Heriot-Watt University Dubai Campus
[B31DG] Embedded Software 2023

REPORT

ASSIGNMENT NO. 3

INTRODUCTION

To develop a program that does the same real-time activities as Assignment #2, but also monitors a digital input connected to a pushbutton and controls an LED. The digital input should be debounced, and the pushbutton should flip the status of the LED.

The code is a microcontroller board program that generates and performs several jobs at the same time using FreeRTOS, a real-time operating system kernel.

Many variables are defined in the code to represent the pins on the microcontroller board that will be utilized as inputs and outputs for various activities. It also generates a semaphore and multiple tasks by calling the xTaskCreate() method, which accepts the task function's name, a string identifier, the task stack size, a parameter provided to the task, the task priority, and a reference to a variable to hold the task handle.

The tasks themselves are defined as functions that take a void pointer parameter and return nothing. They are scheduled to operate in parallel by the FreeRTOS kernel, which controls resource allocation and job scheduling based on priority.

Task 1 uses delays to flash an LED in a predefined sequence of high and low signals on the output pin.

Task 2 computes the frequency in Hertz by measuring the frequency of an input signal with the pulseIn() method. With the restrict() method, the frequency is confined between 333 Hz to 1000 Hz.

Task 3 computes the frequency in Hertz by measuring the frequency of another input signal with the pulseIn() method. With the restrict() method, the frequency is confined between 333 Hz to 1000 Hz.

Task 4 receives an analog signal from a potentiometer using the analogRead() function, computes the frequency in Hertz, and restricts it between 333 and 1000 Hz using the constrain() function. If the frequency falls outside of this range, an LED blinks to signal a problem.

Task 5 waits for a semaphore to be released before doing a sequence of activities not defined in the given code.

Task 6 waits for a button press before switching an LED on and off.

Task 7 waits for a semaphore to be freed before doing a sequence of activities not defined in the given code.

The setup() method uses the xTimerCreate() function to build a timer and the xTimerStart() function to start it. Every 4 milliseconds, the timer calls a callback method, outputTimerCallback().

The loop() method is empty because the FreeRTOS kernel runs all tasks and there is no explicit call to the tasks in the main loop.

1.ESP32

The ESP32 is a 2.4 GHz chip that smoothly incorporates both Wi-Fi and Bluetooth technologies. It is intended to deliver optimal power and RF performance using TSMC's low power 40 nm technology, making it extremely dependable and adaptable in a wide range of applications and power circumstances. The ESP32 supports the addition of numerous components, including as sensors, Lights, and more, to projects by providing 34 digital pins identical to those used in Arduino. The ESP32 WROOM module has 25 GPIO pins, all of which are input pins. Some of these input pins feature an internal pull-up, whereas others do not.

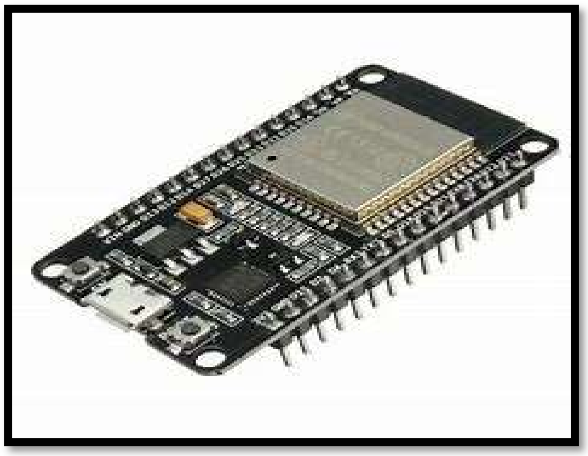


Fig 1.1: ESP32

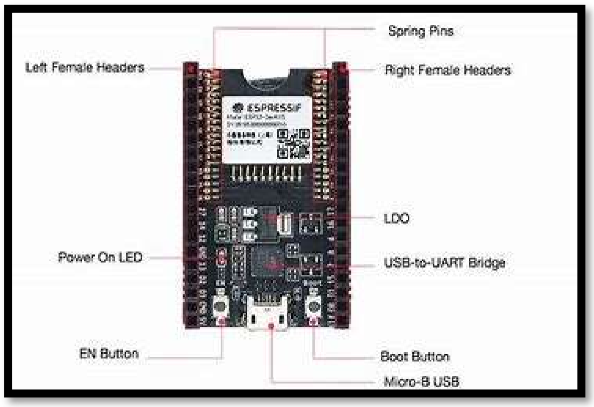


Fig 1.2: ESP32 DIAGRAM

2. Potentiometer

Fig 2.1: Potentiometer

Potentiometers are frequently used in electronic circuits for a number of reasons, including volume and tone control in audio equipment, as variable resistors in sensor circuits, and as variable resistors in power supply circuits. By rotating a knob or shaft, the sliding contact, also known as the wiper, may be moved along the resistor element. This modifies the potentiometer's resistance, which influences the voltage division ratio and the resulting output voltage. Potentiometers exist in a variety of sizes and values, and their reaction to position changes might be linear or logarithmic. Linear potentiometers have a uniform change in resistance for each unit of movement, but logarithmic potentiometers have a non-uniform response that corresponds to how human ears perceive variations in loudness.

3.Design of Cyclic Executive

A cyclic executive is a scheduling technique that is frequently used in real-time systems to carry out a series of periodic activities. The cyclic executive is a straightforward technique that does not necessitate the use of a real-time operating system. It is frequently utilized in embedded systems with low resources.

Tasks are scheduled by the cyclic executive depending on their periods. The duties are carried out in a cyclical fashion, with each duty being carried out once throughout its given period. The cyclic executive assumes that each task's execution time is predictable and constant. The assignments are classified as hard real-time tasks, which means they must be completed before the end of their term.

The cyclic executive divides time into equal time slices, and Each time slice corresponds to the smallest term of the job. The cyclic executive evaluates if any tasks are ready to be done during each time slice. When a job is ready to be completed, it is completed for the entire term. The cyclic executive waits for the next time slice if a job is not ready to be done.

The cyclic executive guarantees that each job is completed only once throughout the time frame set. It does not, however, guarantee that the tasks will be completed by their strict deadlines. If a job's execution time exceeds its period, the task will miss its deadline, potentially resulting in a system failure. Overall, the cyclic executive is a straightforward and efficient system. A scheduling method commonly used in embedded devices. It is not, however, appropriate for all real-time systems, particularly those with complicated needs or where achieving strict deadlines is essential.

CIRCUIT DIAGRAM

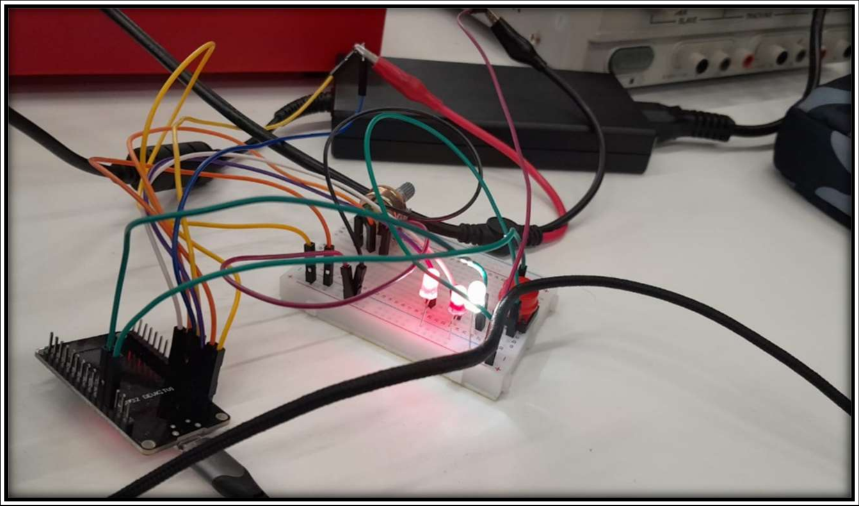


Fig 1.1: Simulation of circuit diagram

Fig 1.2: Signal generator

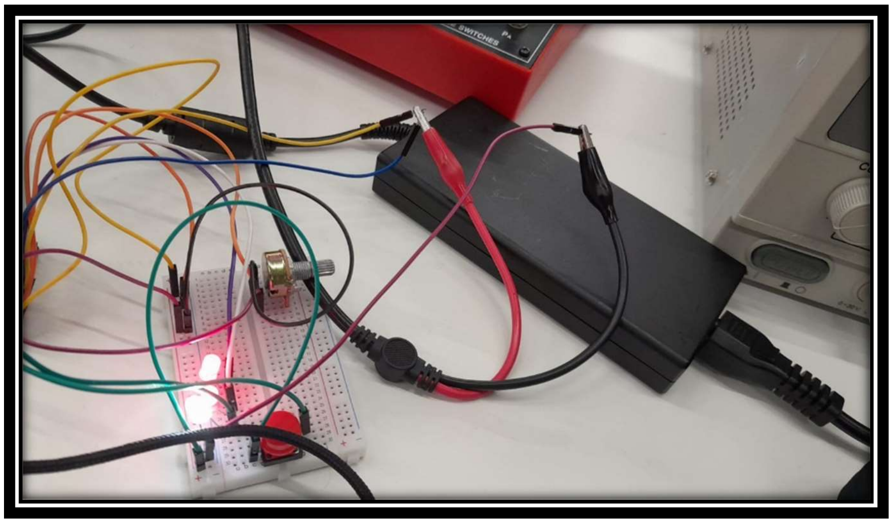
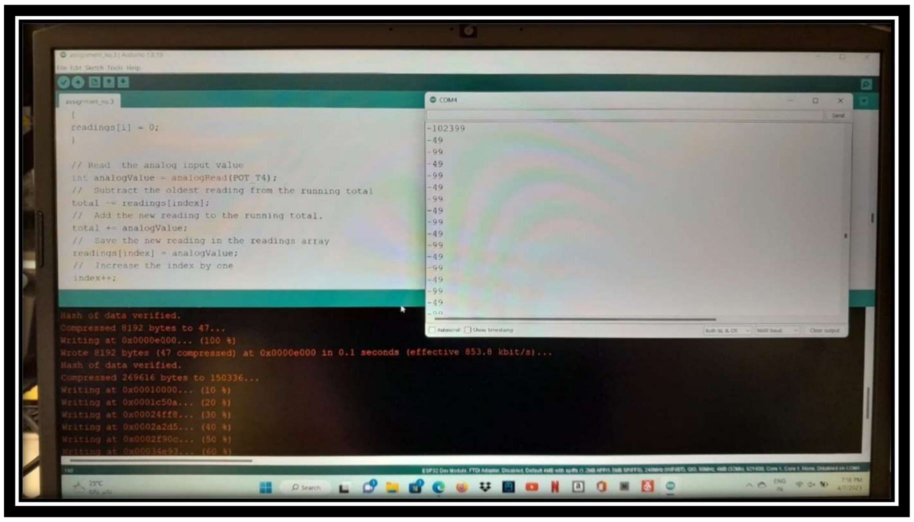


Fig1.2: Simulation of circuit diagram blink light



```
{
readings[i] = 0;
}

// Read  the analog input value
int analogValue = analogRead(POT_T4);
//  Subtract the oldest reading from the running total
total -= readings[index];
//  Add the new reading to the running total.
total += analogValue;
//  Save the new reading in the readings array
readings[index] = analogValue;
//  Increase the index by one
index++;
```

```
-102399
-49
-99
-49
-99
-49
-99
-49
-99
-49
-99
-49
-99
-49
-99
-49
```

```
Hash of data verified.
Compressed 8192 bytes to 47...
Writing at 0x0000e000... (100 %)
Wrote 8192 bytes (47 compressed) at 0x0000e000 in 0.1 seconds (effective 853.8 kbit/s)...
Hash of data verified.
Compressed 269616 bytes to 150336...
Writing at 0x00010000... (10 %)
Writing at 0x0001c50a... (20 %)
Writing at 0x00024ff8... (30 %)
Writing at 0x0002a2d5... (40 %)
Writing at 0x0002f90c... (50 %)
Writing at 0x00034e93... (60 %)
```

Fig: Violations [COM4]

Code description:

This is an Arduino program that implements a cyclic executive for a real-time system.

Task 1 uses delays to flash an LED in a predefined sequence of high and low signals on the output pin.

Task 2 computes the frequency in Hertz by measuring the frequency of an input signal with the pulseIn() method. With the restrict() method, the frequency is confined between 333 Hz to 1000 Hz.

Task 3 computes the frequency in Hertz by measuring the frequency of another input signal with the pulseIn() method. With the restrict() method, the frequency is confined between 333 Hz to 1000 Hz.

Task 4 receives an analog signal from a potentiometer using the analogRead() function, computes the frequency in Hertz, and restricts it between 333 and 1000 Hz using the constrain() function. If the frequency falls outside of this range, an LED blinks to signal a problem.

Task 5 waits for a semaphore to be released before doing a sequence of activities not defined in the given code.

Task 6 waits for a button press before switching an LED on and off.

Task 7 waits for a semaphore to be freed before doing a sequence of activities not defined in the given code.

Code:

```
 #include <freertos/FreeRTOS.h>

#include <freertos/semphr.h>

//Create by Gunjan


int LED_1=13; //For LED of task 1 output port

int LED2_FREQ=12;// To measure the frequency in task-2, an input port is required from the signal generator.

int LED3_FREQ=14;// To measure the frequency in task-3, it is necessary to have an input port connected to the signal generator.

int POT_T4=27;// To display the analog frequency, an input port is needed from the potentiometer.

int LED_ERROR=26;// An output port that blinks an LED to indicate an error from the potentiometer.

const int buttonPin = 2;

const int ledPin = 4;

float frequency;
float frequency2;
SemaphoreHandle_t xButtonSemaphore;
// Function declarations

void Task_1(void *pvParameters);
void Task_2(void *pvParameters);
void Task_3(void *pvParameters);
void Task_4(void *pvParameters);
void Task_5(void *pvParameters);
void Task_6(void *pvParameters);
void Task_7(void *pvParameters);

// Timer function
void outputTimerCallback(TimerHandle_t xTimer) {
  //  The code implementation for the logic of your timer callback function..
}



void setup(void) {
  xButtonSemaphore = xSemaphoreCreateBinary();
  pinMode(LED_1, OUTPUT); //  Configure pin 2 as an output for Task 1.
  pinMode(LED2_FREQ, INPUT); // Configure pin 2 as an output for Task 1
  pinMode(LED3_FREQ, INPUT); // Configure pin 2 as an input for Task 3.
  pinMode(POT_T4, INPUT); //  Set pin 2 as an input for Task 4
  pinMode(LED_ERROR, OUTPUT); // Configure the LED pin as an output for Task 4.
  pinMode(buttonPin, INPUT_PULLUP);// Set the button pin as an input for Task 6.
  pinMode(ledPin, OUTPUT);// Configure the LED pin as an output for Task 6
  //  Set all elements in the readings array to 0 to initialize it
  Serial.begin(9600);
  // Task creation
  xTaskCreate(Task_1, "Task_1", 1024, NULL, 1, NULL);
  xTaskCreate(Task_2, "Task_2", 1024, NULL, 1, NULL);
  xTaskCreate(Task_3, "Task_3", 1024, NULL, 1, NULL);
  xTaskCreate(Task_4, "Task_4", 1024, NULL, 1, NULL);
  xTaskCreate(Task_5, "Task_5", 1024, NULL, 1, NULL);
  xTaskCreate(Task_6, "Task_6", 1024, NULL, 1, NULL);
  xTaskCreate(Task_7, "Task_7", 1024, NULL, 1, NULL);

  // Timer creation
  TimerHandle_t outputTimer = xTimerCreate("OutputTimer", pdMS_TO_TICKS(4), pdTRUE, (void *)0,
outputTimerCallback);
  xTimerStart(outputTimer, 0);
}

void loop() {
  // Empty - all tasks
//Task_1();
//Task_2();
//Task_3();
//Task_4();
//Task_5();
//Task_6();
//Task_7();
}
// Task 1, takes 0.9ms
void Task_1(void *pvParameters)
{
  Serial.println("Task 1 Start");
  for (;;){
```

```
  digitalWrite(LED_1, HIGH); //  Set pin 2 to the high state for 200 microseconds
  delayMicroseconds(200);
  digitalWrite(LED_1, LOW); //  Set pin 2 to the low state for 50 microseconds
  delayMicroseconds(50);
  digitalWrite(LED_1, HIGH); //  Set pin 2 to the high state for 30 microseconds
  delayMicroseconds(30);
  digitalWrite(LED_1, LOW); //  Keep pin 2 in the low state for the remaining time period
  }
  Serial.println("Task 1 Done");
}

void Task_2(void *pvParameters)
{
  Serial.println("Task 2 Start");
  for (;;){
   #define SAMPLES 10 //  Specify the number of samples to be taken
   int count = 0;
   for (int i = 0; i < SAMPLES; i++)
  {
    count += pulseIn(LED2_FREQ, HIGH); // Measure the duration of the pulse width of the input signal
  }
  count = count*2;
  frequency = 1000000.0 / (count / SAMPLES); // Compute the frequency of the signal in hertz (Hz)
  frequency = constrain(frequency, 333, 1000); //  Limit the frequency within the range of 333 Hz to 1000
Hz
  //int scaled_frequency = map(frequency, 333, 1000, 0, 99); //  Scale the frequency to a range of 0 to 99 for
further processing.
  //Serial.println("Frequency_1:"); //  Send the calculated frequency value to the serial port for output.
  //Serial.println(frequency); //Send the calculated frequency value to the serial port for output.
  }
  Serial.println("Task 2 Done");
}

void Task_3(void *pvParameters)
{
  Serial.println("Task 3 Start");
  for (;;){
  #define SAMPLES 8 //  Specify the number of samples to be taken
  int count2 = 0;
  for (int i = 0; i < SAMPLES; i++) {
    count2 += pulseIn(LED3_FREQ, HIGH); //  Measure the duration of the pulse width of the input signal.
  }
 count2 = count2*2;
  frequency2 = 1000000.0 / (count2 / SAMPLES); //  Compute the frequency of the signal in hertz (Hz)
  frequency2 = constrain(frequency2, 500, 1000); //  Limit the frequency within the range of 500 Hz to 1000
Hz.
  //int scaled_frequency2 = map(frequency2, 500, 1000, 0, 99); // Scale the frequency to a range of 0 to 99
for further processing.

  //Serial.println("Frequency_2:"); //  Send the calculated frequency value to the serial port for output.
  //Serial.println(frequency2); //  Send the calculated frequency value to the serial port for output.
  }
  Serial.println("Task 3 Done");
}

void Task_4(void *pvParameters)
{
```

```
Serial.println("Task 4 Start");
for (;;){
const int maxAnalogIn = 1023;
const int numReadings = 4;
int readings[numReadings];
int index = 0;
int total = 0;
int filteredValue = 0;
for (int i = 0; i < numReadings; i++)
{
readings[i] = 0;
}

// Read  the analog input value
int analogValue = analogRead(POT_T4);
//  Subtract the oldest reading from the running total
total -= readings[index];
//  Add the new reading to the running total.
total += analogValue;
//  Save the new reading in the readings array
readings[index] = analogValue;
//  Increase the index by one
index++;
// the index exceeds the number of readings, wrap around to the beginning of the array.
if (index >= numReadings)
 {
  index = 0;
}
//  Calculate the filtered value as the average of all the readings
filteredValue = total / numReadings;
```

```
  // If the filtered value is greater than half of the maximum range, turn on the LED
  if (filteredValue > maxAnalogIn / 2) {
    digitalWrite( LED_ERROR, HIGH);
    //Serial.println("error led HIGH");


  } else {
    digitalWrite( LED_ERROR, LOW);
    //Serial.println("error led LOW");


  }
  //  Transmit the filtered value to the serial port for output
  Serial.println(filteredValue);
  }
  Serial.println("Task 4 Done");
}

void Task_5(void *pvParameters)
{
  Serial.println("Task 5 Start");
  for (;;){
  int Task_2Freq = 0;
  int Task_3Freq = 0;
  Task_2Freq = map(frequency, 333, 1000, 0, 99);
  //  Scale and limit the frequency value to a range of 0 to 99.
  Task_3Freq = map(frequency2, 500, 1000, 0, 99);
  //  Transmit the frequency values to the serial port for output
  Serial.println(Task_2Freq);//To print frequency of given waveform of Task_2
  Serial.println(Task_3Freq);//To print frequency of given waveform of Task_3
  }
  Serial.println("Task 5 Done");
```

```
}

void Task_6(void *pvParameters)
{
  Serial.println("Task 6 Start");
  int buttonState = digitalRead(buttonPin);
  int lastButtonState = buttonState;
  unsigned long lastDebounceTime = 0;
  const unsigned long debounceTime = 50;

  for (;;) {
    int reading = digitalRead(buttonPin);

    if (reading != lastButtonState) {
      lastDebounceTime = millis();
    }

    if ((millis() - lastDebounceTime) > debounceTime) {
      if (reading != buttonState) {
        buttonState = reading;

        if (buttonState == LOW) {
          digitalWrite(ledPin, !digitalRead(ledPin));
          xSemaphoreGive(xButtonSemaphore);
        }
      }
    }

    lastButtonState = reading;
    vTaskDelay(pdMS_TO_TICKS(10));
  }
  Serial.println("Task_6 Done");
}

void Task_7(void *pvParameters)
{
  Serial.println("Task_7 Start");
  bool ledState = false;

  for (;;) {
    if (xSemaphoreTake(xButtonSemaphore, portMAX_DELAY) == pdTRUE) {
      ledState = !ledState;
      digitalWrite(ledPin, ledState ? HIGH : LOW);
    }
  }
  Serial.println("Task_7 Done");
}
```

GitHub link -