REALTEK

UM0201

Ameba Common BT Application User Manual

Abstract

This document describes the information of BLE stack and the BLE sample project.

REVISION HISTORY

Revision	Release Date	Summary
0.1	2019/07/19	First version
0.2	2019/08/22	Add modify the white list
0.3	2019/08/27	Add BLE Scatternet
0.4	2019/09/26	Add method to set peripheral example to encrypted transmission
0.5	2020/02/14	Add the size of white list
0.6	2020/04/09	Add method to set random device address
0.7	2020/08/31	Fix some errors

Contents

Glossary

Terms	Definitions
ATT	Attribute protocol
BLE	Bluetooth Low Energy
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 (BLE) 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 BLE Stack Interfaces. BLE Stack Interfaces can be divided into Generic Access Profile (GAP) interfaces and Generic Attribute Profile (GATT) based profile interfaces.

1.1 Supported BT Features

Table 1-1 Supported BT Features

Spec Version	BT Feature	AmebaD	AmebaZ2	Remark
BT4.0	Advertiser	Y	Y	
	Scanner	Y	Y	
	Initiator	Y	Y	
	Master	Y	Y	3 master link for AmebaD 1 master link for AmebaZ2
	Slave	Y	Y	1 slave link
BT4.1	Low Duty Cycle Directed Advertising	Y	N	
	LE L2CAP Connection Oriented Channel	N	N	
	LE Scatternet	Y	Y	3 master link + 1 slave link for AmebaD 1 master link + 1 slave link for AmebaZ2
	LE Ping	Y	N	
BT4.2	LE Data Packet Length Extension	Y	N	
	LE Secure Connections	Y	Y	
	Link Layer Privacy(Privacy1.2)	N	N	
	Link Layer Extended Filter Policies	N	N	
BT5	2 Msym/s PHY for LE	Y	N	
	LE Long Range	N	N	
	High Duty Cycle Non-Connectable Advertising	Y	N	
	LE Advertising Extensions	N	N	
	LE Channel Selection Algorithm #2	N	N	

1.2 BLE 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 1-1. As shown in Figure 1-1, Profile is illustrated in red rectangular, GAP, Profile #1, Profile #2, and Profile #3.

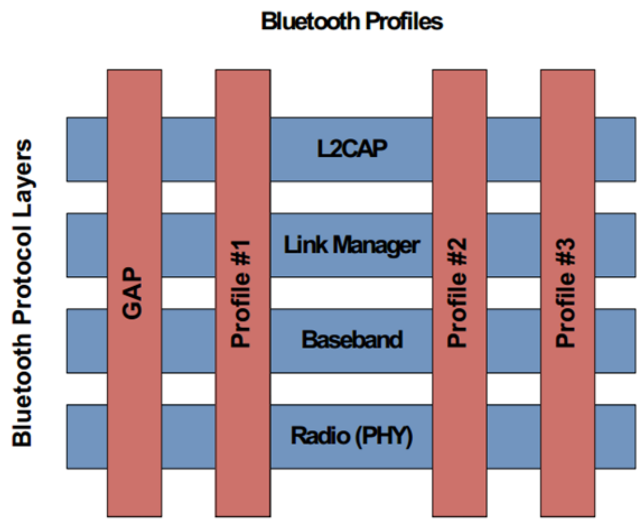

Figure 1-1 Bluetooth Profiles

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 1-2.

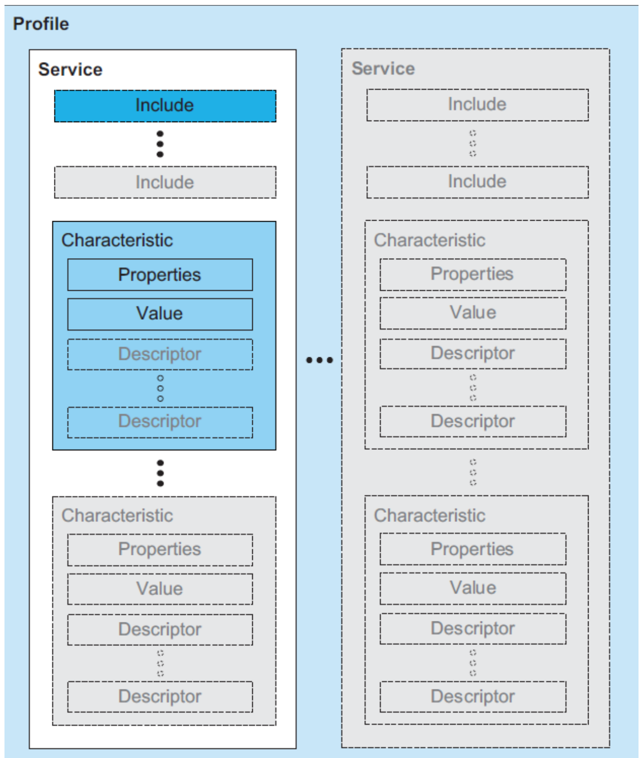

Figure 1-2 GATT Based Profile Hierarchy

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 BT Lib, and provides interfaces to application. BT Lib files directory: component\common\bluetooth\realtek\sdk\board\amebaX\lib.

Figure 2-1 BT Lib

Header files are provided in SDK. GAP header files directory: component\common\bluetooth\realtek\sdk\ inc\bluetooth\gap.

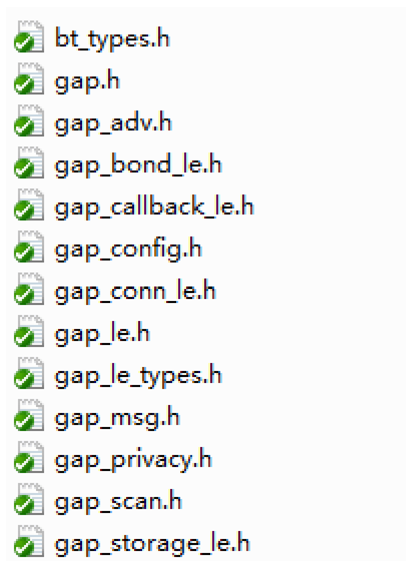

Figure 2-2 GAP Header Files

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 *BLE 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 *BLE GAP Callback*.
- How to use GAP interfaces please refer to chapter *BLE 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 2-3, 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.

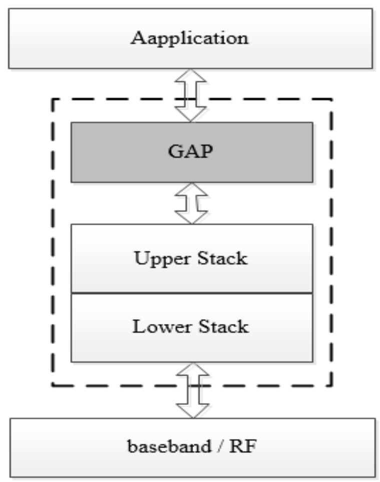

Figure 2-3 GAP Location in SDK

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 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 BLE 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. Taking BLE peripheral as an example, when customers calls the GAP API in their own Task, we recommend sending the message to app_main_task() by send message and then handling the corresponding message in app_main_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. 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.
```

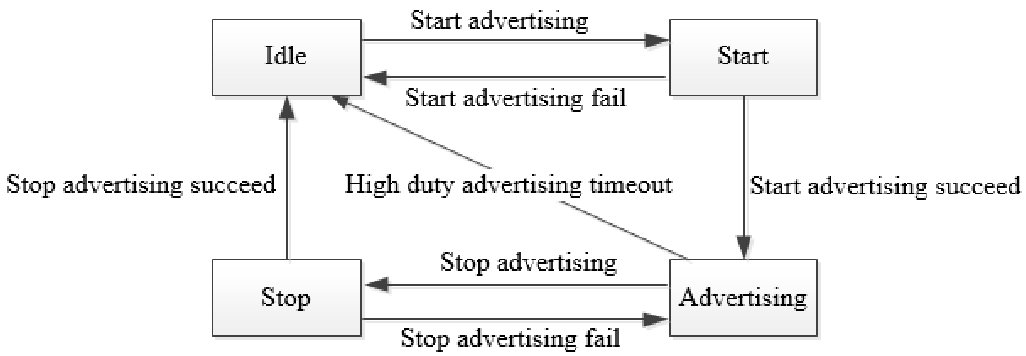

Figure 2-4 Advertising State Transition Mechanism

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
```

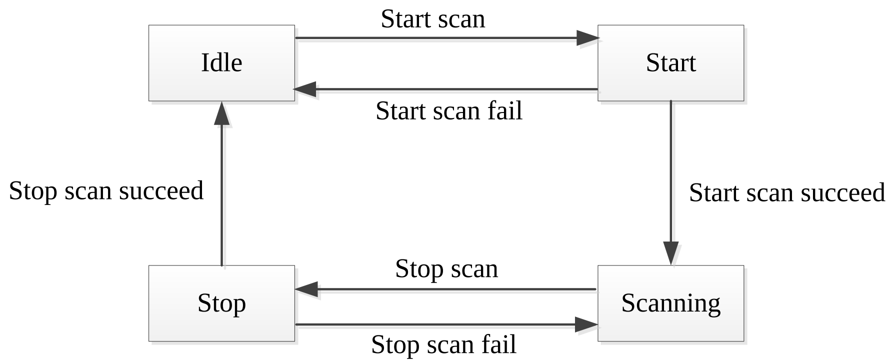

Figure 2-5 Scan State Transition Mechanism

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.

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

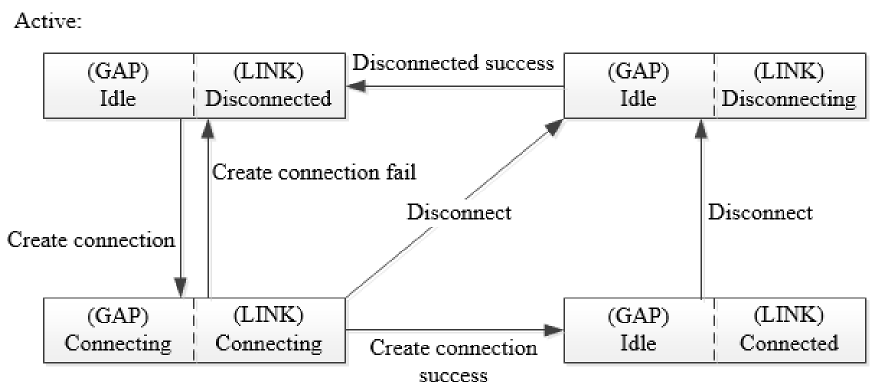

Figure 2-6 Active State Transition Mechanism

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 turn 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

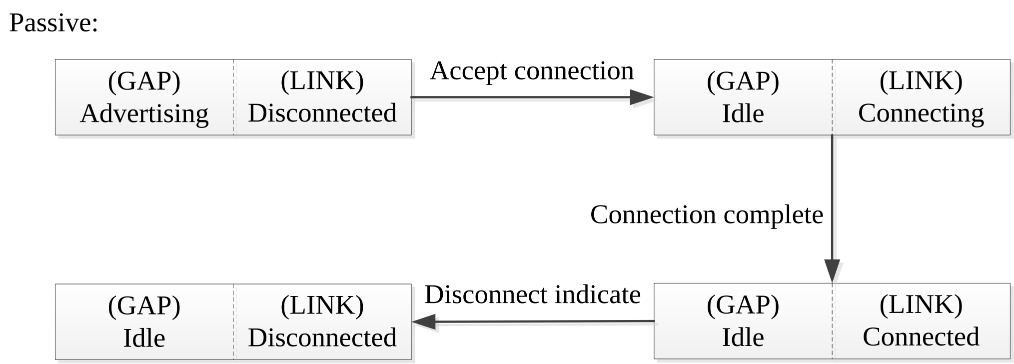

Figure 2-7 Passive State Transition Mechanism

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 process 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.

2.1.4 GAP Message

GAP message includes BT status message and GAP API message. BT 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 *BLE GAP Message* and chapter *BLE GAP Callback*.

2.1.4.1 BT Status Message

BT 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.
```

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_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_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_SCAN_INFO              0x30 // Notification msg type for le scan
#define GAP_MSG_LE_ADV_UPDATE_PARAM        0x40 // response msg type for le_adv_update_param
```

2.1.5 APP Message Flow

APP message flow is shown in Figure 2-8. Mandatory steps are written in solid line box and optional steps are written in dashed line box.

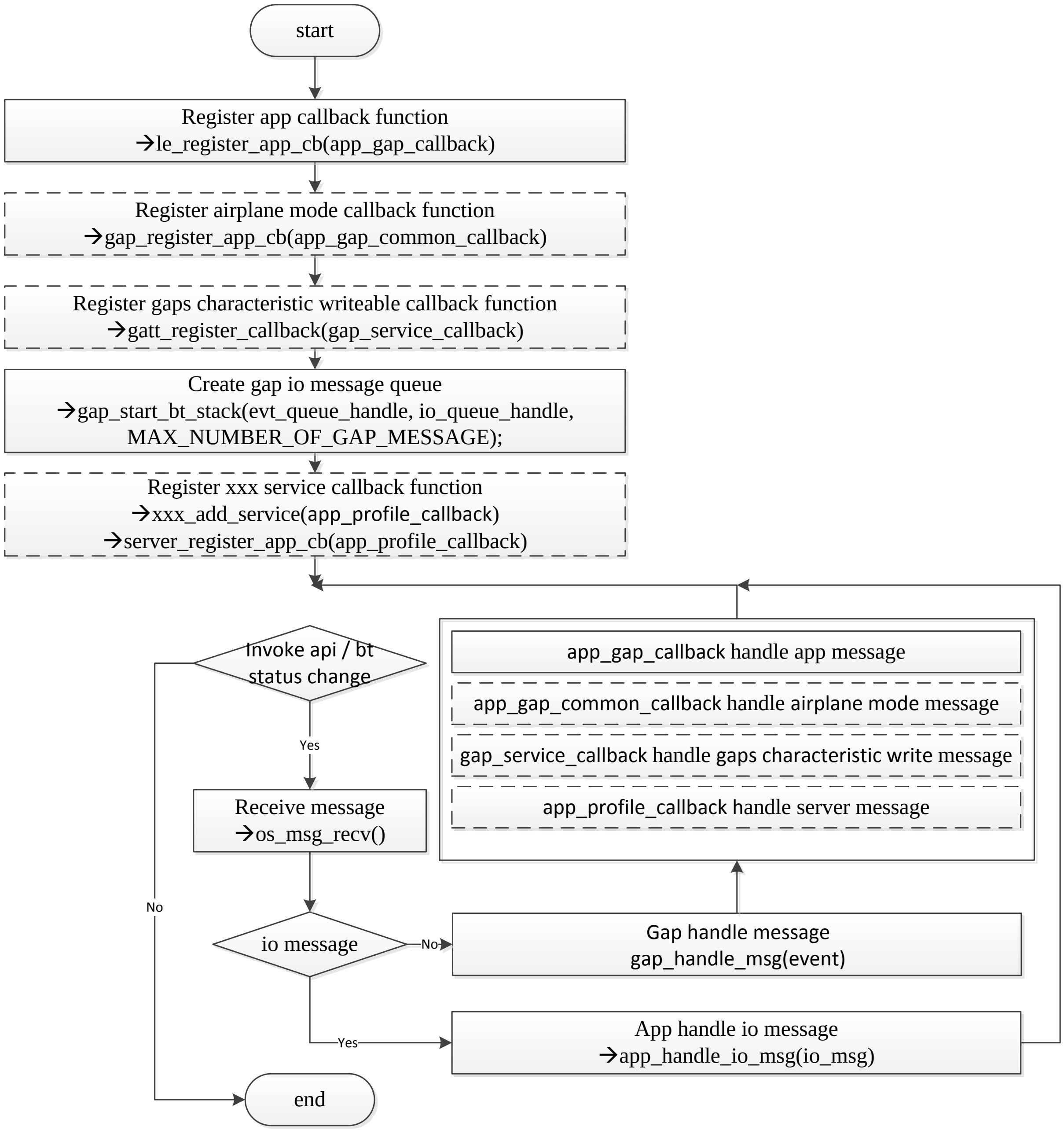

Figure 2-8(a) BLE peripheral APP Message Flow

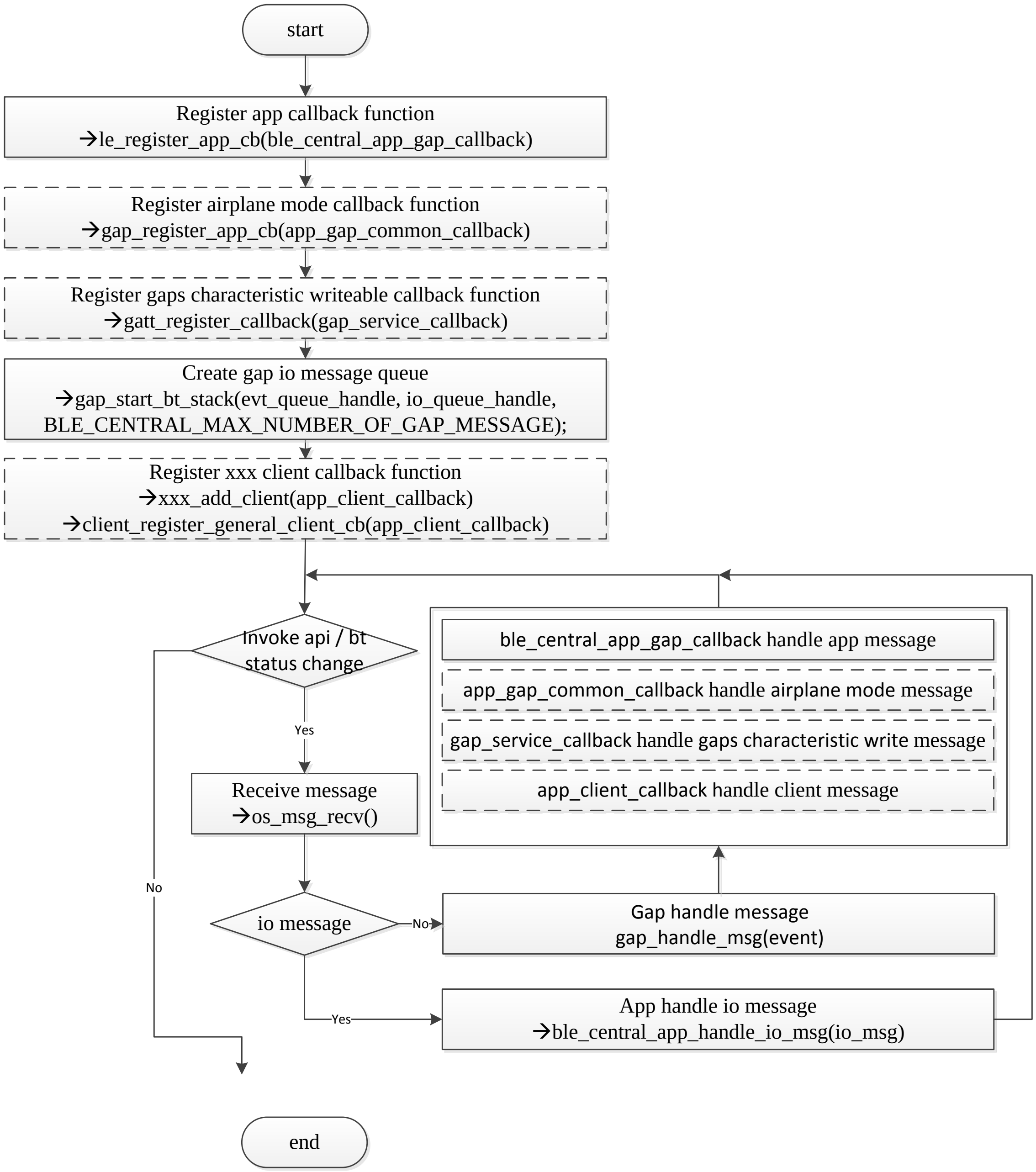

Figure 2-8(b) BLE central APP Message Flow

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 vendor command messages, APP shall register app callback function by invoking vendor_cmd_init() 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 BT 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. According to the type of message, APP calls corresponding function to handle this message in APP layer or 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 and gap internal startup flow.

2.2.1 GAP Parameters Initialization

GAP parameter initialization in function app_le_gap_init() for BLE Peripheral and function ble_central_app_le_gap_init() for BLE Central.

Table 2-1 Supported GAP Parameters Configuration

Parameters configuration	BLE Peripheral	BLE Central
Configure Device Name	Y	Y
Configure Device Appearance	Y	Y
Configure Advertising Parameters	Y	N

Configure Scan Parameters	N	Y
Configure BondManager Parameters	Y	Y

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.

The following is sample code for BLE Peripheral:

```
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);
   ......
}
```

The following is sample code for BLE Central:

```
void ble_central_app_le_gap_init(void)
{
   /* Device name and device appearance */
   uint8_t device_name[GAP_DEVICE_NAME_LEN] = "BLE_CENTRAL_CLIENT";
   ......
   /* 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 BT 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
```

The following is sample code for BLE Peripheral:

```
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);
  ......
}
```

The following is sample code for BLE Central:

```
void ble_central_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.

The advertising parameters configuration of BLE Peripheral which can be customized are listed below:

```
/** @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',
```

```
};

/** @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)
{
   /* 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 2-2.

Table 2-2 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.

The parameters configuration of BLE Central which can be customized are listed below:

```
void ble_central_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.

NOTE: In order to achieve better performance in the case of WIFI/BT coexistence, it is recommended to set the scan interval and scan window to the same value.

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.

The bond manager parameters of BLE Peripheral 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);
}
```

The bond manager parameters of BLE Central which can be customized are listed below:

```
void ble_central_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_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 Other Parameters

2.2.1.5.1 Configure GAP_PARAM_SLAVE_INIT_GATT_MTU_REQ

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

```
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);
    ......
}
```

2.2.2 GAP Startup Flow

1. Initialize GAP

1.1 BLE Peripheral Initialize GAP in ble_app_main ()

```
int ble_app_main (void)
{
    ......
    bt_stack_config_init();
    le_gap_init(APP_MAX_LINKS);
    app_le_gap_init();
    app_le_profile_init();
```

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

- *bt_stack_config_init()* – call gap_config_max_le_link_num(APP_MAX_LINKS), is the configuration of lib
- *le_gap_init()* - Initialize GAP and set link number
- *app_le_gap_init()* – *GAP Parameters Intialization*
- *app_le_profile_init()* - Initialize GATT Profiles

1.2 BLE Central Initialize GAP in ble_central_app_main()

```
int ble_central_app_main (void)
{
    ......
    bt_stack_config_init();
    le_gap_init(BLE_CENTRAL_APP_MAX_LINKS);
    ble_central_app_le_gap_init();
    ble_central_app_le_profile_init();
    ......
}
```

- *bt_stack_config_init()* – call gap_config_max_le_link_num (BLE_CENTRAL_APP_MAX_LINKS), is the configuration of lib
- *le_gap_init()* - Initialize GAP and set link number
- *ble_central_app_le_gap_init()* - *GAP Parameters Intialization*
- *ble_central_app_le_profile_init()* - Initialize GATT Profiles

2. Start BT stack

2.1 BLE Peripheral start BT stack in app_main_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);
    ......
}
```

2.2 BLE Central start BT stack in ble_central_app_main_task()

```
void ble_central_app_main_task(void *p_param)
{
    uint8_t event;
    os_msg_queue_create(&ble_central_io_queue_handle, BLE_CENTRAL_MAX_NUMBER_OF_IO_MESSAGE,
sizeof(T_IO_MSG));
    os_msg_queue_create(&ble_central_evt_queue_handle, BLE_CENTRAL_MAX_NUMBER_OF_EVENT_MESSAGE,
sizeof(uint8_t));
    gap_start_bt_stack(ble_central_evt_queue_handle, ble_central_io_queue_handle,
BLE_CENTRAL_MAX_NUMBER_OF_GAP_MESSAGE);
    ......
}
```

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

2.3 BLE GAP Message

2.3.1 Overview

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

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

BLE Peripheral GAP Message process flow:

1. APP can call gap_start_bt_stack() to initialize the BLE 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 is 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;
        ......
    }
    ......
}
```

BLE Central GAP Message process flow:

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

```
void ble_central_app_main_task(void *p_param)
{
    uint8_t event;
    os_msg_queue_create(&ble_central_io_queue_handle, BLE_CENTRAL_MAX_NUMBER_OF_IO_MESSAGE,
sizeof(T_IO_MSG));
    os_msg_queue_create(&ble_central_evt_queue_handle, BLE_CENTRAL_MAX_NUMBER_OF_EVENT_MESSAGE,
sizeof(uint8_t));
    gap_start_bt_stack(ble_central_evt_queue_handle, ble_central_io_queue_handle,
BLE_CENTRAL_MAX_NUMBER_OF_GAP_MESSAGE);
    ......
}
```

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

```
void ble_central_app_main_task(void *p_param)
{
    ......
    while (true)
    {
        if (os_msg_recv(ble_central_evt_queue_handle, &event, 0xFFFFFFFF) == true)
        {
```

```
        if (event == EVENT_IO_TO_APP)
        {
            T_IO_MSG io_msg;
            if (os_msg_recv(ble_central_io_queue_handle, &io_msg, 0) == true)
            {
                ble_central_app_handle_io_msg(io_msg);
            }
        }
        ......
    }
  }
}
```

3. The gap messages handler function is given below:

```
void ble_central_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:
            {
                ble_central_app_handle_gap_msg(&io_msg);
            }
            break;
        default:
            break;
    }
}

void ble_central_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("ble_central_app_handle_gap_msg: subtype %d", p_gap_msg->subtype);
    switch (p_gap_msg->subtype)
    {
        case GAP_MSG_LE_DEV_STATE_CHANGE:
            {
                ble_central_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 of BLE Peripheral 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");
         printf("\n\r[BLE peripheral] GAP stack ready\n\r");
         le_adv_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");
            printf("\n\rGAP adv stoped: because connection created\n\r");
         }
         else
         {
            APP_PRINT_INFO0("GAP adv stoped");
            printf("\n\rGAP adv stoped\n\r");
         }
      }
```

```
    else if (new_state.gap_adv_state == GAP_ADV_STATE_ADVERTISING)
    {
        APP_PRINT_INFO0("GAP adv start");
        printf("GAP adv start\n\r");
      }
    }
    gap_dev_state = new_state;
}
```

The sample codes of BLE Central are given as below:

```
void ble_central_app_handle_dev_state_evt(T_GAP_DEV_STATE new_state, uint16_t cause)
{
    APP_PRINT_INFO3("ble_central_app_handle_dev_state_evt: init state  %d, scan state %d, cause 0x%x",
                new_state.gap_init_state,
                new_state.gap_scan_state, cause);
    if (ble_central_gap_dev_state.gap_init_state != new_state.gap_init_state)
    {
        if (new_state.gap_init_state == GAP_INIT_STATE_STACK_READY)
        {
            uint8_t bt_addr[6];
            APP_PRINT_INFO0("GAP stack ready");
            /*stack ready*/
            gap_get_param(GAP_PARAM_BD_ADDR, bt_addr);
            data_uart_print("local bd addr: 0x%2x:%2x:%2x:%2x:%2x:%2x\r\n",
                        bt_addr[5],
                        bt_addr[4],
                        bt_addr[3],
                        bt_addr[2],
                        bt_addr[1],
                        bt_addr[0]);
        }
    }
    if (ble_central_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");
            data_uart_print("GAP scan stop\r\n");

        }
        else if (new_state.gap_scan_state == GAP_SCAN_STATE_SCANNING)
        {
            APP_PRINT_INFO0("GAP scan start");
            data_uart_print("GAP scan start\r\n");

        }
    }

    ble_central_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 of BLE Peripheral 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);
                }
                printf("\n\r[BLE peripheral] BT Disconnected,start ADV\n\r");
                le_adv_start();
            }
            break;
        case GAP_CONN_STATE_CONNECTED:
            {
                ......
            }
            break;
        default:
            break;
    }
    gap_conn_state = new_state;
}
```

The sample codes of BLE Central are given as below:

```
void ble_central_app_handle_conn_state_evt (uint8_t conn_id, T_GAP_CONN_STATE new_state, uint16_t disc_cause)
{
    if (conn_id >= BLE_CENTRAL_APP_MAX_LINKS)
    {
        return;
    }

    APP_PRINT_INFO4("ble_central_app_handle_conn_state_evt: conn_id %d, conn_state(%d -> %d), disc_cause 0x%x",
                    conn_id, ble_central_app_link_table[conn_id].conn_state, new_state, disc_cause);

    ble_central_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("ble_central_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(&ble_central_app_link_table[conn_id], 0, sizeof(T_APP_LINK));
          }
          break;

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

           }
           break;

        default:
           break;

    }
}
```

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 of BLE Peripheral 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;
    }
}
```

The sample codes of BLE Central are given below:

```
void ble_central_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 of BLE Peripheral 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);
}
```

The sample codes of BLE Central are given below:

```
void ble_central_app_handle_conn_mtu_info_evt (uint8_t conn_id, uint16_t mtu_size)
{
    APP_PRINT_INFO2("ble_central_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 2-3.

Table 2-3 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

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 of BLE Peripheral 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;
    }
}
```

The sample codes of BLE Central are given as below:

```
void ble_central_app_handle_authen_state_evt (uint8_t conn_id, uint8_t new_state, uint16_t cause)
{
   APP_PRINT_INFO2("ble_central_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("ble_central_app_handle_authen_state_evt: GAP_AUTHEN_STATE_STARTED");
            }
            break;
        case GAP_AUTHEN_STATE_COMPLETE:
            {
                if (cause == GAP_SUCCESS)
                {
                    data_uart_print("Pair success\r\n");
                    APP_PRINT_INFO0("ble_central_app_handle_authen_state_evt: GAP_AUTHEN_STATE_COMPLETE pair
```

```
success");
            }
            else
            {
                data_uart_print("Pair failed: cause 0x%x\r\n", cause);
                APP_PRINT_INFO0("ble_central_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 of BLE Peripheral 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;
}
```

The sample codes of BLE Central are given below:

```
void ble_central_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);
            data_uart_print("GAP_MSG_LE_BOND_PASSKEY_INPUT: conn_id %d\r\n", conn_id);
            //le_bond_passkey_input_confirm(conn_id, passkey, 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 of BLE Peripheral are given below:

```
void app_handle_gap_msg(T_IO_MSG *p_gap_msg)
{
    ......
    case GAP_MSG_LE_BOND_PASSKEY_INPUT:
        {
            suint32_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;
}
```

The sample codes of BLE Central are given below:

```
void ble_central_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);
            data_uart_print("GAP_MSG_LE_BOND_PASSKEY_DISPLAY: conn_id %d, passkey %d\r\n", conn_id, display_value);
        }
        break;
}
```

2.3.4.4 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 of BLE Peripheral 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;
}
```

The sample codes of BLE Central are given below:

```
void ble_central_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);
            data_uart_print("GAP_MSG_LE_BOND_USER_CONFIRMATION: conn_id %d, passkey %d\r\n", conn_id,
display_value);
            //le_bond_user_confirm(conn_id, GAP_CFM_CAUSE_ACCEPT);
        }
        break;
}
```

2.3.4.5 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 of BLE Peripheral 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;
}
```

The sample codes of BLE Central are given below:

```
void ble_central_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;
}
```

2.4 BLE GAP Callback

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

Different from *BLE 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. Alternatively, you can create a lower-priority task to handle time-consuming operations and increase the priority of app_main_task(). (For example, create a new task with a priority of 1 to handle time-consuming operations, and increase the priority of app_main_task() from 1 of default value to 2.)

Usage of BLE GAP Callback in Application consists of the following steps:

1. Register the callback function

1.1 BLE Peripheral register the callback function

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

1.2 BLE Central register the callback function

```
void ble_central_app_le_gap_init(void)
{
    ......
    le_register_app_cb(ble_central_app_gap_callback);
}
```

2. Handle the GAP callback messages

1.1 BLE Peripheral 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;
      ......
    }
}
```

1.2 BLE Central handle the GAP callback messages

```
T_APP_RESULT ble_central_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;
    char adv_type[20];
    char remote_addr_type[10];
    switch (cb_type) {
      case GAP_MSG_LE_SCAN_INFO:
        ble_central_app_parse_scan_info(p_data->p_le_scan_info);
      break;
      ......
    }
}
```

2.4.1 BLE 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 BLE GAP Message information is listed as below:

1. **gap_le.h Related Messages**

Table 2-4 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

2. **gap_conn_le.h Related Messages**

Table 2-5 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_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_REMOTE_FEATS_INFO	T_LE_REMOTE_FEATS_INFO *p_le_remote_feats_info;	

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 ble_central_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 2-6 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;	

1) 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 2-7 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

1) GAP_MSG_LE_SCAN_INFO

Scan state is GAP_SCAN_STATE_SCANNING. When BT stack receives advertising data or scan response data, GAP Layer will use this message to inform application.

5. gap_adv.h Related Messages

Table 2-8 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

2.5 BLE 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 GAP Service Characteristic Writeable

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*.

2.5.2 Local Static Random Address

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 and storage 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.

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)
{
    ……
    T_GAP_LOCAL_ADDR_TYPE local_addr_type = GAP_LOCAL_ADDR_LE_RANDOM;
    ……
    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);
    ……
}
```

2.5.3 Physical (PHY) Setting

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**. 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. **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;
    uint8_t  rx_phys_prefer = GAP_PHYS_PREFER_1M_BIT | GAP_PHYS_PREFER_2M_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:
         {
            ......
#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, BT stack will read remote features. GAP Layer will send GAP_MSG_LE_REMOTE_FEATS_INFO to inform APP the remote features. APP can check whether remote device supports LE 2M PHY.

```
T_APP_RESULT app_gap_callback(uint8_t cb_type, void *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");
            }
         }
         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
    }
}
```

2.6 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 chapter *FTL Introduction*.

2.6.1 FTL Introduction

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

2.6.1.1 FTL Layout

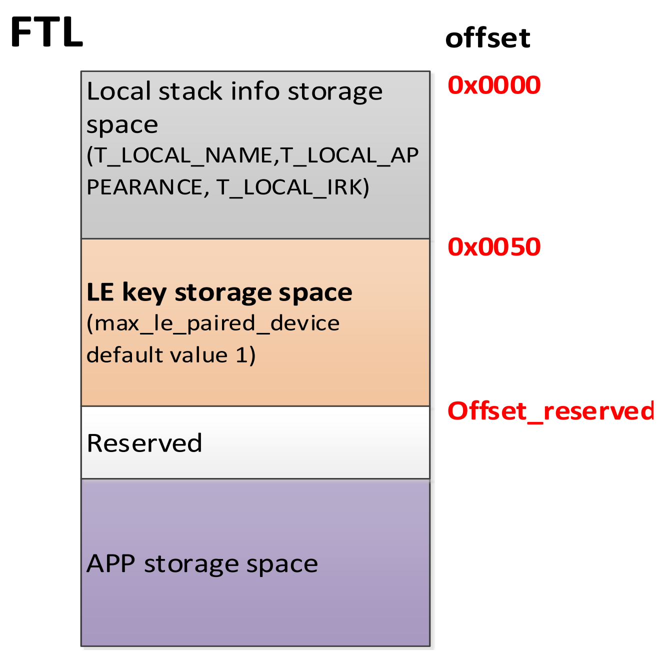

Figure 2-9 FTL Layout

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 chapter *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 chapter *Bond Information Storage*.
3. APP storage space
   1) APP can use this region to store information.

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 chapter *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 chapter *GAP Service Characteristic Writeable.*

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.

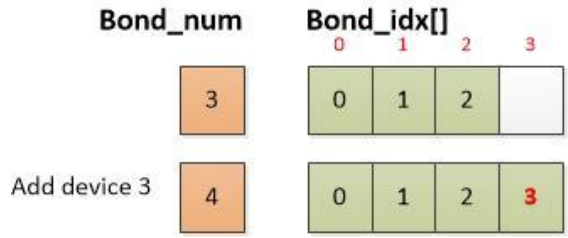

Figure 2-10 Add A Bond Device

2. **Remove a bond device**

   GAP LE API: le_bond_delete_by_idx() or le_bond_delete_by_bd()

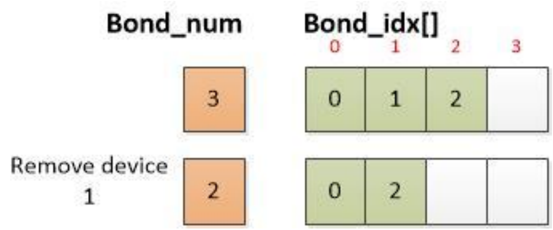

Figure 2-11 Remove A Bond Device

3. **Clear all bond devices**

   GAP LE API: le_bond_clear_all_keys()

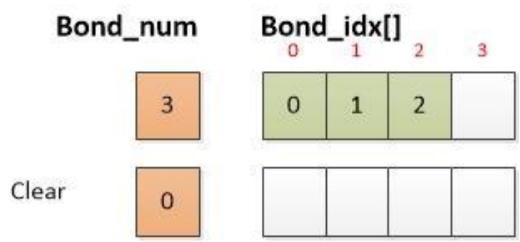

Figure 2-12 Clear All Bond Devices

4. **Set a bond device high priority**

   GAP LE API: le_set_high_priority_bond()

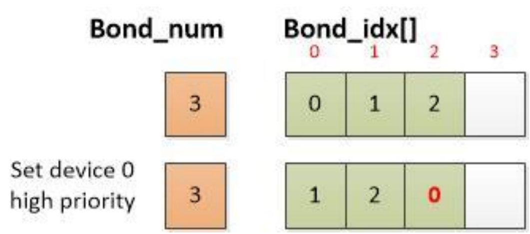

Figure 2-13 Set A Bond Device High Priority

5. **Get high priority device**

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

   GAP LE API: le_get_high_priority_bond()

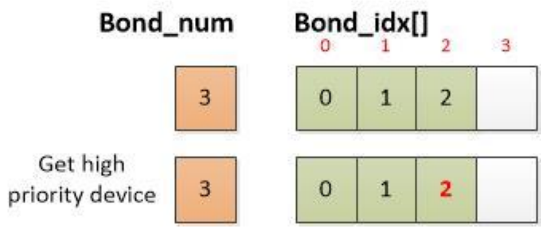

Figure 2-14 Get High Priority Device

6. Get low priority device

The lowest priority device is bond_idx[0].

GAP LE API: le_get_low_priority_bond()

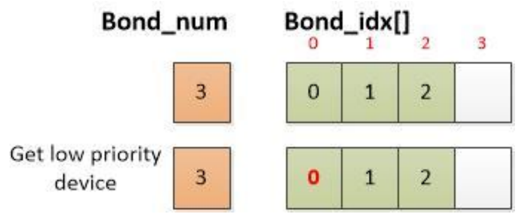

Figure 2-15 Get Low Priority Device

A priority manager example is shown in Figure 2-16:

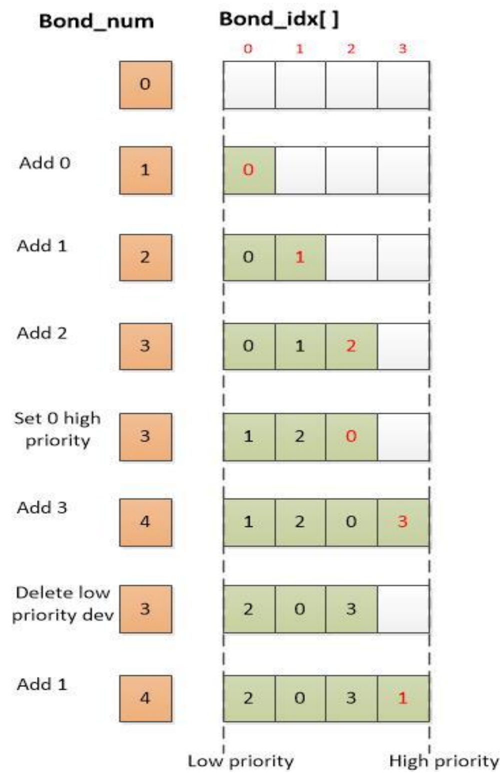

Figure 2-16 Priority Manager Example

2.6.3.2 BLE Key Storage

BLE Key information is stored in LE key storage space.

LE FTL layout is shown in Figure 2-17:

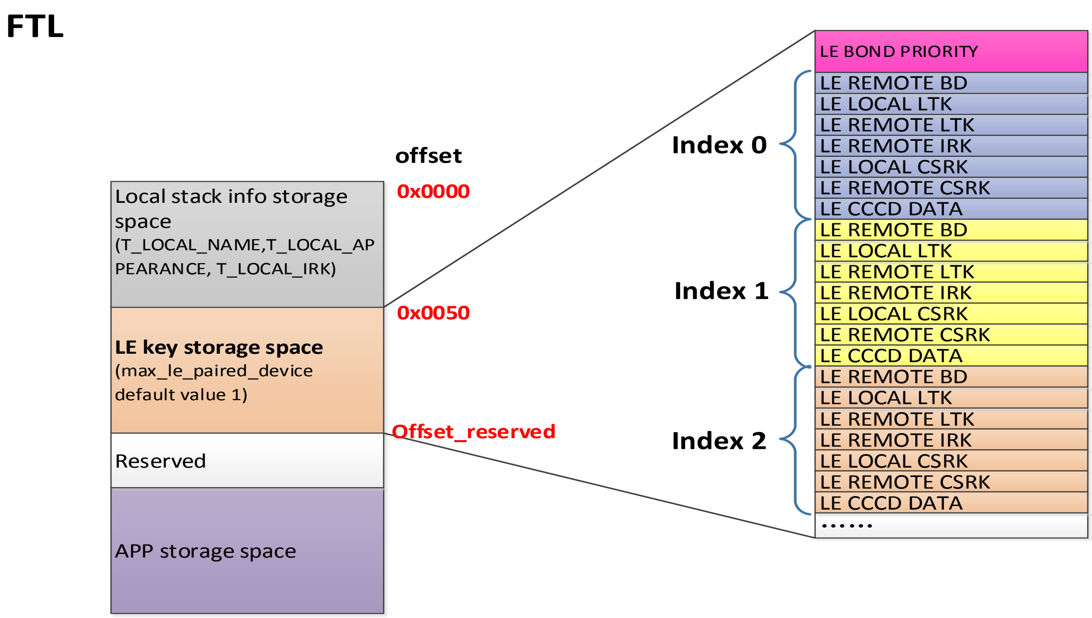

Figure 2-17 LE FTL Layout

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.

2. Maximum CCCD Number
   - Default value is 16.

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 APP the 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 BLE 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);
/* 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);
```

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 *BLE 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 *BLE Profile Client*.

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

GATT Profile header files directory:

component\common\bluetooth\realtek\sdk\inc\bluetooth\profile.

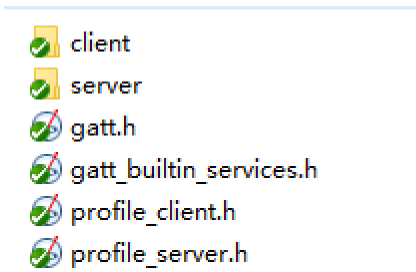

Figure 3-1 GATT Profile Header Files

3.1 BLE 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 BLE 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.

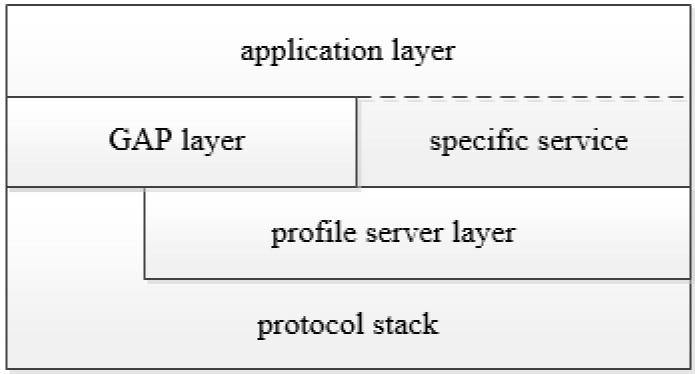

Figure 3-2 Profile Server Hierarchy

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

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

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.

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.

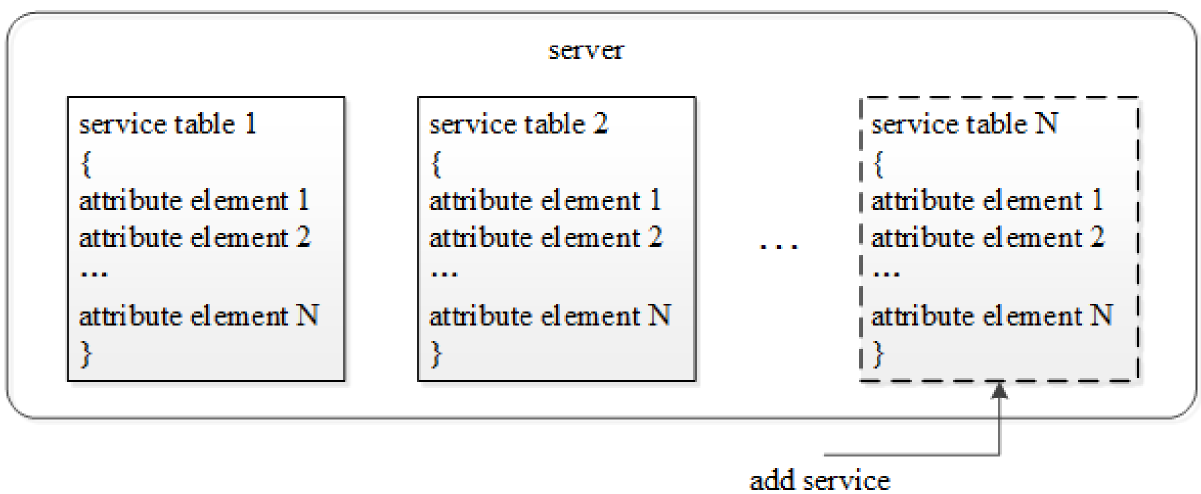

Figure 3-3 Add Services to Server

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 registration process.

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

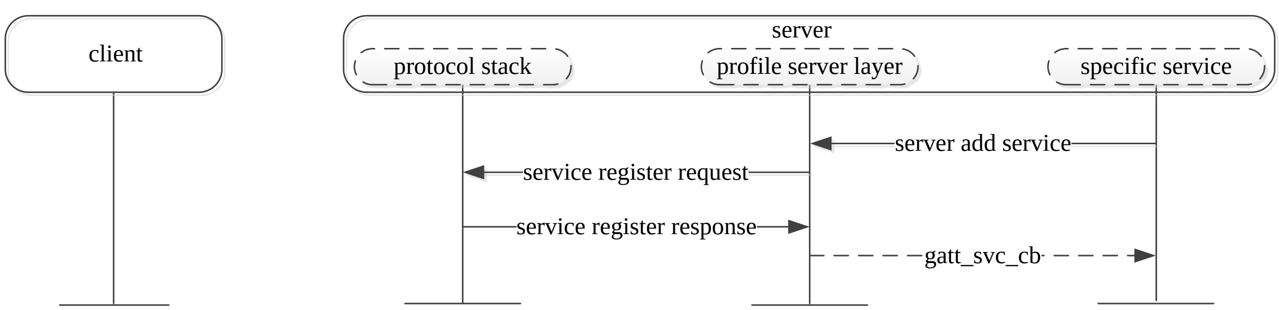

Figure 3-4 Register Service's Process

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 specific service 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 specific service when attribute write request is sent from client side.

cccd_update_cb: Client characteristic configuration descriptor value update callback, which is used to inform specific service 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 interaction 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.

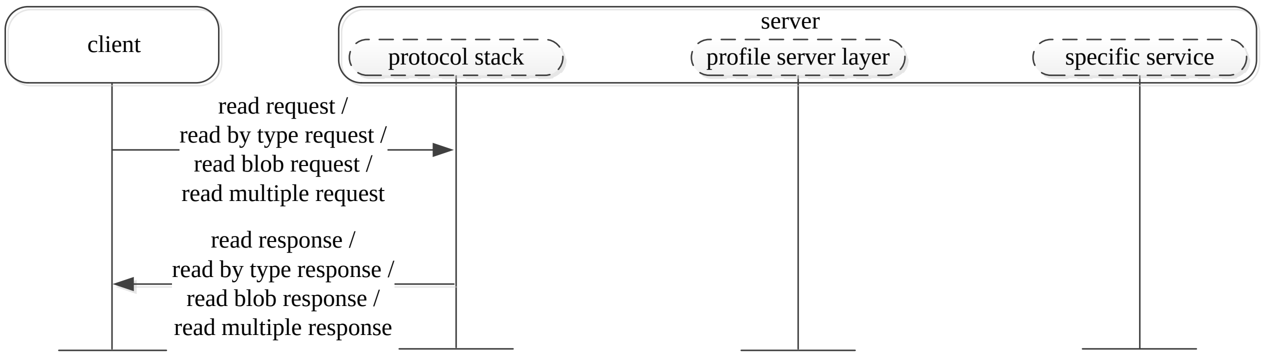

Figure 3-5 Read Characteristic Value - Attribute Value Supplied in Attribute Element

3.1.3.3.2 Attribute Value Supplied by Application without Result Pending

The attribute with flag ATTRIB_FLAG_VALUE_APPL will be involve 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 interaction 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.

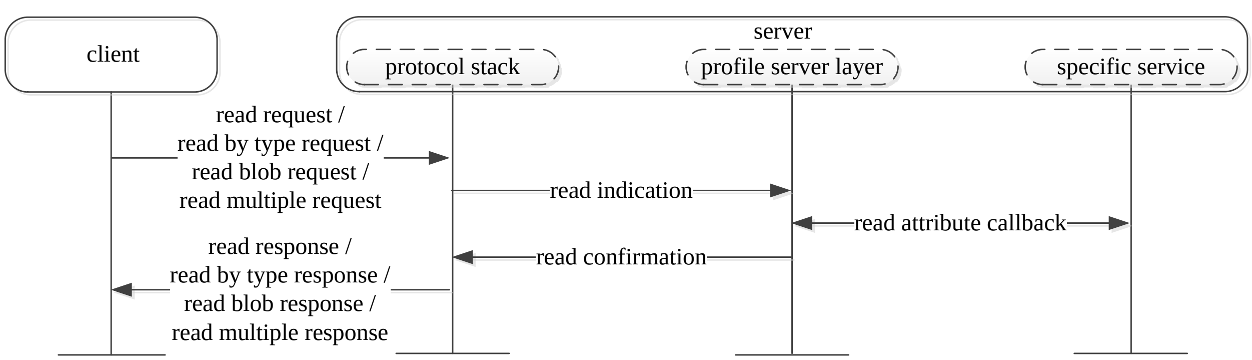

Figure 3-6 Read Characteristic Value - Attribute Value Supplied by Application without Result Pending

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.

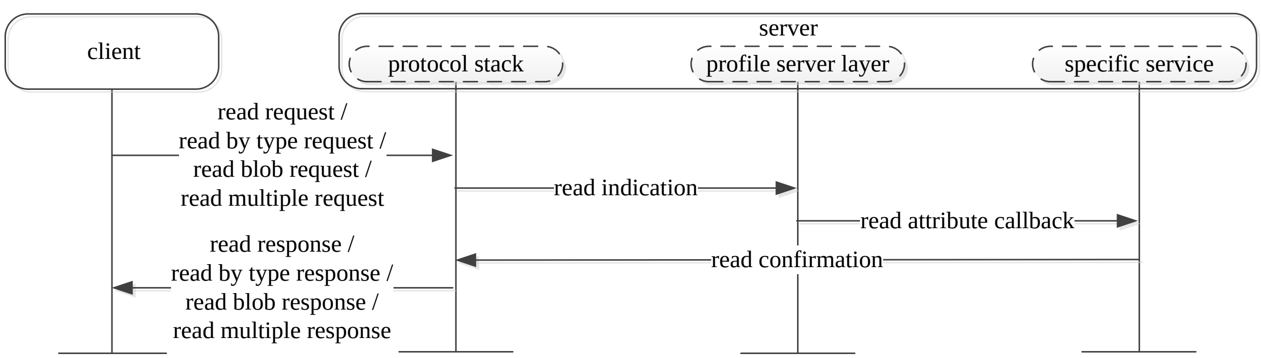

Figure 3-7 Read Characteristic Value - Attribute Value Supplied by Application with Result Pending

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.

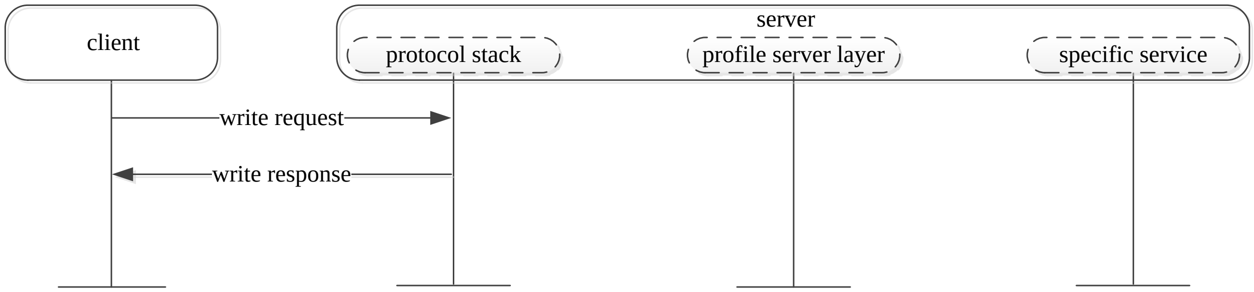

Figure 3-8 Write Characteristic Value - Attribute Value Supplied in Attribute Element

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.

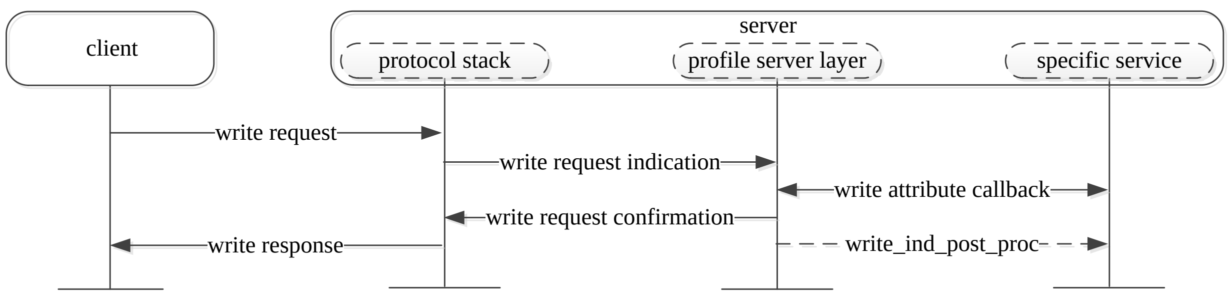

Figure 3-9 Write Characteristic Value - Attribute Value Supplied by Application without Result Pending

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.

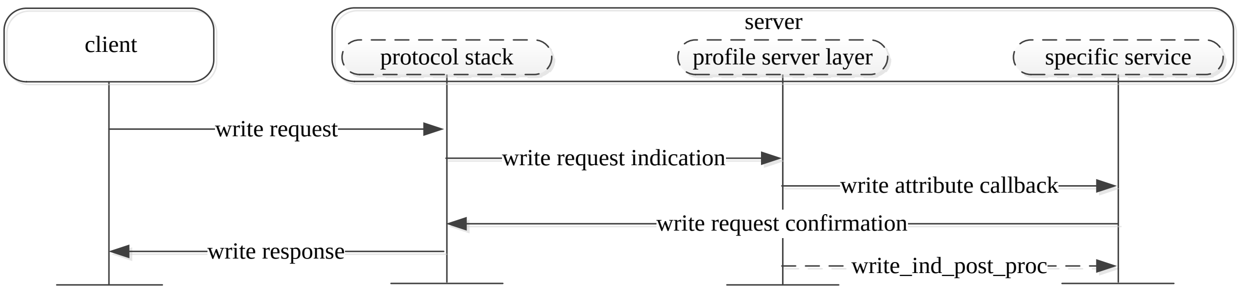

Figure 3-10 Write Characteristic Value - Attribute Value Supplied by Application with Result Pending

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.

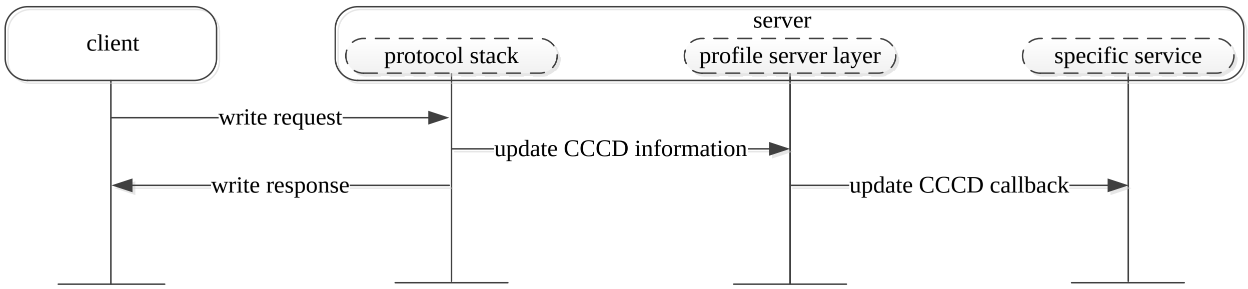

Figure 3-11 Write Characteristic Value – Write CCCD Value

```
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)
      {
         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

The difference between 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, 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.

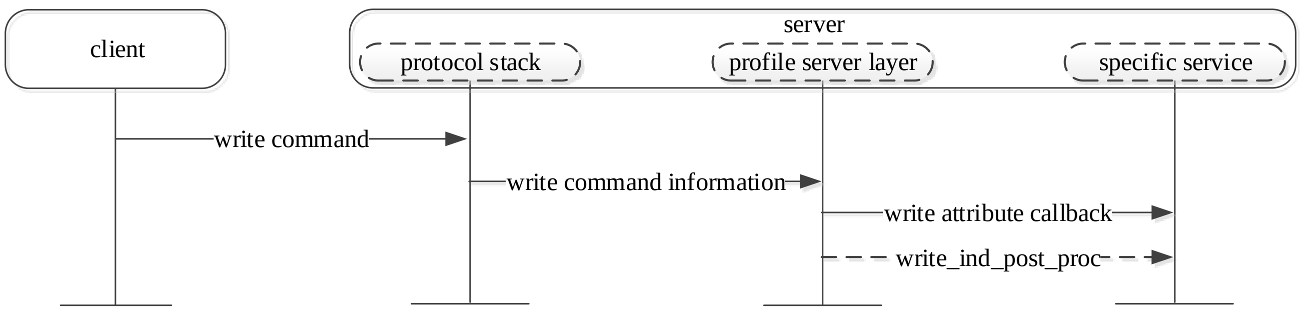

Figure 3-12 Write without Response - Attribute Value Supplied by Application

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 response write prepare write confirmation.

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

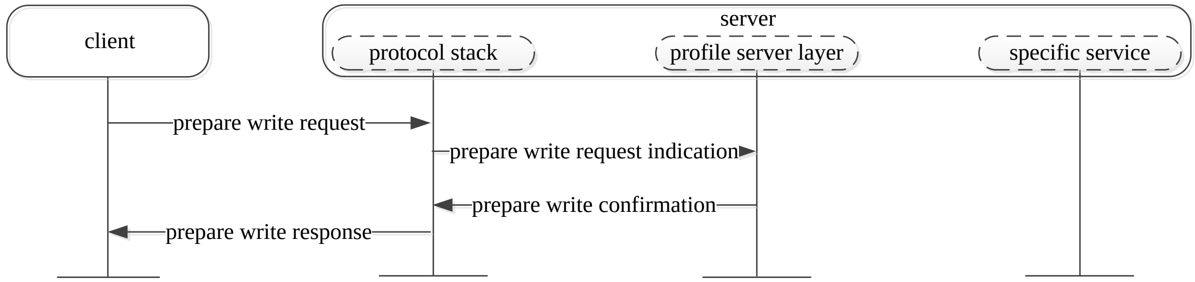

Figure 3-13 Write Long Characteristic Values – Prepare Write Procedure

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.

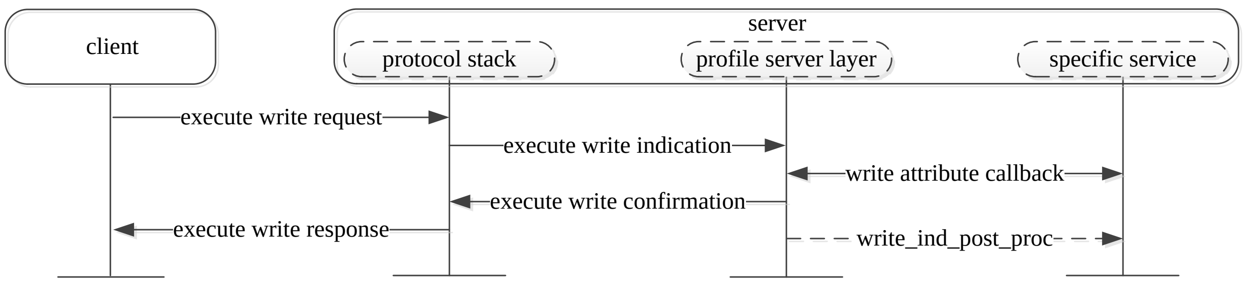

Figure 3-14 Write Long Characteristic Values– Execute Write without Result Pending

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.

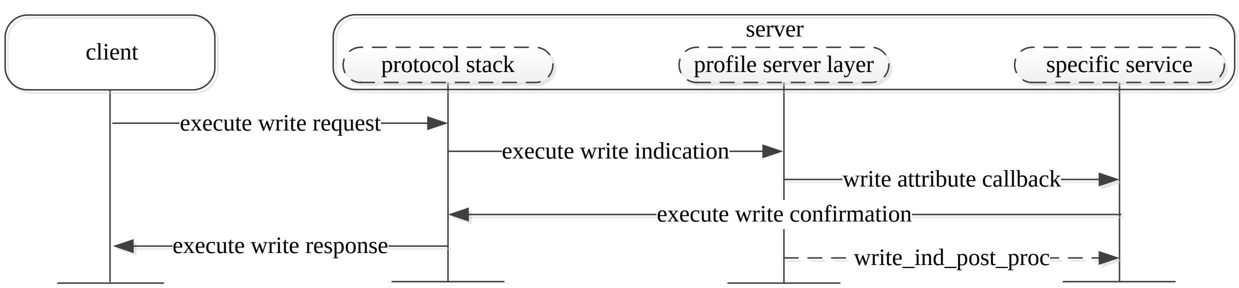

Figure 3-15 Write Long Characteristic Values– Execute Write with Result Pending

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.

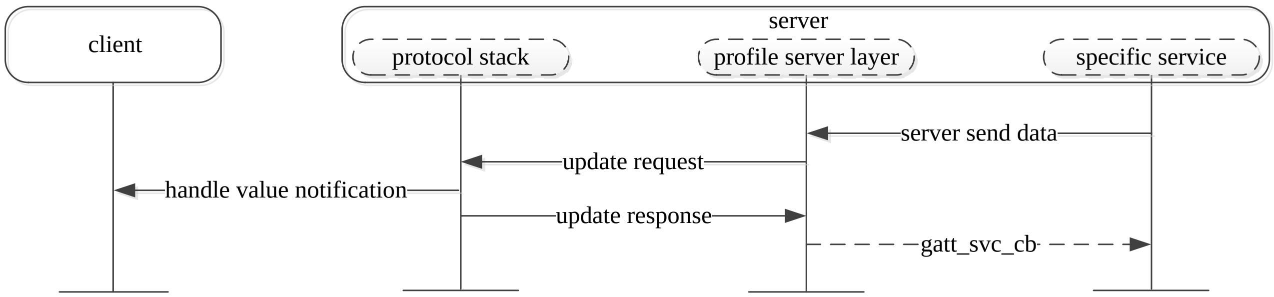

Figure 3-16 Characteristic Value Notification

```
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.

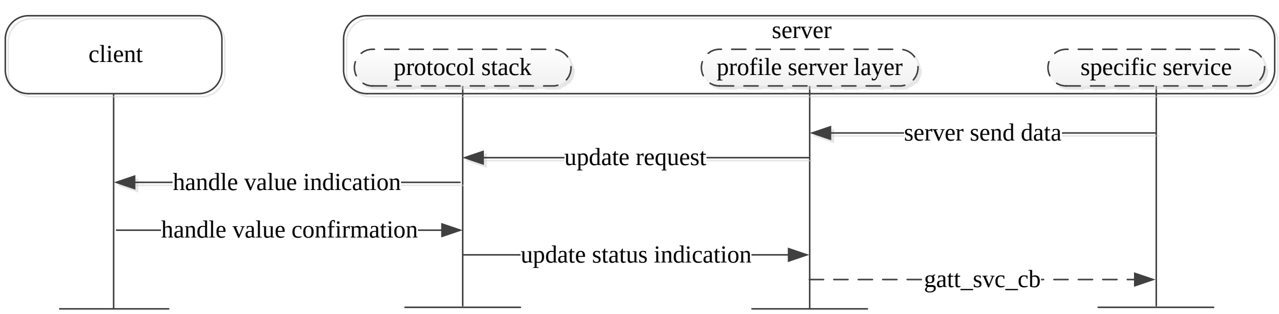

Figure 3-17 Characteristic Value Indication

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

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 read attribute process	**Actual length**	Reply by application	Reply by application	Strlen(value)	Strlen(value)+1	Strlen(value)	Strlen(value)+1
	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),	write

		Value not defined here	
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),
        },
  (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 BLE service as an example, we define a data with type TSIMP_CALLBACK_DATA to hold notification result.

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

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

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 BLE 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.

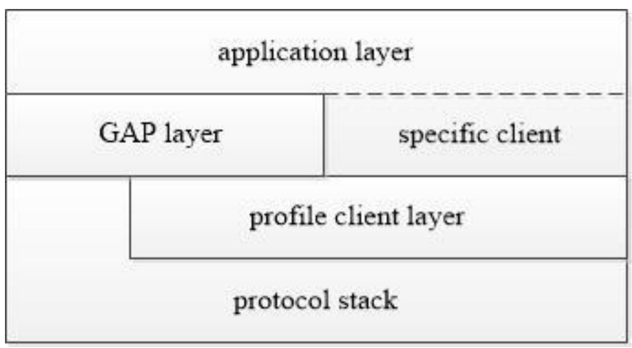

Figure 3-18 Profile Client Hierarchy

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

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.

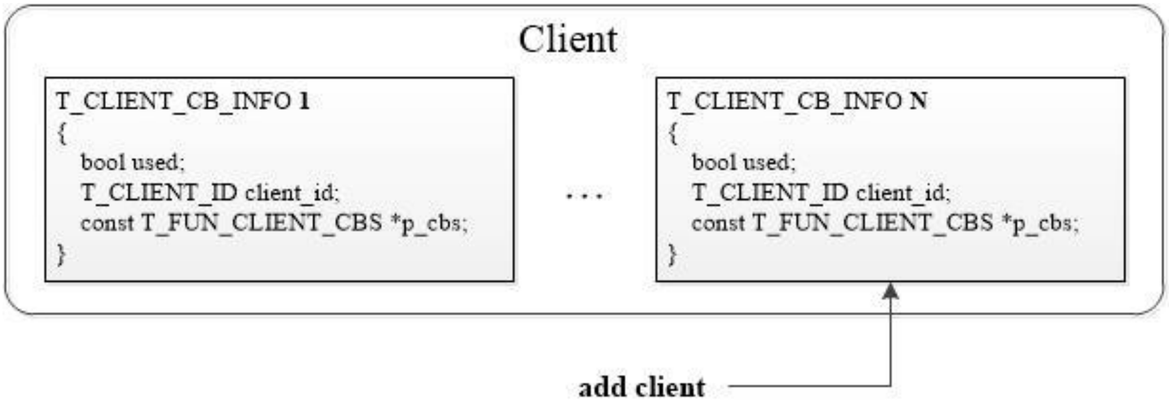

Figure 3-19 Add Specific Clients to Profile Client Layer

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.

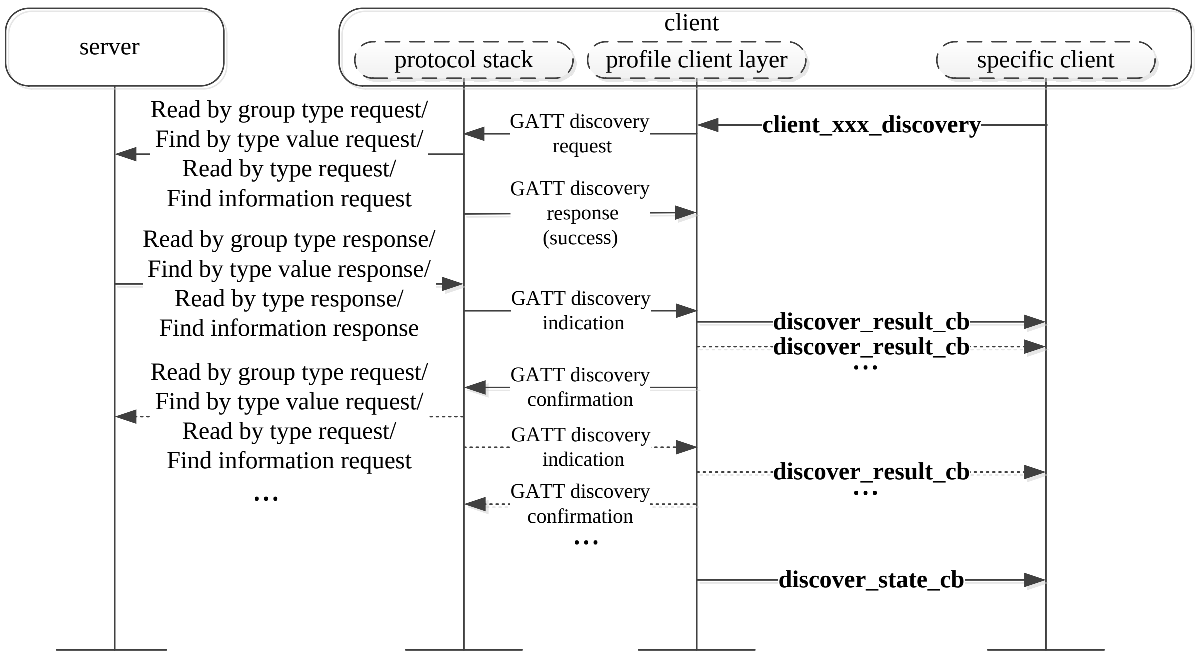

Figure 3-20 GATT Discovery Procedure

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.

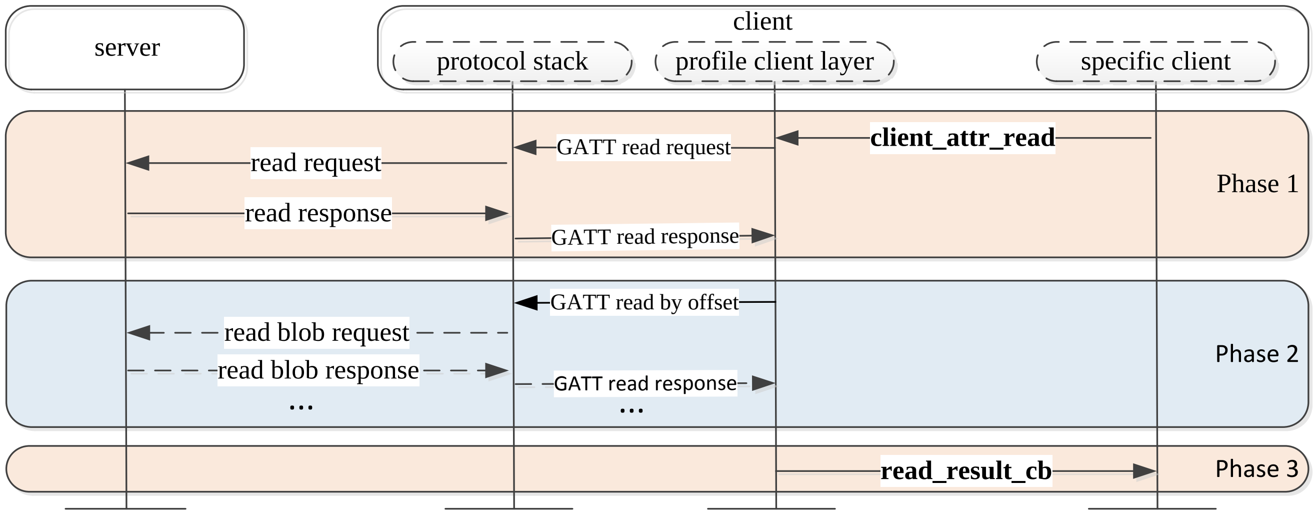

Figure 3-21 Read Characteristic Value by Handle

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.

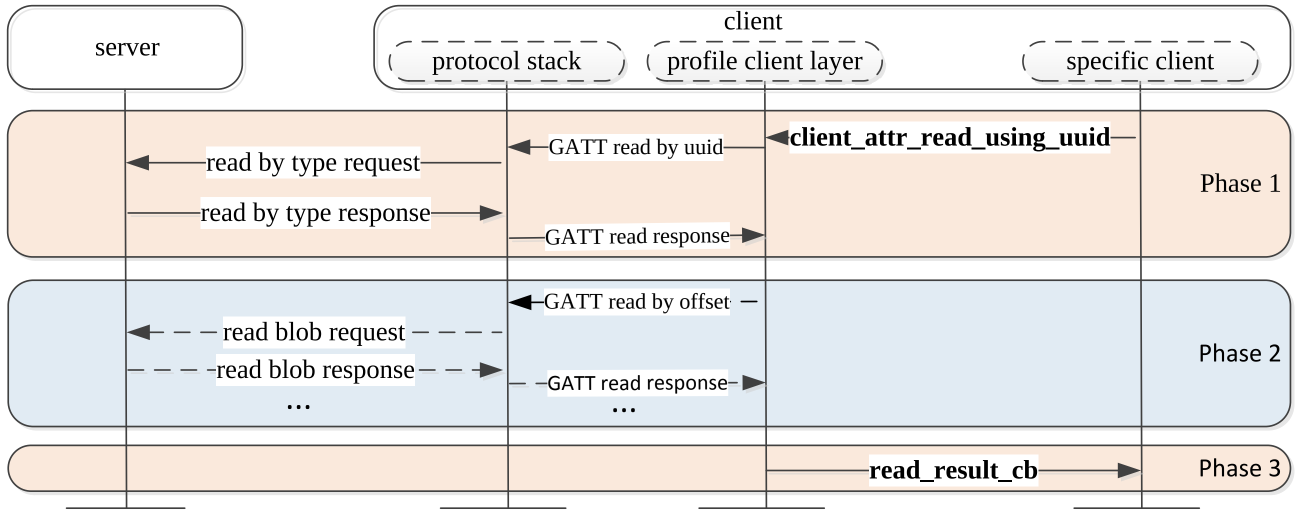

Figure 3-22 Read Characteristic Value by UUID

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

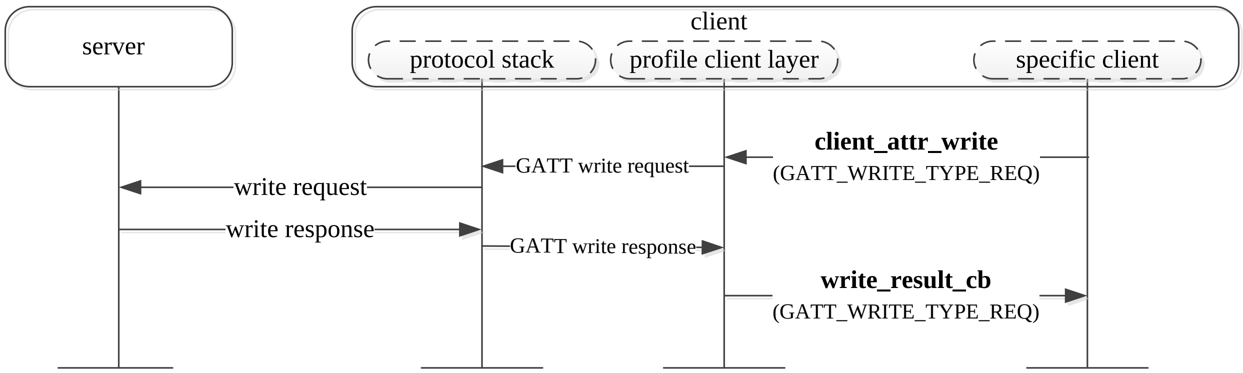

Figure 3-23 Write Characteristic Value

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

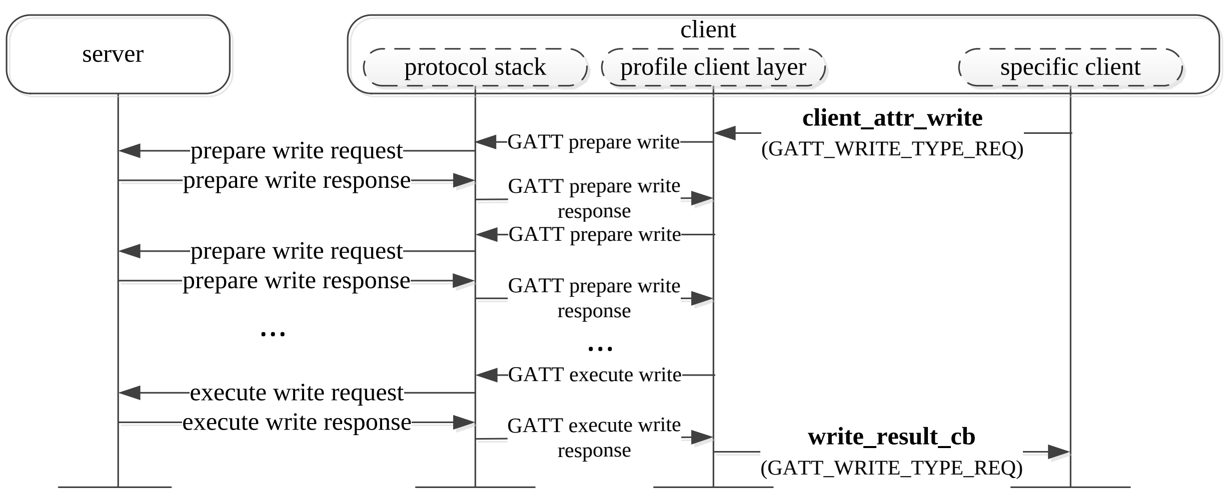

Figure 3-24 Write Long Characteristic Value

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

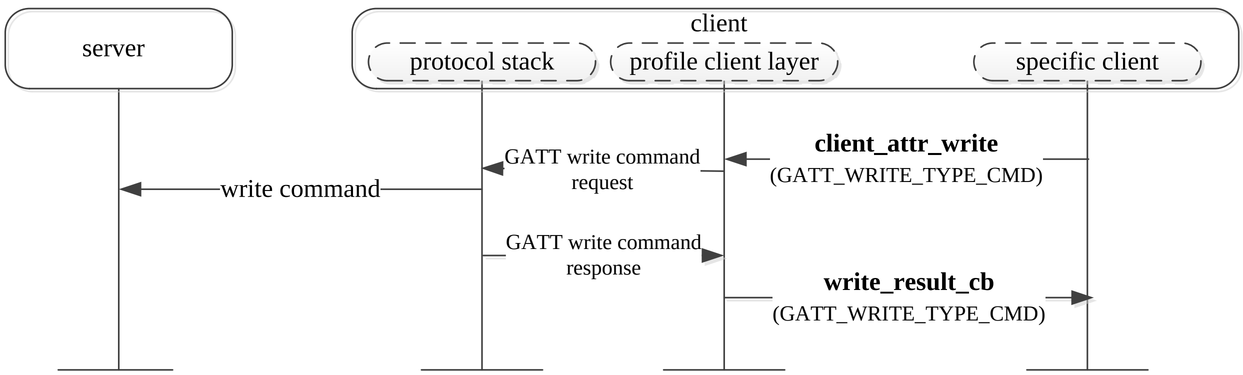

Figure 3-25 Write without Response

3.2.3.6 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 3-26.

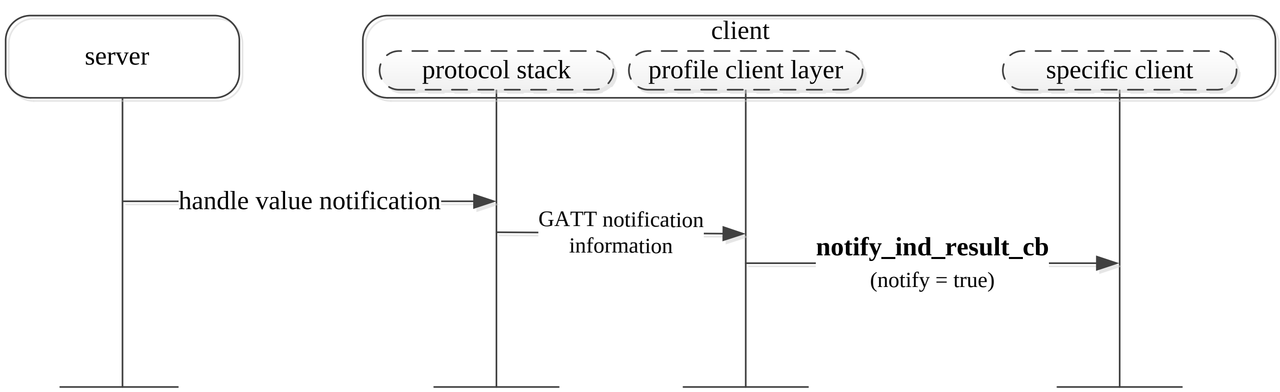

Figure 3-26 Characteristic Value Notification

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 3-27.

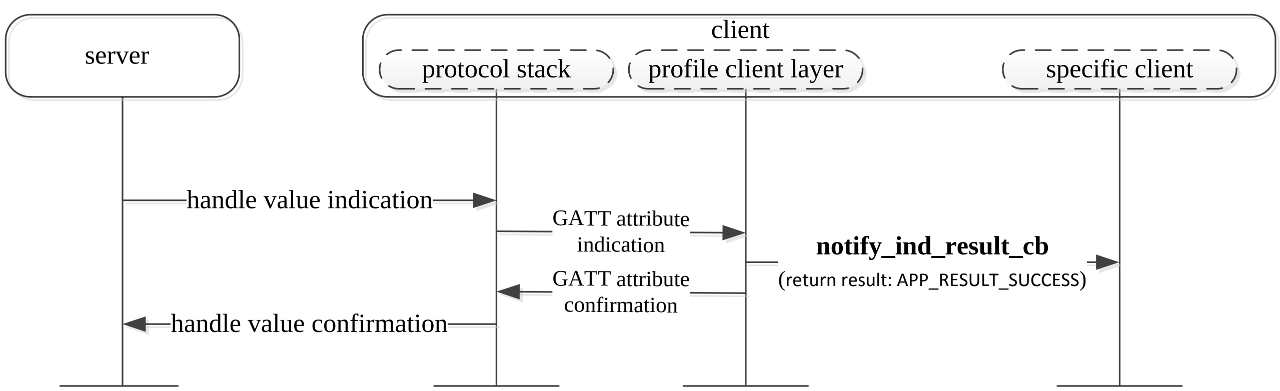

Figure 3-27 Characteristic Value Indication without Result Pending

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 3-28.

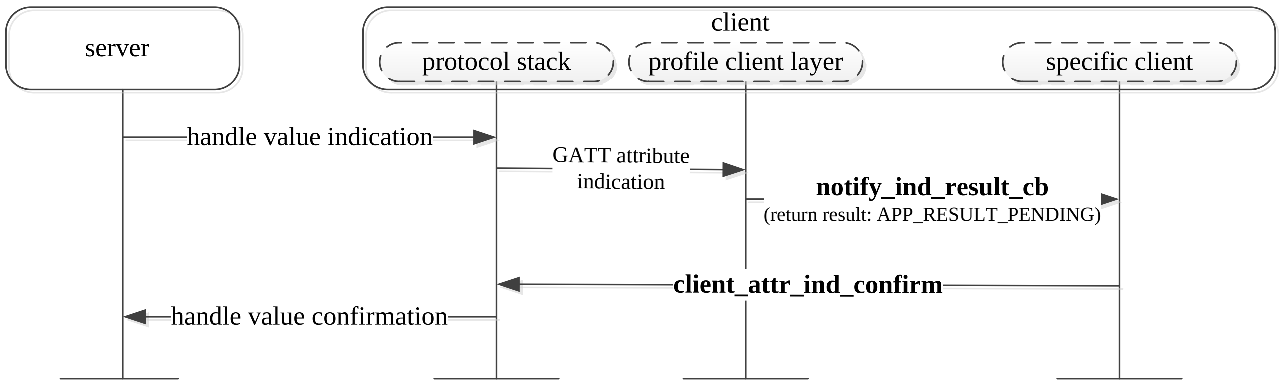

Figure 3-28 Characteristic Value Indication with Result Pending

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.

BT 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, BT 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 BT protocol stack. BT protocol stack can cache commands in number of credits.

- Credit count decreases by one when profile client layer sends command to BT protocol stack
- Credit count increases by one when profile client layer receives the response from BT

protocol stack. When the command is sent to air, BT 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);
}
```

4 BLE 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
2. Observer
    - Scan for advertising events
    - Cannot initiate connections
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*

4.1 BLE Peripheral Application

4.1.1 Introduction

The purpose of this chapter is to give an overview of the BLE peripheral application. The BLE peripheral project implements a simple BLE 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:

- Supported GATT services:
    - GAP and GATT Inbox Services
    - Battery Service
    - Simple BLE Service

4.1.2 Project Overview

This section describes project directory and project structure and which macro control the example. Reference content as follows:

- Project source code directory: component\common\bluetooth\realtek\sdk\example\ble_peripheral
- Controlled by macro: CONFIG_BT_PERIPHERAL

4.1.3 Source Code Overview

The following sections describe important parts of this application.

4.1.3.1 Initialization

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

```
int ble_app_main(void)
{
    bt_trace_init();
    bt_stack_config_init();
    bte_init();
    board_init();
    le_gap_init(APP_MAX_LINKS);
    app_le_gap_init();
    app_le_profile_init();
    pwr_mgr_init();
    task_init();
    return 0;
}
```

GAP and GATT Profiles initialization flow:

- bt_stack_config_init() - call gap_config_max_le_link_num(APP_MAX_LINKS), is the configuration of lib
- le_gap_init() - Initialize GAP and set link number
- 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.

4.1.3.2 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 *BLE 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 BLE 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();
            }
    ......
}
```

4.1.3.3 GAP Callback Handler

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

4.1.3.4 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 BLE Service

   simp_srv_id is the service id of simple BLE 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:
      ......
    }
  }
}
```

4.1.4 Relative AT CMD

4.1.4.1 Introduction

4.1.4.1.1 AT Command List

Description	AT Command
Stop / start BLE peripheral	ATBp

4.1.4.2 AT Command

4.1.4.2.1 ATBp - Stop or Start BLE Peripheral

ATBp=<start_or_stop>	
Description	This command is used to stop or start BLE peripheral.

Parameter	**start_or_stop**	0 means stop BLE peripheral, 1 means start BLE peripheral.
Remark	None.	

4.1.5 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 BLE 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.

4.1.5.1 Test with iOS Device

Procedure Description: iOS-based devices are always compatible with BLE, and devices running BLE Peripheral Application can be discovered in Set Bluetooth interface, but it is recommended to use BLE-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:

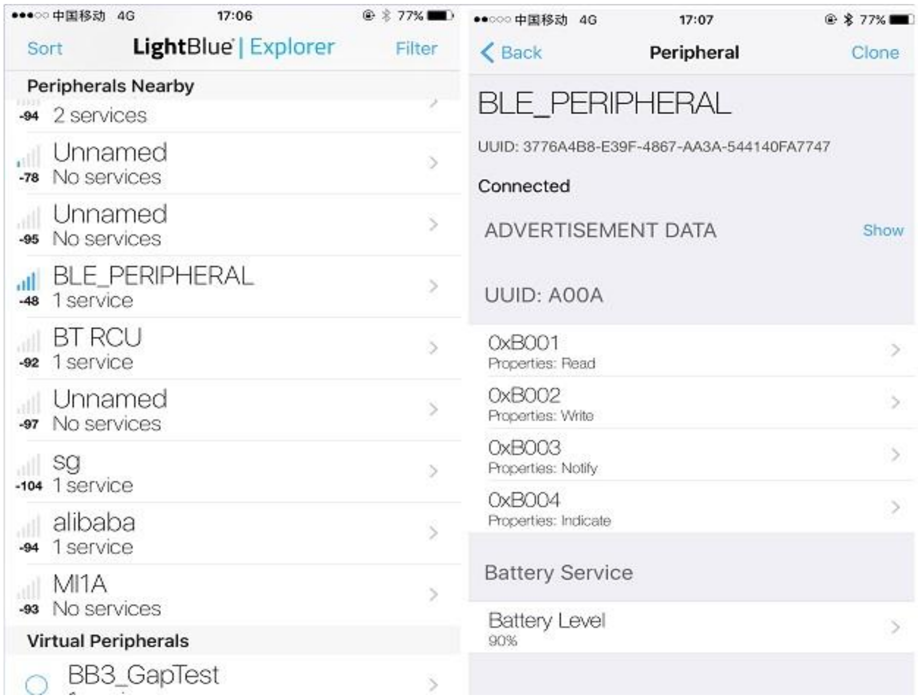

Figure 4-1 Test with iOS Device

4.2 BLE Central Application

4.2.1 Introduction

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

Central role features:

- Scan for advertising events
- Can initiate connection and become a master role in the link
- Discovering the services of a peripheral that you're connected to
- Discovering the Characteristics of a service
- Retrieving and writing the value of a characteristic

4.2.2 Project Overview

This section describes project directory and project structure and which macro control the example. Reference content as follows:

- Project source code directory: component\common\bluetooth\realtek\sdk\example\ble_central

- Controlled by macro: CONFIG_BT_CENTRAL

4.2.3 Source Code Overview

The following sections describe important parts of this application.

4.2.3.1 Initialization

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

```
int ble_central_app_main(void)
{
    bt_trace_init();
    ble_central_bt_stack_config_init ();
    le_gap_init(BLE_CENTRAL_APP_MAX_LINKS);
    ble_central_app_le_gap_init ();
    ble_central_app_le_profile_init ();
    ble_central_task_init ();

    return 0;
}
```

GAP and GATT Profiles initialization flow:

- ble_central_bt_stack_config_init () - call gap_config_max_le_link_num(BLE_CENTRAL_APP_MAX_LINKS), is the configuration of lib
- le_gap_init() - Initialize GAP
- ble_central_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
- ble_central_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*.

4.2.3.2 GAP Message Handler

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

```
void ble_central_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*/
      }
   }
   if (ble_central_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");
         data_uart_print("GAP scan stop\r\n");
      }
      else if (new_state.gap_scan_state == GAP_SCAN_STATE_SCANNING)
      {
         APP_PRINT_INFO0("GAP scan start");
         data_uart_print("GAP scan start\r\n");

      }
   }
}
```

Central application will memset ble_central_app_link_table according to conn_id when receives GAP_CONN_STATE_DISCONNECTED.

```
void ble_central_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_ERROR2("ble_central_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(&ble_central_app_link_table[conn_id], 0, sizeof(T_APP_LINK));
         }
         break;
   ......
}
```

4.2.3.3 GAP Callback Handler

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

4.2.3.4 Profile Message Callback

When APP uses gcs_add_client to register callbacks and add GATT client, APP shall register the callback function to handle the message from the gcs client.

ble_central_gcs_client_callback() function is the general callback function. app_profile_callback() can distinguish different client by client id.

```
void ble_central_app_le_profile_init (void)
{
   client_init(1);
   ble_central_gcs_client_id = gcs_add_client(ble_central_gcs_client_callback, BLE_CENTRAL_APP_MAX_LINKS,
BLE_CENTRAL_APP_MAX_DISCOV_TABLE_NUM);
}
```

1. General profile client callback

   ble_central_gcs_client_id is the general common services client id. Message used by profile client layer contains two message types:

   - GCS_CLIENT_CB_TYPE_DISC_RESULT : Discovery the service which exist in server.
   - GCS_CLIENT_CB_TYPE_READ_RESULT : This message is to inform the result of read attributes.
   - GCS_CLIENT_CB_TYPE_WRITE_RESULT: This message is to inform the result of read attributes.
   - GCS_CLIENT_CB_TYPE_NOTIF_IND: This message is to inform the notification from server.

```
T_APP_RESULT ble_central_gcs_client_callback (T_CLIENT_ID service_id, void *p_data)
{
   T_APP_RESULT  result = APP_RESULT_SUCCESS;
   APP_PRINT_INFO2("ble_central_gcs_client_callback: client_id %d, conn_id %d",
            client_id, conn_id);
   if (client_id == ble_central_gcs_client_id)
   {
      T_GCS_CLIENT_CB_DATA *p_gcs_cb_data = (T_GCS_CLIENT_CB_DATA *)p_data;
      switch (p_gcs_cb_data->cb_type)
      {
      case GCS_CLIENT_CB_TYPE_DISC_RESULT:
         ble_central_gcs_handle_discovery_result(conn_id, p_gcs_cb_data->cb_content.discov_result);
         break;
      case GCS_CLIENT_CB_TYPE_READ_RESULT:
         APP_PRINT_INFO3("READ RESULT: cause 0x%x, handle 0x%x,  value_len %d",
                  p_gcs_cb_data->cb_content.read_result.cause,
                  p_gcs_cb_data->cb_content.read_result.handle,
                  p_gcs_cb_data->cb_content.read_result.value_size);
         data_uart_print("READ RESULT: cause 0x%x, handle 0x%x,  value_len %d\n\r",
                  p_gcs_cb_data->cb_content.read_result.cause,
                  p_gcs_cb_data->cb_content.read_result.handle,
                  p_gcs_cb_data->cb_content.read_result.value_size);

         if (p_gcs_cb_data->cb_content.read_result.cause == GAP_SUCCESS)
         {
```

```
            APP_PRINT_INFO1("READ VALUE: %b",
                        TRACE_BINARY(p_gcs_cb_data->cb_content.read_result.value_size,
                                  p_gcs_cb_data->cb_content.read_result.p_value));
                                                data_uart_print("REAR VALUE:");
                                                for(int i=0; i< p_gcs_cb_data->cb_content.read_result.value_size; i++)
                                                            data_uart_print("0x%2X", *(p_gcs_cb_data-
>cb_content.read_result.p_value + i));
                                                data_uart_print("\n\r");
         }
         break;
      case GCS_CLIENT_CB_TYPE_WRITE_RESULT:
         APP_PRINT_INFO3("WRITE RESULT: cause 0x%x ,handle 0x%x, type %d",
p_gcs_cb_data->cb_content.write_result.cause,
                     p_gcs_cb_data->cb_content.write_result.handle,
                     p_gcs_cb_data->cb_content.write_result.type);
                                    data_uart_print("WRITE RESULT: cause 0x%x ,handle 0x%x, type %d",
                     p_gcs_cb_data->cb_content.write_result.cause,
                     p_gcs_cb_data->cb_content.write_result.handle,
                     p_gcs_cb_data->cb_content.write_result.type);
         break;
      case GCS_CLIENT_CB_TYPE_NOTIF_IND:
         if (p_gcs_cb_data->cb_content.notif_ind.notify == false)
         {
            APP_PRINT_INFO2("INDICATION: handle 0x%x, value_size %d",
                        p_gcs_cb_data->cb_content.notif_ind.handle,
                        p_gcs_cb_data->cb_content.notif_ind.value_size);
            APP_PRINT_INFO1("INDICATION VALUE: %b",
                        TRACE_BINARY(p_gcs_cb_data->cb_content.read_result.value_size,
                                  p_gcs_cb_data->cb_content.read_result.p_value));
         }
         else
         {
            APP_PRINT_INFO2("NOTIFICATION: handle 0x%x, value_size %d",
                        p_gcs_cb_data->cb_content.notif_ind.handle,
                        p_gcs_cb_data->cb_content.notif_ind.value_size);
            APP_PRINT_INFO1("NOTIFICATION VALUE: %b",
                        TRACE_BINARY(p_gcs_cb_data->cb_content.notif_ind.value_size,
                                  p_gcs_cb_data->cb_content.notif_ind.p_value));
         }
         break;
      default:
         break;
      }
   }

   return result;
}
```

4.2.4 Relative AT CMD

4.2.4.1 Introduction

4.2.4.1.1 AT Command list

Description	AT Command
Stop / start BLE Central	ATBc
Scan BT	ATBS
Create a GATT connection	ATBC
Disconnect GATT Connection	ATBD
Reply GAP passkey	ATBK
Get peripheral information	ATBG
Get information of connected device	ATBI
Reply GAP user confirm	ATBY
Update connection request	ATBU
Get / clear bond information	ATBO
GATT client read	ATBR
GATT client write	ATBW
Modify white list	ATBn

4.2.4.2 AT Command

4.2.4.2.1 ATBc - Stop or Start BLE Central

ATBc=<start_stop>	
Description	This command is used to stop or start BLE central.

Parameter	**start_stop**	0 means stop BLE central, 1 means start BLE central.
Remark	None.	

4.2.4.2.2 ATBS - Scan BT

ATBS=<scan_enable>,<filter_policy>,<filter_duplicate>		
Description	This command is used to scan BT devices.	
Parameter	**scan_enable**	1 means start scan, 0 means stop scan, use a decimal value to express.
	filter_policy	0 means any, 1 means white list, 2 means any RPA, 3 means white list RPA, use a decimal value to express.
	filter_duplicate	0 means disable, 1 means enable, use a decimal value to express.
Remark	1. If the parameter of scan_enable is 0, then the cmd is express as ATBS=0. 2. The value of parameter filter_policy are 0 or 1.	

4.2.4.2.3 ATBC - Create a GATT Connection

ATBC=P/R,BLE_BD_ADDR		
Description	This command is used to connect to remote device.	
Parameter	**P/R**	bt addr type, can get from scan result, P means public, R means random.
	BLE_BD_ADDR	bt addr.
Remark	If the bt address of device is 00:11:22:33:44:0b, BLE_BD_ADDR should be 00112233440b.	

4.2.4.2.4 ATBD - Disconnect GATT Connection

ATBD=conn_id		
Description	This command is used to disconnect to remote device.	
Parameter	**conn_id**	use a decimal value to express.
Remark	None.	

4.2.4.2.5 ATBK - Reply GAP Passkey

ATBK=MODE,auth_flags,io_cap,sec_enable,oob_enable		
Description	This command is used to config authentication mode.	
Parameter	auth_flags	bit0-(bonding), bit2-(MITM), bit3-(SC), use a hexadecimal value to express.
	io_cap	io Capabilities: 0-(display only), 1-(display yes/no), 2-(keyboard noly), 3-(no IO), 4-(keyboard display), use a decimal value to express.
	sec_enable	start smp pairing procedure when connected: 0-(disable), 1-(enable), use a decimal value to express.
	oob_enable	enable oob flag: 0-(disable), 1-(enable), use a decimal value to express.
Remark	If the bt address of device is 00:11:22:33:44:0b, BLE_BD_ADDR should be 00112233440b.	

ATBK=KEY,conn_id,passcode		
Description	This command is used to Input passkey when show GAP_MSG_LE_BOND_PASSKEY_INPUT.	
Parameter	**conn_id**	use a decimal value to express.
	passcode	0 – 999999.

Remark	None.

ATBK=SEND,conn_id		
Description	This command is used to send authentication request.	
Parameter	**conn_id**	use a decimal value to express.
Remark	None.	

4.2.4.2.6 ATBG - Get Peripheral Information

ATBG=ALL,connect_id		
Description	This command is used to get all services.	
Parameter	**conn_id**	use a decimal value to express.
Remark	None.	

ATBG=SRV,connect_id,uuid_type,uuid		
Description	This command is used to discover services by uuid.	
Parameter	conn_id	use a decimal value to express.
	uuid_type	0(uuid is 16-bit), 1(uuid is 128-bit), use a decimal value to express.
	uuid	

Remark	For a 16-bit uuid (0x1800), the value of parameter uuid is 1800, a similar relationship for a 128-bit uuid.

ATBG=CHARDIS,connect_id,start_handle,end_handle		
Description	This command is used to discover characteristic.	
Parameter	**conn_id**	use a decimal value to express.
	start_handle	use a hexadecimal value to express.
	end_handle	use a hexadecimal value to express.
Remark	None.	

ATBG=CHARUUID,connect_id,start_handle,end_handle,type,uuid		
Description	This command is used to discover characteristic by uuid.	
Parameter	**conn_id**	use a decimal value to express.
	start_handle	use a hexadecimal value to express.
	end_handle	use a hexadecimal value to express.
	type	0(uuid is 16-bit), 1(uuid is 128-bit), use a decimal value to express.
	uuid	
Remark	For a 16-bit uuid (0x1800), the value of parameter uuid is 1800, a similar relationship for a 128-bit uuid.	

ATBG=CHARDDIS,connect_id,start_handle,end_handle		
Description	This command is used to discover characteristic descriptor.	
Parameter	**conn_id**	use a decimal value to express.
	start_handle	use a hexadecimal value to express.
	end_handle	use a hexadecimal value to express.

Remark	None.

4.2.4.2.7 ATBI - Get Information of Connected Device

ATBI	
Description	This command is used to get information of connected device.
Parameter	None.
Remark	None.

4.2.4.2.8 ATBY - Reply GAP User Confirm

ATBY=conn_id,conf		
Description	This command is used to send user confirmation when show GAP_MSG_LE_BOND_USER_CONFIRMATION.	
Parameter	**conn_id**	use a decimal value to express.
	conf	0-(Reject), 1(Accept), use a decimal value to express.
Remark	None.	

4.2.4.2.9 ATBU - Update Connection Request

ATBU=conn_id,interval_min,interval_max,latency,supervision_timeout	
Description	This command is used to send Connection parameter update request.

Parameter	**conn_id**	use a decimal value to express.
	interval_min	the minimum time interval at the beginning of two consecutive connection events. Range:0x0006 – 0x0C80 (7.5ms – 4000ms, 1.25ms/step), use a hexadecimal value to express.
	interval_max	the maximum time interval at the beginning of two consecutive connection events. Range:0x0006 – 0x0C80(7.5ms – 4000ms,1.25ms/step), use a hexadecimal value to express.
	latency	slave latency for the connection in number of connection events. Range:0x0000 – 0x01F3, use a hexadecimal value to express.
	supervision_timeout	supervision timeout for LE link. Range:0x000A – 0x0C80 (100ms – 32000ms, 10ms/step), use a hexadecimal value to express.
Remark	None.	

4.2.4.2.10 ATBO - Get or Clear Bond Information

ATBO=INFO	
Description	This command is used to get bond information.
Parameter	None.
Remark	None.

ATBO=CLEAR	
Description	This command is used to clear bond information.
Parameter	None.
Remark	None.

4.2.4.2.11 ATBR - GATT Client Read

ATBR=conn_id,handle		
Description	This command is used to read characteristic.	
Parameter	**conn_id**	use a decimal value to express.
	handle	handle of the characteristic, use a hexadecimal value to express.
Remark	Before use this cmd to read characteristic, we should use cmd ATBG=CHARDIS,0,0x1,0xFFFF to get the handle(value handle) of this readable characteristic..	

ATBR=conn_id,start_handle,end_handle,uuid_type,uuid		
Description	This command is used to read characteristic value by uuid.	
Parameter	**conn_id**	use a decimal value to express.
	start_handle	use a hexadecimal value to express.
	end_handle	use a hexadecimal value to express.
	uuid_type	0(uuid is 16-bit), 1(uuid is 128-bit), use a decimal value to express.
	uuid	
Remark	1. For a 16-bit uuid (0x1800), the value of parameter uuid is 1800, a similar relationship for a 128-bit uuid. 2. Before use this cmd to read characteristic, we should use cmd ATBG=CHARDIS,0,0x1,0xFFFF to get the uuid of this readable characteristic.	

4.2.4.2.12 ATBW - GATT Client Write

ATBW=conn_id,type,handle,length,value		
Description	This command is used to write data to service.	
Parameter	**conn_id**	use a decimal value to express.
	type	1-(write request), 2-(write command), use a decimal value to express.
	handle	attribute handle, use a hexadecimal value to express
	length	value length, use a hexadecimal value to express
	value	use a hexadecimal value to express
Remark	Before use this cmd to write data to service, wo should use cmd ATBG=CHARDIS,0,0x1,0xFFFF to get the handle(value handle) of this writable Characteristic.	

4.2.4.2.13 ATBn - Modify White List

ATBn=type,P/R,BLE_BD_ADDR		
Description	This command is used to add/remove a device to/from the white list.	
Parameter	**type**	0 means clear white list, 1 means add a device to the white list, 2 means remove a device from the white list, use a decimal value to express.
	P/R	bt addr type, can get from scan result, P means public, R means random.
	BLE_BD_ADDR	bt addr.
Remark	1. If the bt address of device is 00:11:22:33:44:0b, BLE_BD_ADDR should be 00112233440b. 2. If **type** = 0, the parameter of **P/R** and **BLE_BD_ADDR** are ignored.	

4.2.5 Test Procedure

At first, please build and download the BLE Central application to the Evolution Board. 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.

When BLE Central Application is being run on Evolution Board, the BLE device can scan the peripheral device and create connection with peripheral device and so on.

4.2.5.1 Scan Test

Procedure:

1. Start scan by use command ATBS=1.
2. Stop scan by ATBS=0.

Result:

```
ATBS=1
[BLE_PRINT] Start scan
[BLE_PRINT] scan_filter_policy = 0, scan_filter_duplicate=1

[MEM] After do cmd, available heap 142272
#
GAP scan start
ADVType                 | AddrType       |BT_Addr         |rssi
CON_UNDIRECT            public     00:11:22:33:44:0b  -46
GAP_ADTYPE_FLAGS: 0x5
GAP_ADTYPE_16BIT_XXX: 0xa00a
GAP_ADTYPE_APPEARANCE: 0
GAP_ADTYPE_LOCAL_NAME_XXX: Bee_perip

ATBS=0
```

```
[BLE_PRINT] Start stop scan
```

4.2.5.2 Connect Test

Procedure:

1. Connect to device by use command ATBC=P, e000874c1f21.

Result:

```
ATBC=P,e000874c1f21
cmd_con, DestAddr: 0x21:0x1F:0x4C:0x87:0x00:0xE0

[MEM] After do cmd, available heap 142272
#
Connected success conn_id 0
```

4.2.5.3 Disconnect Test

Procedure:

1. Disconnect to device by use command ATBD=0.

Result:

```
ATBD=0

[MEM] After do cmd, available heap 142272

#
Disconnect conn_id 0
```

4.2.5.4 Pair Test

Procedure:

1. Clear all binding information by use command ATBO=CLEAR.
2. Connect to device by use command ATBC=P, e000874c1f21.
3. Start pair by use command ATBK=SEND, 0.

Result:

```
ATBK=SEND,0

[MEM] After do cmd, available heap 142272

#
Pair success
```

4.2.5.5 Get Service Test

Procedure:

1. Discover characteristic by use command ATBG=CHARDIS, 0, 0x1, 0xFFFF.

Result:

```
ATBG=CHARDIS,0,0x1,0xFFFF

[MEM] After do cmd, available heap 142272
#
conn_id 0, GCS_ALL_CHAR_DISCOV, is_success 1
CHAR UUID16[1]: decl_handle 0x4, properties 0x2, value_handle 0x5, uuid16 0x2a01
properties:indicate 0, read 2, write cmd 0, write 0, notify 0
CHAR UUID16[4]: decl_handle 0xb, properties 0x8, value_handle 0xc, uuid16 0xb002
properties:indicate 0, read 0, write cmd 0, write 8, notify 0
```

4.2.5.6 Read Test

Procedure:

1. Read characteristic value by uuid by use command ATBR=0,0x1,0xFFFF,0,2a01.

Result:

```
ATBR=0,0x1,0xFFFF,0,2a01
conn_id = 0, start_handle = 0x1, end_handle = 0xffff, uuid_type = 0
uuid16 = 0x2a01

[MEM] After do cmd, available heap 142272

#
READ RESULT: cause 0x00000000, handle 0x00000005,  value_len 1
REAR VALUE:0x00
```

4.2.5.7 Write Test

Procedure:

1. Write data to serivce by use command ATBW=0,1,0xc,0x1,0x1.

Result:

```
ATBW=0,1,0xc,0x1,0x1

[MEM] After do cmd, available heap 142272

#
WRITE RESULT: cause 0x00000000 ,handle 0x0000000c, type 1
```

4.2.5.8 Update Connect Request Test

Procedure:

1. Update connection param request by use command ATBU=0,0x30,0x40,0x1,0x1F4.

Result:

```
ATBU=0,0x30,0x40,0x1,0x1F4

[MEM] After do cmd, available heap 142272
#

ble_central_app_handle_conn_param_update_evt update pending: conn_id 0

ble_central_app_handle_conn_param_update_evt update success:conn_id 0, conn_interval 0x0000003c,
conn_slave_latency 0x00000001, conn_supervision_timeout 0x000001f4
```

4.2.5.9 Modify White List

Procedure:

1. Add a device to the white list by use command ATBn=1, R, 6f90bb90553c.
2. Remove a device from the white list by use command ATBn=2, R, 6f90bb90553c.
3. Clear white list by use command ATBn=0.

Result:

```
ATBn=1,R,6f90bb90553c
cmd_modify, DestAddr: 0x3C:0x55:0x90:0xBB:0x90:0x6F
[MEM] After do cmd, available heap 139976
#
GAP_MSG_LE_MODIFY_WHITE_LIST: operation  0x00000001, cause 0x00000000


ATBn=2,R,6f90bb90553c
cmd_modify, DestAddr: 0x3C:0x55:0x90:0xBB:0x90:0x6F
[MEM] After do cmd, available heap 139976
#
GAP_MSG_LE_MODIFY_WHITE_LIST: operation  0x00000002, cause 0x00000000


ATBn=0
cmd_modify, DestAddr: 0x00:0x00:0x00:0x00:0x00:0x00
[MEM] After do cmd, available heap 139976
#
GAP_MSG_LE_MODIFY_WHITE_LIST: operation  0x00000000, cause 0x00000000
```

4.3 BT Config Application

4.3.1 Introduction

The purpose of this chapter is to give an overview of the BT Config application.

4.3.2 Project Overview

This section describes project directory and project structure and which macro control the example. Reference content as follows:

- Project source code directory: component\common\bluetooth\realtek\sdk\example\bt_config

Controlled by macro: CONFIG_BT_CONFIG

4.3.3 Relative AT CMD

4.3.3.1 Introduction

4.3.3.1.1 AT Command List

Description	AT Command
Start / stop BT config	ATBB

4.3.3.2 AT Command

4.3.3.2.1 ATBB - Stop or Start BT Config

ATBB=<start_or_stop>		
Description	This command is used to stop or start BT Config.	
Parameter	**start_or_stop**	0 means stop BT Config, 1 means start BT Config.
Remark	None.	

4.3.4 Relative APIs

4.3.4.1 Introduction

4.3.4.1.1 API list

Description	API
Start BT config	int bt_config_app_init(void)
Stop BT Config	void bt_config_app_deinit(void)

4.3.5 Test Procedure

Please refer to the document **Android_BTConfigAPP_User_Guide_v2.1.pdf**.

4.4 BT Beacon Application

4.4.1 Introduction

The purpose of this chapter is to give an overview of the Beacon application.

4.4.2 Project Overview

This section describes project directory and project structure and which macro control the example. Reference content as follows:

- Project source code directory: component\common\bluetooth\realtek\sdk\example\bt_beacon

Controlled by macro: CONFIG_BT_BEACON

- Beacon application is divided into two categories: I_BEACON and ALT_BEACON

4.4.3 Relative AT CMD

4.4.3.1 Introduction

4.4.3.1.1 AT Command list

Description	AT Command
Start / stop BT Beacon	ATBJ

4.4.3.2 AT Command

4.4.3.2.1 ATBJ - Stop or Start BT Beacon

ATBJ=<start_or_stop>,<beacon_type>		
Description	This command is used to stop or start BT Beacon.	
Parameter	**start_or_stop**	0 means stop BT Beacon, 1 means start BT Beacon.
	beacon_type	1 means I_Beacon , 2 means BT Alt_Beacon ,use a decimal value to express.
Remark	None.	

4.4.4 Relative APIs

4.4.4.1 Introduction

4.4.4.1.1 API list

Description	API
Start Beacon	int bt_beacon_app_init(int type)
Stop Beacon	void bt_beacon_app_deinit(void)

4.4.5 Test Procedure

At first, please build and download the Beacon application to the Evolution Board. Some basic functions of Beacon Application are demonstrated above.

When Beacon Application is being run on Evolution Board, the BLE device becomes discoverable. Remote device can scan the Beacon device.

4.4.5.1 Test with Andriod Device

Procedure Description: Andriod-based devices are always compatible with Beacon, and devices running Beacon Application can be discovered in App (e.g. nRF Connect) to perform search test.

Procedure: Run nRF Connect on Andriod device to search for iBeacon and AltBeacon device, as shown in Figure 4-2 and Figure 4-3:

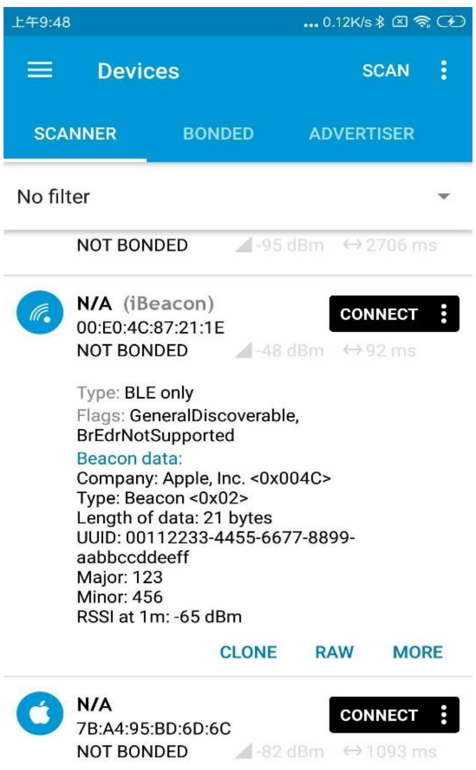

Figure 4-2 Test with Android Device (iBeacon)

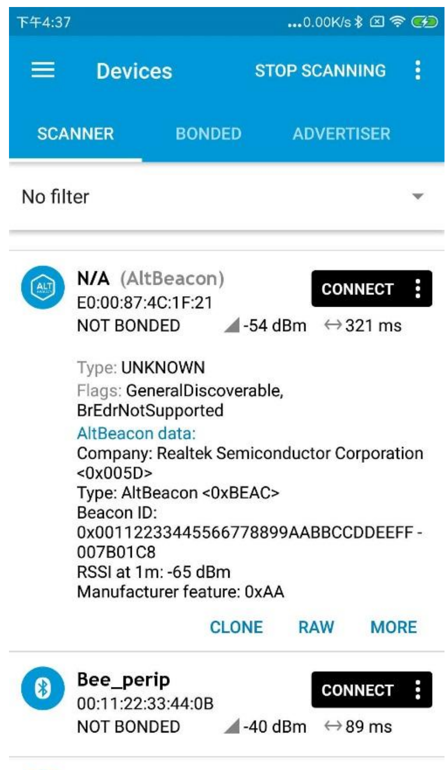

Figure 4-3 Test with Android Device (AltBeacon)

4.5 BT Airsync Config Application

4.5.1 Introduction

The purpose of this chapter is to give an overview of the BT Airsync Config application.

4.5.2 Project Overview

This section describes project directory and project structure and which macro control the example. Reference content as follows:

- Project source code directory: component\common\bluetooth\realtek\sdk\example\bt_airsync_config

Controlled by macro: CONFIG_BT_AIRSYNC_CONFIG

4.5.3 Relative AT CMD

4.5.3.1 Introduction

4.5.3.1.1 AT Command list

Description	AT Command
Stop / start BT Airsync config	ATBb

4.5.3.2 AT Command

4.5.3.2.1 ATBb - Stop or Start BT Airsync Config

ATBb=<start_or_stop>		
Description	This command is used to stop or start BT Airsync Config.	
Parameter	**start_or_stop**	0 means stop BT Airsync Config, 1 means start BT Airsync Config.
Remark	None.	

4.5.4 Relative APIs

4.5.4.1 Introduction

4.5.4.1.1 API list

Description	API
Start BT Airsync Config	int bt_airsync_config_app_init (void)
Stop BT Airsync Config	void bt_airsync_config_app_deinit (void)

4.6 BLE Scatternet Application

4.6.1 Introduction

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

4.6.2 Project Overview

This section describes project directory and project structure and which macro control the example. Reference content as follows:

- Project source code directory:component\common\bluetooth\realtek\sdk\example\ble_scatternet

Controlled by macro: CONFIG_BT_SCATTERNET & CONFIG_BT_CENTRAL & CONFIG_BT_PERIPHERAL

4.6.3 Relative AT CMD

4.6.3.1 Introduction

4.5.3.1.1 AT Command list

Description	AT Command
Stop / start BLE Scatternet	ATBf
set BLE PHY(2M)	ATBg

4.6.3.2 AT Command

The relative AT command of BLE Scatternet (The coexistence of BLE central mode and BLE peripheral mode) can refer to chapter 4.1.4 and chapter 4.2.4, additional AT command is as follows:

4.6.3.2.1 ATBf - Stop or Start BLE Scatternet

ATBf=<start_or_stop>		
Description	This command is used to stop or start BLE Scatternet.	
Parameter	**start_or_stop**	0 means stop BLE Scatternet, 1 means start BLE Scatternet.
Remark	None.	

4.6.3.2.2 ATBg - Set BLE PHY (2M)

ATBg=conn_id,phy		
Description	This command is used to set BLE PHY (2M).	
Parameter	**conn_id**	use a decimal value to express.
	phy	use a decimal value to express.
Remark	1. This command only support the value of parameter **phy**=1. 2. AmebaZ2 not support this command.	

4.6.4 Test Procedure

At first, please build and download the BLE Scatternet application to the Evolution Board. Some basic functions of BLE Scatternet 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 Scatternet Application is being run on Evolution Board, the BLE device can scan the peripheral device and create connection with peripheral device and so on (as central role) , remote device can scan the BLE device and create connection with the BLE device, after disconnection, BLE Application will restore to be discoverable and connectable again(as peripheral role).

BLE Scatternet is the coexistence of peripheral mode and central mode. The test procedure of relative AT command of BLE Scatternet can refer to chapter 4.1.5 and chapter 4.2.5, additional test procedure of new AT command or features is as follows.

4.6.4.1 Modify Device Name

Procedure: Run LightBlue APP on Andriod device to search for and connect a BLE_Scatternet device, in the interface of success connect of LightBlue APP, choose "Generic Access"-->"Device Name". Below the "WRITTEN VALUES", input data (for example input ABCDE), and click the button of "WRITE", then stop ble scatternnet mode by use cmd ATBf=0 and restart ble scatternet mode by use cmd ATBf=1, run LightBlue on Andriod device to search for and connect a BLE_Scatternet device again, in the interface of success connect of LightBlue APP, below the "READ/INDICATED VALUES", click the button of "READ AGAIN", the result show the modifided device name is 0ABCDE00.

4.6.4.2 Modify Apperance

Procedure: Run LightBlue APP on Andriod device to search for and connect a BLE_Scatternet device, in the interface of success connect of LightBlue APP, choose "Generic Access"-->"Appearance". Below the "WRITTEN VALUES", input data(the length of data should be 2 byte)(for example input ABCD),and click the button of "WRITE", then stop ble scatternnet mode by use cmd ATBf=0 and restart ble scatternet mode by use cmd ATBf=1, run LightBlue on Andriod device to search for and connect a BLE_Scatternet device again, in the interface of success connect of LightBlue APP, below the "READ/INDICATED VALUES", click the button of "READ AGAIN", the result show the modifided apperance is ABCD.

4.6.4.3 Set PHY 2M Test

Procedure:

1. Connect to device by use command ATBC=P, e000874c1f21.
2. Set PHY 2M by use cmd ATBg=0, 1.

Result:

```
ATBg=0,1

[MEM] After do cmd, available heap 142272

#
GAP_MSG_LE_PHY_UPDATE_INFO:conn_id 0, cause 0x00000000, rx_phy 2, tx_phy 2
```

5 How to Build and Run BT Example

AmebaD:

Please refer to document **AN0400 Ameba-D Application Note v10.pdf.**

AmebaZ2:

Please refer to document **AN0500 Realtek Ameba-ZII application note.en.docx.**

6 Q&A

6.1 How to Configure BLE Service with 128 Bit UUID

Modify service table as follow to configure BLE service with 128 bit UUID.

```
const uint8_t   GATT_UUID128_CUSTOMIZED_PRIMARY_SERVICE [16] =
{0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF, 0x00};
#define GATT_UUID128_CUSTOMIZED_CHAR   0x01, 0x23, 0x45, 0x67, 0x89, 0x0A, 0xBC, 0xDE, 0xFF, 0x11, 0x22, 0x33,
0x44, 0x55, 0x66, 0x77,

static const T_ATTRIB_APPL customized_UUID128_service_tbl[] =
{
	{
		(ATTRIB_FLAG_VOID | ATTRIB_FLAG_LE),
		{
		   LO_WORD(GATT_UUID_PRIMARY_SERVICE),
		   HI_WORD(GATT_UUID_PRIMARY_SERVICE),
		},
		UUID_128BIT_SIZE,
		(void *) GATT_UUID128_CUSTOMIZED_PRIMARY_SERVICE,
		GATT_PERM_READ
	},

	{
		ATTRIB_FLAG_VALUE_INCL,
		{
		   LO_WORD(GATT_UUID_CHARACTERISTIC),
		   HI_WORD(GATT_UUID_CHARACTERISTIC),
		   GATT_CHAR_PROP_READ | GATT_CHAR_PROP_WRITE,
		},
		1,
		NULL,
		GATT_PERM_READ
	},

	{
		ATTRIB_FLAG_VALUE_APPL | ATTRIB_FLAG_UUID_128BIT,
		{
		   GATT_UUID128_CUSTOMIZED_CHAR
		},
		0,
		NULL,
		GATT_PERM_READ | GATT_PERM_WRITE
	},

};
```

6.2 MTU Size

Ameba has support the feature of MTU exchange. If there is an MTU exchange, Ameba will communicate with the peer device in 247 byte.
In general, the MTU exchange is initiated by the Master (GATT Client, such as a mobile phone). (If Slave initiate the MTU exchange, please refer to component\common\bluetooth\realtek\sdk\example\ble_peripheral\ble_app_main.c)

```
uint8_t  slave_init_mtu_req = true;
le_set_gap_param(GAP_PARAM_SLAVE_INIT_GATT_MTU_REQ, sizeof(slave_init_mtu_req),  &slave_init_mtu_req);
```

If neither Master (GATT Client) nor Slave (GATT Server) initiate MTU request, then according to the definition of BT spec, both parties will default to MTU=23.
If one of the parties has initiated an MTU request, Ameba will do the negotiation with 247 bytes.
Example:
Ameba: 247, Peer device: 100 -> result: MTU = 100
Ameba: 247, Peer device: 300 -> result: MTU = 247

6.3 How to Open BT Upper Stack Log

AmebaD:
Please refer to document **AN0400 Ameba-D Application Note v10.pdf.**
AmebaZ2
Please refer to document **AN0500 Realtek Ameba-ZII application note.en.docx.**
For how to grab the BT log, please open document:

BTSDK log user guide_SOP_for_cu

6.4 The Limitation of Use BT

There are corresponding restrictions on the use of BT:
AmebaD:
Please refer to document **AN0400 Ameba-D Application Note v10.pdf.**
AmebaZ2
Please refer to document **AN0500 Realtek Ameba-ZII application note.en.docx.**

6.5 How to Modify the White List

The type to modify white list include:

- Clear white list

- Remove a device from the white list
- Add a device to the white list

The relative API is as follows:

```
T_GAP_CAUSE le_modify_white_list(T_GAP_WHITE_LIST_OP operation, uint8_t *bd_addr, T_GAP_REMOTE_ADDR_TYPE
bd_type);
```

Parameter Description:

1. *operation* – the type to operation white list, defined in gap_le_types.h.
    - GAP_WHITE_LIST_OP_CLEAR: Clear white list.
    - GAP_WHITE_LIST_OP_ADD: Add a device to the white list.
    - GAP_WHITE_LIST_OP_REMOVE: Remove a device from the white list.
2. *bd_addr* - the bluetooth device address.
3. *bd_type* - the bluetooth device address type, defined in gap.h.
    - GAP_REMOTE_ADDR_LE_PUBLIC: LE Public device address type.
    - GAP_REMOTE_ADDR_LE_RANDOM: LE Random device address type.
4. *retval* - the return value of this function.
    - GAP_CAUSE_SUCCESS: Operation success.
    - GAP_CAUSE_SEND_REQ_FAILED: Operation failure.

The example code of clear white list operation is as follows:

```
void clear_white_list_example()
{
   T_GAP_WHITE_LIST_OP operation = GAP_WHITE_LIST_OP_CLEAR;
   le_modify_white_list(operation, NULL, GAP_REMOTE_ADDR_LE_PUBLIC);
}
```

The example code of add a device to the white list operation is as follows:

```
void add_device_to_white_list_example()
{
   T_GAP_WHITE_LIST_OP operation = GAP_WHITE_LIST_OP_ADD;
   uint8_t bd_addr[6] = {0xD2,0xB7,0xA7,0x55,0xF7,0xE8};
   T_GAP_REMOTE_ADDR_TYPE bd_type = GAP_REMOTE_ADDR_LE_RANDOM;
   le_modify_white_list(operation, bd_addr, bd_type);
}
```

The example code of remove a device from the white list operation is as follows:

```
void remove_device_from_white_list_example()
{
   T_GAP_WHITE_LIST_OP operation = GAP_WHITE_LIST_OP_REMOVE;
   uint8_t bd_addr[6] = {0xD2,0xB7,0xA7,0x55,0xF7,0xE8};
   T_GAP_REMOTE_ADDR_TYPE bd_type = GAP_REMOTE_ADDR_LE_RANDOM;
   le_modify_white_list(operation, bd_addr, bd_type);
}
```

If you want to make the white list take effect during BLE scan,you should do the following operation before call the API of le_scan_start():

```
int  ble_central_at_cmd_scan ()
{
   ……
   u8 scan_filter_policy = GAP_SCAN_FILTER_WHITE_LIST;
```

```
  le_scan_set_param(GAP_PARAM_SCAN_FILTER_POLICY, sizeof(scan_filter_policy),&scan_filter_policy);
  ......
}
```

Note: The maximum number of devices in the white list supported by AmebaZ2 and AmebaD is 32.

6.6 How to Modify ADV Data and Scan RSP Data

AmebaZ2 and AmebaD has not support the feature of modify BT advertising data and scan rsp data dynamic in the case of BT advertising enable. If you want to modify BT advertising data and scan rsp data, first, you must stop advertising, then modify BT advertising data and scan rsp data, finally, restart advertising.

6.7 How to Set BLE Peripheral Example to Encrypted Transmission

Here are three methods to set BLE peripheral example to encrypted transmission:

(1) Configuration GAP_PARAM_BOND_SEC_REQ_ENABLE

The parameter of GAP_PARAM_BOND_SEC_REQ_ENABLE determines whether the security process is started when the connection is successfully established. If this parameter is set to true, the GAP layer will automatically start the security process when the connection is successfully 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) Call function le_bond_pair()

APP can call le_bond_pair() to start the security process. The APP can call le_bond_pair() when the LE link status 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;
    }
}
```

(3) Service security

The GATT Profile procedure is used to access information that may require the client to be authenticated or have an encrypted connection to read or write to the characteristic.

If the service has a security requirement, the client sends a request for a read or write operation when the physical link is unauthenticated or unencrypted, then the server must send an Error Response. If the client want to perform read or write operation, it need to use the GAP authentication procedure to authenticate the physical link.

The Attribute Element is the basic unit of the service, and its structure 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 sample code for the authentication requirements of Characteristic is as follows.

```
/* 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 */
},
```

For IOS systems, it does not provide an interface to start the security process, if the slave device wants to pair with the IOS device, the slave device needs to request pairing. For IOS device, it support BLE peripehral device use method (2) and method (3) to start the security process. it is recommended that the IOS device use method (3).

For Android systems, it support BLE peripheral device use method (1) (2) (3) to start the security process, however, in practical applications, IOT problems may occur using method (1) and method (2). On the other hand, Andriod device can initiate pairings, so it is recommended that the Android device initiate pairing actively.

The example for IOS and Andriod device to start security process:

- For IOS device

Procedure Description: After the BLE peripheral service security is added according to the method (3), run LightBlue on iOS device to scan and connect the BLE peripheral device,, the IOS device will receive the Bluetooth pairing request sent by the BLE peripheral, as shown in Figure 6-1, then click to pair Button, the serial port of the BLE peripheral device will print the log "pair success".

Figure 6-1 Test with IOS Device

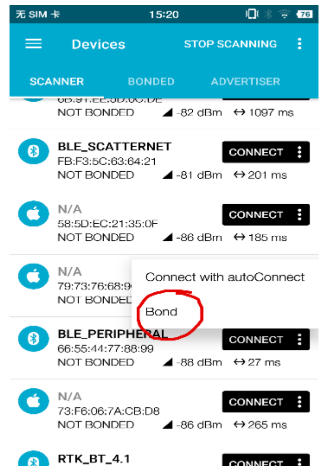

Figure 6-2 Test with Andriod Device

- For Andriod device

Andriod device can initiate pairings, So we don't need to use any of the above methods for the BLE peripheral device.

Procedure Description: Run nRF Connect on Andriod device to scan the BLE peripheral device, click the Bond button directly after scanning, as shown in Figure 6-2, then the Andriod device will connect to the BLE peripheral and Pairing, the serial port of the BLE peripheral will print the log "pair success".

6.8 How to Modify AttHandle

If you want to modify the AttHandle of an attribute, you need to use the API server_add_service_by_start_handle to specify the start_handle when creating the service. Note that the start_handle is preferably greater than or equal to 0x14, because the stack will also register some services internally.

The sample code for call API server_add_service_by_start_handle is as follows.

```
T_SERVER_ID bas_add_service(void *p_func)
{
   T_SERVER_ID service_id;
          if (false == server_add_service_by_start_handle(&service_id,
                                                          (uint8_t *)bas_attr_tbl,
                                                          bas_attr_tbl_size,
                                                          bas_cbs, 0x20))
   {
      APP_PRINT_ERROR1("bas_add_service: service_id %d", service_id);
      service_id = 0xff;
   }
   pfn_bas_cb = (P_FUN_SERVER_GENERAL_CB)p_func;
```

```
    return service_id;
}
```

6.9 How to Set Random Device Address

If you want to set bluetooth device address to random device address, you can refer to the follow steps:
1) Load static random address information from storage or generate a local random address
2) Call the API le_cfg_local_identity_address to config local identity address
3) Call the API le_set_gap_param to set the random device address, note that the first parameter of le_set_gap_param should be set to GAP_PARAM_RANDOM_ADDR
4) If the gap role of the device is Peripheral, you should call the API le_adv_set_param to set the type of bluetooth device address being used in advertisement, note that the first parameter of le_adv_set_param should be set to GAP_PARAM_ADV_LOCAL_ADDR_TYPE; if the gap role of the device is central, you should call the API le_scan_set_param to set the type of bluetooth device address being used in scan request packets, note that the first parameter of le_scan_set_param should be set to GAP_PARAM_SCAN_LOCAL_ADDR_TYPE.
The sample code for setting random device address is as follows. The sample code is already included in the source code of ble scatternet example.

```
T_APP_STATIC_RANDOM_ADDR random_addr;
bool gen_addr = true;
uint8_t local_bd_type = GAP_LOCAL_ADDR_LE_RANDOM;
if (ble_scatternet_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;
        ble_scatternet_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);
```