Bluetooth LE Stack User Manual

V 1.0.4

2020/6/30

Revision History

Date	Version	Comments	Author	Reviewer
2018/02/02	V1.0.0	Formal version	Jane	
2018/05/14	V1.0.1	Formal version	Berni	Jane
2018/09/12	V1.0.2	Formal version	Berni	Jane
2019/02/25	V1.0.3	Add Bluetooth LE Privacy Chapter Add BLE Peripheral Privacy Application Add Request Pairing Chapter Add Supported BT Features Chapter Delete DTM Chapter	Jane	Berni
2020/06/23	V1.0.4	Remove Project Overview Chapter to Bluetooth LE Sample Project User Manual	Carrie	

Contents

Table List

Figure List

Glossary

Terms	Definitions
ATT	Attribute protocol
LE	Low Energy
DTM	Direct Test Mode
GAP	Generic Access Profile
GATT	Generic Attribute Profile
L2CAP	Logical Link Control and Adaptation protocol
SDK	Software Development Kit
SMP	Security Manager protocol
SOC	System on Chip

1 Overview

The Realtek Software Development Kit (SDK) provides software documentation including stack/profiles, reference material, example profiles and user applications, aiming to help with product development using Realtek Series of System on Chip (SOC) devices.

SDK facilitates quick development of Bluetooth Low Energy (LE) application. Profile is one of the modules constituting SDK, which packages underlying implementation details for Low Energy protocol stack, and provides user-friendly and easy-to-use interfaces for use in development of application.

The purpose of this document is to give an overview of Bluetooth LE Stack Interfaces. Bluetooth LE Stack Interfaces can be divided into Generic Access Profile (GAP) interfaces and Generic Attribute Profile (GATT) based profile interfaces.

Architecture of Profile and Bluetooth LE Stack in SDK is showed in Figure 1-1.

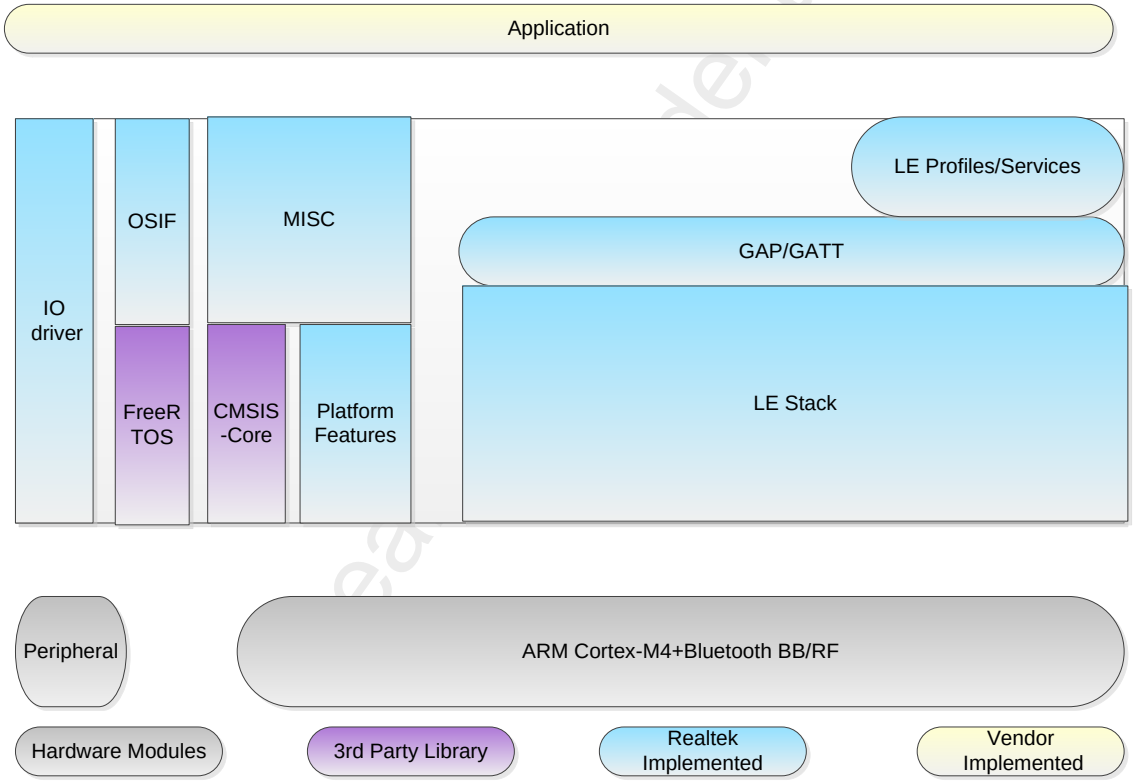

Figure 1-1 SW Architecture

1.1 Supported Bluetooth Technology Features

Table 1-1 Supported Bluetooth Technology Features

Spec Version	Bluetooth technology Feature	Macro Definition	Remark
Bluetooth 4.0	Advertiser		
	Scanner	F_BT_LE_GAP_SCAN_SUPPORT	
	Initiator		
	Master	F_BT_LE_GAP_CENTRAL_SUPPORT	
	Slave	F_BT_LE_GAP_PERIPHERAL_SUPPORT	
Bluetooth 4.1	Low Duty Cycle Directed Advertising		
	LE L2CAP Connection Oriented Channel		
	LE Scatternet		
	LE Ping		
Bluetooth 4.2	LE Data Packet Length Extension	F_BT_LE_4_2_DATA_LEN_EXT_SUPPORT	
	LE Secure Connections	F_BT_LE_4_2_SC_SUPPORT	
	Link Layer Privacy (Privacy1.2)	F_BT_LE_PRIVACY_SUPPORT	
	Link Layer Extended Filter Policies	F_BT_LE_PRIVACY_SUPPORT	
Bluetooth 5	2 Msym/s PHY for LE	F_BT_LE_5_0_SUPPORT	
	LE Long Range	F_BT_LE_5_0_SUPPORT	
	High Duty Cycle Non-Connectable Advertising		
	LE Advertising Extensions	F_BT_LE_5_0_AE_ADV_SUPPORT F_BT_LE_5_0_AE_SCAN_SUPPORT	
	LE Channel Selection Algorithm #2		

Information about LE Link number please refers to *Configuration of LE Link number*.

Compatibility of LE Advertising Extensions in Bluetooth Technology 5 please refers to *BLE BT5 Peripheral Application* and *BLE BT5 Central Application*.

For the comparison of macro definition and reference API, please refer to **Table 1-2**.

Table 1-2 Macro Definition and Reference API

Spec Version	Macro Definition	Feature or Reference API
	F_BT_LE_GAP_CENTRAL_SUPPORT	
	F_BT_LE_GAP_SCAN_SUPPORT	gap_scan.h
	F_BT_LE_GAP_SCAN_FILTER_SUPPORT	
	F_BT_LE_GAP_PERIPHERAL_SUPPORT	
	F_BT_LE_GATT_CLIENT_SUPPORT	profile_client.h
	F_BT_LE_GATT_SERVER_SUPPORT	profile_server.h
	F_BT_LE_SMP_OOB_SUPPORT	le_bond_oob_input_confirm
Bluetooth 4.0	F_BT_LE_READ_REMOTE_FEATS	
	F_BT_LE_ATT_SIGNED_WRITE_SUPPORT	
	F_BT_LE_4_0_DTM_SUPPORT	gap_dtm.h
	F_BT_LE_READ_ADV_TX_POWRE_SUPPORT	le_adv_read_tx_power
	F_BT_LE_READ_CHANN_MAP	le_read_chann_map
	F_BT_LE_READ_REMOTE_VERSION_INFO_SUPPORT	le_read_remote_version
Bluetooth 4.2	F_BT_LE_4_2_SC_SUPPORT	le_bond_user_confirm
	F_BT_LE_4_2_SC_OOB_SUPPORT	le_bond_sc_local_oob_init le_bond_sc_peer_oob_init
	F_BT_LE_4_2_KEY_PRESS_SUPPORT	le_bond_keypress_notify
	F_BT_LE_4_2_DATA_LEN_EXT_SUPPORT	le_set_data_len le_write_default_data_len
	F_BT_LE_PRIVACY_SUPPORT	gap_privacy.h
Bluetooth 5.0	F_BT_LE_5_0_AE_ADV_SUPPORT	gap_ext_adv.h
	F_BT_LE_5_0_AE_SCAN_SUPPORT	gap_ext_scan.h
	F_BT_LE_5_0_DTM_SUPPORT	gap_dtm.h
	F_BT_LE_5_0_SET_PHYS_SUPPORT	le_set_phy
	F_BT_LE_5_0_READ_POWER_SUPPORT	
Function Cconfiguration Flags	F_BT_LE_FIX_CHANN_SUPPORT	
	F_BT_LE_LOCAL_IRK_SETTING_SUPPORT	flash_save_local_irk flash_load_local_irk
	F_BT_OOB_SUPPORT	

1.2 Bluetooth LE Profile Architecture

Definition of profile in Bluetooth specification is different from that of Protocol. Protocol is defined as layer protocols in Bluetooth specification such as Link Layer, Logical Link Control and Adaptation protocol (L2CAP), Security Manager protocol (SMP), and Attribute protocol (ATT), while Profile involves implementation of interoperability of Bluetooth applications from the perspective of how to use layer protocols in Bluetooth specification. Profile defines features and functions that are available in Protocol, and implementation of interaction details between devices, so as to accommodate Bluetooth protocol stack to application development in various scenarios.

The relationship between Profile and Protocol in Bluetooth specification is shown in Figure 1-2.

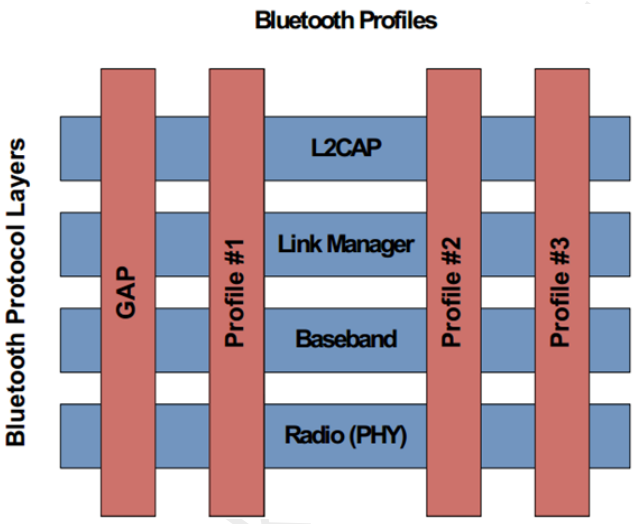

Figure 1-2 Bluetooth Profiles

As shown in Figure 1-2, Profile is illustrated in red rectangular, GAP, Profile #1, Profile #2, and Profile #3. Profiles in Bluetooth specification are classified into two types - GAP and GATT Based Profile (Profile #1, Profile #2 and Profile #3).

1.2.1 GAP

GAP is basic Profile which must be implemented by all Bluetooth devices, and used to describe actions and methods including device discovery, connection, security requirement, and authentication. GAP for Bluetooth Low Energy also defines 4 application roles - Broadcaster, Observer, Peripheral and Central - for optimization in various application scenarios.

Broadcaster is applicable to applications sending data only via broadcast. Observer is applicable to applications receiving data via broadcast. Peripheral is applicable to applications setting link connection. Central is applicable to applications setting a single or multiple link connections.

1.2.2 GATT Based Profile

In Bluetooth specification, another commonly used Profile is GATT Based Profile. GATT is a standard based on server-client interaction defined in Bluetooth specification, and is used to implement provision of service data and access to service data. GATT Based Profile is a standard which is defined based on server-client interaction to meet various application cases and used for data interaction between devices as specified. Profile is made up in the form of Service and Characteristic, as shown in Figure 1-3.

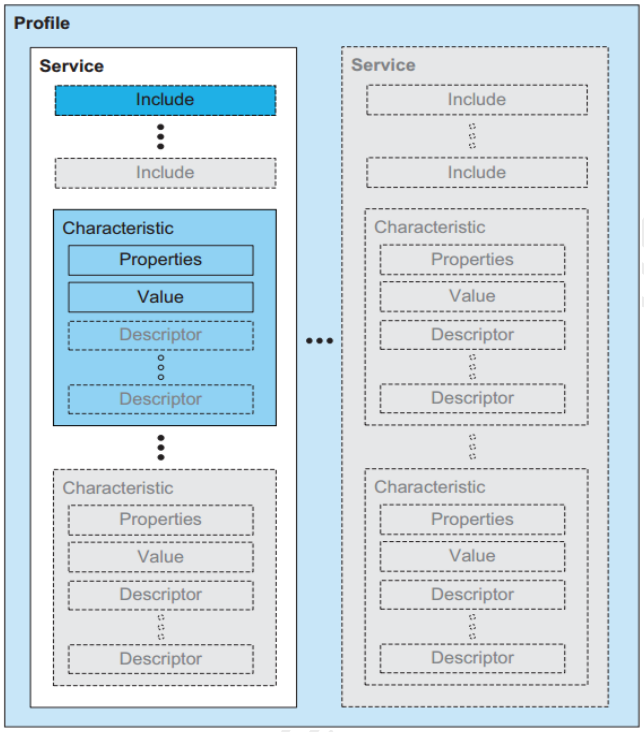

Figure 1-3 GATT Based Profile Hierarchy

2 GAP

GAP is basic Profile which must be implemented by all Bluetooth devices, and is used to describe actions and methods including device discovery, connection, security requirement, and authentication.

GAP Layer has been implemented in ROM, and provides interfaces to application. Header files are provided in SDK.

GAP header files directory: sdk\inc\bluetooth\gap.

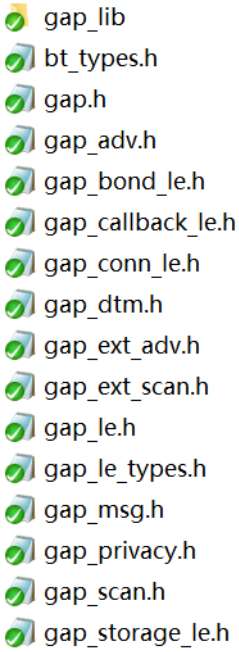

Figure 2-1 GAP Header Files

GAP layer will be introduced according to the following several parts:

- GAP layer structure will be introduced in chapter *GAP Structure Overview*.
- Configuration of GAP parameters and GAP internal startup flow please refer to chapter *GAP Initialization and Startup Flow.*
- GAP message type definitions and GAP Message processes flow please refer to chapter *Bluetooth LE GAP Message*.
- GAP message callback function is used by GAP Layer to send messages to application, more information about GAP message callback please refer to chapter *Bluetooth LE GAP Callback.*
- How to use GAP interfaces please refer to chapter *Bluetooth LE GAP Use Case.*
- Local stack information and bonding device information storage implemented by GAP layer will be introduced in chapter *GAP Information Storage*.

2.1 GAP Structure Overview

2.1.1 GAP Location

GAP is one part of Bluetooth protocol stack, as shown in Figure 2-2, protocol stack is surrounded in dashed box. On top of protocol stack is application, and baseband / RF located beneath protocol stack. GAP provides interfaces for application to access upper stack.

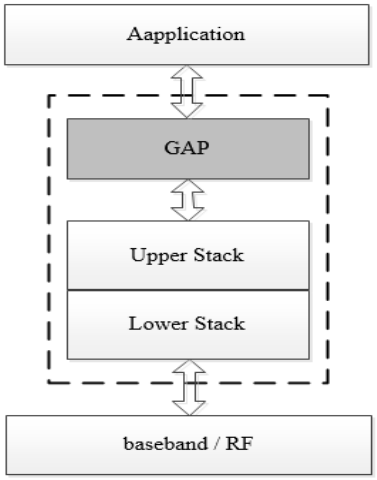

Figure 2-2 GAP Location in SDK

2.1.2 GAP Capacity

The capacity provided by GAP API is as below:

1. **Advertising related**

   Including set / get advertising parameters, start / stop advertising.

2. **Scan related**

   Including set / get scan parameters, start / stop scan.

3. **Connection related**

   Including set connection parameters, create a connection, terminate the existing connection, and update connection parameter.

4. **Pairing related**

   Including set pairing parameters, trigger pairing procedure, input / display passkey using passkey entry, delete keys of bonded device.

5. **Key management**

   Including find key entry by device address and address type, save / load keys about bond information, resolve random address.

6. **Others**

   - Set GAP common parameters including device appearance, device name, etc.
   - Get maximum supported Bluetooth LE link number.
   - Modify local white list.
   - Generate/set local random address
   - Configure local identity address.
   - Etc.

APIs don't support multiple threads, operations of calling APIs and handling message must be in the same task. APIs supplied in SDK can be divided into synchronous API and asynchronous API. The result of synchronous API is represented by return value, such as le_adv_set_param(). If return value of le_adv_set_param() is GAP_CAUSE_SUCCESS, APP sets a GAP advertising parameter successfully. The result of asynchronous API is notified by GAP message, such as le_adv_start(). If return value of le_adv_start() is GAP_CAUSE_SUCCESS, request of starting advertising has been sent successfully. The result of starting advertising is notified by GAP message GAP_MSG_LE_DEV_STATE_CHANGE.

2.1.3 GAP State

GAP State consists of advertising state, scan state, connection state. If LE Advertising Extensions is enabled, extended advertising state is used. Each state has corresponding sub-state, and this part will introduce the state machine of each sub-state.

2.1.3.1 Advertising State

Advertising State has 4 sub-states including idle state, start state, advertising state and stop state. Advertising sub-state is defined in gap_msg.h.

```
/* GAP Advertising State */
#define GAP_ADV_STATE_IDLE                0    // Idle, no advertising
#define GAP_ADV_STATE_START               1    // Start Advertising. A temporary state, haven't received the result.
#define GAP_ADV_STATE_ADVERTISING         2    // Advertising
#define GAP_ADV_STATE_STOP                3    // Stop Advertising. A temporary state, haven't received the result.
```

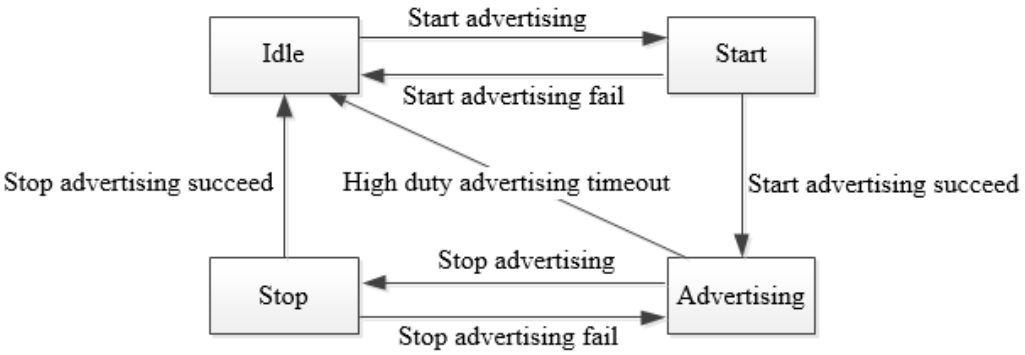

Figure 2-3 Advertising State Transition Machanism

1. **Idle state**

   No advertising, default state.

2. **Start state**

   Start advertising from idle state, but process of enabling advertising hasn't been completed yet. Start state is a temporary state. If advertising is successfully started, then Advertising State will turn into advertising state. Otherwise it will turn back to idle state.

3. **Advertising state**

   Start advertising successfully. In this state, the device is sending advertising packets. If advertising type is high duty cycle directed advertising, Advertising State will change into idle state once high duty cycle directed advertising is timed out.

4. **Stop state**

   Stop advertising from advertising state, but process of disabling advertising hasn't been completed yet. Stop state is a temporary state. If advertising is successfully stopped, Advertising State will turn into idle state. Otherwise Advertising State will turn back to advertising state.

2.1.3.2 Scan State

Scan State has 4 sub-states including idle state, start state, scanning state and stop state. Scan sub-state is defined in gap_msg.h.

```
/* GAP Scan State */
#define GAP_SCAN_STATE_IDLE              0    //Idle, no scanning
#define GAP_SCAN_STATE_START             1    //Start scanning. A temporary state, haven't received the result.
#define GAP_SCAN_STATE_SCANNING          2    //Scanning
#define GAP_SCAN_STATE_STOP              3    //Stop scanning, A temporary state, haven't received the result
```

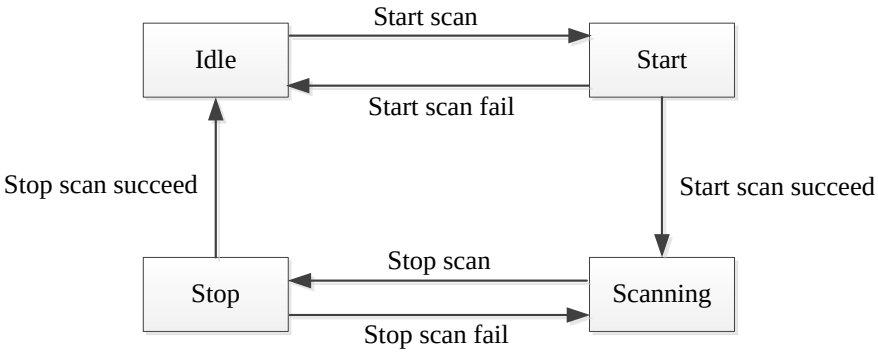

Figure 2-4 Scan State Transition Machanism

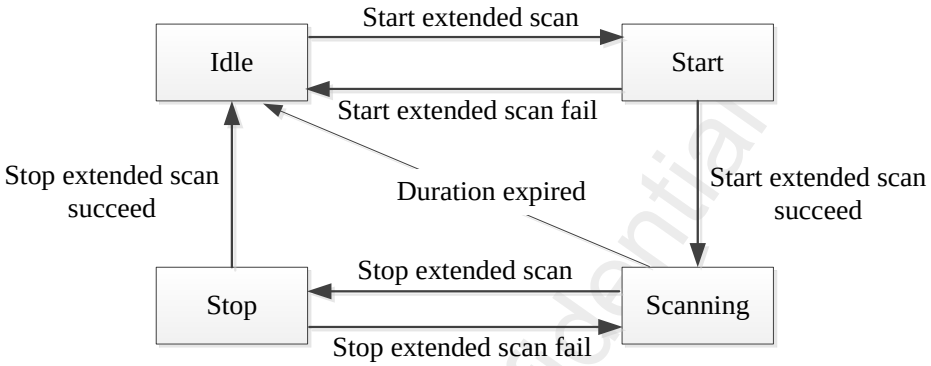

Figure 2-5 Scan State Transition Machanism with Extended Scan

1. **Idle state**

   No scan, default state.

2. **Start state**

   Start scan in idle state, but process of enabling scan hasn't been completed yet. Start state is a temporary state. If scanning is successfully started, Scan State will change to scanning state. Otherwise Scan State will turn back to idle state.

3. **Scanning state**

   Start scan completely. In this state, the device is scanning advertising packets. If extended scan is used and Duration parameter is non-zero and Period parameter is zero, Scan State will enter idle state once duration is expired.

4. **Stop state**

   Stop scan in scanning state, but process of disabling scan hasn't been completed yet. Stop state is a temporary state. If scanning is successfully stopped, Scan State will change to idle state; otherwise Scan State will turn back to scanning state.

2.1.3.3 Connection State

Due to support of multilink, the link could be connected or disconnected when GAP Connection State is idle state,

so the transition of connection state needs to be combined with gap state and link state.

GAP connection sub-state includes idle state, connecting state. GAP connection sub-state is defined in gap_msg.h.

```
#define GAP_CONN_DEV_STATE_IDLE          0    //!< Idle
#define GAP_CONN_DEV_STATE_INITIATING    1    //!< Initiating Connection
```

Note: GAP can only create one link at the same time, which means application cannot create another link when GAP Connection State is in connecting state.

Link sub-state includes disconnected state, connecting state, connected state and disconnecting state. Link sub-state is defined in gap_msg.h.

```
/* Link Connection State */
typedef enum {
    GAP_CONN_STATE_DISCONNECTED, // Disconnected.
    GAP_CONN_STATE_CONNECTING,   // Connecting.
    GAP_CONN_STATE_CONNECTED,    // Connected.
    GAP_CONN_STATE_DISCONNECTING // Disconnecting.
} T_GAP_CONN_STATE;
```

Connection state transition is different between actively creating a connection as master role and passively receiving a connection indication as slave role. This section will describe these two cases separately.

2.1.3.3.1 Active State Change

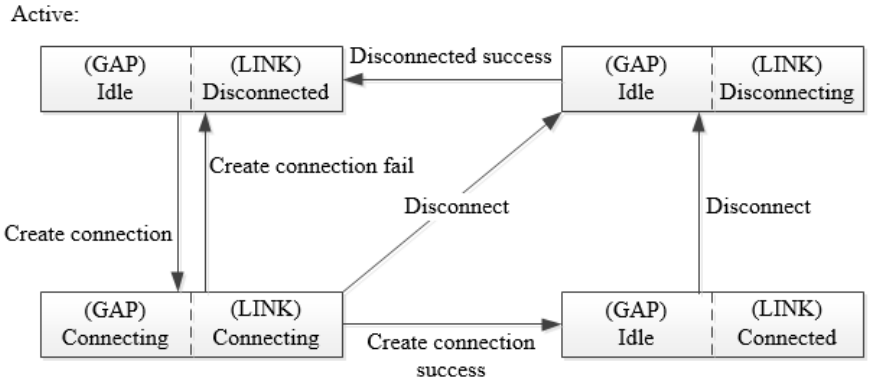

Figure 2-6 Active State Transition Machanism

1. **Idle | Disconnected state**

   GAP Connection State is in idle state, Link State is in disconnected and connection hasn't been established.

2. **Connecting | Connecting state**

   Master creates a connection, and the process hasn't been completed yet. It is a temporary state. GAP Connection State changes to connecting state, and Link State turns to connecting state. If connection is successfully established, Link State will turn to connected state, and GAP Connection State will trun to idle state again. If failing to create connection, Link State will turn back to disconnected state, and GAP Connection State will turn back to idle state. In this state, master can also disconnect the link, if so, Link State will change to disconnecting state and GAP Connection State will turn to idle state.

3. **Idle | Connected state**

   A connection has been created. GAP Connection State is in idle state and Link State is in connected state.

4. **Idle | Disconnecting state**

   Master terminates the link and the process hasn't been completed yet, and it is a temporary state. GAP Connection State is in idle state and Link State is in disconnecting state. If terminate the link successfully, Link State will change to disconnected state.

2.1.3.3.2 Passive State Change

Passive:

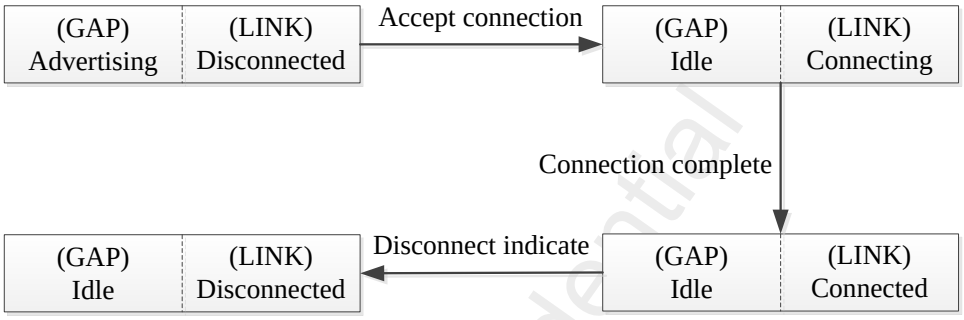

Figure 2-7 Passive State Transition Machanism

1. **Slave accept connection**

   When slave receives connect indication, GAP Advertising State will change to idle state from advertising state and Link State will change to connecting state from disconnected state, after the processs of creating connection has been completed, Link State turns into connected state.

2. **Disconnect by peer**

   When peer device disconnects the link and disconnect indication is received by local device, local device's Link State will change to disconnected state from connected state.

2.1.3.4 Extended Advertising State

Without LE Advertising Extensions, the device uses Advertising State can be considered that there is one simplified advertising set. This section is only applied to device that uses LE Advertising Extensions.

Due to support of LE Advertising Extensions, device may enable or disable one or more advertising sets at a time, so the change of Extended Advertising State has to be combined with advertising set. Extended Advertising State has 4 sub-states including idle state, start state, advertising state and stop state. Extended Advertising sub-state is defined in gap_ext_adv.h.

```
/** @brief GAP extended advertising state. */
typedef enum
{
```

```
    EXT_ADV_STATE_IDLE,          /**< Idle, no advertising. */
    EXT_ADV_STATE_START,         /**< Start Advertising. A temporary state, haven't received the result. */
    EXT_ADV_STATE_ADVERTISING,   /**< Advertising. */
    EXT_ADV_STATE_STOP,          /**< Stop Advertising. A temporary state, haven't received the result. */
} T_GAP_EXT_ADV_STATE;
```

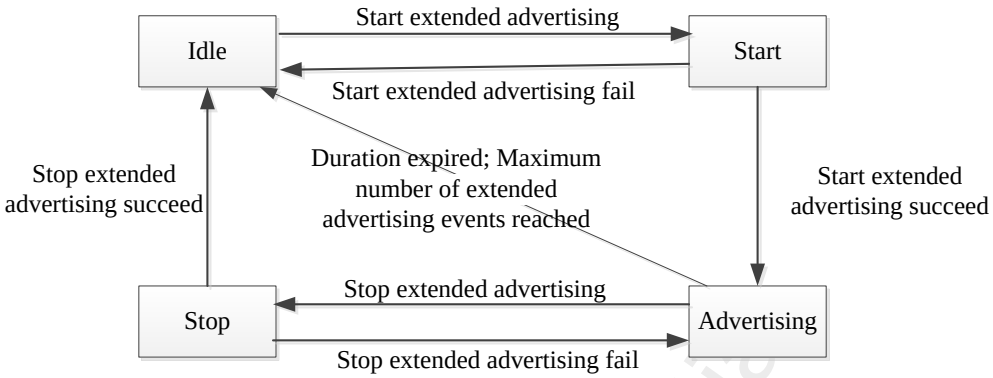

Figure 2-8 Extended Advertising State Transition Machanism for one Advertising Set

1. **Idle state**

   No advertising, default state.

2. **Start state**

   Start extended advertising in idle state, but process of enabling extended advertising has not completed yet. It is a temporary state. If extended advertising is successfully started, Extended Advertising State will turn into advertising state; otherwise Extended Advertising State will turn back to idle state.

3. **Advertising state**

   Start extended advertising sucessfully. In this state, the device is sending advertising packets. If duration is non-zero, Extended Advertising State will turn into idle state once duration is expired. If maximum number of extended advertising events is non-zero, Extended Advertising State will turn into idle state once the maximum number has been reached.

4. **Stop state**

   Stop extended advertising in advertising state, but process of disabling extended advertising hasn't been completed yet. It is a temporary state. If extended advertising is successfully stopped, Extended Advertising State will turn into idle state; otherwise Extended Advertising State will turn back to advertising state.

2.1.4 GAP Message

GAP message includes Bluetooth Technology status message and GAP API message. Bluetooth Technology status message is used to notify APP some information including device state transition, connection state transition, bond state transition etc. GAP API message is used to notify APP that function exec status after the API

has been invoked. Each API has an associated message. More information about GAP message please refers to chapter *Bluetooth LE GAP Message* and chapter *Bluetooth LE GAP Callback*.

2.1.4.1 Bluetooth Technology Status Message

Bluetooth Technology status message is defined in gap_msg.h.

```
/* BT status message */
#define GAP_MSG_LE_DEV_STATE_CHANGE             0x01 // Device state change msg type.
#define GAP_MSG_LE_CONN_STATE_CHANGE            0x02 // Connection state change msg type.
#define GAP_MSG_LE_CONN_PARAM_UPDATE            0x03 // Connection parameter update changed msg type.
#define GAP_MSG_LE_CONN_MTU_INFO                0x04 // Connection MTU size info msg type.
#define GAP_MSG_LE_AUTHEN_STATE_CHANGE          0x05 // Authentication state change msg type.
#define GAP_MSG_LE_BOND_PASSKEY_DISPLAY         0x06 // Bond passkey display msg type.
#define GAP_MSG_LE_BOND_PASSKEY_INPUT           0x07 // Bond passkey input msg type.
#define GAP_MSG_LE_BOND_OOB_INPUT               0x08 // Bond passkey oob input msg type.
#define GAP_MSG_LE_BOND_USER_CONFIRMATION       0x09 // Bond user confirmation msg type.
#define GAP_MSG_LE_BOND_JUST_WORK               0x0A // Bond user confirmation msg type.

#if F_BT_LE_5_0_AE_ADV_SUPPORT
#define GAP_MSG_LE_EXT_ADV_STATE_CHANGE         0x0B // Extended advertising state change msg type.
#endif
```

2.1.4.2 GAP API Message

GAP API message is defined in gap_callback_le.h. Each function-related message please refers to the API comments.

```
/* GAP API message */
#define GAP_MSG_LE_MODIFY_WHITE_LIST            0x01 // response msg type for le_modify_white_list
#define GAP_MSG_LE_SET_RAND_ADDR                0x02 // response msg type for le_set_rand_addr
#define GAP_MSG_LE_SET_HOST_CHANN_CLASSIF       0x03 // response msg type for le_set_host_chann_classif
#define GAP_MSG_LE_WRITE_DEFAULT_DATA_LEN       0x04 // response msg type for
le_write_default_data_len#define GAP_MSG_LE_READ_RSSI                    0x10 // response msg type for
le_read_rssi
#define GAP_MSG_LE_READ_CHANN_MAP               0x11 // response msg type for le_read_chann_map
#define GAP_MSG_LE_DISABLE_SLAVE_LATENCY        0x12 // response msg type for le_disable_slave_latency
#define GAP_MSG_LE_SET_DATA_LEN                 0x13 // response msg type for le_set_data_len
#define GAP_MSG_LE_DATA_LEN_CHANGE_INFO         0x14 // Notification msg type for data length changed
#define GAP_MSG_LE_CONN_UPDATE_IND              0x15 // Indication for le connection parameter update
#define GAP_MSG_LE_CREATE_CONN_IND              0x16 // Indication for create le connection
#define GAP_MSG_LE_PHY_UPDATE_INFO              0x17 // Indication for le phyical update information
#define GAP_MSG_LE_UPDATE_PASSED_CHANN_MAP 0x18 // response msg type for
```

```
le_update_passed_chann_map
#define GAP_MSG_LE_REMOTE_FEATS_INFO            0x19 // Information for remote device supported features
#define GAP_MSG_LE_BOND_MODIFY_INFO             0x20 // Notification msg type for bond modify
#define GAP_MSG_LE_KEYPRESS_NOTIFY              0x21 // response msg type for le_bond_keypress_notify
#define GAP_MSG_LE_KEYPRESS_NOTIFY_INFO         0x22 // Notification msg type forle_bond_keypress_notify
#define GAP_MSG_LE_GATT_SIGNED_STATUS_INFO      0x23 // Notification msg type for le signed status
information
#define GAP_MSG_LE_SCAN_INFO                    0x30 // Notification msg type for le scan
#define GAP_MSG_LE_DIRECT_ADV_INFO              0x31 // Notification msg type for le direct adv info
#define GAP_MSG_LE_ADV_UPDATE_PARAM             0x40 // response msg type for le_adv_update_param
#define GAP_MSG_LE_ADV_READ_TX_POWER            0x41 // response msg type for le_adv_read_tx_power
#if F_BT_LE_5_0_AE_SCAN_SUPPORT
//gap_ext_scan.h
#define GAP_MSG_LE_EXT_ADV_REPORT_INFO      0x50 // Notification msg type for le extended adv report
#endif
#if F_BT_LE_5_0_AE_ADV_SUPPORT
//gap_ext_adv.h
#define GAP_MSG_LE_EXT_ADV_START_SETTING        0x60 // response msg type for le_ext_adv_start_setting
#define GAP_MSG_LE_EXT_ADV_REMOVE_SET           0x61 // response msg type for le_ext_adv_remove_set
#define GAP_MSG_LE_EXT_ADV_CLEAR_SET            0x62 // response msg type for le_ext_adv_clear_set
#define GAP_MSG_LE_EXT_ADV_ENABLE               0x63 // response msg type for le_ext_adv_enable
#define GAP_MSG_LE_EXT_ADV_DISABLE              0x64 // response msg type for le_ext_adv_disable
#define GAP_MSG_LE_SCAN_REQ_RECEIVED_INFO       0x65 // Notification msg type for le scan received info
#endif
```

2.1.5 APP Message Flow

APP message flow is shown in Figure 2-9. Mandatory steps are written in solid line box and optional steps are written in dashed line box.

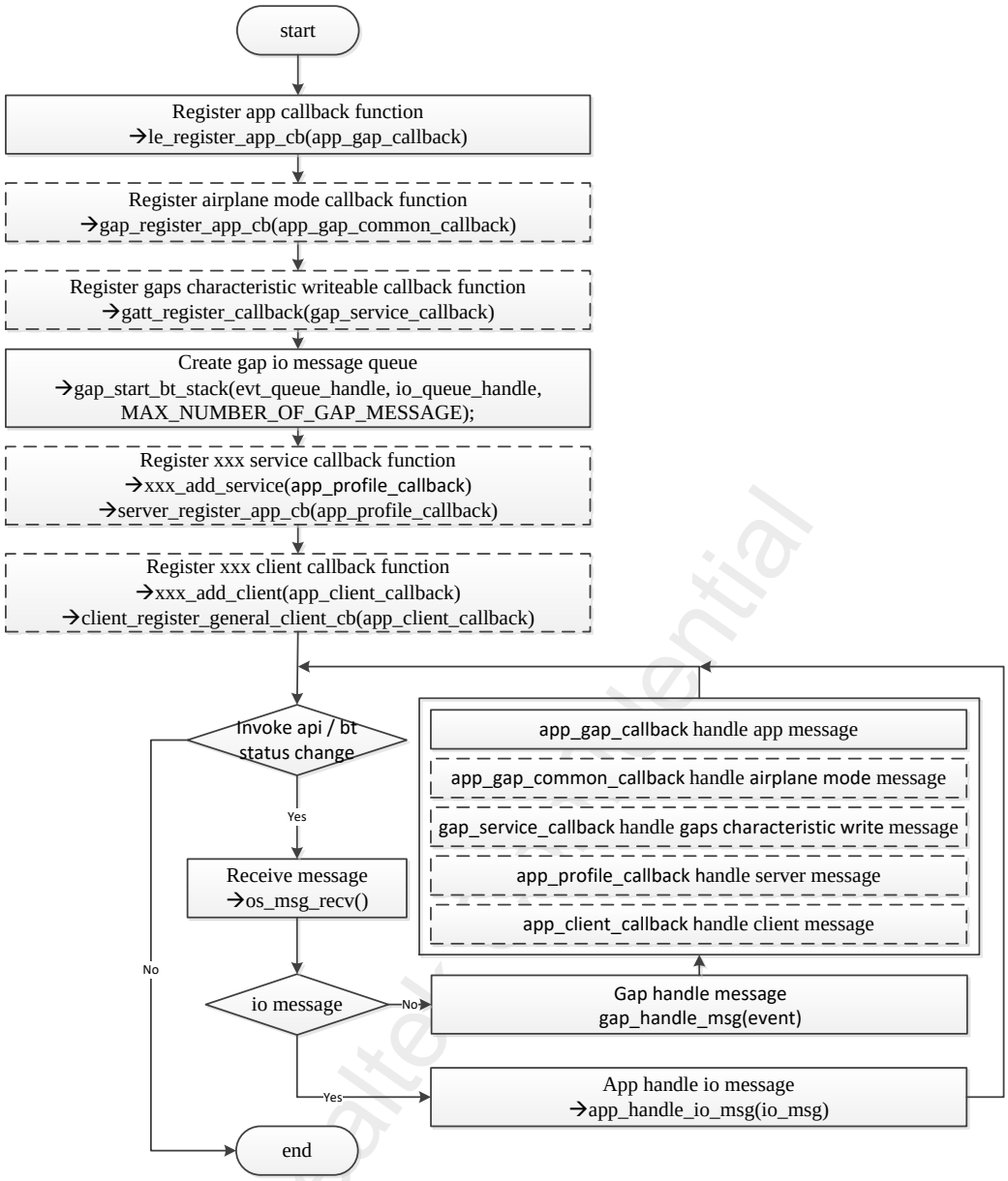

Figure 2-9 APP Message Flow

1. Two methods of sending message to APP

1) Callback function

In this method, APP shall register callback function first. When an upstream message is sent to GAP layer, GAP layer will call this callback function to handle message.

2) Message queue

In this method, APP shall create a message queue first. When an upstream message is sent to GAP layer, GAP layer will send the message into the queue from which APP loops to receive message.

2. Initialization

1) Register callback function

- To receive GAP API messages, APP shall register app callback function by invoking le_register_app_cb() function.
- To receive airplane mode messages, APP shall register app callback function by invoking gap_register_app_cb() function.
- To receive gaps characteristic write messages, APP shall register app callback function by invoking gatt_register_callback() function.
- If peripheral role APP contains services, for receiving server messages, APP shall register service callback function through xxx_add_service() function and server_register_app_cb() function.
- If central role APP contains clients, for receiving client messages, APP shall register client callback function through xxx_add_client() function.

2) Create message queue

For receiving Bluetooth Technology status messages, APP shall create IO message queue through gap_start_bt_stack() function.

3. Loop to receive message

App main task loops to receive messages. If received message is a IO message, APP calls app_handle_io_msg() function to handle this message in APP layer. Otherwise APP invokes gap_handle_msg() function to handle this message in GAP layer.

4. Handle message

If message is sent by callback function, the function registered in initialization procedure shall handle this message. If message is sent by message queue, the message shall be handled by another function.

2.2 GAP Initialization and Startup Flow

This section introduces how to configure LE gap parameters in app_le_gap_init() and gap internal startup flow.

2.2.1 GAP Parameters Initialization

GAP parameter initialization is fulfilled in main.c by modifying codes in function app_le_gap_init().

2.2.1.1 Configure Device Name and Device Appearance

Parameter types are defined in T_GAP_LE_PARAM_TYPE in gap_le.h.

2.2.1.1.1 Device Name Configuration

Device Name Configuration is used to set the value of Device Name Characteristic in GAP Service of the device. If Device Name is set in Advertising Data as well, the Device Name set in Advertising data should have the same value with Device Name Characteristic in GAP Service; otherwise there may be an interoperability problem.

```
/** @brief  GAP - Advertisement data (max size = 31 bytes, best kept short to conserve power) */
static const uint8_t adv_data[] = {
    /* Flags */
    0x02,                   /* length */
    GAP_ADTYPE_FLAGS, /* type="Flags" */
    GAP_ADTYPE_FLAGS_LIMITED | GAP_ADTYPE_FLAGS_BREDR_NOT_SUPPORTED,
    /* Service */
    0x03,                   /* length */
    GAP_ADTYPE_16BIT_COMPLETE,
    LO_WORD(GATT_UUID_SIMPLE_PROFILE),
    HI_WORD(GATT_UUID_SIMPLE_PROFILE),
    /* Local name */
    0x0F,                   /* length */
    GAP_ADTYPE_LOCAL_NAME_COMPLETE,
    'B', 'L', 'E', '_', 'P', 'E', 'R', 'I', 'P', 'H', 'E', 'R', 'A', 'L',
};
void app_le_gap_init(void)
{
    /* Device name and device appearance */
    uint8_t   device_name[GAP_DEVICE_NAME_LEN] = "BLE_PERIPHERAL";
    ......
    /* Set device name and device appearance */
    le_set_gap_param(GAP_PARAM_DEVICE_NAME, GAP_DEVICE_NAME_LEN, device_name);
    ......
}
```

Currently, the maximum length of Device Name character string that Bluetooth Technology Stack supports is 40 bytes (including end mark). If the string exceeds 40 bytes, it will be cut off.

```
#define GAP_DEVICE_NAME_LEN                (39+1)//!< Max length of device name, if device name length exceeds it, it will be truncated.
```

2.2.1.1.2 Device Appearance Configuration

It is used to set the value of Device Appearance Characteristic in GAP Service for the device. If Device Appearance is also set in Advertising data, the Device Appearance set in Advertising data should have the same value with Device Appearance Characteristic in GAP Service; otherwise there may be an interoperability problem. Device Appearance is used to describe the type of a device, such as keyboard, mouse, thermometer, blood pressure meter etc. Available values are defined in gap_le_types.h.

```
/** @defgroup GAP_LE_APPEARANCE_VALUES GAP Appearance Values
  * @{
  */
#define GAP_GATT_APPEARANCE_UNKNOWN                                  0
#define GAP_GATT_APPEARANCE_GENERIC_PHONE                            64
#define GAP_GATT_APPEARANCE_GENERIC_COMPUTER                          128
#define GAP_GATT_APPEARANCE_GENERIC_WATCH                            192
#define GAP_GATT_APPEARANCE_WATCH_SPORTS_WATCH                         193
```

Sample code is shown as below.

```
/** @brief    GAP - scan response data (max size = 31 bytes) */
static const uint8_t scan_rsp_data[] = {
      0x03,                                   /* length */
      GAP_ADTYPE_APPEARANCE,                  /* type="Appearance" */
      LO_WORD(GAP_GATT_APPEARANCE_UNKNOWN),
      HI_WORD(GAP_GATT_APPEARANCE_UNKNOWN),
};
void app_le_gap_init(void)
{
      /* Device name and device appearance */
      uint16_t appearance = GAP_GATT_APPEARANCE_UNKNOWN;
      ......
      /* Set device name and device appearance */
      le_set_gap_param(GAP_PARAM_APPEARANCE, sizeof(appearance), &appearance);
      ......
}
```

2.2.1.2 Configure Advertising Parameters

Advertising parameter types are defined in T_LE_ADV_PARAM_TYPE in gap_adv.h.

Advertising parameters which can be customized are listed below:

```
void app_le_gap_init(void)
{
      /* Advertising parameters */
      uint8_t    adv_evt_type = GAP_ADTYPE_ADV_IND;
      uint8_t    adv_direct_type = GAP_REMOTE_ADDR_LE_PUBLIC;
      uint8_t    adv_direct_addr[GAP_BD_ADDR_LEN] = {0};
      uint8_t    adv_chann_map = GAP_ADVCHAN_ALL;
      uint8_t    adv_filter_policy = GAP_ADV_FILTER_ANY;
      uint16_t adv_int_min = DEFAULT_ADVERTISING_INTERVAL_MIN;
      uint16_t adv_int_max = DEFAULT_ADVERTISING_INTERVAL_MAX;
      ......
      /* Set advertising parameters */
```

```
    le_adv_set_param(GAP_PARAM_ADV_EVENT_TYPE, sizeof(adv_evt_type), &adv_evt_type);
    le_adv_set_param(GAP_PARAM_ADV_DIRECT_ADDR_TYPE, sizeof(adv_direct_type), &adv_direct_type);
    le_adv_set_param(GAP_PARAM_ADV_DIRECT_ADDR, sizeof(adv_direct_addr), adv_direct_addr);
    le_adv_set_param(GAP_PARAM_ADV_CHANNEL_MAP, sizeof(adv_chann_map), &adv_chann_map);
    le_adv_set_param(GAP_PARAM_ADV_FILTER_POLICY, sizeof(adv_filter_policy), &adv_filter_policy);
    le_adv_set_param(GAP_PARAM_ADV_INTERVAL_MIN, sizeof(adv_int_min), &adv_int_min);
    le_adv_set_param(GAP_PARAM_ADV_INTERVAL_MAX, sizeof(adv_int_max), &adv_int_max);
    le_adv_set_param(GAP_PARAM_ADV_DATA, sizeof(adv_data), (void *)adv_data);
    le_adv_set_param(GAP_PARAM_SCAN_RSP_DATA, sizeof(scan_rsp_data), (void *)scan_rsp_data);
}
```

Parameter adv_evt_type defines type of advertising, and different types of advertising needs different parameters, as listed in Table 2-1.

Table 2-1 Advertising Parameters Setting

adv_evt_type	GAP_ADTYPE_ADV_IND	GAP_ADTYPE_ADV_HDC_DIRECT_IND	GAP_ADTYPE_ADV_SCAN_IND	GAP_ADTYPE_ADV_NONCONN_IND	GAP_ADTYPE_ADV_LDC_DIRECT_IND
adv_int_min	Y	Ignore	Y	Y	Y
adv_int_max	Y	Ignore	Y	Y	Y
adv_direct_type	Ignore	Y	Ignore	Ignore	Y
adv_direct_addr	Ignore	Y	Ignore	Ignore	Y
adv_chann_map	Y	Y	Y	Y	Y
adv_filter_policy	Y	Ignore	Y	Y	Ignore
allow establish link	Y	Y	N	N	Y

2.2.1.3 Configure Scan Parameters

Types of scan parameter are defined in T_LE_SCAN_PARAM_TYPE in gap_scan.h.

Scan parameters which can be customized are listed below:

```
void app_le_gap_init(void)
{
    /* Scan parameters */
    uint8_t  scan_mode = GAP_SCAN_MODE_ACTIVE;
    uint16_t scan_interval = DEFAULT_SCAN_INTERVAL;
    uint16_t scan_window = DEFAULT_SCAN_WINDOW;
    uint8_t  scan_filter_policy = GAP_SCAN_FILTER_ANY;
    uint8_t  scan_filter_duplicate = GAP_SCAN_FILTER_DUPLICATE_ENABLE;
```

```
    ......
    /* Set scan parameters */
    le_scan_set_param(GAP_PARAM_SCAN_MODE, sizeof(scan_mode), &scan_mode);
    le_scan_set_param(GAP_PARAM_SCAN_INTERVAL, sizeof(scan_interval), &scan_interval);
    le_scan_set_param(GAP_PARAM_SCAN_WINDOW, sizeof(scan_window), &scan_window);
    le_scan_set_param(GAP_PARAM_SCAN_FILTER_POLICY, sizeof(scan_filter_policy),
                      &scan_filter_policy);
    le_scan_set_param(GAP_PARAM_SCAN_FILTER_DUPLICATES, sizeof(scan_filter_duplicate),
                      &scan_filter_duplicate);
}
```

Parameter Description:

1. *scan_mode* - T_GAP_SCAN_MODE
2. *scan_interval* - Scan Interval, range: 0x0004 to 0x4000 (units of 625us)
3. *scan_window* - Scan window, range: 0x0004 to 0x4000 (units of 625us)
4. *scan_filter_policy* - T_GAP_SCAN_FILTER_POLICY
5. *scan_filter_duplicate* - T_GAP_SCAN_FILTER_DUPLICATE

   Determine whether to filter duplicated advertising data. When the parameter scan_filter_policy is set to GAP_SCAN_FILTER_DUPLICATE_ENABLE, the duplicated advertising data will be filtered in the stack, and will not report to application.

2.2.1.4 Configure Bond Manager Parameters

A part of parameter types are defined in T_GAP_PARAM_TYPE in gap.h.

The others are defined in T_LE_BOND_PARAM_TYPE in gap_bond_le.h.

Bond manager parameters which can be customized are listed below:

```
void app_le_gap_init(void)
{
    /* GAP Bond Manager parameters */
    uint8_t  auth_pair_mode = GAP_PAIRING_MODE_PAIRABLE;
    uint16_t auth_flags = GAP_AUTHEN_BIT_BONDING_FLAG;
    uint8_t  auth_io_cap = GAP_IO_CAP_NO_INPUT_NO_OUTPUT;
    uint8_t  auth_oob = false;
    uint8_t  auth_use_fix_passkey = false;
    uint32_t auth_fix_passkey = 0;
    uint8_t  auth_sec_req_enable = false;
    uint16_t auth_sec_req_flags = GAP_AUTHEN_BIT_BONDING_FLAG;
    ......
    /* Setup the GAP Bond Manager */
    gap_set_param(GAP_PARAM_BOND_PAIRING_MODE, sizeof(auth_pair_mode), &auth_pair_mode);
    gap_set_param(GAP_PARAM_BOND_AUTHEN_REQUIREMENTS_FLAGS, sizeof(auth_flags), &auth_flags);
```

```
    gap_set_param(GAP_PARAM_BOND_IO_CAPABILITIES, sizeof(auth_io_cap), &auth_io_cap);
    gap_set_param(GAP_PARAM_BOND_OOB_ENABLED, sizeof(auth_oob), &auth_oob);
    le_bond_set_param(GAP_PARAM_BOND_FIXED_PASSKEY, sizeof(auth_fix_passkey), &auth_fix_passkey);
    le_bond_set_param(GAP_PARAM_BOND_FIXED_PASSKEY_ENABLE, sizeof(auth_use_fix_passkey),
                      &auth_use_fix_passkey);
    le_bond_set_param(GAP_PARAM_BOND_SEC_REQ_ENABLE, sizeof(auth_sec_req_enable),
                      &auth_sec_req_enable);
    le_bond_set_param(GAP_PARAM_BOND_SEC_REQ_REQUIREMENT, sizeof(auth_sec_req_flags),
                      &auth_sec_req_flags);
}
```

Parameter Description:

1. *auth_pair_mode* - Determine whether the device can be paired in current status.
   - GAP_PAIRING_MODE_PAIRABLE: the device can be paired,
   - GAP_PAIRING_MODE_NO_PAIRING: the device cannot be paired.
2. *auth_flags* - A bit field that indicates the requested security properties.
   - GAP_AUTHEN_BIT_NONE
   - GAP_AUTHEN_BIT_BONDING_FLAG
   - GAP_AUTHEN_BIT_MITM_FLAG
   - GAP_AUTHEN_BIT_SC_FLAG
   - GAP_AUTHEN_BIT_KEYPRESS_FLAG
   - GAP_AUTHEN_BIT_FORCE_BONDING_FLAG
   - GAP_AUTHEN_BIT_SC_ONLY_FLAG
3. *auth_io_cap* - T_GAP_IO_CAP, indicate I/O capacity of the device.
4. *auth_oob* - Indicate whether OOB is enabled.
   - true : set OOB flag
   - false : not set OOB flag
5. *auth_use_fix_passkey* - Indicate whether a random passkey or fixed passkey will be used if pairing mode is passkey entry and the local device needs to generate a passkey.
   - true : use fixed passkey
   - false : use random passkey
6. *auth_fix_passkey* - The default value for passkey is used during pairing, which is valid when auth_use_fix_passkey is true.
7. *auth_sec_req_enable* - Determine whether to send SMP security request when connected.
8. *auth_sec_req_flags* - A bit field that indicates the requested security properties.

2.2.1.5 Configure LE Advertising Extensions Paramters

This section is only applied to device which uses LE Advertising Extensions. First, device needs to configure GAP_PARAM_USE_EXTENDED_ADV to use LE Advertising Extensions. Then device configures extended advertising related parameters or/and exended scan related parameters according to GAP role of the device. For detailed information please refers to *BLE BT5 Peripheral Application* and *BLE BT5 Central Application*.

2.2.1.5.1 Configure GAP_PARAM_USE_EXTENDED_ADV

To use LE Advertising Extensions, it is necessary to configure GAP_PARAM_USE_EXTENDED_ADV as true.

```
void app_le_gap_init(void)
{
    /* LE Advertising Extensions parameters */
    bool use_extended = true;
    ……
    /* Use LE Advertising Extensions */
    le_set_gap_param(GAP_PARAM_USE_EXTENDED_ADV, sizeof(use_extended), &use_extended);
    ……
}
```

2.2.1.5.2 Configure Extended Advertising Related Parameters

This section is applied to peripheral role or broadcaster role. For detailed information on extended advertising please refers to *BLE BT5 Peripheral Application*.

2.2.1.5.3 Configure Extended Scan Related Parameters

This section is applied to observer role or central role. For detailed information on extended scan and extended create connection please refers to *BLE BT5 Central Application*.

2.2.1.6 Configure Other Parameters

2.2.1.6.1 Configure GAP_PARAM_SLAVE_INIT_GATT_MTU_REQ

```
void app_le_gap_init(void)
{
    uint8_t    slave_init_mtu_req = false;
    ......
    le_set_gap_param(GAP_PARAM_SLAVE_INIT_GATT_MTU_REQ, sizeof(slave_init_mtu_req),
                     &slave_init_mtu_req);
```

```
    ......
}
```

This parameter is only applied to peripheral role. This parameter determines whether to send exchange MTU request when connected.

2.2.2 GAP Startup Flow

1. Initialize GAP in main()

```
int main(void)
{
    ......
    le_gap_init(APP_MAX_LINKS);
    gap_lib_init();
    app_le_gap_init();
    app_le_profile_init();
    ......
}
```

- *le_gap_init()* - Initialize GAP and set link number
- *gap_lib_init()* - Initialize gap lib
- *app_le_gap_init()* - *GAP Parameters Initialization*
- *app_le_profile_init()* - Initialize GATT Profiles

2. Start Bluetooth Technology stack in app task

```
void app_main_task(void *p_param)
{
    uint8_t event;
    os_msg_queue_create(&io_queue_handle, MAX_NUMBER_OF_IO_MESSAGE, sizeof(T_IO_MSG));
    os_msg_queue_create(&evt_queue_handle, MAX_NUMBER_OF_EVENT_MESSAGE, sizeof(uint8_t));
    gap_start_bt_stack(evt_queue_handle, io_queue_handle, MAX_NUMBER_OF_GAP_MESSAGE);
    ......
}
```

APP needs to call *gap_start_bt_stack()* to start Bluetooth Technology stack and start GAP initialization flow.

3. GAP internal initialization flow

GAP internal initialization flow is shown in Figure 2-10:

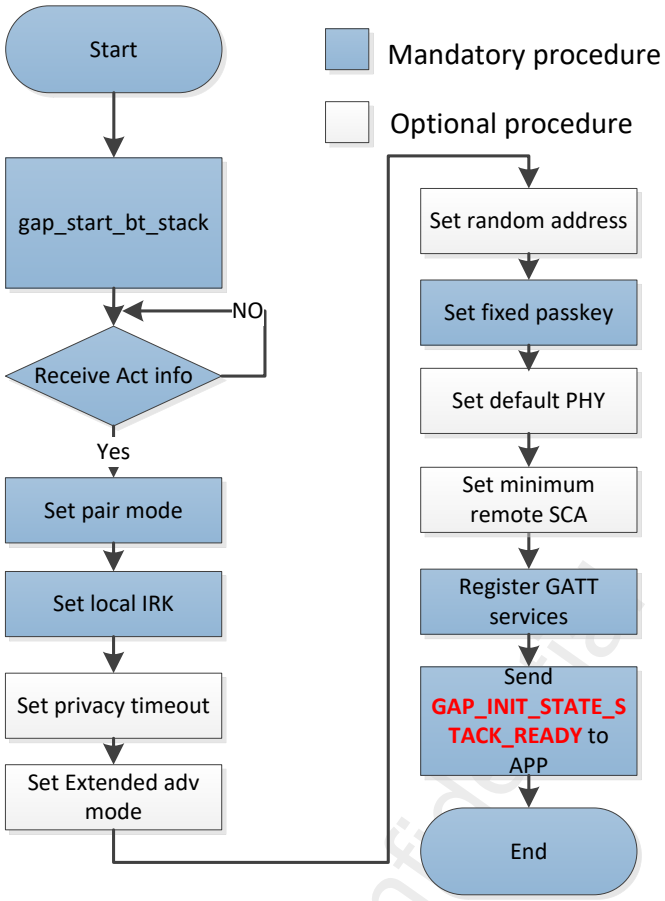

Figure 2-10 GAP Internal Initialization Flow

Flow chart Description:

1. **gap_start_bt_stack**: Make upper stack ready by sending the register request.
2. **Receive act info**: Used by upper stack to inform the stack is ready.
3. **Set pair mode**: Mandatory procedure.

   Set values of GAP_PARAM_BOND_PAIRING_MODE,

   GAP_PARAM_BOND_AUTHEN_REQUIREMENTS_FLAGS,

   GAP_PARAM_BOND_IO_CAPABILITIES and GAP_PARAM_BOND_OOB_ENABLED.
4. **Set local IRK**: Mandatory procedure.

   Set local Identity Resolving Key (IRK). If GAP_PARAM_BOND_GEN_LOCAL_IRK_AUTO is set to true, GAP will use auto-generated IRK and save the IRK to flash. Otherwise, GAP will use the value of GAP_PARAM_BOND_SET_LOCAL_IRK, whose default value is all-zero.
5. **Set privacy timeout**: Optional procedure.

   If APP calls le_privacy_set_param() to set GAP_PARAM_PRIVACY_TIMEOUT, GAP layer will set privacy timeout in initialization flow.
6. **Set extended adv mode**: Optional procedure.

   If APP calls le_set_gap_param() to set GAP_PARAM_USE_EXTENDED_ADV, GAP layer will configure

extended mode in initialization flow.

7. **Set random address**: Optional procedure.

   If APP calls le_set_gap_param() to set GAP_PARAM_RANDOM_ADDR, GAP layer will set random address in initialization flow.

8. **Set fixed passkey**: Mandatory procedure.

   Set value for GAP_PARAM_BOND_FIXED_PASSKEY and GAP_PARAM_BOND_FIXED_PASSKEY_ENABLE

9. **Set default physical**: Optional procedure.

   If APP calls le_set_gap_param() to set GAP_PARAM_DEFAULT_PHYS_PREFER, GAP_PARAM_DEFAULT_TX_PHYS_PREFER or GAP_PARAM_DEFAULT_RX_PHYS_PREFER, GAP layer will set default physical in initialization flow.

10. **Set minimum remote SCA**: Optional procedure.

    If APP calls le_set_gap_param() to set GAP_PARAM_SET_REM_MIN_SCA, GAP layer will set minimum remote Sleep Clock Accuracy (SCA) in initialization flow.

11. **Register GATT services**: Mandatory procedure.

    Register all GATT services to upper stack.

12. **Send GAP_INIT_STATE_STACK_READY to APP**: GAP initialization flow is completed.

2.3 Bluetooth LE GAP Message

2.3.1 Overview

This chapter describes the Bluetooth LE GAP Message Module. The gap message type definitions and message data structures are defined in gap_msg.h. GAP message can be divided into four types:

- *Device State Message*
- *Connection Related Message*
- *Authentication Related Message*
- *Extended Advertising State Message*

Bluetooth LE GAP Message process flow:

1. APP can call *gap_start_bt_stack()* to initialize the Bluetooth LE gap message module. The initialization codes are given below:

```
void app_main_task(void *p_param)
{
    uint8_t event;
    os_msg_queue_create(&io_queue_handle, MAX_NUMBER_OF_IO_MESSAGE, sizeof(T_IO_MSG));
```

```
    os_msg_queue_create(&evt_queue_handle, MAX_NUMBER_OF_EVENT_MESSAGE, sizeof(uint8_t));
    gap_start_bt_stack(evt_queue_handle, io_queue_handle, MAX_NUMBER_OF_GAP_MESSAGE);
    ......
}
```

2. GAP layer sends the gap message to io_queue_handle. App task receives the gap messages and call app_handle_io_msg() to handle.(event: EVENT_IO_TO_APP, type: IO_MSG_TYPE_BT_STATUS)

```
void app_main_task(void *p_param)
{
    ......
    while (true)
    {
        if (os_msg_recv(evt_queue_handle, &event, 0xFFFFFFFF) == true)
        {
            if (event == EVENT_IO_TO_APP)
            {
                T_IO_MSG io_msg;
                if (os_msg_recv(io_queue_handle, &io_msg, 0) == true)
                {
                    app_handle_io_msg(io_msg);
                }
            }
            ......
        }
    }
}
```

3. The gap messages handler function are given below:

```
void app_handle_io_msg(T_IO_MSG io_msg)
{
    uint16_t msg_type = io_msg.type;
    switch (msg_type)
    {
    case IO_MSG_TYPE_BT_STATUS:
        {
            app_handle_gap_msg(&io_msg);
        }
        break;
    default:
        break;
    }
}
void app_handle_gap_msg(T_IO_MSG *p_gap_msg)
{
    T_LE_GAP_MSG gap_msg;
```

```
    uint8_t conn_id;
    memcpy(&gap_msg, &p_gap_msg->u.param, sizeof(p_gap_msg->u.param));

    APP_PRINT_TRACE1("app_handle_gap_msg: subtype %d", p_gap_msg->subtype);
    switch (p_gap_msg->subtype)
    {
    case GAP_MSG_LE_DEV_STATE_CHANGE:
        {
            app_handle_dev_state_evt(gap_msg.msg_data.gap_dev_state_change.new_state,
                                     gap_msg.msg_data.gap_dev_state_change.cause);
        }
        break;
    ......
    }
```

2.3.2 Device State Message

2.3.2.1 GAP_MSG_LE_DEV_STATE_CHANGE

This message is used to inform GAP Device State (T_GAP_DEV_STATE). GAP device state contains five sub-states:

- **gap_init_state** : GAP Initial State
- **gap_adv_state** : GAP Advertising State
- **gap_adv_sub_state**: GAP Advertising Sub State, this state is only applied to the situation that gap_adv_state is GAP_ADV_STATE_IDLE.
- **gap_scan_state** : GAP Scan State
- **gap_conn_state** : GAP Connection State

Message data structure is T_GAP_DEV_STATE_CHANGE.

```
/** @brief  Device State.*/
typedef struct
{
    uint8_t gap_init_state: 1;  //!< @ref GAP_INIT_STATE
    uint8_t gap_adv_sub_state: 1;  //!< @ref GAP_ADV_SUB_STATE
    uint8_t gap_adv_state: 2;  //!< @ref GAP_ADV_STATE
    uint8_t gap_scan_state: 2;  //!< @ref GAP_SCAN_STATE
    uint8_t gap_conn_state: 2;  //!< @ref GAP_CONN_STATE
} T_GAP_DEV_STATE;

/** @brief  The msg_data of GAP_MSG_LE_DEV_STATE_CHANGE.*/
typedef struct
```

```
{
    T_GAP_DEV_STATE new_state;
    uint16_t cause;
} T_GAP_DEV_STATE_CHANGE;
```

The sample codes are given as below:

```
void app_handle_dev_state_evt(T_GAP_DEV_STATE new_state, uint16_t cause)
{
    APP_PRINT_INFO4("app_handle_dev_state_evt: init state  %d, adv state %d, scan state %d, cause 0x%x",
                    new_state.gap_init_state, new_state.gap_adv_state,
                    new_state.gap_scan_state, cause);
    if (gap_dev_state.gap_init_state != new_state.gap_init_state)
    {
        if (new_state.gap_init_state == GAP_INIT_STATE_STACK_READY)
        {
            APP_PRINT_INFO0("GAP stack ready");
        }
    }

    if (gap_dev_state.gap_scan_state != new_state.gap_scan_state)
    {
        if (new_state.gap_scan_state == GAP_SCAN_STATE_IDLE)
        {
            APP_PRINT_INFO0("GAP scan stop");
        }
        else if (new_state.gap_scan_state == GAP_SCAN_STATE_SCANNING)
        {
            APP_PRINT_INFO0("GAP scan start");
        }
    }
    if (gap_dev_state.gap_adv_state != new_state.gap_adv_state)
    {
        if (new_state.gap_adv_state == GAP_ADV_STATE_IDLE)
       {
            if (new_state.gap_adv_sub_state == GAP_ADV_TO_IDLE_CAUSE_CONN)
            {
                APP_PRINT_INFO0("GAP adv stoped: because connection created");
            }
            else
            {
                APP_PRINT_INFO0("GAP adv stoped");
            }
        }
        else if (new_state.gap_adv_state == GAP_ADV_STATE_ADVERTISING)
        {
```

```
                APP_PRINT_INFO0("GAP adv start");
            }
        }
        gap_dev_state = new_state;
}
```

2.3.3 Connection Related Message

2.3.3.1 GAP_MSG_LE_CONN_STATE_CHANGE

This message is used to inform link state (T_GAP_CONN_STATE).

Message data structure is T_GAP_CONN_STATE_CHANGE.

The sample codes are given as below:

```
void app_handle_conn_state_evt(uint8_t conn_id, T_GAP_CONN_STATE new_state, uint16_t disc_cause)
{
    APP_PRINT_INFO4("app_handle_conn_state_evt: conn_id %d old_state %d new_state %d, disc_cause 0x%x",
                    conn_id, gap_conn_state, new_state, disc_cause);
    switch (new_state)
    {
    case GAP_CONN_STATE_DISCONNECTED:
        {
            if ((disc_cause != (HCI_ERR | HCI_ERR_REMOTE_USER_TERMINATE))
                && (disc_cause != (HCI_ERR | HCI_ERR_LOCAL_HOST_TERMINATE)))
            {
                APP_PRINT_ERROR1("app_handle_conn_state_evt: connection lost cause 0x%x", disc_cause);
            }
            le_adv_start();
        }
        break;
    case GAP_CONN_STATE_CONNECTED:
        {
            ......
        }
        break;
    default:
        break;
    }
    gap_conn_state = new_state;
}
```

2.3.3.2 GAP_MSG_LE_CONN_PARAM_UPDATE

This message is used to inform status of connection parameter update procedure.

Update state contains three sub-states:

- *GAP_CONN_PARAM_UPDATE_STATUS_PENDING*: If local device calls le_update_conn_param() to update connection parameter, GAP Layer will send this status message when connection parameter update request succeeded and connection update complete event does not notify.
- *GAP_CONN_PARAM_UPDATE_STATUS_SUCCESS*: Update succeeded.
- *GAP_CONN_PARAM_UPDATE_STATUS_FAIL*: Update failed, parameter cause denotes the failure reason.

Message data structure is T_GAP_CONN_PARAM_UPDATE.

The sample codes are given below:

```
void app_handle_conn_param_update_evt(uint8_t conn_id, uint8_t status, uint16_t cause)
{
    switch (status)
    {
    case GAP_CONN_PARAM_UPDATE_STATUS_SUCCESS:
        ......
        break;
    case GAP_CONN_PARAM_UPDATE_STATUS_FAIL:
        ......
        break;
    case GAP_CONN_PARAM_UPDATE_STATUS_PENDING:
        ......
        break;
    }
}
```

2.3.3.3 GAP_MSG_LE_CONN_MTU_INFO

This message is used to inform that exchange MTU procedure is completed.

Message data structure is T_GAP_CONN_MTU_INFO.

The sample codes are given below:

```
void app_handle_conn_mtu_info_evt(uint8_t conn_id, uint16_t mtu_size)
{
    APP_PRINT_INFO2("app_handle_conn_mtu_info_evt: conn_id %d, mtu_size %d", conn_id, mtu_size);
}
```

2.3.4 Authentication Related Message

The relationship of pairing method and authentication message is shown in Table 2-2.

Table 2-2 Authentication Related Message

Pairing Method	Message
Just Works	GAP_MSG_LE_BOND_JUST_WORK
Numeric Comparison	GAP_MSG_LE_BOND_USER_CONFIRMATION
Passkey Entry	GAP_MSG_LE_BOND_PASSKEY_INPUT GAP_MSG_LE_BOND_PASSKEY_DISPLAY
Out Of Band (OOB)	GAP_MSG_LE_BOND_OOB_INPUT

2.3.4.1 GAP_MSG_LE_AUTHEN_STATE_CHANGE

This message indicates the new authentication state.

- *GAP_AUTHEN_STATE_STARTED* : Authentication started.
- *GAP_AUTHEN_STATE_COMPLETE*: Authentication completed, parameter cause denotes the authentication result.

Message data structure is T_GAP_AUTHEN_STATE.

The sample codes are given as below:

```
void app_handle_authen_state_evt(uint8_t conn_id, uint8_t new_state, uint16_t cause)
{
    APP_PRINT_INFO2("app_handle_authen_state_evt:conn_id %d, cause 0x%x", conn_id, cause);
    switch (new_state)
    {
    case GAP_AUTHEN_STATE_STARTED:
    {
            APP_PRINT_INFO0("app_handle_authen_state_evt: GAP_AUTHEN_STATE_STARTED");
        }
        break;
    case GAP_AUTHEN_STATE_COMPLETE:
        {
            if (cause == GAP_SUCCESS)
            {
                APP_PRINT_INFO0("app_handle_authen_state_evt: GAP_AUTHEN_STATE_COMPLETE pair
                                success");
            }
            else
            {
                APP_PRINT_INFO0("app_handle_authen_state_evt: GAP_AUTHEN_STATE_COMPLETE pair
```

```
                                        failed");
                    }
                }
                break;
        default:
                break;
        }
}
```

2.3.4.2 GAP_MSG_LE_BOND_PASSKEY_DISPLAY

This message is used to indicate that the pairing mode is Passkey Entry.

Passkey is displayed at local device, and the same key will be input at the remote device. Upon receiving the message, APP can display the passkey at its UI terminal (how to handle the passkey depends on APP). APP also needs to call le_bond_passkey_display_confirm() to confirm whether to pair with remote device.

Message data structure is T_GAP_BOND_PASSKEY_DISPLAY.

The sample codes are given below:

```
void app_handle_gap_msg(T_IO_MSG *p_gap_msg)
{
        ......
        case GAP_MSG_LE_BOND_PASSKEY_DISPLAY:
                {
                        uint32_t display_value = 0;
                        conn_id = gap_msg.msg_data.gap_bond_passkey_display.conn_id;
                        le_bond_get_display_key(conn_id, &display_value);
                        APP_PRINT_INFO1("GAP_MSG_LE_BOND_PASSKEY_DISPLAY:passkey %d", display_value);
                        le_bond_passkey_display_confirm(conn_id, GAP_CFM_CAUSE_ACCEPT);
                }
                break;
}
```

2.3.4.3 GAP_MSG_LE_BOND_PASSKEY_INPUT

This message is used to indicate that the pairing mode is Passkey Entry.

Passkey is displayed in remote device, and the same key will be inputted at the local device. Upon receiving the message, APP can call le_bond_passkey_input_confirm() to input key and confirm whether to pair with remote device.

Message data structure is T_GAP_BOND_PASSKEY_INPUT.

The sample codes are given below:

```
void app_handle_gap_msg(T_IO_MSG *p_gap_msg)
{
    ......
    case GAP_MSG_LE_BOND_PASSKEY_INPUT:
        {
            uint32_t passkey = 888888;
            conn_id = gap_msg.msg_data.gap_bond_passkey_input.conn_id;
            APP_PRINT_INFO1("GAP_MSG_LE_BOND_PASSKEY_INPUT: conn_id %d", conn_id);
            le_bond_passkey_input_confirm(conn_id, passkey, GAP_CFM_CAUSE_ACCEPT);
        }
        break;
}
```

2.3.4.4 GAP_MSG_LE_BOND_OOB_INPUT

This message is used to indicate that the pairing mode is OOB.

The local device needs to provide an OOB data which was exchanged with the remote device.

Message data structure is T_GAP_BOND_OOB_INPUT.

The sample codes are given below:

```
void app_handle_gap_msg(T_IO_MSG *p_gap_msg)
{
    ......
    case LE_GAP_MSG_TYPE_BOND_OOB_INPUT:
        {
            uint8_t oob_data[GAP_OOB_LEN] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
            conn_id = gap_msg.msg_data.gap_bond_oob_input.conn_id;
            APP_PRINT_INFO0("GAP_MSG_LE_BOND_OOB_INPUT");
            le_bond_set_param(GAP_PARAM_BOND_OOB_DATA, GAP_OOB_LEN, oob_data);
            le_bond_oob_input_confirm(conn_id, GAP_CFM_CAUSE_ACCEPT);
        }
        break;
}
```

2.3.4.5 GAP_MSG_LE_BOND_USER_CONFIRMATION

This message is used to indicate that the pairing mode is Numeric Comparison.

The keys are displayed at both local device and remote device, and user needs to check whether the keys are the same. APP needs to call le_bond_user_confirm() to confirm whether to pair with remote device.

Message data structure is T_GAP_BOND_USER_CONF.

The sample codes are given below:

```
void app_handle_gap_msg(T_IO_MSG *p_gap_msg)
{
    ......
    case GAP_MSG_LE_BOND_USER_CONFIRMATION:
        {
            uint32_t display_value = 0;
            conn_id = gap_msg.msg_data.gap_bond_user_conf.conn_id;
            le_bond_get_display_key(conn_id, &display_value);
            APP_PRINT_INFO1("GAP_MSG_LE_BOND_USER_CONFIRMATION: passkey %d",
                            display_value);
            le_bond_user_confirm(conn_id, GAP_CFM_CAUSE_ACCEPT);
        }
        break;
}
```

2.3.4.6 GAP_MSG_LE_BOND_JUST_WORK

This message is used to indicate that the pairing mode is Just Work.

APP needs to call le_bond_just_work_confirm() to confirm whether to pair with remote device.

Message data structure is T_GAP_BOND_JUST_WORK_CONF.

The sample codes are given below:

```
void app_handle_gap_msg(T_IO_MSG *p_gap_msg)
{
    ......
    case GAP_MSG_LE_BOND_JUST_WORK:
        {
            conn_id = gap_msg.msg_data.gap_bond_just_work_conf.conn_id;
            le_bond_just_work_confirm(conn_id, GAP_CFM_CAUSE_ACCEPT);
            APP_PRINT_INFO0("GAP_MSG_LE_BOND_JUST_WORK");
        }
        break;
}
```

2.3.5 Extended Advertising State Message

2.3.5.1 GAP_MSG_LE_EXT_ADV_STATE_CHANGE

This message is used to inform extended advertising state (T_GAP_EXT_ADV_STATE).

Message data structure is T_GAP_EXT_ADV_STATE_CHANGE.

The sample codes are given as below:

```
void app_handle_ext_adv_state_evt(uint8_t adv_handle, T_GAP_EXT_ADV_STATE new_state, uint16_t cause)
{
    for(int i = 0; i < APP_MAX_ADV_SET; i++)
    {
        if(ext_adv_state[i].adv_handle == adv_handle)
        {
            APP_PRINT_INFO2("app_handle_ext_adv_state_evt: adv_handle = %d oldState = %d",
                    ext_adv_state[i].adv_handle, ext_adv_state[i].ext_adv_state);
            ext_adv_state[i].ext_adv_state = new_state;
            break;
        }
    }
    APP_PRINT_INFO2("app_handle_ext_adv_state_evt: adv_handle = %d newState = %d",
                    adv_handle, new_state);

    switch (new_state)
    {
    /* device is idle */
    case EXT_ADV_STATE_IDLE:
        {
            APP_PRINT_INFO2("EXT_ADV_STATE_IDLE: adv_handle %d, cause 0x%x", adv_handle, cause);
        }
        break;

    /* device is advertising */
    case EXT_ADV_STATE_ADVERTISING:
        {
            APP_PRINT_INFO2("EXT_ADV_STATE_ADVERTISING: adv_handle %d, cause 0x%x",
adv_handle, cause);
        }
        break;

    default:
        break;
    }
}
```

2.4 Bluetooth LE GAP Callback

This section introduces the Bluetooth LE GAP Callback. This registered callback function is used by Bluetooth LE GAP Layer to send messages to APP.

Different from Bluetooth LE GAP Message, the Callback function is directly called on the GAP layer, so it is not recommended to perform any time-consuming operation in the App Callback function. Any time-consuming operation will keep any underlying process waiting and suspended, which may cause exception in some cases. If Application does need to perform a time-consuming operation immediately after receiving a message from GAP Layer, it is recommended to send this message to event queue in Application through the App Callback function before handling by Application. In such case, App Callback function will terminate after sending message to the queue, so this operation will not keep underlying process waiting.

Usage of Bluetooth LE GAP Callback in Application consists of the following steps:

1. Register the callback function:

```
void app_le_gap_init(void)
{
    ......
    le_register_app_cb(app_gap_callback);
}
```

2. Handle the GAP callback messages:

```
T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    T_LE_CB_DATA *p_data = (T_LE_CB_DATA *)p_cb_data;
    switch (cb_type)
    {
        case GAP_MSG_LE_DATA_LEN_CHANGE_INFO:
            APP_PRINT_INFO3("GAP_MSG_LE_DATA_LEN_CHANGE_INFO: conn_id %d, tx octets 0x%x,
                             max_tx_time 0x%x",
                            p_data->p_le_data_len_change_info->conn_id,
                            p_data->p_le_data_len_change_info->max_tx_octets,
                            p_data->p_le_data_len_change_info->max_tx_time);
            break;
        ......
    }
```

2.4.1 Bluetooth LE GAP Callback Message Overview

This section introduces GAP callback messages. GAP callback message type and message data are defined in gap_callback_le.h.

Most of interfaces provided by GAP Layer are asynchronous, so GAP Layer uses the callback function to send response.

For example, APP calls le_read_rssi() to read RSSI, and return value is GAP_CAUSE_SUCCESS that means sending request successfully. Then application needs to wait GAP_MSG_LE_READ_RSSI to get the result.

Detailed Bluetooth LE GAP Message information is listed as below:

1. gap_le.h Related Messages

Table 2-3 gap_le.h Related Messages

Callback type(cb_type)	Callback data(p_cb_data)	Reference API
GAP_MSG_LE_MODIFY_WHITE_LIST	T_LE_MODIFY_WHITE_LIST_RSP *p_le_modify_white_list_rsp;	le_modify_white_list
GAP_MSG_LE_SET_RAND_ADDR	T_LE_SET_RAND_ADDR_RSP *p_le_set_rand_addr_rsp;	le_set_rand_addr
GAP_MSG_LE_SET_HOST_CHANN_CLASSIF	T_LE_SET_HOST_CHANN_CLASSIF_RSP *p_le_set_host_chann_classif_rsp;	le_set_host_chann_classif
GAP_MSG_LE_VENDOR_SET_MIN_REM_SCA	T_LE_CAUSE le_cause;	le_vendor_set_rem_min_sca

2. gap_conn_le.h Related Messages

Table 2-4 gap_conn_le.h Related Messages

Callback type(cb_type)	Callback data(p_cb_data)	Reference API
GAP_MSG_LE_READ_RSSI	T_LE_READ_RSSI_RSP *p_le_read_rssi_rsp;	le_read_rssi
GAP_MSG_LE_READ_CHANN_MAP	T_LE_READ_CHANN_MAP_RSP *p_le_read_chann_map_rsp;	le_read_chann_map
GAP_MSG_LE_DISABLE_SLAVE_LATENCY	T_LE_DISABLE_SLAVE_LATENCY_RSP *p_le_disable_slave_latency_rsp;	le_disable_slave_latency
GAP_MSG_LE_SET_DATA_LEN	T_LE_SET_DATA_LEN_RSP *p_le_set_data_len_rsp;	le_set_data_len
GAP_MSG_LE_DATA_LEN_CHANGE_INFO	T_LE_DATA_LEN_CHANGE_INFO *p_le_data_len_change_info;	
GAP_MSG_LE_CONN_UPDATE_IND	T_LE_CONN_UPDATE_IND *p_le_conn_update_ind;	
GAP_MSG_LE_CREATE_CONN_IND	T_LE_CREATE_CONN_IND *p_le_create_conn_ind;	
GAP_MSG_LE_PHY_UPDATE_INFO	T_LE_PHY_UPDATE_INFO *p_le_phy_update_info;	
GAP_MSG_LE_UPDATE_PASSED_CHANN_MAP	T_LE_UPDATE_PASSED_CHANN_MAP_RSP *p_le_update_passed_chann_map_rsp;	le_update_passed_chann_map
GAP_MSG_LE_REMOTE_FEATS_INFO	T_LE_REMOTE_FEATS_INFO *p_le_remote_feats_info;	

GAP_MSG_LE_SET_CONN_TX_PWR	T_LE_CAUSE le_cause;	le_set_conn_tx_power

1) GAP_MSG_LE_DATA_LEN_CHANGE_INFO

This message notifies the Application of a change to either the maximum Payload length or the maximum transmission time of packets in either direction in Link Layer.

2) GAP_MSG_LE_CONN_UPDATE_IND

This message is only applied to central role. When the remote Bluetooth device requests connection parameter update, GAP Layer will send this message by callback function and check the return value. Thus, APP can return APP_RESULT_ACCEPT to accept the parameter or return APP_RESULT_REJECT to reject.

```
T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
{
    .....
    case GAP_MSG_LE_CONN_UPDATE_IND:
        APP_PRINT_INFO5("GAP_MSG_LE_CONN_UPDATE_IND: conn_id %d, conn_interval_max 0x%x,
                        conn_interval_min 0x%x, conn_latency 0x%x,supervision_timeout 0x%x",
                        p_data->p_le_conn_update_ind->conn_id,
                        p_data->p_le_conn_update_ind->conn_interval_max,
                        p_data->p_le_conn_update_ind->conn_interval_min,
                        p_data->p_le_conn_update_ind->conn_latency,
                        p_data->p_le_conn_update_ind->supervision_timeout);
        /* if reject the proposed connection parameter from peer device, use APP_RESULT_REJECT. */
        result = APP_RESULT_ACCEPT;
        break;
}
```

3) GAP_MSG_LE_CREATE_CONN_IND

This message is only applied to peripheral role. It is used by APP to decide whether to establish a connection. When the remote central device initiates connection, GAP Layer doesn't send this message by default and accepts this connection.

If APP wants to enable this function, GAP_PARAM_HANDLE_CREATE_CONN_IND must be set to true. The sample codes are given below:

```
void app_le_gap_init(void)
{
    .....
    uint8_t handle_conn_ind = true;
    le_set_gap_param(GAP_PARAM_HANDLE_CREATE_CONN_IND, sizeof(handle_conn_ind),
                     &handle_conn_ind);
}
T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
{
    .....
```

```
    case GAP_MSG_LE_CREATE_CONN_IND:
        /* if reject the connection from peer device, use APP_RESULT_REJECT. */
        result = APP_RESULT_ACCEPT;
        break;
}
```

4) GAP_MSG_LE_PHY_UPDATE_INFO

This message is used to indicate that the Controller has switched the transmitter PHY or receiver PHY in use.

5) GAP_MSG_LE_REMOTE_FEATS_INFO

This message is used to indicate the completion of the process that the Controller obtains the features used on the connection and the features that the remote Bluetooth device supports.

3. gap_bond_le.h Related Messages

Table 2-5 gap_bond_le.h Related Messages

Callback type(cb_type)	Callback data(p_cb_data)	Reference API
GAP_MSG_LE_BOND_MODIFY_INFO	T_LE_BOND_MODIFY_INFO *p_le_bond_modify_info;	
GAP_MSG_LE_KEYPRESS_NOTIFY	T_LE_KEYPRESS_NOTIFY_RSP *p_le_keypress_notify_rsp;	le_bond_keypress_notify
GAP_MSG_LE_KEYPRESS_NOTIFY_INFO	T_LE_KEYPRESS_NOTIFY_INFO *p_le_keypress_notify_info;	
GAP_MSG_LE_GATT_SIGNED_STATUS_INFO	T_LE_GATT_SIGNED_STATUS_INFO *p_le_gatt_signed_status_info;	

1) GAP_MSG_LE_KEYPRESS_NOTIFY_INFO

This message indicates that the SMP Layer has received the keypress notification.

2) GAP_MSG_LE_GATT_SIGNED_STATUS_INFO

This message denotes GATT signed status information.

3) GAP_MSG_LE_BOND_MODIFY_INFO

This message is used to notify app that bond information has been modified. For detailed information please refers to *LE Key Manager*.

4. gap_scan.h Related Messages

Table 2-6 gap_scan.h Related Messages

Callback type(cb_type)	Callback data(p_cb_data)	Reference API
GAP_MSG_LE_SCAN_INFO	T_LE_SCAN_INFO *p_le_scan_info;	le_scan_start
GAP_MSG_LE_DIRECT_ADV_INFO	T_LE_DIRECT_ADV_INFO *p_le_direct_adv_info;	

1) GAP_MSG_LE_SCAN_INFO

Scan state is GAP_SCAN_STATE_SCANNING. When Bluetooth Technology stack receives advertising data

or scan response data, GAP Layer will use this message to inform application.

2) GAP_MSG_LE_DIRECT_ADV_INFO

Scan state is GAP_SCAN_STATE_SCANNING and Scan filter policy is GAP_SCAN_FILTER_ANY_RPA or GAP_SCAN_FILTER_WHITE_LIST_RPA. When Bluetooth Technology stack receives directed advertising packets and the initiator's address is resolvable private address but cannot be resolved, GAP Layer will send this message to application.

5. gap_adv.h Related Messages

Table 2-7 gap_adv.h Related Messages

Callback type(cb_type)	Callback data(p_cb_data)	Reference API
GAP_MSG_LE_ADV_UPDATE_PARAM	T_LE_ADV_UPDATE_PARAM_RSP *p_le_adv_update_param_rsp;	le_adv_update_param
GAP_MSG_LE_ADV_READ_TX_POWER	T_LE_ADV_READ_TX_POWER_RSP *p_le_adv_read_tx_power_rsp;	le_adv_read_tx_power
GAP_MSG_LE_ADV_SET_TX_POWER	T_LE_CAUSE le_cause;	le_adv_set_tx_power
GAP_MSG_LE_VENDOR_ONE_SHOT_ADV	T_LE_CAUSE le_cause;	le_vendor_one_shot_adv

6. gap_dtm.h Related Messages

Table 2-8 gap_dtm.h Related Messages

Callback type(cb_type)	Callback data(p_cb_data)	Reference API
GAP_MSG_LE_DTM_RECEIVER_TEST	T_LE_CAUSE le_cause;	le_dtm_receiver_test
GAP_MSG_LE_DTM_TRANSMITTER_TEST	T_LE_CAUSE le_cause;	le_dtm_transmitter_test
GAP_MSG_LE_DTM_TEST_END	T_LE_DTM_TEST_END_RSP *p_le_dtm_test_end_rsp;	le_dtm_test_end
GAP_MSG_LE_DTM_ENHANCED_RECEIVER_TEST	T_LE_CAUSE le_cause;	le_dtm_enhanced_receiver_test
GAP_MSG_LE_DTM_ENHANCED_TRANSMITTER_TEST	T_LE_CAUSE le_cause;	le_dtm_enhanced_transmitter_test

7. GAP Lib Related Messages

GAP Lib provides extended functionalities.

1) gap_vendor.h Related Messages

Table 2-9 gap_vendor.h Related Messages

Callback type(cb_type)	Callback data(p_cb_data)	Reference API
GAP_MSG_LE_VENDOR_ADV_3_DATA_ENABLE	T_LE_CAUSE le_cause;	le_vendor_adv_3_data_enable
GAP_MSG_LE_VENDOR_ADV_3_DATA_SET	T_LE_VENDOR_ADV_3_DATA_SET_RSP *p_le_vendor_adv_3_data_set_rsp;	le_vendor_adv_3_data_set
GAP_MSG_LE_VENDOR_DROP_ACL_DATA	T_LE_CAUSE le_cause;	le_vendor_drop_acl_data
GAP_MSG_GAP_SW_RESET	T_LE_CAUSE le_cause;	gap_sw_reset_req

2) gap_ext_scan.h Related Messages

Table 2-10 gap_ext_scan.h Related Messages

Callback type(cb_type)	Callback data(p_cb_data)	Reference API
GAP_MSG_LE_EXT_ADV_REPORT_INFO	T_LE_EXT_ADV_REPORT_INFO *p_le_ext_adv_report_info;	le_ext_scan_start

GAP_MSG_LE_EXT_ADV_REPORT_INFO

Using LE Advertising Extensions, scan state is GAP_SCAN_STATE_SCANNING. When Bluetooth Technology stack receives advertising data or scan response data, GAP Layer will use this message to inform application.

3) gap_ext_adv.h Related Messages

Table 2-11 gap_ext_adv.h Related Messages

Callback type(cb_type)	Callback data(p_cb_data)	Reference API
GAP_MSG_LE_EXT_ADV_START_SETTING	T_LE_EXT_ADV_START_SETTING_RSP *p_le_ext_adv_start_setting_rsp;	le_ext_adv_start_setting
GAP_MSG_LE_EXT_ADV_REMOVE_SET	T_LE_EXT_ADV_REMOVE_SET_RSP *p_le_ext_adv_remove_set_rsp;	le_ext_adv_remove_set
GAP_MSG_LE_EXT_ADV_CLEAR_SET	T_LE_EXT_ADV_CLEAR_SET_RSP *p_le_ext_adv_clear_set_rsp;	le_ext_adv_clear_set
GAP_MSG_LE_EXT_ADV_ENABLE	T_LE_CAUSE le_cause;	le_ext_adv_enable
GAP_MSG_LE_EXT_ADV_DISABLE	T_LE_CAUSE le_cause;	le_ext_adv_disable
GAP_MSG_LE_SCAN_REQ_RECEIVED_INFO	T_LE_SCAN_REQ_RECEIVED_INFO *p_le_scan_req_received_info;	le_ext_adv_enable

GAP_MSG_LE_EXT_ADV_START_SETTING

This message is used to indicate the completion of the process that local device sets extended advertising related parameters for an advertising set.

GAP_MSG_LE_EXT_ADV_REMOVE_SET

This message is used to indicate the completion of the process that local device removes an advertising set.

GAP_MSG_LE_EXT_ADV_CLEAR_SET

This message is used to indicate the completion of the process that local device removes all existing advertising sets.

GAP_MSG_LE_EXT_ADV_ENABLE

This message is used to indicate the completion of the process that local device enables extended advertising for one or more advertising sets.

GAP_MSG_LE_EXT_ADV_DISABLE

This message is used to indicate the completion of the process that local device disables extended advertising for one or more advertising sets.

GAP_MSG_LE_SCAN_REQ_RECEIVED_INFO

Extended advertising state is EXT_ADV_STATE_ADVERTISING, and scan request notifications is enabled by calling le_ext_adv_set_adv_param(). When Bluetooth Technology stack receives scan request, GAP Layer will use this message to inform application.

2.5 Bluetooth LE GAP Use Case

This chapter is used to show how to use LE GAP interfaces. This document is to give some of typical use cases.

2.5.1 Airplane Mode Setting

The sample code is located in **BLE scatternet project**.

1. **Register GAP Common Callback**

   APP needs to call gap_register_app_cb() to register callback function.

```
void app_le_gap_init(void)
{
    ......
    gap_register_app_cb(app_gap_common_callback);
}
```

2. **GAP Common Callback Handler**

```
void app_gap_common_callback(uint8_t cb_type, void *p_cb_data)
{
    T_GAP_CB_DATA cb_data;
    memcpy(&cb_data, p_cb_data, sizeof(T_GAP_CB_DATA));
    APP_PRINT_INFO1("app_gap_common_callback: cb_type = %d", cb_type);
```

```
    switch (cb_type)
    {
    case GAP_MSG_WRITE_AIRPLAN_MODE:
        APP_PRINT_INFO1("GAP_MSG_WRITE_AIRPLAN_MODE: cause 0x%x",
                        cb_data.p_gap_write_airplan_mode_rsp->cause);
        break;
    case GAP_MSG_READ_AIRPLAN_MODE:
        APP_PRINT_INFO2("GAP_MSG_READ_AIRPLAN_MODE: cause 0x%x, mode %d",
                        cb_data.p_gap_read_airplan_mode_rsp->cause,
                        cb_data.p_gap_read_airplan_mode_rsp->mode);
        break;
    default:
        break;
    }
    return;
}
```

This callback function is used to handle message GAP_MSG_WRITE_AIRPLAN_MODE and GAP_MSG_READ_AIRPLAN_MODE.

3. Related APIs

gap_write_airplan_mode() function is used to write airplane mode.

gap_read_airplan_mode() function is used to read airplane mode.

```
static T_USER_CMD_PARSE_RESULT cmd_wairplane(T_USER_CMD_PARSED_VALUE *p_parse_value)
{
    T_GAP_CAUSE cause;
    uint8_t mode = p_parse_value->dw_param[0];
    cause = gap_write_airplan_mode(mode);
    return (T_USER_CMD_PARSE_RESULT)cause;
}
static T_USER_CMD_PARSE_RESULT cmd_rairplane(T_USER_CMD_PARSED_VALUE *p_parse_value)
{
    T_GAP_CAUSE cause;
    cause = gap_read_airplan_mode();
    return (T_USER_CMD_PARSE_RESULT)cause;
}
```

2.5.2 Local IRK Setting

Identity Resolving Key (IRK) is a 128-bit key used to generate and resolve random addresses. Default local IRK is all-zero.

If a device supports the generation of resolvable private addresses and generates a resolvable private address for its local address, it shall send Identity Information with SMP, including a valid IRK. If a device does not generate

a resolvable private address for its own address and the Host sends Identity Information with SMP, the Host shall send an all-zero IRK. Under such circumstance, local device does not use resolvable private address and APP needn't set local IRK.

GAP layer provides two methods to set Local IRK.

- **Auto generated local IRK**

  If APP wants to enable this function, needs to set GAP_PARAM_BOND_GEN_LOCAL_IRK_AUTO to true. When the GAP layer starts up, GAP layer will call flash_load_local_irk() to load local IRK. If GAP layer loads local IRK failed, GAP layer will auto generate local IRK and call flash_save_local_irk() to save local IRK. So when this function is enabled, APP can't use flash_load_local_irk() and flash_save_local_irk().

```
void app_le_gap_init(void)
{
    ......
    uint8_t irk_auto = true;
    le_bond_set_param(GAP_PARAM_BOND_GEN_LOCAL_IRK_AUTO, sizeof(uint8_t), &irk_auto);
}
```

- **APP generates local IRK**

  GAP layer uses this method by default and default value is all-zero. APP can call le_bond_set_param() to set GAP_PARAM_BOND_SET_LOCAL_IRK to change local IRK.

```
void app_le_gap_init(void)
{
    T_LOCAL_IRK le_local_irk = {0, 1, 0, 5, 0, 0, 7, 0, 0, 0, 0, 0, 0, 0, 0, 9};
    le_bond_set_param(GAP_PARAM_BOND_SET_LOCAL_IRK, GAP_KEY_LEN, le_local_irk.local_irk);
}
```

  This parameter is only valid in the GAP layer startup process. So APP needs to set GAP_PARAM_BOND_SET_LOCAL_IRK before calling gap_start_bt_stack().

  APP can generate local IRK and call flash_save_local_irk() to save IRK. Then APP can use flash_load_local_irk() to load local IRK and set GAP_PARAM_BOND_SET_LOCAL_IRK to change local IRK.

2.5.3 GAP Service Characteristic Writeable

The sample code is located in **BLE scatternet project**.

Device name characteristic and device appearance characteristic of GAP service have an optional writable property. The writable property is closed by default. APP can call gaps_set_parameter() to set GAPS_PARAM_APPEARANCE_PROPERTY and GAPS_PARAM_DEVICE_NAME_PROPERTY to configure writeable property.

1. Writeable Property Configuration

```
void app_le_gap_init(void)
{
    uint8_t appearance_prop = GAPS_PROPERTY_WRITE_ENABLE;
    uint8_t device_name_prop = GAPS_PROPERTY_WRITE_ENABLE;
    T_LOCAL_APPEARANCE appearance_local;
    T_LOCAL_NAME local_device_name;
    if (flash_load_local_appearance(&appearance_local) == 0)
    {
        gaps_set_parameter(GAPS_PARAM_APPEARANCE, sizeof(uint16_t),
                           &appearance_local.local_appearance);
    }
    if (flash_load_local_name(&local_device_name) == 0)
    {
        gaps_set_parameter(GAPS_PARAM_DEVICE_NAME, GAP_DEVICE_NAME_LEN,
                           local_device_name.local_name);
    }
    gaps_set_parameter(GAPS_PARAM_APPEARANCE_PROPERTY, sizeof(appearance_prop),
                       &appearance_prop);
    gaps_set_parameter(GAPS_PARAM_DEVICE_NAME_PROPERTY, sizeof(device_name_prop),
                       &device_name_prop);
    gatt_register_callback(gap_service_callback);
}
```

2. GAP Service Callback Handler

APP needs to invoke gatt_register_callback() to register callback function. This callback function is used to handle gap service messages.

```
T_APP_RESULT gap_service_callback(T_SERVER_ID service_id, void *p_para)
{
    T_APP_RESULT  result = APP_RESULT_SUCCESS;
    T_GAPS_CALLBACK_DATA *p_gap_data = (T_GAPS_CALLBACK_DATA *)p_para;
    APP_PRINT_INFO2("gap_service_callback conn_id = %d msg_type = %d\n", p_gap_data->conn_id,
                    p_gap_data->msg_type);
    if (p_gap_data->msg_type == SERVICE_CALLBACK_TYPE_WRITE_CHAR_VALUE)
    {
        switch (p_gap_data->msg_data.opcode)
        {
            case GAPS_WRITE_DEVICE_NAME:
                {
                    T_LOCAL_NAME device_name;
                    memcpy(device_name.local_name, p_gap_data->msg_data.p_value,
                           p_gap_data->msg_data.len);
                    device_name.local_name[p_gap_data->msg_data.len] = 0;
                    flash_save_local_name(&device_name);
```

```
                }
                break;
            case GAPS_WRITE_APPEARANCE:
                {
                    uint16_t appearance_val;
                    T_LOCAL_APPEARANCE appearance;
                    LE_ARRAY_TO_UINT16(appearance_val, p_gap_data->msg_data.p_value);
                    appearance.local_appearance = appearance_val;
                    flash_save_local_appearance(&appearance);
                }
                break;
        default:
            break;
        }
    }
    return result;
}
```

APP needs to save device name and device appearance to Flash. Please refer to chapter *Local Stack Information Storage*.

2.5.4 Local Static Random Address

The sample code is located in **BLE scatternet project**.

Local address type that is used in advertising, scanning and connection is Public Address by default, and local address type could be configured as Static Random Address.

1. Generation, storage and configuration of Random Address

APP may call le_gen_rand_addr() to generate random address for the first time, and save generated random address to Flash. If random address has been saved in Flash, APP gets random address by loading from storage. Then APP calls le_set_gap_param() with GAP_PARAM_RANDOM_ADDR to set random address. It is valid only when APP called le_set_gap_param() with GAP_PARAM_RANDOM_ADDR before stack ready, and le_gen_rand_addr() should be called after stack ready.

2. Set Identity Address

Stack uses public address as Identity Address by default. APP needs to call le_cfg_local_identity_address() to modify Identity Address to static random address. If configuration of Identity Address is incorrect, reconnection could not be implemented after pairing.

3. Set local address type

Peripheral role or broadcaster role call le_adv_set_param() to configure local address type to use Static Random Address. Central role or observer role call le_scan_set_param() to configure local address type to use Static

Random Address. Sample codes are listed as below:

```
void app_le_gap_init(void)
{
    ……
    T_APP_STATIC_RANDOM_ADDR random_addr;
    bool gen_addr = true;
    uint8_t local_bd_type = GAP_LOCAL_ADDR_LE_RANDOM;
    if (app_load_static_random_address(&random_addr) == 0)
    {
        if (random_addr.is_exist == true)
        {
            gen_addr = false;
        }
    }
    if (gen_addr)
    {
        if (le_gen_rand_addr(GAP_RAND_ADDR_STATIC, random_addr.bd_addr) == GAP_CAUSE_SUCCESS)
        {
            random_addr.is_exist = true;
            app_save_static_random_address(&random_addr);
        }
    }
    le_cfg_local_identity_address(random_addr.bd_addr, GAP_IDENT_ADDR_RAND);
    le_set_gap_param(GAP_PARAM_RANDOM_ADDR, 6, random_addr.bd_addr);
    //only for peripheral,broadcaster
    le_adv_set_param(GAP_PARAM_ADV_LOCAL_ADDR_TYPE, sizeof(local_bd_type), &local_bd_type);
    //only for central,observer
    le_scan_set_param(GAP_PARAM_SCAN_LOCAL_ADDR_TYPE, sizeof(local_bd_type), &local_bd_type);
    ……
}
```

Central calls le_connect() function to configure local address type to use Static Random Address. Sample codes are listed as below:

```
static T_USER_CMD_PARSE_RESULT cmd_condev(T_USER_CMD_PARSED_VALUE *p_parse_value)
{
    ……
#if F_BT_LE_USE_STATIC_RANDOM_ADDR
    T_GAP_LOCAL_ADDR_TYPE local_addr_type = GAP_LOCAL_ADDR_LE_RANDOM;
#else
    T_GAP_LOCAL_ADDR_TYPE local_addr_type = GAP_LOCAL_ADDR_LE_PUBLIC;
#endif
    ……
    cause = le_connect(GAP_PHYS_CONN_INIT_1M_BIT,
                       dev_list[dev_idx].bd_addr,
```

```
                                (T_GAP_REMOTE_ADDR_TYPE)dev_list[dev_idx].bd_type,
                                local_addr_type,
                                1000);
    ......
}
```

2.5.5 Physical (PHY) Setting

The sample code is located in **BLE scatternet project**.

LE mandatory symbol rate is 1 mega symbol per second (Msym/s), where 1 symbol represents 1 bit therefore supporting a bit rate of 1 megabit per second (Mb/s), which is referred to as the **LE 1M PHY**. The 1 Msym/s symbol rate may optionally support error correction coding, which is referred to as the **LE Coded PHY**. This may use either of two coding schemes: S=2, where 2 symbols represent 1 bit therefore supporting a bit rate of 500 kb/s, and S=8, where 8 symbols represent 1 bit therefore supporting a bit rate of 125 kb/s. An optional symbol rate of 2 Msym/s may be supported, with a bit rate of 2 Mb/s, which is referred to as the **LE 2M PHY**. The 2 Msym/s symbol rate supports uncoded data only. LE 1M PHY and LE 2M PHY are collectively referred to as the LE Uncoded PHYs[1].

1. **Set Default PHY**

   APP can specify its preferred values for the transmitter PHY and receiver PHY to be used for all subsequent connections over the LE transport.

```
void app_le_gap_init(void)
{
    uint8_t  phys_prefer = GAP_PHYS_PREFER_ALL;
    uint8_t  tx_phys_prefer = GAP_PHYS_PREFER_1M_BIT | GAP_PHYS_PREFER_2M_BIT |
                              GAP_PHYS_PREFER_CODED_BIT;
    uint8_t  rx_phys_prefer = GAP_PHYS_PREFER_1M_BIT | GAP_PHYS_PREFER_2M_BIT |
                              GAP_PHYS_PREFER_CODED_BIT;
    le_set_gap_param(GAP_PARAM_DEFAULT_PHYS_PREFER, sizeof(phys_prefer), &phys_prefer);
    le_set_gap_param(GAP_PARAM_DEFAULT_TX_PHYS_PREFER, sizeof(tx_phys_prefer), &tx_phys_prefer);
    le_set_gap_param(GAP_PARAM_DEFAULT_RX_PHYS_PREFER, sizeof(rx_phys_prefer), &rx_phys_prefer);
}
```

2. **Read connection PHY type**

   After establishing connection successfully, APP can call le_get_conn_param() to get TX PHY and RX PHY type.

```
void app_handle_conn_state_evt(uint8_t conn_id, T_GAP_CONN_STATE new_state, uint16_t disc_cause)
{
    ......
    switch (new_state)
    {
```

```
        case GAP_CONN_STATE_CONNECTED:
            {
                ......
                data_uart_print("Connected success conn_id %d\r\n", conn_id);
#if F_BT_LE_5_0_SET_PHY_SUPPORT
                uint8_t tx_phy;
                uint8_t rx_phy;
                le_get_conn_param(GAP_PARAM_CONN_RX_PHY_TYPE, &rx_phy, conn_id);
                le_get_conn_param(GAP_PARAM_CONN_TX_PHY_TYPE, &tx_phy, conn_id);
                APP_PRINT_INFO2("GAP_CONN_STATE_CONNECTED: tx_phy %d, rx_phy %d", tx_phy,
                                rx_phy);
#endif
            }
            break;
}
```

3. Remote Features Info Check

After establishing connection successfully, Bluetooth Technology stack will read remote features. GAP Layer will send GAP_MSG_LE_REMOTE_FEATS_INFO to inform remote features. APP can check whether remote device supports LE 2M PHY or LE Coded PHY.

```
T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    T_LE_CB_DATA *p_data = (T_LE_CB_DATA *)p_cb_data;
    switch (cb_type)
    {
#if F_BT_LE_5_0_SET_PHY_SUPPORT
    case GAP_MSG_LE_REMOTE_FEATS_INFO:
    {
        uint8_t    remote_feats[8];
        APP_PRINT_INFO3("GAP_MSG_LE_REMOTE_FEATS_INFO: conn id %d, cause 0x%x,
                        remote_feats %b",
                        p_data->p_le_remote_feats_info->conn_id,
                        p_data->p_le_remote_feats_info->cause,
                        TRACE_BINARY(8, p_data->p_le_remote_feats_info->remote_feats));
        if (p_data->p_le_remote_feats_info->cause == GAP_SUCCESS)
        {
            memcpy(remote_feats, p_data->p_le_remote_feats_info->remote_feats, 8);
            if (remote_feats[LE_SUPPORT_FEATURES_MASK_ARRAY_INDEX1] &
                LE_SUPPORT_FEATURES_LE_2M_MASK_BIT)
            {
                APP_PRINT_INFO0("GAP_MSG_LE_REMOTE_FEATS_INFO: support 2M");
            }
            if (remote_feats[LE_SUPPORT_FEATURES_MASK_ARRAY_INDEX1] &
```

```
                                LE_SUPPORT_FEATURES_LE_CODED_PHY_MASK_BIT)
                {
                    APP_PRINT_INFO0("GAP_MSG_LE_REMOTE_FEATS_INFO: support CODED");
                }
            }
        }
        break;
#endif
        }
}
```

4. Set PHY

le_set_phy() is used to set the PHY preferences for the connection identified by conn_id. The Controller might not be able to make the change (e.g. because the peer does not support the requested PHY) or may decide that the current PHY is preferable.

```
static T_USER_CMD_PARSE_RESULT cmd_setphy(T_USER_CMD_PARSED_VALUE *p_parse_value)
{
    uint8_t conn_id = p_parse_value->dw_param[0];
    uint8_t all_phys;
    uint8_t tx_phys;
    uint8_t rx_phys;
    T_GAP_PHYS_OPTIONS phy_options = GAP_PHYS_OPTIONS_CODED_PREFER_S8;
    T_GAP_CAUSE cause;
    if (p_parse_value->dw_param[1] == 0)
    {
        all_phys = GAP_PHYS_PREFER_ALL;
        tx_phys = GAP_PHYS_PREFER_1M_BIT;
        rx_phys = GAP_PHYS_PREFER_1M_BIT;
    }
    else if (p_parse_value->dw_param[1] == 1)
    {
        all_phys = GAP_PHYS_PREFER_ALL;
        tx_phys = GAP_PHYS_PREFER_2M_BIT;
        rx_phys = GAP_PHYS_PREFER_2M_BIT;
    }
    ......
    cause = le_set_phy(conn_id, all_phys, tx_phys, rx_phys, phy_options);
    return (T_USER_CMD_PARSE_RESULT)cause;
}
```

5. PHY Update

GAP_MSG_LE_PHY_UPDATE_INFO is used to inform result of updating transmitter PHY or receiver PHY used by the Controller.

```
T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
```

```
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    T_LE_CB_DATA *p_data = (T_LE_CB_DATA *)p_cb_data;
    switch (cb_type)
    {
#if F_BT_LE_5_0_SET_PHY_SUPPORT
        case GAP_MSG_LE_PHY_UPDATE_INFO:
            APP_PRINT_INFO4("GAP_MSG_LE_PHY_UPDATE_INFO:conn_id %d, cause 0x%x, rx_phy %d,
                            tx_phy %d",
                            p_data->p_le_phy_update_info->conn_id,
                            p_data->p_le_phy_update_info->cause,
                            p_data->p_le_phy_update_info->rx_phy,
                            p_data->p_le_phy_update_info->tx_phy);
            break;
#endif
    }
}
```

2.5.6 Bluetooth Technology Stack Features Setting

Configuration of Bluetooth Technology stack related features are divided into configuration of LE Link number and parameter configuration with API.

The sample code is located in **BLE scatternet project**.

2.5.6.1 Configuration of LE Link number

Due to support of multilink and requirements of APP, user needs to configure LE Link number. This section takes **BLE scatternet project** as an example to introduce configuration of LE Link number.

Based on the assumption that **BLE scatternet project** supports one link as master role and one link as slave role at the same time, the configuration is as follows.

1. Configure LE Link number in Config file

User needs to configure LE Link number in Config file with MPTool, and configurable parameters are as follows:

- Maximum LE Link Number: LE Link number, and maximum value is 4
- Master Link Number: link number as master role, and maximum value is 4
- Slave Link Number: link number as slave role, and maximum value is 3

BLE scatternet project supports one link as master role and one link as slave role at the same time. Therefore, Maximum LE Link Number shall be no less than 2, Master Link Number shall be no less than 1, and Slave Link Number shall be no less than 1. The configuration is shown in Figure 2-11.

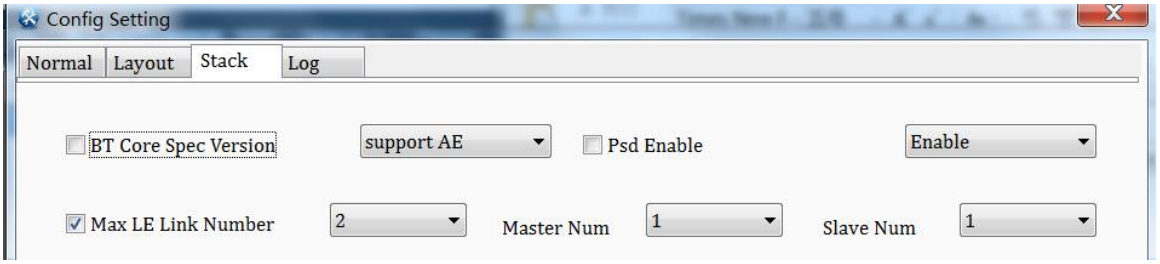

Figure 2-11 LE Link number in Config file

2. Configure LE maximum bonded device number in APP

Due to support of multilink, device needs to save bonding informations of multiple peer devices. **BLE scatternet project** configures LE maximum bonded device number as 2, and more information about configuration please refer to *Parameter Configuration with API*.

3. Initialize LE Link number in APP

As described in *GAP Startup Flow*, APP calls *le_gap_init()* to initialize GAP and set link number. Link number configured by *le_gap_init()* shall be less than or equal to Maximum LE Link Number configured by Config file. If link number configured by *le_gap_init()* is greater than Maximum LE Link Number configured by Config file, initialization of link number will be failed and return value of *le_gap_init()* is false.

In **BLE scatternet project**, link number is configured by macro definition APP_MAX_LINKS.

```
int main(void)
{
    ......
    le_gap_init(APP_MAX_LINKS);
    ......
}
```

Maximum LE Link Number in Config file is configured as 2, so APP_MAX_LINKS is configured as 2 in header file app_flags.h.

```
/** @brief    Config APP LE link number */
#define APP_MAX_LINKS    2
```

2.5.6.2 Parameter Configuration with API

APP can use APIs which are defined in gap_config.h to configure Bluetooth Technology stack related features, such as LE maximum bonded device number, maximum CCCD number. APP shall follow the configuration method to use APIs.

1. Add macro definition in otp_config.h

```
/*============================================================================*
*                          upperstack configuration
*============================================================================*/
#define BT_STACK_CONFIG_ENABLE
```

```
#ifdef BT_STACK_CONFIG_ENABLE
void bt_stack_config_init(void);
#endif
```

2. Configure Bluetooth Technology Stack features in main.c

Default value of LE maximum bonded device number is 1. Due to support of multilink, APP uses gap_config_max_le_paired_device() to configure LE maximum bonded device number.

```
……
#include <gap_config.h>
#include <otp_config.h>
……

#ifdef BT_STACK_CONFIG_ENABLE
#include "app_section.h"

APP_FLASH_TEXT_SECTION void bt_stack_config_init(void)
{
    gap_config_max_le_paired_device(APP_MAX_LINKS);
}
#endif
```

2.5.7 Request Pairing

iOS system doesn't supply interface to initiate security procedure. If the slave device wants to pair with iOS device, the slave device need request pairing.

There are two ways to initiate security procedure, and one way that local device with GATT Server could request iOS device to initiate security procedure.

2.5.7.1 Configure GAP_PARAM_BOND_SEC_REQ_ENABLE

Parameter GAP_PARAM_BOND_SEC_REQ_ENABLE determines whether to initiate security procedure when connection is established. If the parameter is configured as true, the GAP layer will automatically initiate security procedure when connection is established.

```
void app_le_gap_init(void)
{
    ......
    uint8_t    auth_sec_req_enable = true;
    uint16_t auth_sec_req_flags = GAP_AUTHEN_BIT_BONDING_FLAG;
    le_bond_set_param(GAP_PARAM_BOND_SEC_REQ_ENABLE, sizeof(auth_sec_req_enable),
&auth_sec_req_enable);
```

```
    le_bond_set_param(GAP_PARAM_BOND_SEC_REQ_REQUIREMENT, sizeof(auth_sec_req_flags),
                      &auth_sec_req_flags);
    ......
}
```

2.5.7.2 Call function le_bond_pair

Application can call le_bond_pair() to initiate security procedure. The function can only be called when LE link state is GAP_CONN_STATE_CONNECTED.

```
void app_handle_conn_state_evt(uint8_t conn_id, T_GAP_CONN_STATE new_state, uint16_t disc_cause)
{
    ......
    switch (new_state)
    {
    ......
    case GAP_CONN_STATE_CONNECTED:
        {
            le_bond_pair(conn_id);
        }
        break;
    default:
        break;
    }
}
```

2.5.7.3 Security Requirement of Service

The GATT Profile procedures are used to access information that may require the client to be authenticated and have an encrypted connection before a characteristic can be read or written.

If such a request is issued when the physical link is unauthenticated or unencrypted, the server shall send an Error Response. The client wanting to read or write this characteristic can then request that the physical link be authenticated using the GAP authentication procedure, and once this has been completed, send the request again.

The sample is shown in Figure 2-12. When iOS system receives the unauthenticated or unencrypted error reponse, iOS system will initiate security procedure.

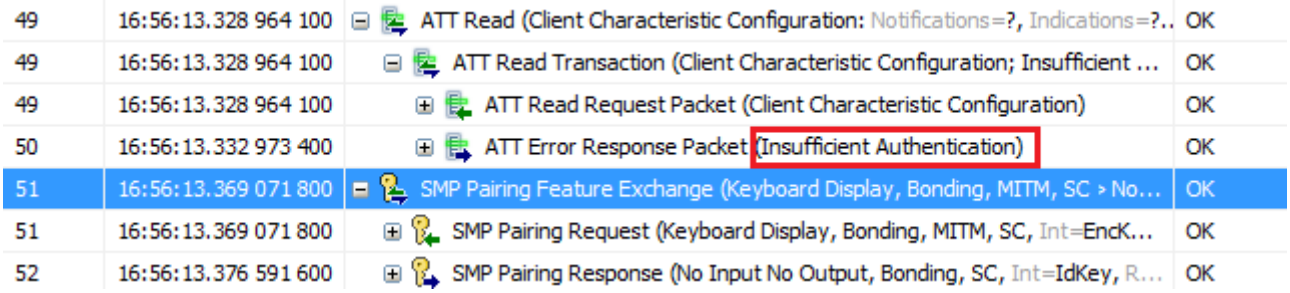

Figure 2-12 ATT Insufficient Authentication

Attribute element is elementary unit of service. The structure of attribute element is defined in gatt.h.

```
typedef struct
{
    uint16_t    flags;              /**< Attribute flags @ref GATT_ATTRIBUTE_FLAG */
    uint8_t     type_value[2 + 14]; /**< 16 bit UUID + included value or 128 bit UUID */
    uint16_t    value_len;          /**< Length of value */
    void         *p_value_context;  /**< Pointer to value if @ref ATTRIB_FLAG_VALUE_INCL
    and @ref ATTRIB_FLAG_VALUE_APPL not set */
    uint32_t    permissions;        /**< Attribute permission @ref GATT_ATTRIBUTE_PERMISSIONS */
} T_ATTRIB_APPL;
```

The parameter permissions is used to define permission of the attribute. More information please refers to *Attribute Element.*

The sample code about authentication requirement of characteristic is as follows.

This characteristic requires an authenticated link before this characteristic can be read and write.

```
    /* client characteristic configuration .. 5*/
    {
        (ATTRIB_FLAG_VALUE_INCL | ATTRIB_FLAG_CCCD_APPL),                    /* wFlags */
        {                                                   /* bTypeValue */
            LO_WORD(GATT_UUID_CHAR_CLIENT_CONFIG),
            HI_WORD(GATT_UUID_CHAR_CLIENT_CONFIG),
            /* NOTE: this value has an instantiation for each client, a write to */
            /* this attribute does not modify this default value:                */
            LO_WORD(GATT_CLIENT_CHAR_CONFIG_DEFAULT), /* client char. config. bit field */
            HI_WORD(GATT_CLIENT_CHAR_CONFIG_DEFAULT)
        },
        2,                                                  /* bValueLen */
        NULL,
        (GATT_PERM_READ_AUTHEN_REQ | GATT_PERM_WRITE_AUTHEN_REQ)        /*
wPermissions */
    },
```

The sample code is located in **hids_kb.c**.

2.6 GAP Information Storage

The constants and functions prototype are defined in gap_storage_le.h.

Local stack and bond information are saved in FTL. More information on FTL can be found in *FTL Introduction*.

2.6.1 FTL Introduction

Bluetooth Technology stack and user application use FTL as abstraction layer to save/load data in flash.

2.6.1.1 FTL Layout

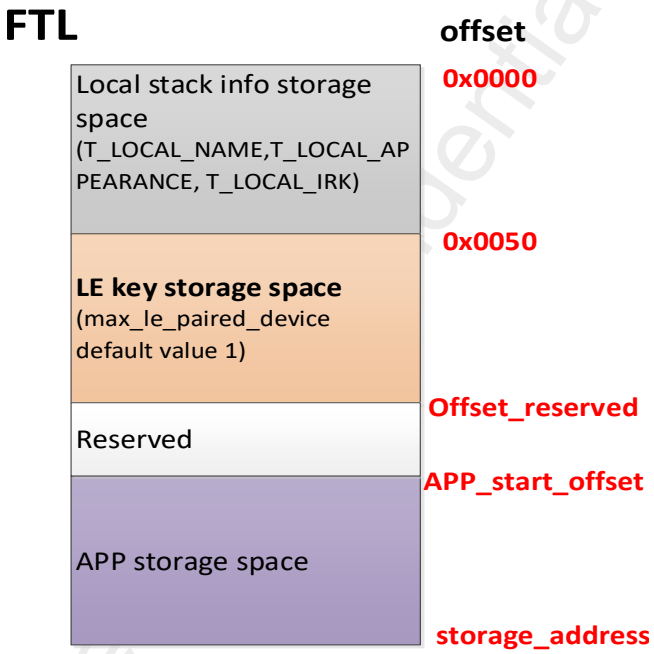

Figure 2-13 FTL Layout

FTL can be divided into four regions:

1. Local stack information storage space
   1) range: 0x0000 - 0x004F
   2) This region is used to store local stack information including device name, device appearance and local IRK. For more information please refers to *Local Stack Information Storage*.
2. LE key storage space
   1) range: 0x0050 - (Offset_reserved – 1)
   2) This region is used to store LE key information. For more information please refers to *Bond Information Storage*.
   3) Offset_reserved is a variable value.

Offset_reserved = 0x0064 + max_le_paired_device*device_block_size

4) max_le_paired_device: This parameter represents the the maximum number of storage devices that can be configured by gap_config_max_le_paired_device() and its default value is 1. More information please refers to *Parameter Configuration with API*.

3. Reserved space
   1) range: Offset_reserved - (APP_start_offset – 1)
   2) This region is reserved space, because variable max_le_paired_device are configurable. LE key storage space can use this region to store key information.
4. APP storage space
   1) range: APP_start_offset – storage_address, more information refers to memory related document [错误!未找到引用源。].
   2) APP can use this region to store information.
   3) APP can use ftl_save() and ftl_load() function to access this storage space.

2.6.1.2 FTL APIs

ftl_save() function is used by APP to save data to FTL.

ftl_load() function is used by APP to load data from FTL.

```
#define FTL_APP_START_ADDR    (3 * 1024)
static inline uint32_t ftl_save(void *pdata, uint16_t offset, uint16_t size)
{
    return ftl_save_to_storage(pdata, offset + FTL_APP_START_ADDR, size);
}
static inline uint32_t ftl_load(void *pdata, uint16_t offset, uint16_t size)
{
    return ftl_load_from_storage(pdata, offset + FTL_APP_START_ADDR, size);
}
```

ftl_save() writes to the FTL starting at FTL_APP_START_ADDR. ftl_save() reads the FTL starting at FTL_APP_START_ADDR.

Offset can be used starting at 0 when APP uses ftl_save() and ftl_load(). The sample codes are given below:

```
/* Define start offset of the flash to save static random address. */
#define APP_STATIC_RANDOM_ADDR_OFFSET 0

uint32_t app_save_static_random_address(T_APP_STATIC_RANDOM_ADDR *p_addr)
{
    APP_PRINT_INFO0("app_save_static_random_address");
    return ftl_save(p_addr, APP_STATIC_RANDOM_ADDR_OFFSET,
                    sizeof(T_APP_STATIC_RANDOM_ADDR));
}
```

```
uint32_t app_load_static_random_address(T_APP_STATIC_RANDOM_ADDR *p_addr)
{
    uint32_t result;
    result = ftl_load(p_addr, APP_STATIC_RANDOM_ADDR_OFFSET,
                      sizeof(T_APP_STATIC_RANDOM_ADDR));
    APP_PRINT_INFO1("app_load_static_random_address: result 0x%x", result);
    if (result)
    {
        memset(p_addr, 0, sizeof(T_APP_STATIC_RANDOM_ADDR));
    }
    return result;
}
```

2.6.2 Local Stack Information Storage

2.6.2.1 Device Name Storage

Currently, the maximum length of device name character string which GAP layer suppotrs is 40 bytes (including end mark).

flash_save_local_name() function is used to save the local name to FTL.

flash_load_local_name() function is used to load the local name from FTL.

If device name characteristic of GAP service is writeable, application can use this function to save the device name. The sample codes are given in *GAP Service Characteristic Writeable*.

2.6.2.2 Device Appearance Storage

Device Appearance is used to describe the type of a device, such as keyboard, mouse, thermometer, blood pressure meter etc.

flash_save_local_appearance() function is used to save the appearance to FTL.

flash_load_local_appearance() function is used to load the appearance from FTL.

If device appearance characteristic of GAP service is writeable, application can use this function to save the device appearance. The sample codes are given in *GAP Service Characteristic Writeable*.

2.6.2.3 Local IRK Storage

IRK is a 128-bit key used to generate and resolve random addresses.

flash_save_local_irk() function is used to save the local IRK to FTL.

flash_load_local_irk() function is used to load the local IRK from FTL. The sample codes are given in Local IRK Setting.

2.6.3 Bond Information Storage

2.6.3.1 Bonded Device Priority Manager

GAP layer implements bonded device priority management mechanism. Priority control block will be saved to FTL. LE device has storage space and priority control block.

Key priority control block contains two parts:

- **bond_num**: Saved bonded devices number
- **bond_idx** array: Saved bonded devices index array. GAP layer can use bonded device index to search for the start offset in FTL.

Priority manager consists of operations listed below:

1. **Add a bond device**

   GAP LE API: Not provided, for internal use.

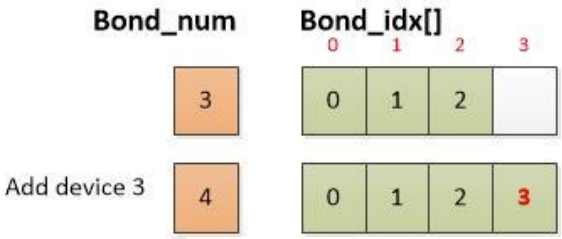

Figure 2-14 Add A Bond Device

2. **Remove a bond device**

   GAP LE API: le_bond_delete_by_idx() or le_bond_delete_by_bd()

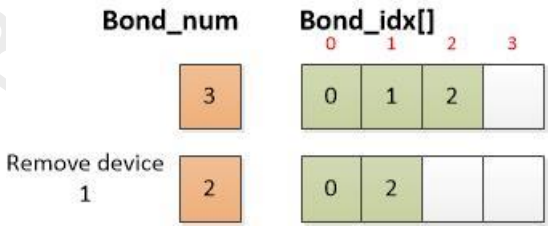

Figure 2-15 Remove A Bond Device

3. **Clear all bond devices**

   GAP LE API: le_bond_clear_all_keys()

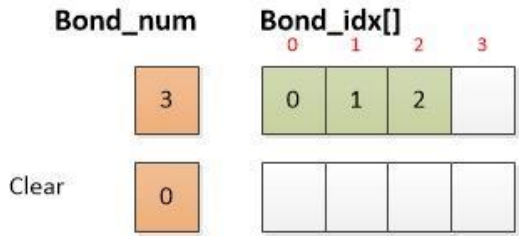

Figure 2-16 Clear All Bond Devices

4. Set a bond device high priority

GAP LE API: le_set_high_priority_bond()

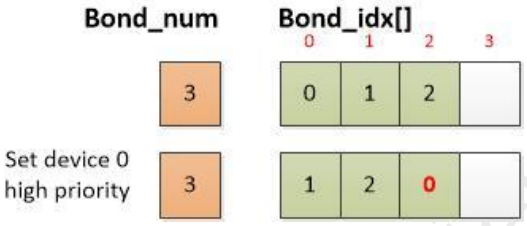

Figure 2-17 Set A Bond Device High Priority

5. Get high priority device

The highest priority device is bond_idx[bond_num - 1].

GAP LE API: le_get_high_priority_bond()

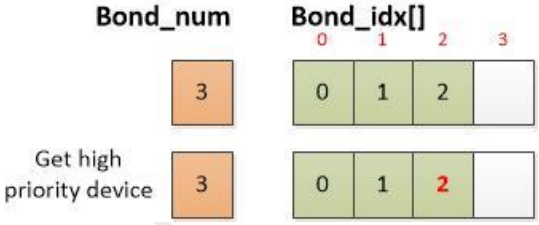

Figure 2-18 Get High Priority Device

6. Get low priority device

The lowest priority device is bond_idx[0].

GAP LE API: le_get_low_priority_bond()

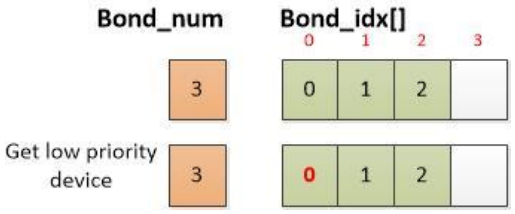

Figure 2-19 Get Low Priority Device

A priority manager example is shown in Figure 2-20 :

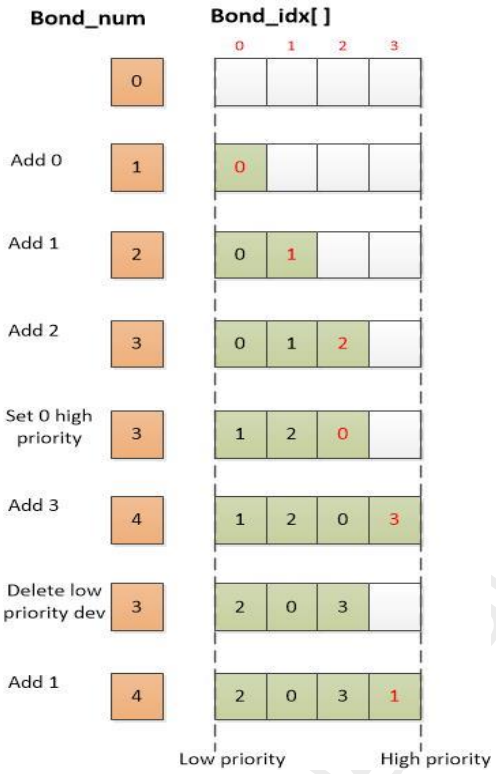

Figure 2-20 Priority Manager Example

2.6.3.2 Bluetooth LE Key Storage

Bluetooth LE Key information is stored in LE key storage space.

LE FTL layout is shown in Figure 2-21:

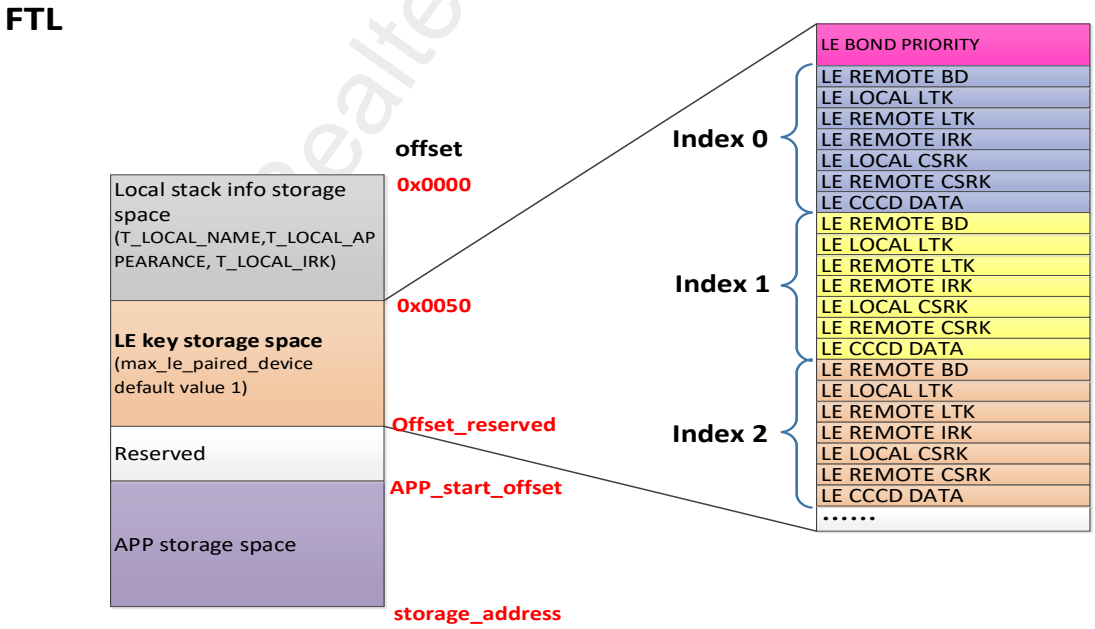

Figure 2-21 LE FTL Layout

LE key storage space can be divided into two regions:

1. **LE BOND PRIORITY**: LE priority control block. For detailed information please refers to *Bonded Device Priority Manager*.
2. **Bonded device keys storage block**: Device index 0, index 1 and so on.
   - LE REMOTE BD: Save remote device address
   - LE LOCAL LTK: Save local Long Term Key (LTK)
   - LE REMOTE LTK: Save remote LTK
   - LE REMOTE IRK: Save remote IRK
   - LE LOCAL CSRK: Save local Connection Signature Resolving Key (CSRK)
   - LE REMOTE CSRK: Save remote CSRK
   - LE CCCD DATA: Save Client Characteristic Configuration declaration (CCCD) data

2.6.3.2.1 Configuration

The size of LE key storage space is related to the following two parameters:

1. LE Maximum Bonded Device Number
   - Default value is 1.
   - Can be configured by gap_config_max_le_paired_device(). More information please refers to *Parameter Configuration with API*.
2. Maximum CCCD Number
   - Default value is 16.
   - Can be configured by gap_config_ccc_bits_count(). More information please refers to *Parameter Configuration with API*.

2.6.3.2.2 LE Key Entry Structure

GAP layer use structure T_LE_KEY_ENTRY to manage bonded device.

```
#define LE_KEY_STORE_REMOTE_BD_BIT      0x01
#define LE_KEY_STORE_LOCAL_LTK_BIT      0x02
#define LE_KEY_STORE_REMOTE_LTK_BIT     0x04
#define LE_KEY_STORE_REMOTE_IRK_BIT     0x08
#define LE_KEY_STORE_LOCAL_CSRK_BIT     0x10
#define LE_KEY_STORE_REMOTE_CSRK_BIT 0x20
#define LE_KEY_STORE_CCCD_DATA_BIT      0x40
#define LE_KEY_STORE_LOCAL_IRK_BIT      0x80
/** @brief LE key entry */
typedef struct
{
    bool is_used;
```

```
    uint8_t idx;
    uint16_t flags;
    uint8_t local_bd_type;
    uint8_t app_data;
    uint8_t reserved[2];
    T_LE_REMOTE_BD remote_bd;
    T_LE_REMOTE_BD resolved_remote_bd;
} T_LE_KEY_ENTRY;
```

Parameter Description:

- **is_used** - Whether to use.
- **idx** - Device index. GAP layer can use idx to find out storage location in FTL.
- **flags** - LE Key Storage Bits, a bit field that indicates whether the key is existing.
- **local_bd_type** - Local address type used in pairing process. T_GAP_LOCAL_ADDR_TYPE
- **remote_bd** - Remote device address.
- **resolved_remote_bd** - Identity address of remote device.

2.6.3.2.3 LE Key Manager

When local device pairs with remote device or encrypts with bonded device, GAP layer will send GAP_MSG_LE_AUTHEN_STATE_CHANGE to notify authentication state change.

```
void app_handle_authen_state_evt(uint8_t conn_id, uint8_t new_state, uint16_t
                                 cause)
{
    APP_PRINT_INFO2("app_handle_authen_state_evt:conn_id %d, cause 0x%x", conn_id, cause);
    switch (new_state)
    {
        case GAP_AUTHEN_STATE_STARTED:
            {
                APP_PRINT_INFO0("app_handle_authen_state_evt: GAP_AUTHEN_STATE_STARTED");
            }
            break;
        case GAP_AUTHEN_STATE_COMPLETE:
            {
                if (cause == GAP_SUCCESS)
                {
                    APP_PRINT_INFO0("app_handle_authen_state_evt: GAP_AUTHEN_STATE_COMPLETE
                                    pair success");
                }
                else
                {
                    APP_PRINT_INFO0("app_handle_authen_state_evt: GAP_AUTHEN_STATE_COMPLETE
                                    pair failed");
```

```
                    }
                }
                break;
        ......
        }
}
```

GAP_MSG_LE_BOND_MODIFY_INFO is used to notify app that bond information has been modified.

```
typedef struct
{
    T_LE_BOND_MODIFY_TYPE type;
    P_LE_KEY_ENTRY        p_entry;
} T_LE_BOND_MODIFY_INFO;

T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    T_LE_CB_DATA *p_data = (T_LE_CB_DATA *)p_cb_data;
    switch (cb_type)
    {
        case GAP_MSG_LE_BOND_MODIFY_INFO:
            APP_PRINT_INFO1("GAP_MSG_LE_BOND_MODIFY_INFO: type 0x%x",
                            p_data->p_le_bond_modify_info->type);
            break;
        ......
    }
}
```

Type of bond modification is defined as below:

```
typedef enum {
    LE_BOND_DELETE,
    LE_BOND_ADD,
    LE_BOND_CLEAR,
    LE_BOND_FULL,
    LE_BOND_KEY_MISSING,
} T_LE_BOND_MODIFY_TYPE;
```

1. LE_BOND_DELETE

LE_BOND_DELETE message means bond information has been deleted. It will be triggered in following conditions.

- Invoked le_bond_delete_by_idx() function.
- Invoked le_bond_delete_by_bd() function.
- The link encryption failed.
- Key storage space is full, and then information of the lowest priority bond will be deleted.

2. **LE_BOND_ADD**

   LE_BOND_ADD message means a new device is bonded. It will only be triggered at the first time of pairing with remote device.

3. **LE_BOND_CLEAR**

   LE_BOND_CLEAR message means all bond information has been deleted. It will only be triggered after invoking le_bond_clear_all_keys() function.

4. **LE_BOND_FULL**

   LE_BOND_FULL message means key storage space is full and this message will only be triggered when parameter of GAP_PARAM_BOND_KEY_MANAGER is set to true. If so, GAP will not delete keys automatically. Otherwise, GAP will first delete the lowest priority bond information and save current bond information, then trigger LE_BOND_DELETE message.

5. **LE_BOND_KEY_MISSING**

   LE_BOND_KEY_MISSING message means the link encryption is failed and the key is no longer valid. This message will only be triggered when parameter of GAP_PARAM_BOND_KEY_MANAGER is set to true. If so, GAP will not delete the key automatically. Otherwise, GAP will delete the key and trigger LE_BOND_DELETE message.

2.6.3.2.4 Bluetooth LE Device Priority Manager in GAP Layer

1. Pair with a new device
   1) Key storage space is not full
      (1) GAP layer will add the bonded device to priority control block and send LE_BOND_ADD to APP. This added device has highest priority.
   2) Key storage space is full
      (1) When GAP_PARAM_BOND_KEY_MANAGER is true, GAP layer will send LE_BOND_FULL to APP.
      (2) When GAP_PARAM_BOND_KEY_MANAGER is false, GAP layer will remove lowest priority bonded device from priority control block and send LE_BOND_DELETE to APP. Then GAP layer will add the bonded device to priority control block and send LE_BOND_ADD to APP. This added device has highest priority.
2. Encryption with bonded device succeeds

   GAP layer will set this bonded device to highest priority.
3. Encryption with bonded device fails
   1) When GAP_PARAM_BOND_KEY_MANAGER is true, GAP layer will send LE_BOND_KEY_MISSING to APP.
   2) When GAP_PARAM_BOND_KEY_MANAGER is false, GAP layer will remove the bonded device

from priority control block and send LE_BOND_DELETE to APP.

2.6.3.2.5 APIs

```
/* gap_storage_le.h */
P_LE_KEY_ENTRY le_find_key_entry(uint8_t *bd_addr, T_GAP_REMOTE_ADDR_TYPE bd_type);
P_LE_KEY_ENTRY le_find_key_entry_by_idx(uint8_t idx);
uint8_t le_get_bond_dev_num(void);
P_LE_KEY_ENTRY le_get_low_priority_bond(void);
P_LE_KEY_ENTRY le_get_high_priority_bond(void);
bool le_set_high_priority_bond(uint8_t *bd_addr, T_GAP_REMOTE_ADDR_TYPE bd_type);
bool le_resolve_random_address(uint8_t *unresolved_addr, uint8_t *resolved_addr,
                               T_GAP_IDENT_ADDR_TYPE *resolved_addr_type);
bool le_get_cccd_data(T_LE_KEY_ENTRY *p_entry, T_LE_CCCD *p_data);
bool le_gen_bond_dev(uint8_t *bd_addr, T_GAP_REMOTE_ADDR_TYPE bd_type,
                     T_GAP_LOCAL_ADDR_TYPE local_bd_type,
                     uint8_t ltk_length, uint8_t *local_ltk, T_LE_KEY_TYPE
                     key_type, T_LE_CCCD *p_cccd);
bool le_set_privacy_info(T_LE_KEY_ENTRY *p_entry, T_LE_PRIVACY_INFO *p_privacy_info);
bool le_get_privacy_info(T_LE_KEY_ENTRY *p_entry, T_LE_PRIVACY_INFO *p_privacy_info);
bool le_check_privacy_bond(T_LE_KEY_ENTRY *p_entry);

/* gap_bond_le.h */
void le_bond_clear_all_keys(void);
T_GAP_CAUSE le_bond_delete_by_idx(uint8_t idx);
T_GAP_CAUSE le_bond_delete_by_bd(uint8_t *bd_addr, T_GAP_REMOTE_ADDR_TYPE bd_type);
```

2.7 Bluetooth LE Privacy

2.7.1 Specification Introduction

Bluetooth LE supports a feature that reduces the ability to track a LE device over a period of time by changing the Bluetooth device address on a frequent basis.

In order for a device using the privacy feature to reconnect to known devices, the device address, referred to as the private address, must be resolved by the other device. The private address is generated using the device's IRK exchanged during the bonding procedure.

2.7.1.1 Device Address

Devices are identified using a device address. Device addresses may be either a public device address or a random

device address.

The random device address may be of either of the following two sub-types:

- Static address
- Private address

The private address may be of either of the following two sub-types:

- Non-resolvable private address
- Resolvable private address

2.7.1.1.1 Static address

The format of a static address is shown in Figure 2-22.

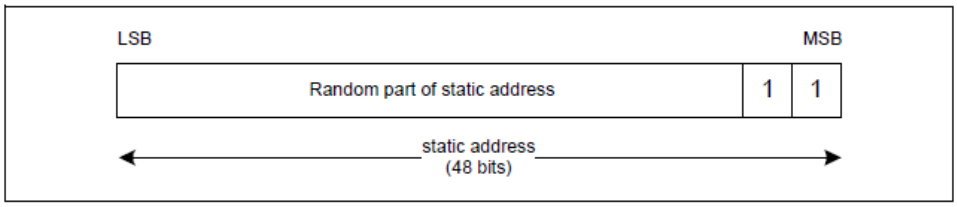

Figure 2-22 Format of static address

2.7.1.1.2 Non-resolvable private address

The format of a non-resolvable private address is shown in Figure 2-23.

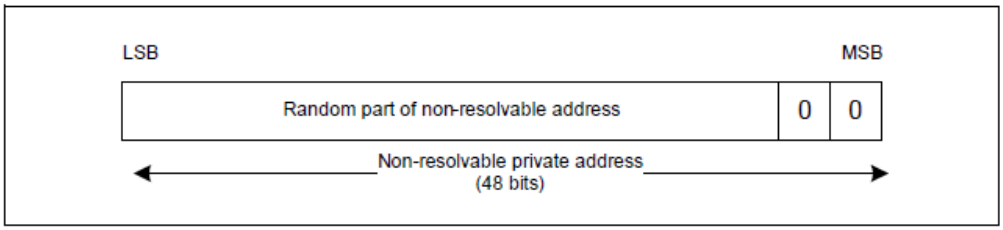

Figure 2-23 Format of non-resolvable private address

2.7.1.1.3 Resolvable private address

The format of the resolvable private address is shown in Figure 2-24.

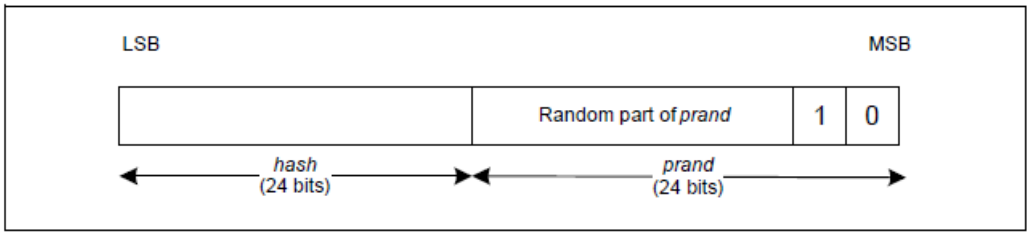

Figure 2-24 Format of resolvable private address

A device's Identity Address is a Public Device Address or Random Static Device Address that it uses in packets it

transmits. If a device is using Resolvable Private Addresses, it shall also have an Identity Address.

To generate a resolvable private address, the device must have either the Local Identity Resolving Key (IRK) or the Peer Identity Resolving Key (IRK). The resolvable private address shall be generated with the IRK and a randomly generated 24-bit number.

A resolvable private address can be resolved by the corresponding device's IRK. If a resolvable private address is resolved, the device can associate this address with the peer device.

2.7.1.2 IRK and Identity Address

The Security Manager (SM) uses a key distribution approach to perform identity and encryption functionalities in radio communication. Identity Resolving Key (IRK) is a 128-bit key used to generate and resolve random addresses. A master that has received IRK from a slave can resolve that slave's random device addresses. A slave that has received IRK from a master can resolve that master's random device addresses. The privacy concept only protects against devices that are not part of the set to which the IRK has been given.

Identity Information is used to distribute the IRK. An all zero Identity Resolving Key data field indicates that a device does not have a valid resolvable private address.

Identity Address Information is used to distribute its public device address or static random address. Figure 2-25 shows an example of all keys and values being distributed by Master and Slave.

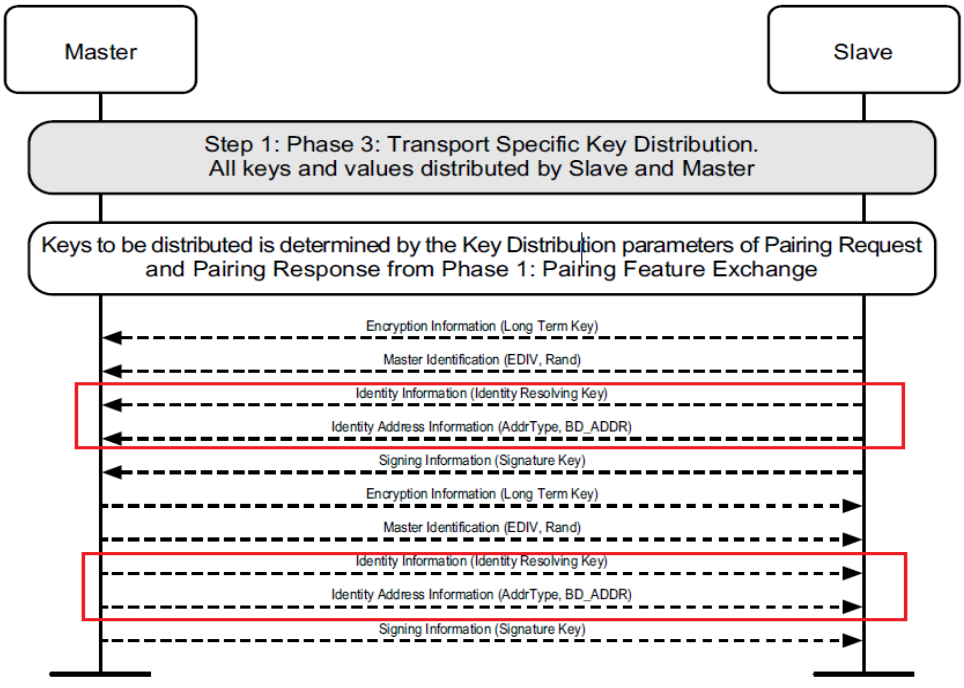

Figure 2-25 Transport Specific Key Distribution

2.7.1.3 Resolving List and Resolution

2.7.1.3.1 Resolving List and White List

The Host maintains a resolving list by adding and removing device identities. A device identity consists of the peer's identity address and a local and peer's IRK pair.

When the Controller performs address resolution and the Host needs to refer to a peer device that is included in the resolving list, it uses the peer's device identity address. Likewise, all incoming events from the Controller to the Host will use the peer's device identity, provided that the peer's device address has been resolved.

Device filtering becomes possible when address resolution is performed in the Controller because the peer's device identity address can be resolved prior to checking whether it is in the White List.

Figure 2-26 shows a logical representation of the relationship between the Controller resolving list and the Controller White List.

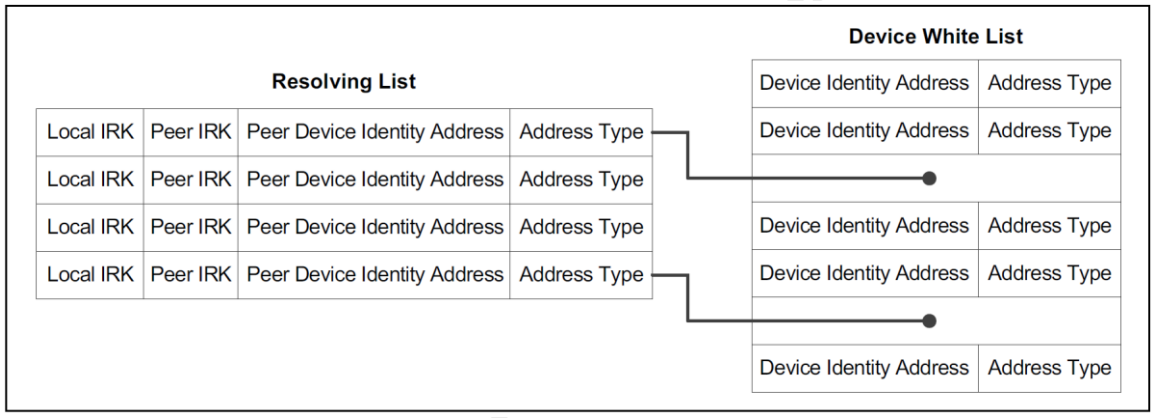

Figure 2-26 Logical Representation of the Resolving List and Device White List

The White List and filter policies set by the Host are applied to the associated Identity Address once the Resolvable Private Address has been resolved.

2.7.1.3.2 Address Resolution

If address resolution is enabled in the Controller, the Controller will use the resolving list whenever the Controller receives a local or peer Resolvable Private Address.

Address resolution can be enabled at any time except when:

- Advertising is enabled
- Scanning is enabled
- Create connection command is outstanding

When address resolution is enabled in the Controller, all references to peer devices that are included in the resolving list from Host to the Controller shall be done using the peer's device identity address. Likewise, all

incoming events from the Controller to the Host will use the peer's device identity, if the peer's device address has been resolved.

2.7.2 Privacy Management Module

Development of Privacy Management Module is based on header file gap_privacy.h. If user needs to use the Privacy Management Module, user can't use the header file gap_privacy.h. Therefore, user can use header file privacy_mgnt.h to develop the privacy related applications.

Directory of Privacy Management Module files is as follows:

- Source code: sdk\ src\ble\privacy\privacy_mgnt.c
- Header file: sdk\ src\ble\privacy\privacy_mgnt.h

User needs to add the privacy_mgnt.c file in the app group of the project, and add header file directory into the include path.

2.7.2.1 APIs

Interfaces of Privacy Management Module are defined in header file privacy_mgnt.h.

```
/* privacy_mgnt.h APIs */
typedef void(*P_FUN_PRIVACY_STATE_CB)(T_PRIVACY_CB_TYPE type, T_PRIVACY_CB_DATA cb_data);
void privacy_init(P_FUN_PRIVACY_STATE_CB p_fun, bool whitelist);
T_PRIVACY_STATE privacy_handle_resolv_list(void);
void privacy_handle_bond_modify_msg(T_LE_BOND_MODIFY_TYPE type, T_LE_KEY_ENTRY *p_entry,
                                    bool handle_add);
bool privacy_add_device(T_LE_KEY_ENTRY *p_entry);
T_GAP_CAUSE privacy_set_addr_resolution(bool enable);
T_GAP_CAUSE privacy_read_peer_resolv_addr(T_GAP_REMOTE_ADDR_TYPE peer_address_type,
                                          uint8_t *peer_address);
T_GAP_CAUSE privacy_read_local_resolv_addr(T_GAP_REMOTE_ADDR_TYPE peer_address_type,
                                           uint8_t *peer_address);
```

2.7.2.2 Usage of Privacy Management Module

The sample code is located in **BLE peripheral privacy project**.

This chapter is used to show how to use Privacy Management Module.

2.7.2.2.1 Initialization

APP needs to call privacy_init() to initialize the Privacy Management Module and register callback function.

```
void app_le_gap_init(void)
{
    ......
    /* register gap message callback */
    le_register_app_cb(app_gap_callback);
#if APP_PRIVACY_EN
    privacy_init(app_privacy_callback, true);
#endif
}
```

The parameter whitelist of privacy_init() is used to configure management of white list. If the parameter whitelist is set to true, Privacy Management Module will manage the white list when Privacy Management Module modifys resolving list. If the parameter whitelist is set to false, application needs to manage the white list.

The callback function is used to handle messages which are sent by Privacy Management Module.

```
void app_privacy_callback(T_PRIVACY_CB_TYPE type, T_PRIVACY_CB_DATA cb_data)
{
    APP_PRINT_INFO1("app_privacy_callback: type %d", type);
    switch (type)
    {
    case PRIVACY_STATE_MSGTYPE:
        app_privacy_state = cb_data.privacy_state;
        APP_PRINT_INFO1("PRIVACY_STATE_MSGTYPE: status %d", app_privacy_state);
        break;
    case PRIVACY_RESOLUTION_STATUS_MSGTYPE:
        app_privacy_resolution_state = cb_data.resolution_state;
        APP_PRINT_INFO1("PRIVACY_RESOLUTION_STATUS_MSGTYPE: status %d",
app_privacy_resolution_state);
        break;
    default:
        break;
    }
}
```

2.7.2.2.2 Resolving List Management

The key function of Privacy Management Module is used to manage the resolving list and white list when bonding informations are changed. The white list management is an optional feature. The parameter whitelist of privacy_init() is used to configure management of white list.

Application shall call function privacy_handle_bond_modify_msg() when application handles message GAP_MSG_LE_BOND_MODIFY_INFO.

Privacy Management Module will handle the resolving list according to modification type of bonding information, and process is shown as follows:

- LE_BOND_DELETE: Delete the device from the resolving list.
- LE_BOND_ADD: Add the device to the resolving list.
- LE_BOND_CLEAR: Clear the resolving list.

```
T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    T_LE_CB_DATA *p_data = (T_LE_CB_DATA *)p_cb_data;
    switch (cb_type)
    {
#if APP_PRIVACY_EN
    case GAP_MSG_LE_BOND_MODIFY_INFO:
        APP_PRINT_INFO1("GAP_MSG_LE_BOND_MODIFY_INFO: type 0x%x",
                        p_data->p_le_bond_modify_info->type);
        privacy_handle_bond_modify_msg(p_data->p_le_bond_modify_info->type,
                                       p_data->p_le_bond_modify_info->p_entry, true);
        break;
#endif
    ……
}
```

Modification procedure of resolving list cannot be executed when address resolution is enabled and:

- Advertising is enabled
- Scanning is enabled
- Create connection command is outstanding

Modification procedure of resolving list can be executed at any time when address resolution is disabled.

Application shall call function privacy_handle_resolv_list() when the gap device state is idle state. The function privacy_handle_resolv_list() handles the pending modification procedure of resolving list.

```
void app_handle_dev_state_evt(T_GAP_DEV_STATE new_state, uint16_t cause)
{
    APP_PRINT_INFO3("app_handle_dev_state_evt: init state %d, adv state %d, cause 0x%x",
                    new_state.gap_init_state, new_state.gap_adv_state, cause);
#if APP_PRIVACY_EN
    if ((new_state.gap_init_state == GAP_INIT_STATE_STACK_READY)
        && (new_state.gap_adv_state == GAP_ADV_STATE_IDLE)
        && (new_state.gap_conn_state == GAP_CONN_DEV_STATE_IDLE))
    {
        privacy_handle_resolv_list();
    }
#endif
    .....
}
```

The callback type PRIVACY_STATE_MSGTYPE is used to inform the state of the resolving list modification

procedure. The callback data is T_PRIVACY_STATE.

```
typedef enum
{
    PRIVACY_STATE_INIT,    //!< Privacy management module is not initialization.
    PRIVACY_STATE_IDLE,    //!< Idle. No pending resolving list modification procedure.
    PRIVACY_STATE_BUSY     //!< Busy. Resolving list modification procedure is not completed.
} T_PRIVACY_STATE;
```

2.7.2.2.3 Address Resolution

If peer device uses the resolvable private address, and application wants to use the white list to filter this peer device. Then, application needs to call function privacy_set_addr_resolution() to enable the address resolution.

If application wants to pair with new devices, application needs to call function privacy_set_addr_resolution() to disable the address resolution. When the address resolution is enabled, the local device cannot connect with the device which is not in the resolving list.

The sample code is shown as follows.

```
void app_adv_start(void)
{
    uint8_t  adv_evt_type = GAP_ADTYPE_ADV_IND;
    uint8_t  adv_filter_policy = GAP_ADV_FILTER_ANY;
#if APP_PRIVACY_EN
    T_LE_KEY_ENTRY *p_entry;
    p_entry = le_get_high_priority_bond();

    if (p_entry == NULL)
    {
        /* No bonded device, send connectable undirected advertisement without using whitelist*/
        app_work_mode = APP_PAIRABLE_MODE;
        adv_filter_policy = GAP_ADV_FILTER_ANY;
        if (app_privacy_resolution_state == PRIVACY_ADDR_RESOLUTION_ENABLED)
        {
            privacy_set_addr_resolution(false);
        }
    }
    else
    {
        app_work_mode = APP_RECONNECTION_MODE;
        adv_filter_policy = GAP_ADV_FILTER_WHITE_LIST_ALL;
        if (app_privacy_resolution_state == PRIVACY_ADDR_RESOLUTION_DISABLED)
        {
            privacy_set_addr_resolution(true);
        }
```

```
    }
#endif
    le_adv_set_param(GAP_PARAM_ADV_EVENT_TYPE, sizeof(adv_evt_type), &adv_evt_type);
    le_adv_set_param(GAP_PARAM_ADV_FILTER_POLICY, sizeof(adv_filter_policy), &adv_filter_policy);
    le_adv_start();
}
```

The function privacy_set_addr_resolution() is used to enable resolution of Resolvable Private Addresses in the Controller. This causes the Controller to use the resolving list whenever the Controller receives a local or peer Resolvable Private Address.

This function can be used at any time except when:

- Advertising is enabled
- Scanning is enabled
- Create connection command is outstanding

The callback type PRIVACY_RESOLUTION_STATUS_MSGTYPE is used to inform the state of address resolution. The callback data is T_PRIVACY_ADDR_RESOLUTION_STATE.

```
/** @brief Define the privacy address resolution state */
typedef enum
{
    PRIVACY_ADDR_RESOLUTION_DISABLED,
    PRIVACY_ADDR_RESOLUTION_DISABLING,
    PRIVACY_ADDR_RESOLUTION_ENABLING,
    PRIVACY_ADDR_RESOLUTION_ENABLED
} T_PRIVACY_ADDR_RESOLUTION_STATE;
```

3 GATT Profile

GATT Profile APIs based on GATT specification are provided in SDK. The implementation of GATT Based Profile consists of two components: Profile-Server and Profile-Client.

Profile-Server is a public interface abstracted from implementation of server terminal of GATT Based Profile. More information could be found in chapter *Bluetooth LE Profile Server*.

Profile-Client is a public interface abstracted from implementation of client terminal of GATT Based Profile. More information could be found in chapter *Bluetooth LE Profile Client*.

GATT Profile Layer has been implemented in ROM, and provides interfaces to application. Header files are provided in SDK.

GATT Profile header files directory: sdk\inc\bluetooth\profile.

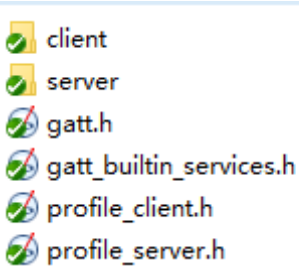

Figure 3-1 GATT Profile Header Files

3.1 Bluetooth LE Profile Server

3.1.1 Overview

Server is the device that accepts incoming commands and requests from the client and sends responses, indications and notifications to a client. GATT profile defines how Bluetooth LE devices transmit data between GATT server and GATT client. Profile may contain one or more GATT services, service is a group of characteristics in set, through which GATT server exposes its characteristics.

Profile Server exports APIs that user can use to implement a specific service.

Figure 3-2 shows the profile server hierarchy.

Content of profile involves profile server layer and specific service. Profile server layer above protocol stack encapsulates interfaces for specific service to access protocol stack. So that, development of specific services does not involve details of protocol stack process and becomes simpler and clearer. Specific service is implemented by application layer which based on the profile server layer. The specific service consists of attribute value and provides interfaces for application to transmit data.

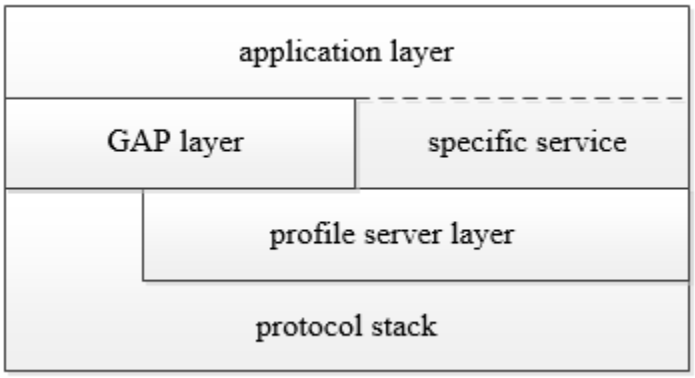

Figure 3-2 Profile Server Hierarchy

3.1.2 Supported Profile and Service

Supported profile list is shown in Table 3-1.

Table 3-1 Supported Profile List

Abbr.	Definition	GATT server	GATT client
GAP	Generic Access Profile	Server role shall support GAS(M)	client role has no claim
PXP	Proximity Profile	Proximity Reporter role shall support LLS(M), IAS(O), TPS(O)	Proximity Monitor role has no claim
ScPP	Scan Parameters Profile	Scan Server role shall support ScPS(M)	Scan Client role has no claim
HTP	Health Thermometer Profile	Thermometer role shall support HTS(M), DIS(M)	Collector role has no claim
HRP	Heart Rate Profile	Heart Rate Sensor role shall support HRS(M), DIS(M)	Collector role has no claim
LNP	Location and Navigation Profile	LN Sensor role shall support LNS(M), DIS(O), BAS(O)	Collector role has no claim
WSP	Weight Scale Profile	Weight Scale role shall support WSS(M), DIS(M), BAS(O)	Weight Scale role shall support WSS(M), DIS(M), BAS(O)
GLP	Glucose Profile	Glucose Sensor role shall support GLS(M), DIS(M)	Collector role has no claim
FMP	Fine Me Profile	Find Me Target role shall support IAS(M)	Find Me Locator role has no claim
HOGP	HID over GATT Profile	HID Device shall support HIDS(M), BAS(M), DIS(O), ScPS(O)	Boot Host has no claim
RSCP	Running Speed and Cadence	RSC Sensor role shall support	Collector role has no

	Profile	RSCS(M), DIS(M)	claim
CSCP	Cycling Speed and Cadence Profile	CSC Sensor role shall support CSCS(M), DIS(M)	Collector role has no claim
IPSP	Internet Protocol Support Profile	Node role shall support IPSS(M)	Router role has no claim

NOTE:

M: mandatory

O: optional

Supported service list is shown in Table 3-2.

Table 3-2 Supported Service List

Abbr.	Definition	**Files**
GATTS	Generic Attribute Service	gatt_builtin_services.h
GAS	Generic Access Service	gatt_builtin_services.h
BAS	Battery Service	bas.c, bas.h bas_config.h
DIS	Device Information Service	dis.c, dis.h dis_config.h
ScPS	Scan Parameters Service	sps.c, sps.h sps_config.h
HIDS	Human Interface Device Service	hids.c, hids.h hids_kb.c, hids_kb.h hids_ms.c, hids_ms.h
TPS	Tx Power Service	tps.c, tps.h
IAS	Immediate Alert Service	ias.c, ias.h
LLS	Link Loss Service	lls.c, lls.h
HTS	Health Thermometer Service	hts.c, hts.h
HRS	Heart Rate Service	hrs.c, hrs.h
LNS	Location and Navigation Service	lns.c, lns.h
WSS	Weight Scale Service	wss.c, wss.h
RSCS	Running Speed and Cadence Service	rscs.c, rscs.h
CSCS	Cycling Speed and Cadence Service	cscs.c, cscs.h cscs_config.h
GLS	Glucose Service	gls.c, gls.h gls_config.h
IPSS	Internet Protocol Support Service	ipss.c, ipss.h

3.1.3 Profile Server Interaction

Profile server layer handles interaction with protocol stack layer, and provides interfaces to design specific service. Profile server interactions include adding service to server, characteristic value read, characteristic value write, characteristic value notification and characteristic value indication.

3.1.3.1 Add service

Protocol stack maintains information of all services which are added from profile server layer. The number of total service attribute table to be added shall be initialized first, profile server layer provides server_init() function to initialize service table number.

```
void app_le_profile_init(void)
{
    server_init(2);
    simp_srv_id = simp_ble_service_add_service(app_profile_callback);
    bas_srv_id  = bas_add_service(app_profile_callback);
    server_register_app_cb(app_profile_callback);
    ...
}
```

Profile server layer provides server_add_service() interface to add services to profile server layer.

```
T_SERVER_ID simp_ble_service_add_service(void *p_func)
{
    if (false == server_add_service(&simp_service_id,
                                    (uint8_t *)simple_ble_service_tbl,
                                    sizeof(simple_ble_service_tbl),
                                    simp_ble_service_cbs))
    {
        APP_PRINT_ERROR0("simp_ble_service_add_service: fail");
        simp_service_id = 0xff;
        return simp_service_id;
    }
    pfn_simp_ble_service_cb = (P_FUN_SERVER_GENERAL_CB)p_func;
    return simp_service_id;
}
```

Figure 3-3 shows a server contains serval service tables.

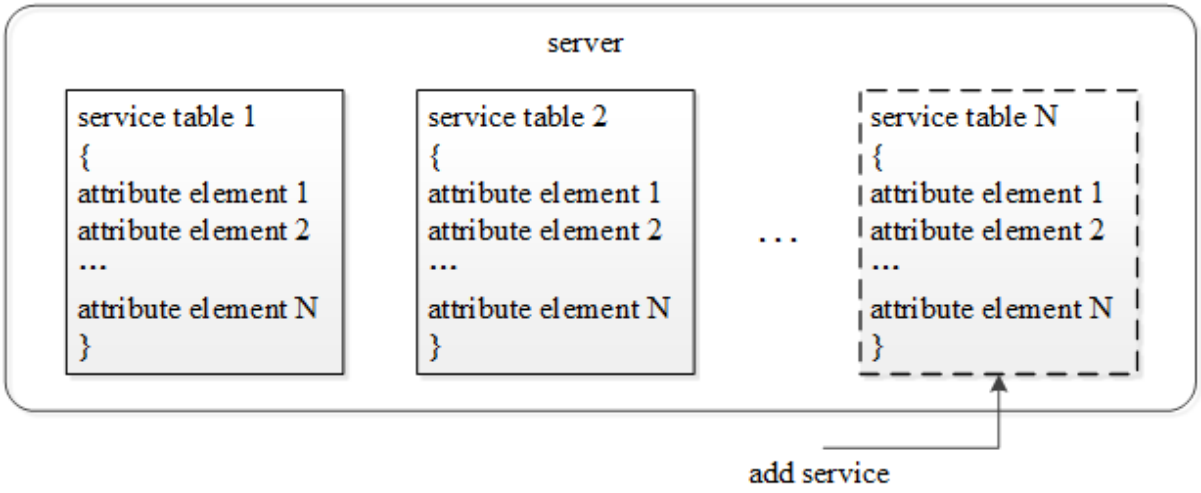

Figure 3-3 Add Services to Server

After service is added to profile server layer, all services will be registered during GAP initialization procedure. GAP layer will send message PROFILE_EVT_SRV_REG_COMPLETE to GAP layer upon completing registeration process.

Register service's process is shown in Figure 3-4.

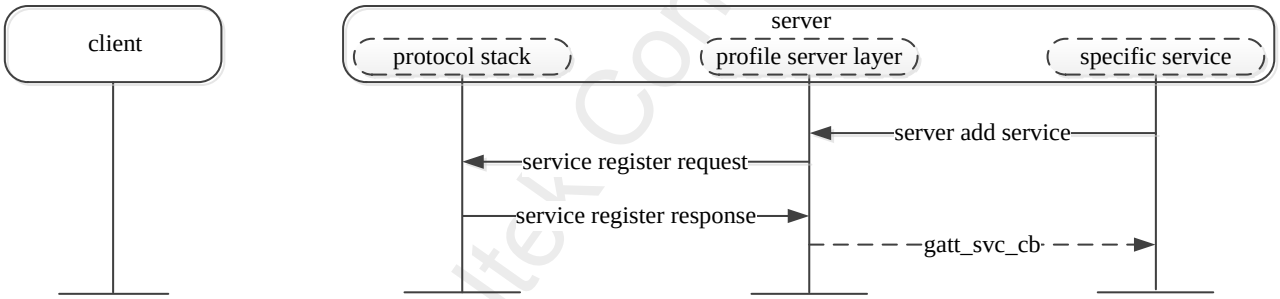

Figure 3-4 Register Service's Process

Registration of services is started by sending service register request to protocol stack during initialization of GAP, and then register all services which have been added. If server general callback function is not NULL, once the last service is registered properly, profile server layer will send PROFILE_EVT_SRV_REG_COMPLETE message to application through registered callback function app_profile_callback().

```
T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    if (service_id == SERVICE_PROFILE_GENERAL_ID)
    {
        T_SERVER_APP_CB_DATA *p_param = (T_SERVER_APP_CB_DATA *)p_data;
        switch (p_param->eventId)
        {
```

```
            case PROFILE_EVT_SRV_REG_COMPLETE:// srv register result event.
                APP_PRINT_INFO1("PROFILE_EVT_SRV_REG_COMPLETE: result %d",
                                p_param->event_data.service_reg_result);
                break;
            ...
}
```

3.1.3.2 Service's Callback

3.1.3.2.1 Server General Callback

Server general callback function is used to send events to application, including service register complete event and send data complete event through characteristic value notification or indication.

This callback function shall be initialized with server_register_app_cb() function.

Server general callback function is defined in profile_server.h.

3.1.3.2.2 Specific Service Callback

For some attributes' value is supplied by application, to access these attributes' value, service's callback functions shall be implemented in specific service, which are used to handle read/write attribute value and update CCCD value process from client.

This callback function struct shall be initialized with server_add_service() function.

Attribute element structure is defiend in profile_server.h.

```
/* service related callback functions struct */
typedef struct {
    P_FUN_GATT_READ_ATTR_CB read_attr_cb;          // Read callback function pointer
    P_FUN_GATT_WRITE_ATTR_CB write_attr_cb;        // Write callback function pointer
    P_FUN_GATT_CCCD_UPDATE_CB cccd_update_cb;      // update cccd callback function pointer
} T_FUN_GATT_SERVICE_CBS;
```

read_attr_cb: Attribute read callback, which is used to acquire value of attribute supplied by application when attribute read request is sent from client side.

write_attr_cb: Attribute write callback, which is used to write value to attribute supplied by application when attribute write request is sent from client side.

cccd_update_cb: Client characteristic configuration descriptor value update callback, which is used to inform application that the value of corresponding CCCD in service is written by client.

```
const T_FUN_GATT_SERVICE_CBS simp_ble_service_cbs =
{
    simp_ble_service_attr_read_cb,   // Read callback function pointer
    simp_ble_service_attr_write_cb, // Write callback function pointer
    simp_ble_service_cccd_update_cb // CCCD update callback function pointer
```

```
};
```

3.1.3.2.3 Write Indication Post Procedure Callback

Write indication post procedure callback function is used to execute some post procedure after handle write request from client.

This callback function is initialized in write attribute callback function. If no post procedure will be executed, the pointer of p_write_post_proc in write attribute callback function shall be assigned with null.

Write indication post procedure callback function is defined in profile_server.h.

3.1.3.3 Characteristic Value Read

This procedure is used to read a characteristic value from a server. There are four sub-procedures that can be used to read a characteristic value, including read characteristic value, read using characteristic UUID, read long characteristic values and read multiple characteristic values. If an attribute want to be readable, it shall be configured with readable permissions. Attribute value can be read from service or application by using different attribute flag.

3.1.3.3.1 Attribute Value Supplied in Attribute Element

The attribute with flag ATTRIB_FLAG_VALUE_INCL will be involved in this procedure.

```
{
    ATTRIB_FLAG_VALUE_INCL,                         /* flags */
    {                                               /* type_value */
        LO_WORD(0x2A04),
        HI_WORD(0x2A04),
        100,
        200,
        0,
        LO_WORD(2000),
        HI_WORD(2000)
    },
    5,                                              /* bValueLen */
    NULL,
    GATT_PERM_READ                                  /* permissions */
},
```

The iinteraction between each layer is shown in Figure 3-5. Protocol stack layer will read value from attribute element and respond this attribute value in read response directly.

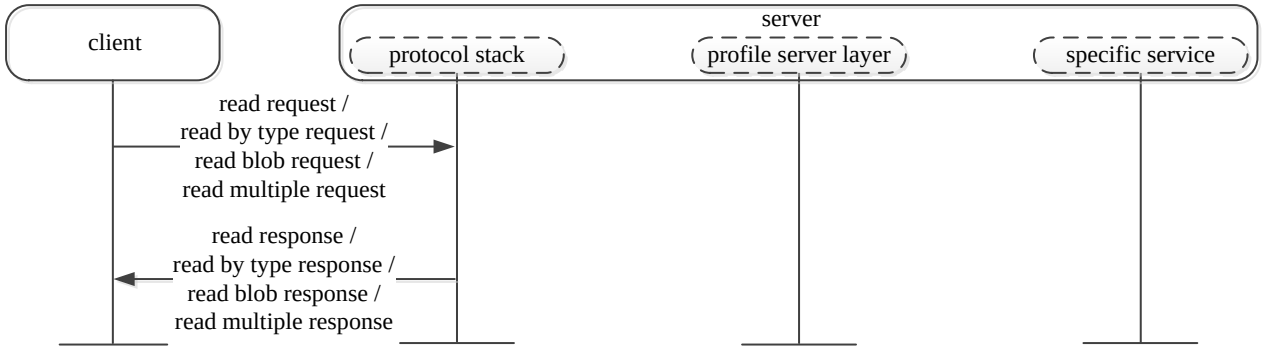

Figure 3-5 Read Characteristic Value - Attribute Value Supplied in Attribute Element

3.1.3.3.2 Attribute Value Supplied by Application without Result Pending

The attribute with flag ATTRIB_FLAG_VALUE_APPL will be involvde in this procedure.

```
{
    ATTRIB_FLAG_VALUE_APPL,
    {
        LO_WORD(GATT_UUID_CHAR_SIMPLE_V1_READ),
        HI_WORD(GATT_UUID_CHAR_SIMPLE_V1_READ)
    },
    0,
    NULL,
    GATT_PERM_READ
},
```

The interation between each layer is shown in Figure 3-6. When local device receives read request, protocol stack will send read indication to profile server layer. Profile server layer will get the value in specific service by calling read attribute callback. Afterwards, profile server layer will return the data to protocol stack through read confirmation.

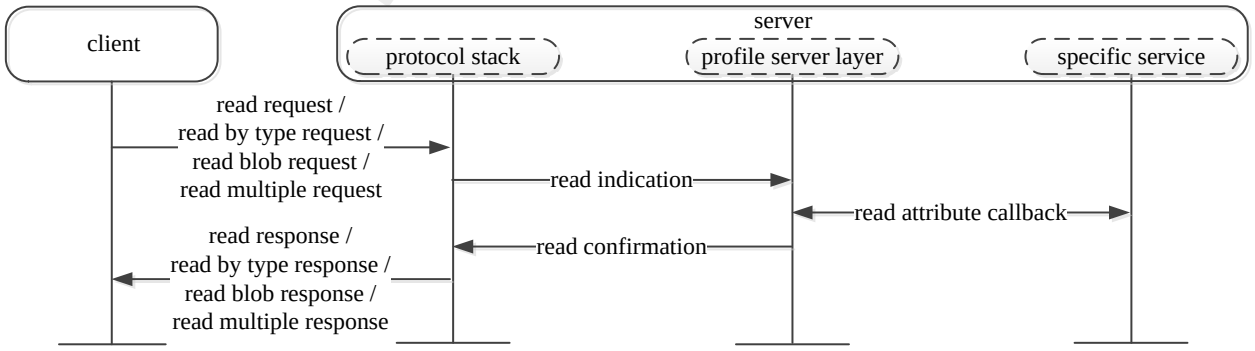

Figure 3-6 Read Characteristic Value - Attribute Value Supplied by Application without Result Pending

The sample code is shown as follows, app_profile_callback shall return APP_RESULT_SUCCESS:

```
T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    ...
    else  if (service_id == simp_srv_id)
    {
        TSIMP_CALLBACK_DATA *p_simp_cb_data = (TSIMP_CALLBACK_DATA *)p_data;
        switch (p_simp_cb_data->msg_type)
        {
            case SERVICE_CALLBACK_TYPE_READ_CHAR_VALUE:
                {
                    if (p_simp_cb_data->msg_data.read_value_index == SIMP_READ_V1)
                    {
                        uint8_t value[2] = {0x01, 0x02};
                        APP_PRINT_INFO0("SIMP_READ_V1");
                        simp_ble_service_set_parameter(
                                SIMPLE_BLE_SERVICE_PARAM_V1_READ_CHAR_VAL, 2, &value);
                     }
                }
                break;
        }
        ...
        return app_result;
}
```

3.1.3.3.3 Attribute Value Supplied by Application with Result Pending

The attribute with flag ATTRIB_FLAG_VALUE_APPL will be involved in this procedure.

Attribute value from application can't be read immediately, so it should be transmitted by invoking server_attr_read_confirm() in specific service. The interaction between each layer is shown in Figure 3-7.

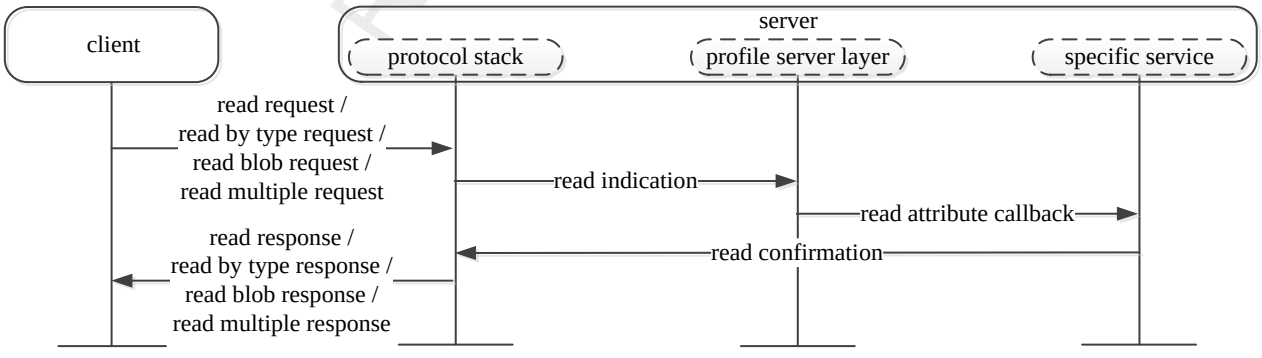

Figure 3-7 Read Characteristic Value - Attribute Value Supplied by Application with Result Pending

The sample code is shown as follows, app_profile_callback() shall return APP_RESULT_PENDING:

```
T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
```

```
{
    T_APP_RESULT app_result = APP_RESULT_PENDING;
    ...
    else if (service_id == simp_srv_id)
    {
        TSIMP_CALLBACK_DATA *p_simp_cb_data = (TSIMP_CALLBACK_DATA *)p_data;
        switch (p_simp_cb_data->msg_type)
        {
            case SERVICE_CALLBACK_TYPE_READ_CHAR_VALUE:
                {
                    if (p_simp_cb_data->msg_data.read_value_index == SIMP_READ_V1)
                    {
                        uint8_t value[2] = {0x01, 0x02};
                        APP_PRINT_INFO0("SIMP_READ_V1");
                        simp_ble_service_set_parameter(
                            SIMPLE_BLE_SERVICE_PARAM_V1_READ_CHAR_VAL, 2, &value);
                     }
                }
                break;
        }
        ...
        return app_result;
    }
```

3.1.3.4 Characteristic Value Write

This procedure is used to write a characteristic value to a server. There are four sub-procedures that can be used to write a characteristic value, including write without response, signed write without response, write characteristic value and write long characteristic values.

3.1.3.4.1 Write Characteristic Value

1. Attribute Value Supplied in Attribute Element

The attribute with flag ATTRIB_FLAG_VOID will be involved in this procedure.

```
uint8_t cha_val_v8_011[1] = {0x08};
const T_ATTRIB_APPL gatt_dfindme_profile[] = {
    ......
    /* handle = 0x000e Characteristic value    -- Value V8 */
    {
        ATTRIB_FLAG_VOID,                               /* flags */
        {                                               /* type_value */
            LO_WORD(0xB008),
```

```
            HI_WORD(0xB008),
        },
        1,                              /* bValueLen */
        (void *)cha_val_v8_011,
        GATT_PERM_READ | GATT_PERM_WRITE                    /* permissions */
    },
    ......
}
```

The procedure executing between each layer is shown in Figure 3-8. The write request is used to request the server to write the value of an attribute and acknowledge that write operation has been achieved with a write response directly.

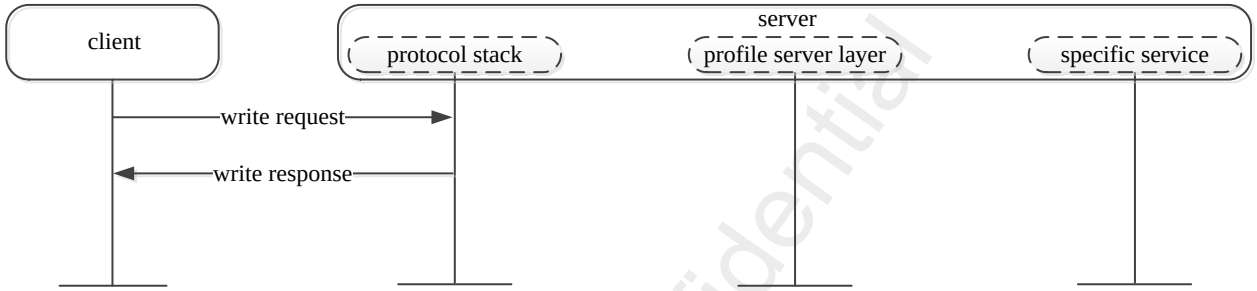

Figure 3-8 Write Characteristic Value - Attribute Value Supplied in Attribute Element

2. Attribute Value Supply by Application without Result Pending

The attribute with flag **ATTRIB_FLAG_VALUE_APPL** will be involved in this procedure.

The interaction between each layer is shown in Figure 3-9. When local device receives write request, protocol stack will send write request indication to profile server layer, and profile server layer will write the value to specific service by calling write attribute callback. Profile server layer will return write result through write request confirmation.

If server need to execute subsequent procedure after profile server layer returns write confirmation, the pointer of callback function write_ind_post_proc() will be invoked if it isn't null.

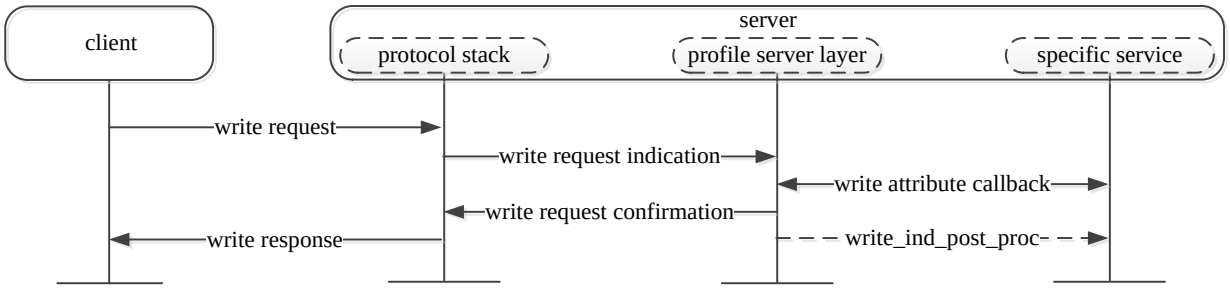

Figure 3-9 Write Characteristic Value - Attribute Value Supplied by Application without Result Pending

Application will be notified with srv_cbs registered by server_add_service(), and the write_type will be

WRITE_REQUEST.

The sample code is shown as follows, app_profile_callback() shall return with result APP_RESULT_SUCCESS:

```
T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    ...
    else  if (service_id == simp_srv_id)
    {
        TSIMP_CALLBACK_DATA *p_simp_cb_data = (TSIMP_CALLBACK_DATA *)p_data;
        switch (p_simp_cb_data->msg_type)
        {
            case SERVICE_CALLBACK_TYPE_WRITE_CHAR_VALUE:
                {
                    switch (p_simp_cb_data->msg_data.write.opcode)
                    {
                        case SIMP_WRITE_V2:
                            {
                                APP_PRINT_INFO2("SIMP_WRITE_V2: write type %d, len %d",
                                                p_simp_cb_data->msg_data.write.write_type,
                                                p_simp_cb_data->msg_data.write.len);
                            }
    ...
    return app_result;
}
```

3. Attribute Value Supply by Application with Result Pending

The attribute of flag is ATTRIB_FLAG_VALUE_APPL will be involved in this procedure.

If write attribute value process cannot be completed immediately, server_attr_write_confirm() shall be invoked by specific service. The interaction between each layer is shown in Figure 3-10.

Write indication post procedure is optional.

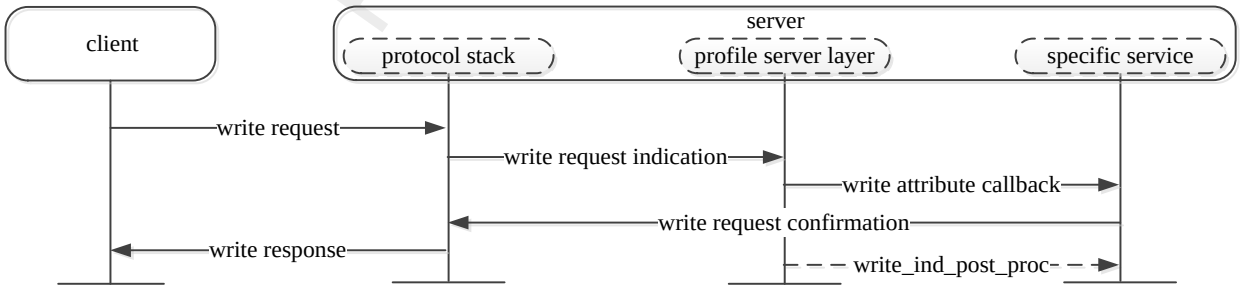

Figure 3-10 Write Characteristic Value - Attribute Value Supplied by Application with Result Pending

Application will be notified with srv_cbs registered by server_add_service(), and the write_type will be WRITE_REQUEST.

The sample code is shown as follows, app_profile_callback() shall return APP_RESULT_PENDING:

```
T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_PENDING;
    ...
    else  if (service_id == simp_srv_id)
    {
        TSIMP_CALLBACK_DATA *p_simp_cb_data = (TSIMP_CALLBACK_DATA *)p_data;
        switch (p_simp_cb_data->msg_type)
        {
            case SERVICE_CALLBACK_TYPE_WRITE_CHAR_VALUE:
                {
                    switch (p_simp_cb_data->msg_data.write.opcode)
                    {
                        case SIMP_WRITE_V2:
                            {
                                APP_PRINT_INFO2("SIMP_WRITE_V2: write type %d, len %d",
                                                p_simp_cb_data->msg_data.write.write_type,
                                                p_simp_cb_data->msg_data.write.len);
                            }
    ...
    return app_result;
}
```

4. Write CCCD Value

If local device receives write request from client of writing characteristic configuration declaration, protocol stack updates CCCD information. Afterwards, profile server layer informs APP that CCCD information has been updated by update CCCD callback function. The interaction between each layer is shown in Figure 3-11.

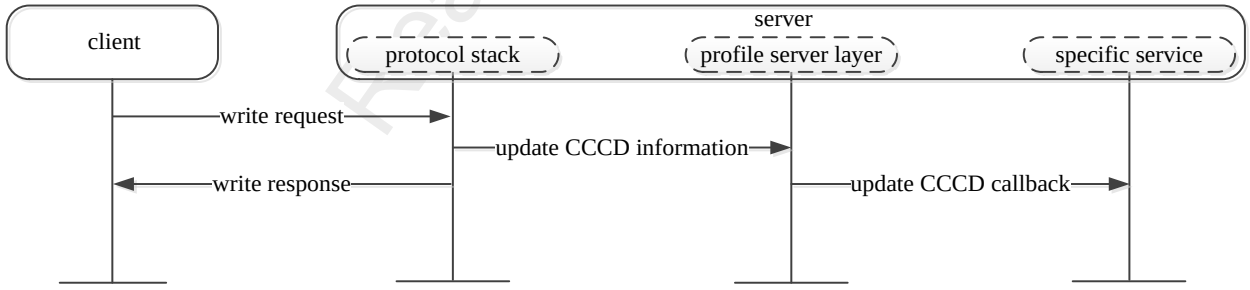

Figure 3-11 Write Characteristic Value – Write CCCD Value

```
void simp_ble_service_cccd_update_cb(uint8_t conn_id, T_SERVER_ID service_id, uint16_t index,
                                     uint16_t cccbits)
{
    TSIMP_CALLBACK_DATA callback_data;
    bool is_handled = false;
```

```
    callback_data.conn_id = conn_id;
    callback_data.msg_type = SERVICE_CALLBACK_TYPE_INDIFICATION_NOTIFICATION;
    APP_PRINT_INFO2("simp_ble_service_cccd_update_cb: index = %d, cccbits 0x%x", index, cccbits);
    switch (index)
    {
    case SIMPLE_BLE_SERVICE_CHAR_NOTIFY_CCCD_INDEX:
    {
        if (cccbits & GATT_CLIENT_CHAR_CONFIG_NOTIFY)
        {
            // Enable Notification
            callback_data.msg_data.notification_indification_index = SIMP_NOTIFY_INDICATE_V3_ENABLE;
        }
        else
        {
            // Disable Notification
            callback_data.msg_data.notification_indification_index = SIMP_NOTIFY_INDICATE_V3_DISABLE;
        }
        is_handled =  true;
    }
    break;
    case SIMPLE_BLE_SERVICE_CHAR_INDICATE_CCCD_INDEX:
    {
        if (cccbits & GATT_CLIENT_CHAR_CONFIG_INDICATE)
        {
            // Enable Indication
            callback_data.msg_data.notification_indification_index = SIMP_NOTIFY_INDICATE_V4_ENABLE;
        }
        else
        {
            // Disable Indication
            callback_data.msg_data.notification_indification_index = SIMP_NOTIFY_INDICATE_V4_DISABLE;
        }
        is_handled =  true;
    }
    break;
    default:
        break;
    }
    /* Notify Application. */
    if (pfn_simp_ble_service_cb && (is_handled == true))
    {
        pfn_simp_ble_service_cb(service_id, (void *)&callback_data);
    }
}
```

Application will be notified with srv_cbs registered by server_add_service(), and the msg_type will be SERVICE_CALLBACK_TYPE_INDIFICATION_NOTIFICATION.

```
T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    ...
    else  if (service_id == simp_srv_id)
    {
        TSIMP_CALLBACK_DATA *p_simp_cb_data = (TSIMP_CALLBACK_DATA *)p_data;
        switch (p_simp_cb_data->msg_type)
        {
            switch (p_simp_cb_data->msg_type)
            {
            case SERVICE_CALLBACK_TYPE_INDIFICATION_NOTIFICATION:
                {
                switch (p_simp_cb_data->msg_data.notification_indification_index)
                {
                case SIMP_NOTIFY_INDICATE_V3_ENABLE:
                    {
                        APP_PRINT_INFO0("SIMP_NOTIFY_INDICATE_V3_ENABLE");
                    }
                break;
                case SIMP_NOTIFY_INDICATE_V3_DISABLE:
                    {
                        APP_PRINT_INFO0("SIMP_NOTIFY_INDICATE_V3_DISABLE");
                    }
                break;
                case SIMP_NOTIFY_INDICATE_V4_ENABLE:
                    {
                        APP_PRINT_INFO0("SIMP_NOTIFY_INDICATE_V4_ENABLE");
                    }
                break;
                case SIMP_NOTIFY_INDICATE_V4_DISABLE:
                    {
                        APP_PRINT_INFO0("SIMP_NOTIFY_INDICATE_V4_DISABLE");
                    }
                break;
                default:
                    break;
                }
            }
            break;
            }
```

```
        ...
        return app_result;
    }
```

3.1.3.4.2 Write without Response / Signed Write without Response

The difference between write without response / signed write without response procedure and write characteristic value procedure is server shall not response write result to client.

1. Attribute Value Supplied by Application

The attribute with flag **ATTRIB_FLAG_VALUE_APPL** will be involved in this procedure.

The procedure executing between each layer is shown in Figure 3-12.

When local device receives write command or signed write command, write_attr_cb() registered by server_add_service() will be called.

Application will be notified with srv_cbs registered by server_add_service(), and the write_type will be WRITE_WITHOUT_RESPONSE or WRITE_SIGNED_WITHOUT_RESPONSE.

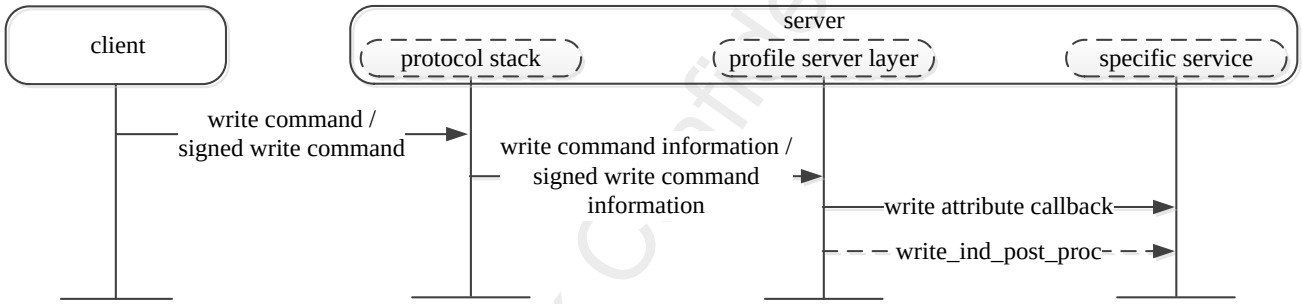

Figure 3-12 Write without Response / Signed Write without Response - Attribute Value Supplied by Application

3.1.3.4.3 Write Long Characteristic Values

1. Prepare Write

If the length of characteristic value is longer than the supported maximum length (ATT_MTU - 3) of characteristic value in a write request attribute protocol message, prepare write request will be used by client. The value to be written is stored in profile server layer buffer first, then profile server layer handle prepare write request indication, and respond write prepare write confirmation.

This procedure executing between each layer is shown in Figure 3-13.

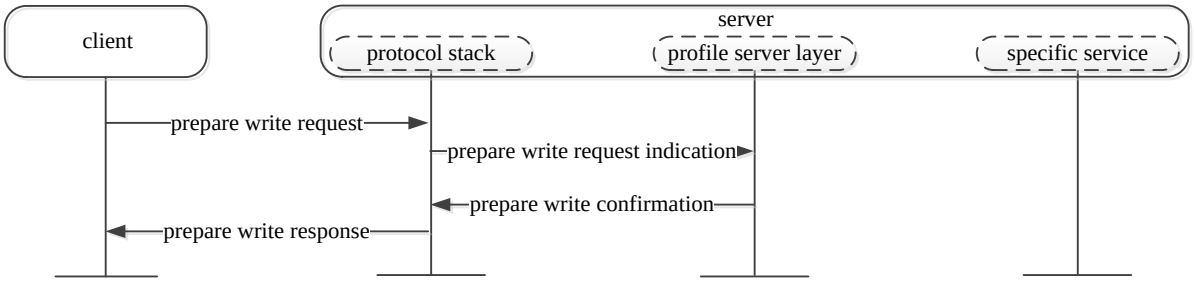

Figure 3-13 Write Long Characteristic Values – Prepare Write Procedure

2. Execute Write without Result Pending

After sending prepare write request, execute write request is used to complete the process of writing attribute value. Application will be notified with srv_cbs registered by server_add_service(), and the write_type will be WRITE_LONG.

Write indication post procedure is optional.

This procedure executing between each layer is shown in Figure 3-14.

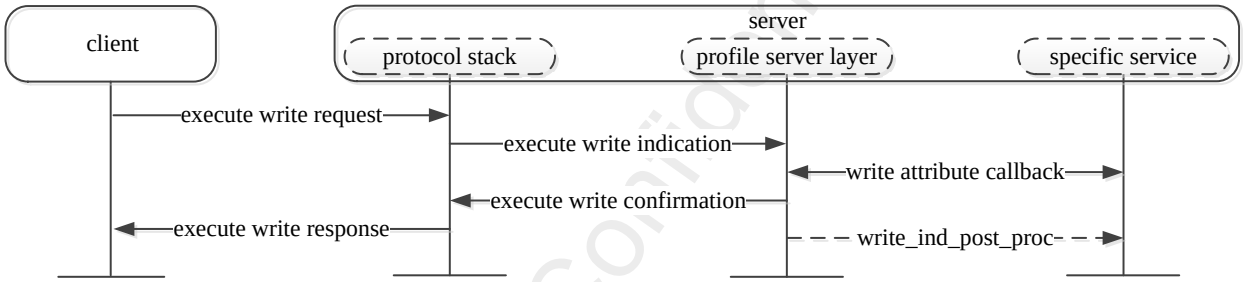

Figure 3-14 Write Long Characteristic Values– Execute Write without Result Pending

3. Execute Write with Result Pending

If the process of writing value can't be completed immediately, server_exec_write_confirm() shall be invoked by specific service.

Write indication post procedure is optional.

This interaction between each layer is shown in Figure 3-15.

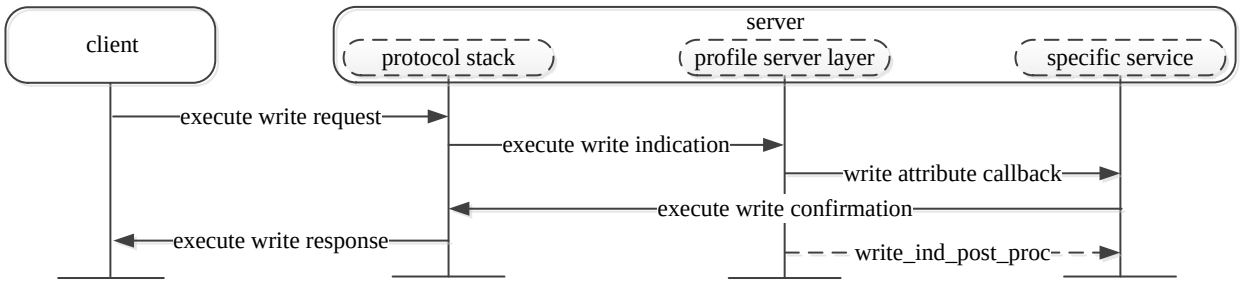

Figure 3-15 Write Long Characteristic Values– Execute Write with Result Pending

3.1.3.5 Characteristic Value Notification

This procedure is used to notify a client of a characteristic value from a server.

Server sends data by actively invoking server_send_data() function. After send data procedure is completed, it is optional to inform application by server general callback function.

The interaction between each layer is shown in Figure 3-16.

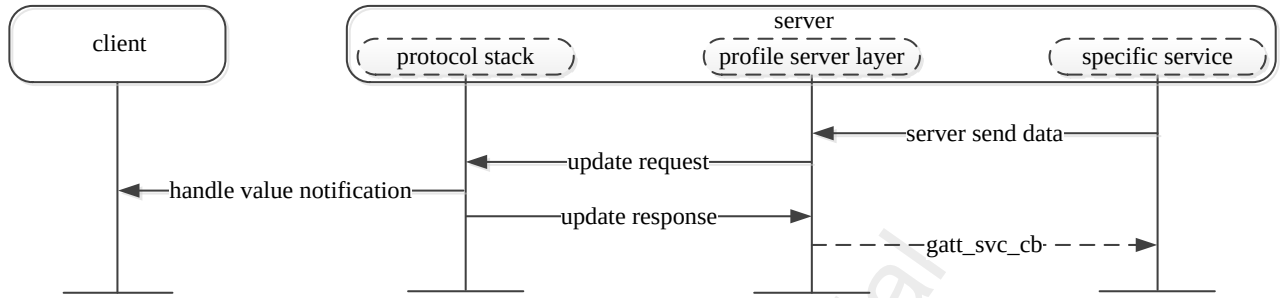

Figure 3-16 Characteristic Value Notification

```
bool simp_ble_service_send_v3_notify(uint8_t conn_id, T_SERVER_ID service_id, void *p_value,
                                     uint16_t length)
{
    APP_PRINT_INFO0("simp_ble_service_send_v3_notify");
    // send notification to client
    return server_send_data(conn_id, service_id, SIMPLE_BLE_SERVICE_CHAR_V3_NOTIFY_INDEX, p_value,
                            length, GATT_PDU_TYPE_ANY);
}
```

3.1.3.6 Characteristic Value Indication

This procedure is used to indicate client of a characteristic value from a server. Once the indication is received, the client shall respond with a confirmation. After server receives handle value confirmation, it is optional to inform application by server general callback function.

The interaction between each layer is shown in Figure 3-17.

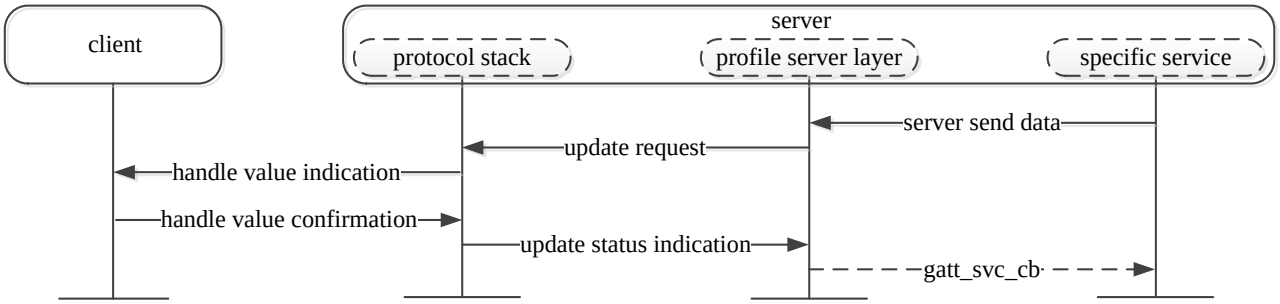

Figure 3-17 Characteristic Value Indication

```
bool simp_ble_service_send_v4_indicate(uint8_t conn_id, T_SERVER_ID service_id, void *p_value,
```

```
                                        uint16_t length)
{
    APP_PRINT_INFO0("simp_ble_service_send_v4_indicate");
    // send indication to client
    return server_send_data(conn_id, service_id, SIMPLE_BLE_SERVICE_CHAR_V4_INDICATE_INDEX,
                            p_value, length, GATT_PDU_TYPE_ANY);
}
```

app_profile_callback() will be called after handle value confirmation.

```
T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    if (service_id == SERVICE_PROFILE_GENERAL_ID)
    {
        T_SERVER_APP_CB_DATA *p_param = (T_SERVER_APP_CB_DATA *)p_data;
        switch (p_param->eventId)
        {
            ...
        case PROFILE_EVT_SEND_DATA_COMPLETE:
            APP_PRINT_INFO5("PROFILE_EVT_SEND_DATA_COMPLETE: conn_id %d, cause 0x%x,
service_id %d, attrib_idx 0x%x, credits %d",
                            p_param->event_data.send_data_result.conn_id,
                            p_param->event_data.send_data_result.cause,
                            p_param->event_data.send_data_result.service_id,
                            p_param->event_data.send_data_result.attrib_idx,
                            p_param->event_data.send_data_result.credits);
            if (p_param->event_data.send_data_result.cause == GAP_SUCCESS)
            {
                APP_PRINT_INFO0("PROFILE_EVT_SEND_DATA_COMPLETE success");
            }
            else
            {
                APP_PRINT_ERROR0("PROFILE_EVT_SEND_DATA_COMPLETE failed");
            }
            break;
        default:
            break;
        }
    }
```

3.1.4 Implementation of Specific Service

A profile is composed of one or more services which are necessary to fulfill a use case. A service is composed of

characteristics. Each characteristic contains a characteristic value and may contain optional characteristic descriptor. The service, characteristic and the components of the characteristic (i.e. value and descriptors) contain the profile data and are all stored in attributes on the server.

The guide line on how to develop a specific service is as follows:

- Define Service and Profile Spec
- Define Service Attribute Table
- Define interface between Service and APP
- Define **xxx_add_service()**, **xxx_set_parameter()**, **xxx_notify()**，**xxx_indicate()** API etc
- Implement **xxx_ble_service_cbs** with type of T_FUN_GATT_SERVICE_CBS APIs

In this chapter, simple BLE service will be tacken as an example, and a guide on how to implement a specific service will be provided.

For more details refer to source code in simple_ble_service.c and simple_ble_service.h.

3.1.4.1 Define Service and Profile Spec

In order to implement a specific service, we need to define the service and profile spec.

3.1.4.2 Define Service Table

Service that is composed of attribute elements is defined by a service table which consists of one or more services.

3.1.4.2.1 Attribute Element

Attribute element is elementary unit of service. The structure of attribute element is defined in gatt.h.

```
typedef struct {
    uint16_t    flags;              /**< Attribute flags @ref GATT_ATTRIBUTE_FLAG */
    uint8_t     type_value[2 + 14]; /**< 16 bit UUID + included value or 128 bit UUID */
    uint16_t    value_len;          /**< Length of value */
    void         *p_value_context;  /**< Pointer to value if @ref ATTRIB_FLAG_VALUE_INCL
    and @ref ATTRIB_FLAG_VALUE_APPL not set */
    uint32_t    permissions;        /**< Attribute permission @ref GATT_ATTRIBUTE_PERMISSIONS */
} T_ATTRIB_APPL;
```

1. **Flags**

   Flags option value and description are shown in Table 3-3.

Table 3-3 Flags Option Value and Description

Option Values	Description
ATTRIB_FLAG_LE	Used only for primary service declaration attributes if GATT over BLE

	is supported
ATTRIB_FLAG_VOID	Attribute value is neither supplied by application nor included following 16bit UUID. Attribute value is pointed by p_value_context and value_len shall be set to the length of attribute value.
ATTRIB_FLAG_VALUE_INCL	Attribute value is included following 16 bit UUID
ATTRIB_FLAG_VALUE_APPL	Application has to supply attribute value
ATTRIB_FLAG_UUID_128BIT	Attribute uses 128 bit UUID
ATTRIB_FLAG_ASCII_Z	Attribute value is ASCII_Z string
ATTRIB_FLAG_CCCD_APPL	Application will be informed if CCCD value is changed
ATTRIB_FLAG_CCCD_NO_FILTER	Application will be informed about CCCD value when CCCD is written by client, no matter it is changed or not

Note:

ATTRIB_FLAG_LE can only be used by attribute whose type is primary service declaration, to indicate that primary service allows LE link access.

Attribute element must pick one value among **ATTRIB_FLAG_VOID**, **ATTRIB_FLAG_VALUE_INCL** and **ATTRIB_FLAG_VALUE_APPL**.

ATTRIB_FLAG_VALUE_INCL flag means attribute value will be put into the last fourteen bytes of type_value (the first two bytes of type_value is used to save UUID), and value_len is the number of the bytes put into the region of the last fourteen bytes. As attribute value has been provided in type_value, p_value_context pointer is assigned with NULL.

ATTRIB_FLAG_VALUE_APPL flag means attribute value is supplied by application. As long as stack is involved in attribute value related operation, it will interact with application to fulfil the corresponding operation process. As attribute value is provided by application, only UUID of attribute is required to be put into type_value, while value_len is 0 and p_value_context pointer is assigned with NULL.

ATTRIB_FLAG_VOID flag means attribute value is neither supplied in the last 14 bytes of type_value nor application. Only UUID of attribute is required in type_value, p_value_context pointer points to attribute value and value_len indicates the length of the attribute value.

Table 3-4 shows the flags value and actual value used by read attribute process.

Table 3-4 Flags Value Select Mode

		APPL	APPL|ASCII_Z	INCL	INCL|ASCII_Z	VOID	VOID|ASCII_Z
If set	**value_len**	Any(NULL)	Any(NULL)	Strlen(value)	Strlen(value)	Strlen(value)	Strlen(value)
	type_value+2	Any(NULL)	Any(NULL)	value	value	Any(NULL)	Any(NULL)
	p_value_context	Any(NULL)	Any(NULL)	Any(NULL)	Any(NULL)	value	value
Actual get by	**Actual length**	Reply by application	Reply by application	Strlen(value)	Strlen(value)+1	Strlen(value)	Strlen(value)+1

read attribute process	**Actual value**	Reply by application	Reply by application	value	Value + '\0'	Value	Value + '\0'

APPL: ATTRIB_FLAG_VALUE_APPL

VOID: ATTRIB_FLAG_VOID

INCL: ATTRIB_FLAG_VALUE_INCL

ASCII_Z: ATTRIB_FLAG_ASCII_Z

2. Permissions

The permissions associated with the attribute specify the security level required for read and / or write access, as well as notification and / or indication. Value of permissions is used to indicate permission of the attribute. Attribute permissions are a combination of access permissions, encryption permissions, authentication permissions and authorization permissions, and its acceptable values are given in Table 3-5.

Table 3-5 Value of Permissions

Types	Permissions
Read Permissions	GATT_PERM_READ
	GATT_PERM_READ_AUTHEN_REQ
	GATT_PERM_READ_AUTHEN_MITM_REQ
	GATT_PERM_READ_AUTHOR_REQ
	GATT_PERM_READ_ENCRYPTED_REQ
	GATT_PERM_READ_AUTHEN_SC_REQ
Write Permissions	GATT_PERM_WRITE
	GATT_PERM_WRITE_AUTHEN_REQ
	GATT_PERM_WRITE_AUTHEN_MITM_REQ
	GATT_PERM_WRITE_AUTHOR_REQ
	GATT_PERM_WRITE_ENCRYPTED_REQ
	GATT_PERM_WRITE_AUTHEN_SC_REQ
Notify/Indicate Permissions	GATT_PERM_NOTIF_IND
	GATT_PERM_NOTIF_IND_AUTHEN_REQ
	GATT_PERM_NOTIF_IND_AUTHEN_MITM_REQ
	GATT_PERM_NOTIF_IND_AUTHOR_REQ
	GATT_PERM_NOTIF_IND_ENCRYPTED_REQ
	GATT_PERM_NOTIF_IND_AUTHEN_SC_REQ

3.1.4.2.2 Service Table

Service contains a group of attributes that are called service table. A service table contains various types of attributes, such as service declaration, characteristic declaration, characteristic value and characteristic descriptor

declaration.

An example of service table is given in Table 3-6, and it is implemented in simple_ble_service.c of ble_peripheral sample project.

Table 3-6 Service Table Example

Flags	Attribute Type	Attribute Value	Permission
INCL | LE	<<primary service declaration>>	<<simple profile UUID – 0xA00A>>	read
INCL	<<characteristic declaration>>	Property(read)	read
APPL	<<characteristic value>>	UUID(0xB001),Value not defined here	read
VOID | ASCII_Z	<<Characteristic User Description>>	UUID(0x2901) Value defined in p_value_context	read
INCL	<<characteristic declaration>>	Property(write | write without response)	read
APPL	<<characteristic value>>	UUID(0xB002), Value not defined here	write
INCL	<<characteristic declaration>>	Property(notify)	read
APPL	<<characteristic value>>	UUID(0xB003), Value not defined here	none
CCCD_APPL	<<client characteristic configuration descriptor>>	Default CCCD value	read | write
INCL	<<characteristic declaration>>	Property(indicate)	read
APPL	<<characteristic value>>	UUID(0xB004), Value not defined here	none
CCCD_APPL	<<client characteristic configuration descriptor>>	Default CCCD value	read | write

Note:

The elements in quotation mark are UUID value, which are either defined in core spec, or customized.

LE is abbreviation of ATTRIB_FLAG_LE

INCL is abbreviation of ATTRIB_FLAG_VALUE_INCL

APPL is abbreviation of ATTRIB_FLAG_VALUE_APPL

The sample code for service table is as follows:

```
const T_ATTRIB_APPL simple_ble_service_tbl[] =
{
    /* <<Primary Service>>, .. */
    {
        (ATTRIB_FLAG_VALUE_INCL | ATTRIB_FLAG_LE),   /* flags        */
        {                                            /* type_value */
            LO_WORD(GATT_UUID_PRIMARY_SERVICE),
            HI_WORD(GATT_UUID_PRIMARY_SERVICE),
```

```
            LO_WORD(GATT_UUID_SIMPLE_PROFILE),          /* service UUID */
            HI_WORD(GATT_UUID_SIMPLE_PROFILE)
        },
        UUID_16BIT_SIZE,                                /* bValueLen        */
        NULL,                                           /* p_value_context */
        GATT_PERM_READ                                  /* permissions    */
    },
    /* <<Characteristic>> demo for read */
    {
        ATTRIB_FLAG_VALUE_INCL,                         /* flags */
        {                                               /* type_value */
            LO_WORD(GATT_UUID_CHARACTERISTIC),
            HI_WORD(GATT_UUID_CHARACTERISTIC),
            GATT_CHAR_PROP_READ                         /* characteristic properties */
            /* characteristic UUID not needed here, is UUID of next attrib. */
        },
        1,                                              /* bValueLen */
        NULL,
        GATT_PERM_READ                                  /* permissions */
    },
    {
        ATTRIB_FLAG_VALUE_APPL,                         /* flags */
        {                                               /* type_value */
            LO_WORD(GATT_UUID_CHAR_SIMPLE_V1_READ),
            HI_WORD(GATT_UUID_CHAR_SIMPLE_V1_READ)
        },
        0,                                              /* bValueLen */
        NULL,
        GATT_PERM_READ                                  /* permissions */
    },
    {
        ATTRIB_FLAG_VOID | ATTRIB_FLAG_ASCII_Z,         /* flags */
        {                                               /* type_value */
            LO_WORD(GATT_UUID_CHAR_USER_DESCR),
            HI_WORD(GATT_UUID_CHAR_USER_DESCR),
        },
        (sizeof(v1_user_descr) - 1),                                        /* bValueLen */
        (void *)v1_user_descr,
        GATT_PERM_READ                  /* permissions */
    },
    /* <<Characteristic>> demo for write */
    {
        ATTRIB_FLAG_VALUE_INCL,                         /* flags */
        {                                               /* type_value */
```

```
            LO_WORD(GATT_UUID_CHARACTERISTIC),
            HI_WORD(GATT_UUID_CHARACTERISTIC),
            (GATT_CHAR_PROP_WRITE | GATT_CHAR_PROP_WRITE_NO_RSP) /* characteristic properties
*/
            /* characteristic UUID not needed here, is UUID of next attrib. */
        },
        1,                                          /* bValueLen */
        NULL,
        GATT_PERM_READ                              /* permissions */
    },
    {
        ATTRIB_FLAG_VALUE_APPL,                     /* flags */
        {                                           /* type_value */
            LO_WORD(GATT_UUID_CHAR_SIMPLE_V2_WRITE),
            HI_WORD(GATT_UUID_CHAR_SIMPLE_V2_WRITE)
        },
        0,                                          /* bValueLen */
        NULL,
        GATT_PERM_WRITE                             /* permissions */
    },
    /* <<Characteristic>>, demo for notify */
    {
        ATTRIB_FLAG_VALUE_INCL,                     /* flags */
        {                                           /* type_value */
            LO_WORD(GATT_UUID_CHARACTERISTIC),
            HI_WORD(GATT_UUID_CHARACTERISTIC),
            (GATT_CHAR_PROP_NOTIFY)                 /* characteristic properties */
            /* characteristic UUID not needed here, is UUID of next attrib. */
        },
        1,                                          /* bValueLen */
        NULL,
        GATT_PERM_READ                              /* permissions */
    },
    {
        ATTRIB_FLAG_VALUE_APPL,                     /* flags */
        {                                           /* type_value */
            LO_WORD(GATT_UUID_CHAR_SIMPLE_V3_NOTIFY),
            HI_WORD(GATT_UUID_CHAR_SIMPLE_V3_NOTIFY)
        },
        0,                                          /* bValueLen */
        NULL,
        GATT_PERM_NONE                              /* permissions */
    },
    /* client characteristic configuration */
```

```
    {
        ATTRIB_FLAG_VALUE_INCL | ATTRIB_FLAG_CCCD_APPL,                 /* flags */
        {                                                               /* type_value */
            LO_WORD(GATT_UUID_CHAR_CLIENT_CONFIG),
            HI_WORD(GATT_UUID_CHAR_CLIENT_CONFIG),
            /* NOTE: this value has an instantiation for each client, a write to */
            /* this attribute does not modify this default value:                 */
            LO_WORD(GATT_CLIENT_CHAR_CONFIG_DEFAULT), /* client char. config. bit field */
            HI_WORD(GATT_CLIENT_CHAR_CONFIG_DEFAULT)
        },
        2,                                                              /* bValueLen */
        NULL,
        (GATT_PERM_READ | GATT_PERM_WRITE)                              /* permissions */
    },
    /* <<Characteristic>> demo for indicate */
    {
        ATTRIB_FLAG_VALUE_INCL,                                         /* flags */
        {                                                               /* type_value */
            LO_WORD(GATT_UUID_CHARACTERISTIC),
            HI_WORD(GATT_UUID_CHARACTERISTIC),
            (GATT_CHAR_PROP_INDICATE)                                   /* characteristic properties */
            /* characteristic UUID not needed here, is UUID of next attrib. */
        },
        1,                                                              /* bValueLen */
        NULL,
        GATT_PERM_READ                                                  /* permissions */
    },
    {
        ATTRIB_FLAG_VALUE_APPL,                                         /* flags */
        {                                                               /* type_value */
            LO_WORD(GATT_UUID_CHAR_SIMPLE_V4_INDICATE),
            HI_WORD(GATT_UUID_CHAR_SIMPLE_V4_INDICATE)
        },
        0,                                                              /* bValueLen */
        NULL,
        GATT_PERM_NONE                                                  /* permissions */
    },
    /* client characteristic configuration */
    {
        ATTRIB_FLAG_VALUE_INCL | ATTRIB_FLAG_CCCD_APPL,                 /* flags */
        {                                                               /* type_value */
            LO_WORD(GATT_UUID_CHAR_CLIENT_CONFIG),
            HI_WORD(GATT_UUID_CHAR_CLIENT_CONFIG),
            /* NOTE: this value has an instantiation for each client, a write to */
```

```
            /* this attribute does not modify this default value:                    */
            LO_WORD(GATT_CLIENT_CHAR_CONFIG_DEFAULT), /* client char. config. bit field */
            HI_WORD(GATT_CLIENT_CHAR_CONFIG_DEFAULT)
        },
        2,                                                  /* bValueLen */
        NULL,
        (GATT_PERM_READ | GATT_PERM_WRITE)                  /* permissions */
    },
};
```

3.1.4.3 Define interface between Service and App

When a service attribute value was read or written, the notification will be passed to application by callback registered by application.

Taking simple Bluetooth LE service as an example, we define a data with type TSIMP_CALLBACK_DATA to hold notification result.

```
typedef struct {
    uint8_t                     conn_id;
    T_SERVICE_CALLBACK_TYPE msg_type;
    TSIMP_UPSTREAM_MSG_DATA msg_data;
} TSIMP_CALLBACK_DATA;
```

msg_type indicates it is a read, write or CCCD update operation.

```
typedef enum {
    SERVICE_CALLBACK_TYPE_INDIFICATION_NOTIFICATION = 1,
    SERVICE_CALLBACK_TYPE_READ_CHAR_VALUE = 2,
    SERVICE_CALLBACK_TYPE_WRITE_CHAR_VALUE = 3,
} T_SERVICE_CALLBACK_TYPE;
```

msg_data holds the data of read, write or CCCD update operation.

3.1.4.4 Define xxx_add_service(), xxx_set_parameter(), xxx_notify(), xxx_indicate() API etc.

xxx_add_service() is used to add service table to profile server layer, and register a callback for service attribute read, write or CCCD update.

xxx_set_parameter() is used to set service related data by application.

xxx_notify() is used to send notification data.

xxx_indicate() is used to send indication data.

3.1.4.5 Implement xxx_ble_service_cbs with type of T_FUN_GATT_SERVICE_CBS APIs

xxx_ble_service_cbs is used to handle read, write or CCCD update operation from remote profile client.

```
const T_FUN_GATT_SERVICE_CBS simp_ble_service_cbs = {
    simp_ble_service_attr_read_cb,    // Read callback function pointer
    simp_ble_service_attr_write_cb, // Write callback function pointer
    simp_ble_service_cccd_update_cb // CCCD update callback function pointer
};
```

Callback is registered by server_add_service() which is called in xxx_ble_service_add_service().

```
T_SERVER_ID simp_ble_service_add_service(void *p_func)
{
    if (false == server_add_service(&simp_service_id,
                                    (uint8_t *)simple_ble_service_tbl,
                                    sizeof(simple_ble_service_tbl),
                                    simp_ble_service_cbs))
    {
        APP_PRINT_ERROR0("simp_ble_service_add_service: fail");
        simp_service_id = 0xff;
        return simp_service_id;
    }
    pfn_simp_ble_service_cb = (P_FUN_SERVER_GENERAL_CB)p_func;
    return simp_service_id;
}
```

3.2 Bluetooth LE Profile Client

3.2.1 Overview

Client interface of profile offers developers the functions to discovery services at GATT Server, receive and handle indications and notifications from GATT Server, and send read/write request to GATT Server.

Figure 3-18 shows the profile client hierarchy.

Content of profile involves profile client layer and specific profile client. Profile client layer above protocol stack encapsulates interfaces for specific client to access protocol stack. Thus, development of specific clients does not involve details of protocol stack process and becomes simpler and clearer. Specific client which is based on the profile client layer is implemented by application layer.

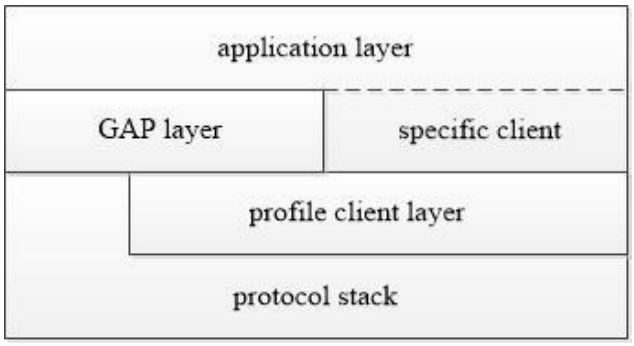

Figure 3-18 Profile Client Hierarchy

Implementation of specific profile client is quite different from that of profile server. Profile client does not involve attribute table, and provides functions to collect and acquire information instead of providing service and information.

3.2.2 Supported Clients

Supported clients are listed in Table 3-7.

Table 3-7 Supported Clients

Terms	Definitions	Files
GAP Client	Attribute Service Client	gaps_client.c gaps_client.h
BAS Client	Battery Service Client	bas_client.c bas_client.h
ANCS Client	Apple Notification Center Service Client	ancs_client.c ancs_client.h
SIMP Client	Simple BLE Service Client	simple_ble_client.c simple_ble_client.h
IPSS Client	Internet Protocol Support Service Client	ipss_client.c ipss_client.h

3.2.3 Profile Client Layer

Profile client layer handles interaction with protocol stack layer and provides interfaces to design specific client. Client will discover services and characteristics of server, read and write attribute, receive and handle notifications and indications from server.

3.2.3.1 Client General Callback

Client general callback function is used to send client_all_primary_srv_discovery() result to application when client_id is CLIENT_PROFILE_GENERAL_ID. This callback function can be initialized with client_register_general_client_cb() function.

```
void app_le_profile_init(void)
{
    client_init(3);
    ......
    client_register_general_client_cb(app_client_callback);
}

static T_USER_CMD_PARSE_RESULT cmd_srvdis(T_USER_CMD_PARSED_VALUE *p_parse_value)
{
    uint8_t conn_id = p_parse_value->dw_param[0];
    T_GAP_CAUSE cause;
    cause = client_all_primary_srv_discovery(conn_id, CLIENT_PROFILE_GENERAL_ID);
    return (T_USER_CMD_PARSE_RESULT)cause;
}
T_APP_RESULT app_client_callback(T_CLIENT_ID client_id, uint8_t conn_id, void *p_data)
{
    T_APP_RESULT    result = APP_RESULT_SUCCESS;
    APP_PRINT_INFO2("app_client_callback: client_id %d, conn_id %d",
                    client_id, conn_id);
    if (client_id == CLIENT_PROFILE_GENERAL_ID)
    {
        T_CLIENT_APP_CB_DATA *p_client_app_cb_data = (T_CLIENT_APP_CB_DATA *)p_data;
        switch (p_client_app_cb_data->cb_type)
        {
            case CLIENT_APP_CB_TYPE_DISC_STATE:
            ......
        }
}
```

If APP does not use client_all_primary_srv_discovery() with CLIENT_PROFILE_GENERAL_ID, APP does not need to register this general callback.

3.2.3.2 Specific Client Callback

3.2.3.2.1 Add Client

Profile client layer maintains information of all added specific clients. The total number of all client tables to be

added shall be initialized by invoking client_init() supplied by profile client layer.

Profile client layer provides client_register_spec_client_cb() interface to register specific client callbacks. Figure 3-19 shows that client layer contains serval specific client tables.

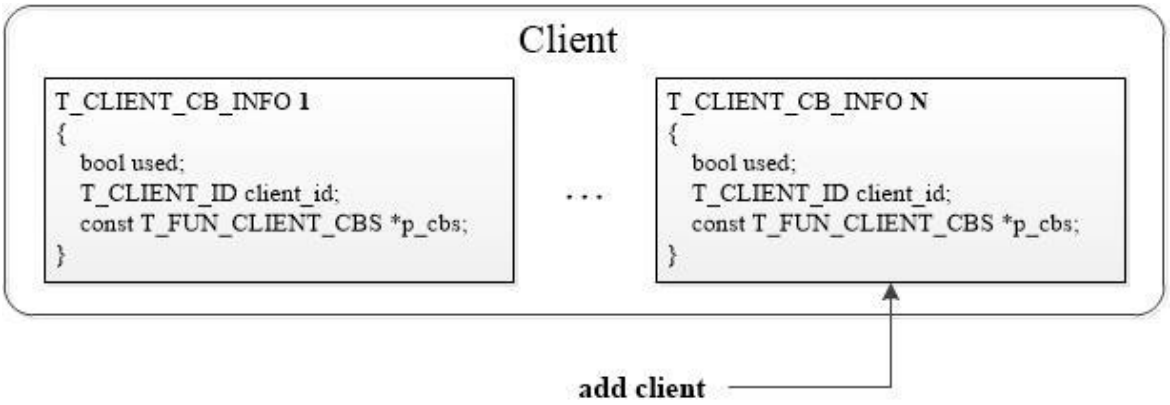

Figure 3-19 Add Specific Clients to Profile Client Layer

APP adds specific client to Profile Client Layer, and APP will record the returned client id to each added specific client for implementation of subsequent data interaction process.

3.2.3.2.2 Callbacks

Specific client's callback functions shall be implemented in specific client module. The specific client callback structure is defined in profile_client.h.

```
typedef struct {
    P_FUN_DISCOVER_STATE_CB       discover_state_cb;     //!< Discovery state callback function pointer
    P_FUN_DISCOVER_RESULT_CB      discover_result_cb;   //!< Discovery reault callback function pointer
    P_FUN_READ_RESULT_CB          read_result_cb;       //!< Read response callback function pointer
    P_FUN_WRITE_RESULT_CB         write_result_cb;      //!< Write result callback function pointer
    P_FUN_NOTIFY_IND_RESULT_CB notify_ind_result_cb;//!< Notify Indication callback function pointer
    P_FUN_DISCONNECT_CB           disconnect_cb;        //!< Disconnection callback function pointer
} T_FUN_CLIENT_CBS;
```

discover_state_cb: Discovery state callback, which is used to inform specific client module the discovery state of client_xxx_discovery.

discover_result_cb: Discovery result callback, which is used to inform specific client module the discovery result of client_xxx_discovery.

read_result_cb: Read result callback, which is used to inform specific client module the read result of client_attr_read() or client_attr_read_using_uuid().

write_result_cb: Write result callback, which is used to inform specific client module the write result of client_attr_write().

notify_ind_result_cb: Notification and indication callback, which is used to inform specific client module that notification or indication data is received from server.

disconnect_cb: Disconnection callback, which is used to inform specific client module that the one LE link is disconnected.

3.2.3.3 Discovery Procedure

After establishing connection to the server, the client generally performs a discovery process if local device does not store the handle information of server. Specific client needs to call client_xxx_discovery() to start discovery procedure. Then specific client needs to handle discovery state in discover_state_cb() callback and discovery result in callback discover_result_cb().

The interaction between each layer is shown in Figure 3-20.

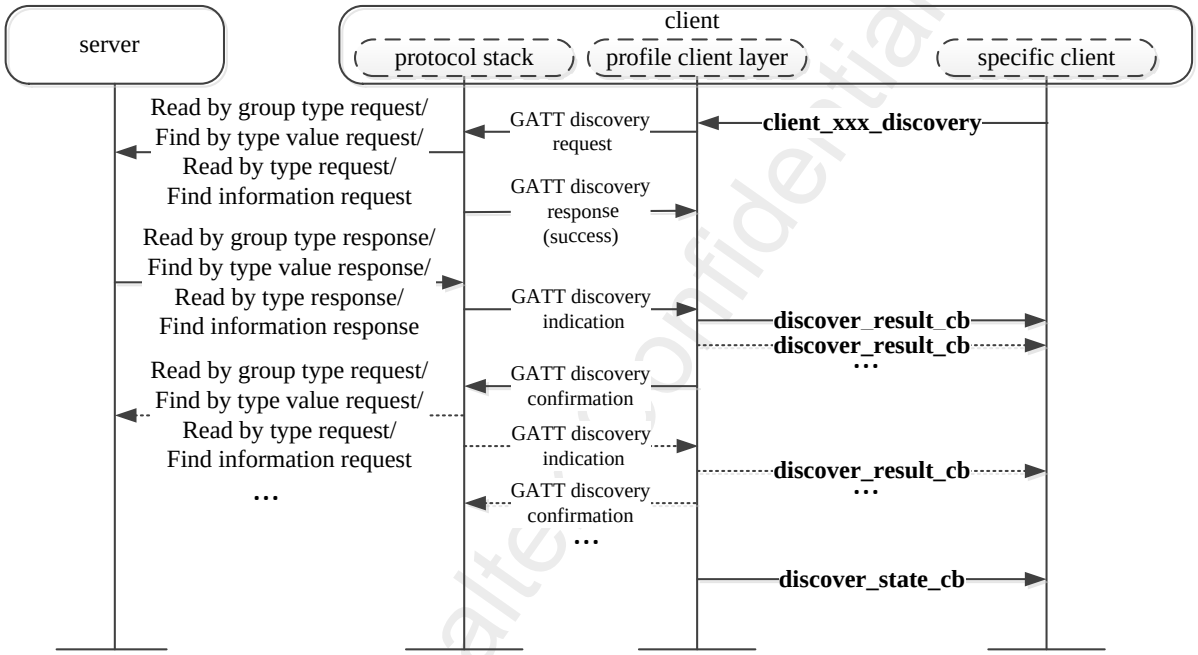

Figure 3-20 GATT Discovery Procedure

3.2.3.3.1 Discovery State

Table 3-8 Discovery State

Reference API	T_DISCOVERY_STATE
client_all_primary_srv_discovery()	DISC_STATE_SRV_DONE, DISC_STATE_FAILED
client_by_uuid_srv_discovery()	DISC_STATE_SRV_DONE DISC_STATE_FAILED
client_by_uuid128_srv_discovery()	DISC_STATE_SRV_DONE DISC_STATE_FAILED

client_all_char_discovery()	DISC_STATE_CHAR_DONE DISC_STATE_FAILED
client_all_char_descriptor_discovery()	DISC_STATE_CHAR_DESCRIPTOR_DONE DISC_STATE_FAILED
client_relationship_discovery()	DISC_STATE_RELATION_DONE DISC_STATE_FAILED
client_by_uuid_char_discovery()	DISC_STATE_CHAR_UUID16_DONE DISC_STATE_FAILED
client_by_uuid128_char_discovery()	DISC_STATE_CHAR_UUID128_DONE DISC_STATE_FAILED

3.2.3.3.2 Discovery Result

Table 3-9 Discovery Result

Reference API	T_DISCOVERY_RESULT_TYPE	T_DISCOVERY_RESULT_DATA
client_all_primary_srv_discovery()	DISC_RESULT_ALL_SRV_UUID16	T_GATT_SERVICE_ELEM16 *p_srv_uuid16_disc_data;
client_all_primary_srv_discovery()	DISC_RESULT_ALL_SRV_UUID128	T_GATT_SERVICE_ELEM128 *p_srv_uuid128_disc_data;
client_by_uuid_srv_discovery(), client_by_uuid128_srv_discovery()	DISC_RESULT_SRV_DATA	T_GATT_SERVICE_BY_UUID_ELEM *p_srv_disc_data;
client_all_char_discovery()	DISC_RESULT_CHAR_UUID16	T_GATT_CHARACT_ELEM16 *p_char_uuid16_disc_data;
client_all_char_discovery()	DISC_RESULT_CHAR_UUID128	T_GATT_CHARACT_ELEM128 *p_char_uuid128_disc_data;
client_all_char_descriptor_discovery()	DISC_RESULT_CHAR_DESC_UUID16	T_GATT_CHARACT_DESC_ELEM16 *p_char_desc_uuid16_disc_data;
client_all_char_descriptor_discovery()	DISC_RESULT_CHAR_DESC_UUID128	T_GATT_CHARACT_DESC_ELEM128 *p_char_desc_uuid128_disc_data;
client_relationship_discovery()	DISC_RESULT_RELATION_UUID16	T_GATT_RELATION_ELEM16 *p_relation_uuid16_disc_data;
client_relationship_discovery()	DISC_RESULT_RELATION_UUID128	T_GATT_RELATION_ELEM128 *p_relation_uuid128_disc_data;
client_by_uuid_char_discovery()	DISC_RESULT_BY_UUID16_CHAR	T_GATT_CHARACT_ELEM16 *p_char_uuid16_disc_data;
client_by_uuid_char_discovery()	DISC_RESULT_BY_UUID128_CHAR	T_GATT_CHARACT_ELEM128

```
*p_char_uuid128_disc_data;
```

3.2.3.4 Characteristic Value Read

This procedure is used to read a characteristic value of a server. There are two sub-procedures in profile client layer that can be used to read a Characteristic value: Read Characteristic Value by Handle and Read Characteristic Value by UUID.

3.2.3.4.1 Read Characteristic Value by Handle

This sub-procedure is used to read a Characteristic Value from a server when the client knows the Characteristic Value Handle. Reading characteristic value by handle is a three-phase process. Phase 1 and phase 3 are always used. The phase 2 is an optional phase (see Figure 3-21):

- Phase 1: Call client_attr_read() to read Characteristic Value.
- Phase 2: Optional phase. If the Characteristic Value is greater than (ATT_MTU - 1) octets in length, the Read Response only contains the first portion of the Characteristic Value and the Read Long Characteristic Value procedure will be used.
- Phase 3: Profile client layer calls read_result_cb() to return read result.

The interaction between each layer is shown in Figure 3-21.

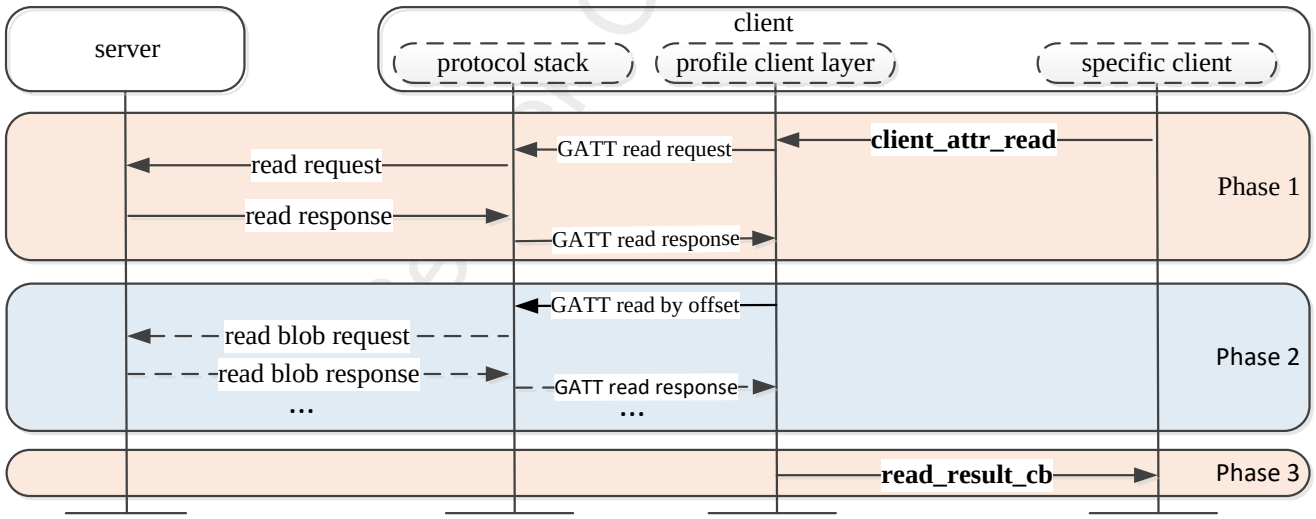

Figure 3-21 Read Characteristic Value by Handle

3.2.3.4.2 Read Characteristic Value by UUID

This sub-procedure is used to read a Characteristic Value from a server when the client only knows the

characteristic UUID and does not know the handle of the characteristic. Reading characteristic value by UUID is a three-phase process. Phase 1 and phase 3 are always used. Phase 2 is optional (see Figure 3-22):

- Phase 1: Call client_attr_read_using_uuid() to read Characteristic Value.
- Phase 2: Optional phase. If the Characteristic Value is greater than (ATT_MTU - 4) octets in length, the Read by Type Response only contains the first portion of the Characteristic Value and the Read Long Characteristic Value procedure will be used.
- Phase 3: Profile client layer calls read_result_cb() to return read result.

The interaction between each layer is shown in Figure 3-22.

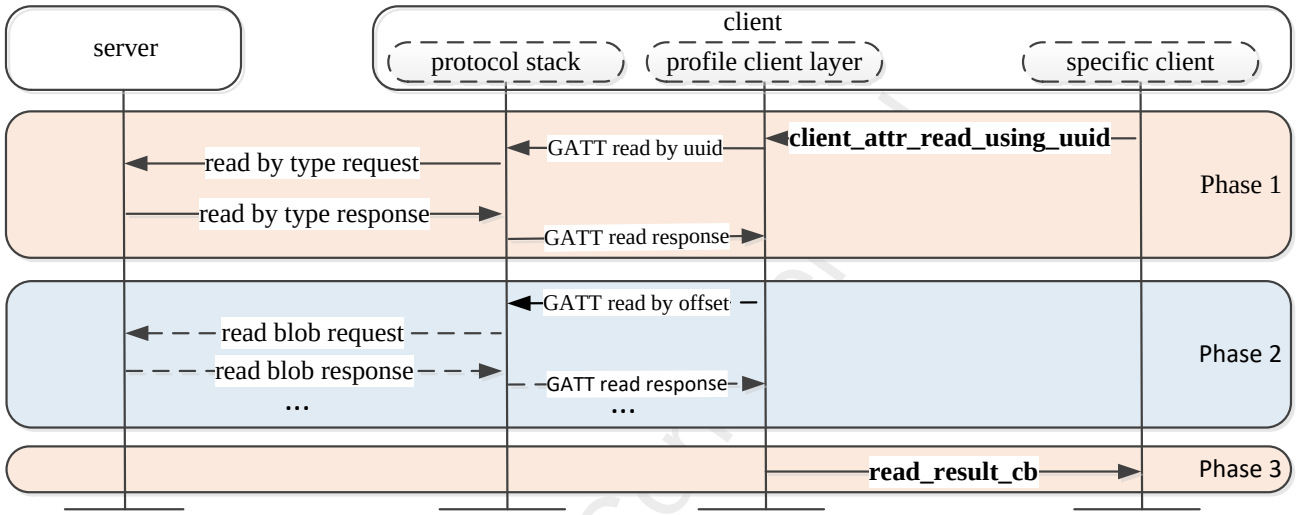

Figure 3-22 Read Characteristic Value by UUID

3.2.3.5 Characteristic Value Write

This procedure is used to write a Characteristic Value to a server. There are four sub-procedures in profile client layer that can be used to write a Characteristic Value: Write without Response, Signed Write without Response, Write Characteristic Value and Write Long Characteristic Values.

3.2.3.5.1 Write Characteristic Value

This sub-procedure is used to write a Characteristic Value to a server when the client knows the Characteristic Value Handle. When the length of value is less than or equal to (ATT_MTU - 3) octets, the procedure will be used. Otherwise, the Write Long Characteristic Values sub-procedure will be used instead.

The interaction between each layer is shown in Figure 3-23.

Value length <= mtu_size -3

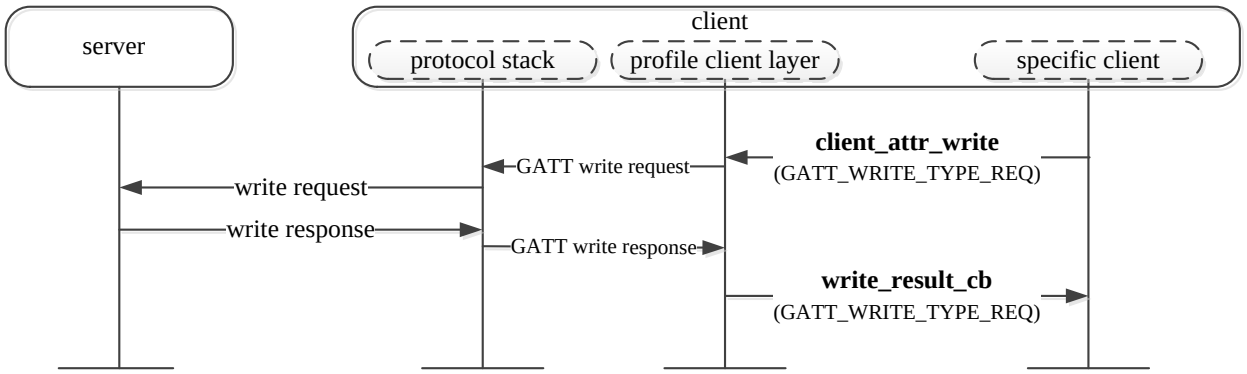

Figure 3-23 Write Characteristic Value

3.2.3.5.2 Write Long Characteristic Values

This sub-procedure is used to write a Characteristic Value to a server when the client knows the Characteristic Value Handle and the length of value is greater than (ATT_MTU - 3) octets.

The interaction between each layer is shown in Figure 3-24.

Value length > mtu_size -3
Value length <= 512

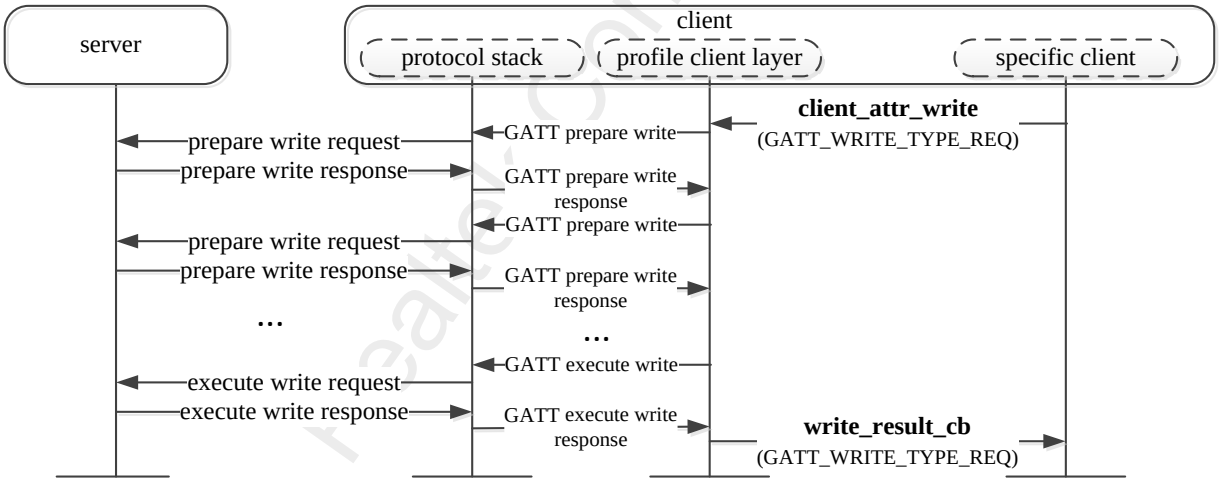

Figure 3-24 Write Long Characteristic Value

3.2.3.5.3 Write Without Response

This sub-procedure is used to write a Characteristic Value to a server when the client knows the Characteristic Value Handle and the client does not need an acknowledgment that the write operation was successfully performed. The length of value is less than or equal to (ATT_MTU - 3) octets.

The interaction between each layer is shown in Figure 3-25.

Value length <= mtu_size -3

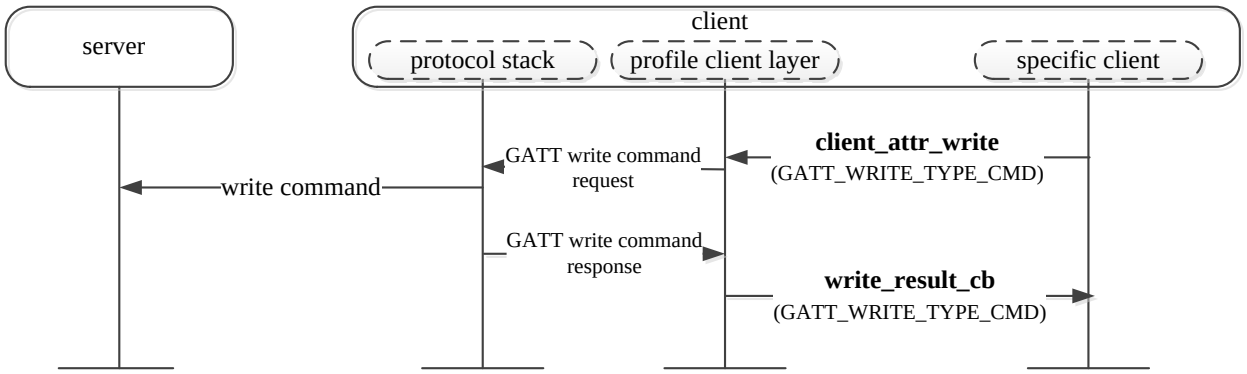

Figure 3-25 Write without Response

3.2.3.5.4 Signed Write without Response

This sub-procedure is used to write a Characteristic Value to a server when the client knows the Characteristic Value Handle and the ATT Bearer is not encrypted. This sub-procedure shall only be used if the Characteristic Properties authenticated bit is enabled and the client and server device are bonded. The length of value is less than or equal to (ATT_MTU - 15) octets.

The interaction between each layer is shown in Figure 3-26.

Value length <= mtu_size - 15

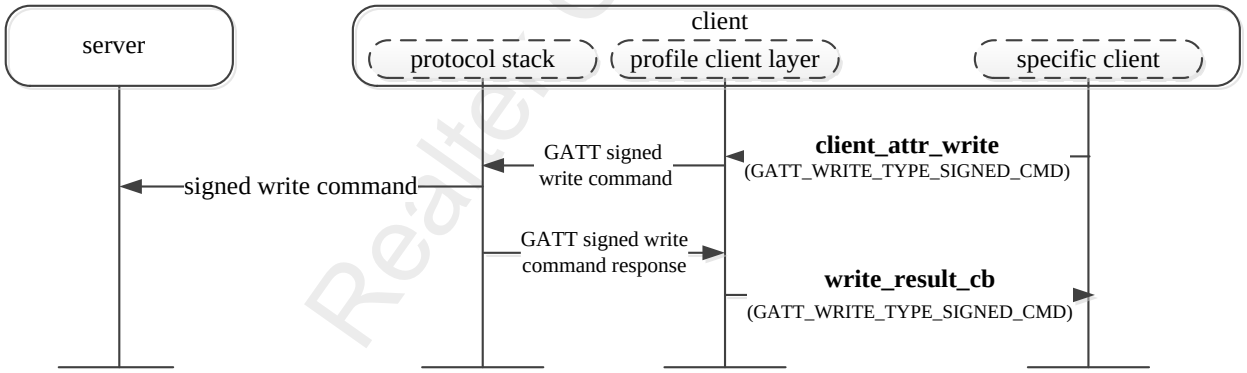

Figure 3-26 Signed Write without Response

3.2.3.6 Characteristic Value Notification

This procedure is used when a server is configured to notify a Characteristic Value to a client without expecting any Attribute Protocol layer acknowledgment that the notification was successfully received.

The interaction between each layer is shown in Figure 3-27.

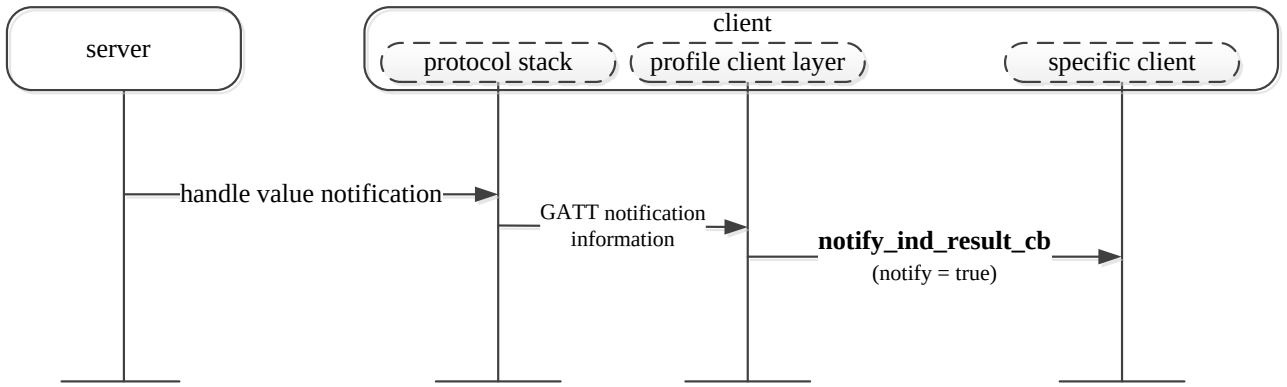

Figure 3-27 Characteristic Value Notification

Because profile client layer does not store the service handle information, profile client layer is not sure which specific client is sent to. So profile client layer will call all registered specific clients. The specific client needs to check whether the notification is sent to itself.

Sample code is shown as below:

```
static T_APP_RESULT bas_client_notify_ind_cb(uint8_t conn_id, bool notify, uint16_t handle,
        uint16_t value_size, uint8_t *p_value)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    T_BAS_CLIENT_CB_DATA cb_data;
    uint16_t *hdl_cache;
    hdl_cache = bas_table[conn_id].hdl_cache;
    cb_data.cb_type = BAS_CLIENT_CB_TYPE_NOTIF_IND_RESULT;

    if (handle == hdl_cache[HDL_BAS_BATTERY_LEVEL])
    {
        cb_data.cb_content.notify_data.battery_level = *p_value;
    }
    else
    {
        return APP_RESULT_SUCCESS;
    }
    if (bas_client_cb)
    {
        app_result = (*bas_client_cb)(bas_client, conn_id, &cb_data);
    }
    return app_result;
}
```

3.2.3.7 Characteristic Value Indication

This procedure is used when a server is configured to indicate a Characteristic Value to a client and expects an

Attribute Protocol layer acknowledgment that the indication was successfully received.

1. Characteristic Value Indication Without Result Pending

Callback function notify_ind_result_cb() return result is not APP_RESULT_PENDING.

The interaction between each layer is shown in Figure 3-28.

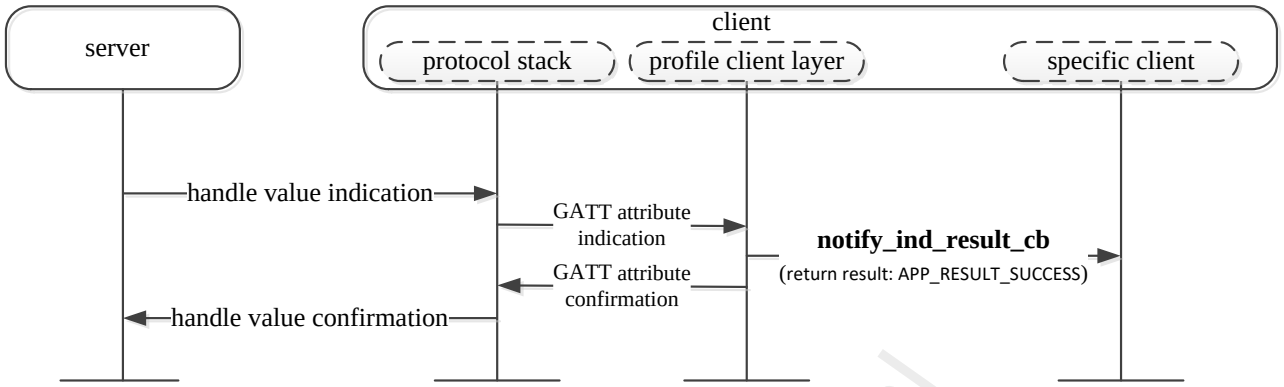

Figure 3-28 Characteristic Value Indication without Result Pending

2. Characteristic Value Indication With Result Pending

Callback function notify_ind_result_cb() return result is APP_RESULT_PENDING. APP needs to call client_attr_ind_confirm() to send confirmation.

The interaction between each layer is shown in Figure 3-29.

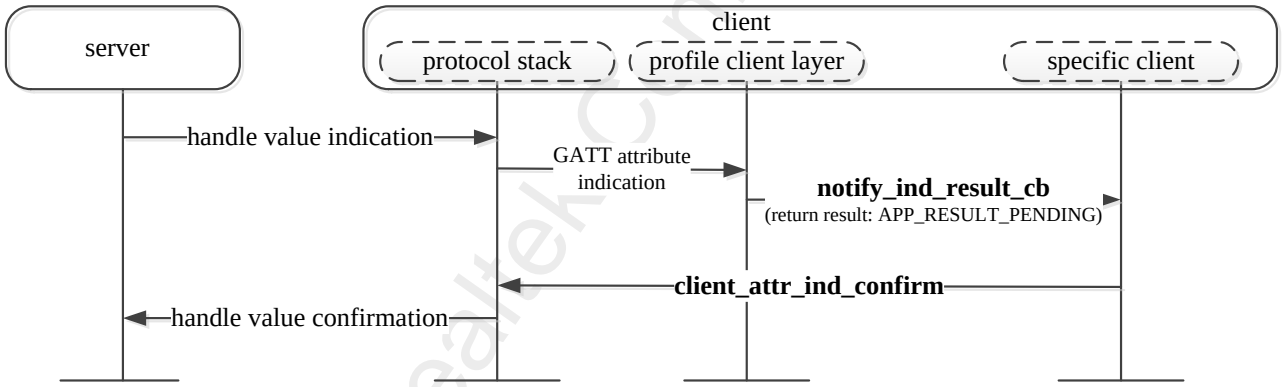

Figure 3-29 Characteristic Value Indication with Result Pending

Profile client layer does not store the service handle information, so profile client layer is not sure to which specific client should the notification be sent. So profile client layer will call all registered specific clients's callback function. The specific client needs to check whether the indication is sent to itself.

Sample code is shown as below:

```
static T_APP_RESULT simp_ble_client_notif_ind_result_cb(uint8_t conn_id, bool notify,
            uint16_t handle, uint16_t value_size, uint8_t *p_value)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    T_SIMP_CLIENT_CB_DATA cb_data;
    uint16_t *hdl_cache;
```

```
    hdl_cache = simp_table[conn_id].hdl_cache;
    cb_data.cb_type = SIMP_CLIENT_CB_TYPE_NOTIF_IND_RESULT;
    if (handle == hdl_cache[HDL_SIMBLE_V3_NOTIFY])
    {
        cb_data.cb_content.notif_ind_data.type = SIMP_V3_NOTIFY;
        cb_data.cb_content.notif_ind_data.data.value_size = value_size;
        cb_data.cb_content.notif_ind_data.data.p_value = p_value;
    }
    else if (handle == hdl_cache[HDL_SIMBLE_V4_INDICATE])
    {
        cb_data.cb_content.notif_ind_data.type = SIMP_V4_INDICATE;
        cb_data.cb_content.notif_ind_data.data.value_size = value_size;
        cb_data.cb_content.notif_ind_data.data.p_value = p_value;
    }
    else
    {
        return app_result;
    }
    /* Inform application the notif/ind result. */
    if (simp_client_cb)
    {
        app_result = (*simp_client_cb)(simp_client, conn_id, &cb_data);
    }
    return app_result;
}
```

3.2.3.8 Sequential Protocol

3.2.3.8.1 Request-response protocol

Many attribute protocol PDUs use a sequential request-response protocol. Once a client sends a request to a server, that client shall send no other request to the same server until a response PDU has been received.

The following procedures are sequential request-response protocol.

- Discovery Procedure
- Read Characteristic Value By Handle
- Read Characteristic Value By UUID
- Write Characteristic Value
- Write Long Characteristic Values

APP can't start other procedure before the current procedure is completed. Otherwise the other procedure will fail to be started.

Bluetooth Technology protocol stack may send exchange MTU request after connection is successfully established. GAP layer will send message GAP_MSG_LE_CONN_MTU_INFO to inform application that the exchange MTU procedure has been completed. So APP can start procedures listed above after receiving GAP_MSG_LE_CONN_MTU_INFO.

```
void app_handle_conn_mtu_info_evt(uint8_t conn_id, uint16_t mtu_size)
{
    APP_PRINT_INFO2("app_handle_conn_mtu_info_evt: conn_id %d, mtu_size %d", conn_id, mtu_size);
    app_discov_services(conn_id, true);
}
```

3.2.3.8.2 Commands

Commands that do not require a response do not have any flow control in ATT Layer.

- Write Without Response
- Signed Write Without Response

Because of limited resource, Bluetooth Technology protocol stack uses flow control to manage commands.

Flow control for Write Command and Signed Write Command is implemented with a credits value maintained in GAP layer, which allows APP to send commands in number of credits without waiting for response from Bluetooth Technologyprotocol stack. Bluetooth Technology protocol stack can cache commands in number of credits.

- Credit count decreases by one when profile client layer sends command to Bluetooth Technology protocol stack
- Credit count increases by one when profile client layer receives the response from Bluetooth Technology protocol stack. When the command is sent to air, Bluetooth Technology protocol stack will send response to profile client layer.
- Command shall only be sent when credit count is greater than zero.

Callback function write_result_cb() can inform the current credit. Or APP can also call le_get_gap_param() to get GAP_PARAM_LE_REMAIN_CREDITS.

```
void test(void)
{
    uint8_t wds_credits;
    le_get_gap_param(GAP_PARAM_LE_REMAIN_CREDITS, &wds_credits);
}
```

3.3 GATT Profile Use Case

This chapter is used to show how to use GATT profile interfaces. This chapter is to give some of typical use cases.

3.3.1 ANCS Client

The sample code is located in **BLE peripheral project**.

ANCS routines are defined in ancs.c and ancs.h. If other applications want to access this function, the procedure is shown as below.

1. Copy ancs.h and ancs.c to the target application directory.
2. Configure parameter auth_sec_req_enable to true.

```
void app_le_gap_init(void)
{
#if F_BT_ANCS_CLIENT_SUPPORT
    uint8_t  auth_sec_req_enable = true;
#else
    uint8_t  auth_sec_req_enable = false;
#endif
    uint16_t auth_sec_req_flags = GAP_AUTHEN_BIT_BONDING_FLAG;
    ......
}
```

3. Initialize ANCS client in app_le_profile_init().

```
void app_le_profile_init(void)
{
    ......
#if F_BT_ANCS_CLIENT_SUPPORT
    client_init(1);
    ancs_init(APP_MAX_LINKS);
#endif
}
```

4. Start ANCS discovery when authentication procedure is completed.

```
void app_handle_authen_state_evt(uint8_t conn_id, uint8_t new_state, uint16_t cause)
{
    ......
    case GAP_AUTHEN_STATE_COMPLETE:
        {
            if (cause == GAP_SUCCESS)
            {
#if F_BT_ANCS_CLIENT_SUPPORT
                ancs_start_discovery(conn_id);
#endif
                APP_PRINT_INFO0("app_handle_authen_state_evt:  GAP_AUTHEN_STATE_COMPLETE
                                pair success");
            }
            else
```

```
                {
                    APP_PRINT_INFO0("app_handle_authen_state_evt:    GAP_AUTHEN_STATE_COMPLETE
                                    pair failed");
                }
            }
            break;
        ......
}
```

5. Customize the ANCS routines in ancs.c.

4 Bluetooth LE Sample Projects

There are four GAP roles defined for devices operating over an LE physical transport. SDK provides corresponding demo application for user's reference in development.

1. Broadcaster
   - Send advertising events
   - Cannot create connections
   - Demo application: *BLE Broadcaster Application*
2. Observer
   - Scan for advertising events
   - Cannot initiate connections
   - Demo application: *BLE Observer Application*
3. Peripheral
   - Send advertising events
   - Can accept the establishment of LE link and become a slave role in the link
   - Demo application: *BLE Peripheral Application*
4. Central
   - Scan for advertising events
   - Can initiate connection and become a master role in the link
   - Demo application: *BLE Central Application*
5. Multiple-role
   - Peripheral + Central
   - Demo application: *BLE Scatternet Application*
6. Peripheral + LE Advertising Extensions
   - Send advertising events with LE Advertising Extensions
   - Enable one or more advertising sets at a time
   - Enable advertising set for a duration or maximum number of extended advertising events
   - Transmit more data with extended advertising PDUs (observer or central support LE Advertising Extensions)
   - Can accept the establishment of LE link and become a slave role in the link on LE 1M PHY LE 2M PHY or LE Coded PHY (central support LE Advertising Extensions)
   - Demo application: *BLE BT5 Peripheral Application*
7. Central + LE Advertising Extensions
   - Extended scan for advertising packets on the primary advertising channel(LE 1M PHY or/and LE Coded

PHY

- Scan for a duration; periodic scan
- Can initiate connection and become a master role in the link on LE 1M PHY
  LE 2M PHY or LE Coded PHY (peripheral support LE Advertising Extensions)
- Demo application: *BLE BT5 Central Application*

8. Peripheral + Privacy
   - Send advertising events
   - Can accept the establishment of LE link and become a slave role in the link
   - Can use white list when remote device uses resolvable private address.
   - Demo application: *BLE Peripheral Privacy Application*

4.1 BLE Broadcaster Application

4.1.1 Introduction

This section provides an overview of BLE broadcaster application. The BLE broadcaster project implements a very simple Bluetooth LE broadcaster device and can be used as a framework for further development of broadcaster-role based applications.

Broadcaster role features:

- Send connectionless advertising events
- Cannot establish Bluetooth LE connections

Expose features:

- DLPS
  Need to configure F_BT_DLPS_EN to enable the function. (Default enabled)

4.1.2 Source Code Overview

The following sections describe vital parts of this application.

4.1.2.1 Initialization

main() function is invoked when the board is powered on or the chip resets and following initialization functions will be invoked:

```
int main(void)
{
```

```
    board_init();
    le_gap_init(0);
    gap_lib_init();
    app_le_gap_init();
    pwr_mgr_init();
    task_init();
    os_sched_start();
    return 0;
}
```

app_le_gap_init() function is used to initialize the GAP parameters, the user can easily customize the application by modifying the following parameters.

Please refer to chapter *Configure Advertising Parameters*.

```
void app_le_gap_init(void)
{
    /* Advertising parameters */
    uint8_t  adv_evt_type = GAP_ADTYPE_ADV_NONCONN_IND;
    uint8_t  adv_direct_type = GAP_REMOTE_ADDR_LE_PUBLIC;
    uint8_t  adv_direct_addr[GAP_BD_ADDR_LEN] = {0};
    uint8_t  adv_chann_map = GAP_ADVCHAN_ALL;
    uint8_t  adv_filter_policy = GAP_ADV_FILTER_ANY;
    uint16_t adv_int_min = DEFAULT_ADVERTISING_INTERVAL_MIN;
    uint16_t adv_int_max = DEFAULT_ADVERTISING_INTERVAL_MIN;
    ......
}
```

A device in the broadcast mode shall send data using non-connectable advertising events. So parameter adv_evt_type shall be configured to GAP_ADTYPE_ADV_NONCONN_IND or GAP_ADTYPE_ADV_SCAN_IND.

More information on LE GAP Initialization and Startup Flow can be found in chapter *GAP Initialization and Startup Flow*.

4.1.2.2 GAP Message Handler

app_handle_gap_msg() function is invoked whenever a GAP messages is received from the GAP. More information on GAP messages can be found in chapter *Bluetooth LE GAP Message*.

Broadcaster application will call le_adv_start() to start advertising when receives GAP_INIT_STATE_STACK_READY. When BLE broadcaster application is running on evolution board, the device sends non-connectable advertising.

```
void app_handle_dev_state_evt(T_GAP_DEV_STATE new_state, uint16_t cause)
{
```

```
    APP_PRINT_INFO3("app_handle_dev_state_evt: init state %d, adv state %d, cause 0x%x",
                    new_state.gap_init_state, new_state.gap_adv_state, cause);
    if (gap_dev_state.gap_init_state != new_state.gap_init_state)
    {
        if (new_state.gap_init_state == GAP_INIT_STATE_STACK_READY)
        {
            APP_PRINT_INFO0("GAP stack ready");
            /*stack ready*/
            le_adv_start();
        }
    }
}
```

4.1.3 APP Configurable Functions

APP Configurable Functions are defined in app_flags.h.

```
/** @brief  Config DLPS: 0-Disable DLPS, 1-Enable DLPS */
#define F_BT_DLPS_EN                    1
```

More information about DLPS refers to DLPS related document [错误!未找到引用源。].

4.1.4 Test Procedure

At first, please build and download the BLE Broadcaster application to the evolution board. Some basic functions of BLE Broadcaster Application are demonstrated above. To implement some complex functions, user needs to refer to the manuals and source codes provided by SDK for development. When BLE Broadcaster Application is being run on evolution board, the device sends non-connectable advertisement.

User can use air sniffer or observer application to look over the advertising packets.

4.1.4.1 Test with BLE Observer Application Device

The test procedure please refers to chapter *Test with BLE Broadcaster Application Device*.

4.2 BLE Observer Application

4.2.1 Introduction

The purpose of this chapter is to give an overview of the BLE observer application. The BLE observer project

implements a simple Bluetooth LE observer device and can be used as a framework for further development of observer-role based applications.

Observer role features:

- Receive advertising events
- Cannot initiate connections

Expose features:

- DLPS

  Need to configure F_BT_DLPS_EN to be enabled. (Default enabled)

4.2.2 Source Code Overview

The following sections describe vital parts of this application.

4.2.2.1 Initialization

main() function is invoked when the board is powered on or the chip resets and following initialization functions will be invoked:

```
int main(void)
{
    board_init();
    le_gap_init(0);
    gap_lib_init();
    app_le_gap_init();
    pwr_mgr_init();
    task_init();
    os_sched_start();
    return 0;
}
```

app_le_gap_init() function is used to initialize the scan parameters, the user can easily customize the application by modifying the following parameter values. Please refer to chapter *Configure Scan Parameters*.

```
void app_le_gap_init(void)
{
    /* Scan parameters */
    uint8_t  scan_mode = GAP_SCAN_MODE_PASSIVE;
    uint16_t scan_interval = DEFAULT_SCAN_INTERVAL;
    uint16_t scan_window = DEFAULT_SCAN_WINDOW;
    uint8_t  scan_filter_policy = GAP_SCAN_FILTER_ANY;
    uint8_t  scan_filter_duplicate = GAP_SCAN_FILTER_DUPLICATE_ENABLE;
    ......
```

```
}
```

More information on LE GAP Initialization and Startup Flow can be found in chapter *GAP Initialization and Startup Flow*.

4.2.2.2 GAP Message Handler

app_handle_gap_msg() function is invoked whenever a GAP messages is received from the GAP. More information on GAP messages can be found in chapter *Bluetooth LE GAP Message*.

Observer application will call le_scan_start() to start scan when receives GAP_INIT_STATE_STACK_READY. When BLE Observer Application is being run on Evolution Board, the Bluetooth LE device will start scan.

```
void app_handle_dev_state_evt(T_GAP_DEV_STATE new_state, uint16_t cause)
{
    if (gap_dev_state.gap_init_state != new_state.gap_init_state)
    {
        if (new_state.gap_init_state == GAP_INIT_STATE_STACK_READY)
        {
            APP_PRINT_INFO0("GAP stack ready");
            /*stack ready*/
            le_scan_start();
        }
    }
    ......
}
```

4.2.2.3 GAP Callback Handler

app_gap_callback() function is used to handle GAP callback messages. More information on GAP callback can be found in chapter *Bluetooth LE GAP Callback*.

When the device is in scanning mode, the device may receive advertising data. GAP Layer will send GAP_MSG_LE_SCAN_INFO to application when receives advertising data or scan response data.

```
T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    T_LE_CB_DATA *p_data = (T_LE_CB_DATA *)p_cb_data;

    switch (cb_type)
    {
        case GAP_MSG_LE_SCAN_INFO:
            APP_PRINT_TRACE5("GAP_MSG_LE_SCAN_INFO: bd_addr %s, bdtype %d, event 0x%x, rssi %d,
                             len %d",
```

```
                                    TRACE_BDADDR(p_data->p_le_scan_info->bd_addr),
                                    p_data->p_le_scan_info->remote_addr_type,
                                    p_data->p_le_scan_info->adv_type,
                                    p_data->p_le_scan_info->rssi,
                                    p_data->p_le_scan_info->data_len);
                /* User can split interested information by using the function as follow. */
                app_parse_scan_info(p_data->p_le_scan_info);
                break;
        }
}
```

app_parse_scan_info() function is sample function to parse advertising data and scan response data.

4.2.3 APP Configurable Functions

APP Configurable Functions are defined in app_flags.h.

```
/** @brief    Config DLPS: 0-Disable DLPS, 1-Enable DLPS */
#define F_BT_DLPS_EN                    1
```

The user can easily change the value of macro definition to switch on/off the function or copy the referred codes to other application.

More information about DLPS refers to DLPS related document [错误!未找到引用源。].

4.2.4 Test Procedure

At first, please build and download the BLE Observer application to the Evolution Board. Some basic functions of BLE Observer Application are demonstrated above. To implement some complex functions, user needs to refer to the manuals and source codes provided by SDK for development.

Please use DebugAnalyser Tool to get the log.

When BLE Observer Application is being run on Evolution Board, the Bluetooth LE device will start scan.

Logs are shown as below:

```
[APP] !**GAP scan start
```

If local device receives advertising data or scan response data, application will receive message GAP_MSG_LE_SCAN_INFO.

Logs are shown as below:

```
[APP] GAP_MSG_LE_SCAN_INFO: bd_addr CC::17::73::36::05::CF, bdtype 1, event 0x0, rssi -81, len 31
[APP] app_parse_scan_info: AD Structure Info: AD type 0x1, AD Data Length 1
[APP] !**GAP_ADTYPE_FLAGS: 0x4
```

4.2.4.1 Test with BLE Broadcaster Application Device

Before demonstration, two Evolution Boards shall be prepared, one for running BLE Observer Application, and the other for running BLE Broadcaster Application.

The logs of broadcaster application are shown as below:

```
[APP] !**GAP adv start
```

When receive the advertising of broadcaster application, observer application logs are shown as below.

```
[APP] GAP_MSG_LE_SCAN_INFO: bd_addr 00::80::25::49::78::43, bdtype 0, event 0x3, rssi -18, len 20
[APP] app_parse_scan_info: AD Structure Info: AD type 0x1, AD Data Length 1
[APP] !**GAP_ADTYPE_FLAGS: 0x4
[APP] app_parse_scan_info: AD Structure Info: AD type 0x9, AD Data Length 15
[APP] !**GAP_ADTYPE_LOCAL_NAME_XXX: BLE_BROADCASTER
```

4.3 BLE Peripheral Application

4.3.1 Introduction

The purpose of this chapter is to give an overview of the BLE peripheral application. The BLE peripheral project implements a simple Bluetooth LE peripheral device with GATT services and can be used as a framework for further development of peripheral-role based applications.

Peripheral role features:

- Send advertising events
- Can accept the establishment of LE link and become a slave role in the link

Expose features:

- DLPS:

  Need to configure F_BT_DLPS_EN to be enabled. (Default enabled)
- Link number:

  APP_MAX_LINKS in app_flags.h. (Default one link)
- Supported GATT services:

  GAP and GATT Inbox Services

  Battery Service

  Simple Bluetooth LE Service
- Supported GATT clients:

ANCS Client: Need to configure F_BT_ANCS_CLIENT_SUPPORT to be enabled. (Default disabled)

4.3.2 Source Code Overview

The following sections describe important parts of this application.

4.3.2.1 Initialization

main() function is invoked when the board is powered on or the chip resets and following initialization functions will be invoked:

```
int main(void)
{
    board_init();
    le_gap_init(APP_MAX_LINKS);
    gap_lib_init();
    app_le_gap_init();
    app_le_profile_init();
    pwr_mgr_init();
    task_init();
    os_sched_start();
    return 0;
}
```

GAP and GATT Profiles initialization flow:

- le_gap_init() - Initialize GAP and set link number
- gap_lib_init() - Initialize GAP lib. Please refer to chapter *错误!未找到引用源。*.
- app_le_gap_init() - GAP Parameter Initialization, the user can easily customize the application by modifying the following parameter values.
  - *Configure Device Name and Device Appearance*
  - *Configure Advertising Parameters*
  - *Configure Bond Manager Parameters*
  - *Configure Other Parameters*
- app_le_profile_init() - Initialize GATT Profile

More information on LE GAP Initialization and Startup Flow can be found in chapter *GAP Initialization and Startup Flow*.

4.3.2.2 GAP Message Handler

app_handle_gap_msg() function is invoked whenever a GAP messages is received from the GAP. More information on GAP messages can be found in chapter *Bluetooth LE GAP Message*.

Peripheral application will call le_adv_start() to start advertising when receives GAP_INIT_STATE_STACK_READY. When BLE Peripheral Application is being run on Evolution Board, the Bluetooth LE device becomes discoverable and connectable. Remote device can scan the peripheral device and create connection with peripheral device.

```
void app_handle_dev_state_evt(T_GAP_DEV_STATE new_state, uint16_t cause)
{
    if (gap_dev_state.gap_init_state != new_state.gap_init_state)
    {
        if (new_state.gap_init_state == GAP_INIT_STATE_STACK_READY)
        {
            APP_PRINT_INFO0("GAP stack ready");
            /*stack ready*/
            le_adv_start();
        }
    }
    ......
}
```

Peripheral application will call le_adv_start() to start advertising when receives GAP_CONN_STATE_DISCONNECTED. After disconnection, Peripheral Application will be restored to the status that is discoverable and connectable again.

```
void app_handle_conn_state_evt(uint8_t conn_id, T_GAP_CONN_STATE new_state, uint16_t disc_cause)
{
    switch (new_state)
    {
    case GAP_CONN_STATE_DISCONNECTED:
        {
            if ((disc_cause != (HCI_ERR | HCI_ERR_REMOTE_USER_TERMINATE))
                && (disc_cause != (HCI_ERR | HCI_ERR_LOCAL_HOST_TERMINATE)))
            {
                APP_PRINT_ERROR1("app_handle_conn_state_evt: connection lost    cause 0x%x", disc_cause);
            }
            le_adv_start();
        }
        break;
    ......
}
```

4.3.2.3 GAP Callback Handler

app_gap_callback() function is used to handle GAP callback messages. More information on GAP callback can be found in chapter *Bluetooth LE GAP Callback*.

4.3.2.4 Profile Message Callback

When APP uses xxx_add_service to register specific service, APP shall register the callback function to handle the message from the specific service. APP shall call server_register_app_cb to register the callback function used to handle the message from the profile server layer.

APP can register different callback functions to handle different services or register the general callback function to handle all messages from specific services and profile server layer.

app_profile_callback() function is the general callback function. app_profile_callback() can distinguish different services by service id.

```
void app_le_profile_init(void)
{
    server_init(2);
    simp_srv_id = simp_ble_service_add_service(app_profile_callback);
    bas_srv_id  = bas_add_service(app_profile_callback);
    server_register_app_cb(app_profile_callback);
}
```

1. General profile server callback

   SERVICE_PROFILE_GENERAL_ID is the service id used by profile server layer. Message used by profile server layer contains two message types:

   - PROFILE_EVT_SRV_REG_COMPLETE : Services registration process has been completed in GAP Start Flow.
   - PROFILE_EVT_SEND_DATA_COMPLETE : This message is used by profile server layer to inform the result of sending the notification/indication.

```
T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    if (service_id == SERVICE_PROFILE_GENERAL_ID)
    {
        T_SERVER_APP_CB_DATA *p_param = (T_SERVER_APP_CB_DATA *)p_data;
        switch (p_param->eventId)
        {
            case PROFILE_EVT_SRV_REG_COMPLETE:// srv register result event.
                APP_PRINT_INFO1("PROFILE_EVT_SRV_REG_COMPLETE: result %d",
```

```
                                        p_param->event_data.service_reg_result);
                    break;
                case PROFILE_EVT_SEND_DATA_COMPLETE:
                    break;
            }
}
```

2. Battery Service

bas_srv_id is the service id of battery service.

```
T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    ......
    else if (service_id == bas_srv_id)
    {
        T_BAS_CALLBACK_DATA *p_bas_cb_data = (T_BAS_CALLBACK_DATA *)p_data;
        switch (p_bas_cb_data->msg_type)
        {
            case SERVICE_CALLBACK_TYPE_INDIFICATION_NOTIFICATION:
            ......
        }
}
```

3. Simple Bluetooth LE Service

simp_srv_id is the service id of simple Bluetooth LE service.

```
T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    ......
    else  if (service_id == simp_srv_id)
    {
        TSIMP_CALLBACK_DATA *p_simp_cb_data = (TSIMP_CALLBACK_DATA *)p_data;
        switch (p_simp_cb_data->msg_type)
        {
            case SERVICE_CALLBACK_TYPE_INDIFICATION_NOTIFICATION:
            ......
        }
}
```

4.3.3 APP Configurable Functions

APP Configurable Functions are defined in app_flags.h.

```
/** @brief  Config APP LE link number */
#define APP_MAX_LINKS  1
```

```
/** @brief    Config DLPS: 0-Disable DLPS, 1-Enable DLPS */
#define F_BT_DLPS_EN                                1
/** @brief    Config ANCS Client: 0-Not built in, 1-Open ANCS client function */
#define F_BT_ANCS_CLIENT_SUPPORT                    0
#define F_BT_ANCS_APP_FILTER                        (F_BT_ANCS_CLIENT_SUPPORT & 1)
#define F_BT_ANCS_GET_APP_ATTR                      (F_BT_ANCS_CLIENT_SUPPORT & 0)
/** @brief    Config ANCS Client debug log: 0-close, 1-open    */
#define F_BT_ANCS_CLIENT_DEBUG                      (F_BT_ANCS_CLIENT_SUPPORT & 0)
```

4.3.3.1 DLPS

More information about DLPS refers to DLPS related document [错误!未找到引用源。].

4.3.3.2 ANCS Client

Need to configure F_BT_ANCS_CLIENT_SUPPORT to be enabled.

More information can be found in chapter *ANCS Client*.

4.3.4 Test Procedure

At first, please build and download the BLE Peripheral application to the Evolution Board. Some basic functions of BLE Peripheral Application are demonstrated above. To implement some complex functions, user needs to refer to the manuals and source codes provided by SDK for development.

When BLE Peripheral Application is being run on Evolution Board, the Bluetooth LE device becomes discoverable and connectable. Remote device can scan the peripheral device and create connection with peripheral device. After disconnection, BLE Peripheral Application will restore to be discoverable and connectable again.

4.3.4.1 Test with iOS Device

Procedure Description: iOS-based devices are always compatible with Bluetooth LE, and devices running BLE Peripheral Application can be discovered. It is recommended to use Bluetooth LE-related App (e.g. LightBlue) in App Store to perform search and connection test.

Procedure: Run LightBlue on iOS device to search for and connect a BLE_PERIPHERAL device, as shown in Figure 4-1:

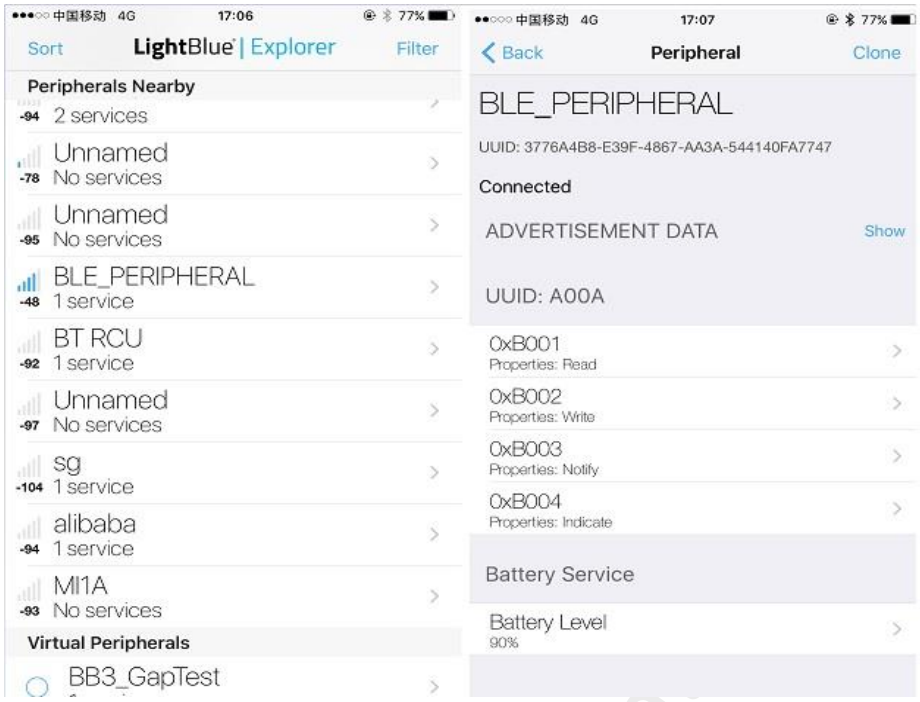

Figure 4-1 Test with iOS Device

4.3.4.2 Test with BLE Central Application Device

BLE peripheral application can test with BLE central application. More information on central application can be found in chapter *BLE Central Application*.

The test procedure please refers to chapter *Test with BLE Peripheral Application Device*.

4.4 BLE Central Application

4.4.1 Introduction

The purpose of this chapter is to give an overview of the BLE central application. The BLE central project implements a very simple Bluetooth LE central device and can be used as a framework for futher development of central-role based applications.

Central role features:

- Scan for advertising events
- Can initiate connection and become a master role in the link
- Can connect with more than one peripheral device

Expose features:

- Link number:

APP_MAX_LINKS in app_flags.h. (Default two link)

- GATT services handles storage:

  Need to configure F_BT_GATT_SRV_HANDLE_STORAGE to open. (Default closed)

- Supported GATT services:

  GAP and GATT Inbox Services

- Supported GATT clients:

  GAP Service Client

  Simple Bluetooth LE Service Client

  Battery Service Client

4.4.2 Source Code Overview

The following sections describe important parts of this application.

4.4.2.1 Initialization

main() is invoked when the board is powered on or the chip resets and following initialization functions will be invoked:

```
int main(void)
{
    board_init();
    le_gap_init(APP_MAX_LINKS);
    gap_lib_init();
    app_le_gap_init();
    app_le_profile_init();
    pwr_mgr_init();
    task_init();
    os_sched_start();

    return 0;
}
```

GAP and GATT Profiles initialization flow:

- le_gap_init() - Initialize GAP and set link number
- gap_lib_init() - Initialize GAP lib. Please refer to chapter *错误!未找到引用源。*.
- app_le_gap_init() - GAP Parameter Initialization, the user can easily customize the application by modifying the following parameter values.
  - *Configure Device Name and Device Appearance*

- *Configure Scan Parameters*
- *Configure Bond Manager Parameters*

- app_le_profile_init() - Initialize GATT Profile

More information on LE GAP Initialization and Startup Flow can be found in chapter *GAP Initialization and Startup Flow*.

4.4.2.2 Multilink Manager

Application link control block is defined in link_mgr.h .

```
typedef struct {
    T_GAP_CONN_STATE            conn_state;
    uint8_t                     discovered_flags;
    uint8_t                     srv_found_flags;
    T_GAP_REMOTE_ADDR_TYPE    bd_type;
    uint8_t                     bd_addr[GAP_BD_ADDR_LEN];
} T_APP_LINK;
extern T_APP_LINK app_link_table[APP_MAX_LINKS];
```

app_link_table is used to save the connection link related information.

4.4.2.3 GAP Message Handler

app_handle_gap_msg() function is invoked whenever a GAP messages is received from the GAP. More information on GAP messages can be found in chapter *Bluetooth LE GAP Message*.

When receives message GAP_MSG_LE_CONN_STATE_CHANGE, app_handle_conn_state_evt() will update the information in app_link_table.

```
void app_handle_conn_state_evt(uint8_t conn_id, T_GAP_CONN_STATE new_state, uint16_t disc_cause)
{
    if (conn_id >= APP_MAX_LINKS)
    {
        return;
    }
    APP_PRINT_INFO4("app_handle_conn_state_evt: conn_id %d, conn_state(%d -> %d), disc_cause 0x%x",
                    conn_id, app_link_table[conn_id].conn_state, new_state, disc_cause);
    app_link_table[conn_id].conn_state = new_state;
    switch (new_state)
    {
    case GAP_CONN_STATE_DISCONNECTED:
    {
        if ((disc_cause != (HCI_ERR | HCI_ERR_REMOTE_USER_TERMINATE))
```

```
                && (disc_cause != (HCI_ERR | HCI_ERR_LOCAL_HOST_TERMINATE)))
            {
                APP_PRINT_ERROR2("app_handle_conn_state_evt: connection lost, conn_id %d, cause 0x%x",
                                 conn_id, disc_cause);
            }

            data_uart_print("Disconnect conn_id %d\r\n", conn_id);
            memset(&app_link_table[conn_id], 0, sizeof(T_APP_LINK));
        }
        break;

        case GAP_CONN_STATE_CONNECTED:
        {
            le_get_conn_addr(conn_id, app_link_table[conn_id].bd_addr,
                             &app_link_table[conn_id].bd_type);
            data_uart_print("Connected success conn_id %d\r\n", conn_id);
        }
        break;
        default:
            break;
        }
}
```

4.4.2.4 GAP Callback Handler

app_gap_callback() function is used to handle GAP callback messages. More information on GAP callback can be found in chapter *Bluetooth LE GAP Callback*.

4.4.2.5 Profile Message Callback

When APP uses xxx_add_client() to register specific client, APP shall register the callback function to handle the message from the specific client.

APP can register different callback functions to handle messages from different clients or register the general callback function to handle all messages from specific clients.

app_client_callback() function is the general callback function. app_client_callback() can distinguish different clients by client id.

```
void app_le_profile_init(void)
{
    client_init(3);
    gaps_client_id    = gaps_add_client(app_client_callback, APP_MAX_LINKS);
```

```
    simple_ble_client_id = simp_ble_add_client(app_client_callback, APP_MAX_LINKS);
    bas_client_id = bas_add_client(app_client_callback, APP_MAX_LINKS);
}
```

More information can be found in chapter *Bluetooth LE Profile Client*.

1. Gap service client

   gaps_client_id is the client id of gap service client.

```
T_APP_RESULT app_client_callback(T_CLIENT_ID client_id, uint8_t conn_id, void *p_data)
{
    T_APP_RESULT  result = APP_RESULT_SUCCESS;
    APP_PRINT_INFO2("app_client_callback: client_id %d, conn_id %d",
                    client_id, conn_id);
    if (client_id == gaps_client_id)
    {
        T_GAPS_CLIENT_CB_DATA *p_gaps_cb_data = (T_GAPS_CLIENT_CB_DATA *)p_data;
        switch (p_gaps_cb_data->cb_type)
        {
            case GAPS_CLIENT_CB_TYPE_DISC_STATE:
            .....
        }
```

2. Battery Service Client

   bas_client_id is the client id of battery service client.

```
T_APP_RESULT app_client_callback(T_CLIENT_ID client_id, uint8_t conn_id, void *p_data)
{
    T_APP_RESULT  result = APP_RESULT_SUCCESS;
    APP_PRINT_INFO2("app_client_callback: client_id %d, conn_id %d",
                    client_id, conn_id);
    else if (client_id == bas_client_id)
    {
        T_BAS_CLIENT_CB_DATA *p_bas_cb_data = (T_BAS_CLIENT_CB_DATA *)p_data;
        switch (p_bas_cb_data->cb_type)
        {
            case BAS_CLIENT_CB_TYPE_DISC_STATE:
            .....
        }
```

3. Simple Bluetooth LE Service Client

   simple_ble_client_id is the client id of simple Bluetooth LE service client.

```
T_APP_RESULT app_client_callback(T_CLIENT_ID client_id, uint8_t conn_id, void *p_data)
{
    T_APP_RESULT  result = APP_RESULT_SUCCESS;
    APP_PRINT_INFO2("app_client_callback: client_id %d, conn_id %d",
                    client_id, conn_id);
    else if (client_id == simple_ble_client_id)
```

```
{
    T_SIMP_CLIENT_CB_DATA *p_simp_client_cb_data = (T_SIMP_CLIENT_CB_DATA *)p_data;
    uint16_t value_size;
    uint8_t *p_value;
    switch (p_simp_client_cb_data->cb_type)
    {
        case SIMP_CLIENT_CB_TYPE_DISC_STATE:
        .....
    }
```

4.4.2.6 Discover GATT Services Procedure

Central application will automatically discover services after receiving message LE_GAP_MSG_TYPE_CONN_MTU_INFO. Discover GATT services procedure is defined in app_discov_services(). More information can be found in app_discov_services() in central_app.c.

```
void app_handle_conn_mtu_info_evt(uint8_t conn_id, uint16_t mtu_size)
{
    APP_PRINT_INFO2("app_handle_conn_mtu_info_evt: conn_id %d, mtu_size %d", conn_id, mtu_size);
    app_discov_services(conn_id, true);
}
```

app_discov_services (uint8_t conn_id, bool start) flow chart is shown in Figure 4-2:

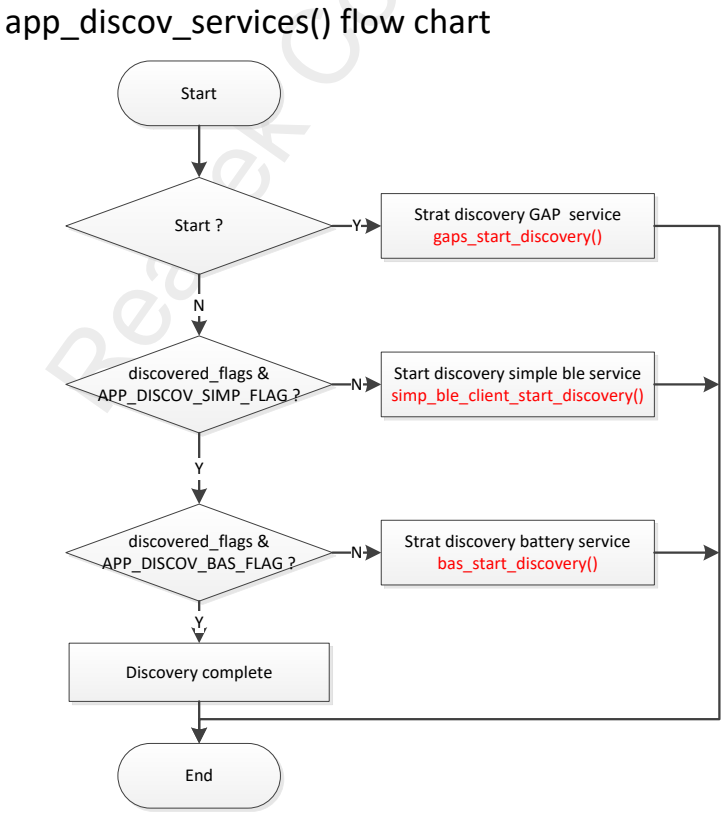

Figure 4-2 app_discov_services() Flow Chart

The application will call app_discov_services() when receives DISC_GAPS_DONE , DISC_SIMP_DONE and DISC_BAS_DONE .

Reference code is shown as below:

```
T_APP_RESULT app_client_callback(T_CLIENT_ID client_id, uint8_t conn_id, void *p_data)
{
    ......
case DISC_GAPS_DONE:
    app_link_table[conn_id].discovered_flags |= APP_DISCOV_GAPS_FLAG;
    app_link_table[conn_id].srv_found_flags |= APP_DISCOV_GAPS_FLAG;
    app_discov_services(conn_id, false);
    /* Discovery Simple BLE service procedure successfully done. */
    APP_PRINT_INFO0("app_client_callback: discover gaps procedure done.");
    break;
    ......
case DISC_SIMP_DONE:
    /* Discovery Simple BLE service procedure successfully done. */
    app_link_table[conn_id].discovered_flags |= APP_DISCOV_SIMP_FLAG;
    app_link_table[conn_id].srv_found_flags |= APP_DISCOV_SIMP_FLAG;
    app_discov_services(conn_id, false);
    APP_PRINT_INFO0("app_client_callback: discover simp procedure done.");
    break;
    ......
case DISC_BAS_DONE:
    /* Discovery BAS procedure successfully done. */
    app_link_table[conn_id].discovered_flags |= APP_DISCOV_BAS_FLAG;
    app_link_table[conn_id].srv_found_flags |= APP_DISCOV_BAS_FLAG;
    app_discov_services(conn_id, false);
    APP_PRINT_INFO0("app_client_callback: discover bas procedure done");
    break;
    ......
}
```

4.4.3 APP Configurable Functions

APP Configurable Functions are defined in app_flags.h.

```
/** @brief  Config APP LE link number */
#define APP_MAX_LINKS  2
/** @brief  Config GATT services storage: 0-Not save, 1-Save to flash
 *
 * If configure to 1, the GATT services discovery results will save to the flash.
 */
#define F_BT_GATT_SRV_HANDLE_STORAGE                0
```

The user can easily change the value of macro definition to switch on/off the function or copy the referenced codes to other application.

4.4.3.1 GATT services handles storage

Reference code is shown as below:

```
typedef struct {
    uint8_t         srv_found_flags;
    uint8_t         bd_type;
    uint8_t         bd_addr[GAP_BD_ADDR_LEN];
    uint32_t        reserved;
    uint16_t        gaps_hdl_cache[HDL_GAPS_CACHE_LEN];
    uint16_t        simp_hdl_cache[HDL_SIMBLE_CACHE_LEN];
    uint16_t        bas_hdl_cache[HDL_BAS_CACHE_LEN];
} T_APP_SRVS_HDL_TABLE;
uint32_t app_save_srvs_hdl_table(T_APP_SRVS_HDL_TABLE *p_info)
uint32_t app_load_srvs_hdl_table(T_APP_SRVS_HDL_TABLE *p_info)
```

app_save_srvs_hdl_table() is used to save the information to flash.

app_load_srvs_hdl_table() is used to load the information from flash.

app_discov_services() flow chart is shown in Figure 4-3:

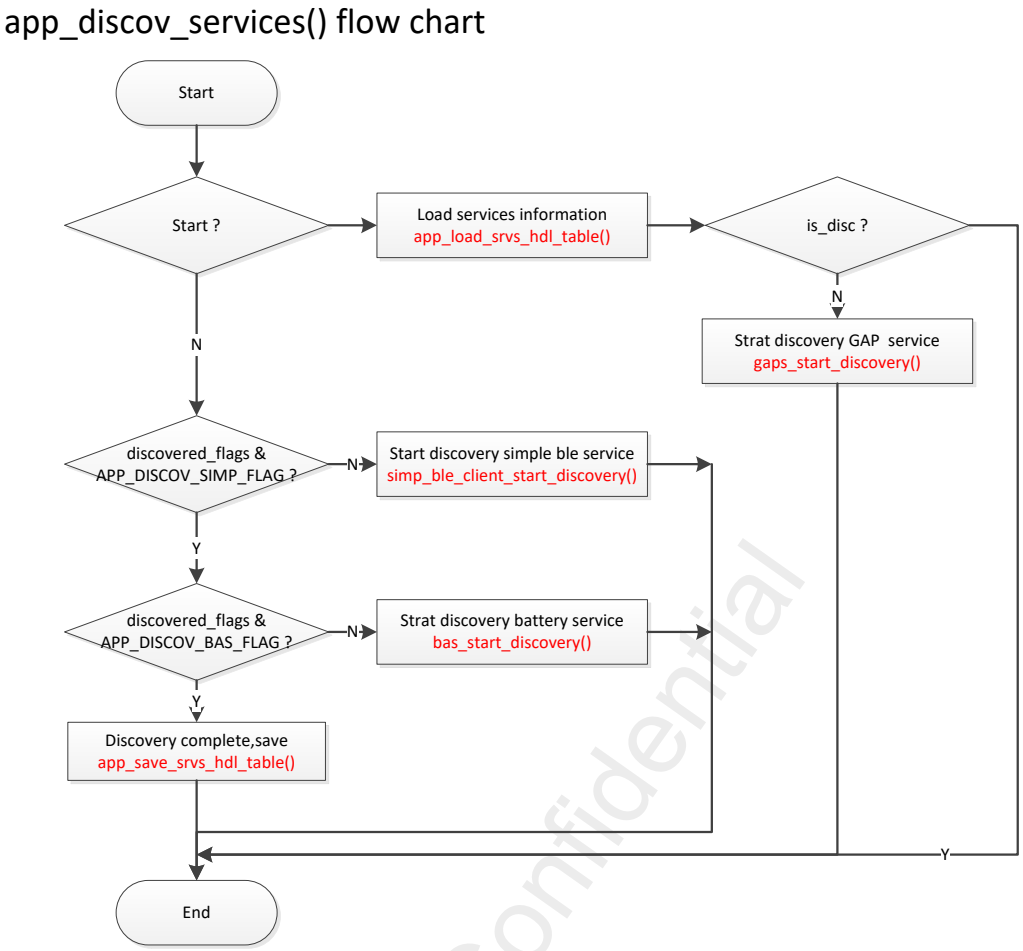

Figure 4-3 app_discov_services() Flow Chart (F_BT_GATT_SRV_HANDLE_STORAGE)

4.4.4 User Command

BLE Central Application needs to perform interaction by inputting command through Data UART in PC. More information on user command can be found in chapter *BLE Application User Command*.

4.4.5 Test Procedure

Firstly, please build and download the BLE Central application to the Evolution Board. Link number shall be configured by MPTool. Information about LE Link number please refers to *Configuration of LE Link number*. Some basic functions of BLE Central Application are demonstrated above. To implement some complex functions, user needs to refer to the manuals and source codes provided by SDK for development.

By using the commands as described in *BLE Application User Command*, some processes of GAP Central Role and GATT Client can be demonstrated.

Please use DebugAnalyser Tool to get the log.

4.4.5.1 Test with BLE Peripheral Application Device

BLE Central Application can test with BLE peripheral application. More information on peripheral can be found in chapter *BLE Peripheral Application*.

1. one link test

   At first, please build and download the BLE Peripheral application to the Evolution Board. Bluetooth address shall be configured by MPTool, please set the address to "x00 x11 x22 x33 x44 x00".

   1) Scanning

      Procedure Description: Demonstrate search for any Bluetooth LE device discoverable nearby.

      Procedure:

      scan 0 - Start scan and view information on Bluetooth LE device nearby discovered.

Log tool shows : [APP] !**GAP scan start [APP] GAP_MSG_LE_SCAN_INFO: bd_addr 00::11::22::33::44::00, bdtype 0, adv_type 0x0, rssi -17, data_len 23

      stopscan - Stop scan.

Log tool shows : [APP] !**GAP scan stop

      showdev - Show scan device list, the device list is filtered by simple Bluetooth LE service uuid.

Serial port assistant tool shows : Advertising and Scan response: filter uuid = 0xA00A dev list RemoteBd[0] = [00:11:22:33:44:00] type = 0

   2) Connection and Reconnection

      Procedure Description: Demonstrate connecting and disconnecting with the peripheral device.

      Procedure:

      condev 0 - Initiate connection.

Serial port assistant tool shows : Connected success conn_id 0

      showcon - Show connection information.

Serial port assistant tool shows :

ShowCon conn_id 0 state 0x00000002 role 1

RemoteBd = [00:11:22:33:44:00] type = 0

active link num 1, idle link num 1

conupdreq 0 1 - Initiate connection parameter update request.

Log tool shows :

[APP] !**app_handle_conn_param_update_evt update success:conn_id 0, conn_interval 0xa, conn_slave_latency 0x0, conn_supervision_timeout 0x3e8

disc 0 - Disconnect with the peripheral device.

Serial port assistant tool shows :

Disconnect conn_id 0

condev 0 - Reconnect with the peripheral device.

Serial port assistant tool shows :

Connected success conn_id 0

3) Pair

Procedure Description: Demonstrate pairing with the peripheral device.

Procedure:

sauth 0 - Send authentication request.

Serial port assistant tool shows :

Pair success

bondinfo - Show bonding information.

Serial port assistant tool shows :

bond_dev[0]: bd 0x001122334400, addr_type 0, flags 0x000000bf

4) GATT services

Procedure Description: Demonstrate GATT services referenced procedures.

GAP service procedure:

gaphdl 0 - Print out the list of handles acquired in GAP service.

Serial port assistant tool shows :

–>Index 0 – Handle 0x00000005

–>Index 1 – Handle 0x0000000d

–>Index 2 – Handle 0x00000007

–>Index 3 – Handle 0x00000009

–>Index 4 – Handle 0x0000000d

–>Index 5 – Handle 0x00000000

gapread 0 0 - Read the value of device name.

Log tool shows :

[APP] !**GAPS_READ_DEVICE_NAME: device name BLE_PERIPHERAL.

gapread 0 1 - Read the value of appearance.

Log tool shows :

[APP] !**GAPS_READ_APPEARANCE: appearance 0

Battery service procedure:

bashdl 0 - Print out the list of handles acquired in battery service.

Serial port assistant tool shows :

–>Index 0 – Handle 0x00000019

> –>Index 1 – Handle 0x0000ffff

> –>Index 2 – Handle 0x0000001b

> –>Index 3 – Handle 0x0000001c

basread 0 0 - Read the value of battery level value.

Log tool shows :

[APP] !**BAS_READ_BATTERY_LEVEL: battery level 90

basread 0 1 - Read CCCD bit of battery level characteristic.

Log tool shows :

[APP] !**BAS_READ_NOTIFY: notify 0

bascccd 0 1 - Write CCCD bit of battery level characteristic.

Log tool shows :

[APP] !**BAS_WRITE_NOTIFY_ENABLE: write result 0x0

Simple Bluetooth LE service procedure:

simphdl 0 - Print out the list of handles acquired in simple Bluetooth LE service.

Serial port assistant tool shows :

–>Index 0 – Handle 0x0000000e

–>Index 1 – Handle 0x00000018

–>Index 2 – Handle 0x00000010

–>Index 3 – Handle 0x00000012

–>Index 4 – Handle 0x00000014

–>Index 5 – Handle 0x00000015

–>Index 6 – Handle 0x00000017

–>Index 7 – Handle 0x00000018

simpread 0 0 0 - Read the value of V1 characteristic through handle.

Log tool shows :

[APP] !**SIMP_READ_V1_READ: value_size 2, value 01-02

simpread 0 0 1 - Read the value of V1 characteristic through uuid.

Log tool shows :

[APP] !**SIMP_READ_V1_READ: value_size 2, value 01-02

simpread 0 1 0 - Read CCCD bit of V3 characteristic.

Log tool shows :

[APP] !**SIMP_READ_V3_NOTIFY_CCCD: notify 0

simpread 0 2 0 - Read CCCD bit of V4 characteristic.

Log tool shows :

[APP] !**SIMP_READ_V4_INDICATE_CCCD: indicate 0

simpcccd 0 0 1 - Write CCCD bit of V3 characteristic.

Log tool shows :

[APP] !**SIMP_WRITE_V3_NOTIFY_CCCD: write result 0x0

simpwritev2 0 1 10 - Write V2 characteristic value using write request.

Log tool shows :

[APP] !**SIMP_WRITE_V2_WRITE: write result 0x0

2. two links test

Two Evolution Boards shall be prepared, and download BLE Peripheral application. Bluetooth address shall be configured by MPTool, please set the address to "x00 x11 x22 x33 x44 x00" and "x00 x11 x22 x33 x44 x01".

Test Procedure:

scan 0 - Start scan and view information on Bluetooth LE device nearby discovered.

Log tool shows :

[APP] !**GAP scan start

[APP] GAP_MSG_LE_SCAN_INFO: bd_addr 00::11::22::33::44::00, bdtype 0, adv_type 0x0, rssi -17, data_len 23

[APP] GAP_MSG_LE_SCAN_INFO: bd_addr 00::11::22::33::44::01, bdtype 0, adv_type 0x0, rssi -17, data_len 23

stopscan - Stop scan.

Log tool shows :

[APP] !**GAP scan stop

showdev - Show scan device list, the device list is filtered by simple Bluetooth LE serice uuid.

Serial port assistant tool shows :

Advertising and Scan response: filter uuid = 0xA00A dev list

RemoteBd[0] = [00:11:22:33:44:00] type = 0

RemoteBd[1] = [00:11:22:33:44:01] type = 0

condev 0 - Initiate connection.

Serial port assistant tool shows :

Connected success conn_id 0

condev 1 - Initiate connection.

Serial port assistant tool shows :

Connected success conn_id 1

sauth 0 - Send authentication request(conn_id=0).

Serial port assistant tool shows :

Pair success

sauth 1 - Send authentication request(conn_id=1).

Serial port assistant tool shows :

Pair success

gapread 0 0 - Read the value of device name(conn_id=0).

Log tool shows :

[APP] !**GAPS_READ_DEVICE_NAME: device name BLE_PERIPHERAL.

gapread 1 0 - Read the value of device name(conn_id=1).

Log tool shows :

[APP] !**GAPS_READ_DEVICE_NAME: device name BLE_PERIPHERAL.

4.5 BLE Scatternet Application

4.5.1 Introduction

The purpose of this chapter is to give an overview of the BLE scatternet application. The BLE scatternet project can be used as a framework for developing multiple-role based applications. This project supports multiple-role including broadcaster, observer, peripheral and central role.

Scatternet topology features:

- Slaves are permitted to establish physical links to more than one master at a time.

- A device is permitted to be master and slave at the same time.
- Not support role switch

Expose features:

- Link number:

  APP_MAX_LINKS in app_flags.h. (Default two link)
- Supported GATT services:

  GAP and GATT Inbox Services

  Simple Bluetooth LE Service

  Battery Service
- Supported GATT clients:

  GAP Service Client

  Simple Bluetooth LE Service Client

  Battery Service Client
- Airplane mode setting:

  Need to configure F_BT_AIRPLANE_MODE_SUPPORT to open. (Default closed)
- GAP Service characteristic writeable:

  Need to configure F_BT_GAPS_CHAR_WRITEABLE to open. (Default closed)
- Use static random address:

  Need to configure F_BT_LE_USE_STATIC_RANDOM_ADDR to open. (Default closed)
- Physical channel setting:

  Need to configure F_BT_LE_5_0_SET_PHY_SUPPORT to open. (Default closed)

4.5.2 Source Code Overview

The following sections describe important parts of this application.

4.5.2.1 Initialization

main() function is invoked when the board is powered on or the chip resets and following initialization functions will be invoked:

```
int main(void)
{
    board_init();
    le_gap_init(APP_MAX_LINKS);
    gap_lib_init();
    app_le_gap_init();
```

```
    app_le_profile_init();
    pwr_mgr_init();
    task_init();
    os_sched_start();
    return 0;
}
```

GAP and GATT Profiles initialization flow:

- le_gap_init() - Initialize GAP and set link number
- gap_lib_init() - Initialize GAP lib. Please refer to chapter *错误!未找到引用源。*.
- app_le_gap_init() - GAP Parameter Initialization, the user can easily customize the application by modifying the following parameter values.
  - *Configure Device Name and Device Appearance*
  - *Configure Advertising Parameters*
  - *Configure Scan Parameters*
  - *Configure Bond Manager Parameters*
- app_le_profile_init() - Initialize GATT Profile

More information on LE GAP Initialization and Startup Flow can be found in chapter *GAP Initialization and Startup Flow*.

4.5.2.2 Multilink Manager

Application link control block is defined in link_mgr.h .

```
typedef struct {
    T_GAP_CONN_STATE            conn_state;
    T_GAP_REMOTE_ADDR_TYPE    bd_type;
    uint8_t                         bd_addr[GAP_BD_ADDR_LEN];
} T_APP_LINK;
extern T_APP_LINK app_link_table[APP_MAX_LINKS];
```

app_link_table is used to save the connection link related information.

4.5.2.3 GAP Message Handler

app_handle_gap_msg() function is invoked whenever a GAP messages is received from the GAP. More information on GAP messages can be found in chapter *Bluetooth LE GAP Message*.

When receives message GAP_MSG_LE_CONN_STATE_CHANGE, app_handle_conn_state_evt will update the information in app_link_table.

```
void app_handle_conn_state_evt(uint8_t conn_id, T_GAP_CONN_STATE new_state,
```

```
                                    uint16_t disc_cause)
{
    if (conn_id >= APP_MAX_LINKS)
    {
        return;
    }
    APP_PRINT_INFO4("app_handle_conn_state_evt: conn_id %d, conn_state(%d -> %d), disc_cause 0x%x",
                    conn_id, app_link_table[conn_id].conn_state, new_state, disc_cause);
    app_link_table[conn_id].conn_state = new_state;
    switch (new_state)
    {
    case GAP_CONN_STATE_DISCONNECTED:
    {
        if ((disc_cause != (HCI_ERR | HCI_ERR_REMOTE_USER_TERMINATE))
            && (disc_cause != (HCI_ERR | HCI_ERR_LOCAL_HOST_TERMINATE)))
        {
            APP_PRINT_ERROR2("app_handle_conn_state_evt: connection lost, conn_id %d, cause 0x%x",
                             conn_id,disc_cause);
        }
        data_uart_print("Disconnect conn_id %d\r\n", conn_id);
        memset(&app_link_table[conn_id], 0, sizeof(T_APP_LINK));
    }
    break;
    case GAP_CONN_STATE_CONNECTED:
    {
    }
    break;
    default:
        break;
    }
}
```

4.5.2.4 GAP Callback Handler

app_gap_callback() function is used to handle GAP callback messages. More information on GAP callback can be found in chapter *Bluetooth LE GAP Callback*.

```
T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    T_LE_CB_DATA *p_data = (T_LE_CB_DATA *)p_cb_data;
    switch (cb_type)
    {
        /* common msg*/
```

```
        case GAP_MSG_LE_READ_RSSI:
            APP_PRINT_INFO3("GAP_MSG_LE_READ_RSSI:conn_id 0x%x cause 0x%x rssi %d",
                            p_data->p_le_read_rssi_rsp->conn_id,
                            p_data->p_le_read_rssi_rsp->cause,
                            p_data->p_le_read_rssi_rsp->rssi);
            break;
    }
}
```

4.5.2.5 Profile Client Message Callback

When APP uses xxx_add_client to register specific client, APP shall register the callback function to handle the message from the specific client. APP can register different callback functions to handle different clients or register the common callback function to handle all messages from specific clients.

app_client_callback() function is the common callback function. app_client_callback() can distinguish different clients by client id.

```
void app_le_profile_init(void)
{
    client_init(3);
    gaps_client_id    = gaps_add_client(app_client_callback, APP_MAX_LINKS);
    simple_ble_client_id = simp_ble_add_client(app_client_callback, APP_MAX_LINKS);
    bas_client_id = bas_add_client(app_client_callback, APP_MAX_LINKS);
    client_register_general_client_cb(app_client_callback);
}
T_APP_RESULT app_client_callback(T_CLIENT_ID client_id, uint8_t conn_id, void
                                 *p_data)
{
    T_APP_RESULT    result = APP_RESULT_SUCCESS;
    APP_PRINT_INFO2("app_client_callback: client_id %d, conn_id %d",
                    client_id, conn_id);
    if (client_id == CLIENT_PROFILE_GENERAL_ID)
    {
        T_CLIENT_APP_CB_DATA *p_client_app_cb_data = (T_CLIENT_APP_CB_DATA *)p_data;
        switch (p_client_app_cb_data->cb_type)
        {
            case CLIENT_APP_CB_TYPE_DISC_STATE:
            .....
        }
    }
    else if (client_id == gaps_client_id)
    {
        T_GAPS_CLIENT_CB_DATA *p_gaps_cb_data = (T_GAPS_CLIENT_CB_DATA *)p_data;
```

```
        switch (p_gaps_cb_data->cb_type)
        {
            case GAPS_CLIENT_CB_TYPE_DISC_STATE:
            .....
        }
    }
```

Scatternet APP supports several GATT profile clients including GAP Service Client, Simple Bluetooth LE Service Client and Battery Service Client.

4.5.2.6 Profile Service Message Callback

When APP uses xxx_add_service to register specific service, APP shall register the callback function to handle the message from the specific service. APP shall call server_register_app_cb to register the callback function used to handle the message from the profile server layer.

APP can register different callback functions to handle different services or register the common callback function to handle all messages from specific services and profile server layer.

app_profile_callback() function is the common callback function. app_profile_callback() can distinguish different services by service id.

```
void app_le_profile_init(void)
{
    server_init(2);
    simp_srv_id = simp_ble_service_add_service(app_profile_callback);
    bas_srv_id  = bas_add_service(app_profile_callback);
    server_register_app_cb(app_profile_callback);
}
```

More information can be found in chapter *Bluetooth LE Profile Server*.

1. General profile server callback

   SERVICE_PROFILE_GENERAL_ID is the service id used by profile server layer. Message used by profile server layer contains two message types:

   - PROFILE_EVT_SRV_REG_COMPLETE : Services registration process has been completed in GAP Start Flow.
   - PROFILE_EVT_SEND_DATA_COMPLETE : This message is used by profile server layer to inform the result of sending the notification/indication.

```
T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    if (service_id == SERVICE_PROFILE_GENERAL_ID)
    {
        T_SERVER_APP_CB_DATA *p_param = (T_SERVER_APP_CB_DATA *)p_data;
```

```
        switch (p_param->eventId)
        {
            case PROFILE_EVT_SRV_REG_COMPLETE:// srv register result event.
                APP_PRINT_INFO1("PROFILE_EVT_SRV_REG_COMPLETE: result %d",
                                p_param->event_data.service_reg_result);
                break;
            case PROFILE_EVT_SEND_DATA_COMPLETE:
            .....
        }
    }
```

2. Battery Service

   bas_srv_id is the service id of battery service.

```
T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    ......
    else if (service_id == bas_srv_id)
    {
        T_BAS_CALLBACK_DATA *p_bas_cb_data = (T_BAS_CALLBACK_DATA *)p_data;
        switch (p_bas_cb_data->msg_type)
        {
            case SERVICE_CALLBACK_TYPE_INDIFICATION_NOTIFICATION:
            ......
        }
```

3. Simple Bluetooth LE Service

   simp_srv_id is the service id of simple Bluetooth LE service.

```
T_APP_RESULT app_profile_callback(T_SERVER_ID service_id, void *p_data)
{
    T_APP_RESULT app_result = APP_RESULT_SUCCESS;
    ......
    else  if (service_id == simp_srv_id)
    {
        TSIMP_CALLBACK_DATA *p_simp_cb_data = (TSIMP_CALLBACK_DATA *)p_data;
        switch (p_simp_cb_data->msg_type)
        {
        case SERVICE_CALLBACK_TYPE_INDIFICATION_NOTIFICATION:
            {
            switch (p_simp_cb_data->msg_data.notification_indification_index)
            {
                case SIMP_NOTIFY_INDICATE_V3_ENABLE:
                ......
            }
```

4.5.3 APP Configurable Functions

APP Configurable Functions are defined in app_flags.h.

```
/** @brief    Config APP LE link number */
#define APP_MAX_LINKS    2
/** @brief    Config airplane mode support: 0-Not built in, 1-built in, use user command to set*/
#define F_BT_AIRPLANE_MODE_SUPPORT                  0
/** @brief    Config device name characteristic and appearance characteristic property: 0-Not writeable, 1-writeable, save to flash*/
#define F_BT_GAPS_CHAR_WRITEABLE                    0
/** @brief    Config set physical: 0-Not built in, 1-built in, use user command to set*/
#define F_BT_LE_5_0_SET_PHY_SUPPORT                 0
/** @brief    Config local address type: 0-pulic address, 1-static random address */
#define F_BT_LE_USE_STATIC_RANDOM_ADDR          0
```

The user can easily change the value of macro definition to on-off the function or copy the referenced codes to other application.

4.5.3.1 Airplane mode setting

Need to configure F_BT_AIRPLANE_MODE_SUPPORT to open.

More information can be found in chapter *Airplane Mode Setting*.

4.5.3.2 GAP Service characteristic writeable

Need to configure F_BT_GAPS_CHAR_WRITEABLE to open.

More information can be found in chapter *GAP Service Characteristic Writeable*.

4.5.3.3 Use static random address

Need to configure F_BT_LE_USE_STATIC_RANDOM_ADDR to open.

More information can be found in chapter *Local Static Random Address*.

4.5.3.4 PHY setting

Need to configure F_BT_LE_5_0_SET_PHY_SUPPORT to open.

More information can be found in chapter *Physical (PHY) Setting*.

4.5.4 User Command

BLE Scatternet Application needs to perform interaction by inputting command through Data UART in PC. More information on user command can be found in chapter *BLE Application User Command*. Due to support of peripheral role and central role, test procedure is the combination of test procedures in *BLE Peripheral Application* and *BLE Central Application*.

4.6 BLE BT5 Peripheral Application

4.6.1 Introduction

The purpose of this chapter is to give an overview of the BLE BT5 peripheral application. The BLE BT5 peripheral project implements a device with extended advertising and can be used as a framework for developing many different peripheral-role or broadcaster-role based applications.

Peripheral role features:

- Send advertising events with LE Advertising Extensions
- Enable one or more advertising sets at the same time
- Enable advertising set for a duration or maximum number of extended advertising events
- Hold more data with extended advertising PDUs (observer or central support LE Advertising Extensions)
- Can accept the establishment of LE link and become a slave role in the link on LE 1M PHY
  LE 2M PHY or LE Coded PHY (central support LE Advertising Extensions)

Expose features:

- DLPS:
  Need to configure F_BT_DLPS_EN to open. (Default opened)
- Link number:
  APP_MAX_LINKS in app_flags.h. (Default one link)
- Advertising PHY:
  ADVERTISING_PHY in app_flags.h. (Default ADVERTISING_PHY is APP_PRIMARY_1M_SECONDARY_2M, primary advertising PHY is LE 1M PHY, secondary advertising PHY is LE 2M PHY.)

Compatibility:

Peripheral device with LE Advertising Extensions can set duration of advertising or maximum number of extended advertising events, and enable or disable one or more advertising sets at a time.

If extended advertising PDUs are used, length of adv data or scan response data could be increased, and distance

that connection can be established could be increased with LE Coded PHY. More information about extended advertising PDUs and legacy advertising PDUs refer to *Configure GAP Extended Advertising Related Parameters*.

If peripheral device uses LE Advertising Extensions, user shall take compatibility with peer device of different Bluetooth Technology versions into account. Compatibility with peer device of different Bluetooth Technology versions is shown in Table 4-1.

Table 4-1 Compatibility of peripheral device with LE Advertising Extensions

Bluetooth 5 Feature	Advertising PDUs	Bluetooth 4.0	Bluetooth 4.1	Bluetooth 4.2	Bluetooth 5.0 (not use LE Advertising Extensions)	Bluetooth 5.0 (use LE Advertising Extensions)
LE Advertising Extensions	extended advertising PDUs	N	N	N	N	Y
LE Advertising Extensions	legacy advertising PDUs	Y	Y	Y	Y	Y

4.6.2 Source Code Overview

Due to support of LE Advertising Extensions, device can use one or more advertising sets. Application needs to create an advertising handle to identify advertising set, and extended advertising related parameters are configured for a specified advertising set. Using LE Advertising Extensions, application can choose to send legacy advertising PDUs (*BLE Peripheral Application* can only send legacy advertising PDUs) or extended advertising PDUs by modify advertising event properties.

The flow of starting extended advertising is shown in Figure 4-4. Optional Procedure is determined by advertising event properties, please refers to *Configure GAP Extended Advertising Related Parameters*. In BT5 peripheral application, implementation of the flow will be introduced in the following sections.

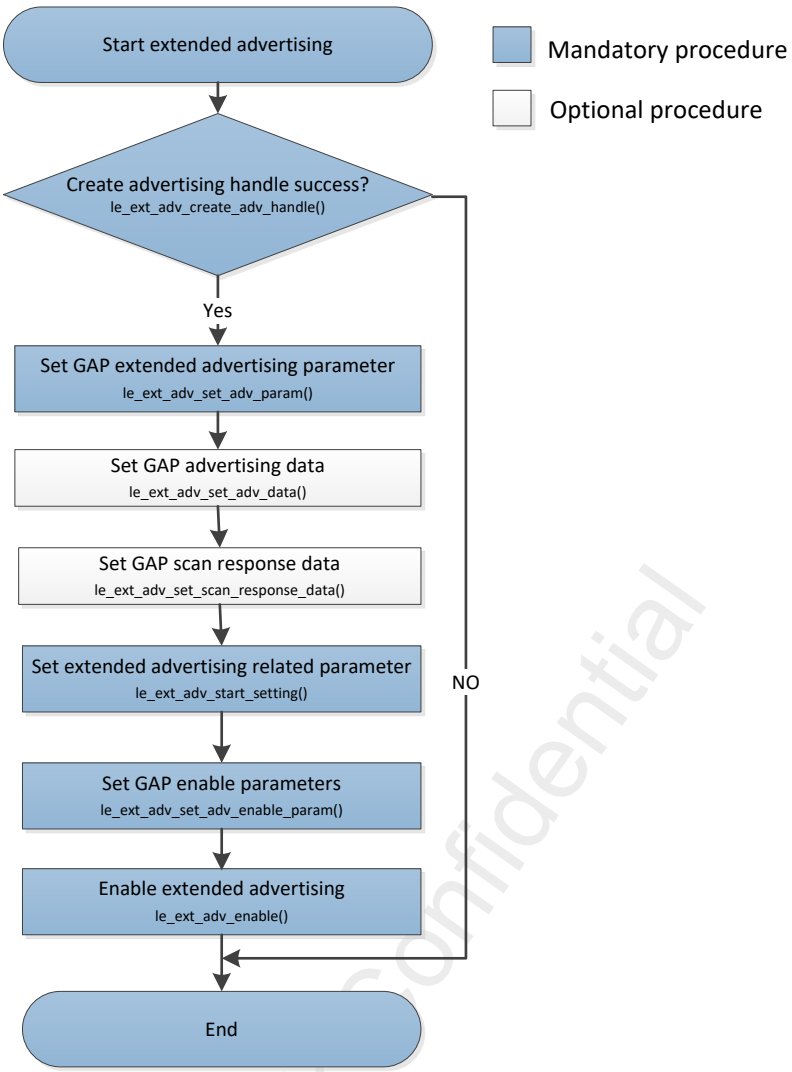

Figure 4-4 Start Extended Advertising Flow Chart

4.6.2.1 Initialization

main() function is invoked when the application is powered on or the chip resets and performs the following initialization functions:

```
int main(void)
{
    board_init();
    le_gap_init(APP_MAX_LINKS);
    gap_lib_init();
    app_le_gap_init();
    pwr_mgr_init();
    task_init();
    os_sched_start();
```

```
    return 0;
}
```

GAP initialization flow:

- le_gap_init() - Initialize GAP and set link number
- gap_lib_init() - Initialize GAP lib..
- app_le_gap_init() - GAP Parameter Initialization, the user can easily customize the application by modifying the following parameter values.
  - *Configure Device Name and Device Appearance*
  - *Configure Bond Manager Parameters*
  - *Configure LE Advertising Extensions Paramters*
  - *Configure Other Parameters*

More information on LE GAP Initialization and Startup Flow can be found in chapter *GAP Initialization and Startup Flow*.

If BT5 peripheral application needs to support GATT Profiles, please refer to *Profile Message Callback* in *BLE Peripheral Application*.

4.6.2.1.1 Configure GAP Extended Advertising Related Parameters

BT5 peripheral application calls le_init_ext_adv_params_ext_conn() to set GAP parameters of Connectable Undirected Advertising with extended advertising PDUs. First, BT5 peripheral application creates an advertising handle by invoking le_ext_adv_create_adv_handle() to identify advertising set. Then, BT5 peripheral application sets GAP extended advertising parameters and advertising data for the advertising set. Sample codes are listed as below:

```
void app_le_gap_init(void)
{
    ……
    /* Initialize extended advertising related parameters */
    le_init_ext_adv_params_ext_conn();
    ……
}

void le_init_ext_adv_params_ext_conn(void)
{
    T_LE_EXT_ADV_EXTENDED_ADV_PROPERTY adv_event_prop =
LE_EXT_ADV_EXTENDED_ADV_CONN_UNDIRECTED;
    uint32_t primary_adv_interval_min = DEFAULT_ADVERTISING_INTERVAL_MIN;
    uint32_t primary_adv_interval_max = DEFAULT_ADVERTISING_INTERVAL_MAX;
    uint8_t    primary_adv_channel_map = GAP_ADVCHAN_ALL;
    T_GAP_LOCAL_ADDR_TYPE own_address_type = GAP_LOCAL_ADDR_LE_PUBLIC;
    T_GAP_REMOTE_ADDR_TYPE peer_address_type = GAP_REMOTE_ADDR_LE_PUBLIC;
    uint8_t p_peer_address[6] = {0};
```

```
    T_GAP_ADV_FILTER_POLICY filter_policy = GAP_ADV_FILTER_ANY;
    uint8_t tx_power = 0;
    T_GAP_PHYS_PRIM_ADV_TYPE primary_adv_phy;
    uint8_t secondary_adv_max_skip = 0;
    T_GAP_PHYS_TYPE secondary_adv_phy;
    uint8_t adv_sid = 0;
    bool scan_req_notification_enable = false;

    /* Initialize primary advertisement PHY and secondary advertisement PHY */
    if (ADVERTISING_PHY == APP_PRIMARY_1M_SECONDARY_2M)
    {
        primary_adv_phy = GAP_PHYS_PRIM_ADV_1M;
        secondary_adv_phy = GAP_PHYS_2M;
    }
    else if (ADVERTISING_PHY == APP_PRIMARY_CODED_SECONDARY_CODED)
    {
        primary_adv_phy = GAP_PHYS_PRIM_ADV_CODED;
        secondary_adv_phy = GAP_PHYS_CODED;
    }

    /* Initialize extended advertising parameters */
    adv_handle = le_ext_adv_create_adv_handle();
    if(adv_handle == APP_IDLE_ADV_SET)
    {
        return;
    }
    if (adv_set_num < APP_MAX_ADV_SET)
    {
        ext_adv_state[adv_set_num++].adv_handle = adv_handle;
    }
    le_ext_adv_set_adv_param(adv_handle, adv_event_prop, primary_adv_interval_min, primary_adv_interval_max,
                             primary_adv_channel_map, own_address_type, peer_address_type, p_peer_address,
                             filter_policy, tx_power, primary_adv_phy, secondary_adv_max_skip,
                             secondary_adv_phy, adv_sid, scan_req_notification_enable);

    /* Initialize extended advertising data(max size = 245 bytes)*/
    le_ext_adv_set_adv_data(adv_handle, sizeof(ext_adv_data), (uint8_t *)ext_adv_data);
}
```

APIs about extended advertising are provided in gap_ext_adv.h, more information please refers to gap_ext_adv.h.

The adv_event_prop defines property of advertising, and different properties of advertising needs different parameters. The advertising event properties are divided into two categories according to advertising PDUs, which are advertising event properties with legacy advertising PDUs and advertising event properties with extended advertising PDUs, respectively.

The advertising event properties with legacy advertising PDUs are listed in Table 4-2. If legacy advertising PDU types are used, the amount of data shall not exceed 31 octets.

Table 4-2 Extended Advertising Parameters Setting with legacy advertising PDUs

adv_event_prop	LE_EXT_ADV_LEGACY_ADV_CONN_SCAN_UNDIRECTED	LE_EXT_ADV_LEGACY_ADV_CONN_HIGH_DUTY_DIRECTED	LE_EXT_ADV_LEGACY_ADV_SCAN_UNDIRECTED	LE_EXT_ADV_LEGACY_ADV_NON_SCAN_NON_CONN_UNDIRECTED	LE_EXT_ADV_LEGACY_ADV_CONN_LOW_DUTY_DIRECTED
adv_handle	Y	Y	Y	Y	Y
primary_adv_interval_min	Y	Ignore	Y	Y	Y
primary_adv_interval_max	Y	Ignore	Y	Y	Y
primary_adv_channel_map	Y	Y	Y	Y	Y
peer_address_type	Ignore	Y	Ignore	Ignore	Y
p_peer_address	Ignore	Y	Ignore	Ignore	Y
filter_policy	Y	Ignore	Y	Y	Ignore
primary_adv_phy	LE 1M PHY	LE 1M PHY	LE 1M PHY	LE 1M PHY	LE 1M PHY
secondary_adv_phy	Ignore	Ignore	Ignore	Ignore	Ignore
allow establish link	Y	Y	N	N	Y
Advertising data	Y	N	Y	Y	N
Scan response data	Y	N	Y	N	N

The advertising event properties with extended advertising PDUs are listed in Table 4-3. The sample codes about extended advertising related parameters setting with extended advertising PDUs are provided in bt5_peripheral_stack_api.h and bt5_peripheral_stack_api.c. If only one advertising set is used, the maximum length of data is listed in Table 4-3.

Table 4-3 Extended Advertising Parameters Setting with extended advertising PDUs

adv_event_prop	LE_EXT_ADV_EXTENDED_ADV_NON_SCAN_NON_CONN_UNDIRECTED	LE_EXT_ADV_EXTENDED_ADV_NON_SCAN_NON_CONN_DIRECTED	LE_EXT_ADV_EXTENDED_ADV_CONN_UNDIRECTED	LE_EXT_ADV_EXTENDED_ADV_CONN_DIRECTED	LE_EXT_ADV_EXTENDED_ADV_SCAN_UNDIRECTED	LE_EXT_ADV_EXTENDED_ADV_SCAN_DIRECTED
adv_handle	Y	Y	Y	Y	Y	Y
primary_adv_interval_min	Y	Y	Y	Y	Y	Y
primary_adv_interval_max	Y	Y	Y	Y	Y	Y
primary_adv_channel_map	Y	Y	Y	Y	Y	Y
peer_address_type	Ignore	Y	Ignore	Y	Ignore	Y
p_peer_address	Ignore	Y	Ignore	Y	Ignore	Y
filter_policy	Y	Ignore	Y	Ignore	Y	Ignore
primary_adv_phy	LE 1M PHY LE Coded PHY	LE 1M PHY LE Coded PHY	LE 1M PHY LE Coded PHY	LE 1M PHY LE Coded PHY	LE 1M PHY LE Coded PHY	LE 1M PHY LE Coded PHY
secondary_adv_phy	LE 1M PHY LE 2M PHY LE Coded PHY	LE 1M PHY LE 2M PHY LE Coded PHY	LE 1M PHY LE 2M PHY LE Coded PHY	LE 1M PHY LE 2M PHY LE Coded PHY	LE 1M PHY LE 2M PHY LE Coded PHY	LE 1M PHY LE 2M PHY LE Coded PHY
allow establish link	N	N	Y	Y	N	N
Advertising data	Y 1024 bytes	Y 1024 bytes	Y 245 bytes	Y 239 bytes	N	N
Scan response data	N	N	N	N	Y 991 bytes	Y 991 bytes

4.6.2.2 GAP Message Handler

app_handle_gap_msg() function is invoked whenever a GAP messages is received from the GAP. More information on GAP messages can be found in chapter *Bluetooth LE GAP Message*.

BT5 peripheral application will call le_ext_adv_start_setting() to set extended advertising related parameters when receives GAP_INIT_STATE_STACK_READY message. EXT_ADV_SET_AUTO means that extended advertising related parameters (including advertising parameters, advertising data and scan response data) for the specified advertising set will be set automatically according to advertising event properties. Otherwise, application could set flags according to advertising event properties with Table 4-2 or Table 4-3.

```
void app_handle_dev_state_evt(T_GAP_DEV_STATE new_state, uint16_t cause)
{
    APP_PRINT_INFO3("app_handle_dev_state_evt: init state %d, adv state %d, cause 0x%x",
                    new_state.gap_init_state, new_state.gap_adv_state, cause);
    if (gap_dev_state.gap_init_state != new_state.gap_init_state)
    {
        if (new_state.gap_init_state == GAP_INIT_STATE_STACK_READY)
        {
            APP_PRINT_INFO0("GAP stack ready");
            /* Stack ready, set extended advertising related parameters */
            le_ext_adv_start_setting(adv_handle, EXT_ADV_SET_AUTO);
        }
    }

    gap_dev_state = new_state;
}
```

BT5 peripheral application will receive extended advertising state message after le_ext_adv_enable() is invoked in app_gap_callback(). More information about handling extended advertising state message can be found in *Extended Advertising State Message*.

BT5 peripheral application will call le_ext_adv_enable() to enable specified advertising set when receives GAP_CONN_STATE_DISCONNECTED. After disconnection, bt5 peripheral application will be restored to the status that is discoverable and connectable again.

```
void app_handle_conn_state_evt(uint8_t conn_id, T_GAP_CONN_STATE new_state, uint16_t disc_cause)
{
    APP_PRINT_INFO4("app_handle_conn_state_evt: conn_id %d old_state %d new_state %d, disc_cause 0x%x",
                    conn_id, gap_conn_state, new_state, disc_cause);
    switch (new_state)
    {
    case GAP_CONN_STATE_DISCONNECTED:
        {
            if ((disc_cause != (HCI_ERR | HCI_ERR_REMOTE_USER_TERMINATE))
```

```
                    && (disc_cause != (HCI_ERR | HCI_ERR_LOCAL_HOST_TERMINATE)))
                {
                    APP_PRINT_ERROR1("app_handle_conn_state_evt: connection lost cause 0x%x", disc_cause);
                }
                /* Enable one advertising set */
                le_ext_adv_enable(1, &adv_handle);
            }
            break;
        ……
        }
        gap_conn_state = new_state;
}
```

4.6.2.3 GAP Callback Handler

app_gap_callback() function is used to handle GAP callback messages. More information on GAP callback can be found in chapter *Bluetooth LE GAP Callback*.

le_ext_adv_start_setting() is invoked after stack is ready, then result of setting extended advertising parameters will be returned by app_gap_callback() with cb_type GAP_MSG_LE_EXT_ADV_START_SETTING. If the result is success, application will enable extended advertising.

First, application sets GAP extended advertising enable parameters which determine whether to stop extended advertising when duration has expired or maximum number of extended advertising events has reached. Then, application calls le_ext_adv_enable() to enable extended advertising for one advertising set. Sample codes are listed as below:

```
T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    T_LE_CB_DATA *p_data = (T_LE_CB_DATA *)p_cb_data;

    switch (cb_type)
    {
    case GAP_MSG_LE_EXT_ADV_START_SETTING:
        APP_PRINT_INFO3("GAP_MSG_LE_EXT_ADV_START_SETTING:cause 0x%x, flag 0x%x,
        adv_handle %d", p_data->p_le_ext_adv_start_setting_rsp->cause,
        p_data->p_le_ext_adv_start_setting_rsp->flag, p_data->p_le_ext_adv_start_setting_rsp->adv_handle);

        if (p_data->p_le_ext_adv_start_setting_rsp->cause == GAP_CAUSE_SUCCESS)
        {
            /* Initialize enable parameters */
            le_init_ext_adv_enable_params(p_data->p_le_ext_adv_start_setting_rsp->adv_handle);
            /* Enable one advertising set */
```

```
                le_ext_adv_enable(1, &p_data->p_le_ext_adv_start_setting_rsp->adv_handle);
            }
            break;
            ……
}
```

Extended advertising state message will be handled in app_handle_gap_msg() function. The result of enabling advertising set will be returned by app_gap_callback() with cb_type GAP_MSG_LE_EXT_ADV_ENABLE. Sample codes are listed as below:

```
T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    T_LE_CB_DATA *p_data = (T_LE_CB_DATA *)p_cb_data;

    switch (cb_type)
    {
    case GAP_MSG_LE_EXT_ADV_ENABLE:
        APP_PRINT_INFO1("GAP_MSG_LE_EXT_ADV_ENABLE:cause 0x%x", p_data->le_cause.cause);
        break;
    ……
}
```

When BLE BT5 Peripheral Application is being run on Evolution Board, the Bluetooth LE device becomes discoverable and connectable. Remote device can scan the peripheral device and create connection with peripheral device.

4.6.3 APP Configurable Functions

APP Configurable Functions are defined in app_flags.h.

```
/** @brief Configure Advertising PHY */
#define ADVERTISING_PHY                 APP_PRIMARY_1M_SECONDARY_2M
/** @brief    Config APP LE link number */
#define APP_MAX_LINKS    1
/** @brief    Config DLPS: 0-Disable DLPS, 1-Enable DLPS */
#define F_BT_DLPS_EN                                1
```

4.6.3.1 DLPS

More information about DLPS refers to DLPS related document [错误!未找到引用源.].

4.6.4 Test Procedure

Firstly, build and download the BLE BT5 peripheral application to the Evolution Board. Some basic functions of BLE BT5 peripheral application are demonstrated above. To implement some complex functions, user needs to refer to the manuals and source codes provided by SDK for development.

When BLE BT5 peripheral application is being run on Evolution Board, the Bluetooth LE device becomes discoverable and connectable. Remote device can scan the peripheral device and create connection with peripheral device. After disconnection, BLE BT5 peripheral application device will be restored to the status that is discoverable and connectable again.

4.6.4.1 Test with BLE BT5 Central Application Device

BLE BT5 peripheral application can test with BLE BT5 central application. More information on BT5 central application can be found in chapter *BLE BT5 Central Application*.
The test procedure please refers to chapter *Test with BLE BT5 Peripheral Application Device*.

4.7 BLE BT5 Central Application

4.7.1 Introduction

The purpose of this chapter is to give an overview of the BLE BT5 central application. The BLE BT5 central project implements a very simple BLE BT5 central device with LE Advertising Extensions and can be used as a framework for further development of central-role or observer-role based applications.

Central role features:

- Extended scan for advertising packets on the primary advertising channel(LE 1M PHY or/and LE Coded PHY
- Scan for a duration; period scan
- Can initiate connection and become a master role in the link on LE 1M PHY
  LE 2M PHY or LE Coded PHY (peripheral support LE Advertising Extensions)

Expose features:

- Link number:
  APP_MAX_LINKS in app_flags.h. (Default one link)

Compatibility:

Central device with LE Advertising Extensions can communicate with device using legacy advertising PDUs or extended advertising PDUs, and set duration of scan.

If central device uses LE Advertising Extensions, compatibility with peer device of different BT versions is shown in Table 4-4.

Table 4-4 Compatibility of central device with LE Advertising Extensions

Bluetooth 5 Feature	Bluetooth 4.0	Bluetooth 4.1	Bluetooth 4.2	Bluetooth 5.0 (not use LE Advertising Extensions)	Bluetooth 5.0 (use LE Advertising Extensions)
LE Advertising Extensions	Y	Y	Y	Y	Y

4.7.2 Source Code Overview

The following sections describe important parts of this application.

4.7.2.1 Initialization

main() is invoked when the application is powered on or the chip resets and performs the following initialization functions:

```
int main(void)
{
    board_init();
    le_gap_init(APP_MAX_LINKS);
    gap_lib_init();
    app_le_gap_init();
    pwr_mgr_init();
    task_init();
    os_sched_start();

    return 0;
}
```

GAP initialization flow:

- le_gap_init() - Initialize GAP and set link number.
- gap_lib_init() - Initialize GAP lib.

- app_le_gap_init() - GAP Parameter Initialization, the user can easily customize the application by modifying the following parameter values.
  - *Configure Device Name and Device Appearance*
  - *Configure Bond Manager Parameters*
  - *Configure LE Advertising Extensions Paramters*

More information on LE GAP Initialization and Startup Flow can be found in chapter *GAP Initialization and Startup Flow*.

If BT5 central application needs to support GATT Profiles, please refer to *Profile Message Callback* and *Discover GATT Services Procedure* in *BLE Central Application*.

If BT5 central application needs to support multilink, please refer to *Multilink Manager* in *BLE Central Application*.

4.7.2.2 GAP Message Handler

app_handle_gap_msg() function is invoked whenever a GAP messages is received from the GAP. More information on GAP messages can be found in chapter *Bluetooth LE GAP Message*.

When receives message GAP_MSG_LE_CONN_STATE_CHANGE, app will update link state.

```
void app_handle_conn_state_evt(uint8_t conn_id, T_GAP_CONN_STATE new_state, uint16_t disc_cause)
{
    APP_PRINT_INFO4("app_handle_conn_state_evt: conn_id %d, conn_state(%d -> %d), disc_cause 0x%x",
                    conn_id, gap_conn_state, new_state, disc_cause);

    switch (new_state)
    {
    case GAP_CONN_STATE_DISCONNECTED:
        {
            if ((disc_cause != (HCI_ERR | HCI_ERR_REMOTE_USER_TERMINATE))
                && (disc_cause != (HCI_ERR | HCI_ERR_LOCAL_HOST_TERMINATE)))
            {
                APP_PRINT_ERROR1("app_handle_conn_state_evt: connection lost cause 0x%x", disc_cause);
            }
        }
        data_uart_print("Disconnect conn_id %d\r\n", conn_id);
        break;

    case GAP_CONN_STATE_CONNECTED:
        {
            uint8_t tx_phy;
            uint8_t rx_phy;
            uint16_t conn_interval;
```

```
                uint16_t conn_latency;
                uint16_t conn_supervision_timeout;
                uint8_t    remote_bd[6];
                T_GAP_REMOTE_ADDR_TYPE remote_bd_type;

                le_get_conn_param(GAP_PARAM_CONN_INTERVAL, &conn_interval, conn_id);
                le_get_conn_param(GAP_PARAM_CONN_LATENCY, &conn_latency, conn_id);
                le_get_conn_param(GAP_PARAM_CONN_TIMEOUT, &conn_supervision_timeout, conn_id);
                le_get_conn_addr(conn_id, remote_bd, (uint8_t *)&remote_bd_type);
                APP_PRINT_INFO5("GAP_CONN_STATE_CONNECTED:remote_bd %s, remote_addr_type %d,
conn_interval 0x%x, conn_latency 0x%x, conn_supervision_timeout 0x%x",
                                TRACE_BDADDR(remote_bd), remote_bd_type,
                                conn_interval, conn_latency, conn_supervision_timeout);

                le_get_conn_param(GAP_PARAM_CONN_TX_PHY_TYPE, &tx_phy, conn_id);
                le_get_conn_param(GAP_PARAM_CONN_RX_PHY_TYPE, &rx_phy, conn_id);
                data_uart_print("Connected success conn_id %d, tx_phy %d, rx_phy %d\r\n", conn_id, tx_phy, rx_phy);
            }
            break;

        default:
            break;
        }
        gap_conn_state = new_state;
}
```

4.7.2.3 GAP Callback Handler

app_gap_callback() function is used to handle GAP callback messages. More information on GAP callback can be found in chapter *Bluetooth LE GAP Callback*.

If device start extended scan and receives advertising data or scan response data, GAP Layer will use GAP_MSG_LE_EXT_ADV_REPORT_INFO message to inform application. Sample codes are listed as below:

```
T_APP_RESULT app_gap_callback(uint8_t cb_type, void *p_cb_data)
{
    T_APP_RESULT result = APP_RESULT_SUCCESS;
    T_LE_CB_DATA *p_data = (T_LE_CB_DATA *)p_cb_data;

    APP_PRINT_TRACE1("app_gap_callback: cb_type = %d", cb_type);

    switch (cb_type)
    {
    case GAP_MSG_LE_EXT_ADV_REPORT_INFO:
```

```
            APP_PRINT_INFO6("GAP_MSG_LE_EXT_ADV_REPORT_INFO:connectable %d, scannable %d,
direct %d, scan response %d, legacy %d, data status 0x%x",
            p_data->p_le_ext_adv_report_info->event_type & GAP_EXT_ADV_REPORT_BIT_CONNECTABLE_ADV,
            p_data->p_le_ext_adv_report_info->event_type & GAP_EXT_ADV_REPORT_BIT_SCANNABLE_ADV,
            p_data->p_le_ext_adv_report_info->event_type & GAP_EXT_ADV_REPORT_BIT_DIRECTED_ADV,
            p_data->p_le_ext_adv_report_info->event_type & GAP_EXT_ADV_REPORT_BIT_SCAN_RESPONSE,
            p_data->p_le_ext_adv_report_info->event_type & GAP_EXT_ADV_REPORT_BIT_USE_LEGACY_ADV,
            p_data->p_le_ext_adv_report_info->data_status);
            APP_PRINT_INFO5("GAP_MSG_LE_EXT_ADV_REPORT_INFO:event_type 0x%x, bd_addr %s,
addr_type %d, rssi %d, data_len %d",
            p_data->p_le_ext_adv_report_info->event_type,
            TRACE_BDADDR(p_data->p_le_ext_adv_report_info->bd_addr),
            p_data->p_le_ext_adv_report_info->addr_type,
            p_data->p_le_ext_adv_report_info->rssi,
            p_data->p_le_ext_adv_report_info->data_len);
            APP_PRINT_INFO5("GAP_MSG_LE_EXT_ADV_REPORT_INFO:primary_phy %d, secondary_phy %d,
adv_sid %d, tx_power %d, peri_adv_interval %d",
            p_data->p_le_ext_adv_report_info->primary_phy,
            p_data->p_le_ext_adv_report_info->secondary_phy,
            p_data->p_le_ext_adv_report_info->adv_sid,
            p_data->p_le_ext_adv_report_info->tx_power,
            p_data->p_le_ext_adv_report_info->peri_adv_interval);
            APP_PRINT_INFO2("GAP_MSG_LE_EXT_ADV_REPORT_INFO:direct_addr_type 0x%x,
direct_addr %s",
            p_data->p_le_ext_adv_report_info->direct_addr_type,
            TRACE_BDADDR(p_data->p_le_ext_adv_report_info->direct_addr));

            link_mgr_add_device(p_data->p_le_ext_adv_report_info->bd_addr,
                                p_data->p_le_ext_adv_report_info->addr_type);

#if APP_RECOMBINE_ADV_DATA
            if (!(p_data->p_le_ext_adv_report_info->event_type &
GAP_EXT_ADV_REPORT_BIT_USE_LEGACY_ADV))
            {
                /* If the advertisement uses extended advertising PDUs, recombine advertising data. */
                app_handle_ext_adv_report(p_data->p_le_ext_adv_report_info->event_type,
                        p_data->p_le_ext_adv_report_info->data_status,    p_data->p_le_ext_adv_report_info->bd_addr,
                         p_data->p_le_ext_adv_report_info->data_len, p_data->p_le_ext_adv_report_info->p_data);
            }
#endif
            break;
            ……
}
```

4.7.3 APP Configurable Functions

APP Configurable Functions are defined in app_flags.h.

```
/** @brief    Config APP LE link number */
#define APP_MAX_LINKS    1
/** @brief Configure APP to recombine advertising data: 0-Disable recombine advertising data feature, 1- recombine advertising data */
#define APP_RECOMBINE_ADV_DATA    0
```

The user can easily change the value of macro definition to on-off the function or copy the referenced codes to other application.

4.7.3.1 Recombine advertising data

Recombination of advertising data implements a recombination for one advertising data or scan response data, and recombined advertising data is deleted before next recombination. The process of recombination can be used as a framework for recombining multi-advertising data at a time.

Memory for recombining advertising data is allocated in GAP Initialization, and sample codes are listed as below.

```
void app_le_gap_init(void)
{
    ……
    /* Allocate memory for recombining advertising data */
#if APP_RECOMBINE_ADV_DATA
    ext_adv_data = os_mem_zalloc(RAM_TYPE_DATA_ON, sizeof(T_EXT_ADV_DATA));
    ext_adv_data->flag = false;
#endif
    ……
}
```

Message data structure is T_EXT_ADV_DATA.

```
typedef struct
{
    uint8_t     bd_addr[GAP_BD_ADDR_LEN];          /**< remote BD */
    bool        flag;                              /**< flag of recombining advertising data, true: recombining,
                                                        false: waiting extended advertising PDUs */
    uint16_t    event_type;                        /**< advertising event type */
    uint16_t    data_len;                          /**< length of recombined advertising data */
    uint8_t     p_data[APP_MAX_EXT_ADV_TOTAL_LEN]; /**< recombined advertising data */
} T_EXT_ADV_DATA;
```

BT5 central application handles GAP_MSG_LE_EXT_ADV_REPORT_INFO message which indicates the receipt of advertising data or scan response data in app_gap_callback() function, please refer to *GAP Callback*

Handler. If the extended advertising report indicates extended advertising PDU is used, application calls app_handle_ext_adv_report() function to recombine advertising data. The following sections describe the recombination of advertising data, please refer to app_handle_ext_adv_report() function in bt5_central_app.c.

4.7.3.1.1 GAP_EXT_ADV_EVT_DATA_STATUS_COMPLETE

GAP_EXT_ADV_EVT_DATA_STATUS_COMPLETE indicates that the data is complete. The process of advertising report with data status GAP_EXT_ADV_EVT_DATA_STATUS_COMPLETE is shown in Figure 4-5.

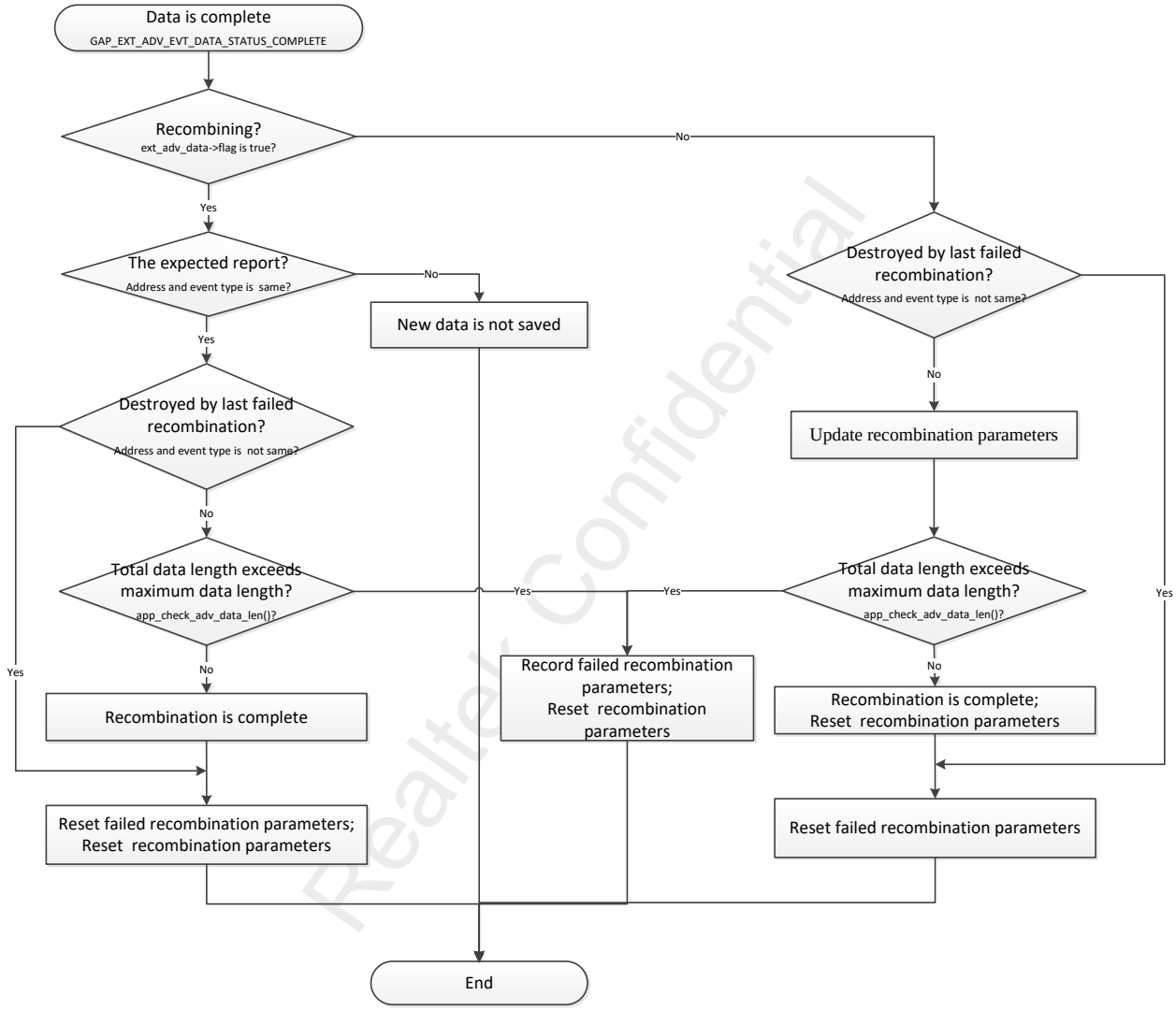

Figure 4-5 Recombination Flow Chart (GAP_EXT_ADV_EVT_DATA_STATUS_COMPLETE)

4.7.3.1.2 GAP_EXT_ADV_EVT_DATA_STATUS_MORE

GAP_EXT_ADV_EVT_DATA_STATUS_MORE indicates that the data is incomplete and more data will be received. The process of advertising report with data status GAP_EXT_ADV_EVT_DATA_STATUS_MORE is shown in Figure 4-6.

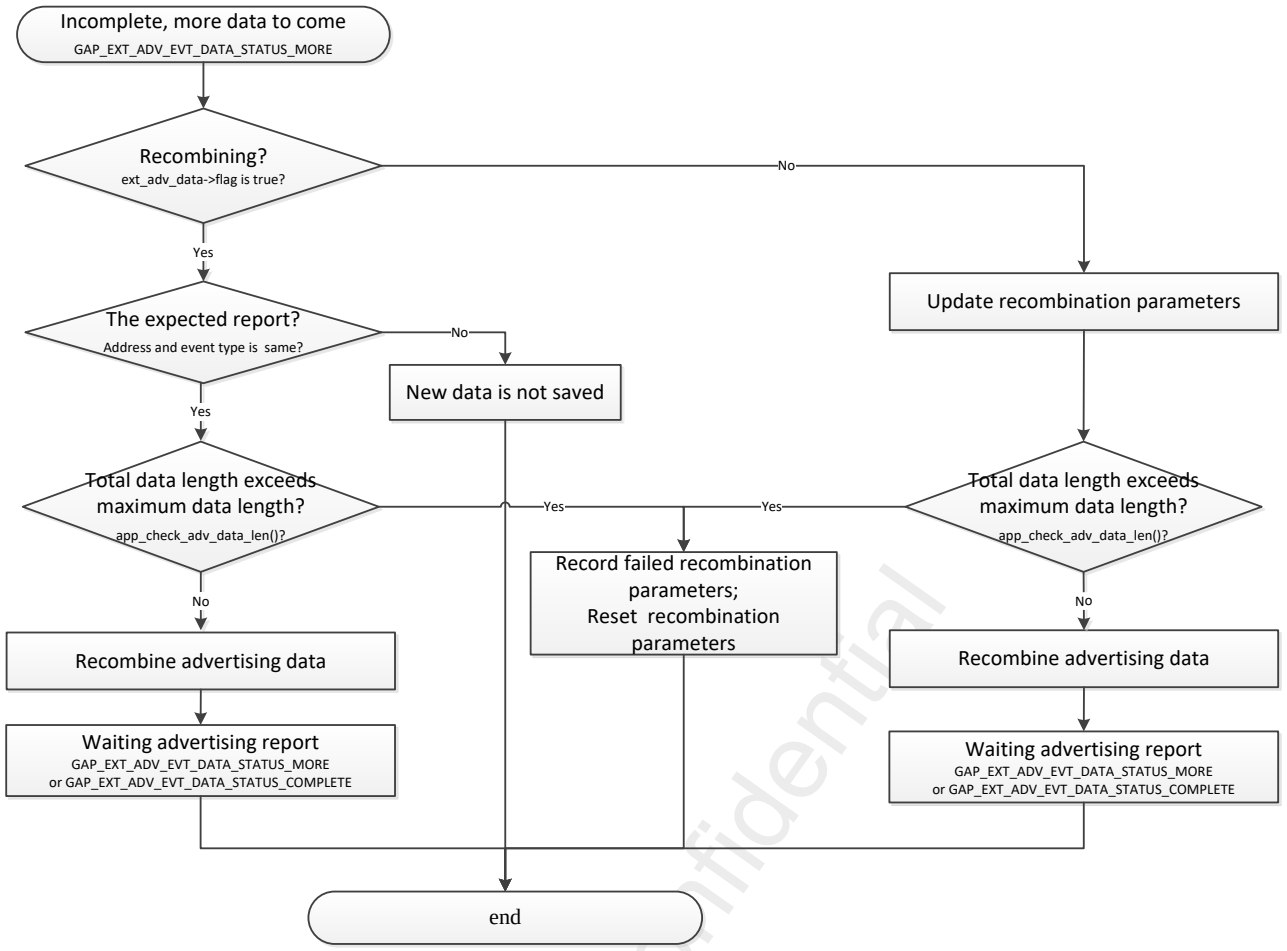

Figure 4-6 Recombination Flow Chart (GAP_EXT_ADV_EVT_DATA_STATUS_MORE)

After process advertising report with GAP_EXT_ADV_EVT_DATA_STATUS_MORE, application waits for advertising report to complete the recombination. The last advertising report with GAP_EXT_ADV_EVT_DATA_STATUS_COMPLETE indicates the recombination process is completed.

4.7.3.1.3 GAP_EXT_ADV_EVT_DATA_STATUS_TRUNCATED

GAP_EXT_ADV_EVT_DATA_STATUS_TRUNCATED indicates that the data is incomplete, data truncated, and no more data to come. If advertising report with truncated data is the expected report, application will terminate the current recombination by resetting recombination parameters and wait advertising report to start next recombination.

4.7.4 User Command

BLE BT5 Central Application needs to perform interaction by inputting command through Data UART in PC. More information on user command can be found in chapter *BLE Application User Command*. The following sections describe some commands used in BT5 central application.

4.7.4.1 Commands about Extended Scanning

Commands about extended scanning include escan and stopescan.

Command escan is used to start extended scanning. First, application calls le_ext_scan_set_param() to initialize extended scan parameters and le_ext_scan_set_phy_param() to initialize extended scan PHY parameters. Then application calls le_ext_scan_start() to enable extended scanning. Implementation of command escan is shown as below.

```
static T_USER_CMD_PARSE_RESULT cmd_escan(T_USER_CMD_PARSED_VALUE *p_parse_value)
{
    ……
    /* Initialize extended scan parameters */
    le_ext_scan_set_param(GAP_PARAM_EXT_SCAN_LOCAL_ADDR_TYPE, sizeof(own_address_type),
                          &own_address_type);
    le_ext_scan_set_param(GAP_PARAM_EXT_SCAN_PHYS, sizeof(scan_phys),
                          &scan_phys);
    le_ext_scan_set_param(GAP_PARAM_EXT_SCAN_DURATION, sizeof(ext_scan_duration),
                          &ext_scan_duration);
    le_ext_scan_set_param(GAP_PARAM_EXT_SCAN_PERIOD, sizeof(ext_scan_period),
                          &ext_scan_period);
    le_ext_scan_set_param(GAP_PARAM_EXT_SCAN_FILTER_POLICY, sizeof(ext_scan_filter_policy),
                          &ext_scan_filter_policy);
    le_ext_scan_set_param(GAP_PARAM_EXT_SCAN_FILTER_DUPLICATES, sizeof(ext_scan_filter_duplicate),
                          &ext_scan_filter_duplicate);

    /* Initialize extended scan PHY parameters */
    le_ext_scan_set_phy_param(LE_SCAN_PHY_LE_1M, &extended_scan_param[0]);
    le_ext_scan_set_phy_param(LE_SCAN_PHY_LE_CODED, &extended_scan_param[1]);

    /* Enable extended scan */
    cause = le_ext_scan_start();
    return (T_USER_CMD_PARSE_RESULT)cause;
}
```

Usage of APIs about extended scanning could be found in gap_ext_scan.h.

Different from ble central application, Duration parameter and Period parameter determine the scanning mode in BT5 central application. If Duration paramter is zero, device scans continuously until scanning is disabled. If Duration and Period parameters are non-zero, device scan for the duration within a scan period, and scan periods continue until scanning is disabled. If Duration parameter is non-zero and Period parameter is zero, device continues scanning until duration has expired.

Different from ble central application, bt5 central application receives advertising packets with legacy advertising PDUs and extended advertising PDUs on the primary advertising channel which could be LE 1M PHY or/and LE

Coded PHY.

Definition of command escan is shown as below.

```
const T_USER_CMD_TABLE_ENTRY user_cmd_table[] =
{
    ……
    {
        "escan",
        "escan [scan_mode] [scan_phys]\n\r",
        "Start extended scan\r\n\
        [scan_mode]: 0-(continue scanning until scanning is disabled)\r\n\
                    1-(scan for the duration within a scan period, and scan periods continue until scanning is disabled)\r\n\
                    2-(continue scanning until duration has expired)\r\n\
        [scan_phys]: set scan PHYs to 1(LE 1M PHY), 4(LE Coded PHY) or 5(LE 1M PHY and LE Coded PHY)\r\n\
        sample: escan 0 4\n\r",
        cmd_escan
    },
    ……
}
```

Command stopescan is used to stop extended scanning by calling le_ext_scan_stop(). Implementation and usage of command stopescan are shown as below.

```
static T_USER_CMD_PARSE_RESULT cmd_stopescan(T_USER_CMD_PARSED_VALUE *p_parse_value)
{
    T_GAP_CAUSE cause;
    cause = le_ext_scan_stop();
    return (T_USER_CMD_PARSE_RESULT)cause;
}

const T_USER_CMD_TABLE_ENTRY user_cmd_table[] =
{
    ……
    {
        "stopescan",
        "stopescan\n\r",
        "Stop extended scan\n\r",
        cmd_stopescan
    },
    ……
}
```

4.7.4.2 Commands about Extended Create Connection

Commands about extended create connection include condev and disc.

Using LE Advertising Extensions, application uses command condev to create connection. Initiating PHYs parameter indicates the PHY(s) on which the advertising packets should be received on the primary advertising channel and the PHYs for which connection parameters have been specified. Primary advertising channel includes LE 1M PHY and LE Coded PHY, so Initiating PHYs parameter shall have one bit set for a PHY allowed for scanning on the primary advertising channel. If peripheral uses extended advertising PDUs and secondary advertising PHY is LE 1M PHY, LE 2M PHY or LE Coded PHY, application can become a master role in the link on LE 1M PHY, LE 2M PHY or LE Coded PHY.

Implementation and usage of command condev are shown as below.

```
static T_USER_CMD_PARSE_RESULT cmd_condev(T_USER_CMD_PARSED_VALUE *p_parse_value)
{
    uint8_t dev_idx = p_parse_value->dw_param[0];
    if (dev_idx < dev_list_count)
    {
        T_GAP_CAUSE cause;
        T_GAP_LE_CONN_REQ_PARAM conn_req_param;
        T_GAP_LOCAL_ADDR_TYPE local_addr_type = GAP_LOCAL_ADDR_LE_PUBLIC;
        uint8_t    init_phys = 0;
        conn_req_param.scan_interval = 0x10;
        conn_req_param.scan_window = 0x10;
        conn_req_param.conn_interval_min = 80;
        conn_req_param.conn_interval_max = 80;
        conn_req_param.conn_latency = 0;
        conn_req_param.supv_tout = 1000;
        conn_req_param.ce_len_min = 2 * (conn_req_param.conn_interval_min - 1);
        conn_req_param.ce_len_max = 2 * (conn_req_param.conn_interval_max - 1);

        le_set_conn_param(GAP_CONN_PARAM_1M, &conn_req_param);
        le_set_conn_param(GAP_CONN_PARAM_2M, &conn_req_param);
        le_set_conn_param(GAP_CONN_PARAM_CODED, &conn_req_param);

        uint32_t input_phys = p_parse_value->dw_param[1];
        switch (input_phys)
        {
        case 0x001:
            init_phys = GAP_PHYS_CONN_INIT_1M_BIT;
            break;
        case 0x011:
            init_phys = GAP_PHYS_CONN_INIT_2M_BIT | GAP_PHYS_CONN_INIT_1M_BIT;
```

```
            break;
        case 0x100:
            init_phys = GAP_PHYS_CONN_INIT_CODED_BIT;
            break;
        case 0x101:
            init_phys = GAP_PHYS_CONN_INIT_CODED_BIT | GAP_PHYS_CONN_INIT_1M_BIT;
            break;
        case 0x110:
            init_phys = GAP_PHYS_CONN_INIT_CODED_BIT | GAP_PHYS_CONN_INIT_2M_BIT;
            break;
        case 0x111:
            init_phys = GAP_PHYS_CONN_INIT_CODED_BIT |
                        GAP_PHYS_CONN_INIT_2M_BIT |
                        GAP_PHYS_CONN_INIT_1M_BIT;
            break;

        default:
            break;
        }

        cause = le_connect(init_phys, dev_list[dev_idx].bd_addr,
                           (T_GAP_REMOTE_ADDR_TYPE)dev_list[dev_idx].bd_type,
                           local_addr_type,
                           1000);
    .......
}

const T_USER_CMD_TABLE_ENTRY user_cmd_table[] =
{
    ……
    {
        "condev",
        "condev [idx] [init_phys]\n\r",
        "Connect to remote device: use showdev to show idx\r\n\
        [idx]: use cmd showdev to show idx before use this cmd\r\n\
        [init_phys]: bit 0(LE 1M PHY) and bit 2(LE Coded PHY), at least one bit is set to one\r\n\
        sample: condev 0 0x100\n\r",
        cmd_condev
    },
    ……
};
```

Command disc is used to terminate an existing connection by calling le_disconnect(). Sample codes are listed as below.

```
static T_USER_CMD_PARSE_RESULT cmd_disc(T_USER_CMD_PARSED_VALUE *p_parse_value)
{
    uint8_t conn_id = p_parse_value->dw_param[0];
    T_GAP_CAUSE cause;
    cause = le_disconnect(conn_id);
    return (T_USER_CMD_PARSE_RESULT)cause;
}
```

4.7.4.3 Commands about Recombining Advertising Data

If macro definition APP_RECOMBINE_ADV_DATA is set to one, commands about recombining advertising data include sadvdata and radvdata. If user needs to develop function about recombining advertising data, please refer to the following section and *Recombine advertising data*.

Command sadvdata is used to disable recombination of advertising data until scanning is disabled. If scanning is disabled, recombination of advertising data is enabled for next scanning. Sample codes are listed as below.

```
static T_USER_CMD_PARSE_RESULT cmd_sadvdata(T_USER_CMD_PARSED_VALUE *p_parse_value)
{
    ext_adv_data->flag = true;
    ext_adv_data->data_len = 0;
    memset(ext_adv_data->bd_addr, 0, GAP_BD_ADDR_LEN);
    return (RESULT_SUCESS);
}

void app_handle_dev_state_evt(T_GAP_DEV_STATE new_state, uint16_t cause)
{
    ……
    if (gap_dev_state.gap_scan_state != new_state.gap_scan_state)
    {
        if (new_state.gap_scan_state == GAP_SCAN_STATE_IDLE)
        {
            ……
            /* Reset flags of recombining advertising data when stop scanning */
#if APP_RECOMBINE_ADV_DATA
            ext_adv_data->flag = false;
            ext_adv_data->data_len = 0;
            memset(fail_bd_addr, 0, GAP_BD_ADDR_LEN);
#endif
        }
        ……
    }

    gap_dev_state = new_state;
```

```
}
```

Command radvdata is used to start new recombination of advertising data. Sample codes are listed as below.

```
static T_USER_CMD_PARSE_RESULT cmd_radvdata(T_USER_CMD_PARSED_VALUE *p_parse_value)
{
    ext_adv_data->flag = false;
    ext_adv_data->data_len = 0;
    memset(fail_bd_addr, 0, GAP_BD_ADDR_LEN);
    return (RESULT_SUCESS);
}
```

4.7.5 Test Procedure

At first, please build and download the BLE BT5 Central application to the Evolution Board. Link number shall be configured by MPTool. Information about LE Link number please refers to *Configuration of LE Link number*. Some basic functions of BLE BT5 Central Application are demonstrated above. To implement some complex functions, user needs to refer to the manuals and source codes provided by SDK for development.

By using the commands as described in *User Command*, some processes of LE Advertising Extensions can be demonstrated.

Please use DebugAnalyser Tool to get the log.

4.7.5.1 Test with BLE BT5 Peripheral Application Device

BLE BT5 central application can test with BLE BT5 peripheral application. More information on peripheral can be found in chapter *BLE BT5 Peripheral Application*. Test Procedures are divided into two categories according to the configuration of BT5 peripheral application: Primary Advertising Channel is LE 1M PHY and Secondary Advertising Channel is LE 2M PHY; Primary Advertising Channel is LE Coded PHY and Secondary Advertising Channel is LE Coded PHY.

1. Primary Advertising Channel is LE 1M PHY, Secondary Advertising Channel is LE 2M PHY

   Firstly, set macro definition ADVERTISING_PHY to APP_PRIMARY_1M_SECONDARY_2M. Please build and download the BLE BT5 peripheral application to the Evolution Board. Bluetooth address shall be configured by MPTool, please set the address to "x00 x11 x22 x33 x44 x00".

   1) Extended Scanning

      Procedure Description: search for any discoverable Bluetooth LE device.

      Procedure:

      escan 2 5- Scan mode is set to 2, continue scanning until duration has expired. Scan PHYs is set to 5(LE 1M PHY and LE Coded PHY). Application starts extended scanning for duration, and examines

information on nearby Bluetooth LE device whose primary advertising channel is LE 1M PHY or LE Coded PHY. As shown below, extended scanning is disabled when duration has expired.

Log tool shows :

18-05-08#13:56:06.541 161 60234 [APP] !**GAP scan start

18-05-08#13:56:11.562 255 65256 [APP] !**GAP scan stop

The serial port assistant tool shows scan state as well.

Serial port assistant tool shows :

escan 2 5

GAP scan start

GAP scan stop

If BT5 central device uses escan to set scan mode to 0 or 1, BT5 central device may use *stopescan* to stop extended scanning.

Macro definition APP_RECOMBINE_ADV_DATA is set to one, the first advertising report from BT5 peripheral device is shown as below. The advertising report indicates that BT5 peripheral device sends Connectable Undirected Advertising with extended advertising PDUs, and the primary advertising PHY is LE 1M PHY, the secondary advertising PHY is LE 2M PHY. Due to more data to come, BT5 central device start the recombination and waiting for more data.

Log tool shows :

[APP] app_gap_callback: cb_type = 0x50

[APP] !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:connectable 1, scannable 0, direct 0, scan response 0, legacy 0, data status 0x1

[APP] !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:event_type 0x1, bd_addr 00::11::22::33::44::00, addr_type 0, rssi -45, data_len 229

[APP] !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:primary_phy 1, secondary_phy 2, adv_sid 0, tx_power 127, peri_adv_interval 0

[APP] !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:direct_addr_type 0x0, direct_addr 00::00::00::00::00::00

[APP] !**app_handle_ext_adv_report: Old ext_adv_data->flag is 0, data status is 0x1

[APP] !**app_handle_ext_adv_report:First Data from bd_addr 00::11::22::33::44::00, data length is 229, and waiting more data

[APP] !**app_handle_ext_adv_report: New ext_adv_data->flag is 1

The last advertising report from BT5 peripheral device is shown as below. The advertising report

indicates that the data is complete. BT5 central device completes the recombination, the length of advertising data received from BT5 peripheral device is 245 bytes.

Log tool shows :

[APP] app_gap_callback: cb_type = 0x50

[APP] !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:connectable 1, scannable 0, direct 0, scan response 0, legacy 0, data status 0x0

[APP] !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:event_type 0x1, bd_addr 00::11::22::33::44::00, addr_type 0, rssi -45, data_len 16

[APP] !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:primary_phy 1, secondary_phy 2, adv_sid 0, tx_power 127, peri_adv_interval 0

[APP] !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:direct_addr_type 0x0, direct_addr 00::00::00::00::00::00

[APP] !**app_handle_ext_adv_report: Old ext_adv_data->flag is 1, data status is 0x0

[APP] !**app_handle_ext_adv_report: Data from bd_addr 00::11::22::33::44::00 is complete, event type is 0x1, total data length is 245

[APP] !**app_handle_ext_adv_report: First five datas are 0x2, 0x1, 0x5, 0x13, 0x9

[APP] !**app_handle_ext_adv_report: Last five datas are 0xd5, 0xd6, 0xd7, 0xd8, 0xd9

[APP] !**app_handle_ext_adv_report: New ext_adv_data->flag is 0

showdev - Show scan device list.

Serial port assistant tool shows :

dev list

RemoteBd[0] = [97:23:89:45:67:10]

RemoteBd[1] = [00:11:22:33:44:04]

RemoteBd[2] = [00:11:22:33:44:00]

RemoteBd[3] = [00:11:22:33:44:02]

2) Connection and Reconnection

Procedure Description: Demonstrate connecting and disconnecting with the peripheral device.

Procedure:

condev 2 x001- Scan connectable advertisements on LE 1M PHY, and initiate connection. Secondary advertising channel is LE 2M PHY, and then TX PHY type and RX PHY type are LE 2M PHY.

Serial port assistant tool shows :

condev 2 x001

Connected success conn_id 0, tx_phy 2, rx_phy 2

showcon - Show connection information.

Serial port assistant tool shows :

showcon

ShowCon conn_id 0 state 0x00000002 role 1

RemoteBd = [00:11:22:33:44:00] type = 0

active link num 1, idle link num 0

disc 0 - Disconnect with the peripheral device.

Serial port assistant tool shows :

disc 0

Disconnect conn_id 0, dis_cause 0x00000116

condev 2 x001 - Reconnect with the peripheral device.

Serial port assistant tool shows :

condev 2 x001

Connected success conn_id 0, tx_phy 2, rx_phy 2

2. Primary Advertising Channel is LE Coded PHY, Secondary Advertising Channel is LE Coded PHY

   At first, set macro definition ADVERTISING_PHY to APP_PRIMARY_CODED_SECONDARY_CODED. Please build and download the BLE BT5 peripheral application to the Evolution Board. Bluetooth address shall be configured by MPTool, please set the address to "x00 x11 x22 x33 x44 x00".

   1) Extended Scanning

      Procedure Description: Demonstrate search for any Bluetooth LE device discoverable nearby.

      Procedure:

      escan 0 5- Scan mode is set to 0, continue scanning until scanning is disabled. Scan PHYs is set to 5(LE 1M PHY and LE Coded PHY). Start extended scanning and view information on Bluetooth LE device nearby whose primary advertising channel is LE 1M PHY or LE Coded PHY.

      Log tool shows :

      [APP] !**GAP scan start

      The serial port assistant tool shows scan state as well.

      Serial port assistant tool shows :

      escan 0 5

      GAP scan start

stopescan – Stop extended scanning.

Log tool shows :

[APP] !**GAP scan stop

The serial port assistant tool shows scan state as well.

Serial port assistant tool shows :

stopescan

GAP scan stop

Macro definition APP_RECOMBINE_ADV_DATA is set to one, the first advertising report from BT5 peripheral device is shown as below. The advertising report indicates that BT5 peripheral device sends Connectable Undirected Advertising with extended advertising PDUs, and the primary advertising PHY is LE Coded PHY, the secondary advertising PHY is LE Coded PHY. Due to more data to come, BT5 central device start the recombination and waiting for more data.

Log tool shows :

[APP] app_gap_callback: cb_type = 0x50

[APP] !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:connectable 1, scannable 0, direct 0, scan response 0, legacy 0, data status 0x1

[APP] !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:event_type 0x1, bd_addr 00::11::22::33::44::00, addr_type 0, rssi -41, data_len 229

[APP] !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:primary_phy 3, secondary_phy 3, adv_sid 0, tx_power 127, peri_adv_interval 0

[APP] !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:direct_addr_type 0x0, direct_addr 00::00::00::00::00::00

[APP] !**app_handle_ext_adv_report: Old ext_adv_data->flag is 0, data status is 0x1

[APP] !**app_handle_ext_adv_report:First Data from bd_addr 00::11::22::33::44::00, data length is 229, and waiting more data

[APP] !**app_handle_ext_adv_report: New ext_adv_data->flag is 1

The last advertising report from BT5 peripheral device is shown as below. The advertising report indicates that the data is complete. BT5 central device completes the recombination, the length of advertising data received from BT5 peripheral device is 245 bytes.

Log tool shows :

[APP] app_gap_callback: cb_type = 0x50

[APP] !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:connectable 1, scannable 0, direct 0, scan response 0, legacy 0, data status 0x0

[APP] !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:event_type 0x1, bd_addr 00::11::22::33::44::00, addr_type 0, rssi -41, data_len 16

[APP] !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:primary_phy 3, secondary_phy 3, adv_sid 0, tx_power 127, peri_adv_interval 0

[APP] !**GAP_MSG_LE_EXT_ADV_REPORT_INFO:direct_addr_type 0x0, direct_addr 00::00::00::00::00::00

[APP] !**app_handle_ext_adv_report: Old ext_adv_data->flag is 1, data status is 0x0

[APP] !**app_handle_ext_adv_report: Data from bd_addr 00::11::22::33::44::00 is complete, event type is 0x1, total data length is 245

[APP] !**app_handle_ext_adv_report: First five datas are 0x2, 0x1, 0x5, 0x13, 0x9

[APP] !**app_handle_ext_adv_report: Last five datas are 0xd5, 0xd6, 0xd7, 0xd8, 0xd9

[APP] !**app_handle_ext_adv_report: New ext_adv_data->flag is 0

showdev - Show scan device list.

Serial port assistant tool shows :

dev list

……

RemoteBd[3] = [47:78:39:48:32:1a]

RemoteBd[4] = [00:11:22:33:44:00]

2) Connection and Reconnection

Procedure Description: Demonstrate connecting and disconnecting with the peripheral device.

Procedure:

condev 4 x100- Scan connectable advertisements on LE Coded PHY, and initiate connection. Secondary advertising channel is LE Coded PHY, and then TX PHY type and RX PHY type are LE Coded PHY.

Serial port assistant tool shows :

condev 4 x100

Connected success conn_id 0, tx_phy 3, rx_phy 3

showcon - Show connection information.

Serial port assistant tool shows :

showcon

ShowCon conn_id 0 state 0x00000002 role 1

RemoteBd = [00:11:22:33:44:00] type = 0

active link num 1, idle link num 0

disc 0 - Disconnect with the peripheral device.

Serial port assistant tool shows :

disc 0

Disconnect conn_id 0, dis_cause 0x00000116

condev 4 x100 - Reconnect with the peripheral device.

Serial port assistant tool shows :

condev 4 x100

Connected success conn_id 0, tx_phy 3, rx_phy 3

4.8 BLE Application User Command

BLE Central Application, BLE Scatternet Application and BLE BT5 Central Application support user command. BLE Central Application, BLE Scatternet Application, BLE Peripherl Privacy Application and BLE BT5 Central Application need to perform interaction by inputting command through Data UART in PC (USB port in PC is connected to Evolution Board via USB-to-serial port module).

4.8.1 Implementation of User Command

The descriptions of each file are shown as below.

Table 4-5 User Command File

File name	Description
data_uart.c	Initialize Data UART and print data through data UART.
data_uart_dlps.c	Data UART DLPS initialization.
user_cmd_parse.c	Used to parse user command from lower Data UART data and execute right commands.
user_cmd.c	Define user commands

Source code directory: sdk\src\ mcu\module\data_uart_cmd

1. Initialization

   data_uart_init() is used to initialize the data UART. user_cmd_init() is used to initialize the user command module.

```
void app_main_task(void *p_param)
{
    uint8_t event;
    .....

    data_uart_init(evt_queue_handle, io_queue_handle);
    user_cmd_init(&user_cmd_if, "central");
    ......
}
```

2. Data UART RX Handler

```
void app_handle_io_msg(T_IO_MSG io_msg)
{
    uint16_t msg_type = io_msg.type;
    uint8_t rx_char;

    switch (msg_type) {
    .......
    case IO_MSG_TYPE_UART:
        /* We handle user command informations from Data UART in this branch. */
        rx_char = (uint8_t)io_msg.subtype;
        user_cmd_collect(&user_cmd_if, &rx_char, sizeof(rx_char), user_cmd_table);
        break;
    default:
        break;
    }
}
```

   user_cmd_collect() is used to collect command characters and execute the command. User can input the command in serial port assistant tool. Command uses line break as the end mark.

3. Data UART TX Handler

   Data UART TX is used to output information and logs.

```
void app_handle_conn_state_evt(T_IO_MSG io_msg)
{
    .......
case GAP_CONN_STATE_CONNECTED: {
        le_get_conn_addr(conn_id, app_link_table[conn_id].bd_addr,
                         &app_link_table[conn_id].bd_type);
        data_uart_print("Connected success conn_id %d\r\n", conn_id);
    }
    break;
```

```
}
```

data_uart_print() is used to output the information through the Data UART. User can see the informations in serial port assistant tool.

4. Define User Command Table

```
static T_USER_CMD_PARSE_RESULT cmd_conupdreq(T_USER_CMD_PARSED_VALUE *p_parse_value)
{
    T_GAP_CAUSE cause;
    uint8_t    conn_id = p_parse_value->dw_param[0];
    uint16_t conn_interval_min = p_parse_value->dw_param[1];
    uint16_t conn_interval_max = p_parse_value->dw_param[2];
    uint16_t conn_latency = p_parse_value->dw_param[3];
    uint16_t supervision_timeout = p_parse_value->dw_param[4];
    cause = le_update_conn_param(conn_id,
                                 conn_interval_min,
                                 conn_interval_max,
                                 conn_latency,
                                 supervision_timeout,
                                 2 * (conn_interval_min - 1),
                                 2 * (conn_interval_max - 1)
                                );
    return (T_USER_CMD_PARSE_RESULT)cause;
}
const T_USER_CMD_TABLE_ENTRY user_cmd_table[] = {
    /*************************** Common cmd ***************************************/
    {
        "conupdreq",
        "conupdreq [conn_id] [interval_min] [interval_max] [latency] [supervision_timeout]\n\r",
        "LE connection param update request\r\n\
        sample: conupdreq 0 0x30 0x40 0 500\n\r",
        cmd_conupdreq
    },
    ......
}
```

4.8.2 Data UART Connection

BLE Central Application, BLE Scatternet Application and BLE BT5 Central Application use P3_0 as TX pin for Data UART, uses P3_1 as RX pin for Data UART by default.

Each application has a copy of board.h file, which contains the definition of the Data UART Pin.

```
#define DATA_UART_TX_PIN      P3_0
#define DATA_UART_RX_PIN      P3_1
```

Data UART Pin can be configured according to the hardware environment.

Therefore PC is connected to Data UART of Evolution Board running BLE Central Application, BLE Scatternet Application or BLE BT5 Central Application via USB-to-serial port module, as shown in Figure 4-7.

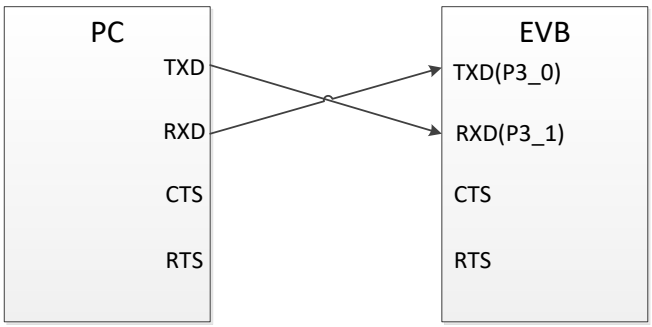

Figure 4-7 Data UART Connection of PC and EVB

Baud rate of Data UART is set to 115,200, while parameters for serial port assistant tool in PC can be set as shown in Figure 4-8.

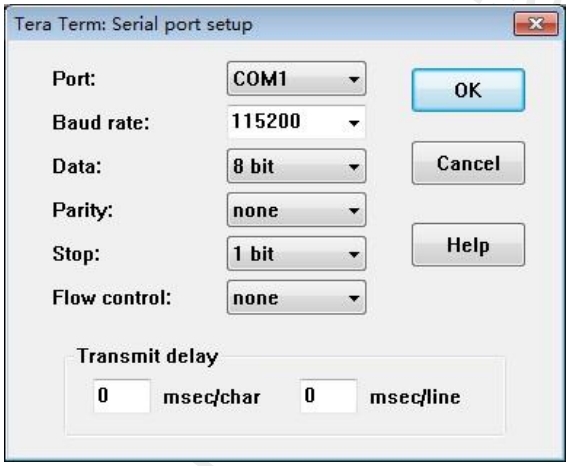

Figure 4-8 Serial Port Setting

4.8.3 User Command

This section describes in details the usage of user commands for BLE Central Application and BLE Scatternet Application. First we briefly describe the format of user commands. Taking "conupdreq [conn_id] [type]" for example, "conupdreq" is the name of the command, followed by parameters separated with Space. You can enter the command in serial port assistant tool in PC, and then press ENTER or click on "Send" to send it. The recommended serial port assistant tool is Tera Term.

BLE Central user command table is shown as below:

```
/** @brief    User command table */
const T_USER_CMD_TABLE_ENTRY user_cmd_table[] = {
    /*************************** Common cmd ************************************/
    {
        "conupdreq",
```

```
        "conupdreq [conn_id] [interval_min] [interval_max] [latency] [supervision_timeout]\n\r",
        "LE connection param update request\r\n\
        sample: conupdreq 0 0x30 0x40 0 500\n\r",
        cmd_conupdreq
    },
    {
        "showcon",
        "showcon\n\r",
        "Show all devices connecting status\n\r",
        cmd_showcon
    },
    {
        "disc",
        "disc [conn_id]\n\r",
        "Disconnect to remote device\n\r",
        cmd_disc
    },
    {
        "authmode",
        "authmode [auth_flags] [io_cap] [sec_enable] [oob_enable]\n\r",
        "Config authentication mode\r\n\
        [auth_flags]:authentication req bit field: bit0-(bonding), bit2-(MITM), bit3-(SC)\r\n\
        [io_cap]:set io Capabilities: 0-(display only), 1-(display yes/no), 2-(keyboard noly), 3-(no IO), 4-(keyboard
                display)\r\n\
        [sec_enable]:Start smp pairing procedure when connected: 0-(disable), 1-(enable)\r\n\
        [oob_enable]:Enable oob flag: 0-(disable), 1-(enable)\r\n\
        sample: authmode 0x5 2 1 0\n\r",
        cmd_authmode
    },
    ……
    /* MUST be at the end: */
    {
        0,
        0,
        0,
        0
    }
};
```

4.9 BLE Peripheral Privacy Application

4.9.1 Introduction

The purpose of this chapter is to give a sample usage of Privacy Management Module. The BLE peripheral privacy project implements a Privacy Management Module sample based on BLE peripheral project.

More information about peripheral role can be found in chapter *BLE Peripheral Application*.

More information about Privacy Management Module can be found in chapter *Privacy Management Module*.

4.9.2 Privacy Usage Flow Chart

There are two modes in the application. The device in pairable mode can be connected with any device. The device in pairable mode shall disable address resolution and disable white list filter policy.

The device in reconnect mode can only be connected with devices which are in the resolving list and white list. If application wants to filter a device which uses the resolvable private address, application shall call function privacy_set_addr_resolution() to enable address resolution.

More information about Privacy Management Module please refers to chapter *Privacy Management Module*.

The flow chart of peripheral privacy application is shown in Figure 4-9.

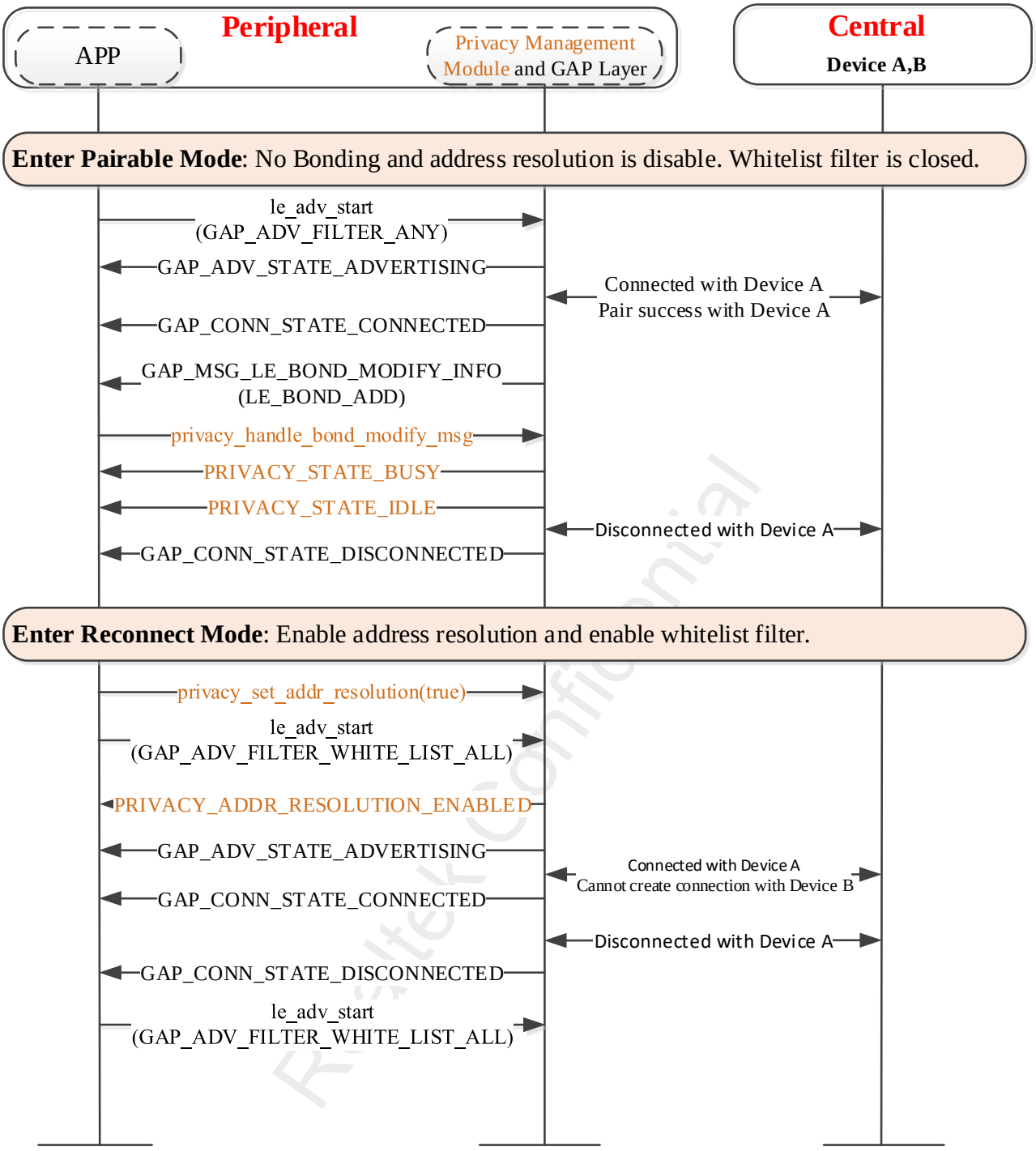

Figure 4-9 Peripheral Privacy APP Flow Chart

4.9.3 APP Configurable Functions

APP Configurable Functions are defined in app_flags.h.

```
/** @brief  Configure APP LE link number */
#define APP_MAX_LINKS  1
```

```
/** @brief    User command: 0-Close user command, 1-Open user command */
#define USER_CMD_EN       1
/** @brief    Configure Privacy1.2 feature: 0-Closed, 1-Open */
#define APP_PRIVACY_EN 1
#if APP_PRIVACY_EN
/** @brief    Configure the authentication requirement of simple_ble_service.c */
#define SIMP_SRV_AUTHEN_EN 1
#endif
```

4.9.3.1 Privacy Configuration

All privacy related code is separated by macro definition APP_PRIVACY_EN.

4.9.3.2 Security Requirement of Service

All service security related code is separated by macro definition SIMP_SRV_AUTHEN_EN.

If application wants to use privacy, application needs to enable security.

More information please refers to chapter *Security Requirement of Service.*

```
    /* client characteristic configuration */
    {
        ATTRIB_FLAG_VALUE_INCL | ATTRIB_FLAG_CCCD_APPL,                    /* flags */
        {                                                                  /* type_value */
            LO_WORD(GATT_UUID_CHAR_CLIENT_CONFIG),
            HI_WORD(GATT_UUID_CHAR_CLIENT_CONFIG),
            /* NOTE: this value has an instantiation for each client, a write to */
            /* this attribute does not modify this default value:                 */
            LO_WORD(GATT_CLIENT_CHAR_CONFIG_DEFAULT), /* client char. config. bit field */
            HI_WORD(GATT_CLIENT_CHAR_CONFIG_DEFAULT)
        },
        2,                                                                 /* bValueLen */
        NULL,
#if SIMP_SRV_AUTHEN_EN
        (GATT_PERM_READ_AUTHEN_REQ | GATT_PERM_WRITE_AUTHEN_REQ) /* permissions */
#else
        (GATT_PERM_READ | GATT_PERM_WRITE)                 /* permissions */
#endif
    },
```

4.9.3.3 User command

BLE Peripheral Privacy Application needs to perform interaction by inputting command through Data UART in PC. More information about user command can be found in chapter *BLE Application User Command*. The following sections describe some commands used in Peripheral Privacy application.

Supported user commands are shown as below:

```
const T_USER_CMD_TABLE_ENTRY user_cmd_table[] =
{
    /*************************** Common cmd ***************************************/
    {
        "disc",
        "disc [conn_id]\n\r",
        "Disconnect to remote device\n\r",
        cmd_disc
    },
    {
        "bondclear",
        "bondclear\n\r",
        "Clear all bonded devices information\n\r",
        cmd_bondclear
    },
    ……
};
```

Command disc is used to disconnect the LE link.

Command bondclear is used to clear all bonding informations.

4.9.4 Test Procedure

At first, please build and download the BLE Peripheral Privacy application to the Evolution Board.

When BLE Peripheral Privacy Application is being run on Evolution Board, the Bluetooth LE device becomes discoverable and connectable. Remote device can scan the peripheral device and create connection with peripheral device. After disconnection, BLE Peripheral Privacy Application will restore to be discoverable and connectable again.

4.9.4.1 Test with iOS Device

Procedure Description: iOS-based devices are always compatible with Bluetooth LE, and devices running BLE Peripheral Privacy Application can be discovered. It is recommended to use Bluetooth LE-related App (e.g.

LightBlue) in App Store to perform search and connection test.

Procedure: Run LightBlue on iOS device to search for and connect with a Bluetooth LE_PRIVACY device, as shown in Figure 4-10:

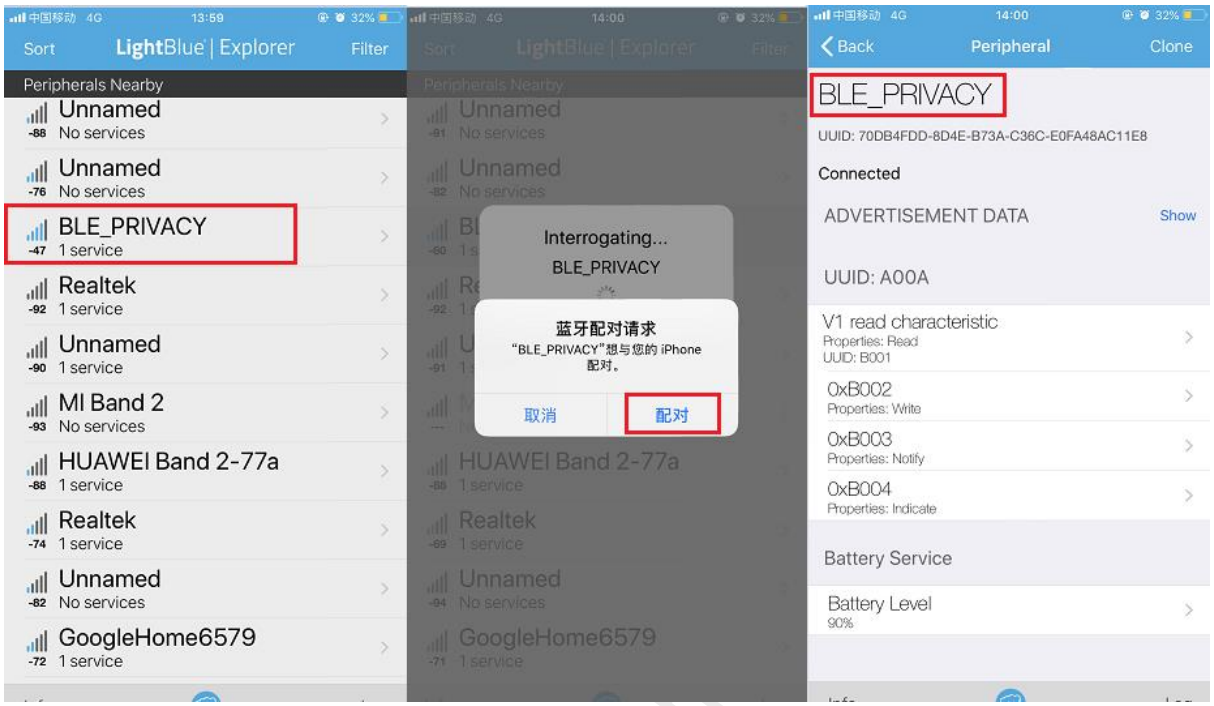

Figure 4-10 Test with iOS Device

References

[1] Bluetooth SIG. Core_v5.0 [M]. 2016, 169.