Sistemas Computacionais Embebidos

ANO LECTIVO: 2023/2024

CURSO: ENG. ELETROTÉCNICA E DE COMPUTADORES

REGIME: D

TRABALHO DE AVALIAÇÃO 1

- ESP32
- DHT11
- LDR
- Display gráfico
- 4 LEDS, 2 Resistências 1k, 1 Botão

AUTORES:

NOME: Rodrigo Duarte

Nº ALUNO: 2210984

Índice

Índice de Figuras

Índice de Tabelas

1. Introdução

O objetivo deste trabalho é simular as luzes externas de um veículo como as luzes de piscas, os faróis dianteiros e a luz de travão. Vai ser usado um LDR para ligar as luzes automaticamente consoante a luz ambiente, um potenciómetro para alterar a intensidade da luz, um LCD para mostrar informações relevantes como a temperatura e a intensidade dos faróis dianteiros (Mínimos, Médios e Máximos) e um DHT11 para obter a temperatura. Este trabalho foi realizado no âmbito da unidade curricular de Sistemas Computacionais Embebidos. Para a sua implementação foi utilizado o seguinte material:

- ESP32:
    - Controlador;
- 3 *Switchs*:
    - Piscas esquerda;
    - Piscas direita;
    - Quatro Piscas;
- 1 Botão:
    - Travão;
- Sensor Board da aula:
    - LDR, para ligar/desligar luzes consoante a iluminação;
    - Potenciómetro, para regular intensidade;
    - DHT11, para obter temperatura;
- TFT ST7789:
    - Mostrar a unidade curricular, nome do aluno, temperatura, e o estado dos faróis;
- 4 LEDs:
    - 2 LEDs amarelos, para os picas;
    - 1 LED azul, para os faróis;
    - 1 LED vermelho, para o travão;

2. Funcionamento

Neste capítulo vai ser explicado o funcionamento do sistema desenvolvido.

O microcontrolador utilizado foi o ESP32, este recebe os valores do sensor DHT por I2C, do LDR e do Potenciómetro por ADC e envia os dados para o Display por SPI. Do sistema FreeRTOS, foram utilizados os seguintes serviços: Tasks, Task Deletion, Message Queues, Interrupts Management e Semaphores. Podemos verificar o funcionamento no diagrama de blocos (figura 1) abaixo e uma fotografia do sistema desenvolvido.

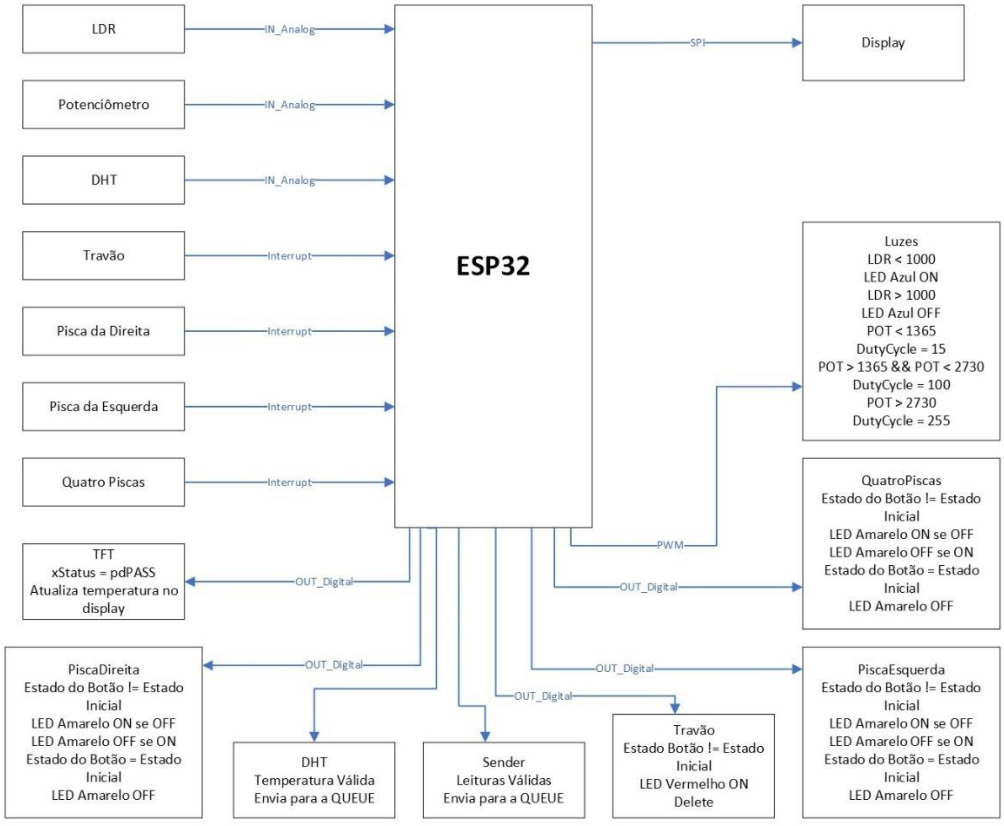

Figura 1 - Diagrama de Blocos

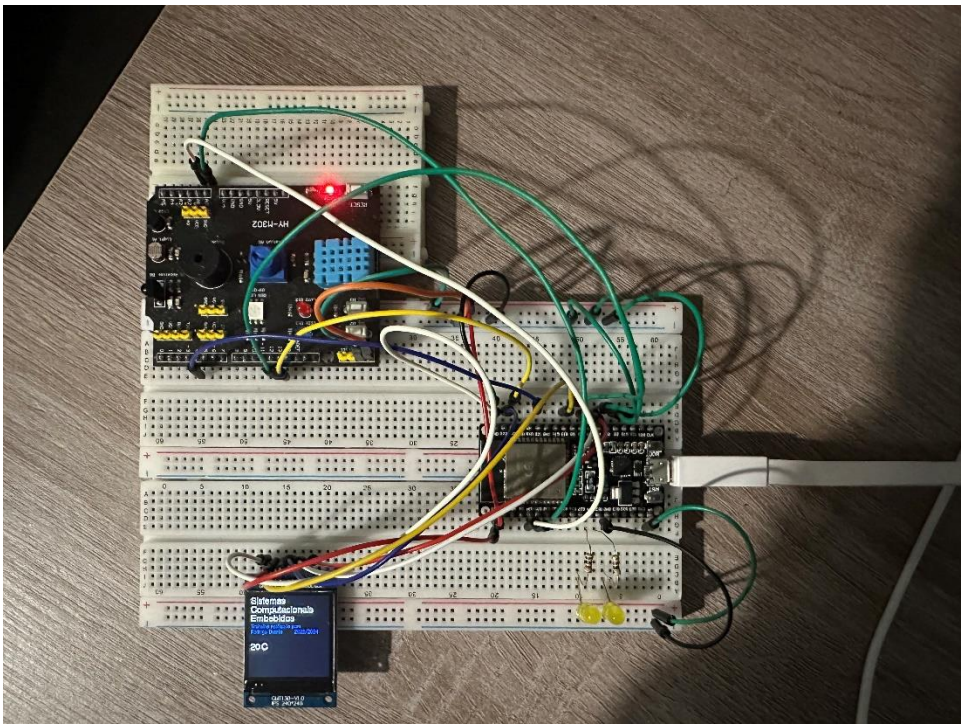

Figura 2 - Sistema Desenvolvido

2.1 Tarefas (Tasks)

De maneira a implementar este sistema foram utilizadas 8 tarefas (Tasks) de modo a implementar o sistema em tempo real. Na seguinte tabela podemos verificar as tarefas e as suas prioridades. Estas serão explicadas nos subcapítulos seguintes.

Tarefa	Prioridade
vTaskPiscaDireita	3
vTaskPiscaEsquerda	3
vTaskQuatroPiscas	3
vTaskTravao	2
vTaskTFT	2
vTaskDHT	1
vLuzesTask	2
vTaskSender	1

Tabela 1 – Tarefas

2.1.1 vTaskTFT

Esta tarefa inicializa o TFT, e escreve o nome da unidade curricular, o nome do aluno e o ano letivo. Recebe também os valores da QueueDHT e se o valor for bem recebido atualiza o display com o respetivo valor.

2.1.2 vTaskDHT

Esta tarefa obtém os valores de temperatura do DHT (figura 3) e envia o respetivo valor para uma QueueDHT. Esta tarefa ocorre de 5 em 5 segundos.

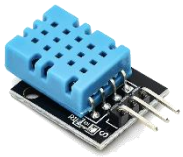

Figura 3 - DHT11

2.1.3 vLuzesTask

Esta tarefa recebe os valores do LDR e do Potenciómetro através da QueueSender e se os valores forem bem recebidos, liga um LED azul consoante a iluminação ambiente, se o valor obtido for inferior a 1000, o LED fica ligado e se for maior que esse valor fica apagado. Para a intensidade da luz, foi obtido o valor máximo do potenciómetro e foi dividida a escala em 3. De 0 a 1365, a intensidade do LED será baixa para simular os mínimos de um carro, de 1365 a 2730, a intensidade será média, simulando os médios, e para valores maiores que 2730 a intensidade será máxima simulando os máximos. Para isso foi utilizado um PWM com diferentes valores de DutyCycle para cada intervalo do Potenciómetro.

Para indicar o utilizador no nível luminoso e o estado do LED é utilizada um circulo verde no display para os estados “Minimos” e “Médios”, um circulo azul para o estado “Máximos” e o circulo desaparece quando o LED fica desligado.

2.1.4 vTaskSender

Esta tarefa verifica se os valores obtidos do potenciómetro e do LDR através de leituras analógicas, são válidos, enviando assim os valores para uma QueueSender que serão recebidos na tarefa vLuzesTask. Para isso foi utilizada uma sensor board (figura 4).

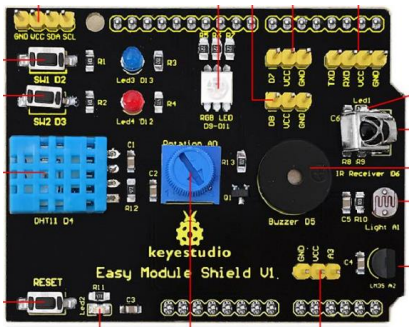

Figura 4 - Sensor Board

2.2 Task Deletion

Para este serviço foi criado um handler, vInterruptTravao, para uma interrupção associada ao PIN 0, ou seja, ao botão de boot do ESP32. Ao clicar no botão a interrupção é ativada criando a vTaskTravao. Nesta tarefa, foi utilizado um semáforo binário, que ao ser iniciada faz um "Take" ao Semáforo e enquanto o botão estiver a ser premido é ligado um LED vermelho para simular as luzes do travão.

Quando deixarem de premir o botão esta tarefa é apagada utilizando o vTaskDelete().

2.3 Message Queues

Os valores obtidos por cada sensor e potenciómetro, são enviados para outras Tasks através de duas Queues.

A QueueDHT é usada para enviar os valores do sensor DHT para a Task do TFT, para atualizar o Display.

A QueueSender é usada para enviar os dados do Potenciómetro e do LDR para a tarefa Luzes, para simular os Faróis dianteiros do carro. Esta Queue utiliza uma estrutura de dados que contém duas variáveis para armazenar cada sensor.

2.4 Interrupts Management e Semaphores

Para cada interrupção é utilizado um Semáforo Binário. Ao ser efetuada a interrupção o respetivo handler, faz um "Give" ao semáforo e este espera por um "Take" que será efetuado na sua respetiva tarefa. Cada interrupção é efetuada utilizando os PINS 2 para o pisca da direita, 4 para o pisca da esquerda e 15 para os quatro piscas, tendo cada uma um handler associado: vInterruptPiscaDireita, vInterruptPiscaEsquerda, vInterruptQuatroPiscas.

Foram utilizadas 3 tarefas para cada pisca vTaskPiscaDireita, vTaskPiscaEsquerda e vTaskQuatroPiscas. Cada tarefa efetua um "Take" ao semáforo e consequentemente enquanto o estado do Switch for diferente do estado inicial, o respetivo LED amarelo fica a piscar com 500ms de intervalo, simulando o pisca. Quando o estado for igual o LED apaga.

3. Conclusões

Inicialmente, pretendia colocar umas setas verdes para simular os piscas como nos carros, mas essa ideia foi descartada, pois ocupava muita memória para tentar estar sempre a escrever no TFT.

Não foi conseguido realizar proteções referentes às interrupções, ou seja, se for ativado um pisca enquanto outro estiver a correr eles correm os dois ao mesmo tempo, em vez de parar um para o outro correr.

Tirando isso não foram encontradas grandes dificuldades na realização do trabalho.

Github Gist Link:

- https://gist.github.com/RDuarte2/80eabefddfde37dd6d823114356323bd

Youtube Link:

- https://youtube.com/shorts/lQt5SdZXXH8

4. Código Fonte

```
#include "Arduino.h"
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_system.h"
#include "nvs_flash.h"
#include "esp_task_wdt.h"
#include <Adafruit_Sensor.h>
#include <DHT.h>
#include <DHT_U.h>
#include <SPI.h>
#include <TFT_eSPI.h>  // Hardware-specific library

// Pin Definitions
#define ledDireita 12
#define piscaDireita 2
#define ledEsquerda 14
#define piscaEsquerda 4
#define quatroPiscas 15
#define LDRPin 34
#define POTPin 35
#define PinTravao 0
#define ledTravao 5
#define ledFarois 22

// ADC Definitions
#define ADC_POT 35
#define ADC_LDR 34
#define ADC_RESOLUTION 12
#define VREF_PLUS 3.3
#define VREF_MINUS 0.0

// DHT Definitions
#define DHTPIN 21
#define DHTTYPE DHT11
DHT dht(DHTPIN, DHTTYPE);

// TFT
TFT_eSPI tft = TFT_eSPI();  // Invoke custom library
/*
-----TFT PINS--------
TFT_MISO 19
```

```
TFT_MOSI 23  //SDA
TFT_SCLK 18
TFT_CS   15  // Chip select control pin
TFT_DC   16  // Data Command control pin
TFT_RST  17  // Reset pin (could connect to RST pin)
*/

// Task Delarations
static void vTaskPiscaDireita(void *pvParameters);
static void vTaskPiscaEsquerda(void *pvParameters);
static void vTaskQuatroPiscas(void *pvParameters);
static void vTaskTravao(void *pvParameters);
static void vTaskTFT(void *pvParameters);
static void vTaskDHT(void *pvParameters);
static void vLuzesTask(void *pvParameters);
static void vTaskSender(void *pvParameters);  //LDR e POT values

// Interrupt Handlers
static void IRAM_ATTR vInterruptPiscaDireita(void);
static void IRAM_ATTR vInterruptPiscaEsquerda(void);
static void IRAM_ATTR vInterruptQuatroPiscas(void);
static void IRAM_ATTR vInterruptTravao(void);

// RTOS Task handlers
TaskHandle_t vTaskPiscaDireita_handle;
TaskHandle_t vTaskPiscaEsquerda_handle;
TaskHandle_t vTaskQuatroPiscas_handle;
TaskHandle_t vTaskTravao_handle;
TaskHandle_t vTaskDHT_handle;
TaskHandle_t vLuzesTask_handle;
TaskHandle_t vTaskSender_handle;
TaskHandle_t vTaskTFT_handle;

//Semaphores
SemaphoreHandle_t xBinarySemaphoreDireita;
SemaphoreHandle_t xBinarySemaphoreEsquerda;
SemaphoreHandle_t xBinarySemaphoreQuatro;
SemaphoreHandle_t xBinarySemaphoreTravao;

//Queues
QueueHandle_t xQueueSender;
QueueHandle_t xQueueDHT;

// Struct definition for Queue data
typedef struct {
```

```
  int POT_value;
  int LDR_value;
} data_t;

void setup(void) {

  vTaskPrioritySet(NULL, configMAX_PRIORITIES - 1);

  // Serial Initialization
  Serial.begin(115200);

  // TFT Initialization
  tft.init();

  // Semaphore Creation
  vSemaphoreCreateBinary(xBinarySemaphoreDireita);
  vSemaphoreCreateBinary(xBinarySemaphoreEsquerda);
  vSemaphoreCreateBinary(xBinarySemaphoreQuatro);
  vSemaphoreCreateBinary(xBinarySemaphoreTravao);

  // Queue Creation
  xQueueSender = xQueueCreate(5, sizeof(data_t));
  xQueueDHT = xQueueCreate(3, sizeof(int));

  // ADC Resolution
  analogReadResolution(ADC_RESOLUTION);

  // Pin Initializations as Inputs or Outputs
  pinMode(piscaDireita, INPUT_PULLUP);
  pinMode(ledDireita, OUTPUT);
  pinMode(ledEsquerda, OUTPUT);
  pinMode(piscaEsquerda, INPUT_PULLUP);
  pinMode(quatroPiscas, INPUT_PULLUP);
  pinMode(PinTravao, INPUT_PULLUP);
  pinMode(ledTravao, OUTPUT);
  pinMode(ledFarois, OUTPUT);

  // PWM Initialization
  ledcSetup(1, 5000, 8);
  ledcAttachPin(ledFarois, 1);

  // Attach the Interrupt to the Pins
  attachInterrupt(digitalPinToInterrupt(piscaDireita),
&vInterruptPiscaDireita, CHANGE);
```

```
  attachInterrupt(digitalPinToInterrupt(piscaEsquerda),
&vInterruptPiscaEsquerda, CHANGE);
  attachInterrupt(digitalPinToInterrupt(quatroPiscas),
&vInterruptQuatroPiscas, CHANGE);
  attachInterrupt(digitalPinToInterrupt(PinTravao), &vInterruptTravao,
FALLING);

  // Task creation
  xTaskCreatePinnedToCore(vTaskTFT, "TFT", 1024, NULL, 2, NULL, 1);
  if (xBinarySemaphoreDireita != NULL) {
    xTaskCreatePinnedToCore(vTaskPiscaDireita, "PiscaDireita", 1024, NULL,
3, &vTaskPiscaDireita_handle, 1);
  }
  if (xBinarySemaphoreEsquerda != NULL) {
    xTaskCreatePinnedToCore(vTaskPiscaEsquerda, "PiscaEsquerda", 1024, NULL,
3, &vTaskPiscaEsquerda_handle, 1);
  }
  if (xBinarySemaphoreQuatro != NULL) {
    xTaskCreatePinnedToCore(vTaskQuatroPiscas, "QuatroPiscas", 1024, NULL,
3, &vTaskQuatroPiscas_handle, 1);
  }
  if (xQueueDHT != NULL) {
    xTaskCreatePinnedToCore(vTaskDHT, "DHT", 1024, NULL, 1,
&vTaskDHT_handle, 1);
  }
  if (xQueueSender != NULL) {
    xTaskCreatePinnedToCore(vLuzesTask, "vLuzesTask", 1024, NULL, 2,
&vLuzesTask_handle, 1);
    xTaskCreatePinnedToCore(vTaskSender, "Sender", 1024, NULL, 1,
&vTaskSender_handle, 1);
  }
}

//------------------- TFT Task -------------------------
// This task updates the temperature data received from a queue
static void vTaskTFT(void *pvParameters) {
  // Variable to store the status of the queue
  portBASE_TYPE xStatus;
  // Variable to store the temperature data
  int temp = 0;

  // Set Black as background color
  tft.fillScreen(TFT_BLACK);

  // Set "cursor" at top left corner of display (0,0) and select font 4
```

```
  tft.setCursor(0, 4, 4);
  tft.setTextColor(TFT_WHITE);

  // Initialize Display
  tft.invertDisplay(true);
  tft.fillScreen(TFT_BLACK);
  tft.setCursor(0, 4, 4);
  tft.setTextColor(TFT_WHITE);
  tft.println(" Sistemas \n Computacionais \n Embebidos");
  tft.setTextFont(2);
  tft.setTextColor(TFT_RED);
  tft.println(" Trabalho realizado por: ");
  tft.println(" Rodrigo Duarte      2023/2024");

  for (;;) {
    // Checks if the queue exists and receive the values
    if (xQueueDHT != NULL) {
      xStatus = xQueueReceive(xQueueDHT, &temp, portMAX_DELAY);

      // Checks if the value was received
      if (xStatus == pdPASS) {
        tft.setCursor(0, 140, 4);
        tft.fillRectHGradient(0, 140, 60, 20, TFT_BLACK, TFT_BLACK);

        // Writes on the display
        tft.setCursor(0, 140, 4);
        tft.setTextColor(TFT_WHITE);
        tft.print(" ");
        tft.print(temp);
        tft.print(" °C");

      } else {
        // Print an error message if the task couldn't receive data from the
queue.
        Serial.println("`vTaskTFT` was unable to receive data from the
Queue");
      }
    } else {
      // Print an error message if the queue was not created successfully.
      Serial.println("Queue não foi criada com sucesso!");
    }
  }
}

//------------------- Right Indicator Task -------------
```

```
// This task blinks a led periodically with a 500 ms interval
static void vTaskPiscaDireita(void *pvParameters) {
  // take the semaphore after you create it so that the task waiting on this
semaphore will block until given by
  // another task.
  xSemaphoreTake(xBinarySemaphoreDireita, 0);
  // Variable to store the state of the switch when the task starts
  int state = digitalRead(piscaDireita);

  for (;;) {
    // Use the semaphore to wait for the event.
    xSemaphoreTake(xBinarySemaphoreDireita, portMAX_DELAY);
    // While the switch state is diferent from the one stored
    while (digitalRead(piscaDireita) == !state) {
      // Turn LED on or off
      digitalWrite(ledDireita, !digitalRead(ledDireita));
      // Print on the serial monitor
      Serial.print("Pisca Direita.\r\n");

      // The task will wake up every 500 milliseconds.
      vTaskDelay(500 / portTICK_PERIOD_MS);
    }
    // Turn the LED off and the arrow black
    digitalWrite(ledDireita, 0);
  }
}

//------------------- Left Indicator Task -------------
// This task blinks a led periodically with a 500 ms interval
static void vTaskPiscaEsquerda(void *pvParameters) {
  // take the semaphore after you create it so that the task waiting on this
semaphore will block until given by
  // another task.
  xSemaphoreTake(xBinarySemaphoreEsquerda, 0);
  // Variable to store the state of the switch when the task starts
  int state = digitalRead(piscaEsquerda);

  for (;;) {
    // Use the semaphore to wait for the event.
    xSemaphoreTake(xBinarySemaphoreEsquerda, portMAX_DELAY);
    // While the switch state is diferent from the one stored
    while (digitalRead(piscaEsquerda) == !state) {
      // Turn LED on or off
      digitalWrite(ledEsquerda, !digitalRead(ledEsquerda));
      // Print on the serial monitor
```

```
      Serial.print("Pisca Esquerda.\r\n");

      // The task will wake up every 500 milliseconds.
      vTaskDelay(500 / portTICK_PERIOD_MS);
    }
    // Turn the LED off and the arrow black
    digitalWrite(ledEsquerda, 0);
  }
}

//------------------- Four Indicator Task -------------
// This task blinks two leds periodically with a 500 ms interval
static void vTaskQuatroPiscas(void *pvParameters) {
  // take the semaphore after you create it so that the task waiting on this
semaphore will block until given by
  // another task.
  xSemaphoreTake(xBinarySemaphoreQuatro, 0);
  // Variable to store the state of the switch when the task starts
  int state = digitalRead(quatroPiscas);

  for (;;) {
    // Use the semaphore to wait for the event.
    xSemaphoreTake(xBinarySemaphoreQuatro, portMAX_DELAY);
    // While the switch state is diferent from the one stored
    while (digitalRead(quatroPiscas) == !state) {
      // Turn LEDs on or off
      digitalWrite(ledEsquerda, !digitalRead(ledEsquerda));
      digitalWrite(ledDireita, !digitalRead(ledDireita));
      // Print on the serial monitor
      Serial.print("Quatro Piscas.\r\n");

      // The task will wake up every 500 milliseconds.
      vTaskDelay(500 / portTICK_PERIOD_MS);
    }
    // Turn the LEDs off and the arrow black
    digitalWrite(ledEsquerda, 0);
    digitalWrite(ledDireita, 0);
  }
}

//------------------- DHT Task -------------
// This task reads the value from the DHT and sends it to a queue
static void vTaskDHT(void *pvParameters) {
  // Define a variable to hold the last wake time.
  TickType_t xLastWakeTime;
```

```
  // Initialize the last wake time with the current tick count.
  xLastWakeTime = xTaskGetTickCount();
  // Variable to store the temperature value
  int temp = 0;
  // DHT Initialization
  dht.begin();

  for (;;) {
    // Check if the temperature reading is valid (not NaN).
    if (isnan(dht.readTemperature())) {
      // Print an error message if the reading is invalid.
      Serial.println(F("Error reading temperature!"));
    } else {
      // Read temperature data from the DHT sensor.
      temp = dht.readTemperature();

      // Check if the DHT queue has available space.
      if (uxQueueSpacesAvailable(xQueueDHT) > 0) {
        Serial.println("Temperature Sent");
        // Send the temperature value to the DHT queue.
        xQueueSend(xQueueDHT, &temp, (TickType_t)0);
      }
    }
    // The task will wake up every 5000 milliseconds (5 seconds).
    vTaskDelayUntil(&xLastWakeTime, (5000 / portTICK_PERIOD_MS));
  }
}

//------------------- LDR and POT Task -------------
// This task reads the value from the LDR and Potentiometer and sends it to
a queue
static void vTaskSender(void *pvParameters) {
  // Define a variable to hold the last wake time.
  TickType_t xLastWakeTime;
  // Initialize the last wake time with the current tick count.
  xLastWakeTime = xTaskGetTickCount();
  // Variable to store the potentiometer value
  int analog_value_POT = 0;
  // Variable to store the LDR value
  int analog_value_LDR = 0;
  // Define a structure to store the data.
  data_t dados;

  for (;;) {
```

```
    // Check if the readings is valid (not NaN). ALTERADO || PARA && (TESTAR
EM CASA)
    if (isnan(analogRead(ADC_POT)) && isnan(analogRead(ADC_LDR))) {
      Serial.println(F("Failed to read from Potenciometer or LDR sensor!"));
    } else {
      // Read data from LDR and Potentiometer
      analog_value_POT = analogRead(ADC_POT);
      analog_value_LDR = analogRead(ADC_LDR);
      // Populate the Data structure with the values.
      dados.POT_value = analog_value_POT;
      dados.LDR_value = analog_value_LDR;

      // Check if the queue has available space.
      if (uxQueueSpacesAvailable(xQueueSender) > 0) {
        Serial.println("POT and LDR data sent.");
        // Send the sensor data to the alarm queue.
        xQueueSend(xQueueSender, (void *)&dados, (TickType_t)0);
      }
    }
    // The task will wake up every 2000 milliseconds (2 seconds).
    vTaskDelayUntil(&xLastWakeTime, (2000 / portTICK_PERIOD_MS));
  }
}

//------------------- Lights Task -------------
// This task receives the values from the QueueSender and lights a LED with
3 light levels
static void vLuzesTask(void *pvParameters) {
  // Variable to store the status of the queue
  portBASE_TYPE xStatus;
  // String to define the light level
  String luz;
  // Structure to store the sensor data
  data_t dados_rec;
  // Variable to store the values obtained
  int POT = 0, LDR = 0;
  // Variable to define the dutycycle
  int dutyCycle = 0;

  for (;;) {
    // Check if the queue exists
    if (xQueueSender != NULL) {
      xStatus = xQueueReceive(xQueueSender, &dados_rec, portMAX_DELAY);
      // Checks if the value was received
      if (xStatus == pdPASS) {
```

```
        // Reads the sensor data
        POT = dados_rec.POT_value;
        LDR = dados_rec.LDR_value;
        // Checks the value from the Potentiometer and defines a light level
based on the value
        if (POT < 1365) {
          dutyCycle = 15;
          luz = "Minimos";
        } else if (POT > 1365 && POT < 2730) {
          dutyCycle = 100;
          luz = "Medios";
        } else if (POT > 2730) {
          dutyCycle = 255;
          luz = "Maximos";
        }
        // If the ambient light is lower than 1000 lights a LED with the
dutycyle based on the Potentiometer value
        if (LDR < 1000) {
          Serial.print("Farois Ligados: ");
          Serial.println(luz);
          ledcWrite(1, dutyCycle);
          if (luz == "Maximos") {
            tft.fillCircle(120, 200, 15, TFT_RED);
          } else {
            tft.fillCircle(120, 200, 15, TFT_GREEN);
          }
        } else {
          ledcWrite(1, 0);
          tft.fillCircle(120, 200, 15, TFT_BLACK);
        }

      } else {
        // Print an error message if the task couldn't receive data from the
queue.
        Serial.println("`vLuzesTask` was unable to receive data from the
Queue");
      }
    } else {
      // Print an error message if the queue was not created successfully.
      Serial.println("Queue não foi criada com sucesso!");
    }
  }
}

//------------------- Brake Task -------------
```

```
// This task turns a Red LED on when a button is pressed the deletes himself
static void vTaskTravao(void *pvParameters) {

  for (;;) {

    // Wait indefinitely for the semaphore to be given.
    if (xSemaphoreTake(xBinarySemaphoreTravao, portMAX_DELAY) == pdTRUE) {
      // while the button is pressed
      while (digitalRead(PinTravao) == 0) {
        // Turn LED on
        digitalWrite(ledTravao, 1);
        // Write on the Serial Monitor
        Serial.println("Travao");
      }
      // Turn LED off
      digitalWrite(ledTravao, 0);
    }
    // Task Deletes himself
    Serial.println("Task Travao Apagada");
    vTaskDelete(NULL);
  }
}

//------------------- Right Indicator Handler -------------
// This is an interrupt handler function. It's executed when an interrupt
occurs.
static void IRAM_ATTR vInterruptPiscaDireita(void) {
  // Declare a variable to track if a higher-priority task should be woken.
  static signed portBASE_TYPE xHigherPriorityTaskWoken;

  // Initialize the flag for higher-priority task wake as false.
  xHigherPriorityTaskWoken = pdFALSE;

  // Give a semaphore from the interrupt context. This allows the waiting
task to proceed.
  xSemaphoreGiveFromISR(xBinarySemaphoreDireita, (signed portBASE_TYPE
*)&xHigherPriorityTaskWoken);

  // If a higher-priority task was woken, request a context switch again.
  if (xHigherPriorityTaskWoken == pdTRUE) {
    portYIELD_FROM_ISR();
  }
}

//------------------- Left Indicator Handler -------------
```

```
// This is an interrupt handler function. It's executed when an interrupt
occurs.
static void IRAM_ATTR vInterruptPiscaEsquerda(void) {
  // Declare a variable to track if a higher-priority task should be woken.
  static signed portBASE_TYPE xHigherPriorityTaskWoken;

  // Initialize the flag for higher-priority task wake as false.
  xHigherPriorityTaskWoken = pdFALSE;

  // Give a semaphore from the interrupt context. This allows the waiting
task to proceed.
  xSemaphoreGiveFromISR(xBinarySemaphoreEsquerda, (signed portBASE_TYPE
*)&xHigherPriorityTaskWoken);

  // If a higher-priority task was woken, request a context switch again.
  if (xHigherPriorityTaskWoken == pdTRUE) {
    portYIELD_FROM_ISR();
  }
}

//------------------- Warning Indicator Handler -------------
// This is an interrupt handler function. It's executed when an interrupt
occurs.
static void IRAM_ATTR vInterruptQuatroPiscas(void) {
  // Declare a variable to track if a higher-priority task should be woken.
  static signed portBASE_TYPE xHigherPriorityTaskWoken;

  // Initialize the flag for higher-priority task wake as false.
  xHigherPriorityTaskWoken = pdFALSE;

  // Give a semaphore from the interrupt context. This allows the waiting
task to proceed.
  xSemaphoreGiveFromISR(xBinarySemaphoreQuatro, (signed portBASE_TYPE
*)&xHigherPriorityTaskWoken);

  // If a higher-priority task was woken, request a context switch again.
  if (xHigherPriorityTaskWoken == pdTRUE) {
    portYIELD_FROM_ISR();
  }
}

//------------------- Brake Handler -------------
// This is an interrupt handler function. It's executed when an interrupt
occurs.
static void IRAM_ATTR vInterruptTravao(void) {
```

```
  // Declare a variable to track if a higher-priority task should be woken.
  static signed portBASE_TYPE xHigherPriorityTaskWoken;

  // Request a context switch to a higher-priority task.
  portYIELD_FROM_ISR();

  // Initialize the flag for higher-priority task wake as false.
  xHigherPriorityTaskWoken = pdFALSE;

  // Create a new task (vTaskTravao) pinned to core 1, the task is named
"TaskTravao"
  xTaskCreatePinnedToCore(vTaskTravao, "TaskTravao", 2048, NULL, 2,
&vTaskTravao_handle, 1);

  // Give a semaphore from the interrupt context. This allows the waiting
task to proceed.
  xSemaphoreGiveFromISR(xBinarySemaphoreTravao, (signed portBASE_TYPE
*)&xHigherPriorityTaskWoken);

  // If a higher-priority task was woken, request a context switch again.
  if (xHigherPriorityTaskWoken == pdTRUE) {
    portYIELD_FROM_ISR();
  }
}

// The main loop function.
void loop() {
  // Delete the current task
  vTaskDelete(NULL);
}
```