Universidade Federal de Juiz de Fora
Departamento de circuitos elétricos
Software Embarcado (CEL080)

Relatório Técnico - Sistema de Multimídia

Alunos: Guilherme Ferrara, Guilherme Santos, Pedro Victor Avila
Professor orientador: Leandro Manso

Juiz de Fora
2024

Universidade Federal de Juiz de Fora
Departamento de circuitos elétricos
Software Embarcado (CEL080)

Sistema de multimídia

Relatório final da matéria de software embarcado

Alunos: Guilherme Ferrara, Guilherme Santos,
Pedro Victor Avila

Professor orientador: Leandro Manso

Juiz de Fora
2024

Conteúdo

1 Introdução

O projeto consiste em um sistema de multimídia para veículos, possuindo as funções de: sensor de estacionamento, medindo a distância da parte traseira do carro e avisando sonoramente quando a ela vai se encurtando. Um sensor de luminosidade, que aciona os faróis assim que escurecer. Um sistema de localização GPS conectado a um satélite e para integrar todos esses recursos em uma interface central. Uma conexão bluettoh (Bluetooth Low Energy) para trasmitir os dados lidos para um aplicativo, que disponibilizará a distância medida durante a ré, a localização do carro em um mapa e comandará a ativação dos faróis quando estes não estiverem no modo automático.

O projeto foi desenvolvido utilizando o microcontrolador ESP-32, assim como o sistema operacional de tempo real FreeRTOS para coordenar a execução das tasks. Os periféricos de hardwre utilizados foram o sensor ultrassonic HC-SR04, um buzzer, um resistor dependente de luz (LDR), um Módulo GPS GY-NEO6MV2 com Antena. O recurso de Bluetooth já é integrado no microprocessador.

2 FreeRTOS

FreeRTOS é um sistema operacional de tempo real (RTOS) para sistemas embarcados. Ele é leve, eficiente e oferece suporte a multitarefa, permitindo que várias tarefas sejam executadas simultaneamente. FreeRTOS é amplamente utilizado em dispositivos de IoT e sistemas embarcados devido à sua flexibilidade e baixo consumo de recursos.

2.1 Queues

As filas (queues) no FreeRTOS são usadas para comunicação entre tarefas. Elas permitem que dados sejam enviados de uma tarefa para outra de forma segura e eficiente. As filas são criadas usando a função `xQueueCreate()`, que define o número de itens que a fila pode conter e o tamanho de cada item. Aqui está um exemplo das filas criadas:

```
QueueHandle_t fila1, fila2, fila3, fila4, fila5;

fila1 = xQueueCreate(1, sizeof(float));
fila2 = xQueueCreate(1, sizeof(float));
fila3 = xQueueCreate(1, sizeof(uint8_t));
fila4 = xQueueCreate(1, sizeof(float));
fila5 = xQueueCreate(1, sizeof(float));
```

As filas são usadas para enviar e receber dados entre tarefas. Por exemplo, uma tarefa pode enviar um valor para uma fila usando `xQueueSend()` e outra tarefa pode receber esse valor usando `xQueueReceive()`.

A fila1 envia a distância medida pelo sensor na taskLeitura, à função de leitura, que enviará o valor por Bluetooth. Não há tempo de espera caso essa fila esteja cheia, porque a função de leitura só será acessada quando for requisitado pelo aplicativo, que é feito em uma frequência bem menor que a leitura.

A fila2 também envia a distância medida, porém pra taskBuzzer. Esta fica em estado blocked até receber o dado.

A fila3 é uma fila do tipo uint8_t, que envia os bytes recebidos pela função de escrita pelo aplicativo, para a taskLDR, fazendo com que ela opere ou não através do conversor analógico-digital. Como essa task opera tem que operar mesmo sem os dados da fila, ela não é bloqueada por ela.

Já as filas 4 e 5 enviam, respectivamente, valores float de latitude e longitude, da taskGPS para a task de leitura, onde serão enviados por bluetooth.

2.2 Tasks

Essa aplicação possui tasks para realizar a leitura do sensor ultrassonico, que será a task de maior prioridade, porque deve operar continuamente. Além dela, existe a task que coordena o buzzer, fazendo operar para uma certa distância, uma task que faz a leitura da luminosidade e taks auxiliares para aplicações como bluetooth e gps.

2.3 App_main

```
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_event.h"
#include "nvs_flash.h"
#include "esp_log.h"
#include "esp_nimble_hci.h"
#include "nimble/nimble_port.h"
#include "nimble/nimble_port_freertos.h"
#include "host/ble_hs.h"
#include "services/gap/ble_svc_gap.h"
#include "services/gatt/ble_svc_gatt.h"
#include "sdkconfig.h"
#include "driver/gpio.h"
#include <freertos/semphr.h>
#include <driver/gpio.h>
```

```c
#include <freertos/queue.h>
#include <ultrasonic.h>
#include "driver/ledc.h"
#include "driver/adc.h"
#include "esp_err.h"
#include <stdlib.h>
#include "driver/adc.h"
#include "driver/ledc.h"
#include "driver/uart.h"

QueueHandle_t  fila1,fila2,fila3,fila4,fila5;

void app_main() //----------------------------------------
{
    //configure perifericos
    sensor.trigger_pin = TRIGGER_GPIO;
    sensor.echo_pin = ECHO_GPIO;
    ultrasonic_init(&sensor);

    init_hw();

        // Configure UART parameters
    uart_config_t uart_config = {
        .baud_rate = GPS_UART_BAUD_RATE,
        .data_bits = UART_DATA_8_BITS,
        .parity = UART_PARITY_DISABLE,
        .stop_bits = UART_STOP_BITS_1,
        .flow_ctrl = UART_HW_FLOWCTRL_DISABLE
    };
    // Set UART parameters
    uart_param_config(UART_NUM, &uart_config);
    // Set UART pins
    uart_set_pin(UART_NUM, GPS_UART_TX_PIN, GPS_UART_RX_PIN,
       UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE);
    // Install UART driver
    uart_driver_install(UART_NUM, BUF_SIZE * 2, 0, 0, NULL,
       0);

    gpio_set_direction(GPIO_NUM_0, GPIO_MODE_INPUT);   //
       boot button
    gpio_set_direction(GPIO_NUM_2, GPIO_MODE_OUTPUT);  // LED
        do ESP
    gpio_set_direction(GPIO_NUM_4, GPIO_MODE_OUTPUT);  //
       buzzer
    gpio_set_direction(GPIO_NUM_25, GPIO_MODE_OUTPUT); // led
        ldr
    connect_ble();

    // cria   o dos objetos
```

```
60      fila1 = xQueueCreate(1,sizeof(float));
61      fila2 = xQueueCreate(1,sizeof(float));
62      fila3 = xQueueCreate(1,sizeof(uint8_t));
63      fila4 = xQueueCreate(1,sizeof(float));
64      fila5 = xQueueCreate(1,sizeof(float));
65
66      // cria   o das tasks
67      xTaskCreate(boot_creds_clear, "boot_creds_clear", 2048,
            NULL, 1, NULL);
68      xTaskCreate(vTaskLeitura,       "LEITURA",          2048,
            NULL, 2, NULL);
69      xTaskCreate(vTaskBuzzer,        "BUZZ",             2048,
            NULL, 1, NULL);
70      xTaskCreate(vTaskLDR,            "LDR",             2048,
            NULL, 1, NULL);
71      xTaskCreate(vTaskGPS,            "GPS",             8192,
            NULL, 1, NULL);
72  }
```

Avaliando-se o escopo da função *app_main*(), inicialmente são configurados os periféricos que realizarão a aquisição dos dados necessários para a efetivação da implementação. Em sequência, foram inicializados os pinos do microcontrolador referentes ao sensor ultrassônico, ao ADC, ao protocolo de comunicação UART e, por fim, às GPIO's relacionadas ao Boot Button, LED pertencente ao ESP32, buzzer e LED conectado ao LDR. A função *connect_ble*() é responsável pela inicialização do Bluetooth Low Energy. Finalmente, são realizadas as definições dos objetos do Kernel. Foram utilizadas as filas de mensagens para a efetivação da comunicação entre tasks, e, ao final, a estruturação das tarefas em si.

A implementação do Sistema de Multimída se concentra no funcionamento de um recurso principal, o Bluetooth. É por meio deste protocolo, que as solicitações dos dados e exibição dos mesmos ocorre. Fica a cargo da task *boot_creds_clear* a definição de uma interface com o Boot Button, na qual o usuário interage com este último por 3 segundos e habilita a descoberta do ESP para uma possível efetivação de conexão. Após a comunicação inicial com o app do celular, as tarefas de leitura iniciam suas respectivas operações, e, a partir do intermédio feito pelas Filas de Mensagens, o aplicativo passa a receber os valores aferidos.

3 Recursos Utilizados

3.1 Bluetooth Low Energy (BLE) e Nimble

Para realizar a comunicação dos entre o ESP-32 e um celular, foi usado o recurso de Bluetooth Low Energy (BLE), já integrado no chip. Comparado ao Bluettoth Classico, o BLE apresenta um menor consumo de energia, porém sua velocidade de troca de dados é mais lenta. Para nossa aplicação, essa desvantagem não é relevante, já que o processamento e ativação dos periféricos é feito em grande parte pelo microcontrolador, deixando apenas o interfaceamento com usuário para disponibilização dos dados por Bluetooth.

Para utilização desse recurso foi utilizado um pilha de protocolos chama Nimble, que possibilitou que em nossa aplicação, o ESP configurado como servidor, anunciasse sua presença para outros dispositivos (usando a camada GAP), permitisse que dispositivos clientes conectem a ele e leiam dados (usando GATT e ATT) e gerencie a conexão.

A função a seguir, configura e inicializa o BLE no ESP utilizando o nimble stack. Nela, o sistema de armazenamento não volátil NVS é inicializado, para armazenar dados na memória flash quando necessário. Em seguida Nimble stack é inicializado para gerenciar a comunicação com outro dispositivo BLE e define o nome do dispositivo que será exibido ao ser descoberto por outros aparelhos, "BLE-Server". Os serviços de GAP (Generic Access Profile) e GATT (Generic Attribute Profile) são inicializados então, e servem para, respectivamente, gerenciar o processo de descoberta e conexão; e estruturar os dados trocados entre servidor e cliente. Então o serviço de callback é definido (ble_app_on_sync) e esta função será chamada após a sincronização de dispositivos, estando pronto então para transferência de dados e conexão. Por fim é feito uma inicialização da pilha Nimble no ambiente de FreeRTOS, fazendo com que a função 'host_task' seja executada como uma task pelo sistema operacional.

```
1 void connect_ble(void)
2 {
3     nvs_flash_init();
          // 1 - Initialize NVS flash using
4     nimble_port_init();
          // 3 - Initialize the host stack
5     ble_svc_gap_device_name_set("BLE-Server");
          // 4 - Initialize NimBLE configuration - server name
6     ble_svc_gap_init();
          // 4 - Initialize NimBLE configuration - gap service
7     ble_svc_gatt_init();
          // 4 - Initialize NimBLE configuration - gatt service
```

```
8       ble_gatts_count_cfg(gatt_svcs);
            // 4 - Initialize NimBLE configuration - config gatt
            services
9       ble_gatts_add_svcs(gatt_svcs);
            // 4 - Initialize NimBLE configuration - queues gatt
            services.
10      ble_hs_cfg.sync_cb = ble_app_on_sync;
            // 5 - Initialize application
11      nimble_port_freertos_init(host_task);
            // 6 - Run the thread
12  }
```

Como ja citado, a função 'host_task' é uma task que é reposnsável por rodar a stack BLE continuamente, garantindo que o sistema estaja pronto para processar eventos de conexão, desconexão e publicidade.

```
1   // The infinite task
2   void host_task(void *param)
3   {
4       nimble_port_run(); // This function will return only when
            nimble_port_stop() is executed
5   }
```

A task 'boot_creads_clear', verifica se o botão de boot do esp foi precionado por um período mínimo de 2 segundos. Caso isso ocorra, o processo de publicidade é iniciado, fazendo com que o ESP fique visível e conectável a outros dispositivos BLE. Isso ocorre comparando a variável m, que tem um contador de tempo iniciado no instante que o ESP é inicializado e que torna a se iniciar toda vez que o botão não está apertado. Quando ele é apertado, uma outra variável n, tem sua contagem de tempo iniciada. Por fim, os valores das duas variáveis são comparadas para ver se o botão esteve precionado por no mínimo 2 segundos. Existe também uma variável booleana 'status' que verifica se o o processo de publicidade está ocorrendo, para que a task mantenha desta forma quando entrar em execução pelo escalonador novamente.

```
1   void boot_creds_clear(void *param)
2   {
3
4       // printf("%lld\n", n - m);
5       int64_t m = esp_timer_get_time();
6       while (1)
7       {
8
9           if (!gpio_get_level(GPIO_NUM_0))
```

```
10          {
11              int64_t n = esp_timer_get_time();
12
13              if ((n - m) / 1000 >= 2000)
14              {
15                  ESP_LOGW("BOOT BUTTON:", "Button Pressed FOR
                        3 SECOND\n");
16
17                  adv_params.conn_mode = BLE_GAP_CONN_MODE_UND;
                        // connectable or non-connectable
18                  adv_params.disc_mode = BLE_GAP_DISC_MODE_GEN;
                        // discoverable or non-discoverable
19                  ble_gap_adv_start(ble_addr_type, NULL,
                        BLE_HS_FOREVER, &adv_params, ble_gap_event
                        , NULL);
20                  status = true;
21                  vTaskDelay(100);
22                  m = esp_timer_get_time();
23              }
24          }
25          else
26          {
27              m = esp_timer_get_time();
28          }
29          vTaskDelay(10);
30
31          if (status)
32          {
33              // ESP_LOGI("GAP", "BLE GAP status");
34              adv_params.conn_mode = BLE_GAP_CONN_MODE_UND; //
                    connectable or non-connectable
35              adv_params.disc_mode = BLE_GAP_DISC_MODE_GEN; //
                    discoverable or non-discoverable
36              ble_gap_adv_start(ble_addr_type, NULL,
                    BLE_HS_FOREVER, &adv_params, ble_gap_event,
                    NULL);
37          }
38      }
39  }
```

A função 'ble_gap_event' é chamada quando ocorrem eventos de conexão ou desconexão. O evento 'BLE_GAP_EVENT_CONNECT' registra a mensagem 'OK' quando o evento de conexão for concluído com sucesso e chama 'ble_app_advertise' caso contrário para reiniciar o processo. Quando o dispositivo é desconextado, o evento 'BLE_GAP_EVENT_DISCONNECT' ocorre e reinicia então o processo.

```c
// BLE advertise()event handling
static int ble_gap_event(struct ble_gap_event *event, void *
    arg)
{
    switch (event->type)
    {
    // Advertise if connected
    case BLE_GAP_EVENT_CONNECT:
        ESP_LOGE("GAP", "BLE␣GAP␣EVENT␣CONNECT␣%s", event->
            connect.status == 0 ? "OK!" : "FAILED!");
        if (event->connect.status != 0)
        {
            ble_app_advertise();
        }
        break;
    // Advertise again after completion of the event
    case BLE_GAP_EVENT_DISCONNECT:
        ESP_LOGE("GAP", "BLE␣GAP␣EVENT␣DISCONNECTED");
        if (event->connect.status != 0)
        {
            ble_app_advertise();
        }
        break;
    case BLE_GAP_EVENT_ADV_COMPLETE:
        ESP_LOGW("GAP", "BLE␣GAP␣EVENT");
        ble_app_advertise();
        break;
    default:
        break;
    }
    return 0;
}
```

Como já dito, a função 'ble_app_on_sync' será executada após a sincronização. Primeiramente o tipo de endereço será definido automaticamente e depois a função 'ble_app_advertise' é chamada para iniciar a publicidade.

```c
// The application
void ble_app_on_sync(void)
{
    ble_hs_id_infer_auto(0, &ble_addr_type); // Determines
        the best address type automatically
    ble_app_advertise();                     // Define the
        BLE connection
}
```

A função 'ble_app_advertise' configura o modo de publicidade BLE, definindo os campos e parâmetros necessários para que o dispositivo possa ser descoberto e conectado. A função lê o nome do dispositivo usando ble_svc_gap_device_name() e o armazena nos campos de publicidade (fields.name). Ela também define os parâmetros de publicidade, como conexão e descoberta, para configurar como este dispositivo será visível a outros.

```
1  void ble_app_advertise(void)
2  {
3      // GAP - device name definition
4      struct ble_hs_adv_fields fields;
5      const char *device_name;
6      memset(&fields, 0, sizeof(fields));
7      device_name = ble_svc_gap_device_name(); // Read the BLE
           device name
8      fields.name = (uint8_t *)device_name;
9      fields.name_len = strlen(device_name);
10     fields.name_is_complete = 1;
11     ble_gap_adv_set_fields(&fields);
12
13     // GAP - device connectivity definition
14     // struct ble_gap_adv_params adv_params;
15     memset(&adv_params, 0, sizeof(adv_params));
16     // adv_params.conn_mode = BLE_GAP_CONN_MODE_UND; //
           connectable or non-connectable
17     // adv_params.disc_mode = BLE_GAP_DISC_MODE_GEN; //
           discoverable or non-discoverable
18     // ble_gap_adv_start(ble_addr_type, NULL, BLE_HS_FOREVER,
           &adv_params, ble_gap_event, NULL);
19 }
```

Os dados trocados entre o cliente, são então estruturados em um array que define os serviços GATT e associa suas características. O serviço é definido como primário, o que indica ser o principal serviço forncido, e então o UUID de 16 bits '0xDEED' é definido para identificar esse serviço. Tem-se então três características, que estão atreladas a esse serviço, possuindo cada uma, um characteristic UUID, uma flag que indica qual tipo de característica (leitura, escrita) e o callback da função atrelada a essa característica, que é chamada quando o cliente lê-la. O array de características termina com um elemento nulo 0 para indicar o final da lista e o array de serviços termina também com um elemento nulo 0 para indicar o final da lista de serviços.

```
// Array of pointers to other service definitions
// UUID - Universal Unique Identifier
static const struct ble_gatt_svc_def gatt_svcs[] = {
    {.type = BLE_GATT_SVC_TYPE_PRIMARY,
     .uuid = BLE_UUID16_DECLARE(0xDEED), // Define UUID for
        device type
     .characteristics = (struct ble_gatt_chr_def[]){
         {.uuid = BLE_UUID16_DECLARE(0xFEF4), // Define UUID
            for reading
          .flags = BLE_GATT_CHR_F_READ,
          .access_cb = device_read},
         {.uuid = BLE_UUID16_DECLARE(0xDEAD), // Define UUID
            for writing
          .flags = BLE_GATT_CHR_F_WRITE,
          .access_cb = device_write},
         {.uuid = BLE_UUID16_DECLARE(0xDEAF), // Define UUID
            for reading
          .flags = BLE_GATT_CHR_F_READ,
          .access_cb = device_read_coordenades},
         {0}}},
    {0}};
```

Por fim, temos as duas funções de read e uma de write. 'device_read' recebe através da fila1 a distância medida pelo sensor, e então esse valor é formatado por um buffer para ser enviado para o cliente. Em 'device_read_coordenades', o mesmo ocorre, porém os valores são armazenados dentro de uma string antes de serem enviados. Na função de escrita 'device_write', os bytes recebidos são interpretados e colocados na fila3, onde serão obtidos pela taskLDR.

```
 // Read data from ESP32 defined as server
static int device_read(uint16_t con_handle, uint16_t
   attr_handle, struct ble_gatt_access_ctxt *ctxt, void *arg)
{
    float dado;

    xQueueReceive(fila1, &dado, portMAX_DELAY);

    //ESP_LOGW("BLE", "Distancia = %f m", dado);

    os_mbuf_append(ctxt->om , &dado, sizeof(dado));

    return 0;
}
```

```
 static int device_read_coordenades(uint16_t con_handle,
    uint16_t attr_handle, struct ble_gatt_access_ctxt *ctxt,
    void *arg)
{

    xQueueReceive(fila4,&Latitude,0);
    xQueueReceive(fila5,&Longitude,0);

    ESP_LOGW("GPS-BLE","Coordenadas recebidas %f , %f",
       Latitude,Longitude);

    char coord_string[50];

    sprintf(coord_string, "%.15f, %.15f", Latitude, Longitude
       );

    os_mbuf_append(ctxt->om, coord_string, strlen(
       coord_string));

    ESP_LOGE("GPS-BLE","Coordenadas enviadas %s",coord_string
       );

    return 0;
}
```

```
// Write data to ESP32 defined as server
static int device_write(uint16_t conn_handle, uint16_t
   attr_handle, struct ble_gatt_access_ctxt *ctxt, void *arg)
{
    uint8_t *data = (uint8_t *)ctxt->om->om_data;

    if (data[0] == 1)
    {
        //ESP_LOGW("BLE", "RECEBA! Valor: %d", data[0]);
        xQueueSendToBack(fila3,&data[0],0);

    }else if (data[0] == 0)
    {
        //ESP_LOGW("BLE", "RECEBA! Valor: %d", data[0]);
        xQueueSendToBack(fila3,&data[0],0);
    }
    else
    {
        ESP_LOGW("BLE", "Erro ao receber mensagem! Valor: %d"
           , data[0]);
    }

```

```
21      //printf("%d %s\n", strcmp(data, (char *)"LIGHT ON") ==
            0, data);
22
23      return 0;
24  }
```

3.1.1 Aplicativo para celular

Para conseguir ler os dados enviados por Bluetooth, assim como comandar ações pelo uso do celular, foi desenvolvido um aplicativo através da aplicação de código aberto MIT APP Inventor. Nele estão disponibilizados a distância medida pelo sensor, a localização e também a opção de colocar os faroís em modo automático ou ligado.

Inicialmente, foram definidos os UUIDs tanto do serviço quanto de cada característica, referentes aos serviços e características do servidor, como é mostrado na imagem 1.

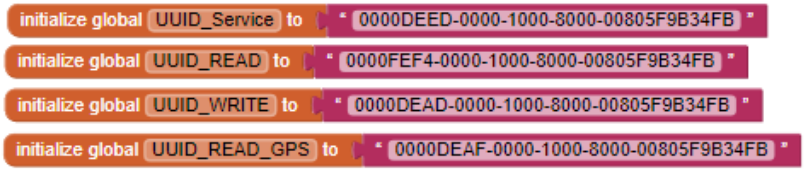


Figura 1: UUIDs definidos no projeto do aplicativo

Em seguida, foi criado um clock para que funções de leitura sejam executadas periodicamente e as características sejam requisitadas no BLE server.

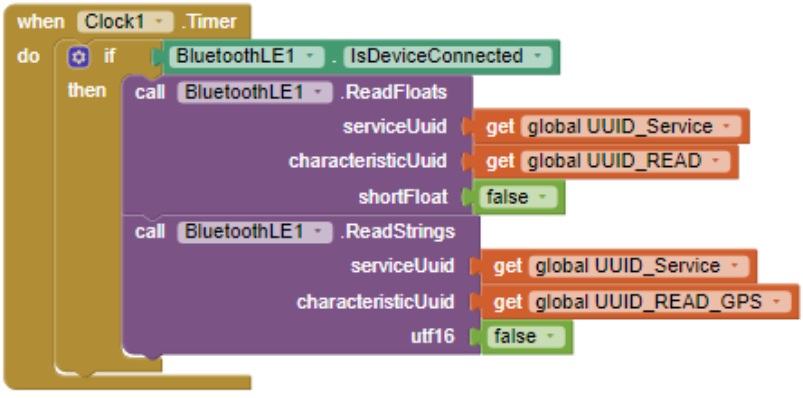


Figura 2: Leitura dos dados do BLE server

A função de escrita, é ativada com um interruptor, enviando bytes para o BLE server interpretar.

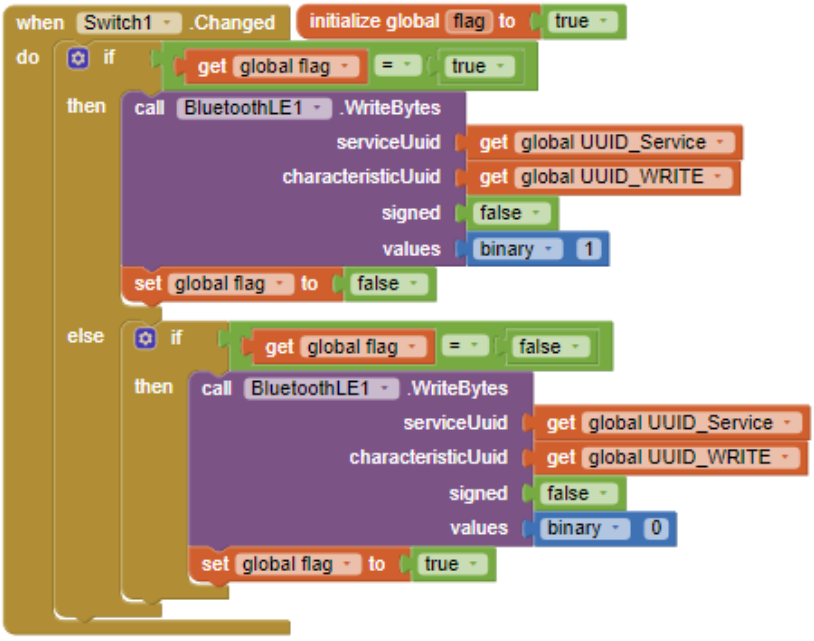


Figura 3: Envio de dados para o BLE server

As demais funções do aplicativo, como conexão, visualização de elementos podem sem encontrados no anexo.

3.2 Sensor Ultrassonico HC-SR04

O sensor ultrassônico HC-SR04 é amplamente utilizado para medir distâncias de forma precisa. Ele funciona emitindo pulsos ultrassônicos e medindo o tempo que esses pulsos levam para retornar após refletirem em um objeto.O HC-SR04 é um sensor versátil e fácil de usar, ideal para projetos que requerem medição de distância precisa.

3.2.1 Componentes e Conexões

O HC-SR04 possui quatro pinos:

- **VCC**: Alimentação (5V)
- **GND**: Terra
- **TRIG**: Pino de disparo do pulso ultrassônico

- **ECHO**: Pino de recepção do pulso refletido

3.2.2 Princípio de Funcionamento

1. O pino **TRIG** envia um pulso de 10 microsegundos.
2. O sensor emite um sinal ultrassônico de 40 kHz.
3. O sinal reflete em um objeto e retorna ao sensor.
4. O pino **ECHO** mede o tempo de retorno do sinal.
5. A distância é calculada pela fórmula:

$$\text{Distância} = \frac{\text{Tempo de retorno} \times \text{Velocidade do som}}{2} \tag{1}$$

3.2.3 Task Ultrassonico

Com a ajuda de uma biblioteca já pronta, podemos somente configurar o sensor ultrassonico de acordo com os parâmetros necessários para a nossa tarefa. As seguintes bibliotecas são utilizadas no código em relação ao sensor ultrassonico:

- `ultrasonic.h`: Contém as definições e funções específicas para o sensor ultrassônico.
- `esp_timer.h`, `ets_sys.h`: Responsáveis por fornecer funcionalidades de temporização e controle de hardware específico.

```
1  #include <ultrasonic.h>
2
3  #define TRIGGER_GPIO 18
4  #define ECHO_GPIO 5       //pinagem da conexao feita pelo
       sensor
5
6  #define MAX_DISTANCE 3 //distancia de leitura maxima
7
8  ultrasonic_sensor_t sensor;
9
10 void vTaskLeitura (void *pvparameters);
11
12 void app_main() //-----------------------------------------
13 {
14     //configure perifericos
15     sensor.trigger_pin = TRIGGER_GPIO;
```

```
16      sensor.echo_pin = ECHO_GPIO;
17      ultrasonic_init(&sensor);
18
19      // cria   o dos objetos
20      fila1 = xQueueCreate(1,sizeof(float));
21      fila2 = xQueueCreate(1,sizeof(float));
22
23      xTaskCreate(vTaskLeitura,       "LEITURA",          2048,
            NULL, 2, NULL);
24  }
25
26  void vTaskLeitura (void *pvparameters)
27  {
28      float distancia;
29      float last_distance=0;
30      int cnt=0;
31
32      while (1)
33      {
34
35          ultrasonic_measure(&sensor, MAX_DISTANCE_CM, &
                distancia);
36
37          //ESP_LOGI("LEITURA","Distancia = %f m",distancia);
38
39          if (distancia < last_distance / 2)
40          {
41              cnt++;
42              if (cnt >= 2)
43              {
44                  last_distance = distancia;
45                  cnt = 0;
46              }
47              //break;
48          }
49          else
50          {
51              last_distance = distancia;
52              cnt = 0;
53          }
54
55          xQueueSendToBack(fila1,&last_distance,0);
56          vTaskDelay(pdMS_TO_TICKS(100));
57          xQueueSendToBack(fila2,&last_distance,portMAX_DELAY);
58          vTaskDelay(pdMS_TO_TICKS(100));
59
60      }
61
62  }
```

3.3 Buzzer

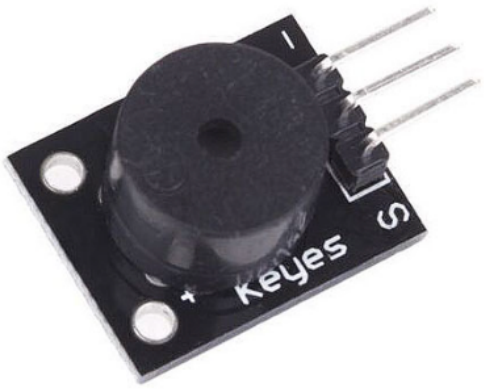


Figura 4: Módulo Buzzer passivo. Modelo KY-006.

Para a produção de sinal sonoro que emula àquele instalado em carros com sensor de baliza, foi utilizado um módulo buzzer passivo.

3.3.1 Sensor de Baliza: Efeito Sonoro

```
1     void vTaskBuzzer (void *pvparameters)
2 {
3     float dis;
4
5     while(1)
6     {
7        xQueueReceive(fila2,&dis,portMAX_DELAY);
8        //ESP_LOGI("BUZZER","recebeu dado distancia = %f",dis)
             ;
9        //vTaskDelay(pdMS_TO_TICKS(100));
10
11         if(dis <= BUZZ)
12         {
13              gpio_set_level(GPIO_NUM_4,1);
14              vTaskDelay(pdMS_TO_TICKS(dis*250));
15              gpio_set_level(GPIO_NUM_4,0);
16              vTaskDelay(pdMS_TO_TICKS(dis*250));
17         }
18    }
19
20 }
```

Com os valores de distância provenientes das mediçõs realizadas pelo sensor ultrassônico, foi produzida uma lógica que gera um sinal no pino D4 no qual está conectado o pino SIGNAL do buzzer. Esta lógica varia a duração dos estímulos realizados com base na distância aferida. Desta forma, a medida que o carro se aproxima de um obstáculo no momento em que realiza a baliza, a frequência de apito do buzzer aumenta.

3.4 Módulo GPS

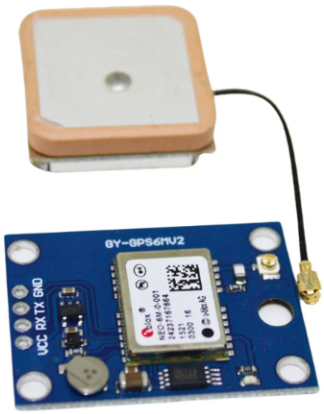


Figura 5: Módulo GPS Neo-6M com antena inclusa. Modelo: GY-NEO6MV2.

Para a aquisição de dados referentes à localização do veículo, foi utilizado um módulo GPS com suporte ao protocolo de comunicção UART. O receptor GPS recebe um sinal transmitido por um satélite localizado na órbita terrestre, e, a partir deste sinal inicial, realiza uma comparação com dados enviados de mais dois satélites para validar a posição do objeto no globo terrestre na forma de latitude e longitude. Este processo é chamado de triangulação.

Cada satélite conta com um relógio atômico, desta forma a transmissão de dados ocorre de forma periódica. Esta característica confere precisão ao GPS inferior a 100 metros, o que o torna muito útil no rastreamento de veículos.

O módulo GPS utilizado na implementação é configurado para a recepção de dados estruturados no formato NMEA. Dentro deste modelo existem diversos padrões de mensagem, sendo o NMEA-0183 o mais usual na comunicação de dispositivos receptores GPS. O NMEA-0183 tem como característica um protocolo específico que permite a comunicação de dados serial. Dentro deste padrão, as mensagens recebidas possuem formatação GPGGA.

Com relação a conexão do módulo GPS ao ESP32: Para a comunicação serial foram utilizados os pinos relativos às GPIO16 e GPIO17, como RX e TX, respectivamente. A alimentação do módulo foi feita pelo pino 3v3 e GND comum.

3.4.1 GPS: Aquisição de Dados

Inicialmente, para configurar a comunicação serial UART, foram utilizadas as funções listadas abaixo, provenientes da biblioteca *driver* *uart.h*.

```
        // Set UART parameters
    uart_param_config(UART_NUM, &uart_config);
    // Set UART pins
    uart_set_pin(UART_NUM, GPS_UART_TX_PIN, GPS_UART_RX_PIN,
       UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE);
    // Install UART driver
    uart_driver_install(UART_NUM, BUF_SIZE * 2, 0, 0, NULL,
       0);
```

```
static void vTaskGPS(void *pvParameters) {

uint8_t data[BUF_SIZE];
int length;

while(1){

// Le os dados via UART
length = uart_read_bytes(UART_NUM, data, BUF_SIZE, 100 /
   portTICK_PERIOD_MS);
if (length > 0) {
// Printa os dados no monitor
data[length] = '\0'; // Null-terminate the string
ESP_LOGE(TAG2, "GPS Data: %s", data);


char *nmea_message = (char *)data;


extract_lat_long(nmea_message, lat_value, lon_value);

// Extrai os sinais de latitude e longitude
float latitude = convert_to_decimal(lat_value, lat_value[
   strlen(lat_value) - 1], 2); // Latitude tem 2 digitos para
```

```
    graus
float longitude = convert_to_decimal(lon_value, lon_value[
    strlen(lon_value) - 1], 3); // Longitude tem 3 digitos
    para graus

// Imprime os valores extraidos
printf("Latitude: %.8f\n", latitude);
printf("Longitude: %.8f\n", longitude);

xQueueSendToBack(fila4,&latitude,0);
xQueueSendToBack(fila5,&longitude,0);


vTaskDelay(pdMS_TO_TICKS(1000));
}
}
}
```

A aquisição dos dados enviados pelo GPS é realizada dentro da rotina da Task vTaskGPS. Em seu escopo é feita a leitura dos dados recebidos via comunicação serial pela função *uart_read_bytes*, esta operação ocorre com periodicidade. Em seguida, é definida uma variável do tipo ponteiro para char com nome *nmea_message*, essa variável irá abrigar os dados recebidos pelo GPS.

A função *exctract_lat_long* tem como argumentos, respectivamente: a string da qual será copiado o seu conteúdo e as strings de latitude e longitude nas quais serão armazenados os valores específicos a elas. Por fim é feita a conversão dos dados de char para float, e, após, o envio destes via Fila de Mensagens para o APP.

```
void extract_lat_long(const char *nmea_message, char *
    latitude, char *longitude) {

// Copiando a mensagem para evitar modificar o original
//char nmea_copy[512];
//strcpy(nmea_copy, nmea_message);


// Tokenizar a mensagem NMEA por sentencas separadas por '\n'
char *sentence = strtok(nmea_message, "\n");

while (sentence != NULL) {
// Verifica se a sentenca e do tipo GPGGA
if (strstr(sentence, "$GPGGA") != NULL) {
// Tokenizar a sentenca NMEA usando ',' como delimitador
```

```
15 char *token = strtok(sentence, ",");
16 int field_count = 0;
17
18 // Identifica o tipo de sentenca e extrai latitude e
      longitude
19 while (token != NULL) {
20 field_count++;
21 if (field_count == 3) {  // Campo de latitude
22 strcpy(latitude, token);
23 } else if (field_count == 4) {  // N/S indicador
24 strcat(latitude, token);  // Anexar N ou S
25 } else if (field_count == 5) {  // Campo de longitude
26 strcpy(longitude, token);
27 } else if (field_count == 6) {  // E/W indicador
28 strcat(longitude, token);  // Anexar E ou W
29 }
30
31 token = strtok(NULL, ",");
32 }
33 // Saimos do loop apos encontrar e processar a sentenca GPGGA
34 break;
35 }
36 // Pega a proxima sentenca na mensagem
37 sentence = strtok(NULL, "\n");
38
39 }
40
41 }
```

Dentro do escopo da função *extract_lat_long*, a mensagem NMEA, entregue pelo GPS, será destrinchada em substrings, ou tokens. Primeiro é verificado se o parâmetro de entrada *nmea_message* possui substrings em seu conteúdo, operação realizada pela função *strtok*. Se a verificação é positiva, a seguir é verificado se a sentença a ser lida corresponde ao tipo de mensagem avaliada, no caso, GPGGA. Por fim, após a última validação, a mensagem é separada e conferida às variáveis responsáveis pelos tipos específicos de dados (latitude, longitude, etc.).

```
1     // Funcao para converter a latitude e longitude em
         formato decimal com sinal
2 double convert_to_decimal(const char *value, char direction,
     int degree_length) {
3     // Separar graus e minutos da latitude/longitude
4     double degrees = 0.0;
5     double minutes = 0.0;
6     // Ajusta a leitura dependendo se e latitude (2 graus) ou
```

```
        longitude (3 graus)
    char degree_part[4] = {0};
    strncpy(degree_part, value, degree_length); // Copia os
        primeiros caracteres como graus
    degrees = atof(degree_part); // Converte para double
    minutes = atof(value + degree_length); // Converte o
        restante para minutos

    // Converter para decimal
    double decimal = degrees + (minutes / 60.0);

    // Aplicar sinal com base na direcao
    if (direction == 'S' || direction == 'W') {
        decimal = -decimal;
    }
    return decimal;
}
```

3.5 Sensor de luminosidade

Neste trabalho, foi utilizado um fotoresistor em conjunto do conversor Analógico/Digital já presente no esp32. Onde o valor lido pelo esp 32 era proporcional à quantidade de luz incidida.

Figura 6: Sensor LDR.

3.5.1 Fotoresistor

Um fotoresistor (ou resistor dependente de luz, LDR, ou) é um resistor variável controlado pela luz. A resistência de um fotoresistor diminui com o aumento da intensidade da luz incidente, ou seja, ele exibe fotocondutividade. Um fotoresistor pode ser aplicado em circuitos detectores sensíveis à luz entre outras aplicações.

Funcionamento: No escuro, um fotoresistor pode ter uma resistência de até vários megaohms (M Ω) , enquanto na luz, a resistência pode ser tão baixa quanto algumas centenas de ohms.A faixa de resistência e a sensibilidade de um fotoresistor podem diferir substancialmente entre dispositivos diferentes. Porém no nosso caso usando parâmetros da sala de aula ajustamos os valores de acordo para a demonstração.

3.5.2 ADC

O ADC tem parâmetros bem específicos para ser utilizado, porém a própria espressif tem explicações em seu site sobre como usá-los:

- **Canais e Modos**: O ESP32 possui dois ADCs de 12 bits, ADC1 (8 canais) e ADC2 (10 canais). ADC1 é usado para leituras gerais(que foi o utilizado nesse trabalho), enquanto ADC2 é compartilhado com o módulo Wi-Fi, o que pode causar falhas de leitura durante o uso do Wi-Fi.

- **Configuração**: Antes de realizar leituras, é necessário configurar o ADC, incluindo a largura de leitura e a atenuação. A atenuação ajusta a faixa de tensão de entrada que o ADC pode medir.

- **Leitura e Calibração**: As leituras são feitas com funções como `adc1_get_raw()` e `adc2_get_raw()`. Para maior precisão, é recomendável usar a calibração para corrigir variações na tensão de referência do ADC.

3.5.3 Task LDR

Depois da inicialização com as definições explicadas anteriormente, fazemos a leitura do valor e comparamos com o valor que "padronizamos"como noite (LUM 450), e valores abaixo desse estipulado acenderiam o farol automaticamente:

```
    static const adc1_channel_t adc_channel = ADC_CHANNEL_4;
       //pino D32 de leitura

    adc1_config_width(ADC_WIDTH_BIT_10);
    adc1_config_channel_atten(adc_channel, ADC_ATTEN_DB_11);
       //inicializacao dos parametros necessarios
}

    //Task Farol

    void vTaskLDR (void *pvparameters)
{
    uint8_t comando = 0;

    while(1)
    {
        xQueueReceive(fila3,&comando,0);
        //ESP_LOGI("LDR","Mensagem recebida: %d",comando);

        uint32_t adc_val = 0;
        for (int i = 0; i < SAMPLE_CNT; ++i)
        {
            adc_val += adc1_get_raw(adc_channel);
        }
        adc_val /= SAMPLE_CNT;

        //ESP_LOGI("LDR","Valor medido : %d",(int)adc_val);

        if(comando == 1)
        {
            gpio_set_level(GPIO_NUM_25,1);
            comando = 1;
        }
        else{
            if(adc_val <= LUM)
            {
            gpio_set_level(GPIO_NUM_25,1);
            }
            else{
                gpio_set_level(GPIO_NUM_25,0);
            }
        }

        vTaskDelay(pdMS_TO_TICKS(500));
    }
}
```

4 Resultados

Ao fim do projeto, temos então um sensor de ré que funciona constantemente juntamente com o buzzer, indicando sonoramente ao usuário a distância. Quanto à essa aplicação, o grupo constatou que para a distância esperada, que é de no máximo aproximadamente 2 metros, o sensor ultrassonico apresenta algumas medições erradas, que ativam o buzzer sem que seja necessário. Para isso, foi feito um algorítimo que compara a medição atual com a distância medida imediatamente antes, e se ela for menor que a metade da anterior, ela é descartada. Isso reduz consideravelmennte esses errors de medição, ao mesmo tempo que não altera em nada o propósito do projeto, já que esse descarte só se aplica a medições isoladas, e se a distância for realmente encurtada, ela será dada por mais de uma amostra, e a amostra descartada não fará diferença.

Além do sensor ultrassonico e o buzzer, o sensor de luminosidade (que é a parte do código que controla o led "farol"a partir da leitura da tensão variada pelo LDR), também funciona integralmente. Portanto, ambas estas funções do sistema de multimídia funcionam independentes da conexão bluetooth. Vale ressaltar que a parte do código que está ligada ao GPS também funciona constantemente, mas a leitura dos dados só é feita pelo aparelho conectado bia bluetooth.

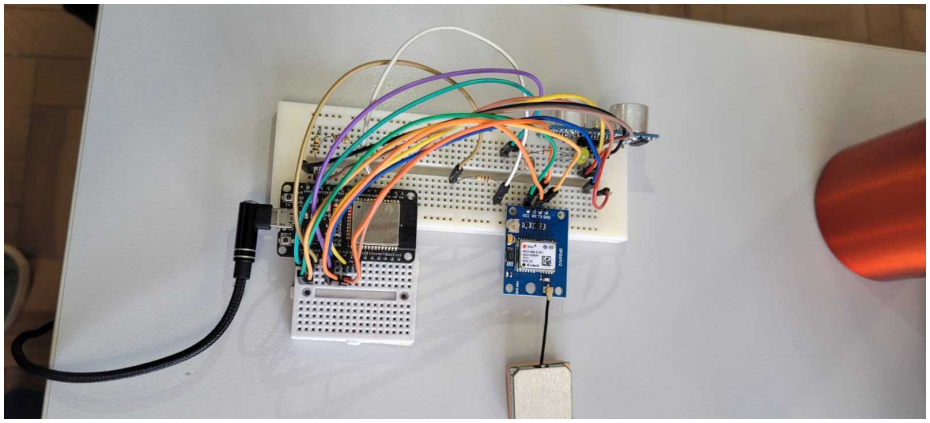


Figura 7: Portótipo final do projeto

Para as demais aplicações então, deve-se pressionar o botão boot por no mínimo 2 segundos, isso tornará o ESP visível aos outros aparelhos BLE. Na imagem 9a podemos ver o aplicativo assim que aberto. Deve-se então habilitar o bluettoh do dispositivo e apertar o botão scan. Irá abrir então uma lista de dispositivos BLE disponíveis para conexão, e então deve-se selecionar o dispositivo "BLE-Server", nome ao qual foi definido a conexão

de nossa aplicação. Após a conexão bem sucedida, temos então as opções de 'Desconectar', 'Sensor de ré' e 'GPS'. Ao clicarmos em 'Sensor de ré' a distância medida é exibida para o usuário, como vemos em 9b. O interruptor 'Farol Automático' define se o faról será aceso permanentemente, quando ligado como na figura 9c, ou ficará no modo automático, acendendo apenas quando há baixa presença de luz no ambiente, como em 9d. O interruptor no modo ligado, resulta na imagem 8.

Por fim, ao clicar no botão 'GPS' um mapa é aberto, centralizando nas coordenadas enviadas pelo módulo GPS e exibindo a latitude e longitude abaixo. Esse posicionamento é atualizado constantemente e o mapa oferece uma opção de zoon. Temos também o botão 'BACK' que fecha tanto a visualização da distância quanto o mapa.

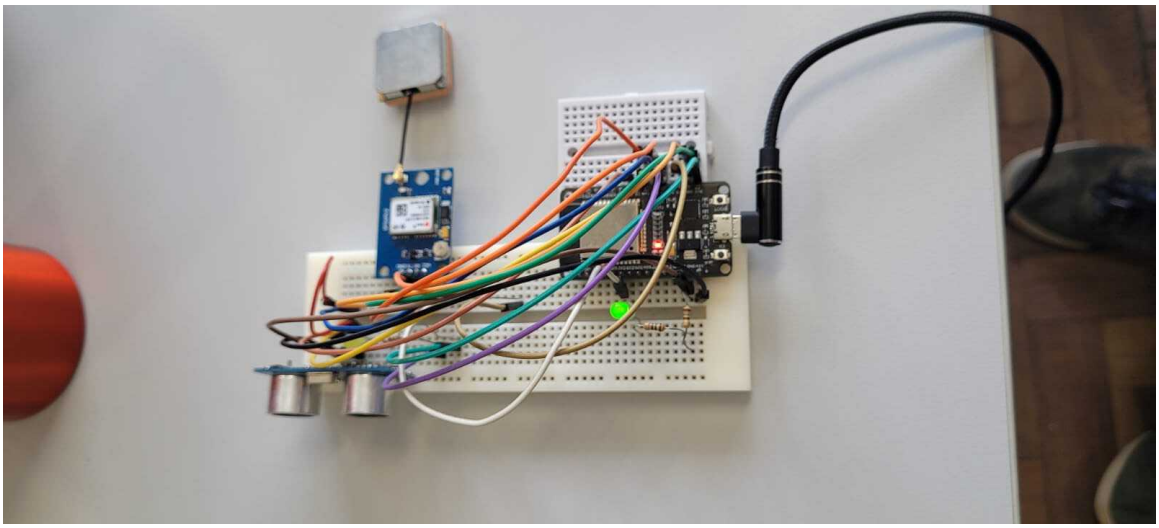
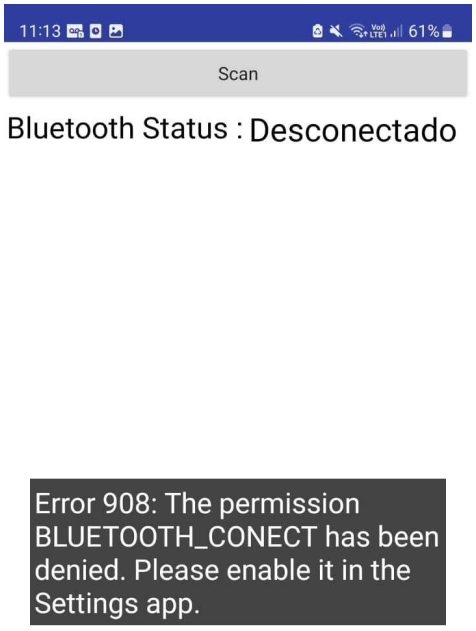
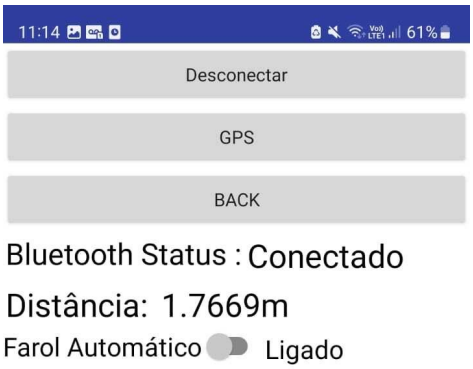


Figura 8: Led 'Farol' comandado pelo celular



(a) Aplicativo antes da conexão com ESP



(b) Leitura da distância

(c) Habilitação do farol modo ligado

(d) Habilitação do farol modo automático

Figura 9: Interface do aplicativo

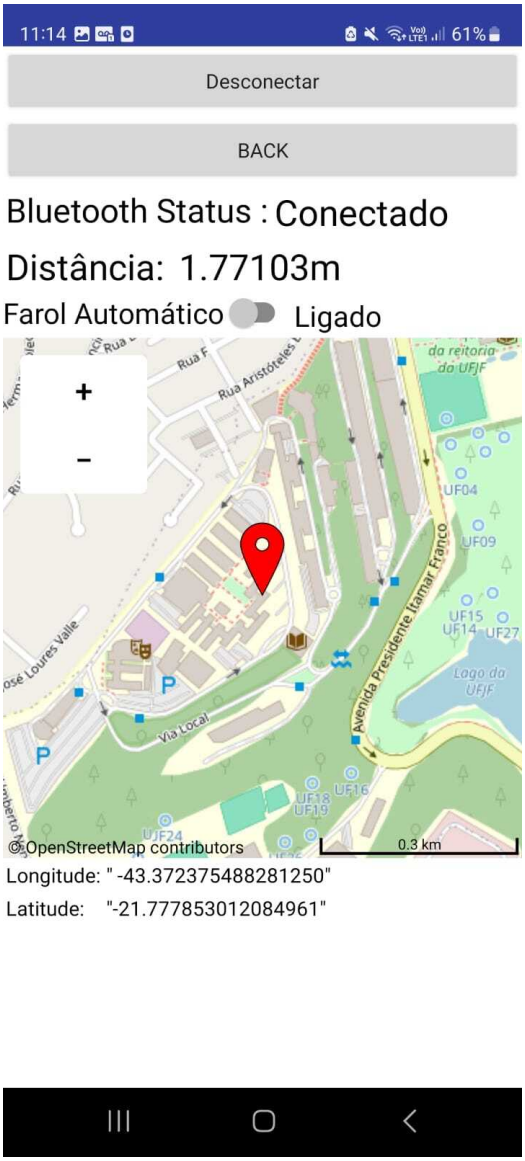


Figura 10: Exibição da localização

5 Conclusão

A utilização do freertos para sincronismo entre atividades distintas realizadas no microcontrolador e acesso restrito a memória compartilhada torna o seu uso fundamental para a execução da implementação da central multimídia do veículo, implementação esta realizada nesta atividade. A presença das filas de mensagem exerce um papel de grande importância na comunicação

entre as tasks de leitura e de processamento de dados.

A implementação do protocolo de comunicação bluetooth e do GPS se mostrou a etapa mais complexa para ser concluída. A comunicação via Bluetooth necessita de um processo de inicialização com diversas etapas. Por outro lado a programação do GPS se mostrou um tanto quanto difícil durante a realização da interface entre o receptor GPS e o microcontrolador. Este processo requeriu um tratamento bem detalhado quanto aos dados que eram recebidos.

Para a realização do aplicativo foi utilizado como ferramenta o site app monitor que disponibiliza diagrama de blocos para a organização de dados e tarefas a serem realizadas pelo aplicativo. Por fim pelo que pode ser avaliado a conclusão deste projeto necessitou do domínio de diversos softwares, periféricos e recursos presentes no freertos. Ademais, o empenho dos integrantes do grupo e a disposição de tempo e ferramentas de pesquisa e teste resultaram em um retorno positivo quanto a conclusão do trabalho.

Bibliografia

https://tecnoblog.net/responde/o-que-e-gps/
https://geoone.com.br/entendendo-nmea/
https://docs.espressif.com/projects/esp-idf/en/stable/esp32/apireference/bluetooth/esp_gatts.html
https://innovationyourself.com/esp32-bluetooth-low-energy-tutorial/
https://docs.espressif.com/projects/esp-idf/en/v5.3.1/esp32/index.html

Anexo

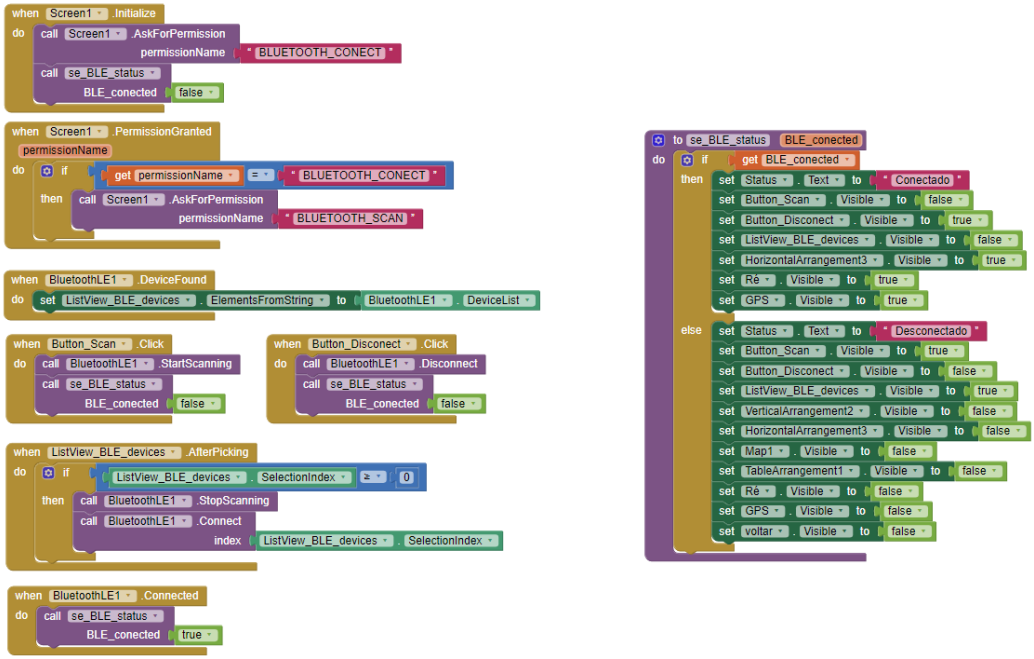


Figura 11: Conexão BLE

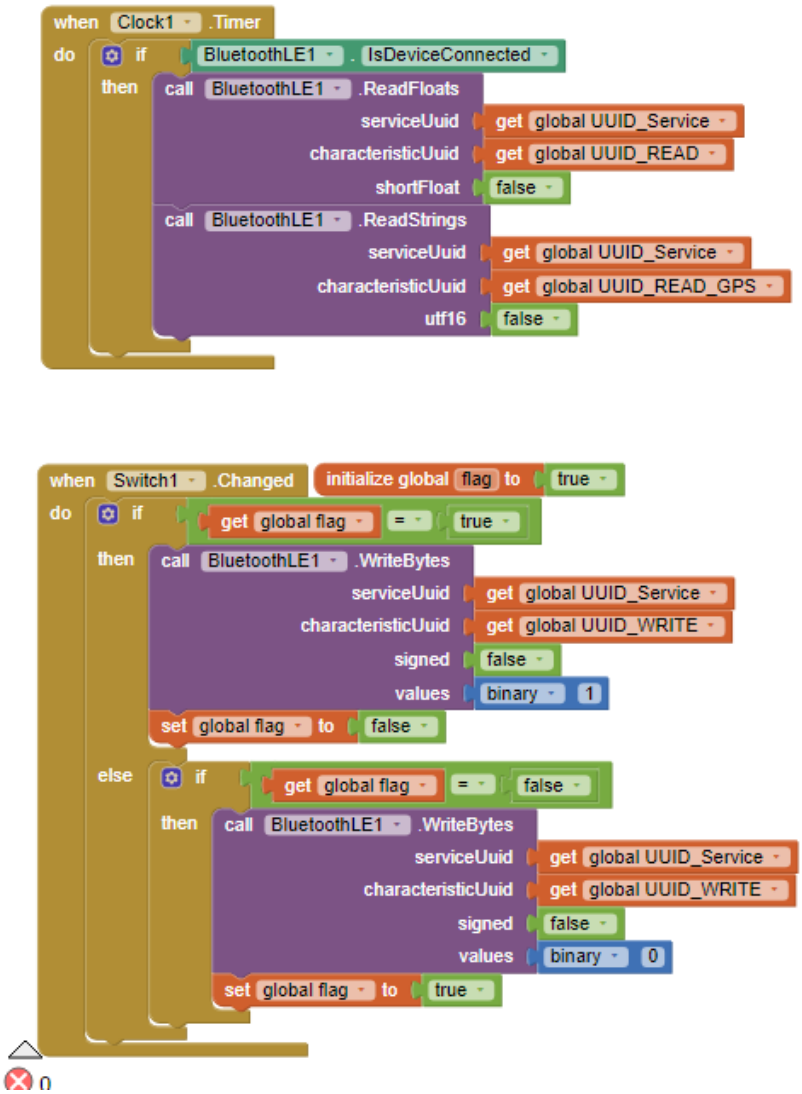


Figura 12: Leitura e escrita

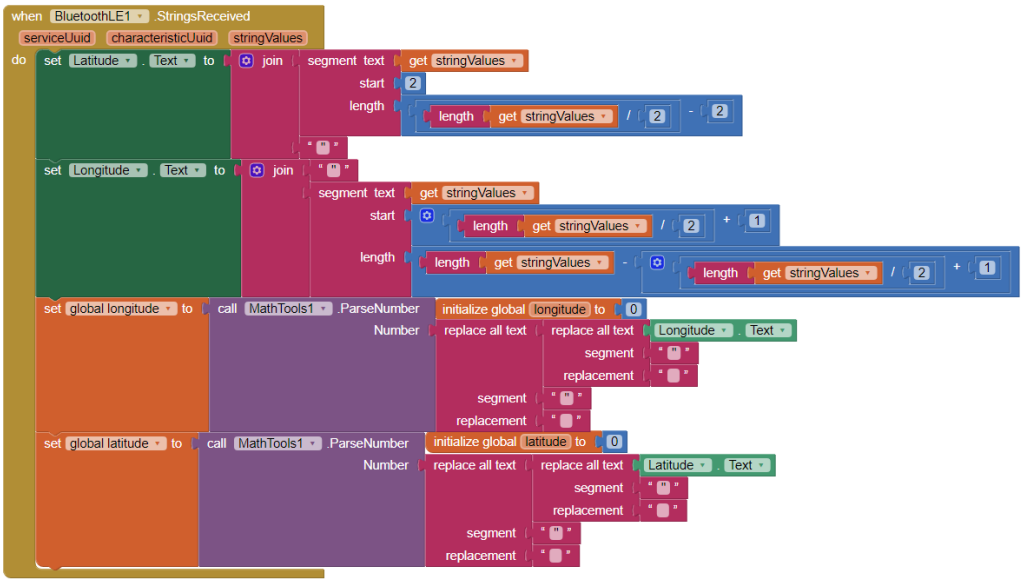


Figura 13: Formatação dos dados recebidos para exibição

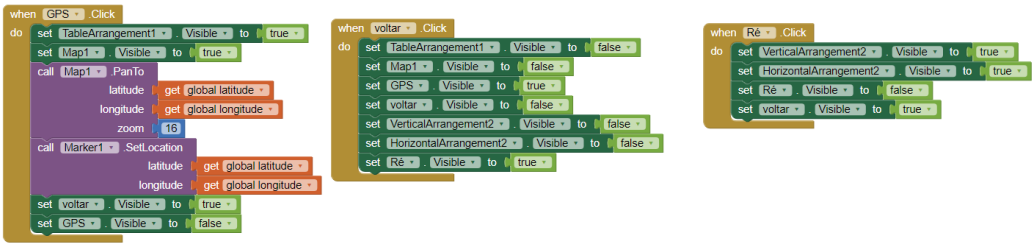


Figura 14: Configuração dos botões