ESP32 BT&BLE Dual-mode Bluetooth

Version 1.0

About This Guide

This document introduces the ESP32 BT&BLE dual-mode bluetooth.

Release Notes

Date	Version	Release notes
2018.02	V1.0	Initial release.

Documentation Change Notification

Espressif provides email notifications to keep customers updated on changes to technical documentation. Please subscribe *here*.

Certification

Download certificates for Espressif products from *here*.

Table of Contents

1. BT&BLE Coexistence Diagram

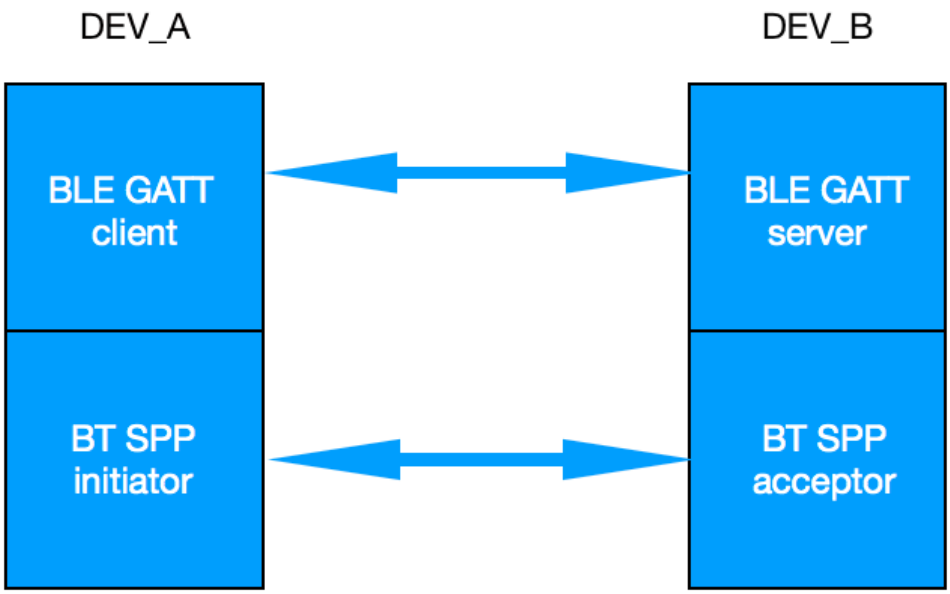

Figure 1-1. BT&BLE Coexistence System

2. Process Description

2.1. Initialization

After the DEV_A is powered on, the functions of the BT SPP initiator and the BLE GATT client are initialized. After the initialization, DEV_A searches for the Classic Bluetooth (SPP) device, and connects to DEV_B upon finding it. After the SPP connection, DEV_A searches for the BLE broadcast and connects to DEV_B upon finding it. The configuration of DEV_A is shown in Figure 2-1 below.

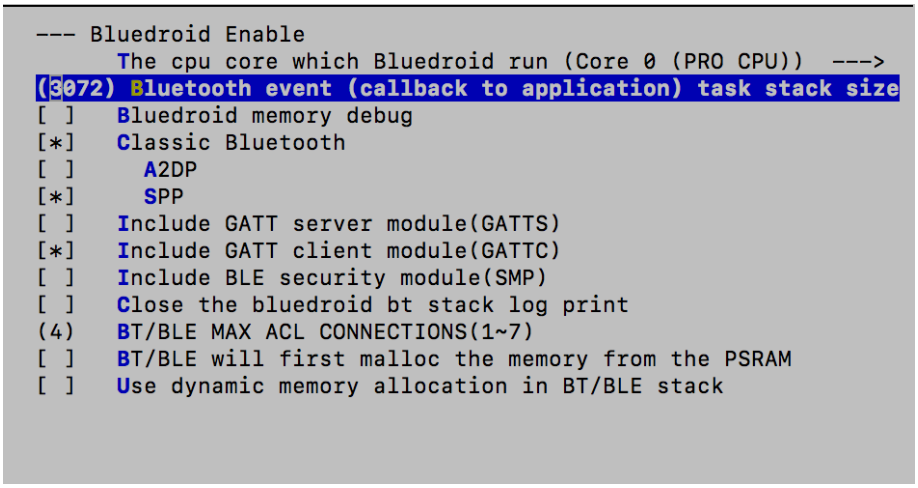

Figure 2-1. The Configuration Interface of DEV_A

After the DEV_B is powered on, the functions of the BT SPP acceptor and the BLE GATT server are initialized. After the initialization, DEV_B starts the Classic Bluetooth inquiry scan and BLE broadcast, waiting to be connected. The configuration of DEV_B is shown in Figure 2-2 below.

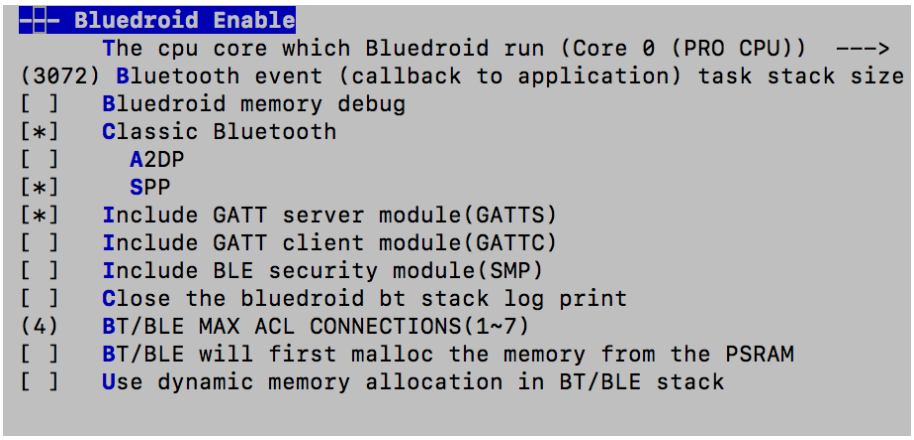

Figure 2-2. The Configuration Interface of DEV_B

> ⚠ ***Notice:***
>
> *The **Classic Bluetooth**, **SPP** and **GATTC/GATTS** options must be enabled in **menuconfig**.*

2.2. Broadcast Instruction

The device name (in Extended Inquiry Response, EIR) of the Classic Bluetooth inquiry can be the same as the device name of the BLE broadcast, or it can be totally different. To choose between device names, BLE can use the ***esp_ble_gap_config_adv_data_raw()*** function and broadcast under a specific Bluetooth device name.

2.3. Connection

DEV_B will automatically start the Classic Bluetooth inquiry scan and BLE broadcast, after its initialization. The BT SPP initiator of DEV_A keeps searching for DEV_B and connects to DEV_B upon finding it; then, it searches for BLE devices and connects to DEV_B again upon finding it. The interaction between DEV_A and DEV_B is shown in Figure 2-3.

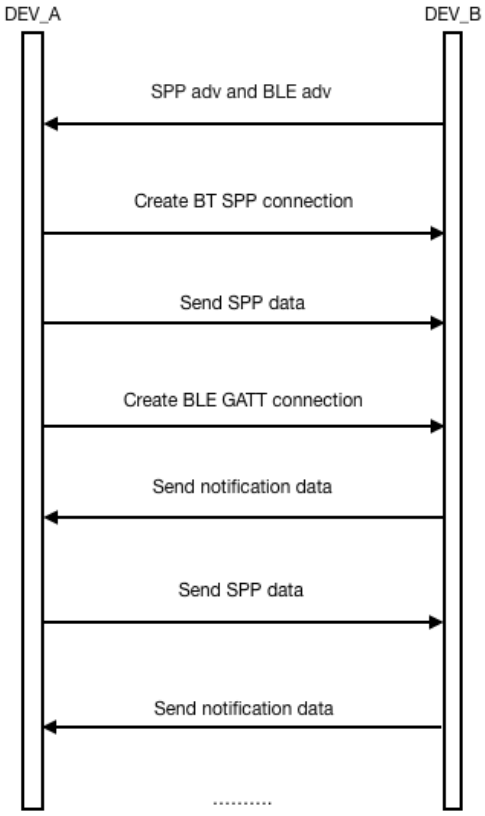

Figure 2-3. Test Flow Chart

3. Code Description

3.1. Initialization

3.1.1. Initialization Process

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

3.1.2. Initialize and Enable the Controller

```
esp_bt_controller_config_t bt_cfg = BT_CONTROLLER_INIT_CONFIG_DEFAULT();
ret = esp_bt_controller_init(&bt_cfg);
if (ret) {
   ESP_LOGE(BT_BLE_COEX_TAG, "%s initialize controller failed\n", __func__);
   return;
}

ret = esp_bt_controller_enable(ESP_BT_MODE_BTDM);
if (ret) {
    ESP_LOGE(BT_BLE_COEX_TAG, "%s enable controller failed\n", __func__);
    return;
}
```

> ⚠ ***Notice:***
>
> *To use dual-mode Bluetooth here, you must initialize the controller to **ESP_BT_MODE_BTDM** and select the appropriate option in the **menuconfig** interface.*

3.1.3. Initialize and Enable the Host

```
ret = esp_bluedroid_init();
    if (ret) {
        ESP_LOGE(BT_BLE_COEX_TAG, "%s init bluetooth failed\n", __func__);
        return;
    }
    ret = esp_bluedroid_enable();
    if (ret) {
        ESP_LOGE(BT_BLE_COEX_TAG, "%s enable bluetooth failed\n", __func__);
        return;
    }
```

3.1.4. Initialize the BT SPP Acceptor and the GATT Server in DEV_B

After the initialization of both the controller and the host, the BT SPP acceptor and the BLE GATT server are initialized:

- bt_spp_init()
- ble_gatts_init()

The actual codes in our example are shown below:

```
void bt_spp_init(void)
{
   //Register spp callback
    esp_err_t ret = esp_spp_register_callback(esp_spp_cb);
    if (ret) {
        ESP_LOGE(BT_SPP_TAG, "%s spp register failed\n", __func__);
        return;
    }
    ret = esp_spp_init(esp_spp_mode);
    if (ret) {
        ESP_LOGE(BT_SPP_TAG, "%s spp init failed\n", __func__);
        return;
    }
}


static void esp_spp_cb(esp_spp_cb_event_t event, esp_spp_cb_param_t *param)
{
    switch (event) {
    case ESP_SPP_INIT_EVT:
    //Once the SPP callback has been registered, ESP_SPP_INIT_EVT is triggered. Here sets
the device name and classic bluetooth scan mode, then starts the advertising.
        ESP_LOGI(BT_SPP_TAG, "ESP_SPP_INIT_EVT\n");
        esp_bt_dev_set_device_name(BT_DEVICE_NAME);
        esp_bt_gap_set_scan_mode(ESP_BT_SCAN_MODE_CONNECTABLE_DISCOVERABLE);
        esp_spp_start_srv(sec_mask,role_slave, 0, SPP_SERVER_NAME);
        break;
    case ESP_SPP_DISCOVERY_COMP_EVT:
        ESP_LOGI(BT_SPP_TAG, "ESP_SPP_DISCOVERY_COMP_EVT\n");
        break;
    //After the SPP connection, ESP_SPP_OPEN_EVT is triggered.
    case ESP_SPP_OPEN_EVT:
        ESP_LOGI(BT_SPP_TAG, "ESP_SPP_OPEN_EVT\n");
        break;
    //After the SPP disconnection, ESP_SPP_CLOSE_EVT is triggered.
    case ESP_SPP_CLOSE_EVT:
        ESP_LOGI(BT_SPP_TAG, "ESP_SPP_CLOSE_EVT\n");
        break;
    case ESP_SPP_START_EVT:
        ESP_LOGI(BT_SPP_TAG, "ESP_SPP_START_EVT\n");
        break;
    case ESP_SPP_CL_INIT_EVT:
        ESP_LOGI(BT_SPP_TAG, "ESP_SPP_CL_INIT_EVT\n");
        break;
    case ESP_SPP_DATA_IND_EVT:
#if (SPP_SHOW_MODE == SPP_SHOW_DATA)
        ESP_LOGI(BT_SPP_TAG, "ESP_SPP_DATA_IND_EVT len=%d handle=%d\n",
```

```
                    param->data_ind.len, param->data_ind.handle);
            esp_log_buffer_hex("",param->data_ind.data,param->data_ind.len);
        #else
            gettimeofday(&time_new, NULL);
            data_num += param->data_ind.len;
            if (time_new.tv_sec - time_old.tv_sec >= 3) {
                print_speed();
            }
        #endif
            break;
        case ESP_SPP_CONG_EVT:
            ESP_LOGI(BT_SPP_TAG, "ESP_SPP_CONG_EVT\n");
            break;
        case ESP_SPP_WRITE_EVT:
            ESP_LOGI(BT_SPP_TAG, "ESP_SPP_WRITE_EVT\n");
            break;
        case ESP_SPP_SRV_OPEN_EVT:
            ESP_LOGI(BT_SPP_TAG, "ESP_SPP_SRV_OPEN_EVT\n");
            gettimeofday(&time_old, NULL);
            break;
        default:
            break;
        }
    }
```

```
    static void ble_gatts_init(void)
    {
        esp_err_t ret = esp_ble_gatts_register_callback(gatts_event_handler);
        if (ret){
            ESP_LOGE(BT_BLE_COEX_TAG, "gatts register error, error code = %x", ret);
            return;
        }
        ret = esp_ble_gap_register_callback(gap_event_handler);
        if (ret){
            ESP_LOGE(BT_BLE_COEX_TAG, "gap register error, error code = %x", ret);
            return;
        }
        ret = esp_ble_gatts_app_register(PROFILE_A_APP_ID);
        if (ret){
            ESP_LOGE(BT_BLE_COEX_TAG, "gatts app register error, error code = %x", ret);
            return;
        }
        ret = esp_ble_gatts_app_register(PROFILE_B_APP_ID);
        if (ret){
            ESP_LOGE(BT_BLE_COEX_TAG, "gatts app register error, error code = %x", ret);
            return;
        }
        esp_err_t local_mtu_ret = esp_ble_gatt_set_local_mtu(500);
        if (local_mtu_ret){
            ESP_LOGE(BT_BLE_COEX_TAG, "set local  MTU failed, error code = %x", local_mtu_ret);
        }
```

```
    xTaskCreate(notify_task, "notify_task", 2048, NULL, configMAX_PRIORITIES - 6, NULL);
    gatts_semaphore = xSemaphoreCreateMutex();
    if (!gatts_semaphore) {
        return;
    }

}
```

> ⚠ ***Notice:***
>
> *For more information about the GATTS-related APIs, please refer to GATT_SERVER_EXAMPLE_WALKTHROUGH.md in the ESP_IDF gatt_server demo.*

3.1.5. Initialize the BT SPP Initiator and the GATT Client in DEV_A

After the initialization of the controller and the host, the BT SPP initiator and the BLE GATT client are initialized:

- bt_spp_init();
- ble_gattc_init()

The actual codes in our example are shown below:

```
void bt_spp_init(void)
{
      esp_err_t ret = esp_bt_gap_register_callback(esp_bt_gap_cb);
    if (ret) {
        ESP_LOGE(SPP_TAG, "%s gap register failed\n", __func__);
        return;
    }
    //Once the SPP callback has been registered, ESP_SPP_INIT_EVT is triggered.
    ret = esp_spp_register_callback(esp_spp_cb);
    if (ret) {
        ESP_LOGE(SPP_TAG, "%s spp register failed\n", __func__);
        return;
    }
    ret = esp_spp_init(esp_spp_mode);
    if (ret) {
        ESP_LOGE(SPP_TAG, "%s spp init failed\n", __func__);
        return;
    }
static void esp_bt_gap_cb(esp_bt_gap_cb_event_t event, esp_bt_gap_cb_param_t *param)
{
    switch(event){
       //Once the broadcast information is found, ESP_BT_GAP_DISC_RES_EVT is triggered. Then
the device is connected.
        case ESP_BT_GAP_DISC_RES_EVT:
            ESP_LOGI(SPP_TAG, "ESP_BT_GAP_DISC_RES_EVT");
            esp_log_buffer_hex(SPP_TAG, param->disc_res.bda, ESP_BD_ADDR_LEN);
            for (int i = 0; i < param->disc_res.num_prop; i++){
                if (param->disc_res.prop[i].type == ESP_BT_GAP_DEV_PROP_EIR
```

```
                    && get_name_from_eir(param->disc_res.prop[i].val, peer_bdname,
&peer_bdname_len)){
                    if (strlen(remote_spp_name) == peer_bdname_len
                        && strncmp(peer_bdname, remote_spp_name, peer_bdname_len) == 0) {
                        memcpy(peer_bd_addr, param->disc_res.bda, ESP_BD_ADDR_LEN);
                        esp_spp_start_discovery(peer_bd_addr);
                        esp_bt_gap_cancel_discovery();
                    }
                }
            }
            break;
        case ESP_BT_GAP_DISC_STATE_CHANGED_EVT:
            ESP_LOGI(SPP_TAG, "ESP_BT_GAP_DISC_STATE_CHANGED_EVT");
            break;
        case ESP_BT_GAP_RMT_SRVCS_EVT:
            ESP_LOGI(SPP_TAG, "ESP_BT_GAP_RMT_SRVCS_EVT");
            break;
        case ESP_BT_GAP_RMT_SRVC_REC_EVT:
            ESP_LOGI(SPP_TAG, "ESP_BT_GAP_RMT_SRVC_REC_EVT");
            break;
        default:
            break;
    }
}

static void esp_spp_cb(esp_spp_cb_event_t event, esp_spp_cb_param_t *param)
{
    switch (event) {
    case ESP_SPP_INIT_EVT:
        ESP_LOGI(SPP_TAG, "ESP_SPP_INIT_EVT");
        esp_bt_dev_set_device_name(EXCAMPLE_DEVICE_NAME);
        esp_bt_gap_set_scan_mode(ESP_BT_SCAN_MODE_CONNECTABLE_DISCOVERABLE);
        esp_bt_gap_start_discovery(inq_mode, inq_len, inq_num_rsps);

        break;
    case ESP_SPP_DISCOVERY_COMP_EVT:
        ESP_LOGI(SPP_TAG, "ESP_SPP_DISCOVERY_COMP_EVT status=%d scn_num=%d",param-
>disc_comp.status, param->disc_comp.scn_num);
        if (param->disc_comp.status == ESP_SPP_SUCCESS) {
            esp_spp_connect(sec_mask, role_master, param->disc_comp.scn[0], peer_bd_addr);
        }
        break;
    case ESP_SPP_OPEN_EVT:
        ESP_LOGI(SPP_TAG, "ESP_SPP_OPEN_EVT");
        //BLE starts scanning
        uint32_t duration = 30;
        esp_ble_gap_start_scanning(duration);

        esp_spp_write(param->srv_open.handle, SPP_DATA_LEN, spp_data);
        gettimeofday(&time_old, NULL);
        break;
```

```
        case ESP_SPP_CLOSE_EVT:
            ESP_LOGI(SPP_TAG, "ESP_SPP_CLOSE_EVT");
            break;
        case ESP_SPP_START_EVT:
            ESP_LOGI(SPP_TAG, "ESP_SPP_START_EVT");
            break;
        case ESP_SPP_CL_INIT_EVT:
            ESP_LOGI(SPP_TAG, "ESP_SPP_CL_INIT_EVT");
            break;
        case ESP_SPP_DATA_IND_EVT:
            ESP_LOGI(SPP_TAG, "ESP_SPP_DATA_IND_EVT");
            break;
        case ESP_SPP_CONG_EVT:
#if (SPP_SHOW_MODE == SPP_SHOW_DATA)
            ESP_LOGI(SPP_TAG, "ESP_SPP_CONG_EVT cong=%d", param->cong.cong);
#endif
            if (param->cong.cong == 0) {
                esp_spp_write(param->cong.handle, SPP_DATA_LEN, spp_data);
            }
            break;
        case ESP_SPP_WRITE_EVT:
#if (SPP_SHOW_MODE == SPP_SHOW_DATA)
            ESP_LOGI(SPP_TAG, "ESP_SPP_WRITE_EVT len=%d cong=%d", param->write.len , param->write.cong);
            esp_log_buffer_hex("",spp_data,SPP_DATA_LEN);
#else
            gettimeofday(&time_new, NULL);
            data_num += param->write.len;
            if (time_new.tv_sec - time_old.tv_sec >= 3) {
                print_speed();
            }
#endif
            if (param->write.cong == 0) {
                esp_spp_write(param->write.handle, SPP_DATA_LEN, spp_data);
            }
            break;
        case ESP_SPP_SRV_OPEN_EVT:
            ESP_LOGI(SPP_TAG, "ESP_SPP_SRV_OPEN_EVT");
            break;
        default:
            break;
        }
    }
```

```
    {
        //Register the callback function to the GAP module.
        esp_err_t ret = esp_ble_gap_register_callback(esp_gap_cb);
        if (ret){
            ESP_LOGE(GATTC_TAG, "%s gap register failed, error code = %x\n", __func__, ret);
```

```
            return;
        }

        //Register the callback function to the GATTC module.
        ret = esp_ble_gattc_register_callback(esp_gattc_cb);
        if(ret){
            ESP_LOGE(GATTC_TAG, "%s gattc register failed, error code = %x\n", __func__, ret);
            return;
        }

        ret = esp_ble_gattc_app_register(PROFILE_A_APP_ID);
        if (ret){
            ESP_LOGE(GATTC_TAG, "%s gattc app register failed, error code = %x\n", __func__,
ret);
        }

        esp_err_t local_mtu_ret = esp_ble_gatt_set_local_mtu(500);
        if (local_mtu_ret){
            ESP_LOGE(GATTC_TAG, "set local  MTU failed, error code = %x", local_mtu_ret);
        }

        xTaskCreate(gattc_notify_task, "gattc_task", 2048, NULL, 10, NULL);
}
```

> ⚠ ***Notice:***
>
> *For more information about the GATTC-related APIs, please refer to gatt_client_example_walkthrough.md in the ESP_IDF gatt_client demo.*

3.2. Connection

After its initialization, DEV_A starts to search for Classic Bluetooth devices, and returns an ***ESP_BT_GAP_DISC_RES_EVT*** event by calling the ***esp_bt_gap_cb*** function upon finding a Classic Bluetooth device. DEV_A connects to this device by calling ***esp_spp_start_discovery(peer_bd_addr)***, if certain conditions are met. After the successful connection, an ***ESP_SPP_OPEN_EVT*** event is returned with the ***esp_spp_cb*** function. In this event, DEV_A starts sending SPP data, calculates the SPP rate, and searches for BLE devices.

The BLE device will return an ***ESP_GAP_SEARCH_INQ_RES_EVT*** event by calling the ***esp_gap_cb*** function upon finding a broadcast, and will connect to the broadcasting device by calling the ***esp_ble_gattc_open()*** function, if certain conditions are met. After successfully connecting to the broadcasting device, the BLE device will return an ***ESP_GATTC_CONNECT_EVT*** event by calling ***gattc_profile_event_handler***. Additionally, after the successful BLE connection, the BLE device will register the GATT notification of the peer device, and prepare for the subsequent data transmission.

3.3. Data Transmission and Reception

After the SPP connection, DEV_A sends SPP data and calculates the SPP rate by calling ***esp_spp_write()***.

After the successful connection, DEV_A registers the GATT notification of the peer device and starts listening. After receiving the notification data, DEV_A returns an ***ESP_GATTC_NOTIFY_EVT*** event by calling ***gattc_profile_event_handler***, and calculates the data length and rate.

After the SPP of DEV_B is connected, it waits for the peer device to send SPP data. After the BLE of DEV_B is connected, DEV_A will enable DEV_B's GATT notification. DEV_B initiates ***notify_task*** during the GATTS init. After ***notify_task*** detects that notifications are enabled, ***esp_ble_gatts_send_indicate()*** is called to send the GATT notification data.

3.4. Performance

The BT SPP, after connecting to the BLE, will automatically send data and calculate the throughput. The rate of BT SPP is about 230 KB/s when it is running alone, and the rate of BLE is about 40 KB/s when it is running alone (the optimized rate can be 90 KB/s). Currently, only the rate of the GATT notification is calculated in the BLE. The current rates of BT SPP and BLE GATT notifications are 120 KB/s and 30 KB/s, respectively, which are both adjustable. Specifically, you can adjust the length and frequency of the GATT notification by adjusting the BLE connection parameters. The throughput log of SPP and GATT can be seen in Figure 3-1.

```
I (218200) COEX_BT_SPP: bt spp speed : 119.4886 kByte/s
I (218690) COEX_BLE_GATTC: ble Notify speed = 31.6670 kByte/s
I (220690) COEX_BLE_GATTC: ble Notify speed = 31.6710 kByte/s
I (221220) COEX_BT_SPP: bt spp speed : 113.7993 kByte/s
I (222690) COEX_BLE_GATTC: ble Notify speed = 31.6710 kByte/s
I (224180) COEX_BT_SPP: bt spp speed : 102.2706 kByte/s
I (224690) COEX_BLE_GATTC: ble Notify speed = 31.6720 kByte/s
I (226690) COEX_BLE_GATTC: ble Notify speed = 31.6610 kByte/s
I (227190) COEX_BT_SPP: bt spp speed : 118.6859 kByte/s
I (228690) COEX_BLE_GATTC: ble Notify speed = 31.6550 kByte/s
I (230190) COEX_BT_SPP: bt spp speed : 120.6793 kByte/s
I (230690) COEX_BLE_GATTC: ble Notify speed = 31.6490 kByte/s
I (232690) COEX_BLE_GATTC: ble Notify speed = 31.6440 kByte/s
I (233200) COEX_BT_SPP: bt spp speed : 114.4587 kByte/s
I (234690) COEX_BLE_GATTC: ble Notify speed = 31.6460 kByte/s
I (236210) COEX_BT_SPP: bt spp speed : 122.5045 kByte/s
I (236690) COEX_BLE_GATTC: ble Notify speed = 31.6540 kByte/s
I (238690) COEX_BLE_GATTC: ble Notify speed = 31.6540 kByte/s
I (239200) COEX_BT_SPP: bt spp speed : 119.1479 kByte/s
I (240690) COEX_BLE_GATTC: ble Notify speed = 31.6590 kByte/s
I (242180) COEX_BT_SPP: bt spp speed : 131.6273 kByte/s
I (242690) COEX_BLE_GATTC: ble Notify speed = 31.6610 kByte/s
I (244690) COEX_BLE_GATTC: ble Notify speed = 31.6610 kByte/s
I (245170) COEX_BT_SPP: bt spp speed : 118.9349 kByte/s
I (246690) COEX_BLE_GATTC: ble Notify speed = 31.6680 kByte/s
I (248170) COEX_BT_SPP: bt spp speed : 112.7763 kByte/s
I (248690) COEX_BLE_GATTC: ble Notify speed = 31.6720 kByte/s
I (250690) COEX_BLE_GATTC: ble Notify speed = 31.6750 kByte/s
I (251220) COEX_BT_SPP: bt spp speed : 117.2336 kByte/s
I (252690) COEX_BLE_GATTC: ble Notify speed = 31.6680 kByte/s
I (254190) COEX_BT_SPP: bt spp speed : 125.9391 kByte/s
I (254690) COEX_BLE_GATTC: ble Notify speed = 31.6640 kByte/s
I (256690) COEX_BLE_GATTC: ble Notify speed = 31.6670 kByte/s
I (257180) COEX_BT_SPP: bt spp speed : 128.9138 kByte/s
I (258690) COEX_BLE_GATTC: ble Notify speed = 31.6650 kByte/s
I (260170) COEX_BT_SPP: bt spp speed : 120.5812 kByte/s
I (260690) COEX_BLE_GATTC: ble Notify speed = 31.6000 kByte/s
I (262690) COEX_BLE_GATTC: ble Notify speed = 31.5780 kByte/s
I (263240) COEX_BT_SPP: bt spp speed : 105.1236 kByte/s
I (264690) COEX_BLE_GATTC: ble Notify speed = 31.5790 kByte/s
I (266180) COEX_BT_SPP: bt spp speed : 123.2576 kByte/s
I (266690) COEX_BLE_GATTC: ble Notify speed = 31.5820 kByte/s
I (268690) COEX_BLE_GATTC: ble Notify speed = 31.5820 kByte/s
I (269180) COEX_BT_SPP: bt spp speed : 122.6515 kByte/s
I (270690) COEX_BLE_GATTC: ble Notify speed = 31.5860 kByte/s
```

Figure 3-1. The Throughput Log of SPP and GATT

Espressif IoT Team

www.espressif.com