Práctica 15: Uso de protocolo Bluetooth LE en la Raspberry Pi Pico W para enviar y recibir datos inal´ambricamente

Microcontroladores

Alumno: Itzel Estrada Arcos (368980)
Docente: Ing. Jesus Padron

Lunes 19 de mayo 2025

Índice general

Índice de figuras

0.1. Introduction.

0.1.1. Objetivos de aprendizaje.

- Entender los conceptos básicos del protocolo Bluetooth LE.
- Aprender a utilizar el microcontrolador como servidor y cliente BLE.
- Aprender a utilizar el sensor de temperatura interno de la RPi Pico W.

  En esta práctica desarrollamos una aplicación basada en tecnología Bluetooth Low Energy (BLE) utilizando dos Raspberry Pi Pico. El objetivo principal fue implementar un método para calcular la temperatura interna de un microcontrolador utilizando la arquitectura cliente-servidor BLE para permitirnos la comunicación inalámbrica. Para lograr esto, integramos la pila de protocolos Bluetooth con BTstack para definir la estructura del proyecto en varios archivos, estableciendo mecanismos de comunicación y control entre los dispositivos involucrados. Está práctica no solo nos muestra las capacidades BLE de Raspberry Pi Pico, sino que también muestra cómo se puede utilizar en una variedad de aplicaciones de software y hardware.

0.2. Desarrollo de la practica.

Este proyecto describe el desarrollo de una aplicación Bluetooth Low Energy utilizando la Raspberry Pi Pico, centrada en la lectura y transmisión de temperatura mediante un servidor y un cliente BLE. En el archivo btstack-config.h, configuramos la pila Bluetooth definiendo funcionalidades necesarias como el soporte para actuar como periférico BLE, control de flujo, parámetros de comunicación, uso eficiente de memoria y almacenamiento no volátil. Ajustamos también parámetros para asegurar una buena interacción con el hardware y garantizar la estabilidad del sistema. Aqui, un breve resumen de los archivos:

En client.c, Implementamos un cliente BLE que escanea dispositivos cercanos, identificamos servidores con servicios específicos, y establecemos una conexión para recibir datos. Manejamos eventos tanto a nivel de conexión como de servicios GATT, controlamos un LED según el estado del cliente y definimos funciones para el escaneo, conexión y comunicación. El cliente opera en segundo plano gracias al bucle principal de BTstack.

Por su parte, server-common.c configuramos el servidor BLE que emite la temperatura del sensor interno de la Raspberry Pi Pico. Gestionamos eventos de conexión y notificación para que responda a solicitudes de lectura de características por parte del cliente y administramos la conversión de lecturas del ADC a valores de temperatura. Implementamos una rutina periódica para tomar lecturas, notificar al cliente y alternar un LED.

En archivo server-common.h declaramos constantes, variables externas y prototipos de funciones esenciales para la operación del servidor, incluyendo el control de notificaciones, lectura de temperatura y manejo de paquetes Bluetooth.

Finalmente, en el archivo CMakeLists.txt estructuramos la construcción del proyecto. Definimos requisitos de versión, importamos el SDK de la Raspberry Pi Pico, configuramos tanto el servidor como el cliente BLE y generamos los archivos ejecutables necesarios. También gestionamos los perfiles GATT y activamos la salida por USB para facilitar el monitoreo del cliente. Todo está diseñado para asegurar que tanto el servidor como el cliente funcionen correctamente dentro del entorno BLE.

0.2.1. Proceso paso a paso.

Primero que nada, abrimos otra carpeta con la numeracion respectiva de la practica, para despues declarar todos los archivos .c, .h y CMakeList que necesitaremos como se muestra en la imagen.

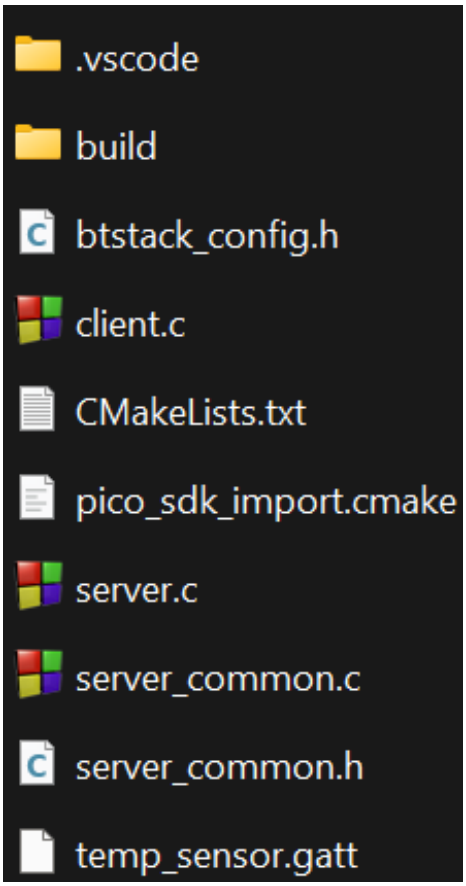

Figura 1: Archivos

btstack-config.h

En primer lugar, establecemos una protección contra inclusiones múltiples para evitar que el archivo se cargue más de una vez durante la compilación. Luego, verificamos que se esté usando correctamente el soporte para Bluetooth Low Energy (BLE), ya que es un requisito esencial para la configuración.

Además, definimos las funcionalidades del stack de Bluetooth que se van a activar, como el soporte para actuar como periférico BLE y la inclusión de registros de información, errores y volcados en formato hexadecimal. Luego, dependiendo de si el dispositivo funcionará como cliente central BLE, habilitamos esa capacidad y asignamos un número máximo de clientes GATT. Si no es cliente, lo limitamos ese número a cero para ahorrar recursos.

Además, definimos los parámetros operativos de la pila, como el tamaño del

buffer de salida de HCI, el alcance de ACL, la cola de ACL, el número máximo de conexiones HCI y las entradas para la base de datos de dispositivos y la lista de pares. La cantidad de buffers ACL y SCO utilizados también está limitada para evitar sobrecargar el bus compartido con chips Wi-Fi, como el CYW43.

Luego, el controlador HCI habilita y configura el control de flujo en el host para administrar mejor el tráfico de datos al configurar el tamaño y la velocidad máximos de los paquetes ACL y SCO.

Posteriormente, la base de datos del dispositivo BLE y el almacenamiento del encabezado los configuramos utilizando una técnica de almacenamiento no volátil basada en TLV en sectores de memoria Flash. También establecemos el tamaño de la base de datos ATT, ya que no se utiliza la asignación de memoria dinámica.

Después, especificamos configuraciones relacionadas con la capa de abstracción de hardware del stack, incluyendo el uso de una función de tiempo embebida, el mapeo de afirmaciones internas del stack hacia el sistema de aserciones del SDK de la Raspberry Pi Pico, un tiempo de espera extendido para el restablecimiento del controlador HCI útil con algunos dongles USB lentos, y el uso de implementaciones de cifrado por software y una biblioteca criptográfica ligera para conexiones seguras.

Finalmente, cerramos la protección del archivo, lo que indica que la configuración está completa y garantiza que todas las configuraciones especificadas se limitan a este contexto.

Client.c

En primer lugar, incluimos las bibliotecas necesarias para gestionar el sistema Bluetooth, la arquitectura del chip CYW43 y las herramientas básicas del entorno. Luego, configuramos una macro para mostrar mensajes de depuración, aunque la macro esté deshabilitada de forma predeterminada.

Además, definimos varios valores constantes para controlar la velocidad de parpadeo de los LED y realizamos un conteo para identificar los diferentes estados del cliente Bluetooth durante el proceso de conexión y comunicación con el servidor.

Luego, declaramos algunas variables globales que se utilizan para registrar eventos de la pila, almacenar las direcciones y servicios de dispositivos externos, configurar la conexión Bluetooth y configurar el reloj que controla los LED.

Luego, creamos una función que comienza a escanear el dispositivo BLE. Esta función establece los parámetros para el escaneo y luego comienza a buscar dispositivos cercanos.

También incluimos funciones que analizan los datos publicitarios de los dispositivos detectados para verificar si ofrecen algún servicio. Esto se logra leyendo y escaneando el contenido del anuncio en busca de señales del servicio esperado.

Luego definimos un controlador de eventos GATT. Esta función controla la respuesta del usuario al buscar servicios, recursos disponibles o al acceder a información. En cada caso, tenemos en cuenta el estado actual del usuario para determinar qué información se recibirá y cómo se responderá. Si se detecta un error, se finaliza la conexión; Si todo se ve bien, pase al siguiente paso, como el reclutamiento de audiencia o los anuncios.

Luego, instalamos un controlador de eventos HCI, que responde a los eventos prioritarios del sistema Bluetooth. Cuando el dispositivo está listo, se muestra su posición y comienza el escaneo. Si se detecta una coincidencia, se guarda la ubicación del dispositivo, se detiene el monitoreo y se inicia la conexión. Una vez finalizada la reunión, busca la tarea adecuada. Si hay un error, se inicia nuevamente el escaneo para encontrar el siguiente disponible.

También implementamos una función de temporizador. Esto cambia el estado del LED en cada llamada, ajustando su brillo según si el sensor está encendido o apagado. Luego, restablecemos el temporizador para que funcione de manera constante.

Finalmente describimos las principales funciones del sistema. En primer lugar instalamos el transmisor suministrado, seguido de la firma CYW43, con Bluetooth habilitado. Instalamos los componentes Bluetooth necesarios, el cliente GATT y el servidor ATT lo configuramos fácilmente paso a paso. Registramos un jefe de proyecto y fijamos un tiempo. Finalmente, habilitamos el controlador Bluetooth y habilitamos el bucle de ejecución BTstack, lo que nos permite que el sistema Bluetooth se ejecute en segundo plano mientras el núcleo principal está disponible para otras tareas.

Server-common.c

Primero incluimos algunas bibliotecas básicas. Por ejemplo, hay uno para controlar el Bluetooth, uno para operar el ADC y otros que son personalizados, como sensores de temperatura y configuraciones del servidor.

Luego definimos un mensaje Bluetooth constante para las banderas, junto con los mensajes enviados durante la transmisión. Estos mensajes incluyen una bandera que indica que se puede localizar el dispositivo, el nombre completo del dispositivo, representado por “Pico 00:00:00:00:00” como un tipo de dirección y el indicador de temperatura.

Si las comunicaciones BLE están habilitadas, se almacena una variable: una para almacenar la referencia a la conexión actual y otra para almacenar la tem-

peratura más reciente.

Además, introducimos una aplicación que gestiona el funcionamiento de la batería Bluetooth. Esta función escucha varios eventos del sistema Bluetooth. Cuando se recibe el evento de tipo "Estado", se considera que la batería está en uso. En este caso, obtenemos la dirección Bluetooth local del dispositivo y se muestra en la consola. Además, configuramos los parámetros del mensaje, incluido el medio y el tipo de mensaje, y habilite la funcionalidad del mensaje para que otros dispositivos puedan detectarlo. Al mismo tiempo se toma la primera lectura de temperatura.

Si se produce un retraso, las notificaciones se marcan como ya no habilitadas. Por otro lado, si el servidor ATT señala que se pueden enviar datos, el cliente conectado es informado de la temperatura actual y envía el valor almacenado.

Luego declaramos una función que es responsable de responder a la lectura del cliente de las características del servicio GATT. Si el cliente intenta leer el valor de la temperatura, se devuelve el valor actual.

Otra función maneja las entradas generadas por el cliente en la función de configuración. Esto sucede cuando, por ejemplo, un cliente desea habilitar o deshabilitar las notificaciones. Si el cliente permite notificaciones, se almacena su identificador de conexión y se solicita al servidor que le notifique cuando se pueden enviar datos.

Finalmente, definimos una función encargada de obtener la temperatura. Para hacerlo, seleccionamos el canal del sensor interno del ADC para que realice una lectura. El valor obtenido se convierte de una lectura a voltaje, que luego se convierte en grados Celsius usando una fórmula basada en las características del sensor del chip. Este valor se escala y se guarda en la variable de temperatura, además de imprimirse en consola.

Server-common.h

Este archivo es un encabezado C que nos sirve como punto de referencia para las funciones y variables relacionadas con el servidor de temperatura Bluetooth. Comienzamos con una condición de preprocesamiento que evita que el contenido se incluya varias veces durante la compilación, para evitar errores resultantes de definiciones duplicadas.

Definimos una constante que representa el número de canales ADC donde está conectado el sensor de temperatura interno del microcontrolador. Esta constante le permite estar siempre en el canal correcto sin tener que usar el número directamente.

Luego declaramos las variables externas. Contiene una variable que indica si las

notificaciones BLE están habilitadas, otra variable que almacena el identificador de conexión del cliente Bluetooth, otra variable que contiene la temperatura actual como valores y una matriz de datos que representa el perfil GATT necesario para que el servidor funcione.

Además, declaramos los encabezados de algunas funciones. El primero maneja los paquetes recibidos de la pila Bluetooth, administrando eventos como conexión, desconexión y disponibilidad para enviar notificaciones. Otra característica permite a los clientes leer valores característicos de temperatura a través del protocolo GATT. También hay una función que controla las escrituras que el cliente realiza en el descriptor de configuración, habilitando o deshabilitando las notificaciones. Finalmente, declaramos una función que se encarga de leer la temperatura actual del sensor interno, procesarla y almacenarla para su uso posterior.

Server.c

Comenzamos cargando las bibliotecas requeridas. Esto incluye soporte de entrada/salida estándar, funciones específicas de la pila Bluetooth (BTstack), control de chip inalámbrico CYW43, administración de ADC y otras funciones centrales del entorno de desarrollo Raspberry Pi Pico. También incluimos un archivo específico del proyecto que contiene definiciones comunes relacionadas con el servidor Bluetooth.

Definimos un intervalo de tiempo fijo en milisegundos que se utilizará para tareas repetitivas llamadas latidos. Esta tarea está programada con el temporizador del sistema BTstack.

Luego declaramos dos variables estáticas: una para el tiempo de latido y otra para registrar la función de manejo de eventos del sistema Bluetooth.

Definimos la función encargada del heartbeat. Esta función se ejecuta cada cierto tiempo y realiza tres tareas principales. Primero, lleva un conteo del número de veces que ha sido llamada, y cada diez veces ejecuta una lectura de temperatura. Si las notificaciones Bluetooth están activadas, solicita que se envíe una notificación al cliente. Después, alterna el estado del LED del dispositivo encendiéndolo y apagándolo de forma intermitente. Por último, vuelve a programar el temporizador para que se ejecute de nuevo después del intervalo definido. La función principal del programa inicializa primero la salida estándar para permitir impresión por consola. Luego intenta inicializar la arquitectura del chip inalámbrico; si falla, se imprime un error y se termina el programa.

Inicializamos el ADC del microcontrolador y configuramos para usar el sensor de temperatura interno. Después damos varios componentes del stack Bluetooth: L2CAP para la comunicación base, SM para el manejo de seguridad, y el servidor ATT con el perfil de servicio que define cómo se accede a la temperatura.

Registramos el manejador de eventos para el sistema Bluetooth, que se usa para controlar conexiones, desconexiones y otros eventos. También registramos la función que manejará los paquetes del servidor ATT. Luego configuramos el temporizador para que ejecute la función heartbeat de forma periódica, iniciando la primera vez con el intervalo definido.

Después se enciende el sistema Bluetooth, lo que pone en marcha toda la funcionalidad inalámbrica. Finalmente, establecemos una condición para ejecutar o no el bucle principal de BTstack. En este caso, no se ejecuta, ya que el ejemplo está usando una arquitectura que maneja Bluetooth en segundo plano. En su lugar, se ejecuta un bucle infinito vacío con pausas de 100 milisegundos, donde el usuario podría añadir otras tareas si lo desea.

CMakeList.txt

Comienzamos describiendo las versiones de CMake necesarias para construir el proyecto, garantizando la compatibilidad con el sistema operativo. Instalamos una página que permite importar el SDK de Raspberry Pi Pico, que es necesario antes de definir el proyecto. Luego verificamos si el SDK es 1.5.0. En caso contrario, se mostrará un mensaje de error y se cancelará la compilación. Esto nos garantiza que los componentes SDK necesarios estén disponibles para que su proyecto funcione sin problemas.

Se lanza el Pico SDK, necesario para habilitar todas las funciones, como control de PIN, Bluetooth y más. Diseñamos la primera parte del sistema, que corresponde al sensor de temperatura Bluetooth. La interfaz consta de dos controles interactivos que afectan el comportamiento del usuario.

También, indicamos dónde buscar los archivos de encabezado necesarios para la configuración de BTstack, en este caso, el directorio actual. Se genera automáticamente un archivo de encabezado a partir de un archivo `.gatt` que define el perfil de servicios GATT, usado para la comunicación BLE del sensor. Esto nos permite que el código fuente acceda fácilmente a las definiciones de características y servicios.

Después añadimos una instrucción para que el ejecutable tenga salidas adicionales si es necesario, como archivos de mapa, UF2 o binarios. Más adelante, configuramos un segundo ejecutable, que corresponde al lector de temperatura Bluetooth. Este programa actuará como cliente que se conecta al servidor para recibir datos de temperatura. Solo utiliza un archivo fuente.

Este segundo ejecutable también se enlaza con las mismas bibliotecas que el primero, lo que le permite comunicarse por Bluetooth, manejar el chip inalámbrico y utilizar el ADC. Habilitamos el uso de salida por USB para este segundo programa, y desactivamos la salida por UART. Esto nos permite ver la información en la consola mediante conexión USB.

Igualmente, incluimos el directorio actual para acceso a archivos de configuración, y definimos una macro llamada `RUNNING_AS_CLIENT` para que el programa se compile con la lógica de cliente habilitada. Por último, generamos las salidas adicionales para el cliente, igual que con el servidor.

0.3. Conclusion.

Este uso de BLE con Raspberry Pi Pico nos demuestra la viabilidad de utilizar redes inalámbricas en sistemas de bajo consumo. Con un rendimiento respetuoso con el medio ambiente, tanto a nivel de pila Bluetooth como a nivel de código, se establece una única comunicación entre el cliente BLE y el servidor para proporcionar una transmisión de datos. Está práctica no solo monitorea sensores de forma remota, sino que sienta las bases para futuras aplicaciones de IoT que requieren conectividad inalámbrica y rendimiento en tiempo real.

0.4. Resultados.

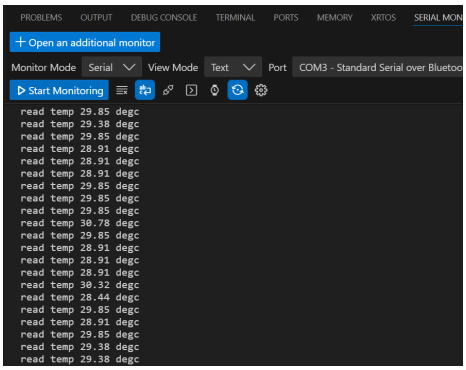

Figura 2: Resultados en el Monitor Serial.

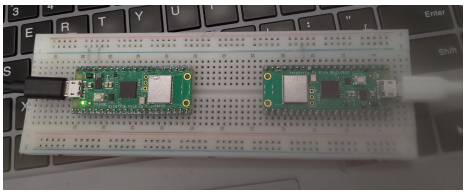

Figura 3: Prueba de las Raspberrys Cliente y Servidor.

0.5. Anexos

0.5.1. btstack-config.h

```
#ifndef _PICO_BTSTACK_BTSTACK_CONFIG_H
#define _PICO_BTSTACK_BTSTACK_CONFIG_H

#ifndef ENABLE_BLE
#error Please link to pico_btstack_ble
#endif

// BTstack features that can be enabled
#define ENABLE_LE_PERIPHERAL
#define ENABLE_LOG_INFO
#define ENABLE_LOG_ERROR
#define ENABLE_PRINTF_HEXDUMP

// for the client
#if RUNNING_AS_CLIENT
#define ENABLE_LE_CENTRAL
#define MAX_NR_GATT_CLIENTS 1
#else
#define MAX_NR_GATT_CLIENTS 0
#endif

// BTstack configuration. buffers, sizes, ...
#define HCI_OUTGOING_PRE_BUFFER_SIZE 4
#define HCI_ACL_PAYLOAD_SIZE (255 + 4)
#define HCI_ACL_CHUNK_SIZE_ALIGNMENT 4
#define MAX_NR_HCI_CONNECTIONS 1
#define MAX_NR_SM_LOOKUP_ENTRIES 3
#define MAX_NR_WHITELIST_ENTRIES 16
#define MAX_NR_LE_DEVICE_DB_ENTRIES 16

// Limit number of ACL/SCO Buffer to use by stack to avoid cyw43 shared bus overrun
#define MAX_NR_CONTROLLER_ACL_BUFFERS 3
#define MAX_NR_CONTROLLER_SCO_PACKETS 3

// Enable and configure HCI Controller to Host Flow Control to avoid cyw43 shared bus overrun
#define ENABLE_HCI_CONTROLLER_TO_HOST_FLOW_CONTROL
#define HCI_HOST_ACL_PACKET_LEN (255+4)
#define HCI_HOST_ACL_PACKET_NUM 3
#define HCI_HOST_SCO_PACKET_LEN 120
#define HCI_HOST_SCO_PACKET_NUM 3

// 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
```

0.5.2. Client.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();
```

```
}

static bool advertisement_report_contains_service(uint16_t service, uint8_t *advertisement_rep
    // get advertisement from report event
    const uint8_t * adv_data = gap_event_advertising_report_get_data(advertisement_report);
    uint8_t adv_len  = gap_event_advertising_report_get_data_length(advertisement_report);

    // iterate over advertisement data
    ad_context_t context;
    for (ad_iterator_init(&context, adv_len, adv_data) ; ad_iterator_has_more(&context) ; ad_i
        uint8_t data_type = ad_iterator_get_data_type(&context);
        uint8_t data_size = ad_iterator_get_data_len(&context);
        const uint8_t * data = ad_iterator_get_data(&context);
        switch (data_type){
            case BLUETOOTH_DATA_TYPE_COMPLETE_LIST_OF_16_BIT_SERVICE_CLASS_UUIDS:
                for (int i = 0; i < data_size; i += 2) {
                    uint16_t type = little_endian_read_16(data, i);
                    if (type == service) return true;
                }
            default:
                break;
        }
    }
    return false;
}

static void handle_gatt_client_event(uint8_t packet_type, uint16_t channel, uint8_t *packet, u
    UNUSED(packet_type);
    UNUSED(channel);
    UNUSED(size);

    uint8_t att_status;
    switch(state){
        case TC_W4_SERVICE_RESULT:
            switch(hci_event_packet_get_type(packet)) {
                case GATT_EVENT_SERVICE_QUERY_RESULT:
                    // store service (we expect only one)
                    DEBUG_LOG("Storing service\n");
                    gatt_event_service_query_result_get_service(packet, &server_service);
                    break;
                case GATT_EVENT_QUERY_COMPLETE:
                    att_status = gatt_event_query_complete_get_att_status(packet);
                    if (att_status != ATT_ERROR_SUCCESS){
                        printf("SERVICE_QUERY_RESULT, ATT Error 0x%02x.\n", att_status);
                        gap_disconnect(connection_handle);
                        break;
                    }
                    // service query complete, look for characteristic
                    state = TC_W4_CHARACTERISTIC_RESULT;
                    DEBUG_LOG("Search for env sensing characteristic.\n");
                    gatt_client_discover_characteristics_for_service_by_uuid16(handle_gatt_cli
                    break;
                default:
                    break;
            }
            break;
        case TC_W4_CHARACTERISTIC_RESULT:
```

```
            switch(hci_event_packet_get_type(packet)) {
                case GATT_EVENT_CHARACTERISTIC_QUERY_RESULT:
                    DEBUG_LOG("Storing characteristic\n");
                    gatt_event_characteristic_query_result_get_characteristic(packet, &server_
                    break;
                case GATT_EVENT_QUERY_COMPLETE:
                    att_status = gatt_event_query_complete_get_att_status(packet);
                    if (att_status != ATT_ERROR_SUCCESS){
                        printf("CHARACTERISTIC_QUERY_RESULT, ATT Error 0x%02x.\n", att_status)
                        gap_disconnect(connection_handle);
                        break;
                    }
                    // register handler for notifications
                    listener_registered = true;
                    gatt_client_listen_for_characteristic_value_updates(&notification_listener
                    // enable notifications
                    DEBUG_LOG("Enable notify on characteristic.\n");
                    state = TC_W4_ENABLE_NOTIFICATIONS_COMPLETE;
                    gatt_client_write_client_characteristic_configuration(handle_gatt_client_ev
                        &server_characteristic, GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_NOTI
                    break;
                default:
                    break;
            }
            break;
        case TC_W4_ENABLE_NOTIFICATIONS_COMPLETE:
            switch(hci_event_packet_get_type(packet)) {
                case GATT_EVENT_QUERY_COMPLETE:
                    DEBUG_LOG("Notifications enabled, ATT status 0x%02x\n", gatt_event_query_c
                    if (gatt_event_query_complete_get_att_status(packet) != ATT_ERROR_SUCCESS)
                    state = TC_W4_READY;
                    break;
                default:
                    break;
            }
            break;
        case TC_W4_READY:
            switch(hci_event_packet_get_type(packet)) {
                case GATT_EVENT_NOTIFICATION: {
                    uint16_t value_length = gatt_event_notification_get_value_length(packet);
                    const uint8_t *value = gatt_event_notification_get_value(packet);
                    DEBUG_LOG("Indication value len %d\n", value_length);
                    if (value_length == 2) {
                        float temp = little_endian_read_16(value, 0);
                        printf("read temp %.2f degc\n", temp / 100);
                    } else {
                        printf("Unexpected length %d\n", value_length);
                    }
                    break;
                }
                default:
                    printf("Unknown packet type 0x%02x\n", hci_event_packet_get_type(packet));
                    break;
            }
            break;
        default:
            printf("error\n");
```

```
            break;
    }
}

static void hci_event_handler(uint8_t packet_type, uint16_t channel, uint8_t *packet, uint16_t
    UNUSED(size);
    UNUSED(channel);
    bd_addr_t local_addr;
    if (packet_type != HCI_EVENT_PACKET) return;

    uint8_t event_type = hci_event_packet_get_type(packet);
    switch(event_type){
        case BTSTACK_EVENT_STATE:
            if (btstack_event_state_get_state(packet) == HCI_STATE_WORKING) {
                gap_local_bd_addr(local_addr);
                printf("BTstack up and running on %s.\n", bd_addr_to_str(local_addr));
                client_start();
            } else {
                state = TC_OFF;
            }
            break;
        case GAP_EVENT_ADVERTISING_REPORT:
            if (state != TC_W4_SCAN_RESULT) return;
            // check name in advertisement
            if (!advertisement_report_contains_service(ORG_BLUETOOTH_SERVICE_ENVIRONMENTAL_SENS
            // store address and type
            gap_event_advertising_report_get_address(packet, server_addr);
            server_addr_type = gap_event_advertising_report_get_address_type(packet);
            // stop scanning, and connect to the device
            state = TC_W4_CONNECT;
            gap_stop_scan();
            printf("Connecting to device with addr %s.\n", bd_addr_to_str(server_addr));
            gap_connect(server_addr, server_addr_type);
            break;
        case HCI_EVENT_LE_META:
            // wait for connection complete
            switch (hci_event_le_meta_get_subevent_code(packet)) {
                case HCI_SUBEVENT_LE_CONNECTION_COMPLETE:
                    if (state != TC_W4_CONNECT) return;
                    connection_handle = hci_subevent_le_connection_complete_get_connection_han
                    // initialize gatt client context with handle, and add it to the list of a
                    // query primary services
                    DEBUG_LOG("Search for env sensing service.\n");
                    state = TC_W4_SERVICE_RESULT;
                    gatt_client_discover_primary_services_by_uuid16(handle_gatt_client_event,
                    break;
                default:
                    break;
            }
            break;
        case HCI_EVENT_DISCONNECTION_COMPLETE:
            // unregister listener
            connection_handle = HCI_CON_HANDLE_INVALID;
            if (listener_registered){
                listener_registered = false;
                gatt_client_stop_listening_for_characteristic_value_updates(&notification_list
            }
```

```
            printf("Disconnected %s\n", bd_addr_to_str(server_addr));
            if (state == TC_OFF) break;
            client_start();
            break;
        default:
            break;
    }
}

static void heartbeat_handler(struct btstack_timer_source *ts) {
    // Invert the led
    static bool quick_flash;
    static bool led_on = true;

    led_on = !led_on;
    cyw43_arch_gpio_put(CYW43_WL_GPIO_LED_PIN, led_on);
    if (listener_registered && led_on) {
        quick_flash = !quick_flash;
    } else if (!listener_registered) {
        quick_flash = false;
    }

    // Restart timer
    btstack_run_loop_set_timer(ts, (led_on || quick_flash) ? LED_QUICK_FLASH_DELAY_MS : LED_SL
    btstack_run_loop_add_timer(ts);
}

int main() {
    stdio_init_all();

    // initialize CYW43 driver architecture (will enable BT if/because CYW43_ENABLE_BLUETOOTH
    if (cyw43_arch_init()) {
        printf("failed to initialise cyw43_arch\n");
        return -1;
    }

    l2cap_init();
    sm_init();
    sm_set_io_capabilities(IO_CAPABILITY_NO_INPUT_NO_OUTPUT);

    // setup empty ATT server - only needed if LE Peripheral does ATT queries on its own, e.g.
    att_server_init(NULL, NULL, NULL);

    gatt_client_init();

    hci_event_callback_registration.callback = &hci_event_handler;
    hci_add_event_handler(&hci_event_callback_registration);

    // set one-shot btstack timer
    heartbeat.process = &heartbeat_handler;
    btstack_run_loop_set_timer(&heartbeat, LED_SLOW_FLASH_DELAY_MS);
    btstack_run_loop_add_timer(&heartbeat);

    // turn on!
    hci_power_control(HCI_POWER_ON);

    // btstack_run_loop_execute is only required when using the 'polling' method (e.g. using p
```

```
    // This example uses the 'threadsafe background' method, where BT work is handled in a low
    // is fine to call bt_stack_run_loop_execute() but equally you can continue executing user

#if 1 // this is only necessary when using polling (which we aren't, but we're showing it is s
    btstack_run_loop_execute();
#else
    // this core is free to do it's own stuff except when using 'polling' method (in which cas
    // btstacK_run_loop_ methods to add work to the run loop.

    // this is a forever loop in place of where user code would go.
    while(true) {
        sleep_ms(1000);
    }
#endif
    return 0;
}
```

0.5.3. 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'
    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;
    if (packet_type != HCI_EVENT_PACKET) return;

    uint8_t event_type = hci_event_packet_get_type(packet);
    switch(event_type){
        case BTSTACK_EVENT_STATE:
            if (btstack_event_state_get_state(packet) != HCI_STATE_WORKING) return;
```

```
            gap_local_bd_addr(local_addr);
            printf("BTstack up and running on %s.\n", bd_addr_to_str(local_addr));

            // setup advertisements
            uint16_t adv_int_min = 800;
            uint16_t adv_int_max = 800;
            uint8_t adv_type = 0;
            bd_addr_t null_addr;
            memset(null_addr, 0, 6);
            gap_advertisements_set_params(adv_int_min, adv_int_max, adv_type, 0, null_addr, 0x
            assert(adv_data_len <= 31); // ble limitation
            gap_advertisements_set_data(adv_data_len, (uint8_t*) adv_data);
            gap_advertisements_enable(1);

            poll_temp();

            break;
        case HCI_EVENT_DISCONNECTION_COMPLETE:
            le_notification_enabled = 0;
            break;
        case ATT_EVENT_CAN_SEND_NOW:
            att_server_notify(con_handle, ATT_CHARACTERISTIC_ORG_BLUETOOTH_CHARACTERISTIC_TEMPE
            break;
        default:
            break;
    }
}

uint16_t att_read_callback(hci_con_handle_t connection_handle, uint16_t att_handle, uint16_t o
    UNUSED(connection_handle);

    if (att_handle == ATT_CHARACTERISTIC_ORG_BLUETOOTH_CHARACTERISTIC_TEMPERATURE_01_VALUE_HAN
        return att_read_callback_handle_blob((const uint8_t *)&current_temp, sizeof(current_te
    }
    return 0;
}

int att_write_callback(hci_con_handle_t connection_handle, uint16_t att_handle, uint16_t trans
    UNUSED(transaction_mode);
    UNUSED(offset);
    UNUSED(buffer_size);

    if (att_handle != ATT_CHARACTERISTIC_ORG_BLUETOOTH_CHARACTERISTIC_TEMPERATURE_01_CLIENT_CO
    le_notification_enabled = little_endian_read_16(buffer, 0) == GATT_CLIENT_CHARACTERISTICS_
    con_handle = connection_handle;
    if (le_notification_enabled) {
        att_server_request_can_send_now_event(con_handle);
    }
    return 0;
}

void poll_temp(void) {
    adc_select_input(ADC_CHANNEL_TEMPSENSOR);
    uint32_t raw32 = adc_read();
    const uint32_t bits = 12;

    // Scale raw reading to 16 bit value using a Taylor expansion (for 8 <= bits <= 16)
```

```
    uint16_t raw16 = raw32 << (16 - bits) | raw32 >> (2 * bits - 16);

    // ref https://github.com/raspberrypi/pico-micropython-examples/blob/master/adc/temperatur
    const float conversion_factor = 3.3 / (65535);
    float reading = raw16 * conversion_factor;

    // The temperature sensor measures the Vbe voltage of a biased bipolar diode, connected to
    // Typically, Vbe = 0.706V at 27 degrees C, with a slope of -1.721mV (0.001721) per degree
    float deg_c = 27 - (reading - 0.706) / 0.001721;
    current_temp = deg_c * 100;
    printf("Write temp %.2f degc\n", deg_c);
}
```

0.5.4. Server-common.h

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

#ifndef SERVER_COMMON_H_
#define SERVER_COMMON_H_

#define ADC_CHANNEL_TEMPSENSOR 4

extern int le_notification_enabled;
extern hci_con_handle_t con_handle;
extern uint16_t current_temp;
extern uint8_t const profile_data[];

void packet_handler(uint8_t packet_type, uint16_t channel, uint8_t *packet, uint16_t size);
uint16_t att_read_callback(hci_con_handle_t connection_handle, uint16_t att_handle, uint16_t o
int att_write_callback(hci_con_handle_t connection_handle, uint16_t att_handle, uint16_t trans
void poll_temp(void);

#endif
```

0.5.5. Server.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/btstack_cyw43.h"
#include "hardware/adc.h"
#include "pico/stdlib.h"

#include "server_common.h"
```

```
#define HEARTBEAT_PERIOD_MS 100

static btstack_timer_source_t heartbeat;
static btstack_packet_callback_registration_t hci_event_callback_registration;

static void heartbeat_handler(struct btstack_timer_source *ts) {
    static uint32_t counter = 0;
    counter++;

    // Update the temp every 10s
    if (counter % 10 == 0) {
        poll_temp();
        if (le_notification_enabled) {
            att_server_request_can_send_now_event(con_handle);
        }
    }

    // Invert the led
    static int led_on = true;
    led_on = !led_on;
    cyw43_arch_gpio_put(CYW43_WL_GPIO_LED_PIN, led_on);

    // Restart timer
    btstack_run_loop_set_timer(ts, HEARTBEAT_PERIOD_MS);
    btstack_run_loop_add_timer(ts);
}

int main() {
    stdio_init_all();

    // initialize CYW43 driver architecture (will enable BT if/because CYW43_ENABLE_BLUETOOTH
    if (cyw43_arch_init()) {
        printf("failed to initialise cyw43_arch\n");
        return -1;
    }

    // Initialise adc for the temp sensor
    adc_init();
    adc_select_input(ADC_CHANNEL_TEMPSENSOR);
    adc_set_temp_sensor_enabled(true);

    l2cap_init();
    sm_init();

    att_server_init(profile_data, att_read_callback, att_write_callback);

    // inform about BTstack state
    hci_event_callback_registration.callback = &packet_handler;
    hci_add_event_handler(&hci_event_callback_registration);

    // register for ATT event
    att_server_register_packet_handler(packet_handler);

    // set one-shot btstack timer
    heartbeat.process = &heartbeat_handler;
    btstack_run_loop_set_timer(&heartbeat, HEARTBEAT_PERIOD_MS);
```

```
    btstack_run_loop_add_timer(&heartbeat);

    // turn on bluetooth!
    hci_power_control(HCI_POWER_ON);

    // btstack_run_loop_execute is only required when using the 'polling' method (e.g. using p
    // This example uses the 'threadsafe background' method, where BT work is handled in a low
    // is fine to call bt_stack_run_loop_execute() but equally you can continue executing user

#if 0 // btstack_run_loop_execute() is not required, so lets not use it
    btstack_run_loop_execute();
#else
    // this core is free to do it's own stuff except when using 'polling' method (in which cas
    // btstacK_run_loop_ methods to add work to the run loop.

    // this is a forever loop in place of where user code would go.
    while(true) {
        sleep_ms(100);
    }
#endif
    return 0;
}
```

0.5.6. CMakeList.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
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)
```