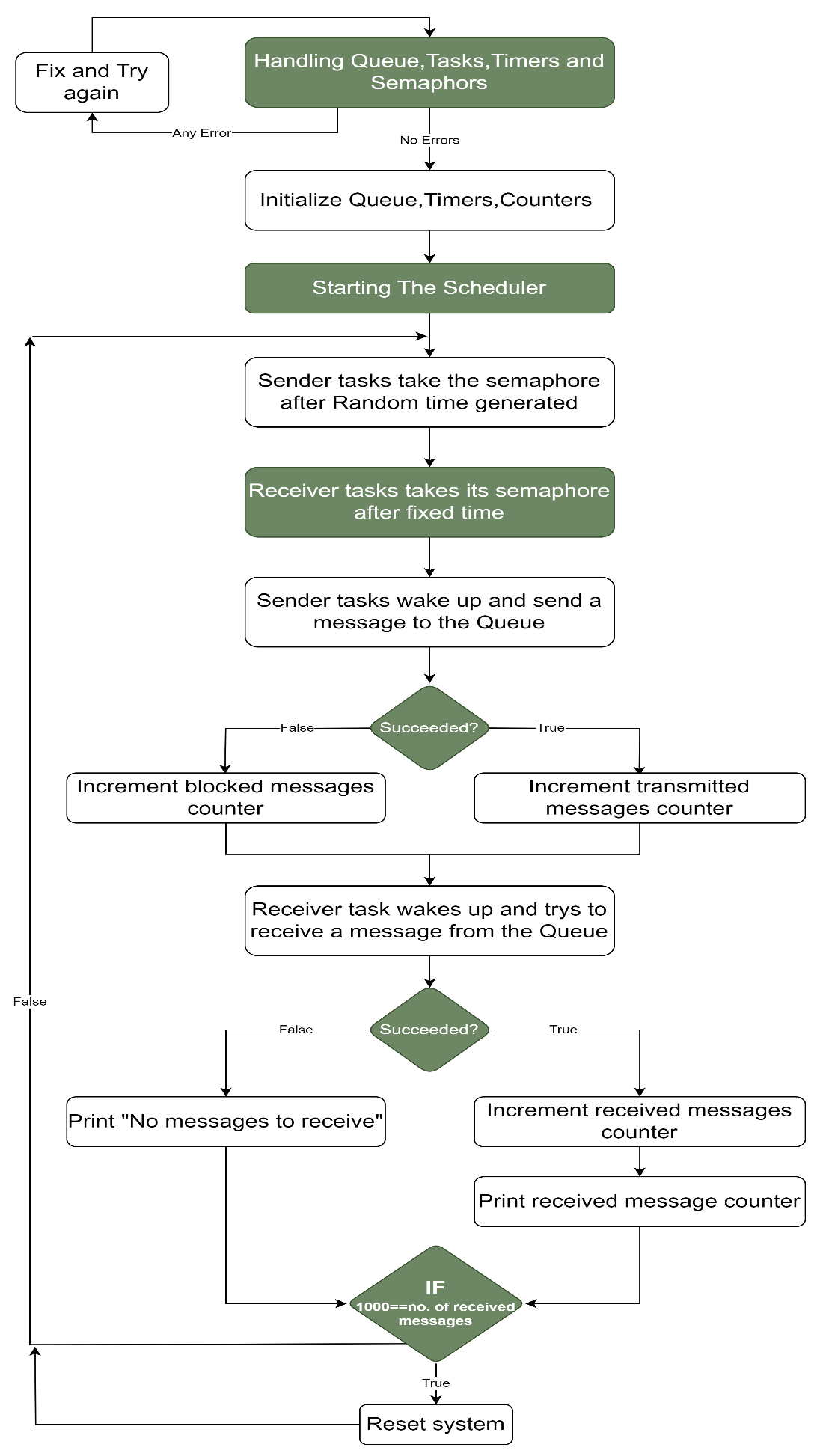
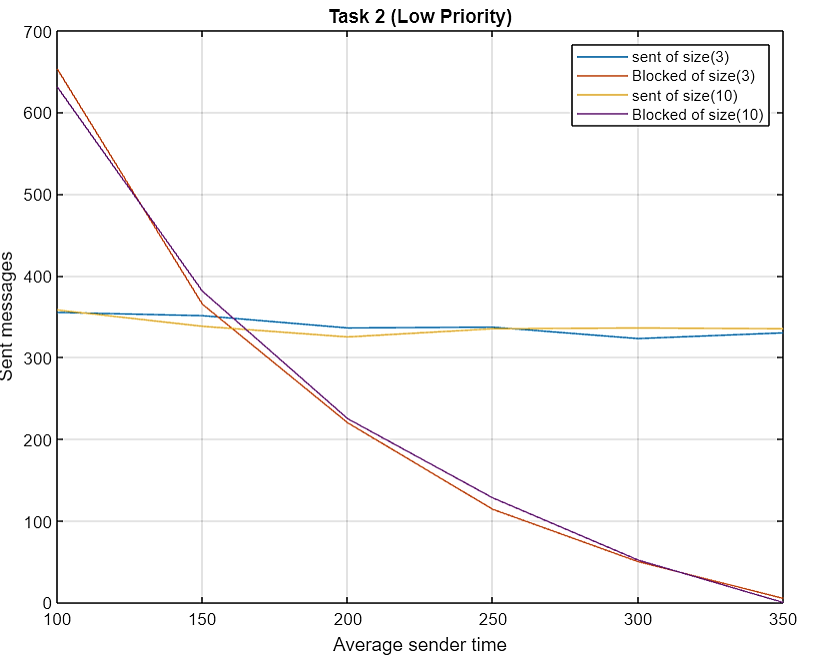
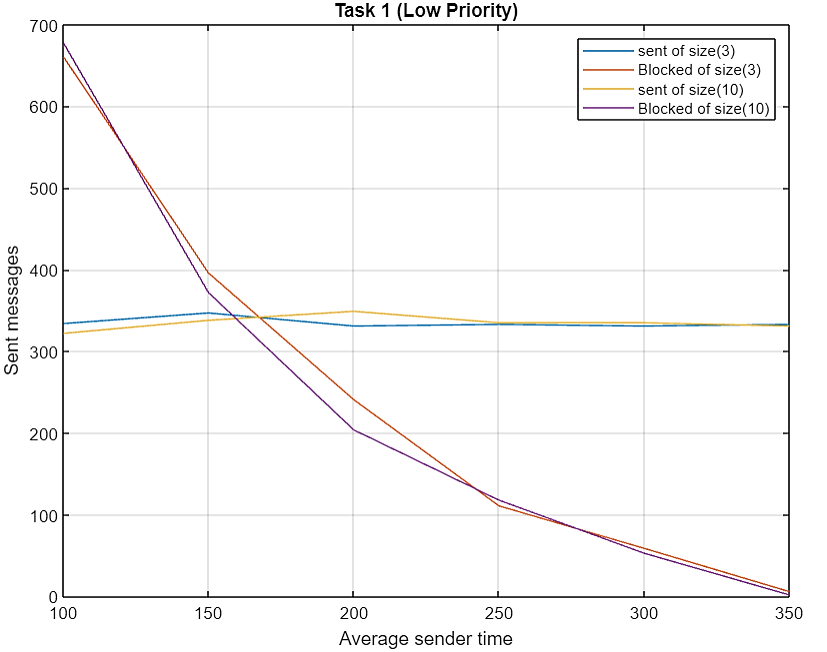
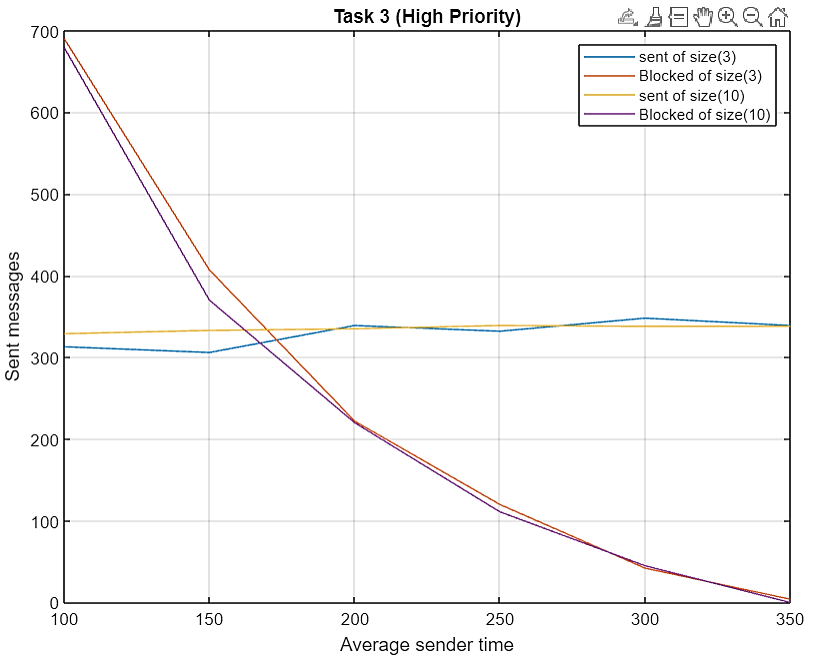


Figure 2: System overall design view

1. **Results and Conclusion**

What happens when queue size increases?

As the queue size expands, the number of successfully transmitted messages grows while the occurrence of blocked or failed messages decreases. This is due to the absence of accumulation in the queue when compared to a smaller queue size. the larger queue allows for more vacant spaces, allowing the receiver task to be able to receive these transmitted messages.