BLE 入门示例开发指引

发布 ***1.0.0***

2018 年 **11** 月 **02** 日

目录

CHAPTER 1

文档介绍

1.1 文档目的

通过一个简单 BLE 示例介绍 BLE 应用的基本开发流程.

1.2 术语说明

表 1.1: 术语说明

术语	说明
DATS	一个自定义的 GATT Service，用于数据的接收和发送
BLE	蓝牙低功耗技术

1.3 参考文档

- http://docs.zephyrproject.org/

1.4 版本历史

表 1.2: 版本历史

日期	版本	注释	作者
2018-08-22	1.0	初始版本	ZS110A 项目组

CHAPTER 2

应用开发流程介绍

2.1 应用开发框架简介

应用示例是协议栈 profile 软件系统的一部分，如下图所所示

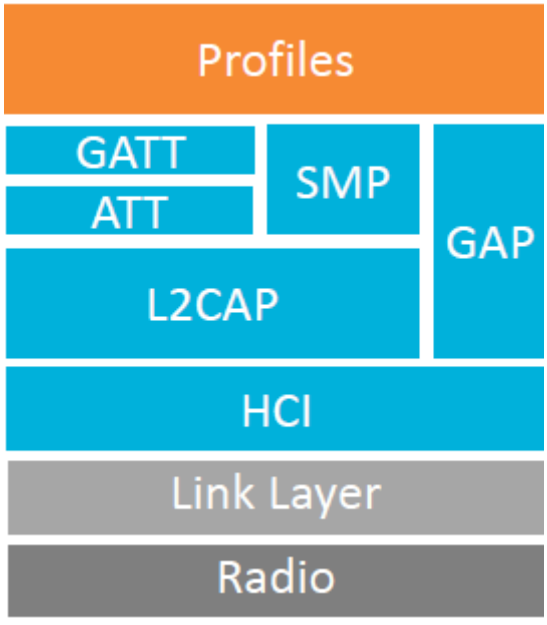

图 2.1: 应用示例层次框图

应用示例开发需要基于 profiles 和 Services 进行开发，profile 和 service 主要提供蓝牙规范要求的顶层协议接口，同时还需要基于应用框架层进行开发，应用框架层主要提供连接和设备管理，用户接口，设备数据库以及硬件 sensor 接口。

2.2 应用开发流程

BLE 应用示例可以按照以下流程来开发：

- BLE 参数配置：数据结构主要用于控制广播、安全和连接的行为。
- GATT 属性协议数据：数据结构和常量用于给 ATT client 和 sever 配置服务发现和管理 client 属性配置描述符（CCCD）
- 协议栈和应用框架层回调函数注册：这些函数对接协议栈和应用框架层事件到示例应用的事件处理函数

CHAPTER 3

DATS 应用示例代码简介

该部分主要介绍 DATS 应用示例工作流程的一些细节。

DATS 的工程路径在：\samples\bluetooth\peripheral_dats\

3.1 BLE 参数配置

3.1.1 广播参数配置

```
#define BT_LE_ADV_PARAM(_options, _int_min, _int_max) \
                (&(struct bt_le_adv_param) { \
                        .options = (_options), \
                        .interval_min = (_int_min), \
                        .interval_max = (_int_max), \
                 })

#define BT_LE_ADV_CONN BT_LE_ADV_PARAM(BT_LE_ADV_OPT_CONNECTABLE,
↪ \
                                                BT_GAP_ADV_FAST_INT_
↪MIN_2, \
                                                BT_GAP_ADV_FAST_INT_
↪MAX_2)
```

- options 配置
  - BT_LE_ADV_OPT_CONNECTABLE，连接断开后会继续广播。
  - BT_LE_ADV_OPT_ONE_TIME，连接断开后，不会继续广播。
- interval_min 最小广播间隔，单位为 0.625ms，如果设置为 N，即 N × 0.625 ms

- interval_max 最大广播间隔，单位为 0.625ms，如果设置为 N，即 N × 0.625 ms

3.1.2 安全参数配置

```
static struct appSecCfg_t datsSecCfg = {
        BT_SMP_AUTH_BONDING | BT_SMP_AUTH_MITM,
        BT_SMP_DIST_ID_KEY,
        BT_SMP_DIST_ENC_KEY | BT_SMP_DIST_ID_KEY,
        BT_SMP_OOB_NOT_PRESENT,
        false
};
```

- auth 授权方式配置
- iKeyDist 发起方需要发布的 Key
- rKeyDist 应答方需要发布的 key
- oob 是否存在带外，当前协议栈只能配置为不存在
- initiateSec slave 是否发起安全设置请求

3.1.3 连接参数配置

```
static struct bt_le_conn_param datsUpdateCfg = {
        .interval_min = (BT_GAP_INIT_CONN_INT_MIN),
        .interval_max = (BT_GAP_INIT_CONN_INT_MAX),
        .latency = (0),
        .timeout = (2000),
};
```

- interval_min 连接间隔的最小值，单位为 1.25ms，即 N * 1.25 ms
- interval_max 连接间隔的最大值，单位为 1.25ms，即 N * 1.25 ms
- latency slave 的潜伏值，是允许设备跳过的最大连接次数。
- timeout 连接超时时间

3.1.4 广播内容设置

```
static const struct bt_data ad[] = {
        BT_DATA_BYTES(BT_DATA_FLAGS, (BT_LE_AD_GENERAL | BT_LE_
↪AD_NO_BREDR)),
        BT_DATA_BYTES(BT_DATA_UUID16_ALL,
                      0x0d, 0x18, 0x0f, 0x18, 0x05, 0x18),
        BT_DATA_BYTES(BT_DATA_UUID128_ALL,
                      0xf0, 0xde, 0xbc, 0x9a, 0x78, 0x56,␣
↪0x34, 0x12,
```

(continues on next page)

(续上页)

```
                                0x78, 0x56, 0x34, 0x12, 0x78, 0x56,␣
↪0x34, 0x12),
};

static const struct bt_data sd[] = {
BT_DATA(BT_DATA_NAME_COMPLETE, DEVICE_NAME, DEVICE_NAME_LEN),
};
```

3.2 ATT 属性协议数据

bt_gatt_service 是 GATT Service 结构体，通常是通过 bt_gatt_service_register API 注册到协议栈。

DATS 应用作为一个 GATT 服务端，首先需要创建一个用于 GATT 通讯的 Service，按照如下代码所示，创建一个用于特殊功能的 dats 服务。

```
static struct bt_gatt_attr wp_attrs[] = {
        /* Vendor Primary Service Declaration */
        BT_GATT_PRIMARY_SERVICE(&wp_uuid),
        BT_GATT_CHARACTERISTIC(&wp_rx_uuid.uuid, BT_GATT_CHRC_
↪NOTIFY),
        BT_GATT_DESCRIPTOR(&wp_rx_uuid.uuid,
        BT_GATT_PERM_READ, NULL, NULL, NULL),
        BT_GATT_CCC(wp_rx_ccc_cfg, wp_rx_ccc_cfg_changed),
        BT_GATT_CHARACTERISTIC(&wp_tx_uuid.uuid,
        BT_GATT_CHRC_WRITE_WITHOUT_RESP),
        BT_GATT_DESCRIPTOR(&wp_tx_uuid.uuid,
        BT_GATT_PERM_WRITE, NULL, write_wp_tx, tx_value),
        BT_GATT_CHARACTERISTIC(&wp_ctrl_uuid.uuid,
        BT_GATT_CHRC_NOTIFY | BT_GATT_CHRC_WRITE_WITHOUT_RESP),
        BT_GATT_DESCRIPTOR(&wp_ctrl_uuid.uuid,
        BT_GATT_PERM_READ | BT_GATT_PERM_WRITE, NULL,
        write_wp_ctrl, &ctrl_value),
        BT_GATT_CCC(wp_ctrl_ccc_cfg, wp_ctrl_ccc_cfg_changed),
};
```

```
static struct bt_gatt_service wp_svc = BT_GATT_SERVICE(wp_attrs);
```

3.3 协议栈回调函数配置

bt_conn_cb 结构体主要用于跟踪连接过程相关状态，它是通过 bt_conn_cb_register() API 进行注册的，在不同的应用实例中，你可能只对部分回调感兴趣，对于不感兴趣的回调可以设置为 NULL。

```
static struct bt_conn_cb conn_callbacks = {
        .connected = connected,
        .disconnected = disconnected,
        .le_param_req = NULL,
        .le_param_updated = le_param_updated,
        .identity_resolved = NULL,
        .security_changed = security_changed,
};
```

- connected 此回调函数通知应用，一个新的连接建立，如 err 参数为非零，则意味着连接建立失败
- disconnected 此回调函数通知应用，连接已经断开，reason 表示断开原因
- le_param_req 此回调函数协议栈通知应用，远端设备请求更新连接参数，应用程序可以通过返回 true 接收参数，返回 false 拒绝接受。
- le_param_updated 此回调函数通知 LE 连接的参数已经被更新，interval、latency、timeout 代表更新的参数
- identity_resolved 此回调函数通知应用层，一个远程设备的身份地址已经被解析。
- security_changed 此回调函数通知应用连接的安全级别发生了变化。

bt_conn_auth_cb 授权配对过程的回调函数结构，主要通过 bt_conn_auth_cb_register API 来注册。

```
static struct bt_conn_auth_cb auth_cb_display = {
        .passkey_display = auth_passkey_display,
        .passkey_entry = NULL,
        .cancel = auth_cancel,
};
```

- passkey_display 如果应用有 IO 用于显示 passkey，可以注册此函数，用于配对过程中显示 passkey.
- passkey_entry 如果应用有 IO 用于输入 passkey，可以注册此函数，用于配对过程中输入 passkey.
- passkey_confirm 如果应用有 IO 用于 yes/no 输入的，可以注册此函数，用于配对过程中 passkey 确认。
- cancel 用于配对过程用户取消配对
- pairing_confirm 配对确认，通常不需要注册

该回调函数的注册，也一定程度告诉协议栈栈，应用程序具备哪些 IO 能力。

- 如果 bt_conn_auth_cb 注册为如下方式，表示应用只有显示 passkey 能力

```
static struct bt_conn_auth_cb auth_cb_display = {
        .passkey_display = auth_passkey_display,
        .passkey_entry = NULL,
        .passkey_confirm = NULL,
```

(continues on next page)

(续上页)

```
        .cancel = auth_cancel,
        .pairing_confirm = auth_pairing_confirm,
};
```

- 如果 bt_conn_auth_cb 注册为如下方式，表示应用有显示 passkey 能力, 同时还具备 YES/NO 输入能力

```
static struct bt_conn_auth_cb auth_cb_display_yes_no = {
        .passkey_display = auth_passkey_display,
        .passkey_entry = NULL,
        .passkey_confirm = auth_passkey_confirm,
        .cancel = auth_cancel,
        .pairing_confirm = auth_pairing_confirm,
};
```

- 如果 bt_conn_auth_cb 注册为如下方式，表示应用有输入 passkey 的能力

```
static struct bt_conn_auth_cb auth_cb_input = {
        .passkey_display = NULL,
        .passkey_entry = auth_passkey_entry,
        .passkey_confirm = NULL,
        .cancel = auth_cancel,
        .pairing_confirm = auth_pairing_confirm,
};
```

- 如果 bt_conn_auth_cb 注册为如下方式，表示应用有显示 passkey 能力，同时还有输入 passkey 的能力

```
static struct bt_conn_auth_cb auth_cb_all = {
        .passkey_display = auth_passkey_display,
        .passkey_entry = auth_passkey_entry,
        .passkey_confirm = auth_passkey_confirm,
        .cancel = auth_cancel,
        .pairing_confirm = auth_pairing_confirm,
};
```

3.4 应用启动流程

```
void app_main(void)
    {
            int err;

            err = bt_set_conn_update_param(K_SECONDS(5), &
↪datsUpdateCfg);

            bt_set_smp_cfg(BT_SMP_MIN_ENC_KEY_SIZE, BT_SMP_MAX_
↪ENC_KEY_SIZE);
```

(continues on next page)

(续上页)

```
            bt_set_sec_cfg(datsSecCfg.auth, datsSecCfg.iKeyDist,␣
↪datsSecCfg.rKeyDist, datsSecCfg.oob);

            err = bt_enable(bt_ready);
            if (err) {
                    printk("Bluetooth init failed (err %d)\n",␣
↪err);
                    return;
            }

            bt_conn_cb_register(&conn_callbacks);
            bt_conn_auth_cb_register(&auth_cb_display);

            /* Implement notification. At the moment there is no␣
↪suitable way
             * of starting delayed work so we do it here
             */
            while (1) {
                    int i;

                    for (i = 0; i < BT_GATT_CCC_MAX; i++) {
                            struct bt_conn *conn;

                            if (wp_rx_ccc_cfg[i].value != BT_
↪GATT_CCC_NOTIFY) {
                                    continue;
                            }

                            conn = bt_conn_lookup_state_le(&wp_
↪rx_ccc_cfg[i].peer, BT_CONN_CONNECTED);
                            if (!conn) {
                                    continue;
                            }

                            /* Vendor rx simulation */
                            wp_rx_notify(conn);

                            bt_conn_unref(conn);
                    }
            }
    }

    static void bt_ready(int err)
    {
            if (err) {
                    printk("Bluetooth init failed (err %d)\n",␣
↪err);
                    return;
```

(continues on next page)

(续上页)

```
            }

            printk("Bluetooth initialized\n");
            bt_gatt_service_register(&wp_svc);

            err = bt_le_adv_start(BT_LE_ADV_CONN, ad, ARRAY_
↪SIZE(ad),
                                  sd, ARRAY_SIZE(sd));
            if (err) {
                    printk("Advertising failed to start (err
↪%d)\n", err);
                    return;
            }

            printk("Advertising successfully started\n");
    }
```

- app_main 为应用入口函数
- bt_set_conn_update_param 配置应用期望的连接参数给协议栈
- bt_set_smp_cfg 和 bt_set_sec_cfg 配置安全参数给协议栈
- bt_enable(bt_ready) 初始化并使能 BLE 协议栈
- bt_ready 当 BLE 使能后，就会调用此回调函数
- bt_gatt_service_register 注册应用 profile 到协议栈
- bt_le_adv_start 开始发送广播
- bt_conn_cb_register 注册连接相关回调函数
- bt_conn_auth_cb_register 注册安全相关回调函数
- while (1) {} 应用进入主循环，等待消息处理

CHAPTER 4

DATS 应用功能演示

4.1 数据接收和发送功能展示

- 手机端发送数据给设备端
  - 将示例代码烧写到开发板
  - 设备端 DATS 应用启动完成后，会发送广播
  - 在 Android 手机端安装上 Actions 的 bletest APK，启动 APK，然后点击“Start Scan”并连接上。
  - 在 Actions Transmission Service 上，点击“SEND FILE”，然后选择一个文件，点击“START SENDING”
  - 这样，数据就不停的从手机端通过 BLE 传送到设备端。
  - 手机端和设备端看到的效果如下面 2 个图所示

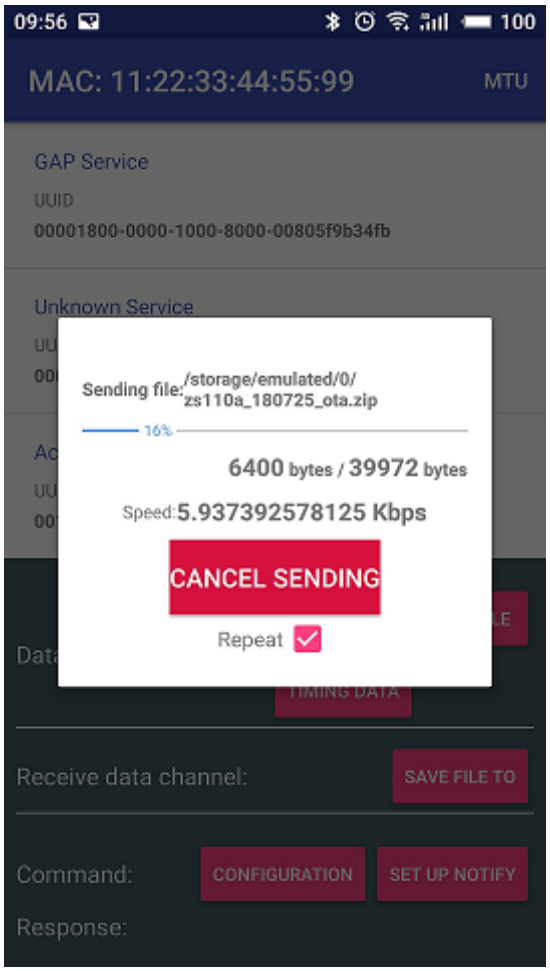

图 4.1: 手机端数据发送示意图

图 4.2: 设备端接收数据打印示意图

注解: Actions APK 路径: \scripts\support\actions\utils\android_apk\

- 手机从设备端接收数据
  - 将示例代码烧写到开发板
  - 设备 DATS 应用启动完成后，会发送广播
  - 在 Android 手机端安装上 Actions 的 bletest APK，启动 APK，然后点击“Start Scan”并连接上。
  - 在 Actions Transmission Service 上，点击“SAVE FILE TO”，然后选择一个文件用于保存数据
  - 这样，数据就会不停的从设备端传送到手机端。
  - 手机端和设备端看到的效果如下面 2 个图所示

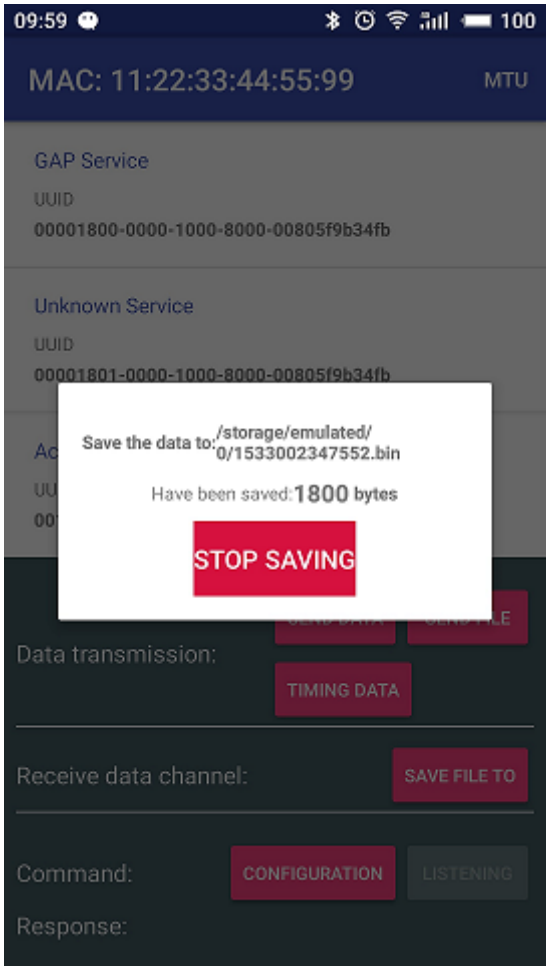

图 4.3: 手机端数据接收示意图

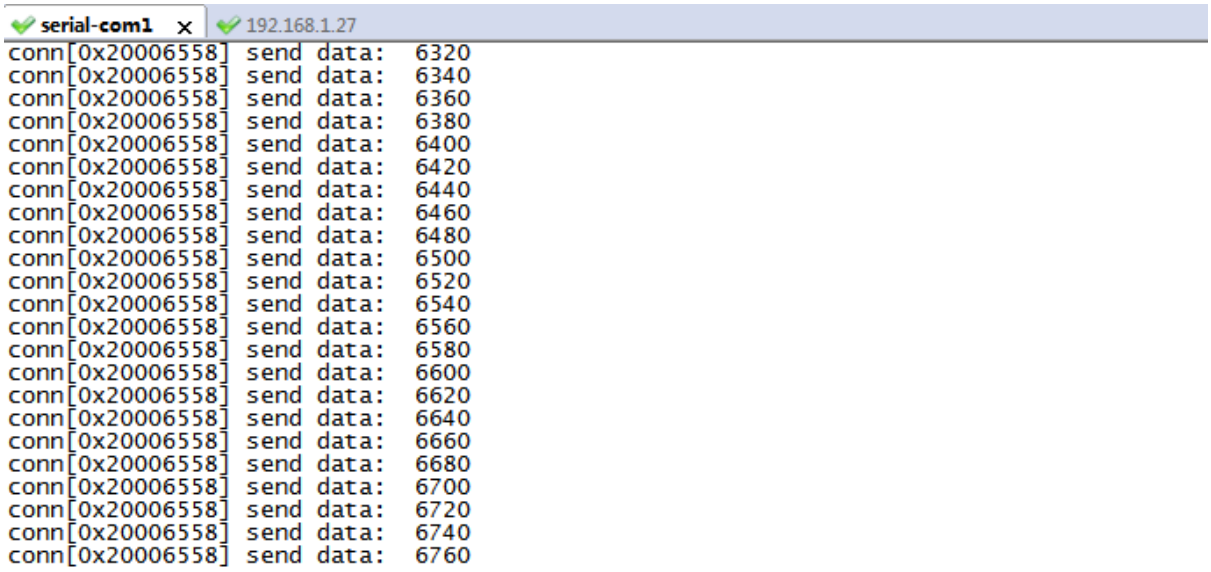

图 4.4: 设备端发送数据打印示意图

注解: Actions APK 路径: \scripts\support\actions\utils\android_apk\

List of Figures

List of Tables