



# FreeRTOS based Windows Simulator

Project Report

Embedded System Design (Group 2)

---

Sadia Asif  
Farheen Asif  
M Usamah Shahid

# Table of Contents

<b>Problem Statement</b>	<b>2</b>
<b>Hardware/ Software Used</b>	<b>2</b>
<b>Methodology</b>	<b>2</b>
Data Acquisition:	2
Control Unit:	2
Manipulation:	3
Periodic printing:	3
User Input:	3
<b>Communication and Synchronisation</b>	<b>3</b>
Data Structures:	3
Queues:	4
Semaphores:	4
<b>Workflow</b>	<b>4</b>
<b>Task Descriptions</b>	<b>5</b>
vTaskPrint()	5
vTaskStats()	5
vTaskAlert()	6
vTaskWrite()	6
vTaskRandSender()	7
vTaskController()	7
<b>Output Description:</b>	<b>7</b>
Sender Task	7
Periodic Task	8
Alert Task	9
Write Task	10
<b>Conclusion:</b>	<b>10</b>

## Introduction

FreeRTOS is the real time operating system for many embedded devices. It provides various tasks, semaphores and software timers for a varying number of applications with less power mode. Compatibility of FreeRTOS with large no. of microcontrollers makes it highly demanding for IoT based and other applications. Queues in FreeRTOS provide the easiest way of inter task communication.

In this project, we have performed windows based simulation of FreeRTOS using STM32 and various functions for Queues, semaphores, timers and tasks that are explored and implemented in this project for data acquisition, periodic interrupts and for other controlled functions. Details of each module used in this project is explained in this report.

## Problem Statement

To apply the basic free RTOS concepts to three different sensor values(assumed in this case), manipulate them and take the user input for run time value set.

## Hardware/ Software Used


Visual studio was used for setting the FreeRtos setup and all c based coding. As it is just a simulation based project so no hardware was used for this project.

## Methodology

The basic methodology of the project is the division of the work in different modules. There are five steps for the accomplishment of the tasks. The four steps are given below:

- Data Acquisition
- Control Unit
- Manipulation
- Periodic Printing
- User Input

## Data Acquisition:



In this first step after all the initialization, there are three data sources or sensor values resources. The three sender tasks as given in the diagram which show this step. They have been given different value ranges and after selecting a random value from the specific range, the sender tasks send the values to the control unit.

## **Control Unit:**

Control Unit is managed by the controller function/task in the project. This function receives all the values from the relevant sender functions, identifies the sender function and sends the values to further relevant tasks to perform statistical operations. Also the controller function also displays the name of the sender task and the value sent by the tasks on the console. Further processing and description of the task have been given below. This module has been shown in the work flow diagram as the controller unit.

## **Manipulation:**

In this step, the values which have been sent by the control hub are processed in different functions. Stats Task and Alert Task both lie in this module. Three tasks of the stats function are to count the total number of values coming so far, keep track of the values which have been coming, take the mean of all the values and find the moving average of the last five values. Receiving data from the queue and writing back are all in this step. For the alert task the track of the task 2 values has been kept to send a warning that repetitively a value has been sent three times.

## **Periodic printing:**

This printing task is a periodic function which displays the results of moving average, mean and total count after 5 seconds.

## **User Input:**

This is the writing task which takes the input from the user to change the value of the task 3 constant number.

## **Communication and Synchronisation**

All inter-task communication is achieved using queues of various lengths and data types. These queues and the console are thus considered as “shared resources” and protected using

dedicated semaphores i.e. each call to either of these resources is sandwiched between lines that take and give the relevant semaphore.

The use of queues is assisted in some cases by defining data structures so a single item in a queue can carry a bundle of meaningful, interdependent or similar data. A brief introduction to these items is provided.

## Data Structures:

- DataSource\_t: Enumerated data type containing 3 values to represent the sending tasks
- Data\_t: Data structure consisting of an integer value, and a value of type DataSource\_t
- Stats\_t: Contains three integer values, each represents a different statistical parameter

## Queues:

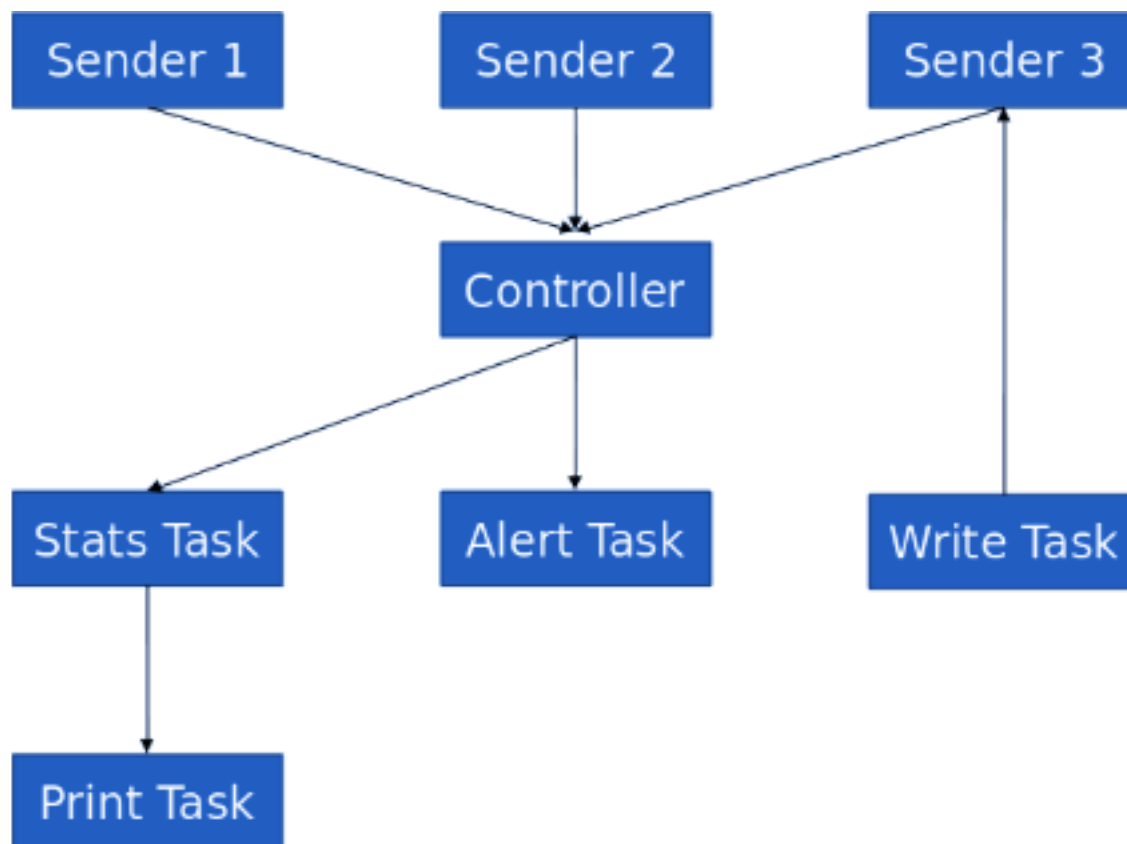
- senderQueue: of size 3 and data type Data\_t
- alertQueue: of size 3 and integer data type
- statsQueue: of size 1 and data type Stats\_t
- writeQueue: of size 1 and integer data type
- movingAveQueue: of size 5 and integer data type

## Semaphores:

- qCountingSem: Counting semaphore of value 2 for senderQueue
- consoleBinarySem: Binary semaphore for the console
- alertqBinarySem: Binary semaphore for the alertQueue
- statsqBinarySem: Binary semaphore for the statsQueue
- movAveqBinarySem: Binary semaphore for the movingAveQueue
- writeqBinarySem: Binary semaphore for the writeQueue

## Workflow

The work-flow diagram is given below with each row representing a different module.



Further the description of different tasks is as follows.

## Task Descriptions


### vTaskPrint()

This is the periodic function that prints statistics from task 1

- Reads from: statsQueue
- Writes to: Console

Periodicity is achieved using the vTaskDelayUntil() function, which blocks the function for 5 seconds from the last wake time.

### vTaskStats()



This task keeps track of the count and mean of all values received from task 1 so far, along with the moving average over a window of 5 values.

- Reads from: movingAveQueue
- Writes to: statsQueue

The task reads all five values from the queue, so it checks if the queue is full beforehand. It updates the values for sum and count for only the first value read (earliest value in the queue). It stores the remaining four values in an array.

After calculating the moving average, the four values in the array are written back to the queue, so only the earliest value is removed and the queue is ready to receive the latest value.

## **vTaskAlert()**

This task prints an alert if the last 3 values sent by task 3 are the same

- Reads from: alertQueue
- Write to: console

After ensuring that the queue is full, the task reads two values and peeks at the third. Peeking does not remove the value from although it has been copied into a variable. After comparison, the send value read is written to the front of the queue, so the incoming order is preserved.

## **vTaskWrite()**

This task updates the constant value sent by task 3 with a user input

- Reads from: writeQueue
- Writes to: writeQueue

The task checks for a key press by the user after which it obtains the binary semaphore for the console and prints a prompt. Even though other tasks keep printing on the console, the value entered by the user over time is accurately read and stored in the queue. The updated value can be seen printing after pressing Enter

## **vTaskRandSender()**

This is the random number generator task, simulating our sensors

- Reads from: writeQueue
- Writes to: senderQueue

As this function is a template for 3 tasks, it identifies the ID and the range for the random number to generate from the parameters passed. If the minimum and maximum values passed are the same, it sends a constant value read from the queue.

## **vTaskController()**

This task acts as our hub, receiving all data and sending it to relevant tasks

- Reads from: senderQueue
- Writes to:
  1. movingAveQueue for task 1
  2. alertQueue for task 2
  3. console

The task uses the first value in the data structure to identify the sending task, and simple If .. else statements to write to the relevant queue. It prints both the value and the ID of the sending task to show the synchronization achieved by our program.

## **Code:**

```
/* Standard includes. */
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
/* Kernel includes. */
#include "FreeRTOS.h"
#include "task.h"
```



```
#include "semphr.h"

#define sendFrequency pdMS_TO_TICKS(100)    /*This is the data rate of all three
simulated sensors*/

/*Enumerated data type to identify what task is sending the data*/
typedef enum
{
    vTaskRandSender1,
    vTaskRandSender2,
    constSender3
} DataSource_t;

/* Define the structure type that will be passed on the sender queue.*/
typedef struct
{
    uint8_t randVal;
    DataSource_t eDataSource;
} Data_t;

/*The data structure to pass on the printing queue used by the periodic printing task*/
typedef struct
{
    uint32_t count;
    uint32_t mean;
    uint32_t movAve;
} Stats_t;

/*Initialise queue and semaphore handles*/
QueueHandle_t senderQueue; /*The main queue used for generating tasks to send data*/
```

```

QueueHandle_t alertQueue;           /*Queue to hold the last 3 values sent by task 2*/
QueueHandle_t statsQueue;           /*Queue to hold a single value of type Stats_t for printing*/
QueueHandle_t movingAveQueue;       /*Queue to hold the last 5 values sent by task 1*/
QueueHandle_t writeQueue;           /*Queue to change value sent by task 3 */

SemaphoreHandle_t qCountingSem;      /*Counting semaphore for Sender queue*/
SemaphoreHandle_t consoleBinarySem;  /*Binary semaphore for console. Taken
before each print statement*/
SemaphoreHandle_t alertqBinarySem;   /*Binary semaphore for alertQueue*/
SemaphoreHandle_t statsqBinarySem;   /*Binary semaphore for statsQueue*/
SemaphoreHandle_t movAveqBinarySem; /*Binary semaphore for movingAverageQueue*/
SemaphoreHandle_t writeqBinarySem;   /*Binary semaphore for writeQueue*/

/*Periodic task to print statistics of task 1*/
void vTaskPrint(void* pvParameters) {
    vTaskDelay(pdMS_TO_TICKS(5000)); /*Block for first 5 seconds*/
    Stats_t receivedStats;            /*To receive data structure to print*/
    TickType_t xLastWakeTime; /*To record the start time of when this function is
unblocked*/
    for (;;) {
        xLastWakeTime = xTaskGetTickCount(); /*Record time of unblocking*/
        xSemaphoreTake(statsqBinarySem, pdMS_TO_TICKS(10)); /*Take semaphore
for queue*/
        xQueueReceive(statsQueue, &receivedStats, pdMS_TO_TICKS(10)); /*Read
data from queue*/
        xSemaphoreGive(statsqBinarySem); /*Give back semaphore for queue*/
        xSemaphoreTake(consoleBinarySem, pdMS_TO_TICKS(10)); /*Take
semaphore for console*/
        /*Format and print the recorded statisice of data from task 1*/
        printf("\n\tPeriodic Check\n\tSamples: %d\tMean: %d\tMoving average of 5:
%d\n\n",receivedStats.count,receivedStats.mean,receivedStats.movAve);
    }
}

```

```

        xSemaphoreGive(consoleBinarySem);
        /*Give back semaphore for console*/

        vTaskDelayUntil(&xLastWakeTime,pdMS_TO_TICKS(5000));
/*Block for 5 seconds from the recorded time of unblocking*/
    }

}

/*Task to calculate and record statistics for task 1*/
void vTaskStats(void* pvParameters) {

    static uint32_t count = 0;                /*To record total number of values read*/
    static uint32_t sum = 0;                  /*To record the sum of all values read*/
    uint32_t temp = 0;                        /*Temporary storage variable*/
    uint32_t movAve = 0;

    uint32_t t_arr[4] = { 0 }; /*Array to store latest 4 values from queue; to be written
back to queue*/
    for (;;) {

        if (uxQueueMessagesWaiting(movingAveQueue) == 5) { /*Queue should be
full*/

            movAve = 0;                            /*Reset moving average*/
            xSemaphoreTake(movAveqBinarySem, pdMS_TO_TICKS(10)); /*Take
semaphore for queue*/

            xQueueReceive(movingAveQueue,&temp,0); /*Read earliest value
from queue*/

            sum += temp;        /*Add to sum and current moving average*/
            movAve += temp;
            count++;            /*Increment count of total values read*/

```

```

        for (uint8_t i = 0; i < 4; ++i) {
            xQueueReceive(movingAveQueue, &t_arr[i], 0); /*Read all
remaining values in queue into array*/

            movAve += t_arr[i]; /*Add remaining values current moving
average*/
        }

        movAve = movAve / 5; /*Calculate moving average; mean of last 5
values*/

        Stats_t newSt = { count , (sum / count) , movAve }; /*Create data structure to
write single value on writeQueue*/
        for (uint8_t i = 0; i < 4; ++i) {
            xQueueSendToBack(movingAveQueue, &t_arr[i], 0); /*Write back the
last 4 values to the queue, preserving initial order*/
        }
        xSemaphoreGive(movAveqBinarySem); /*Give back semaphore for queue*/
        xSemaphoreTake(statsqBinarySem, pdMS_TO_TICKS(10)); /*Take semaphore
for queue*/
        xQueueOverwrite(statsQueue, &newSt); /*Overwrite current value;
overwrite is preferred for queues of size 1*/
        xSemaphoreGive(statsqBinarySem); /*Give back semaphore for queue*/
    }
}

//vTaskDelay(pdMS_TO_TICKS(100));
}

/*Task to print an alert if task 2 sends the same value three times in a row*/
void vTaskAlert(void* pvParameters) {
    /*Variables to store last 3 values as read from queue*/
    uint8_t t1;

```

```

uint8_t t2;
uint8_t t3;
for (;;) {
    if (uxQueueMessagesWaiting(alertQueue) == 3) {
        xSemaphoreTake(alertqBinarySem, pdMS_TO_TICKS(10));
        /*Take semaphore for queue*/
        xQueueReceive(alertQueue, &t1, 0); /*Read 2 values from the queue*/
        xQueueReceive(alertQueue, &t2, 0);
        xQueuePeek(alertQueue, &t3, 0); /*And peek at the third; peeking
does not remove value from queue, but can only read value at front of queue*/
        xQueueSendToFront(alertQueue, &t2, 0); /*Write back the second
value read to FRONT of queue. This makes it so only the earliest value from queue is
removed and order is preserved*/
        xSemaphoreGive(alertqBinarySem); /*Give back semaphore for queue*/
        if (t1 == t2 && t2 == t3) { /*If all three values are same*/
            xSemaphoreTake(consoleBinarySem, pdMS_TO_TICKS(10)); /*Take
semaphore for console*/
            vPrintString("Alert: Task 2 has sent "); /*Print the alert on the console*/
            printf("%d", t1);
            vPrintString(" three times in a row\n");
            xSemaphoreGive(consoleBinarySem); /*Give back semaphore for
console*/
        }
    }
    vTaskDelay(pdMS_TO_TICKS(10));
}

/*Task to change constant value sent by task 3*/
void vTaskWrite(void* pvParameters) {
    uint32_t qval = 0;

```

```

for (;;) {
    if (_kbhit() != 0)
        /*Check for key press by user*/
        {
            xSemaphoreTake(consoleBinarySem, 20); /*Take semaphore for console*/
            /* Remove the key from the input buffer. */
            (void)_getch();
            qval = 0;
            /*Prompt and read new value from console*/
            vPrintString("Enter new value for task 3: ");
            scanf_s(" %d", &qval);
            xSemaphoreGive(consoleBinarySem); /*Give back semaphore for console*/
            xSemaphoreTake(writeqBinarySem, pdMS_TO_TICKS(10)); /*Take semaphore
for queue*/
            xQueueOverwrite(writeQueue, &qval); /*Overwrite the value in writeQueue
that task 3 reads from*/
            xSemaphoreGive(writeqBinarySem); /*Give back semaphore for queue*/
        }
        vTaskDelay(pdMS_TO_TICKS(10));
    }
}

/*Task to send random values*/
void vTaskRandSender(uint8_t* piParameters) /*An integer array: [ID, MIN, MAX] is passed
as parameter*/
{
    /*Read parameters into meaningful variables*/
    DataSource_t senderID = piParameters[0];
    uint8_t min = piParameters[1];
    uint8_t max = piParameters[2];

    BaseType_t xStatus;

```

```

    for (;;)
    {
        uint8_t val = 0;
        if (max != min)
            val = min + rand() % (max - min);    /*Calculate random value in valid
range; tasks 1 and 2*/
        else {
            xSemaphoreTake(writeqBinarySem, pdMS_TO_TICKS(10)); /*Take semaphore
for queue*/
            xQueuePeek(writeQueue, &val, 0); /*Peek at value for task 3: constant
sender*/
            xSemaphoreGive(writeqBinarySem); /*Give back semaphore for queue*/
        }

        Data_t structToSend = { val, senderID }; /*Create data structure to write on queue*/

        xSemaphoreTake(qCountingSem, 100);    /*Take semaphore for queue*/
        xQueueSendToBack(senderQueue, &structToSend, pdMS_TO_TICKS(10)); /*Write
data structure to queue*/
        xSemaphoreGive(qCountingSem);          /*Give back semaphore for queue*/
        vTaskDelay(sendFrequency);             /*Block for simulated frequency*/
    }
}

void vTaskController(void* pvParameters)
{
    Data_t xReceivedStruct;
    for (;;)
    {

```

```

xSemaphoreTake(qCountingSem, pdMS_TO_TICKS(10));
    /*Take semaphore for queue*/

xQueueReceive(senderQueue, &xReceivedStruct, sendFrequency);
/*Receive data structure containing value and task ID*/

xSemaphoreGive(qCountingSem);
    /*Give back semaphore for queue*/

1*/
if (xReceivedStruct.eDataSource == 1) {          /*If data received form task

    xSemaphoreTake(consoleBinarySem, pdMS_TO_TICKS(10));
        /*Take semaphore for console*/

    vPrintString("Receved from Task 1:");
        /*Print value sent by task 1*/

    printf("%d\n", xReceivedStruct.randVal);

    xSemaphoreGive(consoleBinarySem);
        /*Give back semaphore for console*/

    xSemaphoreTake(movAveqBinarySem, pdMS_TO_TICKS(10));
        /*Take semaphore for queue*/

    xQueueSendToBack(movingAveQueue, &xReceivedStruct.randVal,
pdMS_TO_TICKS(100)); /*Write data from task 1 to movingAverageQueue*/

    xSemaphoreGive(movAveqBinarySem);
        /*Give back semaphore for queue*/

}

if (xReceivedStruct.eDataSource == 2) {          /*If data from task 2*/

    xSemaphoreTake(consoleBinarySem, pdMS_TO_TICKS(10));
        /*Take semaphore for console*/

    vPrintString("Receved from Task 2: ", xReceivedStruct.randVal);
/*Print value sent by task 2*/

    printf("%d\n", xReceivedStruct.randVal);

    xSemaphoreGive(consoleBinarySem);
        /*Give back semaphore for console*/

    xSemaphoreTake(alertqBinarySem, pdMS_TO_TICKS(10));
        /*Take semaphore for queue*/

```



```

        xQueueSendToBack(alertQueue, &xReceivedStruct.randVal,
pdMS_TO_TICKS(100));          /*Write data from task 1 to alertQueue*/
        xSemaphoreGive(alertqBinarySem);
                                /*Give back semaphore for queue*/
    }
    if (xReceivedStruct.eDataSource == 3) {    /*If data from task 3*/
        xSemaphoreTake(consoleBinarySem, pdMS_TO_TICKS(10)); /*Take
semaphore for console*/
        vPrintString("Receved from Task 3: ", xReceivedStruct.randVal);
        /*Print value sent by task 1*/
        printf("%d\n", xReceivedStruct.randVal);
        xSemaphoreGive(consoleBinarySem);/*Give back semaphore for console*/
    }
}
}

```

```

int main(void)
{
    time_t t;
    srand((unsigned)time(&t)); /*Initialise srand for random number generator*/
    /*Initialise semaphores*/
    qCountingSem = xSemaphoreCreateCounting(2,2);
    consoleBinarySem = xSemaphoreCreateBinary();
    alertqBinarySem = xSemaphoreCreateBinary();
    statsqBinarySem = xSemaphoreCreateBinary();
    movAveqBinarySem = xSemaphoreCreateBinary();
    writeqBinarySem = xSemaphoreCreateBinary();
    /*Arrays to send as parameters to vTaskRandSender*/
    uint8_t range1[3] = { 1, 100, 200 };
    uint8_t range2[3] = { 2, 0, 5 };
}

```

```

uint8_t range3[3] = { 3, 10, 10 };    /*iF min=max, constant value sent by task*/
uint32_t init = 10;
/*Initiaise Queues*/
senderQueue = xQueueCreate(3, sizeof(Data_t));
alertQueue = xQueueCreate(3, sizeof(uint8_t));
statsQueue = xQueueCreate(1, sizeof(Stats_t));
writeQueue = xQueueCreate(1, sizeof(uint32_t));
movingAveQueue = xQueueCreate(5, sizeof(uint8_t));

/*The initial value for task 3 is 10*/
xQueueSend(writeQueue,&init,pdMS_TO_TICKS(100));
/*Three instances created of vTaskRandSender simulating 3 sensors*/
xTaskCreate(vTaskRandSender, "Rand Sender 1", 200, range1, 4, NULL);
xTaskCreate(vTaskRandSender, "Rand Sender 2", 200, range2, 4, NULL);
xTaskCreate(vTaskRandSender, "Const Sender ", 200, range3, 4, NULL);
/*Create all remaining tasks*/
xTaskCreate(vTaskController, "Controller", 200, NULL, 3, NULL);
xTaskCreate(vTaskStats, "Task 1", 200, NULL, 2, NULL);
xTaskCreate(vTaskAlert, "Task 2", 200, NULL, 2, NULL);
xTaskCreate(vTaskWrite, "Task 3", 200, NULL, 2, NULL);
xTaskCreate(vTaskPrint, "Task 4", 200, NULL, 2, NULL);
/*Start the Scheduler*/
vTaskStartScheduler();
for (;;)
}

```

## Output Description:

### Sender Task

Sender function runs three times for three different inputs. One for the range of 100-200. The other for 0 to 5. The third one is a constant single number '10' as shown in the picture above for task 3. So Task 1 and 2 are randomly generating numbers between a specified range but task 3 is always 10.

```

C:\Users\Power\Downloads\Programs\FreeRTOSv202012.00\FreeRTOS\Demo\Ireal\Debug\Ireal.exe
Received from Task 1: 141
Received from Task 2: 1
Received from Task 3: 10
Received from Task 1: 167
Received from Task 2: 2
Received from Task 3: 10
Received from Task 1: 134
Received from Task 2: 4
Received from Task 3: 10
Received from Task 1: 100
Received from Task 2: 0
Received from Task 3: 10
Received from Task 1: 169
Received from Task 2: 4
Received from Task 3: 10
Received from Task 1: 124
Received from Task 2: 4
Received from Task 3: 10
Received from Task 1: 178
Received from Task 2: 3
Received from Task 3: 10
Received from Task 1: 158
Received from Task 2: 3
Received from Task 3: 10
Received from Task 1: 162
Received from Task 2: 2
Received from Task 3: 10
Received from Task 1: 164
Received from Task 2: 4

```

### Periodic Task

For the periodic function we have used the print function which executes after every 5 seconds. The outputs of controller and stats functions/tasks are received and print function displays total

count/number of samples, the mean of all the numbers and moving average of the last five numbers periodically after 5 seconds.

```

Receivied from Task 3: 10
Receivied from Task 1: 189
Receivied from Task 2: 4
Receivied from Task 3: 10
Receivied from Task 1: 176
Receivied from Task 2: 1
Receivied from Task 3: 10
Periodic Check
Samples: 297      Mean: 148      Moving average of 5: 161
Receivied from Task 1: 129
Receivied from Task 2: 4
Receivied from Task 3: 10
Receivied from Task 1: 168
Receivied from Task 2: 3
Receivied from Task 3: 10
Receivied from Task 1: 192
Receivied from Task 2: 2

```

## Alert Task

Since task 2 has a very short range, it is likely that it may send the same number multiple times in a row. This represents a failure or weakness of the random number generator used. Hence we treat this as simulating a monitoring parameter, where three values sent in a row is an abnormal occurrence and should be reported. From the below output we can actually see that a warning has been sent that something wrong has been done and task 2 has sent a specific value three times in a row. We have kept track of the values for the sensors. Because for the warning we are also displaying which number has been sent multiple times. “Task 2 has sent 2” three times in a row” shows that value 2 was sent more than 3 times.

```

Receivied from Task 1: 144
Receivied from Task 2: 4
Receivied from Task 3: 10
Receivied from Task 1: 162
Receivied from Task 2: 2
Receivied from Task 3: 10
Receivied from Task 1: 157
Receivied from Task 2: 2
Receivied from Task 3: 10
Receivied from Task 1: 137
Receivied from Task 2: 2
Alert: Task 2 has sent 2 three times in a row
Receivied from Task 3: 10
Receivied from Task 1: 159
Receivied from Task 2: 4
Receivied from Task 3: 10

```

## Write Task

For the write task, a user input is taken to change the value of the task 3 constant number. This is the task which allows the user to change that constant number being given as output every time from task 3. By default the set value is 10 but we can change it to any number.

```
Recevied from Task 3: 10
Enter new value for task 3: Recevied from Task 1: 192
Recevied from Task 2: 2
Recevied from Task 3: 10
1Recevied from Task 1: 182
Recevied from Task 2: 2
Recevied from Task 3: 10
2Recevied from Task 1: 121
Recevied from Task 2: 1
Recevied from Task 3: 10

Recevied from Task 1: 116
Recevied from Task 2: 1
Recevied from Task 3: 12
Recevied from Task 1: 118
Recevied from Task 2: 3
Recevied from Task 3: 12
Recevied from Task 1: 195
Recevied from Task 2: 0
Recevied from Task 3: 12
```

## Conclusion:

Conclusively, In this project we have successfully implemented all the required concepts for projects named as FreeRtos simulation by creating different tasks, Queues for inter tasks communication and dummy values were assumed for sensor's data. Timers were used for periodic output and semaphores for protection purposes.