Gatt サーバ実装例のウォークスルー

はじめに

本ドキュメントでは、ESP32 上の Bluetooth Low Energy(BLE)一般属性プロファイル（GATT）を実装している GATT サーバの実装例を見直します。この実装例では 2 つアプリケーションプロファイルと一連の設定ステップを実行するために取り扱うイベントを設計しています。例えば広告パラメータ定義や接続パラメータの更新、サービスと属性の作成などです。さらにこの実装例ではイベントの読み書きを扱います。これには長い属性の書き込み要求が含まれ、受信データをチャンクに分けて属性プロトコルメッセージ（ATT）にサイズが合うようにします。本ドキュメントではプログラムのフローに続いて、実装の背後にある理由や各部分の意味が分かるようになるためにコードを読み解いていきます。

インクルード

はじめに、次のインクルードファイルを見てみましょう。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/event_groups.h"
#include "esp_system.h"
#include "esp_log.h"
#include "nvs_flash.h"
#include "bt.h"
#include "bta_api.h"
#include "esp_gap_ble_api.h"
#include "esp_gatts_api.h"
#include "esp_bt_defs.h"
#include "esp_bt_main.h"
#include "sdkconfig.h"
```

これらのインクルードファイルは FreeRTOS および配下のシステムコンポーネントが動作するのに必要なモノです。ログ機能や不揮発性フラッシュメモリへのデータ保存用のライブラリも含んでいます。ここでは”bt.h”,”esp_bt_main.h”, “esp_gap_ble_api.h”,”esp_gatts_api.h”に注意を引きます。これらは本サンプルにて実装するのに必要な BLE API を提供しています。

- bt.h: BT コントローラおよびホスト側からの VHCI 設定プロシージャを実装します。
- esp_bt_main.h: Bluedroid スタックを初期化し有効にする実装です。
- esp_gap_ble_api.h: GAP 設定たとえば広告と接続パラメータを実装します。
- esp_gatts_api.h: GATT 設定たとえばサービスや属性を実装します。

メインエントリポイント

本サンプルのエントリポイントは app_main()関数です。

```
void app_main()
{
    esp_err_t ret;

    // Initialize NVS.
    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(GATTS_TAG, "%s initialize controller failed\n", __func__);
        return;
    }

    ret = esp_bt_controller_enable(ESP_BT_MODE_BLE);
    if (ret) {
        ESP_LOGE(GATTS_TAG, "%s enable controller failed\n", __func__);
        return;
    }
    ret = esp_bluedroid_init();
    if (ret) {
        ESP_LOGE(GATTS_TAG, "%s init bluetooth failed\n", __func__);
        return;
    }
    ret = esp_bluedroid_enable();
    if (ret) {
        ESP_LOGE(GATTS_TAG, "%s enable bluetooth failed\n", __func__);
        return;
    }

    ret = esp_ble_gatts_register_callback(gatts_event_handler);
    if (ret){
        ESP_LOGE(GATTS_TAG, "gatts register error, error code = %x", ret);
        return;
    }
    ret = esp_ble_gap_register_callback(gap_event_handler);
    if (ret){
        ESP_LOGE(GATTS_TAG, "gap register error, error code = %x", ret);
        return;
    }
    ret = esp_ble_gatts_app_register(PROFILE_A_APP_ID);
    if (ret){
        ESP_LOGE(GATTS_TAG, "gatts app register error, error code = %x", ret);
        return;
    }
    ret = esp_ble_gatts_app_register(PROFILE_B_APP_ID);
    if (ret){
        ESP_LOGE(GATTS_TAG, "gatts app register error, error code = %x", ret);
        return;
```

```
    }
    esp_err_t local_mtu_ret = esp_ble_gatt_set_local_mtu(512);
    if (local_mtu_ret){
        ESP_LOGE(GATTS_TAG, "set local  MTU failed, error code = %x",
local_mtu_ret);
    }
    return;
}
```

メイン関数は不揮発性メモリのライブラリを初期化することから開始します。このライブラリはフラッシュメモリ内にキーバリューペアを保存し Wi-Fi ライブラリなどのコンポーネントから利用して SSID やパスワードを保存できるようにします。

```
ret = nvs_flash_init();
```

BT コントローラとスタック初期化

メイン関数では BT コントローラ設定構造体 esp_bt_controller_config_t をまず作成します。BT_CONTROLLER_INIT_CONFIG_DEFAULT()マクロを使ってデフォルト設定を生成します。BT コントローラはコントローラ側のホストコントローラインタフェース（HCI）とリンクレイヤ（LL）および物理レイヤ（PHY）を実装します。BT コントローラはユーザアプリケーションからは見えず、BLE スタックの低レイヤを扱います。BT コントローラの設定には BT コントローラのスタックサイズや優先度、HCI ボーレートの設定が含まれます。この設定を生成することで、BT コントローラは初期化されて esp_bt_controller_init()関数を使って有効化できます。

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

次に、BT コントローラは BLE モードを有効にします。

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

BT コントローラはデュアルモード（BLE+BT）を使いたい場合には ESP_BT_MODE_BTDM を有効にする必要があります。Bluetooth モードを 4 つサポートしています。

1. ESP_BT_MODE_IDLE: Bluetooth を起動しない
2. ESP_BT_MODE_BLE: BLE モード
3. ESP_BT_MODE_CLASSIC_BT: BT クラシックモード
4. ESP_BT_MODE_BTDM: デュアルモード（BLE+BT クラシック）

BT コントローラを初期化したあとに Bluedroid スタックは BT クラシックと BLE 両方に共通する定義や API を含んでいますが、これを初期化し次のように有効化します。

```
ret = esp_bluedroid_init();
ret = esp_bluedroid_enable();
```

Bluetooth スタックが立ち上がり、この時点でプログラム内にて起動します。しかしアプリケーションの機能をまだ定義していません。この機能は他のデバイスがパラメータを読み書きしようとしたり接続を確立させようとしたりといったイベントに反応する形で定義されます。イベントを管理する 2 つの主なものは GAP と GATT イベ

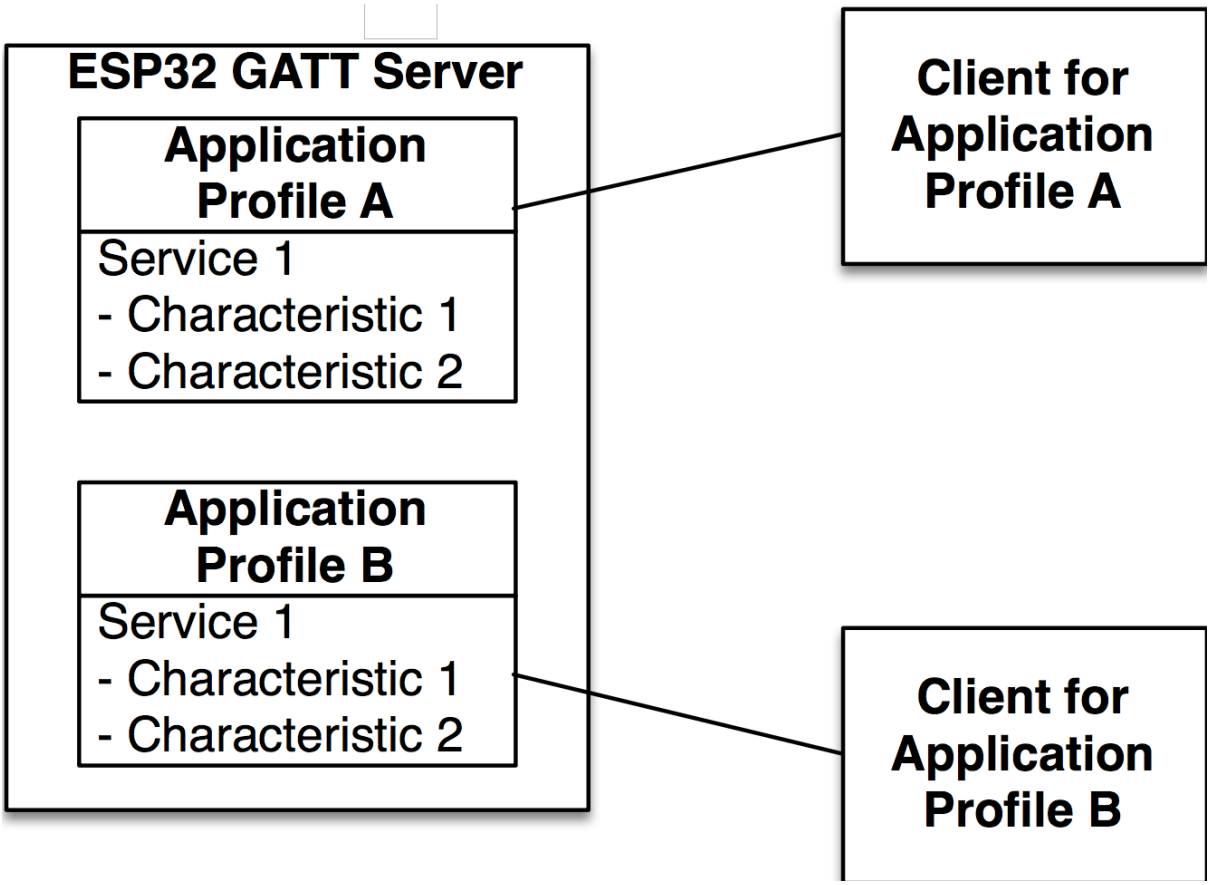
ントハンドラです。アプリケーションではそれぞれのイベントハンドラ用のコールバック関数を登録する必要があり、GAP や GATT イベントを扱う関数をアプリケーションに教えてあげる必要があります。

```
esp_ble_gatts_register_callback(gatts_event_handler);
esp_ble_gap_register_callback(gap_event_handler);
```

gatts_event_handler()および gap_event_handler()関数は BLE スタックからアプリケーションに送られてくるイベントをすべて扱います。

アプリケーションプロファイル

GATT サーバのサンプルでは下図に示すようにアプリケーションプロファイルを利用して構成されています。各アプリケーションプロファイルは、スマートフォンやタブレット上で動作するアプリケーションのようなクライアントアプリケーション用に設計された機能をグループ分けする方法を表現します。この方法によって、異なるアプリケーションプロファイルで有効化された設計については、異なるスマートフォンアプリから利用されるとそれぞれ違った振る舞いができるようになります。利用されているクライアントアプリケーションによってサーバ側の振る舞いを変えることができるからです。実際には、プロファイルはクライアントからはそれぞれ独立した BLE サービスとして見えています。興味のあるサービスを区別するのはクライアント次第です。



各プロファイルは 1 個の構造体として定義されます。ここで構造体のメンバはアプリケーションプロファイルで実装されるサービスやキャラクタリスティックによって変わります。メンバには GATT インタフェースやアプリケーション

ID、接続 ID そしてプロファイルのイベントを扱うコールバック関数も含まれます。本サンプルでは次のものが定義されます。

- GATT インタフェース
- アプリケーション ID
- 接続 ID
- サービスハンドル
- サービス ID
- キャラクタリスティックハンドル
- キャラクタリスティック UUID
- 属性のアクセス許可
- キャラクタリスティックのプロパティ
- クライアントのキャラクタリスティック設定ディスクリプタハンドル
- クライアントのキャラクタリスティック設定ディスクリプタの UUID

この構造体から観測できることは、このプロファイルはサービス 1 個、キャラクタリスティック 1 個を持つように設計されており、1 つのキャラクタリスティックは 1 つのディスクリプタを持っているということです。サービスは 1 個のハンドルと ID を持っており、同様に各属性は 1 個のハンドルと UUID、属性のアクセス許可とプロパティを持っています。さらにキャラクタリスティックが通知や表示をサポートしていればクライアントのキャラクタリスティック設定ディスクリプタ（CCCD）を実装する必要があります。これは通知や表示が有効化されていて特定のクライアントからそのキャラクタリスティックがどのように設定可能になっているかを定義しています。このディスクリプタにも 1 個のハンドルと UUID があります。

構造体の実装は次のようになっています。

```
struct gatts_profile_inst {
    esp_gatts_cb_t gatts_cb;
    uint16_t gatts_if;
    uint16_t app_id;
    uint16_t conn_id;
    uint16_t service_handle;
    esp_gatt_srvc_id_t service_id;
    uint16_t char_handle;
    esp_bt_uuid_t char_uuid;
    esp_gatt_perm_t perm;
    esp_gatt_char_prop_t property;
    uint16_t descr_handle;
    esp_bt_uuid_t descr_uuid;
};
```

アプリケーションプロファイルは 1 個の配列内で保持されており、対応するコールバック関数 gatts_profile_a_event_hander()と gatts_profile_b_event_handler()が割り当てられています。GATT クライアント上の異なるアプリケーションは異なるインタフェースを利用します。これは gatts_if パラメー

タで示されます。初期化するときには、このパラメータは ESP_GATT_IF_NONE に設定されています。これはまだアプリケーションプロファイルがクライアントとリンクしていないということを意味しています。

```
static struct gatts_profile_inst gl_profile_tab[PROFILE_NUM] = {
    [PROFILE_A_APP_ID] = {
        .gatts_cb = gatts_profile_a_event_handler,
        .gatts_if = ESP_GATT_IF_NONE,
    [PROFILE_B_APP_ID] = {
        .gatts_cb = gatts_profile_b_event_handler,
        .gatts_if = ESP_GATT_IF_NONE,
    },
};
```

最後に、アプリケーション ID を用いてアプリケーションプロファイルを登録します。ID はユーザが割り当てた番号で、各プロファイルを特定するものです。この方法によって複数のアプリケーションプロファイルを 1 つのサーバで動作させることができます。

```
esp_ble_gatts_app_register(PROFILE_A_APP_ID);
esp_ble_gatts_app_register(PROFILE_B_APP_ID);
```

GAP パラメータの設定

アプリケーション登録イベントがプログラムの実行中に発生する最初のイベントです。このサンプルではプロファイル A の GATT イベントハンドルを使って、登録時に広告用パラメータを設定します。このサンプルでは標準 Bluetooth コア規格の広告パラメータ、もしくはカスタマイズした生データのいずれかを用いるオプションがあります。オプションは CONFIG_SET_RAW_ADV_DATA 定義で選択可能です。生の広告データを用いると、iBeacon や Eddystone などの他の商用の危機を実装したり、標準規格と異なった屋内位置サービスに使用するようなカスタムのフレームタイプを利用したりできます。

Bluetooth 規格の広告パラメータを設定するのに利用する関数は esp_ble_gap_config_adv_data()です。この関数は esp_ble_adv_data_t 構造体へのポインタを取ります。広告データ用の esp_ble_adv_data_t データ構造は次の定義になっています。

```
typedef struct {
    bool set_scan_rsp;            /*!< Set this advertising data as scan response
or not*/
    bool include_name;            /*!< Advertising data include device name or not
*/
    bool include_txpower;         /*!< Advertising data include TX power */
    int min_interval;             /*!< Advertising data show slave preferred
connection min interval */
    int max_interval;             /*!< Advertising data show slave preferred
connection max interval */
    int appearance;               /*!< External appearance of device */
    uint16_t manufacturer_len;    /*!< Manufacturer data length */
    uint8_t *p_manufacturer_data; /*!< Manufacturer data point */
    uint16_t service_data_len;    /*!< Service data length */
    uint8_t *p_service_data;      /*!< Service data point */
    uint16_t service_uuid_len;    /*!< Service uuid length */
    uint8_t *p_service_uuid;      /*!< Service uuid array point */
```

```
    uint8_t flag;                   /*!< Advertising flag of discovery mode, see
BLE_ADV_DATA_FLAG detail */
} esp_ble_adv_data_t;
```

本サンプルでは、esp_ble_adv_data_t 構造体は次のように初期化されます。

```
static esp_ble_adv_data_t adv_data = {
    .set_scan_rsp = false,
    .include_name = true,
    .include_txpower = true,
    .min_interval = 0x0006,
    .max_interval = 0x0010,
    .appearance = 0x00,
    .manufacturer_len = 0, //TEST_MANUFACTURER_DATA_LEN,
    .p_manufacturer_data =  NULL, //&test_manufacturer[0],
    .service_data_len = 0,
    .p_service_data = NULL,
    .service_uuid_len = 32,
    .p_service_uuid = test_service_uuid128,
    .flag = (ESP_BLE_ADV_FLAG_GEN_DISC | ESP_BLE_ADV_FLAG_BREDR_NOT_SPT),
};
```

スレーブの最小・最大接続間隔は 1.25ms 倍で設定されます。本サンプルでは、スレーブの奨励最小接続間隔は 0x0006 * 1.25 ms = 7.5 ミリ秒です。奨励最大接続間隔は 0.0010 * 1.25ms = 20 ミリ秒です。

広告データのペイロードは 31 バイトまで作れます。パラメータデータは広告パケットの上限値である 31 バイトを超える量を保持することはできますが、スタックが広告パケットを切断してパラメータのいくつかを送信しないようにしてしまいます。本サンプルでは、manufacturer_len と data をコメントアウトしてみると再コンパイルとテストするとサービスが広告されなくなることが分かります。

esp_ble_gap_config_adv_data_raw()と esp_ble_gap_config_scan_rsp_data_raw()関数を用いてカスタマイズした生データを広告することもできます。この関数では広告データおよびスキャンに対する応答データに関するバッファを作成して渡す必要があります。本サンプルでは、raw_adv_data[]と raw_scan_rsp_data[]配列で生データを表現しています。

最後に、デバイス名を設定するには esp_ble_gap_set_device_name()関数を使用します。イベントハンドラは次のように示されます。

```
static void gatts_profile_a_event_handler(esp_gatts_cb_event_t event,
esp_gatt_if_t gatts_if, esp_ble_gatts_cb_param_t *param) {
    switch (event) {
    case ESP_GATTS_REG_EVT:
         ESP_LOGI(GATTS_TAG, "REGISTER_APP_EVT, status %d, app_id %d\n", param-
>reg.status, param->reg.app_id);
         gl_profile_tab[PROFILE_A_APP_ID].service_id.is_primary = true;
         gl_profile_tab[PROFILE_A_APP_ID].service_id.id.inst_id = 0x00;
         gl_profile_tab[PROFILE_A_APP_ID].service_id.id.uuid.len =
ESP_UUID_LEN_16;
         gl_profile_tab[PROFILE_A_APP_ID].service_id.id.uuid.uuid.uuid16 =
GATTS_SERVICE_UUID_TEST_A;

         esp_ble_gap_set_device_name(TEST_DEVICE_NAME);
```

```
#ifdef CONFIG_SET_RAW_ADV_DATA
        esp_err_t raw_adv_ret = esp_ble_gap_config_adv_data_raw(raw_adv_data,
sizeof(raw_adv_data));
        if (raw_adv_ret){
            ESP_LOGE(GATTS_TAG, "config raw adv data failed, error code = %x ",
raw_adv_ret);
        }
        adv_config_done |= adv_config_flag;
        esp_err_t raw_scan_ret =
esp_ble_gap_config_scan_rsp_data_raw(raw_scan_rsp_data,
sizeof(raw_scan_rsp_data));
        if (raw_scan_ret){
            ESP_LOGE(GATTS_TAG, "config raw scan rsp data failed, error code
= %x", raw_scan_ret);
        }
        adv_config_done |= scan_rsp_config_flag;
#else
        //config adv data
        esp_err_t ret = esp_ble_gap_config_adv_data(&adv_data);
        if (ret){
            ESP_LOGE(GATTS_TAG, "config adv data failed, error code = %x", ret);
        }
        adv_config_done |= adv_config_flag;
        //config scan response data
        ret = esp_ble_gap_config_adv_data(&scan_rsp_data);
        if (ret){
            ESP_LOGE(GATTS_TAG, "config scan response data failed, error code
= %x", ret);
        }
        adv_config_done |= scan_rsp_config_flag;
#endif
```

GAP イベントハンドラ

広告パケットデータが設定されると、GAP イベント ESP_GAP_BLE_ADV_DATA_SET_COMPLETE_EVT が設定されます。生の広告データを設定した場合は ESP_GAP_BLE_ADV_DATA_RAW_SET_COMPLETE_EVT イベントが設定されます。さらに生のスキャン応答データが設定された場合は ESP_GAP_BLE_SCAN_RSP_DATA_RAW_SET_COMPLETE_EVT イベントが設定されます。

```
static void gap_event_handler(esp_gap_ble_cb_event_t event, esp_ble_gap_cb_param_t
*param)
{
    switch (event) {
#ifdef CONFIG_SET_RAW_ADV_DATA
    case ESP_GAP_BLE_ADV_DATA_RAW_SET_COMPLETE_EVT:
         adv_config_done &= (~adv_config_flag);
         if (adv_config_done==0){
             esp_ble_gap_start_advertising(&adv_params);
         }
         break;
    case ESP_GAP_BLE_SCAN_RSP_DATA_RAW_SET_COMPLETE_EVT:
         adv_config_done &= (~scan_rsp_config_flag);
         if (adv_config_done==0){
             esp_ble_gap_start_advertising(&adv_params);
```

```
        }
        break;
#else
    case ESP_GAP_BLE_ADV_DATA_SET_COMPLETE_EVT:
        adv_config_done &= (~adv_config_flag);
        if (adv_config_done == 0){
            esp_ble_gap_start_advertising(&adv_params);
        }
        break;
    case ESP_GAP_BLE_SCAN_RSP_DATA_SET_COMPLETE_EVT:
        adv_config_done &= (~scan_rsp_config_flag);
        if (adv_config_done == 0){
            esp_ble_gap_start_advertising(&adv_params);
        }
        break;
#endif
…
```

いずれの場合でも esp_ble_gap_start_advertising()関数を使ってサーバは広告パケットを送信開始します。この関数は、スタックが動作するのに必須である広告パラメータを付けた esp_ble_adv_params_t 構造体を取ります。

```
// Advertising parameters
typedef struct {
    uint16_t adv_int_min;
    /*!< Minimum advertising interval for undirected and low duty cycle directed
advertising.
                                        Range: 0x0020 to 0x4000
                                        Default: N = 0x0800 (1.28 second)
                                        Time = N * 0.625 msec
                                        Time Range: 20 ms to 10.24 sec */
    uint16_t adv_int_max;
    /*!< Maximum advertising interval for undirected and low duty cycle directed
advertising.
                                        Range: 0x0020 to 0x4000
                                        Default: N = 0x0800 (1.28 second)
                                        Time = N * 0.625 msec
                                        Time Range: 20 ms to 10.24 sec */
    esp_ble_adv_type_t adv_type;           /*!< Advertising type */
    esp_ble_addr_type_t own_addr_type;     /*!< Owner bluetooth device address
type */
    esp_bd_addr_t peer_addr;               /*!< Peer device bluetooth device
address */
    esp_ble_addr_type_t peer_addr_type;    /*!< Peer device bluetooth device
address type */
    esp_ble_adv_channel_t channel_map;     /*!< Advertising channel map */
    esp_ble_adv_filter_t adv_filter_policy; /*!< Advertising filter policy */
}
esp_ble_adv_params_t;
```

esp_ble_gap_config_adv_data()関数はクライアントに広告されるデータを設定し、esp_ble_adv_data_t 構造体を引数に取るのに対し、esp_ble_gap_start_advertising()関数はサーバが実際に広告パケットを送信開始するのに使用される関数で esp_ble_adv_params_t 構造体を取ると言うことに注意してください。広告データはクライアントに見せる情報ですが、広告用パラメータは GAP が動作するのに必要な設定です。

このサンプルでは、広告パラメータは次のように初期化されます。

```
static esp_ble_adv_params_t test_adv_params = {
    .adv_int_min        = 0x20,
    .adv_int_max        = 0x40,
    .adv_type           = ADV_TYPE_IND,
    .own_addr_type      = BLE_ADDR_TYPE_PUBLIC,
    //.peer_addr        =
    //.peer_addr_type   =
    .channel_map        = ADV_CHNL_ALL,
    .adv_filter_policy  = ADV_FILTER_ALLOW_SCAN_ANY_CON_ANY,
};
```

これらのパラメータは広告間隔を 40ms から 80ms の間に設定します。本広告は ADV_IND であり一般的なもので、特定の中心デバイスや接続可能な種類のものに向けたものではありません。アドレス種類はパブリックで、全チャネルを利用し、任意の中心デバイスからのスキャンおよび接続要求を許します。

広告パケット送信が成功した場合、ESP_GAP_BLE_ADV_START_COMPLETE_EVT イベントが生成されます。このイベント処理において、本サンプルでは広告ステータスが本当に広告パケット送信になったかをチェックしています。そうでなければエラーメッセージを表示します。

```
…
    case ESP_GAP_BLE_ADV_START_COMPLETE_EVT:
    //advertising start complete event to indicate advertising start successfully
or failed
         if (param->adv_start_cmpl.status != ESP_BT_STATUS_SUCCESS) {
             ESP_LOGE(GATTS_TAG, "Advertising start failed\n");
         }
         break;
…
```

GATT イベントハンドラ

アプリケーションプロファイルを登録すると、ESP_GATTS_REG_EVT イベントが発送されます。

ESP_GATTS_REG_EVT のパラメータは次の通りです。

```
esp_gatt_status_t status;  /* Operation status */
uint16_t app_id;           /* Application id which input in register API */
```

先のパラメータに加えて、このイベントでは BLE スタックが割り当てた GATT インタフェースも含んでいます。このイベントは gatts_event_handler()で捕捉されます。これはプロファイルテーブル上で生成されたインタフェースを格納するのに使われます。そして対応したプロファイルのイベントハンドラに本イベントを中継します。

```
static void gatts_event_handler(esp_gatts_cb_event_t event, esp_gatt_if_t
gatts_if, esp_ble_gatts_cb_param_t *param)
{
    /* If event is register event, store the gatts_if for each profile */
    if (event == ESP_GATTS_REG_EVT) {
        if (param->reg.status == ESP_GATT_OK) {
            gl_profile_tab[param->reg.app_id].gatts_if = gatts_if;
        } else {
            ESP_LOGI(GATTS_TAG, "Reg app failed, app_id %04x, status %d\n",
                    param->reg.app_id,
                    param->reg.status);
```

```
            return;
        }
    }

/* If the gatts_if equal to profile A, call profile A cb handler,
 * so here call each profile's callback */
    do {
        int idx;
        for (idx = 0; idx < PROFILE_NUM; idx++) {
            if (gatts_if == ESP_GATT_IF_NONE||gatts_if ==
gl_profile_tab[idx].gatts_if)                    {
                if (gl_profile_tab[idx].gatts_cb) {
                    gl_profile_tab[idx].gatts_cb(event, gatts_if, param);
                }
            }
        }
    } while (0);
}
```

サービスの生成

この登録イベントは esp_ble_gatts_create_attr_tab()関数を用いてプロファイル属性を生成するのにも使われます。サービステーブルの情報をすべてもった esp_gatts_attr_db_t 型の構造体を取ります。このテーブルを作成する方法は次の通りです。

```
…
esp_ble_gatts_create_service(gatts_if,
&gl_profile_tab[PROFILE_A_APP_ID].service_id, GATTS_NUM_HANDLE_TEST_A);
break;
…
```

ハンドルの数は 4 に定義されています。

```
#define GATTS_NUM_HANDLE_TEST_A     4
```

ハンドルは

1. サービスハンドル
2. 属性ハンドル
3. 属性値ハンドル
4. 属性値の標準テーブル

です。サービスは、UUID16 ビット長の主サービスとして定義されます。サービス ID はインスタンス ID=0 で初期化され UUID は GATTS_SERVICE_UUID_TEST_A に定義されます。

サービスインスタンス ID は同一の UUID の複数サービスを区別するために用いることができます。本サンプルではアプリケーションプロファイルが 1 つのサービスでしかなく、サービスも異なる UUID を与えているため、サービスインスタンス ID はプロファイル A でも B でも 0 に定義して良いことになります。しかし、1 つのアプリケーションプロファイルに 2 つ同一の UUID を用いたサービスがあった場合、2 つのサービスを表すために異なるインスタンス ID を使用する必要があるでしょう。

アプリケーションプロファイル B は、プロファイル A と同じ方法でサービスを作成します。

```
static void gatts_profile_b_event_handler(esp_gatts_cb_event_t event,
esp_gatt_if_t gatts_if, esp_ble_gatts_cb_param_t *param) {
    switch (event) {
    case ESP_GATTS_REG_EVT:
         ESP_LOGI(GATTS_TAG, "REGISTER_APP_EVT, status %d, app_id %d\n", param-
>reg.status, param->reg.app_id);
         gl_profile_tab[PROFILE_B_APP_ID].service_id.is_primary = true;
         gl_profile_tab[PROFILE_B_APP_ID].service_id.id.inst_id = 0x00;
         gl_profile_tab[PROFILE_B_APP_ID].service_id.id.uuid.len =
ESP_UUID_LEN_16;
         gl_profile_tab[PROFILE_B_APP_ID].service_id.id.uuid.uuid.uuid16 =
GATTS_SERVICE_UUID_TEST_B;

         esp_ble_gatts_create_service(gatts_if,
&gl_profile_tab[PROFILE_B_APP_ID].service_id, GATTS_NUM_HANDLE_TEST_B);
         break;
…
}
```

サービスの開始と属性の作成

サービスが正常に作成されるとプロファイルの GATT ハンドラが管理している ESP_GATTS_CREATE_EVT イベントが発生します。そしてサービスを開始してサービスに属性を追加するのにイベントを利用することが可能です。プロファイル A の場合、サービスを起動して属性を次のように追加します。

```
case ESP_GATTS_CREATE_EVT:
     ESP_LOGI(GATTS_TAG, "CREATE_SERVICE_EVT, status %d, service_handle %d\n",
param->create.status, param->create.service_handle);
     gl_profile_tab[PROFILE_A_APP_ID].service_handle = param-
>create.service_handle;
     gl_profile_tab[PROFILE_A_APP_ID].char_uuid.len = ESP_UUID_LEN_16;
     gl_profile_tab[PROFILE_A_APP_ID].char_uuid.uuid.uuid16 =
GATTS_CHAR_UUID_TEST_A;

     esp_ble_gatts_start_service(gl_profile_tab[PROFILE_A_APP_ID].service_handle);
     a_property = ESP_GATT_CHAR_PROP_BIT_READ | ESP_GATT_CHAR_PROP_BIT_WRITE |
ESP_GATT_CHAR_PROP_BIT_NOTIFY;
     esp_err_t add_char_ret =
     esp_ble_gatts_add_char(gl_profile_tab[PROFILE_A_APP_ID].service_handle,
                            &gl_profile_tab[PROFILE_A_APP_ID].char_uuid,
                            ESP_GATT_PERM_READ | ESP_GATT_PERM_WRITE,
                            a_property,
                            &gatts_demo_char1_val,
                            NULL);
    if (add_char_ret){
        ESP_LOGE(GATTS_TAG, "add char failed, error code =%x",add_char_ret);
    }
    break;
```

はじめに、BLE スタックが生成するサービスハンドルをプロファイルテーブルに格納します。これはアプリケーションから本サービスを表すのに使われます。属性の UUID と UUID 長を設定します。属性 UUID 長は再度 16 ビットです。サービスは esp_ble_gatts_start_service()関数に、先に作成されたサービスハンドルを与えて開始します。ESP_GATTS_START_EVT イベントは情報を出力するために使われますが、このイベントが発

生します。esp_ble_gatts_start_service()関数を属性のパーミッションとプロパティを漬けて呼び出して属性を追加します。本サンプルでは、A、B 両方のプロファイルの属性は次のように設定されます。

パーミッション：

- ESP_GATT_PERM_READ: 属性値を読み込むことを許可
- ESP_GATT_PERM_WRITE：属性値を書き込むことを許可

プロパティ：

- ESP_GATT_CHAR_PROP_BIT_READ: 属性は読み込める
- ESP_GATT_CHAR_PROP_BIT_WRITE: 属性は書き込める
- ESP_GATT_CHAR_BIT_NOTIFY: 属性は値が変化したことを通知できる

パーミッションとプロパティ両方で読み書きに関する情報を持つのは冗長と思われるかもしれませんが、属性の読み書きプロパティは、サーバが読み書きの要求を受け取るかどうかをクライアントに伝えるための情報なのです。この点で、プロパティはクライアントに対して適切にサーバリソースへアクセスするヒントとして提供されています。一方で、パーミッションは属性を読むか書くかするためにクライアントに与えられる権限です。例えば書き込み許可を持っていない属性にクライアントが書き込もうとする場合には、サーバはその要求を拒否します。たとえ書き込みプロパティが設定されている属性であってもです。

さらに、本サンプルでは gatts_demo_char1_val で表現されるキャラクタリスティックに初期値を与えています。この初期値は次のように定義されています。

```
esp_attr_value_t gatts_demo_char1_val =
{
    .attr_max_len = GATTS_DEMO_CHAR_VAL_LEN_MAX,
    .attr_len     = sizeof(char1_str),
    .attr_value   = char1_str,
};
```

ここで、char1_str はダミーデータです。

```
uint8_t char1_str[] = {0x11,0x22,0x33};
```

キャラクタリスティックの長さは次のように定義されています。

```
#define GATTS_DEMO_CHAR_VAL_LEN_MAX 0x40
```

キャラクタリスティックの初期値は null ではないオブジェクトである必要があり、長さは常に 0 より大きな値である必要があります。そうでなければ Bluetooth スタックがエラーを返します。

最後にキャラクタリスティックは、それが読み書きされたときにはその都度マニュアルで返事を送る必要があるように設定されます。自動で応答するようにはなっていません。これは esp_ble_gatts_add_char()関数の最後のパラメータで設定されています。これは属性の応答制御パラメータで ESP_GATT_RSP_BY_APP あるいは NULL に設定されています。

キャラクタリスティック・ディスクリプタの作成

サービスにキャラクタリスティックを追加すると ESP_GATTS_ADD_CHAR_EVT イベントが発生します。このイベントは今追加されたキャラクタリスティックに対してスタックが生成したハンドルを返します。本インベントは次のパラメータを含んでいます。

```
esp_gatt_status_t status;          /*!< Operation status */
uint16_t attr_handle;               /*!< Characteristic attribute handle */
uint16_t service_handle;           /*!< Service attribute handle */
esp_bt_uuid_t char_uuid;           /*!< Characteristic uuid */
```

本イベントによって返される属性ハンドルはプロファイルテーブルに保存されます。性質ディスクリプタと UUID についても同様に設定されます。キャラクタリスティックの長さは esp_ble_gatts_get_attr_value()関数で読み込まれ、情報表示の目的で表示されます。最後に esp_ble_gatts_add_char_descr()関数を用いてキャラクタリスティック・ディスクリプションを追加します。使用されるパラメータはサービスハンドルとディスクリプタの UUID、読み書きのパーミッション、初期値、自動応答設定です。キャラクタリスティック・ディスクリプタの初期値は NULL ポインタであっても良く、自動応答設定パラメータも同様に NULL に設定されます。これは応答が必要な要求にはマニュアルで返すという意味です。

```
case ESP_GATTS_ADD_CHAR_EVT: {
         uint16_t length = 0;
         const uint8_t *prf_char;

         ESP_LOGI(GATTS_TAG, "ADD_CHAR_EVT, status %d,  attr_handle %d,
service_handle %d\n",
                 param->add_char.status, param->add_char.attr_handle, param-
>add_char.service_handle);
                 gl_profile_tab[PROFILE_A_APP_ID].char_handle = param-
>add_char.attr_handle;
                 gl_profile_tab[PROFILE_A_APP_ID].descr_uuid.len =
ESP_UUID_LEN_16;
                 gl_profile_tab[PROFILE_A_APP_ID].descr_uuid.uuid.uuid16 =
ESP_GATT_UUID_CHAR_CLIENT_CONFIG;
                 esp_err_t get_attr_ret = esp_ble_gatts_get_attr_value(param-
>add_char.attr_handle, &length, &prf_char);
         if (get_attr_ret == ESP_FAIL){
                   ESP_LOGE(GATTS_TAG, "ILLEGAL HANDLE");
         }
         ESP_LOGI(GATTS_TAG, "the gatts demo char length = %x\n", length);
         for(int i = 0; i < length; i++){
             ESP_LOGI(GATTS_TAG, "prf_char[%x] = %x\n",i,prf_char[i]);
         }
         esp_err_t add_descr_ret = esp_ble_gatts_add_char_descr(
                                   gl_profile_tab[PROFILE_A_APP_ID].service_handle,
                                   &gl_profile_tab[PROFILE_A_APP_ID].descr_uuid,
                                   ESP_GATT_PERM_READ | ESP_GATT_PERM_WRITE,
                                   NULL,NULL);
         if (add_descr_ret){
            ESP_LOGE(GATTS_TAG, "add char descr failed, error code = %x",
add_descr_ret);
         }
         break;
    }
```

プロファイル B のイベントハンドラでこの手続きが繰り返されます。このプロファイルに対してサービスとキャラクタリスティックを作るためです。

接続イベント

クライアントが GATT サーバに接続したとき ESP_GATTS_CONNECT_EVT イベントが通知されます。このイベントは接続パラメータの更新に使われます。例えば遅延時間、接続までの開始間隔、最大接続間隔そしてタイムアウトです。接続パラメータは esp_ble_conn_update_params_t 構造であり、esp_ble_gap_update_conn_params()関数に渡されます。接続パラメータの更新プロシージャは一度だけ行う必要があります。従ってプロファイル B の接続イベントハンドラは esp_ble_gap_update_conn_parame()関数は含まれていません。最後に本イベントで返される接続 ID がプロファイルテーブルに保存されます。

プロファイル A の接続イベントは次の通りです。

```
case ESP_GATTS_CONNECT_EVT: {
     esp_ble_conn_update_params_t conn_params = {0};
     memcpy(conn_params.bda, param->connect.remote_bda, sizeof(esp_bd_addr_t));
     /* For the IOS system, please reference the apple official documents about
the ble connection parameters restrictions. */
     conn_params.latency = 0;
     conn_params.max_int = 0x30;    // max_int = 0x30*1.25ms = 40ms
     conn_params.min_int = 0x10;    // min_int = 0x10*1.25ms = 20ms
     conn_params.timeout = 400;     // timeout = 400*10ms = 4000ms
     ESP_LOGI(GATTS_TAG, "ESP_GATTS_CONNECT_EVT, conn_id %d,
remote %02x:%02x:%02x:%02x:%02x:%02x:, is_conn %d",
              param->connect.conn_id,
              param->connect.remote_bda[0],
              param->connect.remote_bda[1],
              param->connect.remote_bda[2],
              param->connect.remote_bda[3],
              param->connect.remote_bda[4],
              param->connect.remote_bda[5],
              param->connect.is_connected);
          gl_profile_tab[PROFILE_A_APP_ID].conn_id = param->connect.conn_id;
          //start sent the update connection parameters to the peer device.
          esp_ble_gap_update_conn_params(&conn_params);
          break;
```

コネクション B の接続イベントは次の通りです。

```
case ESP_GATTS_CONNECT_EVT:
     ESP_LOGI(GATTS_TAG, "CONNECT_EVT, conn_id %d,
remote %02x:%02x:%02x:%02x:%02x:%02x:, is_conn %d\n",
              param->connect.conn_id,
              param->connect.remote_bda[0],
              param->connect.remote_bda[1],
              param->connect.remote_bda[2],
              param->connect.remote_bda[3],
              param->connect.remote_bda[4],
              param->connect.remote_bda[5],
              param->connect.is_connected);
```

```
            gl_profile_tab[PROFILE_B_APP_ID].conn_id = param->connect.conn_id;
            break;
```

esp_ble_gap_update_conn_params()関数は GAP イベント ESP_GAP_BLE_UPDATE_CONN_PARAMS_EVT を発生させます。これは接続情報を表示するのに使います。

```
    case ESP_GAP_BLE_UPDATE_CONN_PARAMS_EVT:
         ESP_LOGI(GATTS_TAG, "update connection params status = %d, min_int = %d, max_int = %d,
                  conn_int = %d,latency = %d, timeout = %d",
                  param->update_conn_params.status,
                  param->update_conn_params.min_int,
                  param->update_conn_params.max_int,
                  param->update_conn_params.conn_int,
                  param->update_conn_params.latency,
                  param->update_conn_params.timeout);
         break;
```

読み込みイベントの処理

ここまででサービスとキャラクタリスティックを生成し、開始しましたので、プログラムは読み込み・書き込みイベントを受け取ることができます。読み込みイベントは ESP_GATTS_READ_EVT イベントで表されており、これは次のパラメータを取ります。

```
uint16_t conn_id;          /*!< Connection id */
uint32_t trans_id;         /*!< Transfer id */
esp_bd_addr_t bda;         /*!< The bluetooth device address which been read */
uint16_t handle;           /*!< The attribute handle */
uint16_t offset;           /*!< Offset of the value, if the value is too long */
bool is_long;              /*!< The value is too long or not */
bool need_rsp;             /*!< The read operation need to do response */
```

この例では、応答はダミーデータで作成されて、イベントで与えられた同じハンドルを用いてホストに返します。応答に加えて、GATT インタフェース、接続 ID、転送 ID が esp_ble_gatts_send_response()関数内のパラメータとして含まれています。この関数はキャラクタリスティックあるいはディスクリプタを作成した際に自動応答バイトが NULL に設定されている場合には必要です。

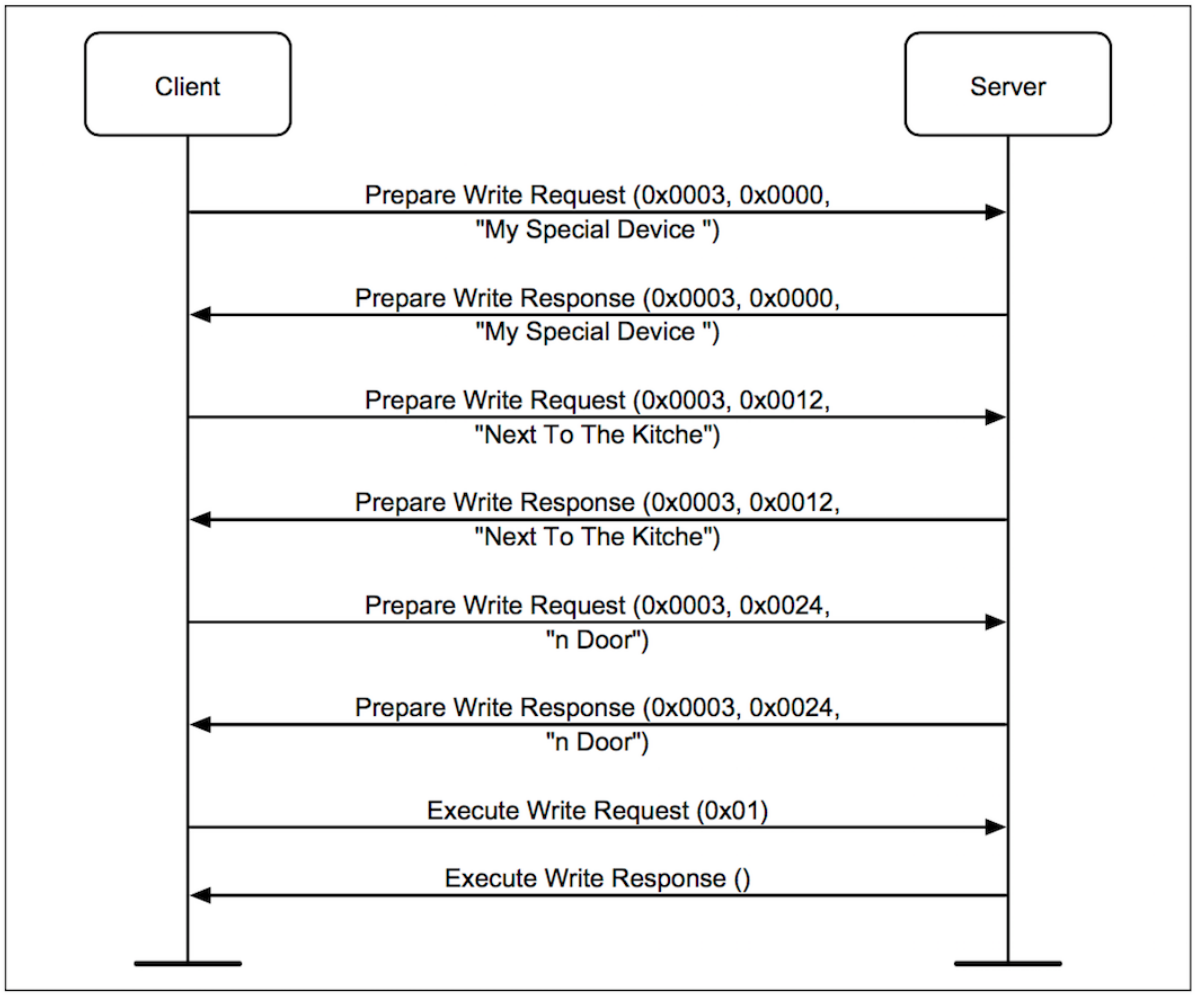
```
case ESP_GATTS_READ_EVT: {
     ESP_LOGI(GATTS_TAG, "GATT_READ_EVT, conn_id %d, trans_id %d, handle %d\n",
              param->read.conn_id, param->read.trans_id, param->read.handle);
              esp_gatt_rsp_t rsp;
              memset(&rsp, 0, sizeof(esp_gatt_rsp_t));
              rsp.attr_value.handle = param->read.handle;
              rsp.attr_value.len = 4;
              rsp.attr_value.value[0] = 0xde;
              rsp.attr_value.value[1] = 0xed;
              rsp.attr_value.value[2] = 0xbe;
              rsp.attr_value.value[3] = 0xef;
              esp_ble_gatts_send_response(gatts_if,
                                          param->read.conn_id,
                                          param->read.trans_id,
                                          ESP_GATT_OK, &rsp);
     break;
    }
```

書き込みイベント処理

書き込みイベントは ESP_GATTS_WRITE_EVT イベントで表現され、次のパラメータを持ちます。

```
uint16_t conn_id;          /*!< Connection id */
uint32_t trans_id;         /*!< Transfer id */
esp_bd_addr_t bda;         /*!< The bluetooth device address which been written */
uint16_t handle;           /*!< The attribute handle */
uint16_t offset;           /*!< Offset of the value, if the value is too long */
bool need_rsp;             /*!< The write operation need to do response */
bool is_prep;              /*!< This write operation is prepare write */
uint16_t len;              /*!< The write attribute value length */
uint8_t *value;            /*!< The write attribute value */
```

本サンプルでは 2 種類の書き込みイベントが実装されています。通常のキャラクタリスティック値の書き込みと長いキャラクタリスティック値の書き込みです。通常のキャラクタリスティック値の書き込みは属性のプロトコルでの最大送信ユニット長（ATT MTU）に収まる場合に利用されます。通常 MTU は 23 バイト長です。長いキャラクタリスティック値の書き込みは、単一の ATT メッセージで送ることの可能なデータより長いものを書き込む際に利用されます。データを Prepare Write Response を使って複数のチャンクに分割して送信します。Prepare Write Response のあとに、Execute Write Request を使って書き込み要求を確定したり、キャンセルしたりします。この挙動は Bluetooth Specification Version 4.2 の第 3 巻パート Google の 4.9 節にて定義されています。長いキャラクタリスティック値の書き込みフローについては次の図で示しています。



書き込みイベントが発生すると、本サンプルではログメッセージが出力され example_write_event_env() 関数が実行されます。

```
case ESP_GATTS_WRITE_EVT: {
     ESP_LOGI(GATTS_TAG, "GATT_WRITE_EVT, conn_id %d, trans_id %d, handle %d\n",
param->write.conn_id, param->write.trans_id, param->write.handle);
     if (!param->write.is_prep){
        ESP_LOGI(GATTS_TAG, "GATT_WRITE_EVT, value len %d, value :", param-
>write.len);
        esp_log_buffer_hex(GATTS_TAG, param->write.value, param->write.len);
        if (gl_profile_tab[PROFILE_B_APP_ID].descr_handle == param->write.handle
&& param->write.len == 2){
            uint16_t descr_value= param->write.value[1]<<8 | param-
>write.value[0];
            if (descr_value == 0x0001){
                if (b_property & ESP_GATT_CHAR_PROP_BIT_NOTIFY){
                    ESP_LOGI(GATTS_TAG, "notify enable");
                    uint8_t notify_data[15];
                    for (int i = 0; i < sizeof(notify_data); ++i)
                    {
                         notify_data[i] = i%0xff;
                     }
                     //the size of notify_data[] need less than MTU size
                     esp_ble_gatts_send_indicate(gatts_if, param->write.conn_id,
```

```
gl_profile_tab[PROFILE_B_APP_ID].char_handle,
                                                sizeof(notify_data),
                                                notify_data, false);
                }
            }else if (descr_value == 0x0002){
                 if (b_property & ESP_GATT_CHAR_PROP_BIT_INDICATE){
                     ESP_LOGI(GATTS_TAG, "indicate enable");
                     uint8_t indicate_data[15];
                     for (int i = 0; i < sizeof(indicate_data); ++i)
                     {
                         indicate_data[i] = i % 0xff;
                      }
                      //the size of indicate_data[] need less than MTU size
                     esp_ble_gatts_send_indicate(gatts_if, param->write.conn_id,

gl_profile_tab[PROFILE_B_APP_ID].char_handle,
                                                sizeof(indicate_data),
                                                indicate_data, true);
                }
             }
             else if (descr_value == 0x0000){
                 ESP_LOGI(GATTS_TAG, "notify/indicate disable ");
             }else{
                 ESP_LOGE(GATTS_TAG, "unknown value");
             }
        }
    }
    example_write_event_env(gatts_if, &a_prepare_write_env, param);
    break;
}
```

example_write_event_env()関数には長いキャラクタリスティック値の書き込みプロシージャについてのロジックが含まれています。

```
void example_write_event_env(esp_gatt_if_t gatts_if, prepare_type_env_t
*prepare_write_env, esp_ble_gatts_cb_param_t *param){
    esp_gatt_status_t status = ESP_GATT_OK;
    if (param->write.need_rsp){
       if (param->write.is_prep){
            if (prepare_write_env->prepare_buf == NULL){
                prepare_write_env->prepare_buf = (uint8_t
*)malloc(PREPARE_BUF_MAX_SIZE*sizeof(uint8_t));
                prepare_write_env->prepare_len = 0;
                if (prepare_write_env->prepare_buf == NULL) {
                    ESP_LOGE(GATTS_TAG, "Gatt_server prep no mem\n");
                    status = ESP_GATT_NO_RESOURCES;
                }
            } else {
                if(param->write.offset > PREPARE_BUF_MAX_SIZE) {
                    status = ESP_GATT_INVALID_OFFSET;
                }
                else if ((param->write.offset + param->write.len) >
PREPARE_BUF_MAX_SIZE) {
                    status = ESP_GATT_INVALID_ATTR_LEN;
                }
            }
```

```
            esp_gatt_rsp_t *gatt_rsp = (esp_gatt_rsp_t
*)malloc(sizeof(esp_gatt_rsp_t));
            gatt_rsp->attr_value.len = param->write.len;
            gatt_rsp->attr_value.handle = param->write.handle;
            gatt_rsp->attr_value.offset = param->write.offset;
            gatt_rsp->attr_value.auth_req = ESP_GATT_AUTH_REQ_NONE;
            memcpy(gatt_rsp->attr_value.value, param->write.value, param-
>write.len);
            esp_err_t response_err = esp_ble_gatts_send_response(gatts_if, param-
>write.conn_id,
                                                                param-
>write.trans_id, status, gatt_rsp);
            if (response_err != ESP_OK){
               ESP_LOGE(GATTS_TAG, "Send response error\n");
            }
            free(gatt_rsp);
            if (status != ESP_GATT_OK){
                return;
            }
            memcpy(prepare_write_env->prepare_buf + param->write.offset,
                   param->write.value,
                   param->write.len);
            prepare_write_env->prepare_len += param->write.len;

        }else{
            esp_ble_gatts_send_response(gatts_if, param->write.conn_id, param-
>write.trans_id, status, NULL);
        }
    }
}
```

クライアントが Write Request もしくは Prepare Write Request を送信すると、サーバは応答する必要があります。しかし、クライアントが Write Without Response コマンドを送信する場合にはサーバは応答を返す必要がありません。これは書き込みプロシージャ内で write.need_rsp パラメータの値を調べることでチェックしています。応答が必要である場合は、プロシージャは応答の準備を続けます。もし必要ない場合はプロシージャは終了します。

```
void example_write_event_env(esp_gatt_if_t gatts_if, prepare_type_env_t
*prepare_write_env,
                             esp_ble_gatts_cb_param_t *param){
    esp_gatt_status_t status = ESP_GATT_OK;
    if (param->write.need_rsp){
```

この関数では write.is_prep で表される Prepare Write Request パラメータが設定されているかをチェックします。これはクライアントが Write Long Characteristic を要求していることを表しています。これが存在していれば、複数の書き込みへの応答を準備するためにプロシージャを続けます。もし存在していなければサーバは単純に単一の書き込み応答を書き戻します。

```
…
if (param->write.is_prep){
…
}else{
        esp_ble_gatts_send_response(gatts_if, param->write.conn_id, param-
>write.trans_id, status, NULL);
}
…
```

長いキャラクタリスティック値の書き込みを扱うため、事前準備バッファ構造が定義され、初期化されます。

```
typedef struct {
    uint8_t                 *prepare_buf;
    int                      prepare_len;
} prepare_type_env_t;

static prepare_type_env_t a_prepare_write_env;
static prepare_type_env_t b_prepare_write_env;
```

事前準備バッファを利用するため、あるメモリ領域を確保します。メモリ不足でこのメモリ確保が失敗した場合、エラーを出力します。

```
if (prepare_write_env->prepare_buf == NULL) {
    prepare_write_env->prepare_buf =
    (uint8_t*)malloc(PREPARE_BUF_MAX_SIZE*sizeof(uint8_t));
    prepare_write_env->prepare_len = 0;
    if (prepare_write_env->prepare_buf == NULL) {
       ESP_LOGE(GATTS_TAG, "Gatt_server prep no mem\n");
       status = ESP_GATT_NO_RESOURCES;
    }
}
```

バッファが NULL ではない場合、初期化は完了していることを表していますので、入力された書き込みのメッセージ長とオフセットがバッファにはまるかをチェックします。

```
else {
        if(param->write.offset > PREPARE_BUF_MAX_SIZE) {
                status = ESP_GATT_INVALID_OFFSET;
        }
        else if ((param->write.offset + param->write.len) > PREPARE_BUF_MAX_SIZE)
{
                 status = ESP_GATT_INVALID_ATTR_LEN;
        }
}
```

プロシージャは、クライアントにメッセージを送り返すため esp_gatt_rsp_t 型の応答を準備します。応答は書き込みリクエストと同一のパラメータ、例えば長さ、ハンドル、オフセットなどを用いて生成されます。さらに GATT 認証タイプが必要で、本キャラクタリスティックへの書き込みは ESP_GATT_AUTH_REQ_NONE が必要です。これはクライアントが最初に認証せずに本キャラクタリスティックへの書き込みができると言うことです。応答が設定されたら利用していたメモリは解放されます。

```
esp_gatt_rsp_t *gatt_rsp = (esp_gatt_rsp_t *)malloc(sizeof(esp_gatt_rsp_t));
gatt_rsp->attr_value.len = param->write.len;
gatt_rsp->attr_value.handle = param->write.handle;
gatt_rsp->attr_value.offset = param->write.offset;
gatt_rsp->attr_value.auth_req = ESP_GATT_AUTH_REQ_NONE;
memcpy(gatt_rsp->attr_value.value, param->write.value, param->write.len);
esp_err_t response_err = esp_ble_gatts_send_response(gatts_if, param-
>write.conn_id,
                                                     param->write.trans_id,
status, gatt_rsp);
if (response_err != ESP_OK){
        ESP_LOGE(GATTS_TAG, "Send response error\n");
}
```

```
free(gatt_rsp);
if (status != ESP_GATT_OK){
        return;
}
```

最後に、入力されたデータは作成されたバッファにコピーされ、その長さはオフセットから足されていきます。

```
memcpy(prepare_write_env->prepare_buf + param->write.offset,
       param->write.value,
       param->write.len);
prepare_write_env->prepare_len += param->write.len;
```

クライアントは長いメッセージの書き込みシーケンスを Executive Write Request を送って終了します。このメッセージによって ESP_GATTS_EXEC_WRITE_EVT イベントが発生します。サーバではこのイベントに対して応答を返し、example_exec_write_event_env()関数を読んで扱います。

```
case ESP_GATTS_EXEC_WRITE_EVT:
     ESP_LOGI(GATTS_TAG,"ESP_GATTS_EXEC_WRITE_EVT");
     esp_ble_gatts_send_response(gatts_if, param->write.conn_id, param-
>write.trans_id, ESP_GATT_OK, NULL);
     example_exec_write_event_env(&a_prepare_write_env, param);
     break;
```

Execute Write 関数を見てみましょう。

```
void example_exec_write_event_env(prepare_type_env_t *prepare_write_env,
esp_ble_gatts_cb_param_t *param){
    if (param->exec_write.exec_write_flag == ESP_GATT_PREP_WRITE_EXEC){
        esp_log_buffer_hex(GATTS_TAG, prepare_write_env->prepare_buf,
prepare_write_env->prepare_len);
    }
    else{
        ESP_LOGI(GATTS_TAG,"ESP_GATT_PREP_WRITE_CANCEL");
    }
    if (prepare_write_env->prepare_buf) {
        free(prepare_write_env->prepare_buf);
        prepare_write_env->prepare_buf = NULL;
    }
####     prepare_write_env->prepare_len = 0;
}
```

書き込みの実行は以前に行った書き込みプロシージャを確定させるか、キャンセルするかのために使われます。長いキャラクタリスティック値の書き込みプロシージャによって利用されます。これを行うために、この関数ではイベントと一緒に受け取ったパラメータの exec_write_flag をチェックします。このフラグが exec_write_flag で表現される実行フラグと等しい場合、書き込みは確定されてログにバッファが書き込まれます。異なっていれば、書き込みはキャンセルされたので、バッファに書かれていたデータはすべて消去されます。

```
if (param->exec_write.exec_write_flag == ESP_GATT_PREP_WRITE_EXEC){
   esp_log_buffer_hex(GATTS_TAG,
                      prepare_write_env->prepare_buf,
                      prepare_write_env->prepare_len);
 }
else{
    ESP_LOGI(GATTS_TAG,"ESP_GATT_PREP_WRITE_CANCEL");
 }
```

最後に、長い書き込み操作から得たデータチャンクを保存するために作られていたバッファ構造が解放されてポインタは NULL に設定されて次の長い書き込み操作を待ち受けられるようにします。

```
if (prepare_write_env->prepare_buf) {
    free(prepare_write_env->prepare_buf);
    prepare_write_env->prepare_buf = NULL;
}
prepare_write_env->prepare_len = 0;
```

最後に

本ドキュメントでは GATT サーバのサンプルコードを各章で一通り見てきました。アプリケーションはアプリケーションプロファイルの概念で設計されます。さらに、イベントハンドラの登録を行うために使用しているプロシージャについて説明しました。イベントは設定のステップごとに発生します。たとえば広告パラメータを定義したり、接続パラメータを更新したり、サービスとキャラクタリスティックを作成したりしたときです。最後に読み込み、書き込みイベントの扱い方を説明しています。これには属性プロトコルメッセージサイズに収まるようなチャンクに分割することで長い性質を書き込む方法を含んでいます。