BN	Section	Student ID	اسم الطالب
10	**2**	**9221266**	**رنا محمد محمود المغربل**
1	**4**	**9220812**	**مريم احمد حامد**

1- System Design

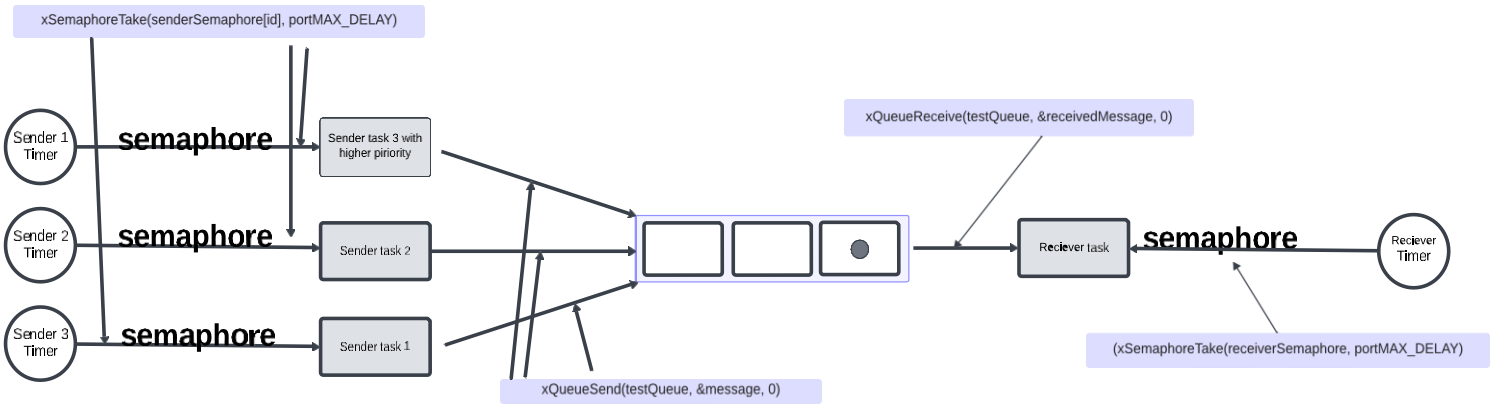


Figure 1: Data Structure Design

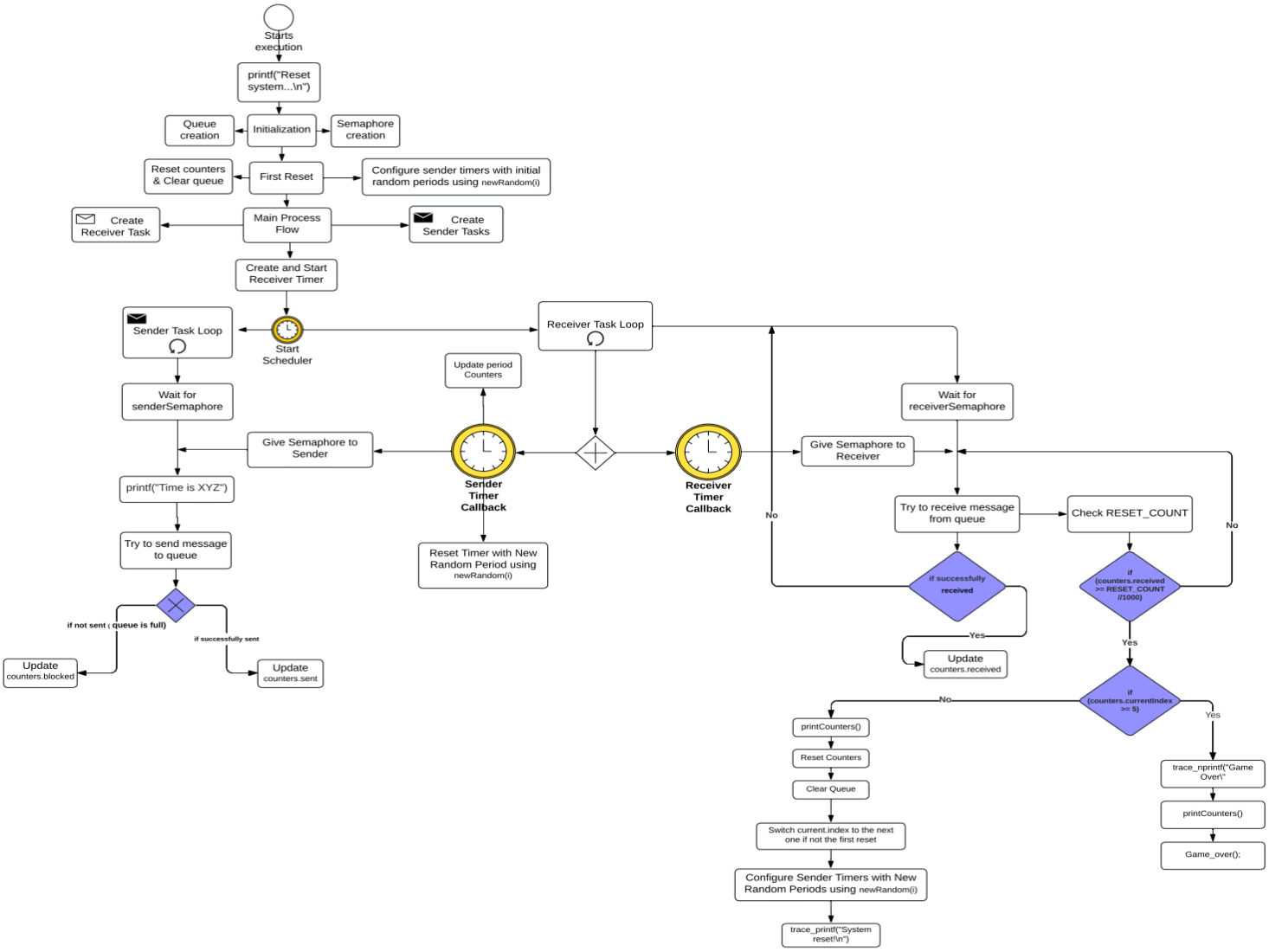


Figure 2: BPMN process flow

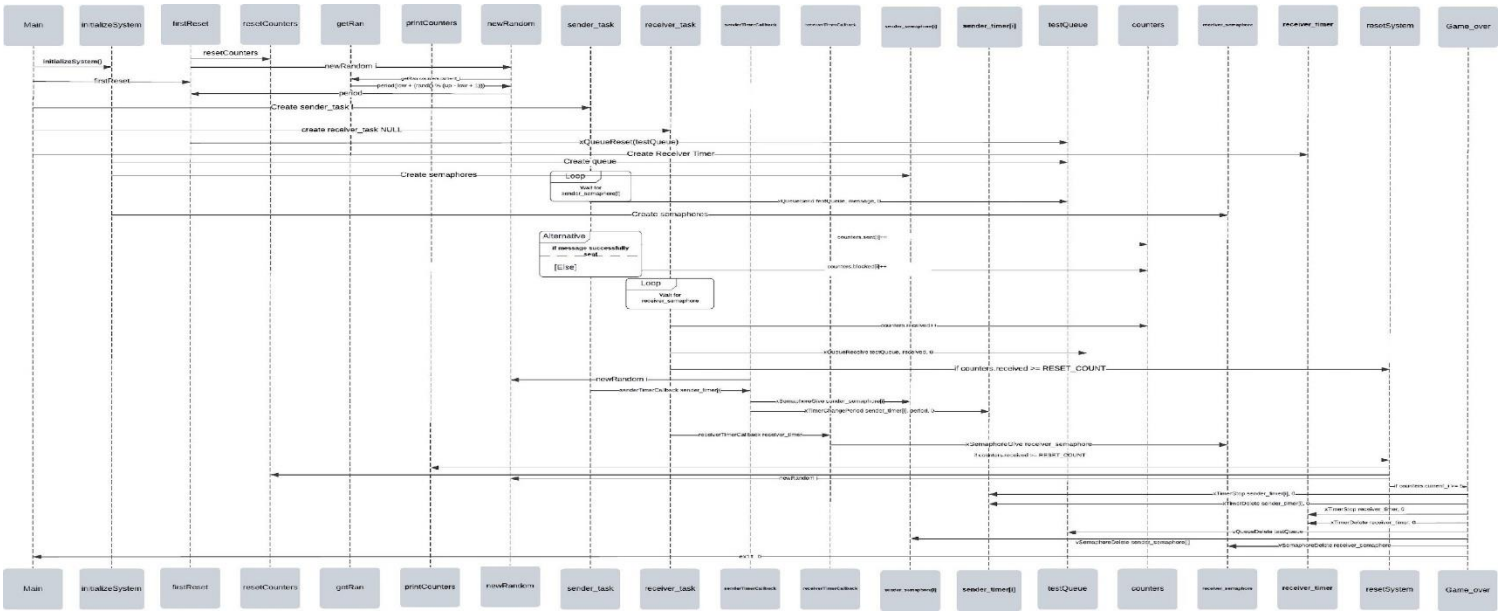


Figure 3: Message Sequence showing object interaction

2-Results and Discussion

1-Queue Size 3

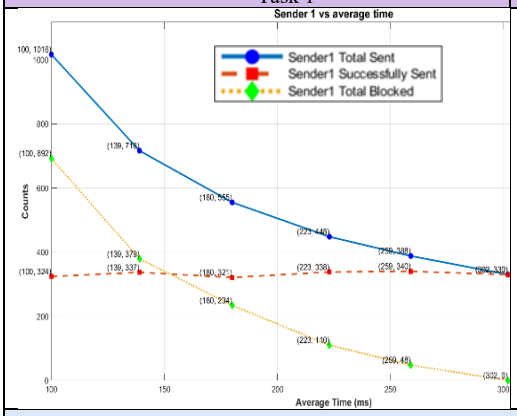
Iteration Number	Average Time	Total Succes_S	Total Sent	Total Blocked	Sender1 Success.	Sender1 Total	Sender 1 Blocked	Average Send1Time
0	98	1002	3033	2031	330	1010	680	100
1	139	1002	2148	1146	321	712	391	139
2	181	1002	1656	654	324	549	223	180
3	221	1002	1354	352	318	452	132	223
4	260	1002	1154	152	324	384	60	259
5	301	1001	1011	10	330	335	5	302
Iteration Number	Sender2 Success.	Sender2 Total Sent	Sender 2 Blocked	Average Send2Time	Sender 3 Success.	Sender 3 Total Sent	Sender 3Blocked	Average Send3Time
0	330	1018	696	98	350	1005	655	98
1	345	714	369	139	336	722	386	139
2	342	555	213	181	336	552	216	181
3	330	449	119	221	354	453	99	221
4	336	383	47	261	342	387	45	260
5	338	339	1	302	333	337	4	302

1-Queue Size 10

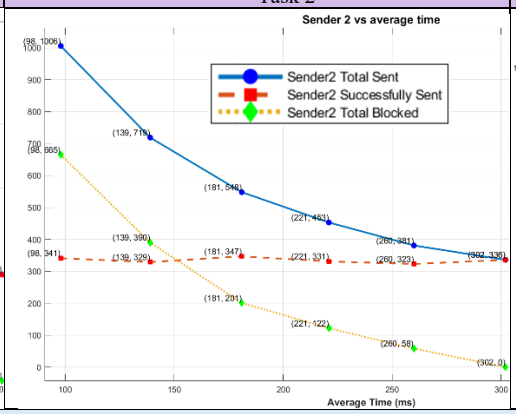
Iteration Number	Average Time	Total-Succes_S	Total Sent	Total Blocked	Sender1 Success.	Sender1 Total	Sender 1 Blocked	Average Send1Time
0	98	1009	3033	2024	324	1016	692	98
1	139	1009	2148	1139	337	716	379	139
2	181	1009	1656	647	321	555	234	181
3	221	1009	1355	346	338	448	110	221
4	260	1009	1154	145	340	388	48	260
5	302	1001	1001	0	330	330	0	301
Iteration Number	Sender2 Success.	Sender2 Total Sent	Sender 2 Blocked	Average Send2Time	Sender 3 Success.	Sender 3 Total Sent	Sender 3Blocked	Average Send3Time
0	341	1006	665	98	344	1011	667	98
1	329	719	390	140	343	713	370	139
2	347	548	201	181	341	553	212	181
3	331	453	122	221	340	454	114	220
4	323	381	58	260	346	385	39	260
5	336	336	0	301	335	335	0	300

Queue Size 3

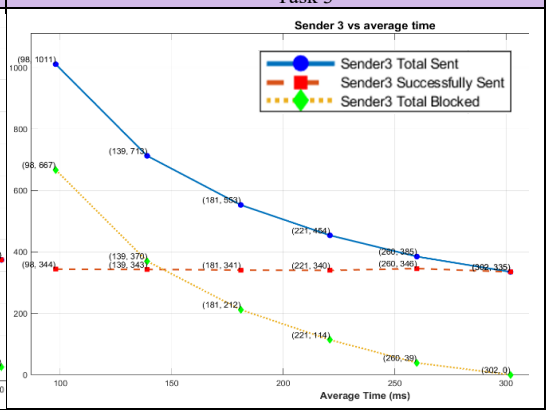
Task 1



Task 2

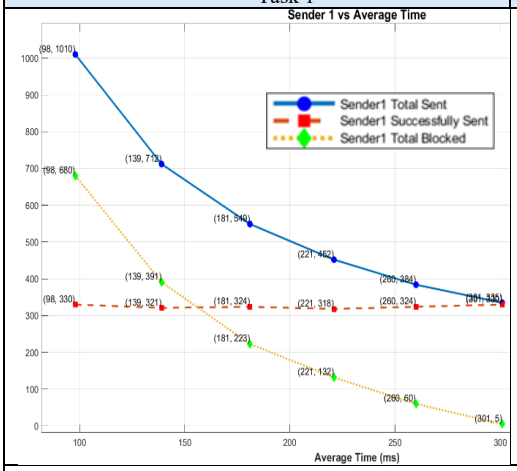


Task 3

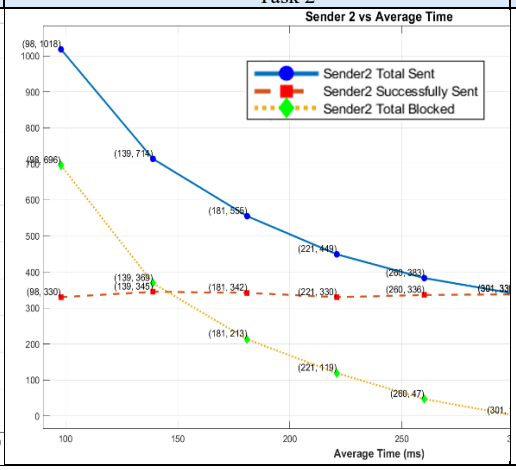


Queue size 10

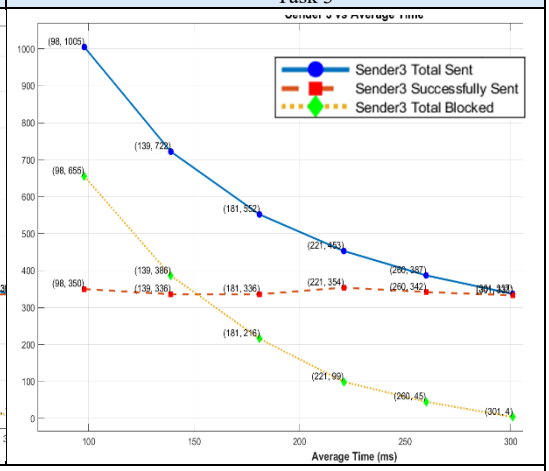
Task 1



Task 2

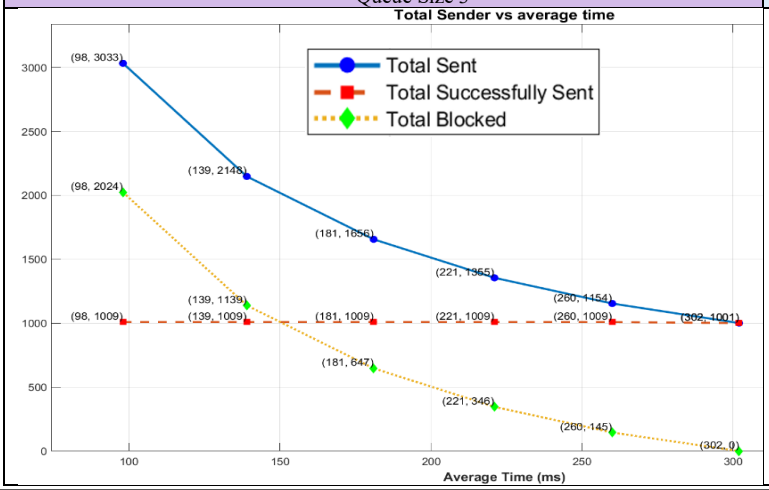


Task 3

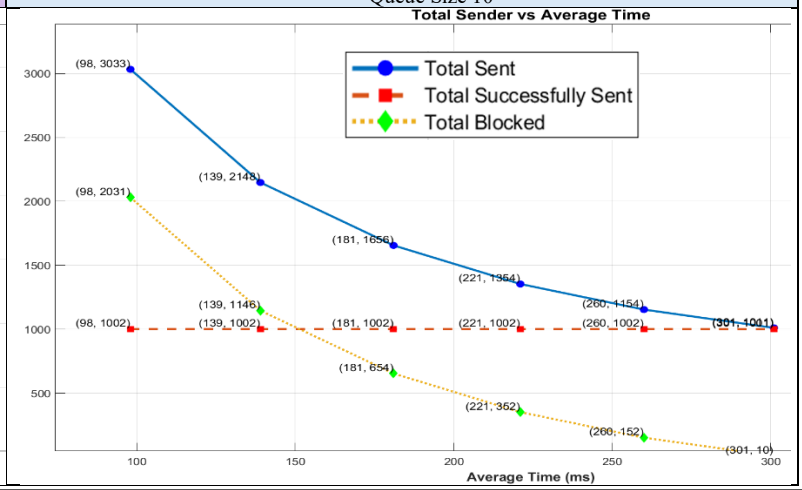


Total send

Queue Size 3



Queue Size 10



Explain the gap between the number of sent and received messages in the running period.

The gap is because the number of sent is not equal the number of recieved because:

-If the queue is full ,any message the sender tries to send will get blocked.

-The receiver can't recieve more than 1000 messages .

So ,the recieved message could equal successfully sent not the total send (which include blocked)

And by looking to results table ,By increasing the average period the blocked messages decreases because the sender tasks attempt to send messages less frequently therefore the queue is less likely to become full and with lower blocked messages, the gap decreases.

What happens when queue size increases?

By observing the tables when queue size increases from 3 to 10 ,the number of successfully sent messages increases as there is more space space in the queue and takes more time to be full therefore the total blocked messages decreases and by increasing the period it could reach zero

REFERENCES:

[1] FreeRTOS API categories. FreeRTOS. (2022, January 26). https://www.freertos.org/a00106.html

2.3 Code Snippets

```
int main(void) {
    trace_printf("Reset system...\n");
    initializeSystem();// create queue & semaphores
    firstReset(); // Call firstReset at the beginning
   // Create sender tasks
    for (int i = 0; i < 3; i++) {
    xTaskCreate(sender_task, "Sender", 1000, (void*)i, (i == 2) ? 2 : 1,
NULL);  }
    // Create receiver task
    xTaskCreate(receiver_task, "Receiver", 1000, NULL, 3, NULL);
    // Create and start receiver timer
    receiver_timer = xTimerCreate("Receiver Timer",
pdMS_TO_TICKS(RECEIVER_PERIOD_MS), pdTRUE, NULL, receiverTimerCallback);
    xTimerStart(receiver_timer, 0);
    vTaskStartScheduler();
    return 0; }
```

The code begins by resetting the system and initializing necessary components. It creates three sender tasks with varying priorities, followed by a receiver task. A timer is set to repeatedly trigger the receiver task. Finally, it starts the task scheduler to manage task execution.

```
void initializeSystem() {
    // Create queue
    testQueue = xQueueCreate(QUEUE_SIZE,
sizeof(char[20]));
    // Create semaphores
    for (int i = 0; i < 3; ++i) {
        sender_semaphore[i] =
xSemaphoreCreateBinary(); }}
receiver_semaphore =
xSemaphoreCreateBinary();}
```

The code snippet creates a queue testQueue. It also creates binary semaphores for three senders and one receiver, which are used to manage task synchronization. Then, the first reset clear all counters and the queue setting them all to zero and intializing sender tasks timers

```
void firstReset() {
    // Reset counters
    resetCounters();
    // Clear queue
    xQueueReset(testQueue);
    // Configure sender timers with initial
random periods
    for (int i = 0; i < 3; i++) {
        sender_timer[i] =
xTimerCreate("Sender timer",
pdMS_TO_TICKS(newRandom(i)), pdTRUE,
(void*)i, senderTimerCallback);
        xTimerStart(sender_timer[i], 0) }}
```

```
uint32_t newRandom(uint8_t i){
    uint32_t period= getRan(counters.current_i);
//counters.periods[i] = period; // Store the
period
    counters.total_period += period; // Add
period to total sum
    counters.count_period++;
    counters.count_period_task[i]++;
    counters.periods[i] += period; //Update
period
    return period;}
```

```
uint32_t getRan (int i) {
    // Get a random period between lower and
upper bounds
    uint32_t low = lower[i];
    uint32_t up = upper[i];
    return low + (rand() % (up - low + 1)); //
using uniform distribution}
```

The newRandom function generates a random period for a given task using getRan, which returns a random period within specified bounds. The function updates the total period sum, the overall period count, and the specific task's period count. It also updates the task's period and returns the new period. The getRan function calculates a random period between the specified lower and upper bounds using a uniform distribution.

```
void senderTimerCallback(TimerHandle_t xTimer) {
    int i = (int)pvTimerGetTimerID(xTimer);
    xSemaphoreGive(sender_semaphore[i]);  // Give semaphore
to sender
xTimerChangePeriod(xTimer, pdMS_TO_TICKS(newRandom(i)), 0);
// Reset the timer with a new random period }
```

```
void receiverTimerCallback(TimerHandle_t xTimer) {
    xSemaphoreGive(receiver_semaphore);  // Give semaphore to
receiver
    (void)xTimer; // To avoid unused parameter warning}
```

The senderTimerCallback function gives the semaphore to the sender task and resets the timer with a new random period. It retrieves the timer ID, gives the semaphore, and changes the timer period using xTimerChangePeriod. The receiverTimerCallback function gives the semaphore to the receiver task.

```
void sender_task(void *parameters) {
    int i = (int)parameters;
    char message[20];
    BaseType_t xStatus;
    while (1) { // Wait for the semaphore
        if (xSemaphoreTake(sender_semaphore[i], portMAX_DELAY)
== pdTRUE) {
            snprintf(message, sizeof(message), "Time is %lu",
xTaskGetTickCount());
            // Successfully received
          trace_printf("Time is %lu\n", xTaskGetTickCount());
          // Try to send the message to the queue
            xStatus = xQueueSend(testQueue, &message, 0);
            if (xStatus != pdPASS) { // Failed to send
                counters.blocked[i]++;
            } else {  // Successfully sent
                counters.sent[i]++; }}}}
```

The sender_task function continuously waits for a semaphore. Once it successfully takes the semaphore, it creates a message with the current tick count and attempts to send it to the queue. If the message fails to send, it increments the blocked counter for that sender. If successful, it increments the sent counter for that sender. The function includes trace printing to log the current tick count.

```
void receiver_task(void *parameters) {
    char received[20];
    BaseType_t xStatus;
    while (1) {  // Wait for the semaphore
        if (xSemaphoreTake(receiver_semaphore, portMAX_DELAY)
== pdTRUE) {
            xStatus = xQueueReceive(testQueue, &received, 0);
            if (xStatus == pdPASS) {
                counters.received++;
            } else { // Failed to receive}
            if (counters.received >= RESET_COUNT) {
resetSystem();}}}
    (void)parameters; // Avoid unused parameter warning }
```

The receiver_task function continuously waits for a semaphore. When it successfully takes the semaphore, it attempts to receive a message from the queue. If successful, it increments the received counter. If the counter reaches the RESET_COUNT threshold, it calls the resetSystem function.

```
void resetSystem() {
      if (counters.current_i >= 5) {
          trace_printf("Game Over\n");
          printCounters();
          Game_over(); // End the program }
      else {    printCounters(); // Print counters
                uint32_t temp = counters.current_i;
               resetCounters(); // Reset counters
                xQueueReset(testQueue); // Clear queue
                // Switch current.index to the next one if not the first reset
                counters.current_i = temp + 1;
                // Configure sender timers with new random periods
                for (int i = 0; i < 3; i++) {
                    xTimerChangePeriod(sender_timer[i], pdMS_TO_TICKS(newRandom(i)), 0);
                    xTimerStart(sender_timer[i], 0); }
                trace_printf("System reset!\n"); }}
```

```
void Game_over(){
      xTimerStop(sender_timer[0],0);
       xTimerStop(sender_timer[1],0);
       xTimerStop(sender_timer[2],0);
       xTimerStop(receiver_timer,0);
       xTimerDelete(sender_timer[0],0);
       xTimerDelete(sender_timer[1],0);
       xTimerDelete(sender_timer[2],0);
       xTimerDelete(receiver_timer,0);
       vQueueDelete(testQueue);
       vSemaphoreDelete(sender_semaphore[0]);
       vSemaphoreDelete(sender_semaphore[1]);
       vSemaphoreDelete(sender_semaphore[2]);
       vSemaphoreDelete(receiver_semaphore);
       exit(0);}
```

The resetSystem function checks if the current_i counter is 5 or more. If it is, it prints "Game Over," shows the counters, and calls Game_over to end the program. If not, it prints and resets the counters, clears the queue, increments current_i, reconfigures, and restarts the sender timers with new random periods, then prints "System reset!" The Game_over function stops and deletes all timers, deletes the queue and semaphores, and exits the program.