ESP32 BT&BLE 双模蓝牙共存说明

版本 1.0

关于本手册

本文档为 ESP32 BT&BLE 双模蓝牙共存说明。

发布说明

日期	版本	发布说明
2018.02	V1.0	首次发布。

文档变更通知

用户可通过乐鑫官网订阅技术文档变更的电子邮件通知。

证书下载

用户可通过乐鑫官网下载产品证书。

目录

1. BT&BLE 共存结构图 1
2. 流程说明 2
    2.1. 初始化流程 2
    2.2. 广播说明 3
    2.3. 连接流程 3
3. 代码说明 4
    3.1. 初始化 4
        3.1.1. 初始化流程 4
        3.1.2. 初始化并使能 controller 4
        3.1.3. 初始化并使能 host 4
        3.1.4. 在 DEV_B 中初始化 BT SPP acceptor 和 GATT server 5
        3.1.5. 在 DEV_A 中初始化 BT SPP initiator 和 GATT client 7
    3.2. 连接 10
    3.3. 数据发送与接收 11
    3.4. 性能说明 11

1. BT&BLE 共存结构图

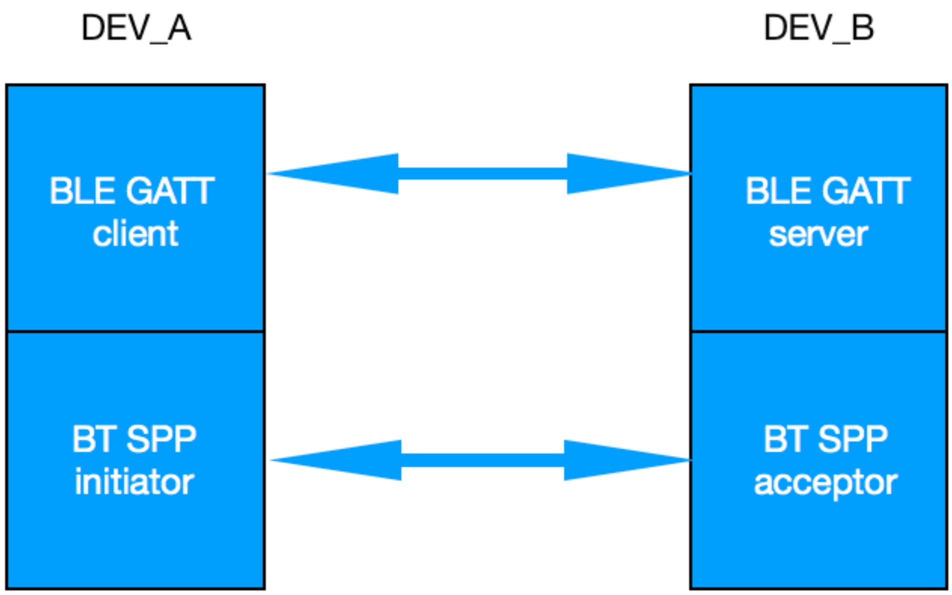

图 1-1. BT&BLE 共存系统结构图

2. 流程说明

2.1. 初始化流程

DEV_A 上电后将初始化 BT SPP initiator 和 BLE GATT client 功能。初始化完成后，开始查找经典蓝牙 (SPP) 设备，找到设备后进行连接。SPP 连接完成后开始搜索 BLE 广播，搜索到设备后进行连接。DEV_A 配置如下图 2-1 所示。

```
--- Bluedroid Enable
          The cpu core which Bluedroid run (Core 0 (PRO CPU))  --->
(3072) Bluetooth event (callback to application) task stack size
[ ]    Bluedroid memory debug
[*]    Classic Bluetooth
[ ]      A2DP
[*]      SPP
[ ]    Include GATT server module(GATTS)
[*]    Include GATT client module(GATTC)
[ ]    Include BLE security module(SMP)
[ ]    Close the bluedroid bt stack log print
(4)    BT/BLE MAX ACL CONNECTIONS(1~7)
[ ]    BT/BLE will first malloc the memory from the PSRAM
[ ]    Use dynamic memory allocation in BT/BLE stack
```

图 2-1. DEV_A 配置界面

DEV_B 上电后将初始化 BT SPP acceptor 和 BLE GATT server 功能。初始化完成后，经典蓝牙开始 inquire scan 和 BLE 广播，等待被连接。DEV_B 配置如下图 2-2 所示。

```
--- Bluedroid Enable
          The cpu core which Bluedroid run (Core 0 (PRO CPU))  --->
(3072) Bluetooth event (callback to application) task stack size
[ ]    Bluedroid memory debug
[*]    Classic Bluetooth
[ ]      A2DP
[*]      SPP
[*]    Include GATT server module(GATTS)
[ ]    Include GATT client module(GATTC)
[ ]    Include BLE security module(SMP)
[ ]    Close the bluedroid bt stack log print
(4)    BT/BLE MAX ACL CONNECTIONS(1~7)
[ ]    BT/BLE will first malloc the memory from the PSRAM
[ ]    Use dynamic memory allocation in BT/BLE stack
```

图 2-2. DEV_B 配置界面

> ⚠️ **注意：**
>
> 在 ***menuconfig*** 中必须选中 ***Classic Bluetooth***、***SPP*** 和 ***GATTC/GATTS*** 选项。

2.2. 广播说明

经典蓝牙 inquiry scan 时的设备名称（EIR 中）与 BLE 广播时的设备名称可以是同一个名称，也可以是不同名称。若需区分两者的设备名称，BLE 可以使用 ***esp_ble_gap_config_adv_data_raw()*** 函数，广播特定的蓝牙设备名称。

2.3. 连接流程

DEV_B 初始化完成后会自动进入 BT inquiry scan 状态，并且进行 BLE 广播，DEV_A 的 BT SPP initiator 搜索到对应设备后进行连接，然后再开始搜索 BLE 设备，搜索到设备后进行连接。DEV_A 和 DEV_B 之间的通信，请见图 2-3。

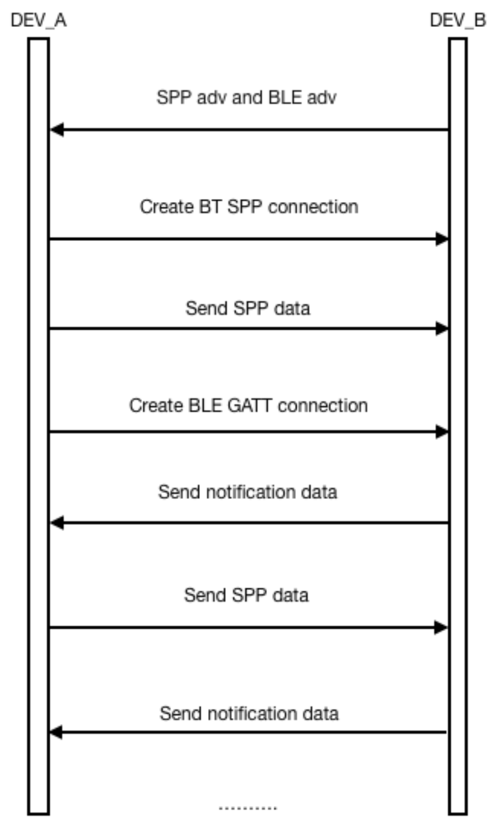

图 2-3. DEV_A 和 DEV_B 测试流程图

3. 代码说明

3.1. 初始化

3.1.1. 初始化流程

```
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. 初始化并使能 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;
}
```

> ⚠️ **注意：**
>
> 这里如需使用双模蓝牙，则必须将 *controller* 初始化为 ***ESP_BT_MODE_BTDM***，且在 ***menuconfig*** 中选择相应的选项。

3.1.3. 初始化并使能 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. 在 DEV_B 中初始化 BT SPP acceptor 和 GATT server

Controller 和 host 初始化完成后，则开始初始化 BT SPP 和 BLE GATT server：

- bt_spp_init()
- ble_gatts_init()

具体代码如下：

```
void bt_spp_init(void)
{
    //注册 SPP 的 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:
    //SPP callback 注册成功后，将返回 ESP_SPP_INIT_EVT 事件，在此事件中设置蓝牙名称，设置经
典蓝牙 Scan 模式
        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;
    //SPP 连接成功后，将返回 ESP_SPP_OPEN_EVT 事件
    case ESP_SPP_OPEN_EVT:
        ESP_LOGI(BT_SPP_TAG, "ESP_SPP_OPEN_EVT\n");
        break;
    //SPP 断开后，将返回 ESP_SPP_CLOSE_EVT 事件
    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;
        }

    }
```

> ⚠️ **注意：**
>
> *GATTS* 的相关 *API* 不在这里一一说明，具体可参考 *ESP_IDF gatt_server demo* 中的 *GATT_SERVER_EXAMPLE_WALKTHROUGH.md*

3.1.5. 在 DEV_A 中初始化 BT SPP initiator 和 GATT client

Controller 和 host 初始化完成后，开始初始化 BT SPP initiator 和 BLE GATT client：

- bt_spp_init();
- ble_gattc_init()

具体代码如下：

```
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;
    }
    //注册 SPP callback，注册成功后对应的 callback 中会有 ESP_SPP_INIT_EVT 事件回调
    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){
    //搜索到的扫描结果会在 ESP_BT_GAP_DISC_RES_EVT 中，查找符合条件的设备，然后连接
```

```
        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 开始扫描
        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;
        }
    }
```

```
{
    //注册 callback 功能至 GAP
    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;
    }

    //注册 callback 功能至 GATTC
    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);
}
```

⚠️ **注意：**

GATTC 代码相关 API 说明可以参考 [ESP_IDF gatt_client demo](#) 中的 [gatt_client_example_walkthrough.md](#)。

3.2. 连接

DEV_A 在完成初始化后，开始搜索经典蓝牙设备，搜索到设备后会在 ***esp_bt_gap_cb*** 函数中回调 ***ESP_BT_GAP_DISC_RES_EVT***，符合条件后调用 ***esp_spp_start_discovery(peer_bd_addr)*** 连接设备。设备连接成功后会在 ***esp_spp_cb*** 函数中回调 ***ESP_SPP_OPEN_EVT***，在此事件中开始发送 SPP 数据，计算 SPP 速率，并开始搜索 BLE 设备。

BLE 搜索到广播后会回调 ***esp_gap_cb*** 函数中的 ***ESP_GAP_SEARCH_INQ_RES_EVT*** 事件，找到符合条件的设备后使用 ***esp_ble_gattc_open()*** 连接，连接成功后会回调 ***gattc_profile_event_handler*** 中的 ***ESP_GATTC_CONNECT_EVT*** 事件。BLE 连接成功后会注册对端设备 GATT notification，为后续的数据发送做准备。

3.3. 数据发送与接收

DEV_A SPP 连接以后使用 ***esp_spp_write()*** 函数发送 SPP 数据并计算速率。

DEV_A BLE 连接成功后，注册对端设备的 GATT notification，并开始监听，接收 GATT notification 数据后会回调 ***gattc_profile_event_handler*** 中的 ***ESP_GATTC_NOTIFY_EVT*** 事件，统计数据长度并计算速率。

DEV_B 的 SPP 被连接以后，等待对端设备发送 SPP 数据。DEV_B BLE 被连接后，DEV_A 会使能 DEV_B 的 GATT notification。DEV_B 在 GATTS init 时初始化了 ***notify_task***，当 ***notify_task*** 检测到 GATT notification 被使能后，调用 ***esp_ble_gatts_send_indicate()*** 发送 GATT notification 数据。

3.4. 性能说明

BT SPP 和 BLE 连接成功以后会自动发送数据，计算吞吐率。单独运行 BT SPP 的速率为 230 KB/s 左右，单独运行 BLE 的速率为 40 KB/s 左右（优化后有 90 KB/s）。目前，在 BLE 中仅计算了 GATT notification 的速率，使用当前的参数速率 BT SPP 120 KB/s、BLE GATT notification 30 KB/s 左右，BT SPP 和 BLE GATT notification 的速率是可调的。具体可以通过调节 BLE 的连接参数，发送 GATT notification 的长度和频率调整。SPP 和 GATT 的吞吐量 log，请见图 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
```

图 3-1. SPP 和 GATT 的吞吐量 log

乐鑫 IoT 团队

www.espressif.com