Práctica 6. Bluetooth Low Energy (BLE)

Objetivos

- Diseccionar en detalle un *firmware* de un cliente GATT utilizando la API de ESP-IDF.
- Aprender a realizar un escaneo de dispositivos BLE
- Conocer la información disponible en los anuncios BLE
- Gestionar la conexión desde un cliente BLE a un servidor (GATT) BLE

Implementación de un cliente GATT para escaneo y conexión a servidor

En esta práctica, se revisa el código de ejemplo para la construcción de un cliente GATT para el ESP32 utilizando ESP-IDF. El código implementa un cliente de Atributo Genérico (GATT) de Bluetooth Low Energy (BLE), que escanea servidores periféricos cercanos y se conecta a un servicio predefinido. El cliente busca características disponibles y se suscribe a una característica conocida para recibir notificaciones o indicaciones. El ejemplo puede registrar un Perfil de Aplicación e inicializa una secuencia de eventos que se pueden utilizar para configurar parámetros del Perfil de Acceso Genérico (GAP) y para manejar eventos como el escaneo, la conexión a periféricos y la lectura y escritura de características.

El desarrollo de esta práctica requiere el uso de dos placas: una ejecutando el servidor GATT básico (o modificado) que usaste en la práctica anterior, y otra ejecutando el código cliente.

Descripción del código de ejemplo

El ejemplo que seguiremos y adaptaremos se encuentra en la carpeta de ejemplos de ESP-IDF en bluetooth/bluedroid/ble/gatt_client/main (../main). El archivo gattc_demo.c (../main/gattc_demo.c), ubicado en la carpeta principa, contiene todas las funcionalidades que vamos a revisar.

> **Tarea previa**
>
> Antes de comenzar, asegúrate de que la variable `remote_device_name` (línea 41) NO coincide con la de tu servidor GATT.

Los archivos de encabezado contenidos en gattc_demo.c (../main/gattc_demo.c) son:

Ficheros de cabecera

Observa los ficheros de cabecera incluidos (similares a los que utilizaste en la práctica anterior):

```
#include <stdint.h>
#include <string.h>
#include <stdbool.h>
#include <stdio.h>
#include "nvs.h"
#include "nvs_flash.h"
#include "controller.h"

#include "bt.h"
#include "esp_gap_ble_api.h"
#include "esp_gattc_api.h"
#include "esp_gatt_defs.h"
#include "esp_bt_main.h"
#include "esp_gatt_common_api.h"
```

Estos `includes` son necesarios para que funcionen los componentes del sistema subyacente y FreeRTOS, incluida la funcionalidad de registro y una biblioteca para almacenar datos en memoria flash no volátil. Estamos interesados en `"bt.h"`, `"esp_bt_main.h"`, `"esp_gap_ble_api.h"` y `"esp_gattc_api.h"`, que exponen las API de BLE necesarias para implementar este ejemplo.

- `bt.h`: configura el controlador BT y VHCI desde el lado del host.
- `esp_bt_main.h`: inicializa y habilita la pila Bluedroid.
- `esp_gap_ble_api.h`: implementa la configuración GAP, por ejemplo los anuncios de dispositivos y los parámetros de conexión.
- `esp_gattc_api.h`: implementa la configuración del Cliente GATT, como la conexión a periféricos y la búsqueda de servicios.

Punto de Entrada Principal

La función de punto de entrada del programa es `app_main()`:

```
void app_main()
{
    // Inicializar NVS.
    esp_err_t ret = nvs_flash_init();
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }
    ESP_ERROR_CHECK( ret );

    esp_bt_controller_config_t bt_cfg = BT_CONTROLLER_INIT_CONFIG_DEFAULT();
    ret = esp_bt_controller_init(&bt_cfg);
    if (ret) {
        ESP_LOGE(GATTC_TAG, "%s inicialización del controlador fallida, código de error = %x", _
        return;
    }

    ret = esp_bt_controller_enable(ESP_BT_MODE_BLE);
    if (ret) {
        ESP_LOGE(GATTC_TAG, "%s habilitación del controlador fallida, código de error = %x", __f
        return;
    }

    esp_bluedroid_config_t bluedroid_cfg = BT_BLUEDROID_INIT_CONFIG_DEFAULT();
    ret = esp_bluedroid_init_with_cfg(&bluedroid_cfg);
    if (ret) {
        ESP_LOGE(GATTC_TAG, "%s inicialización de Bluetooth fallida, código de error = %x", __fu
        return;
    }

    ret = esp_bluedroid_enable();
    if (ret) {
        ESP_LOGE(GATTC_TAG, "%s habilitación de Bluetooth fallida, código de error = %x", __func
        return;
    }

    // Registrar la función de devolución de llamada en el módulo GAP
    ret = esp_ble_gap_register_callback(esp_gap_cb);
    if (ret){
        ESP_LOGE(GATTC_TAG, "%s registro de GAP fallido, código de error = %x", __func__, ret);
        return;
    }

    // Registrar la función de devolución de llamada en el módulo GATTC
    ret = esp_ble_gattc_register_callback(esp_gattc_cb);
    if(ret){
        ESP_LOGE(GATTC_TAG, "%s registro de GATTC fallido, código de error = %x", __func__, ret)
        return;
    }

    ret = esp_ble_gattc_app_register(PROFILE_A_APP_ID);
    if (ret){
        ESP_LOGE(GATTC_TAG, "%s registro de la aplicación GATTC fallido, código de error = %x",
    }

    esp_err_t local_mtu_ret = esp_ble_gatt_set_local_mtu(500);
    if (local_mtu_ret){
        ESP_LOGE(GATTC_TAG, "configuración del MTU local fallida, código de error = %x", local_m
    }
```

```
`
```

La función principal comienza inicializando la biblioteca de almacenamiento no volátil. Esta biblioteca permite guardar pares clave-valor en la memoria flash y se utiliza en algunos componentes, como la biblioteca Wi-Fi, para guardar el SSID y la contraseña:

```
esp_err_t ret = nvs_flash_init();
if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret == ESP_ERR_NVS_NEW_VERSION_FOUND) {
    ESP_ERROR_CHECK(nvs_flash_erase());
    ret = nvs_flash_init();
}
ESP_ERROR_CHECK( ret );
```

Inicialización del Controlador Bluetooth y la Pila BLE

La función principal también inicializa el controlador BT al crear primero una estructura de configuración del controlador BT llamada `esp_bt_controller_config_t` con ajustes predeterminados generados por la macro `BT_CONTROLLER_INIT_CONFIG_DEFAULT()`. El controlador BT implementa la Interfaz del Controlador Host (HCI) en el lado del controlador, la Capa de Enlace (LL) y la Capa Física (PHY). El controlador BT es invisible para las aplicaciones de usuario y se encarga de las capas inferiores de la pila BLE. La configuración del controlador incluye el tamaño de la pila del controlador BT, la prioridad y la velocidad de baudios HCI. Con la configuración creada, se inicializa y habilita el controlador BT con la función `esp_bt_controller_init()`:

```
esp_bt_controller_config_t bt_cfg = BT_CONTROLLER_INIT_CONFIG_DEFAULT();
ret = esp_bt_controller_init(&bt_cfg);
```

A continuación, el controlador se habilita en el modo BLE.

```
ret = esp_bt_controller_enable(ESP_BT_MODE_BLE);
```

> El controlador debe habilitarse en `ESP_BT_MODE_BTDM` si desea utilizar el modo dual (BLE + BT).

Hay cuatro modos de Bluetooth admitidos:

1. `ESP_BT_MODE_IDLE`: Bluetooth no se está ejecutando.
2. `ESP_BT_MODE_BLE`: Modo BLE.
3. `ESP_BT_MODE_CLASSIC_BT`: Modo BT clásico.
4. `ESP_BT_MODE_BTDM`: Modo dual (BLE + BT clásico).

Después de la inicialización del controlador BT, se inicializa y habilita la pila Bluedroid, que incluye las definiciones y API comunes tanto para BT clásico como para BLE. Esto se realiza mediante:

```
esp_bluedroid_config_t bluedroid_cfg = BT_BLUEDROID_INIT_CONFIG_DEFAULT();
ret = esp_bluedroid_init_with_cfg(&bluedroid_cfg);
ret = esp_bluedroid_enable();
```

La función principal finaliza registrando los controladores de eventos GAP y GATT, así como el Perfil de Aplicación y configurando el tamaño máximo admitido de MTU.

```
    // Registrar la función de devolución de llamada en el módulo GAP
    ret = esp_ble_gap_register_callback(esp_gap_cb);

    // Registrar la función de devolución de llamada en el módulo GATTC
    ret = esp_ble_gattc_register_callback(esp_gattc_cb);

    ret = esp_ble_gattc_app_register(PROFILE_A_APP_ID);

    esp_err_t local_mtu_ret = esp_ble_gatt_set_local_mtu(500);
    if (local_mtu_ret){
        ESP_LOGE(GATTC_TAG, "configuración del MTU local fallida, código de error = %x", local_m
    }
```

Los controladores de eventos GAP y GATT son las funciones utilizadas para capturar los eventos generados por la pila BLE y ejecutar funciones para configurar los parámetros de la aplicación. Además, los controladores de eventos también se utilizan para manejar eventos de lectura y escritura que provienen del dispositivo central. El controlador de eventos GAP se encarga del escaneo y la conexión a servidores, y el controlador GATT administra los eventos que ocurren después de que el cliente se haya conectado a un servidor, como la búsqueda de servicios y la escritura y lectura de datos. Los controladores de eventos GAP y GATT se registran mediante:

```
esp_ble_gap_register_callback();
esp_ble_gattc_register_callback();
```

Las funciones `esp_gap_cb()` y `esp_gattc_cb()` manejan todos los eventos generados por la pila BLE.

Perfiles de Aplicación

Los Perfiles de Aplicación son una forma de agrupar funcionalidades diseñadas para una o más aplicaciones de servidor. Por ejemplo, puede tener un Perfil de Aplicación conectado a sensores de ritmo cardíaco y otro conectado a sensores de temperatura. Cada Perfil de Aplicación crea una interfaz GATT para conectarse a otros dispositivos. Las estructuras de Perfiles de Aplicación en el código son instancias de la estructura `gattc_profile_inst`, que se define como:

```
struct gattc_profile_inst {
    esp_gattc_cb_t gattc_cb;
    uint16_t gattc_if;
    uint16_t app_id;
    uint16_t conn_id;
    uint16_t service_start_handle;
    uint16_t service_end_handle;
    uint16_t char_handle;
    esp_bd_addr_t remote_bda;
};
```

La estructura de Perfil de Aplicación contiene:

- `gattc_cb`: función de devolución de llamada del cliente GATT
- `gattc_if`: número de interfaz del cliente GATT para este perfil
- `app_id`: número de ID del Perfil de Aplicación
- `conn_id`: ID de conexión
- `service_start_handle`: mango de inicio del servicio
- `service_end_handle`: mango de fin del servicio
- `char_handle`: mango de característica
- `remote_bda`: dirección del dispositivo remoto conectado a este cliente.

En este ejemplo, hay un Perfil de Aplicación y su ID se define como:

```
#define PROFILE_NUM 1
#define PROFILE_A_APP_ID 0
```

Los Perfiles de Aplicación se almacenan en el array `gl_profile_tab`, que se inicializa de la siguiente manera:

```
/* Un perfil basado en GATT, un app_id y un gattc_if, este arreglo almacenará el gattc_if devuel
static struct gattc_profile_inst gl_profile_tab[PROFILE_NUM] = {
        [PROFILE_A_APP_ID] = {.gattc_cb = gattc_profile_event_handler,
                                .gattc_if = ESP_GATT_IF_NONE, /* No se obtiene el gatt_if, por
    },
};
```

La inicialización del array de tablas de Perfiles de Aplicación incluye la definición de la función de devolución de llamada para el Perfil. En este caso, es `gattc_profile_event_handler()`. Además, la interfaz GATT se inicializa con el valor predeterminado de `ESP_GATT_IF_NONE`. Más adelante, cuando se registre el Perfil de Aplicación, la pila BLE devolverá una instancia de interfaz GATT para usar con ese Perfil de Aplicación.

El registro del perfil desencadena un evento `ESP_GATTC_REG_EVT`, que es manejado por el manejador de eventos `esp_gattc_cb()`. El manejador toma la interfaz GATT devuelta por el evento y la almacena en la tabla de perfiles:

```
static void esp_gattc_cb(esp_gattc_cb_event_t event, esp_gatt_if_t gattc_if, esp_ble_gattc_cb_pa
{
    ESP_LOGI(GATTC_TAG, "EVT %d, gattc if %d", event, gattc_if);

    /* Si el evento es un evento de registro, almacena el gattc_if para cada perfil */
    if (event == ESP_GATTC_REG_EVT) {
        if (param->reg.status == ESP_GATT_OK) {
            gl_profile_tab[param->reg.app_id].gattc_if = gattc_if;
        } else {
            ESP_LOGI(GATTC_TAG, "registro de aplicación fallido, app_id %04x, estado %d",
                    param->reg.app_id,
                    param->reg.status);
            return;
        }
    }
…
```

Finalmente, la función de devolución de llamada invoca el manejador de eventos correspondiente para cada perfil en la tabla `gl_profile_tab`.

```
…
/* Si gattc_if es igual al perfil A, llamar al manejador de cb del perfil A,
     * por lo tanto, aquí llamar a la función cb de cada perfil */
    do {
        int idx;
        for (idx = 0; idx < PROFILE_NUM; idx++) {
            if (gattc_if == ESP_GATT_IF_NONE || /* ESP_GATT_IF_NONE, no especifica un cierto gat
                    gattc_if == gl_profile_tab[idx].gattc_if) {
                if (gl_profile_tab[idx].gattc_cb) {
                    gl_profile_tab[idx].gattc_cb(event, gattc_if, param);
                }
            }
        }
    } while (0);
}
```

Configuración de Parámetros de Escaneo

El cliente GATT normalmente escanea servidores cercanos y trata de conectarse a ellos si está interesado. Sin embargo, para realizar el escaneo, primero es necesario configurar los parámetros de configuración. Esto se hace después del registro de los Perfiles de Aplicación, porque una vez completado el registro, desencadena un evento `ESP_GATTC_REG_EVT`. La primera vez que se desencadena este evento, el manejador de eventos GATT lo captura y asigna una interfaz GATT al Perfil A. Luego, el evento se reenvía al manejador de eventos GATT del Perfil A. En este manejador de eventos, el evento se utiliza para llamar a la función `esp_ble_gap_set_scan_params()`, que toma una instancia de estructura `ble_scan_params` como parámetro. Esta estructura se define como:

```
/// Parámetros de escaneo BLE
typedef struct {
    esp_ble_scan_type_t     scan_type;              /*!< Tipo de escaneo */
    esp_ble_addr_type_t     own_addr_type;          /*!< Tipo de dirección propia */
    esp_ble_scan_filter_t   scan_filter_policy;     /*!< Política de filtro de escaneo */
    uint16_t                scan_interval;          /*!< Intervalo de escaneo. Se define como el
    //Rango: 0x0004 to 0x4000
    //Predeterminado: 0x0010 (10 ms)
    //Tiempo = N * 0.625 ms
    //Rango de tiempo: 2.5 ms a 10.24 segundos
    uint16_t                scan_window;            /*!< Ventana de escaneo. La duración del esc
    //Rango: 0x0004 to 0x4000
    //Predeterminado: 0x0010 (10 ms)
    //Tiempo = N * 0.625 ms
    //Rango de tiempo: 2.5 ms a 10240 ms
} esp_ble_scan_params_t;
```

Y se inicializa de la siguiente manera:

```
static esp_ble_scan_params_t ble_scan_params = {
    .scan_type              = BLE_SCAN_TYPE_ACTIVE,
    .own_addr_type          = BLE_ADDR_TYPE_PUBLIC,
    .scan_filter_policy     = BLE_SCAN_FILTER_ALLOW_ALL,
    .scan_interval          = 0x50,
    .scan_window            = 0x30
};
```

Los parámetros de escaneo BLE se configuran de manera que el tipo de escaneo sea activo (incluye la lectura de la respuesta de escaneo), es de tipo público, permite leer cualquier dispositivo anunciado y tiene un intervalo de escaneo de 100 ms (1.25 ms * 0x50) y una ventana de escaneo de 60 ms (1.25 ms * 0x30).

Los valores de escaneo se establecen utilizando la función `esp_ble_gap_set_scan_params()`:

```
case ESP_GATTC_REG_EVT:
        ESP_LOGI(GATTC_TAG, "REG_EVT");
        esp_err_t scan_ret = esp_ble_gap_set_scan_params(&ble_scan_params);
        if (scan_ret){
            ESP_LOGE(GATTC_TAG, "error al configurar parámetros de escaneo, código de error = %x
        }
        break;
```

Tarea Básica

Configura los parámetros de escaneo para que éste se produzca con menos frecuencia (e.g. 1 segundo o un valor superior). Para ello, adapta el valor del campo `scan_interval` con el valor apropiado.

Una vez que se establecen los parámetros de escaneo, se desencadena un evento `ESP_GAP_BLE_SCAN_PARAM_SET_COMPLETE_EVT`, que es manejado por el manejador de eventos GAP `esp_gap_cb()`. Este evento se utiliza para iniciar el escaneo de los servidores GATT cercanos:

```
case ESP_GAP_BLE_SCAN_PARAM_SET_COMPLETE_EVT: {
        //la unidad de la duración es en segundos
        uint32_t duration = 30;
        esp_ble_gap_start_scanning(duration);
        break;
}
```

Tarea Básica

El valor de duración del proceso de escaneado es configurable. En esta primera parte de la práctica, auméntalo para que el proceso sea más largo y así tengas más tiempo para analizar la salidaproducida. Comprueba que efectivamente el tiempo en el que el dispositivo se encuentra en fase de escaneo es el seleccionado.

Tarea Adicional

Analiza el código y modifícalo para que el proceso de escaneo se produzca cíclicamente, con un parámetro de separación entre fases de escaneado definido a través de los menús de configuració. De la misma manera, añade a dichos menús un parámetro adicional que indique el inervalo entre eventos de escaneo.

El escaneo se inicia mediante la función `esp_ble_gap_start_scanning()`, que toma un parámetro que representa la duración del escaneo continuo (en segundos). Una vez que finaliza el período de escaneo, se desencadena un evento `ESP_GAP_SEARCH_INQ_CMPL_EVT`.

Los resultados del escaneo se muestran tan pronto como llegan con el evento `ESP_GAP_BLE_SCAN_RESULT_EVT`, que incluye los siguientes parámetros:

```
    /**
     * @brief ESP_GAP_BLE_SCAN_RESULT_EVT
     */
    struct ble_scan_result_evt_param {
        esp_gap_search_evt_t search_evt;            /*!< Tipo de evento de búsqueda */
        esp_bd_addr_t bda;                          /*!< Dirección del dispositivo Bluetooth que
        esp_bt_dev_type_t dev_type;                 /*!< Tipo de dispositivo */
        esp_ble_addr_type_t ble_addr_type;          /*!< Tipo de dirección del dispositivo BLE *
        esp_ble_evt_type_t ble_evt_type;            /*!< Tipo de evento de resultado de escaneo
        int rssi;                                   /*!< RSSI del dispositivo buscado */
        uint8_t  ble_adv[ESP_BLE_ADV_DATA_LEN_MAX + ESP_BLE_SCAN_RSP_DATA_LEN_MAX]; /*!< EIR rec
        int flag;                                   /*!< Bit de indicación de datos de publicida
        int num_resps;                              /*!< Número de resultados de escaneo */
        uint8_t adv_data_len;                       /*!< Longitud de datos de publicidad */
        uint8_t scan_rsp_len;                       /*!< Longitud de respuesta de escaneo */
    } scan_rst;                                     /*!< Parámetro de evento de ESP_GAP_BLE_SCAN
```

Este evento también incluye una lista de subeventos, como se muestra a continuación:

```
/// Sub Evento de ESP_GAP_BLE_SCAN_RESULT_EVT
typedef enum {
    ESP_GAP_SEARCH_INQ_RES_EVT             = 0,     /*!< Resultado de la investigación para un
    ESP_GAP_SEARCH_INQ_CMPL_EVT            = 1,     /*!< Búsqueda completa. */
    ESP_GAP_SEARCH_DISC_RES_EVT            = 2,     /*!< Resultado del descubrimiento para un d
    ESP_GAP_SEARCH_DISC_BLE_RES_EVT        = 3,     /*!< Resultado del descubrimiento para serv
    ESP_GAP_SEARCH_DISC_CMPL_EVT           = 4,     /*!< Descubrimiento completo. */
    ESP_GAP_SEARCH_DI_DISC_CMPL_EVT        = 5,     /*!< Descubrimiento completo. */
    ESP_GAP_SEARCH_SEARCH_CANCEL_CMPL_EVT  = 6,     /*!< Búsqueda cancelada */
} esp_gap_search_evt_t;
```

Nos interesa el evento `ESP_GAP_SEARCH_INQ_RES_EVT`, que se llama cada vez que se encuentra un nuevo dispositivo. También nos interesa el evento `ESP_GAP_SEARCH_INQ_CMPL_EVT`, que se desencadena cuando se completa la duración del escaneo y se puede utilizar para reiniciar el procedimiento de escaneo:

```
case ESP_GAP_BLE_SCAN_RESULT_EVT: {
        esp_ble_gap_cb_param_t *scan_result = (esp_ble_gap_cb_param_t *)param;
        switch (scan_result->scan_rst.search_evt) {
            case ESP_GAP_SEARCH_INQ_RES_EVT:
                esp_log_buffer_hex(GATTC_TAG, scan_result->scan_rst.bda, 6);
                ESP_LOGI(GATTC_TAG, "Longitud de datos de publicidad buscada %d, Longitud de res
                adv_name = esp_ble_resolve_adv_data(scan_result->scan_rst.ble_adv, ESP_BLE_AD_TY
                ESP_LOGI(GATTC_TAG, "Longitud del nombre del dispositivo buscado %d", adv_name_l
                esp_log_buffer_char(GATTC_TAG, adv_name, adv_name_len);
                ESP_LOGI(GATTC_TAG, " ");
                if (adv_name != NULL) {
                    if (strlen(remote_device_name) == adv_name_len && strncmp((char *)adv_name,
                    ESP_LOGI(GATTC_TAG, "dispositivo buscado %s", remote_device_name);
                    if (connect == false) {
                        connect = true;
                        ESP_LOGI(GATTC_TAG, "conectar al dispositivo remoto.");
                        esp_ble_gap_stop_scanning();
                        esp_ble_gattc_open(gl_profile_tab[PROFILE_A_APP_ID].gattc_if, scan_resul
                    }
                }
            }
            break;
```

Primero, se resuelve el nombre del dispositivo y se compara con el nombre del dispositivo del servidor GATT en el que estamos interesados. Si coincide con el nombre del dispositivo del servidor GATT que estamos buscando, se detiene el escaneo.

Cada vez que recibimos un resultado del evento `ESP_GAP_SEARCH_INQ_RES_EVT`, el código primero imprime la dirección del dispositivo remoto:

```
case ESP_GAP_SEARCH_INQ_RES_EVT:
     esp_log_buffer_hex(GATTC_TAG, scan_result->scan_rst.bda, 6);
```

Luego, el cliente imprime la longitud de los datos anunciados y la longitud de la respuesta de escaneo:

```
ESP_LOGI(GATTC_TAG, "searched Adv Data Len %d, Scan Response Len %d", scan_result->scan_rst.adv_
```

Para obtener el nombre del dispositivo, utilizamos la función `esp_ble_resolve_adv_data()`, que toma los datos anunciados almacenados en `scan_result->scan_rst.ble_adv`, el tipo de datos anunciados y la longitud, para extraer el valor del paquete publicitario. Luego, se imprime el nombre del dispositivo.

```
adv_name = esp_ble_resolve_adv_data(scan_result->scan_rst.ble_adv, ESP_BLE_AD_TYPE_NAME_CMPL, &a
ESP_LOGI(GATTC_TAG, "searched Device Name Len %d", adv_name_len);
esp_log_buffer_char(GATTC_TAG, adv_name, adv_name_len);
```

Tarea básica

Muestra por pantalla el valor de RSSI de cada dispositivo BLE encontrado. Para ello, busca en la estructura de parámetros recibidos el campo correspondiente.

Tarea básica

Añade a los menús de configuración un campo que indique la dirección MAC BLE de un dispositivo (conocido) que esté en fase de anuncio. Puedes utilizar los códigos de servidor de la práctica anterior.

Tarea básica

Modifica la fase de escaneado para que únicamente se muestre información (al menos nombre y RSSI) sobre el dispositivo cuya dirección MAC se haya configurado.

Tarea básica

Modifica el *firmware* básico para que, en función del valor de RSSI obtenido para el dispositivo de interés, el cliente reporte con más o menos frecuencia por pantalla un valor proporcional a la distancia supuesta para dicho dispositivo. Si dispones de un LED conectado a la placa, puedes hacer que la frecuencia de parpadeo del mismo sea proporcional a dicha distancia.

Tarea previa

A partir de este punto puedes volver a fijar el nombre del dispositivo remoto de interés en función del nombre otorgado en el servidor GATT.

Finalmente, si el nombre del dispositivo remoto es el mismo que hemos definido anteriormente, el dispositivo local detiene el escaneo y trata de abrir una conexión con el dispositivo remoto utilizando la función `esp_ble_gattc_open()`. Esta función toma como parámetros la interfaz GATT del Perfil de Aplicación, la dirección del servidor remoto y un valor booleano. El valor booleano se utiliza para indicar si la conexión se realiza directamente o en segundo plano (autoconexión). En este momento, este valor booleano debe establecerse en verdadero para establecer la conexión. Ten en cuenta que el cliente abre una conexión virtual con el servidor. La conexión virtual devuelve un ID de conexión. La conexión virtual es la conexión entre el Perfil de Aplicación y el servidor remoto. Dado que muchos Perfiles de Aplicación pueden ejecutarse en un ESP32, podría haber muchas conexiones virtuales abiertas al mismo servidor remoto. También está la conexión física, que es el enlace BLE real entre el cliente y el servidor. Por lo tanto, si la conexión física se desconecta con la función `esp_ble_gap_disconnect()`, se cierran todas las demás conexiones virtuales. En este ejemplo, cada Perfil de Aplicación crea una conexión virtual al mismo servidor con la función `esp_ble_gattc_open()`, por lo que cuando se llama a la función de cierre, solo se cierra esa conexión del Perfil de Aplicación, mientras que si se llama a la función de desconexión GAP, se cerrarán ambas conexiones. Además, los eventos de conexión se propagan a todos los perfiles porque se relacionan con la conexión física, mientras que los eventos de apertura se propagan solo al perfil que crea la conexión virtual.

Configuración del Tamaño de MTU

El ATT_MTU se define como el tamaño máximo de cualquier paquete enviado entre un cliente y un servidor. Cuando el cliente se conecta al servidor, informa al servidor qué tamaño de MTU usar intercambiando unidades de datos de protocolo de solicitud y respuesta de MTU (PDUs). Esto se hace después de abrir la conexión. Después de abrir la conexión, se desencadena un evento `ESP_GATTC_CONNECT_EVT`:

```
    case ESP_GATTC_CONNECT_EVT:
       //p_data->connect.status siempre será ESP_GATT_OK
       ESP_LOGI(GATTC_TAG, "ESP_GATTC_CONNECT_EVT conn_id %d, if %d, status %d", conn_id, gattc
       conn_id = p_data->connect.conn_id;
       gl_profile_tab[PROFILE_A_APP_ID].conn_id = p_data->connect.conn_id;
       memcpy(gl_profile_tab[PROFILE_A_APP_ID].remote_bda, p_data->connect.remote_bda, sizeof(e
       ESP_LOGI(GATTC_TAG, "BDA REMOTO:");
       esp_log_buffer_hex(GATTC_TAG, gl_profile_tab[PROFILE_A_APP_ID].remote_bda, sizeof(esp_bd
       esp_err_t mtu_ret = esp_ble_gattc_send_mtu_req (gattc_if, conn_id);
       if (mtu_ret){
           ESP_LOGE(GATTC_TAG, "error de configuración de MTU, código de error = %x", mtu_ret);
       }
       break;
```

El ID de conexión y la dirección del dispositivo remoto (servidor) se almacenan en la tabla de Perfiles de la Aplicación y se imprimen en la consola:

```
conn_id = p_data->connect.conn_id;
gl_profile_tab[PROFILE_A_APP_ID].conn_id = p_data->connect.conn_id;
memcpy(gl_profile_tab[PROFILE_A_APP_ID].remote_bda, p_data->connect.remote_bda, sizeof(esp_bd_ad
ESP_LOGI(GATTC_TAG, "BDA REMOTO:");
esp_log_buffer_hex(GATTC_TAG, gl_profile_tab[PROFILE_A_APP_ID].remote_bda, sizeof(esp_bd_addr_t)
```

El tamaño típico del MTU para una conexión Bluetooth 4.0 es de 23 bytes. Un cliente puede cambiar el tamaño del MTU utilizando la función `esp_ble_gattc_send_mtu_req()`, que toma la interfaz GATT y el ID de conexión. El tamaño del MTU solicitado se define mediante `esp_ble_gatt_set_local_mtu()`. Luego, el servidor puede aceptar o rechazar la solicitud. El ESP32 admite un tamaño de MTU de hasta 517 bytes, que se define en `ESP_GATT_MAX_MTU_SIZE` en `esp_gattc_api.h`. En este ejemplo, el tamaño del MTU se establece en 500 bytes. En caso de que la configuración falle, se imprime el error devuelto:

```
esp_err_t mtu_ret = esp_ble_gattc_send_mtu_req (gattc_if, conn_id);
if (mtu_ret){
    ESP_LOGE(GATTC_TAG, "error de configuración de MTU, código de error = %x", mtu_ret);
}
break;
```

La apertura de la conexión también desencadena un evento `ESP_GATTC_OPEN_EVT`, que se utiliza para comprobar si la apertura de la conexión se realizó con éxito; de lo contrario, se imprime un error y se sale del programa:

```
case ESP_GATTC_OPEN_EVT:
        if (param->open.status != ESP_GATT_OK){
            ESP_LOGE(GATTC_TAG, "apertura fallida, estado %d", p_data->open.status);
            break;
        }
ESP_LOGI(GATTC_TAG, "apertura exitosa");
```

Cuando se intercambia el MTU, se desencadena un evento `ESP_GATTC_CFG_MTU_EVT`, que en este ejemplo se utiliza para imprimir el nuevo tamaño del MTU:

```
case ESP_GATTC_CFG_MTU_EVT:
        if (param->cfg_mtu.status != ESP_GATT_OK){
            ESP_LOGE(GATTC_TAG,"configuración de MTU fallida, estado de error = %x", param->cfg_
        }
        ESP_LOGI(GATTC_TAG, "ESP_GATTC_CFG_MTU_EVT, Estado %d, MTU %d, ID de conexión %d", param
…
```

La configuración del MTU se utiliza también para comenzar a descubrir los servicios disponibles en el servidor al que se ha conectado el cliente. Para descubrir los servicios, se utiliza la función `esp_ble_gattc_search_service()`. Los parámetros de la función son la interfaz GATT, el ID de conexión del perfil de la aplicación y el UUID de la aplicación de servicio que interesa al cliente. El servicio que estamos buscando se define de la siguiente manera:

```
static esp_bt_uuid_t remote_filter_service_uuid = {
    .len = ESP_UUID_LEN_16,
    .uuid = {.uuid16 = REMOTE_SERVICE_UUID,},
};
```

Donde,

```
#define REMOTE_SERVICE_UUID        0x00FF
```

Si el UUID de la aplicación de servicio que interesa al usuario es de 128 bits, hay una nota relevante para el usuario que se relaciona con el modo de almacenamiento en "little-endian" de la arquitectura del procesador.

La estructura del UUID se define de la siguiente manera:

```
typedef struct {
#define ESP_UUID_LEN_16     2
#define ESP_UUID_LEN_32     4
#define ESP_UUID_LEN_128    16
    uint16_t len;                           /*!< Longitud del UUID, 16 bits, 32 bits o 128 bits
    union {
        uint16_t    uuid16;                 /*!< UUID de 16 bits */
        uint32_t    uuid32;                 /*!< UUID de 32 bits */
        uint8_t     uuid128[ESP_UUID_LEN_128]; /*!< UUID de 128 bits */
    } uuid;                                 /*!< UUID */
} __attribute__((packed)) esp_bt_uuid_t;
```

Nota: En el modo de almacenamiento en "little-endian", puedes definir directamente el UUID del servicio en el orden normal si es un UUID de 16 bits o 32 bits, pero si el UUID del servicio es de 128 bits, hay una pequeña diferencia. Por ejemplo, si el UUID de la aplicación de servicio que le interesa al usuario es 12345678-a1b2-c3d4-e5f6-9fafd205e457, `REMOTE_SERVICE_UUID` debería definirse como {0x57,0xE4,0x05,0xD2,0xAF,0x9F,0xF6,0xE5,0xD4,0xC3,0xB2,0xA1,0x78,0x56,0x34,0x12}.

Los servicios se descubren de la siguiente manera:

```
esp_ble_gattc_search_service(gattc_if, param->cfg_mtu.conn_id, &remote_filter_service_uuid);
        break;
```

El servicio encontrado, si lo hay, se devolverá desde un evento `ESP_GATTC_SEARCH_RES_EVT`. Para cada servicio encontrado, se desencadena un evento para imprimir información sobre el servicio descubierto, según el tamaño del UUID:

```
 case ESP_GATTC_SEARCH_RES_EVT: {
        esp_gatt_srvc_id_t *srvc_id = &p_data->search_res.srvc_id;
        conn_id = p_data->search_res.conn_id;
        Si el `len` del UUID en `srvc_id->id.uuid` es igual a `ESP_UUID_LEN_16` y `uuid16` coinc
        se establece `get_server` en true. Luego, se almacena el valor de inicio (`start_handle`
        que se utilizarán más adelante para obtener todas las características de ese servicio.
        Finalmente, después de que se devuelvan todos los resultados de los servicios,
        la búsqueda se completa y se desencadena un evento `ESP_GATTC_SEARCH_CMPL_EVT`.
        ESP_LOGI(GATTC_TAG, "UUID16: %x", srvc_id->id.uuid.uuid.uuid16);
        }
        break;
```

En caso de que el cliente encuentre el servicio que busca, la bandera `get_server` se establece en true y se guardan los valores de inicio y fin que se utilizarán posteriormente para obtener todas las características de ese servicio. Una vez que se han devuelto todos los resultados de los servicios, se completa la búsqueda y se desencadena un evento `ESP_GATTC_SEARCH_CMPL_EVT`.

Obteniendo Características

Este ejemplo implementa la obtención de datos de características de un servicio predefinido. El servicio del cual queremos obtener características tiene un UUID de 0x00FF, y la característica de interés tiene un UUID de 0xFF01:

```
#define REMOTE_NOTIFY_CHAR_UUID    0xFF01
```

Un servicio se define utilizando la estructura `esp_gatt_srvc_id_t` de la siguiente manera:

```
/**
 * @brief Gatt id, include uuid and instance id
 */
typedef struct {
    esp_bt_uuid_t   uuid;                   /*!< UUID */
    uint8_t         inst_id;                /*!< ID de instancia */
} __attribute__((packed)) esp_gatt_id_t;
```

En este ejemplo, definimos el servicio del cual queremos obtener las características de la siguiente manera:

```
static esp_gatt_srvc_id_t remote_service_id = {
    .id = {
        .uuid = {
            .len = ESP_UUID_LEN_16,
            .uuid = {.uuid16 = REMOTE_SERVICE_UUID,},
        },
        .inst_id = 0,
    },
    .is_primary = true,
};
```

Una vez definido, podemos obtener las características de ese servicio utilizando la función `esp_ble_gattc_get_characteristic()`, la cual se llama en el evento `ESP_GATTC_SEARCH_CMPL_EVT` después de que se haya completado la búsqueda de servicios y el cliente ha encontrado el servicio que estaba buscando.

```
case ESP_GATTC_SEARCH_CMPL_EVT:
    if (p_data->search_cmpl.status != ESP_GATT_OK){
        ESP_LOGE(GATTC_TAG, "búsqueda de servicio fallida, estado de error = %x", p_data->search
        break;
    }
    conn_id = p_data->search_cmpl.conn_id;
    if (get_server){
        uint16_t count = 0;
        esp_gatt_status_t status = esp_ble_gattc_get_attr_count(gattc_if,
                           p_data->search_cmpl.conn_id, ESP_GATT_DB_CHARACTERISTIC,
                           gl_profile_tab[PROFILE_A_APP_ID].service_start_handle,
                           gl_profile_tab[PROFILE_A_APP_ID].service_end_handle,
                           INVALID_HANDLE,
                           &count);
        if (status != ESP_GATT_OK){
            ESP_LOGE(GATTC_TAG, "esp_ble_gattc_get_attr_count error");
        }

        if (count > 0){
            char_elem_result = (esp_gattc_char_elem_t*)malloc(sizeof(esp_gattc_char_elem_t) * co
            if (!char_elem_result){
                ESP_LOGE(GATTC_TAG, "gattc sin memoria");
            }else{
                status = esp_ble_gattc_get_char_by_uuid(gattc_if,
                                                        p_data->search_cmpl.conn_id,
                                                        gl_profile_tab[PROFILE_A_APP_ID].service_
                                                        gl_profile_tab[PROFILE_A_APP_ID].service_
                                                        remote_filter_char_uuid,
                                                        char_elem_result,
                                                        &count);
                if (status != ESP_GATT_OK){
                    ESP_LOGE(GATTC_TAG, "esp_ble_gattc_get_char_by_uuid error");
                }

                /* Cada servicio tiene solo una característica en nuestro demo 'ESP_GATTS_DEMO',
                por lo que usamos el primer 'char_elem_result' */
                if (count > 0 && (char_elem_result[0].properties
                                & ESP_GATT_CHAR_PROP_BIT_NOTIFY)){
                    gl_profile_tab[PROFILE_A_APP_ID].char_handle =
                    char_elem_result[0].char_handle;
                    esp_ble_gattc_register_for_notify(gattc_if,
                                    gl_profile_tab[PROFILE_A_APP_ID].remote_bda,
                                    char_elem_result[0].char_handle);
                }
            }
            /* Liberar char_elem_result */
            free(char_elem_result);
        }else{
            ESP_LOGE(GATTC_TAG, "ninguna característica encontrada");
        }
    }
    break;
```

`esp_ble_gattc_get_attr_count()` obtiene el número de atributos con el tipo de atributo dado en la caché GATT. Los parámetros de la función `esp_ble_gattc_get_attr_count()` son la interfaz GATT, el ID de conexión, el tipo de atributo definido en `esp_gatt_db_attr_type_t`, el valor de inicio del atributo, el

valor final del atributo, el identificador de la característica (este parámetro solo es válido cuando el tipo se establece en `ESP_GATT_DB_DESCRIPTOR`) y la salida del número de atributos encontrados en la caché GATT con el tipo de atributo dado.

Luego, se asigna un búfer para guardar la información de la característica para la función `esp_ble_gattc_get_char_by_uuid()`. Esta función busca la característica con el UUID de característica dado en la caché GATT. Simplemente obtiene la característica de la caché local, en lugar de los dispositivos remotos. En un servidor, puede haber más de una característica con el mismo UUID. Sin embargo, en nuestro ejemplo de gatt_server, cada característica tiene un UUID único y es por eso que solo usamos la primera característica en `char_elem_result`, que es el puntero a la característica del servicio. El contador inicialmente almacena la cantidad de características que el cliente desea encontrar y se actualizará con la cantidad de características que se han encontrado en la caché GATT con `esp_ble_gattc_get_char_by_uuid`.

Registro para Notificaciones

El cliente puede registrarse para recibir notificaciones del servidor cada vez que cambia el valor de la característica. En este ejemplo, queremos registrarnos para recibir notificaciones de la característica identificada con un UUID de 0xff01. Después de obtener todas las características, verificamos las propiedades de la característica recibida y luego utilizamos la función `esp_ble_gattc_register_for_notify()` para registrarnos para notificaciones. Los argumentos de la función son la interfaz GATT, la dirección del servidor remoto y el identificador del que queremos recibir notificaciones.

```
...
/* Cada servicio tiene solo una característica en nuestra demostración 'ESP_GATTS_DEMO', por lo
                    if (count > 0 && (char_elem_result[0].properties & ESP_GATT_CHAR_PROP_BIT_NO
                        gl_profile_tab[PROFILE_A_APP_ID].char_handle = char_elem_result[0].char_
                        esp_ble_gattc_register_for_notify(gattc_if, gl_profile_tab[PROFILE_A_APP
                        char_elem_result[0].char_handle);
                        }
...
```

Este procedimiento registra notificaciones en la pila BLE y desencadena un evento `ESP_GATTC_REG_FOR_NOTIFY_EVT`. Este evento se utiliza para escribir en el Descriptor de Configuración del Cliente del servidor:

```
    case ESP_GATTC_REG_FOR_NOTIFY_EVT: {
        ESP_LOGI(GATTC_TAG, "ESP_GATTC_REG_FOR_NOTIFY_EVT");
        if (p_data->reg_for_notify.status != ESP_GATT_OK){
            ESP_LOGE(GATTC_TAG, "REG FOR NOTIFY falló: estado de error = %d", p_data->reg_for_no
        }else{
            uint16_t count = 0;
            uint16_t notify_en = 1;
            esp_gatt_status_t ret_status = esp_ble_gattc_get_attr_count( gattc_if, gl_profile_ta
                                                    ESP_GATT_DB_DESCRIPTOR,
                                                    gl_profile_tab[PROFILE_A_APP_ID].service
                                                    gl_profile_tab[PROFILE_A_APP_ID].service
                                                    gl_profile_tab[PROFILE_A_APP_ID].char_ha
            if (ret_status != ESP_GATT_OK){
                ESP_LOGE(GATTC_TAG, "esp_ble_gattc_get_attr_count error");
            }
            if (count > 0){
                descr_elem_result = malloc(sizeof(esp_gattc_descr_elem_t) * count);
                if (!descr_elem_result){
                    ESP_LOGE(GATTC_TAG, "error de asignación de memoria, gattc sin memoria");
                }else{
                    ret_status = esp_ble_gattc_get_descr_by_char_handle(
                    gattc_if,
                    gl_profile_tab[PROFILE_A_APP_ID].conn_id,
                    p_data->reg_for_notify.handle,
                    notify_descr_uuid,
                    descr_elem_result,&count);

                    if (ret_status != ESP_GATT_OK){
                        ESP_LOGE(GATTC_TAG, "esp_ble_gattc_get_descr_by_char_handle
                                                                        error");
                    }

                    /* Cada característica tiene solo un descriptor en nuestra demostración 'ESP
                    if (count > 0 && descr_elem_result[0].uuid.len == ESP_UUID_LEN_16 && descr_e
                        ret_status = esp_ble_gattc_write_char_descr( gattc_if,
                                                    gl_profile_tab[PROFILE_A_APP_ID].conn_id
                                                    descr_elem_result[0].handle,
                                                    sizeof(notify_en),
                                                    (Uint8 *)&notify_en,
                                                    ESP_GATT_WRITE_TYPE_RSP,
                                                    ESP_GATT_AUTH_REQ_NONE);
                    }

                    if (ret_status != ESP_GATT_OK){
                        ESP_LOGE(GATTC_TAG, "esp_ble_gattc_write_char_descr error");
                    }

                    /* Liberar descr_elem_result */
                    free(descr_elem_result);
                }
            }
            else{
                ESP_LOGE(GATTC_TAG, "descripción no encontrada");
            }

        }
        break;
    }
```

El evento se utiliza primero para imprimir el estado del registro de notificaciones y los UUID del servicio y de la característica que acaban de registrarse. Luego, el cliente escribe en el Descriptor de Configuración del Cliente utilizando la función `esp_ble_gattc_write_char_descr()`. Hay muchos descriptores de características definidos en la especificación de Bluetooth. Sin embargo, en este caso, nos interesa escribir en el descriptor que se encarga de habilitar las notificaciones, que es el descriptor de Configuración del Cliente. Para pasar este descriptor como parámetro, primero lo definimos como:

```
static esp_gatt_id_t notify_descr_id = {
    .uuid = {
        .len = ESP_UUID_LEN_16,
        .uuid = {.uuid16 = ESP_GATT_UUID_CHAR_CLIENT_CONFIG,},
    },
    .inst_id = 0,
};
```

Donde `ESP_GATT_UUID_CHAR_CLIENT_CONFIG` se define con el UUID para identificar la Configuración del Cliente de la Característica:

```
#define ESP_GATT_UUID_CHAR_CLIENT_CONFIG            0x2902          /*  Configuración del Client
```

El valor a escribir es "1" para habilitar las notificaciones. También pasamos `ESP_GATT_WRITE_TYPE_RSP` para solicitar que el servidor responda a la solicitud de habilitar las notificaciones y `ESP_GATT_AUTH_REQ_NONE` para indicar que la solicitud de escritura no requiere autorización.

Tarea adicional

Modifica el cliente para que las notificaciones enviadas por tu servidor GATT modificado en la práctica anterior sean recibidas y monitorizadas por parte del cliente GATT.
