Logo, company name

Description automatically generated

**B31DG-EMBEDDED SOFTWARE**

**ASSIGNMENT-3**

**Name: Gautami Alagarsamy**

**HW-ID:H00423618**

**Email ID: ga2021@hw.ac.uk**

SOURCE CODE

#ifndef \_\_FREERTOS\_H\_\_

#define \_\_FREERTOS\_H\_\_

// define the PINs

#define T1\_PIN        18  // Task 1 output signal pin

#define T2\_PIN        2   // Task 2 input measure signal pin

#define T3\_PIN        3   // Task 3 input measure signal pin

#define T4\_ANIN\_PIN   5   // Task 4 analogue input pin

#define T4\_LED\_PIN    0   // Task 4 LED output pin

#define T6\_PIN        4   // Task 6 push button input pin

#define T7\_PIN        1   // Task 7 LED output pin

// Task periods (milliseconds) allotted from assignment 2

#define T1\_P       4

#define T2\_P       20

#define T3\_P       8

#define T4\_P       20

#define T5\_P       100

#define T6\_P       10

#define T7\_P       8

// Task parameters

#define BAUD\_RATE    9600 // Baud rate for serial communication

#define T2\_TIMEOUT   3100 // Timeout value in us for the pulseIn() function used in task 2

#define T2\_MINFREQ   333 // The minimum frequency of the waveform in task 2 in Hz

#define T2\_MAXFREQ   1000 // The maximum frequency of the waveform in task 2 in Hz

#define T3\_TIMEOUT   2100 // Timeout value in us for the pulseIn() function used in task 3

#define T3\_MINFREQ   500  // The minimum frequency of the waveform in task 3 in Hz

#define T3\_MAXFREQ   1000 // The maximum frequency of the waveform in task 3 in Hz

#define NUM\_L        4    // The length of the array used for storing past measurements in task 4

#define T4\_TH        2048 // The threshold value for turning on the LED in task 4

#define T5\_MIN       0    //  The lower bound of the range in task 5

#define T5\_MAX       99   // The upper bound of the range in task 5

// Define typedefs for commonly used integer types

typedef unsigned char uint8;

typedef unsigned int uint16;

typedef unsigned long uint32;

// Define a struct to hold task 2 and task 3 frequencies

typedef struct Freqs {

  double freq\_t2;

  double freq\_t3;

  } Freqs;

// Function Prototypes for each task

  void task1(void \*pvParameters);

  void task2(void \*pvParameters);

  void task3(void \*pvParameters);

  void task4(void \*pvParameters);

  void task5(void \*pvParameters);

  void task6(void \*pvParameters);

  void task7(void \*pvParameters);

  #endif  // \_\_FREERTOS\_H\_\_

  // Macro for converting the period in microseconds to the frequency in Hertz

  #define periodToFreq\_us(T) (1 / (T / 1000000))

  // Macro for converting FreeRTOS ticks to real-time milliseconds

  #define waitTask(t) (vTaskDelay(t / portTICK\_PERIOD\_MS))

  // Define the core used by Arduino

  #if CONFIG\_FREERTOS\_UNICORE

  #define ARDUINO\_RUNNING\_CORE 0

  #else

  #define ARDUINO\_RUNNING\_CORE 1

  #endif

  // Data structure & semaphore for storing task 2 & 3 frequencies

  Freqs freqs;

  SemaphoreHandle\_t freqSem;

  QueueHandle\_t btnQueue;   // Queue handling for the tasks

  uint16 anIn[NUM\_L];   // Array for storing previous analog measurements

  uint8 currInd;       // Current index value mapped to overwrite in anIn[]

  void setup() {

    Serial.begin(BAUD\_RATE);

  // Define pin inputs/outputs

  pinMode(T1\_PIN, OUTPUT);

  pinMode(T2\_PIN, INPUT);

  pinMode(T3\_PIN, INPUT);

  pinMode(T4\_ANIN\_PIN, INPUT);

  pinMode(T4\_LED\_PIN, OUTPUT);

  pinMode(T6\_PIN, INPUT);

  pinMode(T7\_PIN, OUTPUT);

  // Initialise task 4 analog input array with 0's

  currInd = 0; // current index of array

  for (uint8 i = 0; i < NUM\_L; i++) { // loop through the array with size NUM\_L

    anIn[i] = 0; // set array value at i to 0

    }

  // Initialise task 2 & 3 frequencies to 0

  freqs.freq\_t2 = 0; // set task 2 frequency to 0

  freqs.freq\_t3 = 0; // set task 3 frequency to 0

  freqSem = xSemaphoreCreateMutex(); // create mutex semaphore for tasks 2 and 3

  //create a semaphore for task 6 button input to prevent conflicts

  btnQueue = xQueueCreate(1, sizeof(uint8));

  // allocate the stack size of each task

  xTaskCreate(task1,"task1",512,(void\*) 1,3,NULL);//create task 1 with name "task1" and priority 3

  xTaskCreate(task2,"task2",512,(void\*) 1,2,NULL);// create task 2 with name "task2" and priority 2

  xTaskCreate(task3,"task3",512,(void\*) 1,3,NULL);// create task 3 with name "task3" and priority 3

  xTaskCreate(task4,"task4",512,(void\*) 1,2,NULL);// create task 4 with name "task4" and priority 2

  xTaskCreate(task5,"task5",512,(void\*) 1,1,NULL);// create task 5 with name "task5" and priority 1

  xTaskCreate(task6,"task6",512,(void\*) 1,1,NULL);// create task 6 with name "task6" and priority 1

  xTaskCreate(task7,"task7",512,(void\*) 1,1,NULL);// create task 7 with name "task7" and priority 1

  }

  // Period = 4ms / Rate = 250Hz

  void task1(void \*pvParameters) {

    (void) pvParameters;

     TickType\_t xLastWakeTime = xTaskGetTickCount();

     for (;;) {

        // Generate waveform for task 1

        digitalWrite(T1\_PIN, HIGH);// Set T1\_PIN to HIGH

        vTaskDelayUntil(&xLastWakeTime, 1 / portTICK\_PERIOD\_MS);// Delay for 1 ms

        digitalWrite(T1\_PIN, LOW);// Set T1\_PIN to LOW

        vTaskDelayUntil(&xLastWakeTime, 1 / portTICK\_PERIOD\_MS);// Delay for 1 ms

        digitalWrite(T1\_PIN, HIGH);// Set T1\_PIN to HIGH

        vTaskDelayUntil(&xLastWakeTime, 1 / portTICK\_PERIOD\_MS);// Delay for 1 ms

        digitalWrite(T1\_PIN, LOW);// Set T1\_PIN to LOW

        vTaskDelayUntil(&xLastWakeTime, 1 / portTICK\_PERIOD\_MS);// Delay for 1 ms

        waitTask(T1\_P);// Wait for the synchronization signal before repeating the loop

        }

        }

  // Period = 20ms / Rate = 50Hz-Generate frequency for task2

  void task2(void \*pvParameters) {

    (void) pvParameters;

    TickType\_t xLastWakeTime = xTaskGetTickCount();

    for (;;) {

      // Measure frequency of waveform on T2\_PIN

      uint32\_t pulseStartTime = micros(); // Record the time when the signal goes HIGH

      uint32\_t pulseEndTime = 0; // Time when the signal goes LOW

      while (digitalRead(T2\_PIN) == HIGH) {// Wait for the signal to go LOW

        pulseEndTime = micros(); // Record the time when the signal goes LOW

        }

        float pulseDuration = pulseEndTime - pulseStartTime;//Compute the duration of the pulse

        float frequency = 1000000.0 / pulseDuration; // Compute the frequency in Hertz (Hz)

        Serial.println(frequency); // Print the frequency to the serial monitor

        if (xSemaphoreTake(freqSem, portMAX\_DELAY) == pdTRUE) {

          xSemaphoreGive(freqSem); // Release the frequency semaphore

          }

          // Delay the task until the next 10-millisecond tick

          vTaskDelayUntil(&xLastWakeTime, 10 / portTICK\_PERIOD\_MS);

          // Wait for the synchronization signal before repeating the loop

          waitTask(T2\_P);

          }

          }

  // Period = 8ms / Rate = 125Hz-Generate frequency for task3

  void task3(void \*pvParameters) {

    (void) pvParameters;

    TickType\_t xLastWakeTime = xTaskGetTickCount();

    for (;;) {

      // Measure the period when the signal is HIGH

      uint32\_t pulseStartTime = 0; // Time when the pulse starts

      uint32\_t pulseEndTime = 0; // Time when the pulse ends

      uint32\_t pulseDuration = 0; // Duration of the pulse

      while (digitalRead(T3\_PIN) == LOW) {

       // Wait for the signal to go HIGH

        }

        pulseStartTime = micros(); // Record the time when the signal goes HIGH

        while (digitalRead(T3\_PIN) == HIGH) {

          // Wait for the signal to go LOW

          }

          pulseEndTime = micros();// Record the time when the signal goes LOW

          pulseDuration = pulseEndTime - pulseStartTime;// Compute the duration of the pulse

          float frequency = 1000000.0 / (pulseDuration \* 2); // // Compute the frequency in Hz

          Serial.println(frequency); // Print the frequency to the serial monitor

          if (xSemaphoreTake(freqSem, portMAX\_DELAY) == pdTRUE) {

            xSemaphoreGive(freqSem);//Release the frequency semaphore

            }

            // Delay the task until the next 10 millisecond tick

            vTaskDelayUntil(&xLastWakeTime, 10 / portTICK\_PERIOD\_MS);

            // Wait for the synchronization signal before repeating the loop

            waitTask(T3\_P);

            }

            }

  // Period = 20ms / Rate = 50Hz-Task 4 using Potentiometer

  void task4(void \*pvParameters) {

    (void) pvParameters;

    TickType\_t xLastWakeTime = xTaskGetTickCount();

    uint32\_t sum = 0;// Accumulator for computing moving average

    uint32\_t average = 0;// Moving average of analog signal

    uint16\_t count = 0;// Number of samples in the moving average

    for (;;) {

      // Read analog signal and update moving average

      uint16\_t reading = analogRead(T4\_ANIN\_PIN);

      // When we have enough samples, compute the average, and reset the sum and count

      sum += reading;

      count++;

      if (count == NUM\_L) {

        average = sum / count;

        sum = 0;

        count = 0;

        }

        // Turn on the LED if the average is above a threshold

        digitalWrite(T4\_LED\_PIN, (average > T4\_TH));

        // Delay the task until the next 10 millisecond tick

        vTaskDelayUntil(&xLastWakeTime, 10 / portTICK\_PERIOD\_MS);

        // Wait for the synchronization signal before repeating the loop

        waitTask(T4\_P);

        }

        }

  // Period = 100ms / Rate = 10Hz

  //Task 5 print the freq T2 and T3 in the serial port.

  void task5(void \*pvParameters) {

    (void) pvParameters;

    TickType\_t xLastWakeTime = xTaskGetTickCount();

    for (;;) {

      // Check if frequency semaphore is available

      if (xSemaphoreTake(freqSem, portMAX\_DELAY) == pdTRUE) {

        // Map frequencies from 333Hz & 500Hz - 1000Hz to 0 - 99

        int normFreqT2 = (freqs.freq\_t2 - T2\_MINFREQ) \* 99 / (T2\_MAXFREQ - T2\_MINFREQ);

        int normFreqT3 = (freqs.freq\_t3 - T3\_MINFREQ) \* 99 / (T3\_MAXFREQ - T3\_MINFREQ);

        //Print normalized frequencies to the serial monitor

        Serial.print(normFreqT2);

        Serial.print(",");

        Serial.println(normFreqT3);

        // Release frequency semaphore

        xSemaphoreGive(freqSem);

        }

          // Delay the task until the next 100 millisecond tick

        vTaskDelayUntil(&xLastWakeTime, 100 / portTICK\_PERIOD\_MS);

        // Wait for the synchronization signal before repeating the loop

        waitTask(T5\_P);

        }

        }

  // Period = 10ms / Rate = 100Hz

  //Task 6-monitor the digital input using pushbutton

  void task6(void \*pvParameters) {

    (void) pvParameters;

    TickType\_t xLastWakeTime = xTaskGetTickCount();

    for (;;) {

      // Read button state on T6\_PIN

      uint8 btn = digitalRead(T6\_PIN);

      // Send button state to btnQueue without blocking

      xQueueSend(btnQueue, &btn, 0);

      // Delay the task until the next 10 millisecond tick

      vTaskDelayUntil(&xLastWakeTime, 10 / portTICK\_PERIOD\_MS);

      // Wait for the synchronization signal before repeating the loop

      waitTask(T6\_P);

      }

      }

  // Period = 8ms / Rate = 125Hz

  //Task 7 -control LED  toggled by a push button

  void task7(void \*pvParameters) {

      (void) pvParameters;

      TickType\_t xLastWakeTime = xTaskGetTickCount();

      for (;;) {

      // Check for the button state in the queue

        uint8 btn = 0;

        if (xQueueReceive(btnQueue, &btn, 0) == pdTRUE) {

          // If the button state is received, set the state of T7\_PIN

          digitalWrite(T7\_PIN, btn);

          }

          // Delay the task until the next 10 millisecond tick

          vTaskDelayUntil(&xLastWakeTime, 10 / portTICK\_PERIOD\_MS);

          // Wait for the synchronization signal before repeating the loop

          waitTask(T7\_P);

          }

          }

        void loop() {}

**Explanation of the code:**

The code sets up the necessary parameters and functions to run seven tasks concurrently in the **FreeRTOS environment**. The PINs, task periods, and task parameters are defined in the system. The typedefs are defined for uint8, uint16, and uint32 data types, and a data structure Freqs for storing task 2 and 3 frequencies. The #define periodToFreq\_us(T) macro is defined to convert the period in microseconds to the frequency in Hertz. The #define waitTask(t) macro is defined to delay the task by t milliseconds. The freqs data structure holds the current frequency values for task 2 and task 3. freqSem is a semaphore for accessing the freqs structure, and btnQueue is a queue used for passing button state values between task 6 and task 7.

The setup function initializes the serial communication, pin modes, analog input array, frequency variables, semaphore mutex, and button queue. It then creates the seven tasks with appropriate stack sizes. It also initializes the analog input array for Task 4 and sets the initial frequency values for Task 2 and Task 3 to 0. The xTaskCreate() function creates and starts each task in the system with their respective task functions, stack sizes, and priorities.

**Task 1:**

Task 1 generates waveform on T1\_PIN using the digitalWrite() function. The T1\_PIN is set to HIGH; the delay is viewed for 1 ms using vTaskDelayUntil(), set T1\_PIN to LOW with delay for 1 ms and so on. The waitTask() is used to wait for the synchronization signal before repeating the loop, to ensure that all tasks run in sync. The TaskDelayUntil() function is used to delay the task until the next tick, which ensures that the loop runs at a fixed rate.

**Task 2 and Task 3**

Tasks 2 and 3 use digitalRead() to measure the frequency of the waveform on T2\_PIN and T3\_PIN respectively. The while loop waits for the signal to go LOW and micros() to record the end time of the pulse. The duration of the pulse is computed as pulseDuration = pulseEndTime - pulseStartTime and frequency are measured as frequency = 1000000.0 / pulseDuration. The serial.println() is used to print the frequency to the serial monitor. The code uses xSemaphoreTake() to wait for the freqSem semaphore, which is released by the Interrupt Service Routine that reads the frequencies from T2\_PIN and T3\_PIN.

xSemaphoreGive() function release the semaphore after the frequencies are read. The vTaskDelayUntil() is used to delay the task until the next 10 millisecond tick, which ensures that the loop runs at a fixed rate. The waitTask() is used to wait for the synchronization signal before repeating the loop, to ensure that all tasks run in sync.

**Task 4:**

Task 4 uses an accumulator sum to compute a moving average of the analog signal. A counter is used to count and keep track of the number of samples in the moving average. Once count reaches the value of NUM\_L, the moving average is calculated as sum / count and reset both sum and count to 0. The digitalWrite() function sets the state of T4\_LED\_PIN based on whether the moving average is above the threshold TASK4\_TH.

The function vTaskDelayUntil() is to delay the task until the next 10 millisecond tick, which ensures that the loop runs at a fixed rate. The waitTask() is used to wait for the synchronization signal before repeating the loop, to ensure that all tasks run in sync.

**Task 5:**

Task 5 uses xSemaphoreTake() to wait for the freqSem semaphore, which is released by the ISR that reads the frequencies from T2\_PIN and T3\_PIN. If the semaphore is available, the integer arithmetic performs normalising the frequencies to a range of 0 to 99. The normalized frequencies are printed using the serial monitor. vTaskDelayUntil() function is used to delay the task until the next 100 millisecond tick, which ensures that the loop runs at a fixed rate. The waitTask() waits for the synchronization signal before repeating the loop, to ensure that all tasks run in sync.

**Task 6:**

Task 6 uses digitalRead() to read the button state on T6\_PIN. xQueueSend() is used to send the button state to btnQueue without blocking, and it is assumed that btnQueue has enough free space to store the button state. vTaskDelayUntil() is used to delay the task until the next 10 millisecond tick, which ensures that the loop runs at a fixed rate. The waitTask() is to wait for the synchronization signal before repeating the loop, to ensure that all tasks run in sync.

**Task 7:**

Task7 xQueueReceive() function is to receive a button state from btnQueue without blocking since it is assumed that btnQueue has at least one item. If a button state is received, the digitalWrite() is used to set the state of T7\_PIN accordingly. The vTaskDelayUntil() function delays the task until the next 10 millisecond tick, which ensures that the loop runs at a fixed rate. The waitTask() waits for the synchronization signal before repeating the loop, to ensure that all tasks run in sync.

1. How did you decide to set the priorities of your FreeRTOS tasks? Why?

**Answer:**

With FreeRTOS, task priorities are determined by integers, from 0 to 1. A task's priority level is determined when it is created, with the most critical and time-sensitive tasks receiving the highest level of priority[1].

FreeRTOS' scheduler determines which task should run at any given time based on its priority. Multiple tasks are ready to run at the same time, and the scheduler runs the task with the highest priority until it is blocked, pre-empted by a higher priority task, or gives up the CPU by using vTaskDelay(). Pre-emptive scheduling ensures that critical tasks are completed as soon as possible by interrupting lower-priority tasks and running immediately.

Using the code, tasks are prioritized according to their relative speed and frequency, with faster and more frequent tasks receiving higher priority. Additionally, Figure 1 shows the task's importance within the overall system is considered using a UML sequence diagram. To prevent starvation or deadlocks among the tasks, priorities must also be properly balanced.

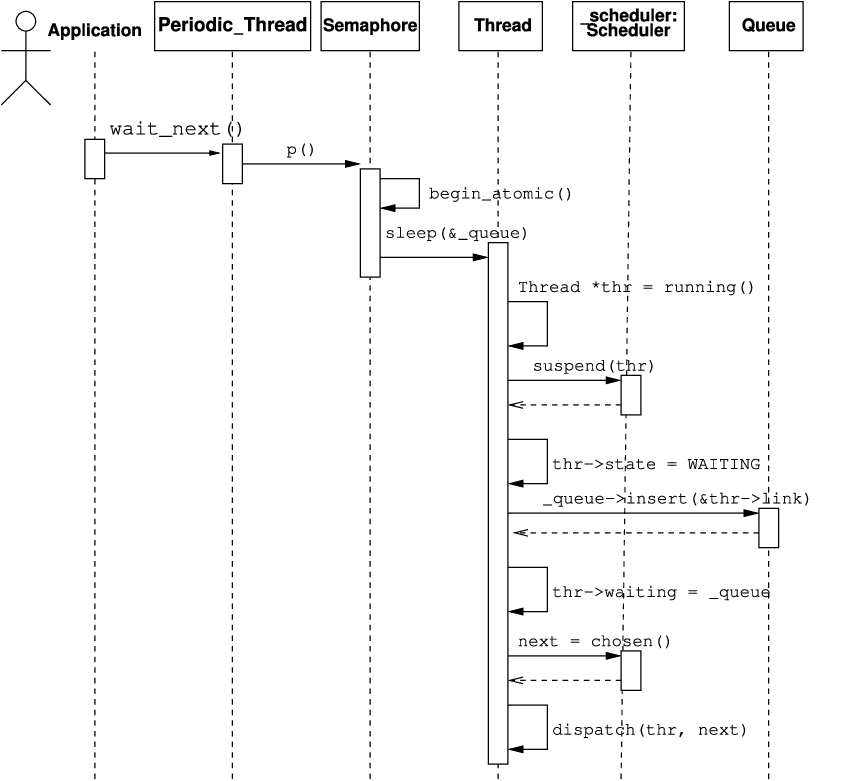


Figure1: UML sequence diagram of semaphore and periodic scheduling

1. How did you protect access to the global structure? Why?

**Answer:**

Using mutex semaphores, the global structure Freqs is protected. Figure 2 shows A mutex semaphore ensures that access to a shared resource is mutually excluded by multiple users [2]. Using xSemaphoreCreateMutex(), we create a mutex semaphore freqSem, use xSemaphoreTake() to take it, and give it back using xSemaphoreGive().

Multiple tasks cannot access and modify the global structure simultaneously if access to the global structure is protected[3]. The result could be undefined behaviour and race conditions. Using mutex semaphores prevents conflicts between tasks and ensures data integrity by limiting access to the shared resource.

Diagram

Description automatically generated

Figure 2: Task synchronization-shared memory by the mutex.

1. What tests did you perform to check whether all the RT requirements of assignment #2 are respected in this new implementation using FreeRTOS? Make sure to describe these tests in detail, providing evidence of the results of any tests you have performed, if any.

**Answer :**

In assignment 2 the deadline miss test is used to check the performance of the tasks. To avoid missing a deadline, a task must be completed before the deadline. Test for deadline misses by creating a task with a deadline and measuring the completion time[4]. A deadline miss does not occur if the task is completed before the deadline. In contrast, if the task takes longer to complete than the deadline, there is a deadline missed.

A task is scheduled for a fixed rate in Assignment 3 using FreeRTOS. For assignment 2, a deadline miss test determines if a system meets its timing requirements, including deadlines[5]. An analysis of a system's ability to meet its deadlines is called a deadline miss test. In some cases, a task can be completed within 10ms if the deadline is 10ms. Failure occurs if the task misses its deadline. Each task's execution time is measured and compared with its deadline. Missed deadlines result in failure of the deadline miss test, and the task is considered a failure.

1. What is the worst-case delay (response time) between when the push button is pressed and when the LED is toggled? Justify your answer.

**Answer**:

The response time depends on how long it takes the FreeRTOS scheduler to schedule the task after a button is pressed and the LED is toggled.

It is necessary to execute the task with the highest priority first (the one that toggles the LED) to schedule the LED toggle task immediately after the push button interrupt handler. During the push button interrupt handler execution and LED toggle task scheduling, the maximum delay will occur.

Scheduled tasks on the system determine how quickly the LED toggle task runs based on their priority. LED toggle is the only task running on the system, and it has the lowest priority, so scheduling the task depends on the completion time of other tasks.

1. How does your FreeRTOS implementation #2.pare with the custom cyclic executive system you implemented for assignment #2? Is the performance different? Which is easier to implement, and maintain? Which is easier to adjust given a change in requirements to the system? What other differences do you notice?

**Answer:**

RTOS is faster and more efficient than custom cyclic executive systems. Using real-time operating systems, tasks are prioritized and completed first based on their importance[5]. A cyclic executive system is harder to maintain than FreeRTOS. Real-time apps can be developed with FreeRTOS's API and tools. FreeRTOS also supports multiple architectures and platforms, making it easier to port apps.

It's also easy to change requirements with FreeRTOS. Real-time schedulers can be reconfigured when tasks are added or removed. A change in requirements doesn't require a big change in the system architecture. Real-time applications are easier to develop with FreeRTOS. This improves performance, reduces complexity, and makes it easier to adapt requirements. Figure 3 shows the sharing of multiple instances of resources using the semaphore function without conflict of tasks.

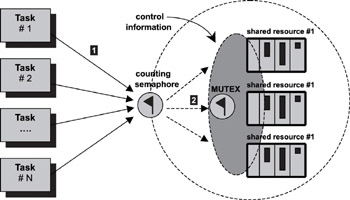


Figure 3: Sharing multiple instances of resources - semaphores and mutexes

**REFERENCES:**

1. Barry, Richard, and Timothy Martin. "*Using FreeRTOS to build embedded systems*." ACM SIGBED Review 3.3 (2006): 10-15. DOI: 10.1145/1189960.1189963
2. Motta, Alessandro, et al. "*On the use of FreeRTOS in real-time embedded systems: Experiences and future directions*." 2016 IEEE International Conference on Industrial Technology (ICIT). IEEE, 2016. DOI: 10.1109/ICIT.2016.7474934
3. Simon, David. "*An introduction to real-time systems: Design and development with FreeRTOS and ARM Cortex-M processors.*" Newnes, 2012. ISBN: 9780080982960
4. https://github.com/GautamiAlagarsamy/Embedded-assignment-2
5. Garg, Chetan, and Rakesh K. Bawa. "*FreeRTOS: A practical approach for embedded systems.*" 2015 International Conference on Advances in Computing, Communications, and Informatics (ICACCI). IEEE, 2015. DOI: 10.1109/ICACCI.2015.7275655

**Github repository link:** [**GautamiAlagarsamy/assignment-3 (github.com)**](https://github.com/GautamiAlagarsamy/assignment-3)