Práctica 15

Uso de protocolo Bluetooth LE en la Raspberry Pi Pico W para enviar y recibir datos inalambricamentel

Juan Manuel Miranda Padrón 370924

Profesor: Ing. Jesús Padrón

Facultad de Estudios Profesionales Zona Media - UASLP

19 de Mayo de 2025

1. Introducción

La practica 15 consiste en entender los conceptos básicos del protocolo Bluetooth LE, aprender a utilizar el microcontrolador Raspberry Pi Pico W como servidor y cliente BLE, Aprender a utilizar el sensor de temperatura interno del Raspberry.
La elaboración de esta practica consiste en usar dos Raspberry Pi Pico W una como cliente y la otra como servidor BLE con las que a través de el monitor serial se podrá ver mediciones de temperatura a través de de un sensor de temperatura integrado en la Raspberry.

2. Desarrollo de la practica

La practica 15 cuenta con varios archivos de código para segmentar y organizar el proyecto, tendrá dos ejecutables .uf2, el que funciona como servidor necesita de los códigos: server.c, server common.c,server common.h y el archivo temp sensor.gatt, estos activaran el ADC de la placa y actuara como servidor enviando información a una segunda Raspberry que actuara como cliente, pero, esta necesitara del código client.c para poder generar el ejecutable .uf2, leyendo la información e imprimiendola en el monitor serial

2.1. CMakeLists.txt

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.12)

# Pull in SDK (must be before project)
include(pico_sdk_import.cmake)

if (PICO_SDK_VERSION_STRING VERSION_LESS "1.5.0")
    message(FATAL_ERROR "Raspberry Pi Pico SDK version
    1.5.0 (or later) required. Your version is ${
    PICO_SDK_VERSION_STRING}")
endif()

project(Practica_15)
set(PICO_BOARD pico_w)
set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 17)

# Initialize the SDK
pico_sdk_init()

# Standalone example that reads from the on board
# temperature sensor and sends notifications via BLE
# Flashes slowly each second to show  i t s  running
add_executable(picow_ble_temp_sensor
    server.c server_common.c
```

```
)

target_link_libraries(picow_ble_temp_sensor
    pico_stdlib
    pico_btstack_ble
    pico_btstack_cyw43
    pico_cyw43_arch_none
    hardware_adc
)

target_include_directories(picow_ble_temp_sensor PRIVATE
    ${CMAKE_CURRENT_LIST_DIR} # For btstack config
)

pico_btstack_make_gatt_header(picow_ble_temp_sensor
    PRIVATE "${CMAKE_CURRENT_LIST_DIR}/temp_sensor.gatt"
)

pico_add_extra_outputs(picow_ble_temp_sensor)

# Flashes twice quickly each second when connected to
# another device and reading it's␣temperature
add_executable(picow_ble_temp_reader
    client.c
)

target_link_libraries(picow_ble_temp_reader
    pico_stdlib
    pico_btstack_ble
    pico_btstack_cyw43
    pico_cyw43_arch_none
    hardware_adc
)

pico_enable_stdio_usb(picow_ble_temp_reader 1)
pico_enable_stdio_uart(picow_ble_temp_reader 0)

target_include_directories(picow_ble_temp_reader PRIVATE
    ${CMAKE_CURRENT_LIST_DIR} # For btstack config
)

target_compile_definitions(picow_ble_temp_reader PRIVATE
    RUNNING_AS_CLIENT=1
)

pico_add_extra_outputs(picow_ble_temp_reader)
```

2.2. Archivo para el servidor

Este archivo generara el servidor BLE y sera del protocolo del tipo GAT (General Attribute Profile) ademas de que definirá los servicios, características y permisos del servidor.

temp sensor.gatt

```
PRIMARY_SERVICE, GAP_SERVICE
CHARACTERISTIC, GAP_DEVICE_NAME, READ, "picow_temp"

PRIMARY_SERVICE, GATT_SERVICE
CHARACTERISTIC, GATT_DATABASE_HASH, READ,

PRIMARY_SERVICE, ORG_BLUETOOTH_SERVICE_ENVIRONMENTAL_SENSING
CHARACTERISTIC, ORG_BLUETOOTH_CHARACTERISTIC_TEMPERATURE,
    READ | NOTIFY | INDICATE | DYNAMIC,
```

Ahora con el anterior código mostrado necesitamos modificar el servidor para que muestre las mediciones de temperatura y para eso se usa el archivo server common.c y que sera complementado con server common.h.

server common.c

```
/**
 * Copyright (c) 2023 Raspberry Pi ( Trading ) Ltd .
 *
 * SPDX - License - Identifier : BSD -3- Clause
 */
#include <stdio.h>
#include "btstack.h"
#include "hardware/adc.h"

#include "temp_sensor.h"
#include "server_common.h"

#define APP_AD_FLAGS 0x06
static uint8_t adv_data[] = {
    // Flags general discoverable
    0x02, BLUETOOTH_DATA_TYPE_FLAGS, APP_AD_FLAGS,
    // Name
    0x17, BLUETOOTH_DATA_TYPE_COMPLETE_LOCAL_NAME, 'P', 'i',
        'c', 'o', '␣', '0', '0', ':', '0', '0', ':', '0', '0'
        , ':', '0', '0', ':', '0', '0', ':', '0', '0',
    0x03,
        BLUETOOTH_DATA_TYPE_COMPLETE_LIST_OF_16_BIT_SERVICE_CLASS_UUIDS
        , 0x1a, 0x18,
};
static const uint8_t adv_data_len = sizeof(adv_data);

int le_notification_enabled;
hci_con_handle_t con_handle;
uint16_t current_temp;

void packet_handler(uint8_t packet_type, uint16_t channel,
    uint8_t *packet, uint16_t size) {
    UNUSED(size);
    UNUSED(channel);
    bd_addr_t local_addr;
```

```
31      if (packet_type != HCI_EVENT_PACKET) return;
32
33      uint8_t event_type = hci_event_packet_get_type(packet);
34      switch(event_type){
35          case BTSTACK_EVENT_STATE:
36              if (btstack_event_state_get_state(packet) !=
                    HCI_STATE_WORKING) return;
37              gap_local_bd_addr(local_addr);
38              printf("BTstack␣up␣and␣running␣on␣%s.\n",
                    bd_addr_to_str(local_addr));
39
40              // setup advertisements
41              uint16_t adv_int_min = 800;
42              uint16_t adv_int_max = 800;
43              uint8_t adv_type = 0;
44              bd_addr_t null_addr;
45              memset(null_addr, 0, 6);
46              gap_advertisements_set_params(adv_int_min,
                    adv_int_max, adv_type, 0, null_addr, 0x07, 0
                    x00);
47              assert(adv_data_len <= 31); // ble limitation
48              gap_advertisements_set_data(adv_data_len, (
                    uint8_t*) adv_data);
49              gap_advertisements_enable(1);
50
51              poll_temp();
52
53              break;
54          case HCI_EVENT_DISCONNECTION_COMPLETE:
55              le_notification_enabled = 0;
56              break;
57          case ATT_EVENT_CAN_SEND_NOW:
58              att_server_notify(con_handle,
                    ATT_CHARACTERISTIC_ORG_BLUETOOTH_CHARACTERISTIC_TEMPERATURE_01_VALUE_HANDLE
                    , (uint8_t*)&current_temp, sizeof(
                    current_temp));
59              break;
60          default:
61              break;
62      }
63  }
64
65  uint16_t att_read_callback(hci_con_handle_t connection_handle
        , uint16_t att_handle, uint16_t offset, uint8_t * buffer,
         uint16_t buffer_size) {
66      UNUSED(connection_handle);
67
68      if (att_handle ==
            ATT_CHARACTERISTIC_ORG_BLUETOOTH_CHARACTERISTIC_TEMPERATURE_01_VALUE_HANDLE
            ){
69          return att_read_callback_handle_blob((const uint8_t
                *)&current_temp, sizeof(current_temp), offset,
                buffer, buffer_size);
70      }
71      return 0;
```

```
72 }
73
74 int att_write_callback(hci_con_handle_t connection_handle,
       uint16_t att_handle, uint16_t transaction_mode, uint16_t
       offset, uint8_t *buffer, uint16_t buffer_size) {
75     UNUSED(transaction_mode);
76     UNUSED(offset);
77     UNUSED(buffer_size);
78
79     if (att_handle !=
           ATT_CHARACTERISTIC_ORG_BLUETOOTH_CHARACTERISTIC_TEMPERATURE_01_CLIENT_CONFIGURATION
           ) return 0;
80     le_notification_enabled = little_endian_read_16(buffer,
           0) ==
           GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_NOTIFICATION
           ;
81     con_handle = connection_handle;
82     if (le_notification_enabled) {
83         att_server_request_can_send_now_event(con_handle);
84     }
85     return 0;
86 }
87
88 void poll_temp(void) {
89     adc_select_input(ADC_CHANNEL_TEMPSENSOR);
90     uint32_t raw32 = adc_read();
91     const uint32_t bits = 12;
92
93     // Scale raw reading to 16 bit value using a Taylor
           expansion (for 8 <= bits <= 16)
94     uint16_t raw16 = raw32 << (16 - bits) | raw32 >> (2 *
           bits - 16);
95
96     // ref https://github.com/raspberrypi/pico-micropython-
           examples/blob/master/adc/temperature.py
97     const float conversion_factor = 3.3 / (65535);
98     float reading = raw16 * conversion_factor;
99
100    // The temperature sensor measures the Vbe voltage of a
           biased bipolar diode, connected to the fifth ADC
           channel
101    // Typically, Vbe = 0.706V at 27 degrees C, with a slope
           of -1.721mV (0.001721) per degree.
102    float deg_c = 27 - (reading - 0.706) / 0.001721;
103    current_temp = deg_c * 100;
104    printf("Write␣temp␣%.2f␣degc\n", deg_c);
105 }
```

server common.h

```
1 /**
2  * Copyright (c) 2023 Raspberry Pi (Trading) Ltd.
3  *
```

```
4  * SPDX-License-Identifier: BSD-3-Clause
5  */
6
7 #ifndef SERVER_COMMON_H_
8 #define SERVER_COMMON_H_
9
10 #define ADC_CHANNEL_TEMPSENSOR 4
11
12 extern int le_notification_enabled;
13 extern hci_con_handle_t con_handle;
14 extern uint16_t current_temp;
15 extern uint8_t const profile_data[];
16
17 void packet_handler(uint8_t packet_type, uint16_t channel,
       uint8_t *packet, uint16_t size);
18 uint16_t att_read_callback(hci_con_handle_t connection_handle
       , uint16_t att_handle, uint16_t offset, uint8_t * buffer,
        uint16_t buffer_size);
19 int att_write_callback(hci_con_handle_t connection_handle,
       uint16_t att_handle, uint16_t transaction_mode, uint16_t
       offset, uint8_t *buffer, uint16_t buffer_size);
20 void poll_temp(void);
21
22 #endif
```

Con el código server.c se iniciara el modulo Bluetooth de la Raspberry y se configuraran sus parámetros.

server,c

```
1 /**
2  * Copyright (c) 2023 Raspberry Pi (Trading) Ltd.
3  *
4  * SPDX-License-Identifier: BSD-3-Clause
5  */
6
7 #include <stdio.h>
8 #include "btstack.h"
9 #include "pico/cyw43_arch.h"
10 #include "pico/btstack_cyw43.h"
11 #include "hardware/adc.h"
12 #include "pico/stdlib.h"
13
14 #include "server_common.h"
15
16 #define HEARTBEAT_PERIOD_MS 100
17
18 static btstack_timer_source_t heartbeat;
19 static btstack_packet_callback_registration_t
       hci_event_callback_registration;
20
21 static void heartbeat_handler(struct btstack_timer_source *ts
       ) {
```

```
22      static uint32_t counter = 0;
23      counter++;
24
25      // Update the temp every 10s
26      if (counter % 10 == 0) {
27          poll_temp();
28          if (le_notification_enabled) {
29              att_server_request_can_send_now_event(con_handle)
                    ;
30          }
31      }
32
33      // Invert the led
34      static int led_on = true;
35      led_on = !led_on;
36      cyw43_arch_gpio_put(CYW43_WL_GPIO_LED_PIN, led_on);
37
38      // Restart timer
39      btstack_run_loop_set_timer(ts, HEARTBEAT_PERIOD_MS);
40      btstack_run_loop_add_timer(ts);
41  }
42
43  int main() {
44      stdio_init_all();
45
46      // initialize CYW43 driver architecture (will enable BT
            if/because CYW43_ENABLE_BLUETOOTH == 1)
47      if (cyw43_arch_init()) {
48          printf("failed to initialise cyw43_arch\n");
49          return -1;
50      }
51
52      // Initialise adc for the temp sensor
53      adc_init();
54      adc_select_input(ADC_CHANNEL_TEMPSENSOR);
55      adc_set_temp_sensor_enabled(true);
56
57      l2cap_init();
58      sm_init();
59
60      att_server_init(profile_data, att_read_callback,
            att_write_callback);
61
62      // inform about BTstack state
63      hci_event_callback_registration.callback = &
            packet_handler;
64      hci_add_event_handler(&hci_event_callback_registration);
65
66      // register for ATT event
67      att_server_register_packet_handler(packet_handler);
68
69      // set one-shot btstack timer
70      heartbeat.process = &heartbeat_handler;
71      btstack_run_loop_set_timer(&heartbeat,
            HEARTBEAT_PERIOD_MS);
```

```
72      btstack_run_loop_add_timer(&heartbeat);
73
74      // turn on bluetooth!
75      hci_power_control(HCI_POWER_ON);
76
77      // btstack_run_loop_execute is only required when using
            the 'polling' method (e.g. using pico_cyw43_arch_poll
             library).
78      // This example uses the 'threadsafe background' method,
            where BT work is handled in a low priority IRQ, so it
79      // is fine to call bt_stack_run_loop_execute() but
            equally you can continue executing user code.
80
81  #if 0 // btstack_run_loop_execute() is not required, so lets
        not use it
82      btstack_run_loop_execute();
83  #else
84      // this core is free to do it's own stuff except when
            using 'polling' method (in which case you should use
85      // btstacK_run_loop_ methods to add work to the run loop.
86
87      // this is a forever loop in place of where user code
            would go.
88      while(true) {
89          sleep_ms(100);
90      }
91  #endif
92      return 0;
93  }
```

2.3. Archivos para el cliente BLE

El archivo ejecutable para el cliente necesita del archivo client.c que configura los parámetros Bluetooth necesarios para conectarse con el servidor de la primera Raspberry.
El funcionamiento del código es de la siguiente manera.

- Inicializacion: Configura el sistema, el stack BLE e inicia el escaneo del dispositivo BLE.
- Escaneo BLE: Detecta dispositivos cercanos con servicio BLE y cuando los encuentra guarda su dirección y se conecta.
- Conexión y descubrimiento GATT: Al conectarse busca un servicio de sensores ambientales, busca características de temperatura dentro de este y activa las notificaciones para recibir datos.
- Recepción de datos: Cuando el dispositivo envía los datos de temperatura estos se convierten a grados Celsius.
- Desconexion: Si se pierde la conexión se reinicia el proceso de escaneo y conexión.

- LED indicativo: Parpadeo rápidamente si se esta recibiendo datos y parpadeara lentamente si no.

client.c

```c
/**
 * Copyright (c) 2023 Raspberry Pi (Trading) Ltd.
 *
 * SPDX-License-Identifier: BSD-3-Clause
 */

#include <stdio.h>
#include "btstack.h"
#include "pico/cyw43_arch.h"
#include "pico/stdlib.h"

#if 0
#define DEBUG_LOG(...) printf(__VA_ARGS__)
#else
#define DEBUG_LOG(...)
#endif

#define LED_QUICK_FLASH_DELAY_MS 100
#define LED_SLOW_FLASH_DELAY_MS 1000

typedef enum {
    TC_OFF,
    TC_IDLE,
    TC_W4_SCAN_RESULT,
    TC_W4_CONNECT,
    TC_W4_SERVICE_RESULT,
    TC_W4_CHARACTERISTIC_RESULT,
    TC_W4_ENABLE_NOTIFICATIONS_COMPLETE,
    TC_W4_READY
} gc_state_t;

static btstack_packet_callback_registration_t
    hci_event_callback_registration;
static gc_state_t state = TC_OFF;
static bd_addr_t server_addr;
static bd_addr_type_t server_addr_type;
static hci_con_handle_t connection_handle;
static gatt_client_service_t server_service;
static gatt_client_characteristic_t server_characteristic;
static bool listener_registered;
static gatt_client_notification_t notification_listener;
static btstack_timer_source_t heartbeat;

static void client_start(void){
    DEBUG_LOG("Start␣scanning!\n");
    state = TC_W4_SCAN_RESULT;
    gap_set_scan_parameters(0,0x0030, 0x0030);
    gap_start_scan();
}

```

```
50 static bool advertisement_report_contains_service(uint16_t
       service, uint8_t *advertisement_report){
51     // get advertisement from report event
52     const uint8_t * adv_data =
           gap_event_advertising_report_get_data(
           advertisement_report);
53     uint8_t adv_len  =
           gap_event_advertising_report_get_data_length(
           advertisement_report);
54
55     // iterate over advertisement data
56     ad_context_t context;
57     for (ad_iterator_init(&context, adv_len, adv_data) ;
           ad_iterator_has_more(&context) ; ad_iterator_next(&
           context)){
58         uint8_t data_type = ad_iterator_get_data_type(&
               context);
59         uint8_t data_size = ad_iterator_get_data_len(&context
               );
60         const uint8_t * data = ad_iterator_get_data(&context)
               ;
61         switch (data_type){
62             case
                   BLUETOOTH_DATA_TYPE_COMPLETE_LIST_OF_16_BIT_SERVICE_CLASS_UUIDS
                   :
63                 for (int i = 0; i < data_size; i += 2) {
64                     uint16_t type = little_endian_read_16(
                           data, i);
65                     if (type == service) return true;
66                 }
67             default:
68                 break;
69         }
70     }
71     return false;
72 }
73
74 static void handle_gatt_client_event(uint8_t packet_type,
       uint16_t channel, uint8_t *packet, uint16_t size) {
75     UNUSED(packet_type);
76     UNUSED(channel);
77     UNUSED(size);
78
79     uint8_t att_status;
80     switch(state){
81         case TC_W4_SERVICE_RESULT:
82             switch(hci_event_packet_get_type(packet)) {
83                 case GATT_EVENT_SERVICE_QUERY_RESULT:
84                     // store service (we expect only one)
85                     DEBUG_LOG("Storing␣service\n");
86                     gatt_event_service_query_result_get_service
                           (packet, &server_service);
87                     break;
88                 case GATT_EVENT_QUERY_COMPLETE:
```

```
89                      att_status =
                            gatt_event_query_complete_get_att_status
                            (packet);
90                      if (att_status != ATT_ERROR_SUCCESS){
91                          printf("SERVICE_QUERY_RESULT, ATT
                                Error 0x%02x.\n", att_status);
92                          gap_disconnect(connection_handle);
93                          break;
94                      }
95                      // service query complete, look for
                            characteristic
96                      state = TC_W4_CHARACTERISTIC_RESULT;
97                      DEBUG_LOG("Search for env sensing
                            characteristic.\n");
98                      gatt_client_discover_characteristics_for_service_by_uuid16
                            (handle_gatt_client_event,
                            connection_handle, &server_service,
                            ORG_BLUETOOTH_CHARACTERISTIC_TEMPERATURE
                            );
99                      break;
100                 default:
101                     break;
102             }
103             break;
104         case TC_W4_CHARACTERISTIC_RESULT:
105             switch(hci_event_packet_get_type(packet)) {
106                 case GATT_EVENT_CHARACTERISTIC_QUERY_RESULT:
107                     DEBUG_LOG("Storing characteristic\n");
108                     gatt_event_characteristic_query_result_get_characteristic
                            (packet, &server_characteristic);
109                     break;
110                 case GATT_EVENT_QUERY_COMPLETE:
111                     att_status =
                            gatt_event_query_complete_get_att_status
                            (packet);
112                     if (att_status != ATT_ERROR_SUCCESS){
113                         printf("CHARACTERISTIC_QUERY_RESULT,
                                ATT Error 0x%02x.\n", att_status)
                                ;
114                         gap_disconnect(connection_handle);
115                         break;
116                     }
117                     // register handler for notifications
118                     listener_registered = true;
119                     gatt_client_listen_for_characteristic_value_updates
                            (&notification_listener,
                            handle_gatt_client_event,
                            connection_handle, &
                            server_characteristic);
120                     // enable notifications
121                     DEBUG_LOG("Enable notify on
                            characteristic.\n");
122                     state =
                            TC_W4_ENABLE_NOTIFICATIONS_COMPLETE;
```

```
123                     gatt_client_write_client_characteristic_configuration
                            (handle_gatt_client_event,
                            connection_handle,
124                         &server_characteristic,
                                GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_NOTIFICATION
                                );
125                     break;
126                 default:
127                     break;
128             }
129             break;
130         case TC_W4_ENABLE_NOTIFICATIONS_COMPLETE:
131             switch(hci_event_packet_get_type(packet)) {
132                 case GATT_EVENT_QUERY_COMPLETE:
133                     DEBUG_LOG("Notifications␣enabled,␣ATT␣
                            status␣0x%02x\n",
                            gatt_event_query_complete_get_att_status
                            (packet));
134                     if (
                            gatt_event_query_complete_get_att_status
                            (packet) != ATT_ERROR_SUCCESS) break;
135                     state = TC_W4_READY;
136                     break;
137                 default:
138                     break;
139             }
140             break;
141         case TC_W4_READY:
142             switch(hci_event_packet_get_type(packet)) {
143                 case GATT_EVENT_NOTIFICATION: {
144                     uint16_t value_length =
                            gatt_event_notification_get_value_length
                            (packet);
145                     const uint8_t *value =
                            gatt_event_notification_get_value(
                            packet);
146                     DEBUG_LOG("Indication␣value␣len␣%d\n",
                            value_length);
147                     if (value_length == 2) {
148                         float temp = little_endian_read_16(
                                value, 0);
149                         printf("read␣temp␣%.2f␣degc\n", temp
                                / 100);
150                     } else {
151                         printf("Unexpected␣length␣%d\n",
                                value_length);
152                     }
153                     break;
154                 }
155                 default:
156                     printf("Unknown␣packet␣type␣0x%02x\n",
                            hci_event_packet_get_type(packet));
157                     break;
158             }
159             break;
```

```
160         default:
161             printf("error\n");
162             break;
163     }
164 }
165
166 static void hci_event_handler(uint8_t packet_type, uint16_t
        channel, uint8_t *packet, uint16_t size) {
167     UNUSED(size);
168     UNUSED(channel);
169     bd_addr_t local_addr;
170     if (packet_type != HCI_EVENT_PACKET) return;
171
172     uint8_t event_type = hci_event_packet_get_type(packet);
173     switch(event_type){
174         case BTSTACK_EVENT_STATE:
175             if (btstack_event_state_get_state(packet) ==
                    HCI_STATE_WORKING) {
176                 gap_local_bd_addr(local_addr);
177                 printf("BTstack␣up␣and␣running␣on␣%s.\n",
                        bd_addr_to_str(local_addr));
178                 client_start();
179             } else {
180                 state = TC_OFF;
181             }
182             break;
183         case GAP_EVENT_ADVERTISING_REPORT:
184             if (state != TC_W4_SCAN_RESULT) return;
185             // check name in advertisement
186             if (!advertisement_report_contains_service(
                    ORG_BLUETOOTH_SERVICE_ENVIRONMENTAL_SENSING,
                    packet)) return;
187             // store address and type
188             gap_event_advertising_report_get_address(packet,
                    server_addr);
189             server_addr_type =
                    gap_event_advertising_report_get_address_type
                    (packet);
190             // stop scanning, and connect to the device
191             state = TC_W4_CONNECT;
192             gap_stop_scan();
193             printf("Connecting␣to␣device␣with␣addr␣%s.\n",
                    bd_addr_to_str(server_addr));
194             gap_connect(server_addr, server_addr_type);
195             break;
196         case HCI_EVENT_LE_META:
197             // wait for connection complete
198             switch (hci_event_le_meta_get_subevent_code(
                    packet)) {
199                 case HCI_SUBEVENT_LE_CONNECTION_COMPLETE:
200                     if (state != TC_W4_CONNECT) return;
201                     connection_handle =
                            hci_subevent_le_connection_complete_get_connection_handle
                            (packet);
```

```
202                     // initialize gatt client context with
                            handle, and add it to the list of
                            active clients
203                     // query primary services
204                     DEBUG_LOG("Search␣for␣env␣sensing␣service
                            .\n");
205                     state = TC_W4_SERVICE_RESULT;
206                     gatt_client_discover_primary_services_by_uuid16
                            (handle_gatt_client_event,
                            connection_handle,
                            ORG_BLUETOOTH_SERVICE_ENVIRONMENTAL_SENSING
                            );
207                     break;
208                 default:
209                     break;
210             }
211             break;
212         case HCI_EVENT_DISCONNECTION_COMPLETE:
213             // unregister listener
214             connection_handle = HCI_CON_HANDLE_INVALID;
215             if (listener_registered){
216                 listener_registered = false;
217                 gatt_client_stop_listening_for_characteristic_value_updates
                        (&notification_listener);
218             }
219             printf("Disconnected␣%s\n", bd_addr_to_str(
                    server_addr));
220             if (state == TC_OFF) break;
221             client_start();
222             break;
223         default:
224             break;
225     }
226 }
227
228 static void heartbeat_handler(struct btstack_timer_source *ts
        ) {
229     // Invert the led
230     static bool quick_flash;
231     static bool led_on = true;
232
233     led_on = !led_on;
234     cyw43_arch_gpio_put(CYW43_WL_GPIO_LED_PIN, led_on);
235     if (listener_registered && led_on) {
236         quick_flash = !quick_flash;
237     } else if (!listener_registered) {
238         quick_flash = false;
239     }
240
241     // Restart timer
242     btstack_run_loop_set_timer(ts, (led_on || quick_flash) ?
            LED_QUICK_FLASH_DELAY_MS : LED_SLOW_FLASH_DELAY_MS);
243     btstack_run_loop_add_timer(ts);
244 }
245
```

```
246 int main() {
247     stdio_init_all();
248 
249     // initialize CYW43 driver architecture (will enable BT
        if/because CYW43_ENABLE_BLUETOOTH == 1)
250     if (cyw43_arch_init()) {
251         printf("failed to initialise cyw43_arch\n");
252         return -1;
253     }
254 
255     l2cap_init();
256     sm_init();
257     sm_set_io_capabilities(IO_CAPABILITY_NO_INPUT_NO_OUTPUT);
258 
259     // setup empty ATT server - only needed if LE Peripheral
        does ATT queries on its own, e.g. Android and iOS
260     att_server_init(NULL, NULL, NULL);
261 
262     gatt_client_init();
263 
264     hci_event_callback_registration.callback = &
        hci_event_handler;
265     hci_add_event_handler(&hci_event_callback_registration);
266 
267     // set one-shot btstack timer
268     heartbeat.process = &heartbeat_handler;
269     btstack_run_loop_set_timer(&heartbeat,
        LED_SLOW_FLASH_DELAY_MS);
270     btstack_run_loop_add_timer(&heartbeat);
271 
272     // turn on!
273     hci_power_control(HCI_POWER_ON);
274 
275     // btstack_run_loop_execute is only required when using
        the 'polling' method (e.g. using pico_cyw43_arch_poll
         library).
276     // This example uses the 'threadsafe background' method,
        where BT work is handled in a low priority IRQ, so it
277     // is fine to call bt_stack_run_loop_execute() but
        equally you can continue executing user code.
278 
279 #if 1 // this is only necessary when using polling (which we
    aren't, but we're showing it is still safe to call in
    this case)
280     btstack_run_loop_execute();
281 #else
282     // this core is free to do it's own stuff except when
        using 'polling' method (in which case you should use
283     // btstacK_run_loop_ methods to add work to the run loop.
284 
285     // this is a forever loop in place of where user code
        would go.
286     while(true) {
287         sleep_ms(1000);
288     }
```

```
289 #endif
290     return 0;
291 }
```

2.4. Configuración de BTStack

btstack config.h

```
1  #ifndef _PICO_BTSTACK_BTSTACK_CONFIG_H
2  #define _PICO_BTSTACK_BTSTACK_CONFIG_H
3
4  #ifndef ENABLE_BLE
5  #error Please link to pico_btstack_ble
6  #endif
7
8  // BTstack features that can be enabled
9  #define ENABLE_LE_PERIPHERAL
10 #define ENABLE_LOG_INFO
11 #define ENABLE_LOG_ERROR
12 #define ENABLE_PRINTF_HEXDUMP
13
14 // for the client
15 #if RUNNING_AS_CLIENT
16 #define ENABLE_LE_CENTRAL
17 #define MAX_NR_GATT_CLIENTS 1
18 #else
19 #define MAX_NR_GATT_CLIENTS 0
20 #endif
21
22 // BTstack configuration. buffers, sizes, ...
23 #define HCI_OUTGOING_PRE_BUFFER_SIZE 4
24 #define HCI_ACL_PAYLOAD_SIZE (255 + 4)
25 #define HCI_ACL_CHUNK_SIZE_ALIGNMENT 4
26 #define MAX_NR_HCI_CONNECTIONS 1
27 #define MAX_NR_SM_LOOKUP_ENTRIES 3
28 #define MAX_NR_WHITELIST_ENTRIES 16
29 #define MAX_NR_LE_DEVICE_DB_ENTRIES 16
30
31 // Limit number of ACL/SCO Buffer to use by stack to avoid
       cyw43 shared bus overrun
32 #define MAX_NR_CONTROLLER_ACL_BUFFERS 3
33 #define MAX_NR_CONTROLLER_SCO_PACKETS 3
34
35 // Enable and configure HCI Controller to Host Flow Control
       to avoid cyw43 shared bus overrun
36 #define ENABLE_HCI_CONTROLLER_TO_HOST_FLOW_CONTROL
37 #define HCI_HOST_ACL_PACKET_LEN (255+4)
38 #define HCI_HOST_ACL_PACKET_NUM 3
39 #define HCI_HOST_SCO_PACKET_LEN 120
40 #define HCI_HOST_SCO_PACKET_NUM 3
41
42 // Link Key DB and LE Device DB using TLV on top of Flash
```

```
      Sector interface
#define NVM_NUM_DEVICE_DB_ENTRIES 16
#define NVM_NUM_LINK_KEYS 16

// We don't give btstack a malloc, so use a fixed-size ATT DB
      .
#define MAX_ATT_DB_SIZE 512

// BTstack HAL configuration
#define HAVE_EMBEDDED_TIME_MS
// map btstack_assert onto Pico SDK assert()
#define HAVE_ASSERT
// Some USB dongles take longer to respond to HCI reset (e.g.
       BCM20702A).
#define HCI_RESET_RESEND_TIMEOUT_MS 1000
#define ENABLE_SOFTWARE_AES128
#define ENABLE_MICRO_ECC_FOR_LE_SECURE_CONNECTIONS

#endif // MICROPY_INCLUDED_EXTMOD_BTSTACK_BTSTACK_CONFIG_H
```

2.5. Compilación de los archivos

El CMakeListscreara la carpeta build que compilara los archivos .uf2 necesarios para el servidor y el cliente y si no se encuentran todos los archivos necesarios no se compilara correctamente.

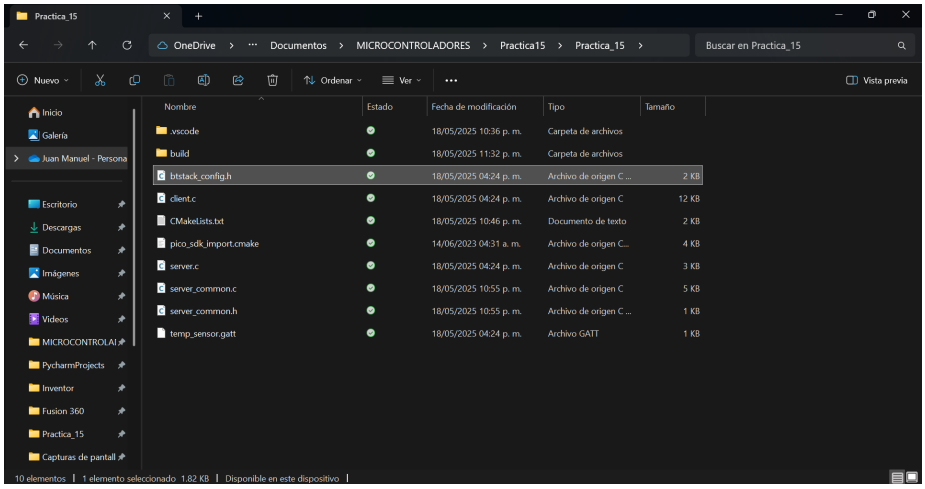

Figura 1: Ejemplo de como debería verse la carpeta de la practica (sin incluir la carpeta build)

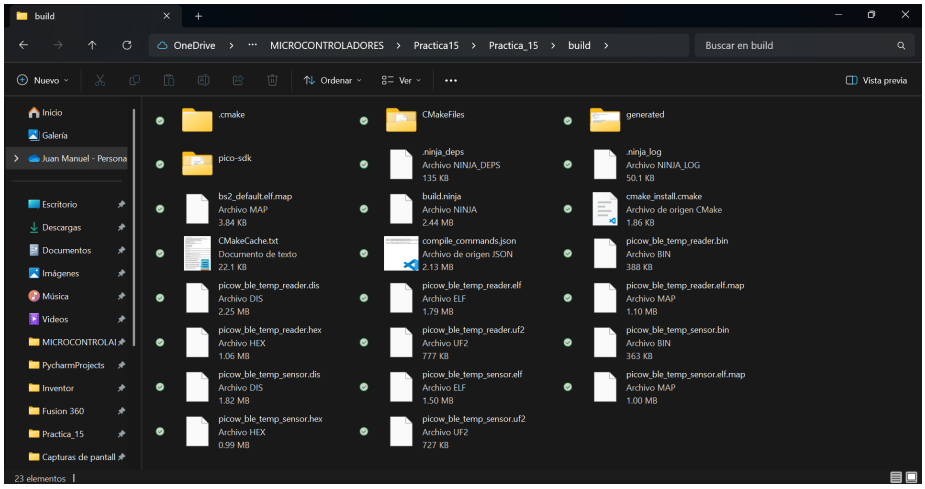

Figura 2: Carpeta build

2.6. Montaje y prueba

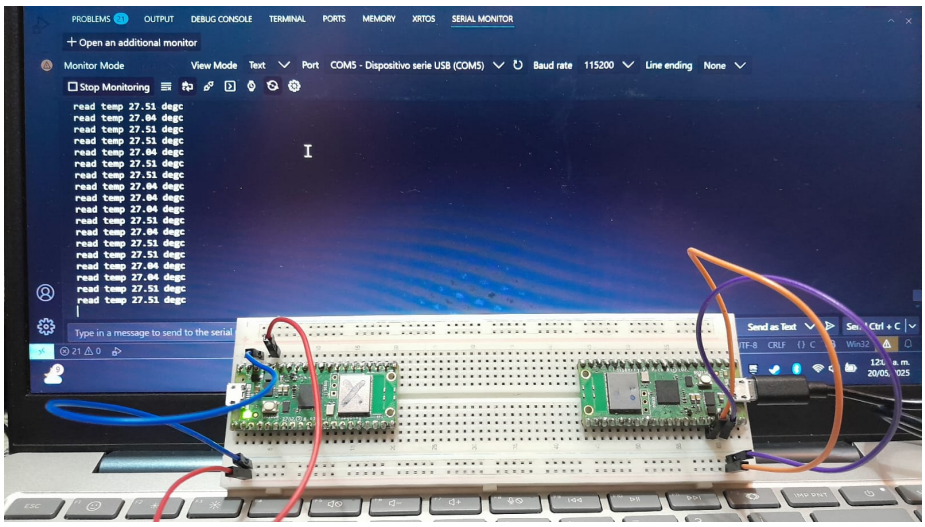

Figura 3: Montaje de la practica 15 con las mediciones del monitor serial

2.7. Preguntas de repaso

1. ¿Cual es la principal diferencia entre Bluetooth clásico y Bluetooth LE? EL Bluetooth clásico esta diseñado para conexiones de alta velocidad y

mayor consumo de energía, mientras que el Bluetooth LE esta diseñado para transmisiones periódicas de bajo consumo.

2. ¿Que ventajas ofrece Bluetooth LE en comparación con otras tecnologías inalámbricas?
   - Bajo consumo de energía
   - Una conexión rápida
   - Bajos costos
   - Alta escalabilidad
   - Una buena compatibilidad
3. ¿Cual es el propósito del archivo btstack config.h en esta practica? El archivo btstack config.h permite configurar y personalizar el comportamiento del stack de Bluetooth (BTstack) antes de compilar. Sirve para optimizar el funcionamiento y reducir el uso de recursos, incluyendo solo lo necesario para la práctica.
4. ¿Que ventajas crees que tiene usar comunicación BLE en un sistema de monitoreo frente a WiFi o comunicación por cable?
   - Un menor consumo de energía
   - Una Mayor movilidad y flexibilidad
   - Es mas simple de configurar
   - Una conexión directa con smartphones
   - Una menor latencia
5. ¿Por que crees que es importante que el código del cliente BLE este diseñado para reconectarse automáticamente al servidor en caso de desconexion? Por estabilidad y confiabilidad en el sistema, tolerancia a fallos, por continuidad de monitoreo o control y una mejor experiencia de usuario.
6. En un ambiente con muchos dispositivos BLE, ¿que estrategias se podrían usar para evitar interferencias o confusiones al conectar dispositivos?
   - Uso de nombres únicos o UUIDs personalizados para cada dispositivo
   - Filtrado de dispositivos por dirección MAC o servicios GATT
   - Conexión basada en proximidad (RSSI alto)
   - Publicación de anuncios en intervalos personalizados para reducir colisiones
   - Implementar "whitelisting"(lista de dispositivos permitidos)
   - Agrupar dispositivos en canales o zonas para minimizar interferencia

7. ¿Como modificarías esta practica para mostrar los datos recolectados en una aplicación móvil usando Bluetooth LE? Implementar un servicio GATT personalizado en el servidor BLE (ej. Raspberry Pi Pico o similar), crear una característica GATT para transmitir los datos al cliente (app móvil).
   En la app móvil:

   - Escanear dispositivos BLE y conectar al correcto
   - Leer o suscribirse a la característica para recibir datos (modo notification)
   - Mostrar los datos en tiempo real en la interfaz (ej. gráfico o texto)

   Usar una plataforma como MIT App Inventor, Flutter, React Native o Android Studio con acceso a Bluetooth LE.

3. Conclusion

Esta práctica me brindó una comprensión más clara y práctica del protocolo Bluetooth Low Energy (BLE) y su aplicación en sistemas embebidos mediante la Raspberry Pi Pico W. Mediante la creación de un cliente y un servidor BLE, conseguí establecer una comunicación inalámbrica eficiente y económica para la transmisión de datos de temperatura. La implementación de BTstack fue esencial para simplificar esta implementación, posibilitando la configuración organizada de perfiles y servicios GATT.
Esta práctica, además de fortalecer mis habilidades en comunicación inalámbrica y configuración de microcontroladores, me evidenció la relevancia de conceptos como la eficiencia energética, la reconexión automática y la modularidad del código. Observar el sistema en funcionamiento, con la información actualizada en el monitor serie, resultó ser una experiencia estimulante y de gran valor.

3.1. Posibles mejoras

1. Mostrar los datos en una app móvil usando Bluetooth LE para una visualización más accesible.

2. Agregar alertas visuales o sonoras (LEDs o buzzer) si la temperatura supera ciertos límites.

3. Guardar el historial de temperaturas en una memoria externa para análisis posterior.

4. Mejorar el manejo de errores ante desconexiones o fallos en la lectura del sensor.

5. Incluir más sensores (como humedad o luz) y enviar sus datos también vía BLE.