GR551x Bluetooth Low Energy Stack User Guide

Version: 1.8

Release Date: 2021-08-20

Shenzhen Goodix Technology Co., Ltd.

Trademarks and Permissions

GOODiX and other Goodix trademarks are trademarks of Shenzhen Goodix Technology Co., Ltd. All other trademarks and trade names mentioned in this document are the property of their respective holders.

Disclaimer

Information contained in this document is intended for your convenience only and is subject to change without prior notice. It is your responsibility to ensure its application complies with technical specifications.

Shenzhen Goodix Technology Co., Ltd. (hereafter referred to as “Goodix”) makes no representation or guarantee for this information, express or implied, oral or written, statutory or otherwise, including but not limited to representation or guarantee for its application, quality, performance, merchantability or fitness for a particular purpose. Goodix shall assume no responsibility for this information and relevant consequences arising out of the use of such information.

Without written consent of Goodix, it is prohibited to use Goodix products as critical components in any life support system. Under the protection of Goodix intellectual property rights, no license may be transferred implicitly or by any other means.

Shenzhen Goodix Technology Co., Ltd.

Headquarters: 2F. & 13F., Tower B, Tengfei Industrial Building, Futian Free Trade Zone, Shenzhen, China

TEL: +86-755-33338828 FAX: +86-755-33338099

Website: www.goodix.com

Preface

Purpose

This document introduces the layers and basic layer functions of a GR551x Bluetooth Low Energy (Bluetooth LE) Protocol Stack. It discusses how applications interact with the protocol stack using APIs on the stack, aiming to enable developers to efficiently use the APIs in developing Bluetooth LE applications.

Audience

This document is intended for:

- GR551x user
- GR551x developer
- GR551x tester
- Technical writer

Release Notes

This document is the sixth release of *GR551x Bluetooth Low Energy Stack User Guide*, corresponding to GR551x SoC series.

Revision History

Version	Date	Description
1.0	2019-12-08	Initial release
1.3	2020-03-16	Updated the release time in the footers.
1.5	2020-05-30	• Modified description on GAP_ADV_PROP_SCANNABLE_BIT in the prop parameter in "Enable Extended Advertising". • Deleted APIs related to initializing Generic Access Service in "Generic Access Service".
1.6	2020-06-30	Updated the document version based on SDK changes.
1.7	2021-05-10	Updated SoC model descriptions.
1.8	2021-08-20	Modified SoC model descriptions.

Contents

1 Introduction

The Bluetooth LE software architecture of a GR551x SoC comprises the Application/Profile layer, a Software Development Kit (SDK), and a BLE Protocol Stack (BLE Stack), as shown in the figure below.

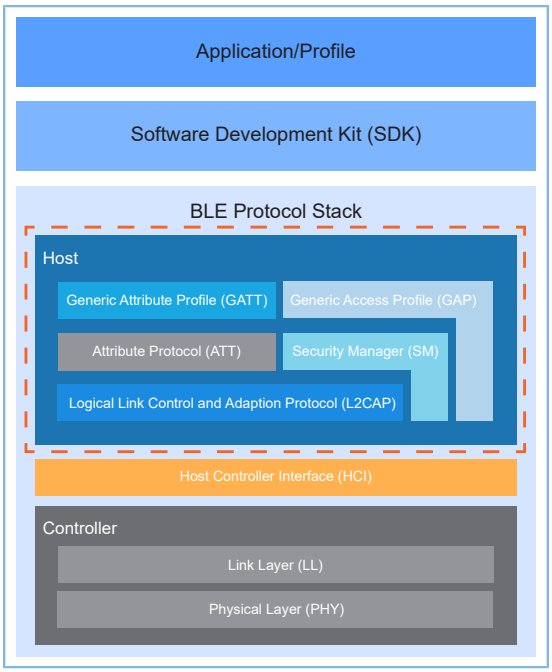

Figure 1-1 GR551x Bluetooth LE software architecture

In this architecture, the Application/Profile layer interacts with the BLE Stack through APIs provided by the SDK. Developers can invoke APIs of GAP, GATT, SM, and L2CAP layers on the BLE Stack during application development.

This document focuses on the composition and functions of the BLE Stack as well as how user applications interact with the BLE Stack. In addition, the document introduces code examples contained in the GR551x SDK to help developers better understand the BLE Stack. Codes are stored in `SDK_Folder\projects\ble\ble_basic_example\`. SDK_Folder is the root directory of GR551x SDK in use.

Tip:

For more information about Bluetooth LE technologies and protocols, visit the official Bluetooth SIG website: www.bluetooth.com. Specifications of GAP, GATT, SM, and L2CAP are provided in *Bluetooth Core Spec*. Specifications of other Bluetooth LE profiles are available on the *GATT Specs* page. Assigned numbers, identifiers, and code which may be used by Bluetooth LE applications are listed on the *Assigned Numbers* page.

---

2 Generic Access Profile (GAP)

The Generic Access Profile (GAP) defines how Bluetooth devices discover others and how to establish secure and insecure connections.

2.1 GAP Roles

GAP defines four device roles: Broadcaster, Observer, Central, and Peripheral with details shown in Figure 2-1.

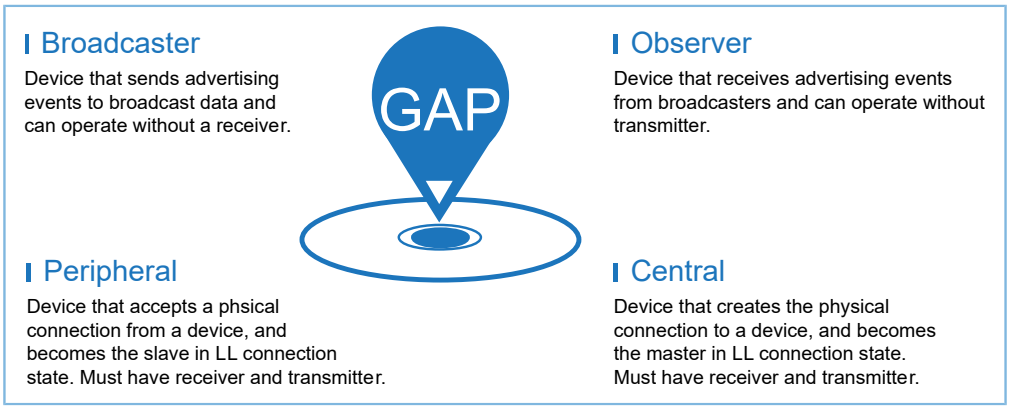

Figure 2-1 GAP-defined device roles

- Broadcaster: A device that sends advertising events to broadcast data and that can operate without a receiver
- Observer: A device that receives advertising events from broadcasters and that can operate without a transmitter
- Central: A device that creates physical connections to a device and that becomes the master in Link Layer (LL) connection state. Central devices must operate in scenarios with both a receiver and a transmitter.
- Peripheral: A device that accepts physical connections from a device and that becomes the slave in LL connection state. Peripherals must operate in scenarios with both a receiver and a transmitter.

Note:

A device supports one or more GAP roles at the same time. For example, a device can be a broadcaster and a peripheral at the same time.

2.2 GAP Modes and Procedures

GAP provides multiple access modes and device procedures: discover devices, establish a connection, terminate a connection, and configure device parameters.

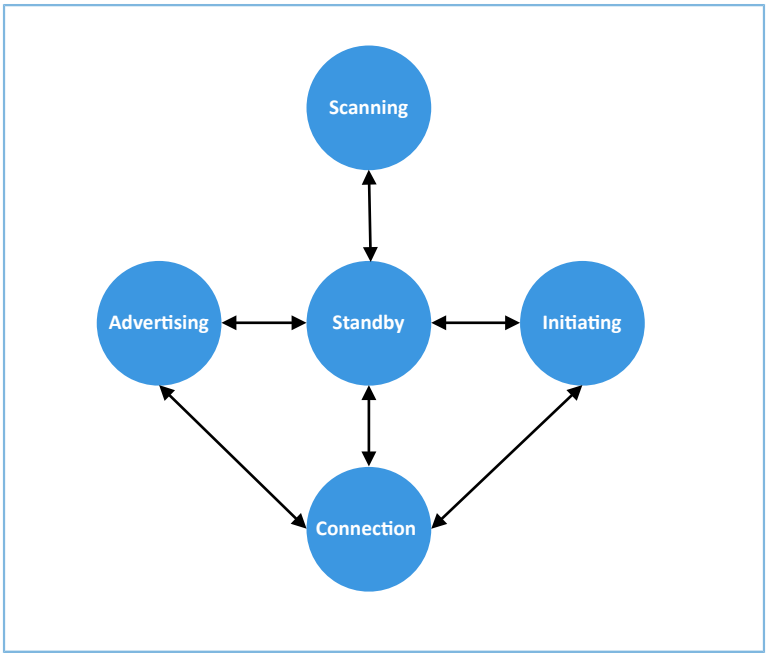

Figure 2-2 State diagram of the LL state machine

Based on the roles the device is configured, the device is in different states. Figure 2-2 shows the the conversion between different states at the Link Layer. Details of each state are as follows:

- Standby State: The device is in the initial standby state upon powering on.
- Advertising State: The device is advertising specified data, allowing the initiating device to discover the advertising device. The advertising data contains the advertising device address and other information (such as device name).
- Scanning State: The device is receiving advertising data and sends scanning requests to the advertiser. After receiving the scanning request, the advertiser replies to the scanning device with scan response data. This process is known as device discovery in Bluetooth communications.
- Initiating State: The device in initiating state must specify a peer device address to which to connect. If the received advertiser address matches with the specified one, the initiating device (initiator) sends a connection request to the advertiser. The connection request packet contains some specified connection parameters (for details, see "Section 2.3.2 Connection Parameters").
- Connection State: When a connection is established, the advertiser functions as a slave, and the initiator as a master. Both the devices switch their states to connection state.

2.3 Connection

2.3.1 Connection Event

A connection event is the start of data packets that are sent from the master to the slave and back again.

As illustrated in Figure 2-3, each connection event is a connection interval apart. Each connection event starts with a single packet from the master, and can continue until either the master or slave stops responding. At times between connection events, no packets from the master to this slave or the other way around are involved.

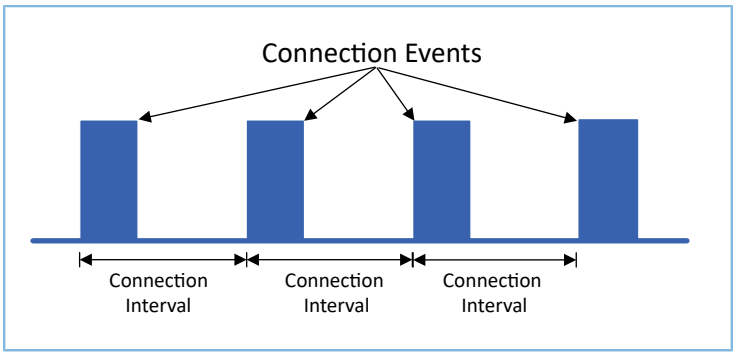

Figure 2-3 Connection events

2.3.2 Connection Parameters

Connection request packets sent by an initiator (master) contain connection parameters, and both the initiator and advertiser (slave) can modify the connection parameters after the connection is established.

Description of connection parameters:

- Connection interval

  The connection interval determines how frequently the master interacts with the slave and lasts from the start of the last connection event to the start of the next connection event.

  The connection interval can be any period lasting from 7.5 milliseconds to 4 seconds in multiples of 1.25 milliseconds.

- Slave latency

  The slave latency refers to the number of connection events that a slave (peripheral) can ignore. The slave is allowed to ignore a certain number of connection events when the slave does not need to send any data. This means the slave does not need to reply to the data packets sent by the master during these connection events, helping save power for the slave. The number of connection events that a slave can ignore should not be greater than the preset value of slave latency.

  The slave latency ranges from 0 to 499.

- Supervision timeout

  The supervision timeout refers to the maximum time periods between two successful connection events. If no connection event succeeds during the supervision timeout, the connection between the master and the slave terminates.

  The supervision timeout can be any period lasting from 100 milliseconds to 32 seconds in multiples of 10 milliseconds.

Note:

A successful connection between a master and a slave must be based on the following formula: Supervision timeout > (1 + Slave latency) x (Connection interval) x 2

The configurations on connection interval, slave latency, and supervision timeout affect the communications rate and power consumption between a master and a slave.

- A shorter connection interval means a shorter period for sending data, resulting in more frequent communications between a master and a slave and higher power consumption.
- A longer connection interval means a longer period for sending data, resulting in less frequent communications between a master and a slave and lower power consumption.
- If the slave latency is set to zero, the slave needs to respond to data packets sent by the master for each connection event. This results in a higher data transmission speed and higher power consumption.
- A longer slave latency slows the data transmission speed down and reduces power consumption.

2.3.3 Connection Parameter Update

When a master connects to a slave, the master sends connection parameters in a connection request packet. The connection parameters may no longer be suitable for the current application after the connection stays active for a period of time. Therefore, the master needs to send a connection parameter update request to the slave, or notifies the slave of updated connection parameters without negotiation.

In addition, the slave may need to update connection parameters during a connection based on requirements from Bluetooth LE applications. In this case, the slave sends a connection parameter update request to the master. For Bluetooth 4.1 compatible devices, connection parameter update requests are handled at the Link Layer. For Bluetooth 4.0 devices, the requests are handled at the L2CAP layer. The BLE Stack automatically selects an update method.

For both connection parameter update requests sent by the master and by the slave, only the master is allowed to send update notifications to apply the updates.

Note:

For details about connection parameter update, see “Connection update (Vol 6, Part D)” and “ Connection parameters request (Vol 6, Part D)” in *Bluetooth Core Spec*.

2.3.4 Connection Termination

Connection termination means disconnecting a link during which both the master and the slave switch their states from connection to standby. Both a master and a slave can initiate a connection termination notification message (LL_TERMINATE_IND). When the initiator receives the acknowledgement of the peer device (LL_ACK), both the devices disconnect from each other. The detailed termination procedures are illustrated in Figure 2-4.

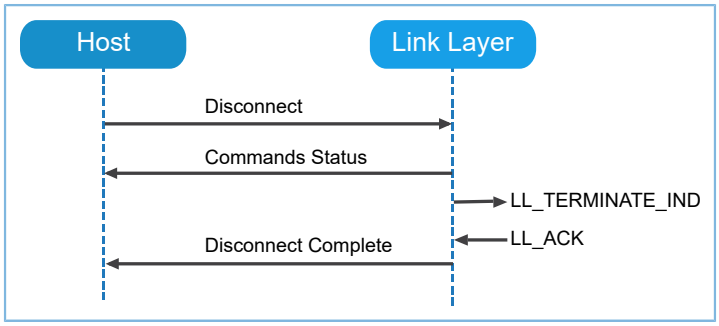

Figure 2-4 Connection termination

Moreover, a connection can be terminated due to a supervision timeout or a Message Integrity Check (MIC) failure.

2.4 Fundamental GAP Procedures

This section introduces the basic operating procedures of GAP using peripheral and central as examples.

2.4.1 Peripheral

Peripherals support legacy advertising, extended advertising, and periodic advertising.

2.4.1.1 Enable Legacy Advertising

When legacy advertising on a peripheral is enabled, interactions between the Bluetooth LE applications and BLE Stack are shown in Figure 2-5 .

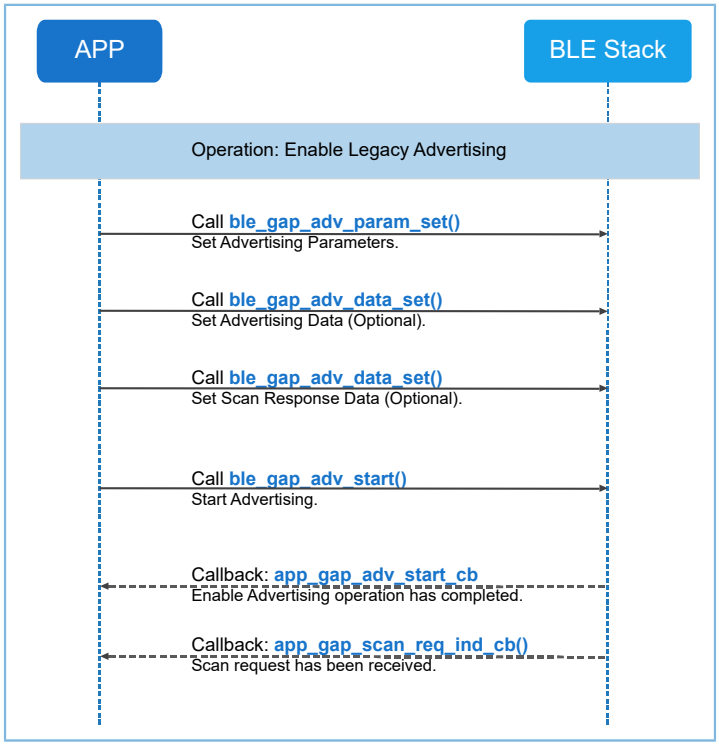

Figure 2-5 App-BLE-Stack interactions when legacy advertising is enabled

Follow the steps below to enable legacy advertising on a peripheral:

Note:

Code snippets in the steps below are extracted from the legacy advertising example: ble_app_gap_legacy_adv (SDK_Folder\projects\ble\ble_basic_example\).

1. Set advertising parameters.

```
s_gap_adv_param.adv_intv_max = APP_ADV_MAX_INTERVAL;
s_gap_adv_param.adv_intv_min = APP_ADV_MIN_INTERVAL;
s_gap_adv_param.adv_mode = GAP_ADV_TYPE_ADV_IND;
s_gap_adv_param.chnl_map = GAP_ADV_CHANNEL_37_38_39;
s_gap_adv_param.disc_mode = GAP_DISC_MODE_NON_DISCOVERABLE;
s_gap_adv_param.filter_pol = GAP_ADV_ALLOW_SCAN_ANY_CON_ANY;
error_code = ble_gap_adv_param_set(0, BLE_GAP_OWN_ADDR_STATIC,
                                   &s_gap_adv_param);
APP_ERROR_CHECK(error_code);
```

Note:

- The discoverability mode of directed advertising can be set to GAP_DISC_MODE_NON_DISCOVERABLE only (non-discoverable).
- If the discoverability mode is GAP_DISC_MODE_BROADCASTER, the advertising mode can be set to GAP_ADV_TYPE_ADV_NONCONN_IND only (non-connectable and non-scannable).
- The peer_addr parameter is used only for directed advertising or when controller privacy is enabled (the second parameter, BLE_GAP_PRIV_CFG_PRIV_EN_BIT, of ble_gap_privacy_params_set is set).
- Code path:

  SDK_Folder\projects\ble\ble_basic_example\ble_app_gap_legacy_adv\Src\user\user_app.c

2. Set advertising data (optional).

   Setting advertising data is not required only when the adv_mode is GAP_ADV_TYPE_ADV_HIGH_DIRECT_IND or GAP_ADV_TYPE_ADV_LOW_DIRECT_IND.

3. Set scan response data (optional).

   Setting scan response data is required only when the adv_mode is GAP_ADV_TYPE_ADV_IND or GAP_ADV_TYPE_ADV_SCAN_IND.

```
static const uint8_t s_adv_data_set[] =
{
    0x03,
    BLE_GAP_AD_TYPE_COMPLETE_LIST_16_BIT_UUID,
    0x01, 0x00,
};
static const uint8_t s_adv_rsp_data_set[] =
{
    0x0b,
    BLE_GAP_AD_TYPE_COMPLETE_NAME,
    'L', 'e', 'g', 'a', 'c', 'y', '_', 'A', 'D', 'V',
```

```
};
error_code = ble_gap_adv_data_set(0, BLE_GAP_ADV_DATA_TYPE_DATA,
                                     s_adv_data_set, sizeof(s_adv_data_set));
APP_ERROR_CHECK(error_code);
error_code = ble_gap_adv_data_set(0, BLE_GAP_ADV_DATA_TYPE_SCAN_RSP,
                                     s_adv_rsp_data_set,
                                     sizeof(s_adv_rsp_data_set));
APP_ERROR_CHECK(error_code);
```

Note:

Set the adv_data and adv_rsp_data (length, type, and data) in compliance with format regulations in *Bluetooth Core Spec*. The length refers to the total length of the type and data fields.

Code path:

`SDK_Folder\projects\ble\ble_basic_example\ble_app_gap_legacy_adv\Src\user\user_app.c`

4. Enable advertising.

   Users are required to configure the adv_idx parameter when using ble_gap_adv_start to enable advertising. This helps specify an advertising instance index. Using ble_gap_adv_start can establish up to five legacy advertisements concurrently, so the value of adv_idx ranges from 0, 1, 2, 3, to 4.

```
s_gap_adv_time_param.duration = 0;
s_gap_adv_time_param.max_adv_evt = 0;
error_code = ble_gap_adv_start(0, &s_gap_adv_time_param);
APP_ERROR_CHECK(error_code);
```

Note:

Code path:

`SDK_Folder\projects\ble\ble_basic_example\ble_app_gap_legacy_adv\Src\user\user_app.c`

5. Call the app_gap_adv_start_cb callback function after enabling the legacy advertising.

Note:

If users hope to update advertising parameters, call the ble_gap_adv_data_set and ble_gap_adv_param_set API functions after the advertising terminates to reconfigure advertising data and parameters. You can restart advertising by using ble_gap_adv_start.

6. If the value of scan_req_ind_en is true, the app_gap_scan_req_ind_cb callback function is called when the local device receives a scanning request from the peer device.

2.4.1.2 Enable Extended Advertising

When extended advertising on a peripheral is enabled, interactions between the Bluetooth LE applications and BLE Stack are shown in Figure 2-6.

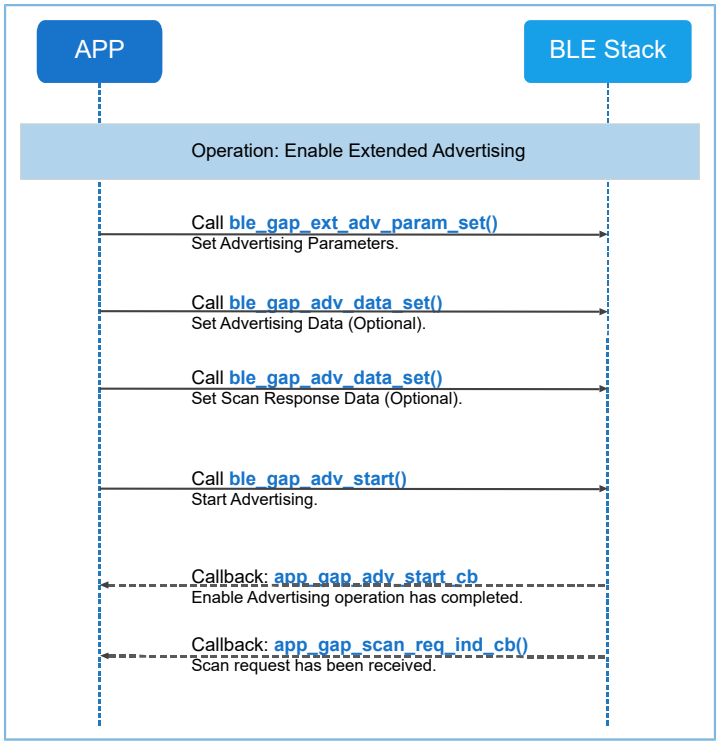

Figure 2-6 App-BLE-Stack interactions when extended advertising is enabled

Follow the steps below to enable extended advertising on a peripheral:

Note:

Code snippets in the steps below are extracted from the extended advertising example: ble_app_gap_extended_adv (`SDK_Folder\projects\ble\ble_basic_example\`).

---

1. Set extended advertising parameters.

```
s_gap_adv_param.type = GAP_ADV_TYPE_EXTENDED;
s_gap_adv_param.disc_mode = GAP_DISC_MODE_GEN_DISCOVERABLE;
/* The advertisement shall not be both connectable and scannable, and
    High duty cycle directed advertising cannot be used */
s_gap_adv_param.prop = GAP_ADV_PROP_CONNECTABLE_BIT;
s_gap_adv_param.max_tx_pwr = 0;
s_gap_adv_param.filter_pol = GAP_ADV_ALLOW_SCAN_ANY_CON_ANY;
memset(&s_gap_adv_param.peer_addr, 0, sizeof(gap_bdaddr_t));
s_gap_adv_param.prim_cfg.adv_intv_min = APP_ADV_MIN_INTERVAL;
s_gap_adv_param.prim_cfg.adv_intv_max = APP_ADV_MAX_INTERVAL;
s_gap_adv_param.prim_cfg.chnl_map = GAP_ADV_CHANNEL_37_38_39;
s_gap_adv_param.prim_cfg.phy = GAP_PHY_1MBPS_VALUE;
s_gap_adv_param.second_cfg.max_skip = 0;
s_gap_adv_param.second_cfg.phy = GAP_PHY_1MBPS_VALUE;
s_gap_adv_param.second_cfg.adv_sid = 0x00;
s_gap_adv_param.period_cfg.adv_intv_min = 0;
s_gap_adv_param.period_cfg.adv_intv_max = 0;
error_code = ble_gap_ext_adv_param_set(0, BLE_GAP_OWN_ADDR_STATIC,
                                       &s_gap_adv_param);
APP_ERROR_CHECK(error_code);
```

Note:

- GAP_ADV_PROP_DIRECTED_BIT in the prop parameter can be set in two scenarios: a. When the disc_mode parameter = GAP_DISC_MODE_NON_DISCOVERABLE; b. When the disc_mode parameter = GAP_DISC_MODE_BROADCASTER and the GAP_ADV_PROP_ANONYMOUS_BIT is set.
- High duty cycle directed advertising mode is not supported in GAP extended advertising, so GAP_ADV_PROP_HDC_BIT in the prop parameter cannot be set.
- Scannable and connectable advertising mode is not supported in GAP extended advertising, so GAP_ADV_PROP_CONNECTABLE_BIT and GAP_ADV_PROP_SCANNABLE_BIT in the prop parameter cannot be set concurrently.
- If GAP_ADV_PROP_ANONYMOUS_BIT is set in the prop parameter, the disc_mode can be set to either GAP_DISC_MODE_NON_DISCOVERABLE or GAP_DISC_MODE_BROADCASTER.
- If GAP_ADV_PROP_ANONYMOUS_BIT is set in the prop parameter, neither GAP_ADV_PROP_CONNECTABLE_BIT nor GAP_ADV_PROP_SCANNABLE_BIT can be set.
- The peer_addr parameter is used only for directed advertising or when controller privacy is enabled (the second parameter, BLE_GAP_PRIV_CFG_PRIV_EN_BIT, of ble_gap_privacy_params_set is set).

  Code path:

  `SDK_Folder\projects\ble\ble_basic_example\ble_app_gap_extended_adv\Src\user\user_app.c`

2. Set advertising data (optional). Setting extended advertising data is not required when GAP_ADV_PROP_SCANNABLE_BIT is set in the prop parameter.
3. Set scan response data (optional). Setting scan response data is required when GAP_ADV_PROP_SCANNABLE_BIT is set in the prop parameter.

```
static const uint8_t s_adv_data_set[] =
{
    0x03,
    BLE_GAP_AD_TYPE_COMPLETE_LIST_16_BIT_UUID,
    0x01, 0x00,
};

static const uint8_t s_adv_rsp_data_set[] =
{
    0x0d,
    BLE_GAP_AD_TYPE_COMPLETE_NAME,
    'E', 'x', 't', 'e', 'n', 'd', 'e', 'd', '_', 'A', 'D', 'V',
};
error_code = ble_gap_adv_data_set(0, BLE_GAP_ADV_DATA_TYPE_DATA,
                                  s_adv_data_set,sizeof(s_adv_data_set));
APP_ERROR_CHECK(error_code);
error_code = ble_gap_adv_data_set(0, BLE_GAP_ADV_DATA_TYPE_SCAN_RSP,
                                  s_adv_rsp_data_set,sizeof(s_adv_rsp_data_set));
APP_ERROR_CHECK(error_code);
```

Note:

Set the adv_data and adv_rsp_data (length, type, and data) in compliance with format regulations in *Bluetooth Core Spec*. The length refers to the total length of the type and data fields.

Code path:

`SDK_Folder\projects\ble\ble_basic_example\ble_app_gap_extended_adv\Src\user\user_app.c`

4. Enable advertising.

   Users are required to configure the adv_idx parameter when using ble_gap_adv_start to enable advertising. This helps specify an advertising instance index. Using ble_gap_adv_start can establish up to five extended advertisements concurrently, so the value of adv_idx ranges from 0, 1, 2, 3, to 4.

```
s_gap_adv_time_param.duration = 0;
s_gap_adv_time_param.max_adv_evt = 0;
error_code = ble_gap_adv_start(0, &s_gap_adv_time_param);
APP_ERROR_CHECK(error_code);
```

Note:

Code path:

`SDK_Folder\projects\ble\ble_basic_example\ble_app_gap_extended_adv\Src\user\user_app.c`

5. Call the app_gap_adv_start_cb callback function after enabling the extended advertising.
6. If GAP_ADV_PROP_SCAN_REQ_NTF_EN_BIT in the prop parameter is set, the app_gap_scan_req_ind_cb callback function is called when the local device receives a scanning request from the peer device.

2.4.1.3 Enable Periodic Advertising

When periodic advertising on a peripheral is enabled, interactions between the Bluetooth LE applications and BLE Stack are shown in Figure 2-7.

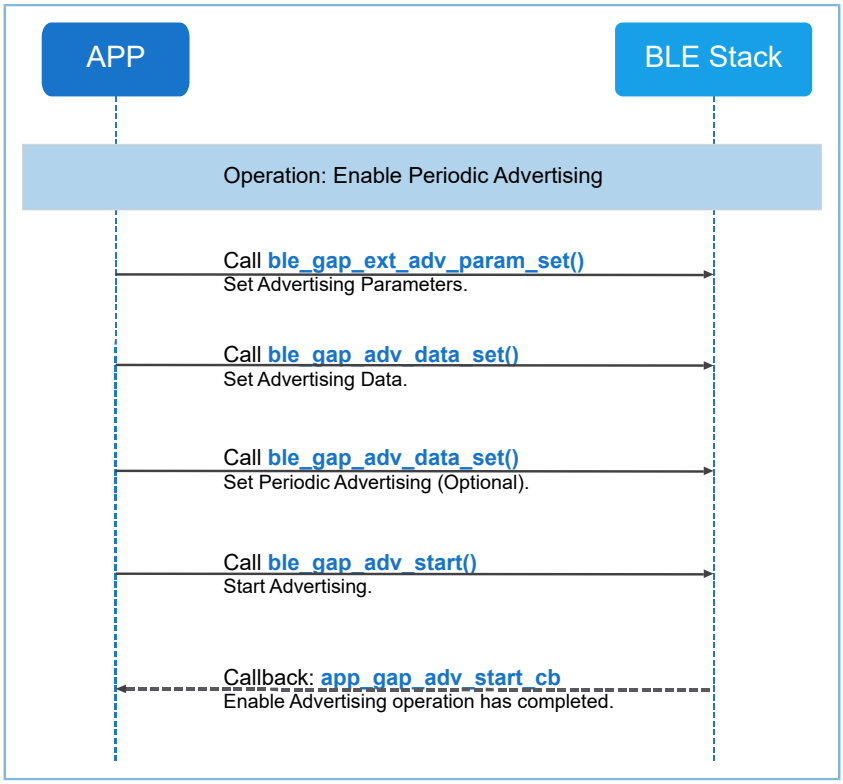

Figure 2-7 App-BLE-Stack interactions when periodic advertising is enabled

Follow the steps below to enable periodic advertising on a peripheral:

Note:

Code snippets in the steps below are extracted from the periodic advertising example: ble_app_gap_periodic_adv (`SDK_Folder\projects\ble\ble_basic_example\`).

1. Set advertising parameters.

```
s_gap_adv_param.type = GAP_ADV_TYPE_PERIODIC;
s_gap_adv_param.disc_mode = GAP_DISC_MODE_GEN_DISCOVERABLE;
/* Connectable, anonymous, scannable, high duty circle bit must be set
    to  0 */
s_gap_adv_param.prop = 0;
s_gap_adv_param.max_tx_pwr = 0;
s_gap_adv_param.filter_pol = GAP_ADV_ALLOW_SCAN_ANY_CON_ANY;
memset(&s_gap_adv_param.peer_addr, 0, sizeof(gap_bdaddr_t));
s_gap_adv_param.prim_cfg.adv_intv_min = APP_PRIMARY_ADV_MIN_INTERVAL;
s_gap_adv_param.prim_cfg.adv_intv_max = APP_PRIMARY_ADV_MAX_INTERVAL;
s_gap_adv_param.prim_cfg.chnl_map = GAP_ADV_CHANNEL_37_38_39;
s_gap_adv_param.prim_cfg.phy = GAP_PHY_1MBPS_VALUE;
s_gap_adv_param.second_cfg.max_skip = 0;
s_gap_adv_param.second_cfg.phy = GAP_PHY_1MBPS_VALUE;
s_gap_adv_param.second_cfg.adv_sid = 0x00;
s_gap_adv_param.period_cfg.adv_intv_min = APP_PERIODIC_ADV_MIN_INTERVAL;
s_gap_adv_param.period_cfg.adv_intv_max = APP_PERIODIC_ADV_MAX_INTERVAL;
error_code = ble_gap_ext_adv_param_set(0, BLE_GAP_OWN_ADDR_STATIC,
                                       &s_gap_adv_param);
APP_ERROR_CHECK(error_code);
```

Note:

- When you set the periodic advertising parameters, the following four macro definitions in the prop parameter cannot be set: GAP_ADV_PROP_CONNECTABLE_BIT, GAP_ADV_PROP_SCANNABLE_BIT, GAP_ADV_PROP_ANONYMOUS_BIT, and GAP_ADV_PROP_HDC_BIT.
- The peer_addr parameter is used only for directed advertising or when controller privacy is enabled (the second parameter, BLE_GAP_PRIV_CFG_PRIV_EN_BIT, of ble_gap_privacy_params_set is set).

  Code path:

  `SDK_Folder\projects\ble\ble_basic_example\ble_app_gap_periodic_adv\Src\user\user_app.c`

2. Set advertising data.
3. Set periodic advertising data (optional).

```
static const uint8_t s_adv_data_set[] =
{
    0x03,
    BLE_GAP_AD_TYPE_COMPLETE_LIST_16_BIT_UUID,
    0x01, 0x00,
    0x0d,
    BLE_GAP_AD_TYPE_COMPLETE_NAME,
    'P', 'e', 'r', 'i', 'o', 'd', 'i', 'c', '_', 'A', 'D', 'V',
};
static const uint8_t s_periodic_adv_data[] =
{
    0x01, 0x02, 0x03, 0x04, 0x05, 0x06
};
error_code = ble_gap_adv_data_set(0, BLE_GAP_ADV_DATA_TYPE_DATA,
                                        s_adv_data_set, sizeof(s_adv_data_set));
APP_ERROR_CHECK(error_code);
error_code = ble_gap_adv_data_set(0, BLE_GAP_ADV_DATA_TYPE_PER_DATA,
                                  s_periodic_adv_data,sizeof(s_periodic_adv_data));
APP_ERROR_CHECK(error_code);
```

Note:

Set the adv_data and periodic_adv_data (length, type, and data) in compliance with format regulations in *Bluetooth Core Spec*. The length refers to the total length of the type and data fields.

Code path:

`SDK_Folder\projects\ble\ble_basic_example\ble_app_gap_periodic_adv\Src\user\user_app.c`

4. Enable advertising. Users are required to configure the adv_idx parameter when using ble_gap_adv_start to enable advertising. This helps specify an advertising instance index. Using ble_gap_adv_start can establish up to five periodic advertisements concurrently, so the value of adv_idx ranges from 0, 1, 2, 3, to 4.

```
s_gap_adv_time_param.duration = 0;
s_gap_adv_time_param.max_adv_evt = 0;
error_code = ble_gap_adv_start(0, &s_gap_adv_time_param);
```

```
APP_ERROR_CHECK(error_code);
```

Note:

Code path:

`SDK_Folder\projects\ble\ble_basic_example\ble_app_gap_periodic_adv\Src\user\user_app.c`

5. Call the app_gap_adv_start_cb callback function after enabling the periodic advertising.

It should be noted that a Bluetooth LE device supports up to five advertisements concurrently, including legacy, extended, and periodic advertising at the GAP layer.

2.4.2 Central

The central devices support legacy scanning, extended scanning, initiating a legacy or an extended Bluetooth connection, and establishing periodic advertising synchronization.

2.4.2.1 Enable Legacy Scanning

When legacy scanning on a central is enabled, interactions between the Bluetooth LE applications and BLE Stack are shown in Figure 2-8.

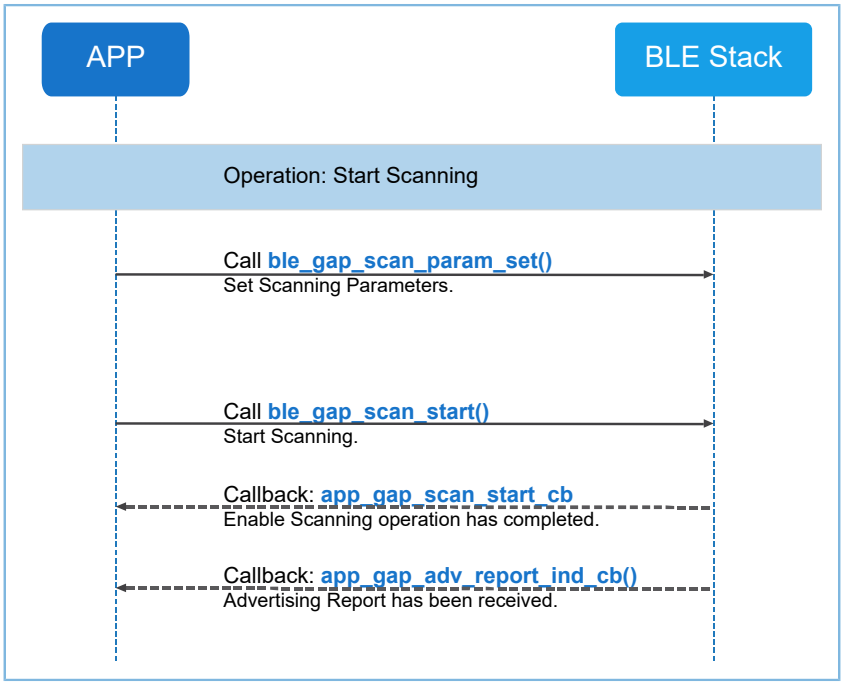

Figure 2-8 App-BLE-Stack interactions when legacy scanning is enabled

Follow the steps below to enable legacy scanning on a central:

Note:

Code snippets in the steps below are extracted from the legacy scanning example: ble_app_gap_legacy_scan (SDK_Folder\projects\ble\ble_basic_example\).

1. Set scanning parameters.

```
s_scan_param.scan_type = GAP_SCAN_ACTIVE;
s_scan_param.scan_mode = GAP_SCAN_OBSERVER_MODE;
s_scan_param.scan_dup_filt = GAP_SCAN_FILT_DUPLIC_DIS;
s_scan_param.use_whitelist = 0;
s_scan_param.interval= APP_SCAN_INTERVAL;
s_scan_param.window= APP_SCAN_WINDOW;
s_scan_param.timeout = 0;
error_code = ble_gap_scan_param_set(BLE_GAP_OWN_ADDR_STATIC,
                                    &s_scan_param);
APP_ERROR_CHECK(error_code);
```

Note:

The value of s_scan_param.interval should be equal to or greater than that of s_scan_param.window.

If scan_mode = GAP_SCAN_LIM_DISC_MODE or GAP_SCAN_GEN_DISC_MODE, the default scanning timeout is 10.24 seconds.

Code path:

SDK_Folder\projects\ble\ble_basic_example\ble_app_gap_legacy_scan\Src\user\user_app.c

2. Start scanning.

```
error_code = ble_gap_scan_start();
APP_ERROR_CHECK(error_code);
```

3. Call the app_gap_scan_start_cb callback function after enabling the legacy scanning.
4. The app_gap_adv_report_ind_cb callback function is called when the local device receives advertising data or a scanning request from the peer device.

2.4.2.2 Enable Extended Scanning

When extended scanning on a central is enabled, interactions between the Bluetooth LE applications and BLE Stack are shown in Figure 2-9.

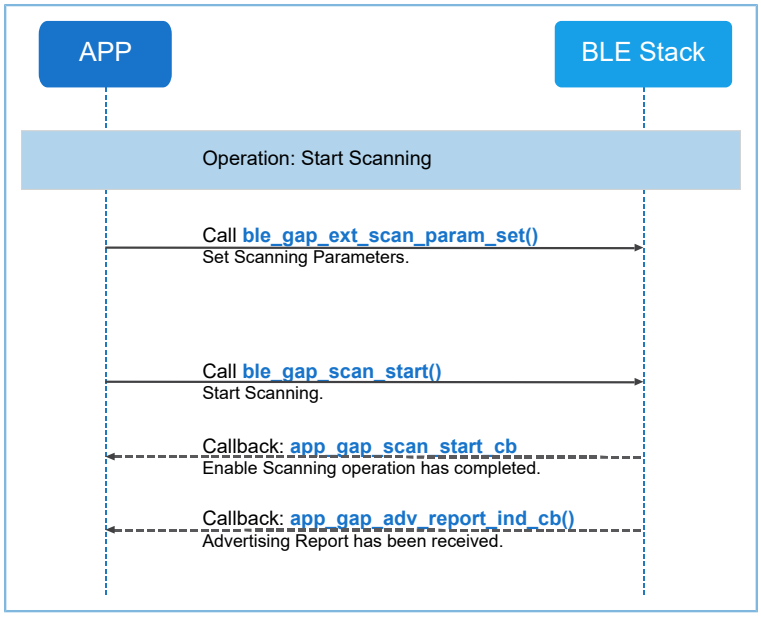

Figure 2-9 App-BLE-Stack interactions when extended scanning is enabled

Follow the steps below to enable extended scanning on a central:

Note:

Code snippets in the steps below are extracted from the extended scanning example: ble_app_gap_extended_scan (`SDK_Folder\projects\ble\ble_basic_example\`).

1. Set extended scanning parameters.

```
s_scan_param.type = GAP_EXT_SCAN_TYPE_OBSERVER;
s_scan_param.prop = GAP_SCAN_PROP_PHY_1M_BIT |
                    GAP_SCAN_PROP_FILT_TRUNC_BIT;
s_scan_param.dup_filt_pol = GAP_EXT_DUP_FILT_DIS;
s_scan_param.scan_param_1m.scan_intv = APP_SCAN_INTERVAL;
s_scan_param.scan_param_1m.scan_wd= APP_SCAN_WINDOW;
s_scan_param.duration= 0;
s_scan_param.period= 0;
error_code = ble_gap_ext_scan_param_set(BLE_GAP_OWN_ADDR_STATIC,
                                        &s_scan_param);
APP_ERROR_CHECK(error_code);
```

Note:

- The value of scan_param.scan_param_1m.scan_intv should be equal to or greater than that of scan_param.scan_param_1m.scan_wd.

  If scan_param.type = GAP_EXT_SCAN_TYPE_LIM_DISC or GAP_EXT_SCAN_TYPE_LIM_DISC, the default scanning timeout is 10.24 seconds.

- Code path:

  `SDK_Folder\projects\ble\ble_basic_example\ble_app_gap_extended_scan\Src\user\user_app.c`

2. Start scanning.

```
error_code = ble_gap_scan_start();
APP_ERROR_CHECK(error_code);
```

3. Call the app_gap_scan_start_cb callback function after enabling the extended scanning.
4. The app_gap_adv_report_ind_cb callback function is called when the local device receives advertising data or a scanning request from the peer device.

2.4.2.3 Initiate a Legacy Connection

When a central initiates a legacy connection, the interaction process between the Bluetooth LE applications and BLE Stack is shown in Figure 2-10.

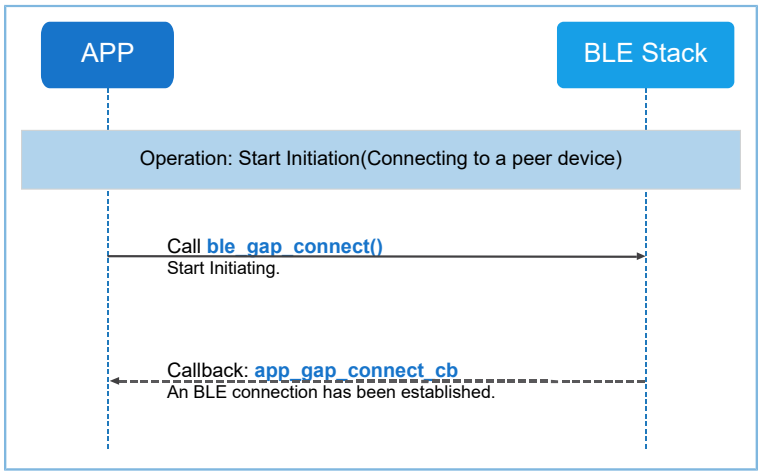

Figure 2-10 App-BLE-Stack interactions when a legacy connection is initiated

Follow the steps below to initiate a legacy connection on a central:

Note:

Code snippets in the steps below are extracted from the legacy connection example: ble_app_gap_legacy_connect (S`DK_Folder\projects\ble\ble_basic_example\`).

1. Set parameters for initiating a legacy connection.

```
//peer device address
uint8_t peer_dev_addr[] = {0x01, 0x23, 0x45, 0x67, 0x89, 0xA0};
memcpy(s_conn_param.peer_addr.gap_addr.addr, peer_dev_addr, 6);
s_conn_param.type = GAP_INIT_TYPE_DIRECT_CONN_EST;
//address type is public
s_conn_param.peer_addr.addr_type = 0;
s_conn_param.interval_min = APP_CONNECTION_MIN_INTERVAL;
s_conn_param.interval_max = APP_CONNECTION_MAX_INTERVAL;
s_conn_param.slave_latency = APP_CONNECTION_SLAVE_LATENCY;
s_conn_param.sup_timeout = APP_CONNECTION_MAX_TIMEOUT;
```

Note:

During testing, users can set m_conn_param.peer_addr to the actual address to which the central connects on demand. If s_conn_param.type = GAP_INIT_TYPE_NAME_DISC, the app_gap_connect_cb callback function does not report an event to the application layer after the connection is established. Instead, the central automatically accesses the device name of the peer device, after which the app_gap_peer_name_ind_cb callback function returns a value to the application layer. Afterwards, the central will terminate the connection, and the app_gap_disconnect_cb callback function does not report a value to the application layer.

Code path:

`SDK_Folder\projects\ble\ble_basic_example\ble_app_gap_legacy_connect\Src\user\user_app.c`

2. Initiate a connection.

```
error_code = ble_gap_connect(BLE_GAP_OWN_ADDR_STATIC,&s_conn_param);
APP_ERROR_CHECK(error_code);
```

3. The app_gap_connect_cb callback function is called regardless of whether the connection is established successfully.

2.4.2.4 Initiate an Extended Connection

When a central initiates an extended connection, the interaction process between the Bluetooth LE applications and BLE Stack is shown in Figure 2-11.

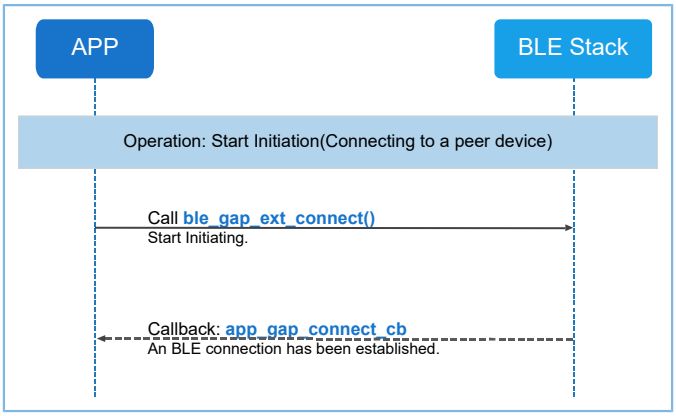

Figure 2-11 App-BLE-Stack interactions when an extended connection is initiated

Follow the steps below to initiate an extended connection on a central:

Note:

Code snippets in the steps below are extracted from the extended connection example: ble_app_gap_extended_connect (SDK_Folder\projects\ble\ble_basic_example\).

1. Set parameters for initiating an extended connection.

```
uint8_t test_addr[] = {0x01, 0x23, 0x45, 0x67, 0x89, 0xA0};
gap_ext_init_param_t ext_conn_param;
memset(&ext_conn_param, 0 , sizeof(ext_conn_param));
ext_conn_param.type = GAP_INIT_TYPE_DIRECT_CONN_EST;
ext_conn_param.prop = GAP_INIT_PROP_1M_BIT;
ext_conn_param.conn_to = 0;
ext_conn_param.scan_param_1m.scan_intv = 15;
ext_conn_param.scan_param_1m.scan_wd = 15;
ext_conn_param.conn_param_1m.conn_intv_min = 6;
ext_conn_param.conn_param_1m.conn_intv_max = 10;
ext_conn_param.conn_param_1m.conn_latency = 1;
ext_conn_param.conn_param_1m.supervision_to = 100;
ext_conn_param.conn_param_1m.ce_len = 0;
ext_conn_param.peer_addr.addr_type = 0;
memcpy(ext_conn_param.peer_addr.gap_addr.addr, test_addr, 6);
```

Note:

During testing, users can set ext_conn_param.peer_addr to the actual address to which the central connects on demand. If ext_conn_param.type = GAP_INIT_TYPE_NAME_DISC, the app_gap_connect_cb callback function does not report an event to the application layer after the connection is established. Instead, the central automatically accesses the device name of the peer device, after which the app_gap_peer_name_ind_cb callback function returns a value to the application layer. Afterwards, the central will terminate the connection, and the app_gap_disconnect_cb callback function does not report a value to the application layer.

Code path:

SDK_Folder\projects\ble\ble_basic_example\ble_app_gap_extended_connect\Src\user\user_app.c

2. Initiate a connection.

```
error_code = ble_gap_ext_connect(BLE_GAP_OWN_ADDR_STATIC,&ext_conn_param);
APP_ERROR_CHECK(error_code);
```

3. The app_gap_connect_cb callback function is called regardless of whether the connection is established successfully.

2.4.2.5 Establish Periodic Advertising Synchronization

When a central establishes periodic advertising synchronization, the interaction process between the Bluetooth LE applications and BLE Stack is shown in Figure 2-12.

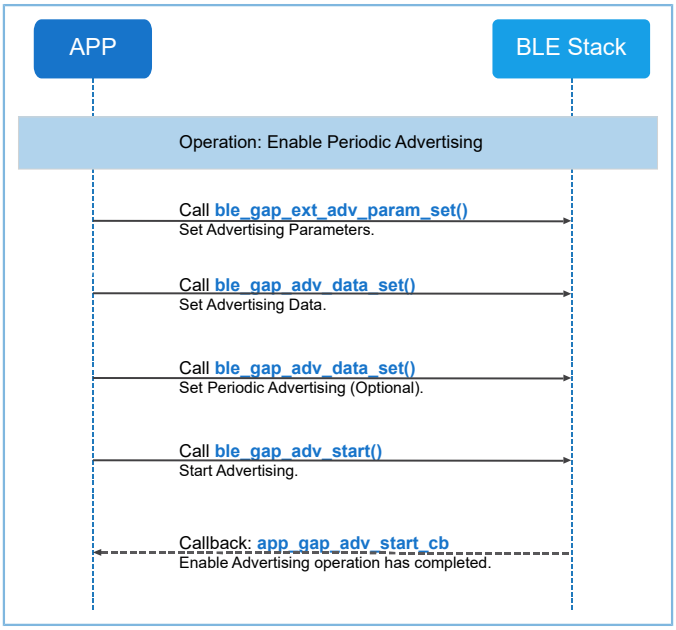

Figure 2-12 App-BLE-Stack interactions when a periodic advertising synchronization is established

Follow the steps below to establish periodic advertising synchronization on a central:

Note:

Code snippets in the steps below are extracted from the periodic advertising synchronization example: ble_app_gap_periodic _sync (SDK_Folder\projects\ble\ble_basic_example\).

---

1. Extended scanning is required before establishing periodic advertising synchronization. For details about enabling extended scanning, see "Section 2.4.2.2 Enable Extended Scanning".

2. Set extended scanning parameters.

```
s_scan_param.type = GAP_EXT_SCAN_TYPE_OBSERVER;
s_scan_param.prop = GAP_SCAN_PROP_PHY_1M_BIT |
                    GAP_SCAN_PROP_FILT_TRUNC_BIT;
s_scan_param.dup_filt_pol = GAP_EXT_DUP_FILT_DIS;
s_scan_param.scan_param_1m.scan_intv = APP_SCAN_INTERVAL;
s_scan_param.scan_param_1m.scan_wd= APP_SCAN_WINDOW;
s_scan_param.duration= 0;
```

```
s_scan_param.period= 0;
error_code = ble_gap_ext_scan_param_set(BLE_GAP_OWN_ADDR_STATIC,
                                        &s_scan_param);
APP_ERROR_CHECK(error_code);
```

Note:

Code path:

`SDK_Folder\projects\ble\ble_basic_example\ble_app_gap_periodic_sync\Src\user\user_app.c`

3. Enable extended scanning.

```
error_code = ble_gap_scan_start();
APP_ERROR_CHECK(error_code);
```

4. Set periodic advertising synchronization data.

   In the static void app_gap_adv_report_ind_cb(uint8_t conidx, const gap_ext_adv_report_ind_t *param) callback function, set the periodic advertising synchronization parameter, and enable periodic advertising synchronization.

```
s_per_sync_param.skip = 0;
s_per_sync_param.sync_to = APP_SYNC_TIMEOUT;
s_per_sync_param.type = GAP_PER_SYNC_TYPE_GENERAL;
s_per_sync_param.adv_addr.adv_sid = p_adv_report->adv_sid;
memcpy(s_per_sync_param.adv_addr.bd_addr.gap_addr.addr, s_peer_dev_addr,
        6);
s_per_sync_param.adv_addr.bd_addr.addr_type = 0;
ble_gap_per_sync_param_set(0, &s_per_sync_param);
```

Note:

During testing, set peer_dev_addr to the address of the device on which periodic advertising is enabled.

Code path:

`SDK_Folder\projects\ble\ble_basic_example\ble_app_gap_periodic_sync\Src\user_callback\user_gap_callback.c`

5. Start periodic advertising synchronization. Users are required to configure the per_sync_idx parameter when using ble_gap_per_sync_start to establish periodic advertising synchronization. The value of per_sync_idx ranges from 0, 1, 2, 3, to 4.

```
ble_gap_per_sync_start(0);
```

6. The app_gap_sync_establish_cb callback function is called regardless of whether the periodic advertising synchronization is established successfully.
7. After periodic advertising synchronization is established, the periodic advertising data is reported to users by using the app_gap_adv_report_ind_cb callback function.

3 Generic Attribute Profile (GATT)

Generic Attribute Profile (GATT) is used by the application layer for data communications between two connected devices. At the GATT layer of BLE Stack, data is passed and stored in the form of characteristics.

3.1 GATT Roles

At the GATT layer, when two devices are connected, they are each in one of the following two roles:

- GATT Client: a device that initiates commands and requests, and receives responses, notifications, and indications from the GATT Server
- GATT Server: a device that receives commands and requests from the GATT Client, and sends responses, notifications, and indications to the GATT Client

Figure 3-1 shows the connection relationship between two Bluetooth LE devices.

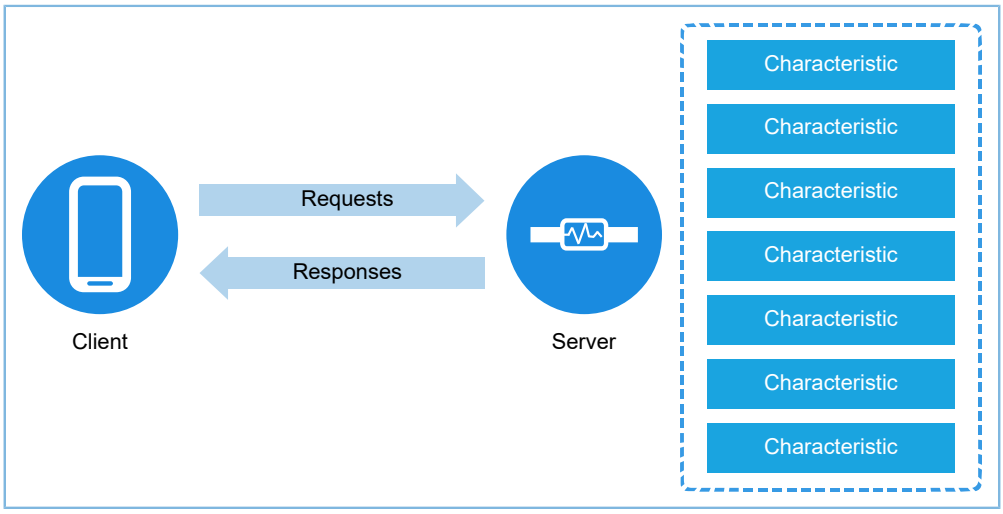

Figure 3-1 GATT Server-Client connection diagram

In the figure above, the peripheral (a Bluetooth wristband) serves as the GATT Server, and the central (a mobile phone with Bluetooth enabled) serves as the GATT Client. GATT roles (GATT Client and GATT Server) are independent of GAP roles (the peripheral and the central). Both a peripheral and a central can not only serve as a GATT Client, but also a GATT Server.

3.2 GATT Profile Hierarchy

GATT defines the hierarchy to exchange profile data, as shown in Figure 3-2:

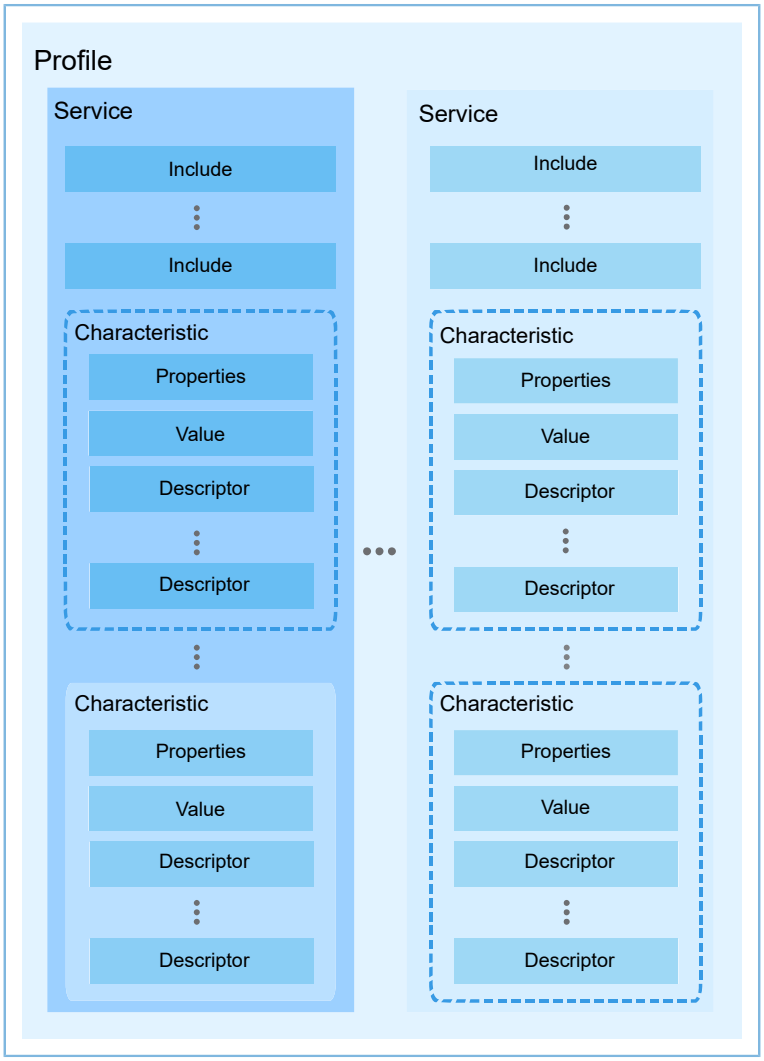

Figure 3-2 GATT profile hierarchy

One profile includes one or more services; one service includes one or more characteristics; one characteristic includes at least two attributes, including characteristic declaration and characteristic value.

Attributes, characteristics, and services are described in the following sections.

3.2.1 Attribute

In Bluetooth LE connections, characteristics are considered as groups of attribute information, including characteristic declarations, characteristic values, and characteristic descriptors. At the GATT layer, data is exchanged in the form of attributes.

Attributes include the following parts:

- Handle: It is the index of an attribute in a GATT attribute table. Each attribute is assigned with a unique handle.
- Type: It indicates what the data represents. It is usually represented by a Universally Unique Identifier (UUID), which is specified by Bluetooth SIG or customized by users.
- Value: It is the data value of an attribute.

- Permission: It specifies how the GATT Client can interact with a specific attribute value, including whether and how the GATT Client can access the attribute value.

3.2.2 Characteristic

A typical characteristic is composed of the following attributes:

- Characteristic declaration: It describes properties, the storage location (handle), and the type of a characteristic.
- Characteristic value: It contains the data value of a characteristic.
- Characteristic descriptor: It describes additional information or configuration of a characteristic.

3.2.3 Service

A GATT service is a collection of characteristics. For example, the Heart Rate Service includes a Heart Rate Measurement characteristic and a body sensor location characteristic. A profile includes one or more services.

Common GATT services are described below:

- Generic Access Service: This service includes information about device and access, such as device name, vendor ID, and product ID. This service defines the following characteristics: Device Name, Appearance, and Peripheral Preferred Connection Parameters.
- Generic Attribute Service: This service is used by the GATT Server to notify the connected peer device (GATT Client) that the service provided by the server has been changed. This service includes the Service Changed characteristic.

Note:

By default, two services mentioned above are added to the server database after initialization of BLE Stack.

3.2.3.1 Generic Access Service

The Generic Access Service is mandatory for a Bluetooth LE device which serves as a central or a peripheral. The Generic Access Service is mainly used for device discovery and connection establishment.

Table 3-1 shows the attribute table of the Generic Access Service in the GATT Server.

Table 3-1 Attribute table of Generic Access Service

Handle	UUID	UUID Description	Value	Property
0x0001	0x2800	Primary Service Declaration	00:18 (Generic access)	
0x0002	0x2803	Characteristic Declaration	02:03:00:00:2A	
0x0003	0x2A00	Device Name		0x02
0x0004	0x2803	Characteristic Declaration	02:05:00:01:2A	
0x0005	0x2A01	Appearance		0x02
0x0006	0x2803	Characteristic Declaration	02:07:00:C9:2A	
0x0007	0x2AC9	Resolvable Private Address		0x02

Note:

Table 3-1 shows default characteristics, not all, of the Generic Access Service.

(The API functions used to set and obtain the attributes (partial) of this service are in SDK_Folder\components\sdk\ble_sdk_api\ble_gapm.h)

The API functions used to set characteristic values:

```
uint16_t ble_gap_device_name_set(gap_dev_name_write_perm_t write_perm,
                                          unit8_t const *p_dev_name, uint16_t length)
void ble_gap_appearance_set(uint16_t appearance)
```

These two API functions set values of Device Name and Appearance characteristics of Generic Access Service.

Optional characteristics of Generic Access Service, such as Peripheral Preferred Connection Parameters, can be added and set by using the following three API functions:

```
void ble_gap_ppcp_present_set(bool present_flag)
uint16_t ble_gap_ppcp_set(gap_conn_param_t const *p_conn_params)
uint16_t ble_gap_privacy_params_set(uint16_t renew_dur, bool enable_flag)
```

3.2.3.2 Generic Attribute Service

The Generic Attribute Service includes the Service Changed characteristic. This characteristic is used by the GATT Server to notify the bonded device (GATT Client) that the service provided by the server has been changed. After the GATT Server is disconnected from the GATT Client, if the service is changed and the indication is enabled for the Client Characteristic Configuration Descriptor (CCCD) on the GATT Client, the GATT Server sends a Service Changed indication to the GATT Client when reconnection is established between the server and the client. However, the GATT Client cannot read or write the Service Changed characteristic value from/to the GATT Server.

3.3 Attribute Table

The GATT Server organizes data to be sent based on an attribute table.

The table below demonstrates the format of attributes included in the GATT Server by taking the Heart Rate Service as an example.

Table 3-2 Attribute table of Heart Rate Service

Handle	UUID	UUID Description	Value	Property
0x002B	0x2800	Primary Service Declaration	0D:18 (Heart Rate)	
0x002C	0x2803	Characteristic Declaration	10:2D:00:37:2A	
0x002D	0x2A37	Heart Rate Measurement		0x10
0x002E	0x2902	Client Characteristic Configuration		
0x002F	0x2803	Characteristic Declaration	02:30:00:38:2A	
0x0030	0x2A38	Body Sensor Location		0x02
0x0031	0x2803	Characteristic Declaration	08:32:00:39:2A	

Handle	UUID	UUID Description	Value	Property
0x0032	0x2A39	Heart Rate Control Point		0x08

The attribute table of the Heart Rate Service includes the following characteristics:

- Heart Rate Measurement: The GATT Client receives notifications of this characteristic value from the GATT Server.
- Body Sensor Location: The GATT Client reads this characteristic.
- Heart Rate Control Point: The GATT Client writes values to this characteristic.

A handle-by-handle description of the attribute table is as follows:

- 0x002B: Represent the declaration for the Heart Rate Service. The UUID of the declaration is 0x2800, and the attribute value is the UUID of the heart rate service.
- 0x002C: Represent the declaration for the Heart Rate Measurement characteristic. This declaration can be regarded as a pointer to the value of the Heart Rate Measurement characteristic. The UUID of this declaration is 0x2803. The declaration value has 5 bytes, and each byte from the most significant bit (MSB) to the least significant bit (LSB) is defined below:
  - Byte 0 defines characteristic properties:
    - 0x01: Permit broadcasting the characteristic value.
    - 0x02: Permit reading the characteristic value.
    - 0x04: Permit writes to the characteristic value (without a response).
    - 0x08: Permit writes to the characteristic value (with a response).
    - 0x10: Permit notifications of the characteristic value to the GATT Client (without acknowledgment).
    - 0x20: Permit indications of the characteristic value to the GATT Client (with acknowledgment).
    - 0x40: Permit signed writes to the characteristic value.
    - 0x80: There is an extended properties bit which is defined in the characteristic extended properties descriptor.
  - Bytes 1–2: the handle of the characteristic value
  - Bytes 3–4: the type (UUID) of the characteristic value
- 0x002D: Represent the value of the Heart Rate Measurement characteristic. The UUID of this value is 0x2A37. The GATT Client receives notifications of the characteristic value from the GATT Server.
- 0x002E: Represent the CCCD of the Heart Rate Measurement characteristic. The UUID of this descriptor is 0x2902. The GATT Server enables or disables notification of the characteristic value to the GATT Client according to the written attribute value.
- Descriptions of handles 0x002F to 0x0032 are similar to those of the above attributes.

3.3.1 Definition

Each service must define an attribute table which is set to BLE Stack by the initialization function of the profile. The attribute table arrays defined in SDK_Folder\components\profiles\hrs\hrs.c are as follows:

- static const attm_desc_t hrs_attr_tab[HRS_IDX_NB];

  A UUID can be 16 bits or 128 bits; a 128-bit UUID is in the attm_desc_128_t type. Developers can use the UUID (defined in *ble_att.h*) specified by Bluetooth SIG or a custom UUID defined in the profile.

  The data structure of a 16-bit UUID is provided below:

```
/**
 * @brief Service(16bits UUID) description.
 */
typedef struct
{
    uint16_t uuid;        /**< 16 bits UUID LSB First. */
    uint16_t perm;        /**< Attribute permissions, see @ref BLE_GATTS_ATTR_PERM. */
    uint16_t ext_perm;    /**<Attribute extended permissions, see @ref
 BLE_GATTS_ATTR_EXT_PERM. */
    uint16_t max_size;    /**< Attribute max size. */
} attm_desc_t;
```

  The data structure of a 128-bit UUID is provided below:

```
/**
 * @brief Service(128bits UUID) description.
 */
typedef struct
{
    uint8_t uuid[16];     /**< 128 bits UUID LSB First. */
    uint16_t perm;        /**< Attribute permissions, see @ref BLE_GATTS_ATTR_PERM. */
    uint16_t ext_perm;    /**<Attribute extended permissions, see @ref
 BLE_GATTS_ATTR_EXT_PERM. */
    uint16_t max_size;    /**< Attribute max size. */
} attm_desc_128_t;
```

- perm (attribute permissions)

  Attribute permissions define whether the peer device (GATT Client) can access and how to access attribute values stored in the GATT Server. Permissions are defined as below:

```
/**< Default Read permission. No encrypt*/
#define READ_PERM_UNSEC      (READ << 8)
/**< Read permission set with authenticate level *
#define READ_PERM(sec_level)  (READ << 8 | ((sec_level &  SEC_LEVEL_MASK) << READ_POS))

/**< Default Write Permission. No encrypt */
#define WRITE_REQ_PERM_UNSEC  (WRITE_REQ << 8)

/**< Write permission set with authenticate level */
#define WRITE_REQ_PERM(sec_level) (WRITE_REQ << 8 | ((sec_level &  SEC_LEVEL_MASK) <<
 WRITE_POS))

/**< Default Write without Response Permission. No encrypt */
#define WRITE_CMD_PERM_UNSEC  (WRITE_CMD << 8)

/**<  Write without Response permission set with authenticate level */
```

```
#define WRITE_CMD_PERM(sec_level)   (WRITE_CMD << 8 | ((sec_level & SEC_LEVEL_MASK) <<
 WRITE_POS))

/**< Default Authenticated Signed Write Permission. No encrypt */
#define WRITE_SIGNED_PERM_UNSEC    (WRITE_SIGNED << 8)

/**< Authenticated Signed Write permission set with authenticate level */
#define WRITE_SIGNED_PERM(sec_level)   (WRITE_SIGNED << 8 |   ((sec_level & SEC_LEVEL_MASK) <<
 WRITE_POS))

/**< Default Indicate Permission. No encrypt */
#define INDICATE_PERM_UNSEC            (INDICATE << 8)

/**< Indicate permission set with authenticate level */
#define INDICATE_PERM(sec_level)    (INDICATE << 8 | ((sec_level &  SEC_LEVEL_MASK) <<
 INDICATE_POS))

/**< Default Notify Permission. No encrypt */
#define NOTIFY_PERM_UNSEC                (NOTIFY << 8))

/**< Notify permission set with authenticate level */
#define NOTIFY_PERM(sec_level)        (NOTIFY << 8 | ((sec_level &  SEC_LEVEL_MASK) <<
 NOTIFY_POS))

/**< Broadcast enable. */
#define BROADCAST_ENABLE                (BROADCAST << 8)

/**< Extended Properties enable. */
#define EXT_PROP_ENABLE                   (EXT_PROP << 8)
```

- ext_perm (attribute extended permissions)

  Attribute extended permissions define the encryption key size of an encrypted link, the UUID length of an attribute, and the location of an attribute value. Code is provided below:

```
/**< 16 bytes encryption key size . */
#define ATT_ENC_KEY_SIZE_16     (0x1000)
/**< Attribute UUID length set. See @ref BLE_GATTS_UUID_TYPE */
#define ATT_UUID_TYPE_SET (uuid_len)    (uuid_len << 13)
/**< Value location, means value saved in user space, the profile's read/write callback will
 be called.. */
#define ATT_VAL_LOC_USER    (0x8000)
```

- max_size (maximum characteristic size)

  If the attribute serves as a characteristic value, the max_size defines the maximum size of the characteristic value.

  The following part describes different types of attribute definitions of attm_desc_t hrs_attr_tab by taking the Heart Rate Profile (SDK_`Folder\components\profiles\hrs`) as an example.

- Service declaration

  It represents the service declaration for the Heart Rate Profile service:

```
{
    BLE_ATT_DECL_PRIMARY_SERVICE,           //16bits UUID
    READ_PERM_UNSEC,                        //permission
    0,                                      //ext_permission
```

```
    0                                       //max size
}
```

The UUID is set to a primary service UUID (0x2800) defined by Bluetooth SIG. The permission of the attribute is set to READ_PERM_UNSEC, so that the GATT Client can read the attribute.

- Characteristic declaration

  It represents the characteristic declaration for the Heart Rate Profile Measurement:

```
{
    BLE_ATT_DECL_CHARACTERISTIC,            //16bits UUID
    READ_PERM_UNSEC,                        //permission
    0,                                      //ext_permission
    0                                       //max size
}
```

The UUID is set to a characteristic declaration UUID (0x2803) defined by Bluetooth SIG. The permission of the attribute is set to READ_PERM_UNSEC, so that the GATT Client can read the attribute.

- Characteristic value

  It contains the characteristic value of the Heart Rate Profile Measurement:

```
{
    BLE_ATT_CHAR_HEART_RATE_MEAS,           //16bits uuid
    READ_PERM_UNSEC|NOTIFY_PERM_UNSEC,      //permission
    ATT_VAL_LOC_USER,                       //ext_permission
    HRS_MEAS_MAX_LEN                        //max size
}
```

The UUID is set to a user-defined UUID. If the MSB of the ext_perm is set (indicating the characteristic value is stored in user space), the GATT Client reads or writes the characteristic value from/to the GATT Server, and BLE Stack calls the read or write callback function (if any) of the heart rate profile.

- Client characteristic configuration

  It represents the characteristic configuration value of the Heart Rate Profile Measurement:

```
{
    BLE_ATT_DESC_CLIENT_CHAR_CFG,               //UUID
    READ_PERM_UNSEC| WRITE_REQ_PERM_UNSEC,      //permission
    0,                                          //ext_permission
    0                                           //max size
}
```

The UUID is set to a client characteristic configuration UUID (0x2902) defined by Bluetooth SIG. The GATT Client shall be granted with read and write permissions, so that it can read and write the attribute. Due to the existing multiple connections, multiple CCCD values exist. The CCCD values must be stored in user space only, and the MSB of the ext_perm of the attribute is set mandatorily.

3.3.2 Create an Attribute Table

When a device is powered on or resets, applications need to create a service list during initialization. Each service includes some attributes and necessary callback functions based on demands. The attribute table and callback functions are set to the ble_server_prf_add function and stored in BLE Stack.

The attribute table must be initialized in the initialization function of applications. For example, functions A_service_init and B_service_init are called successively in the initialization function of applications. The implementation procedures are shown in Figure 3-3.

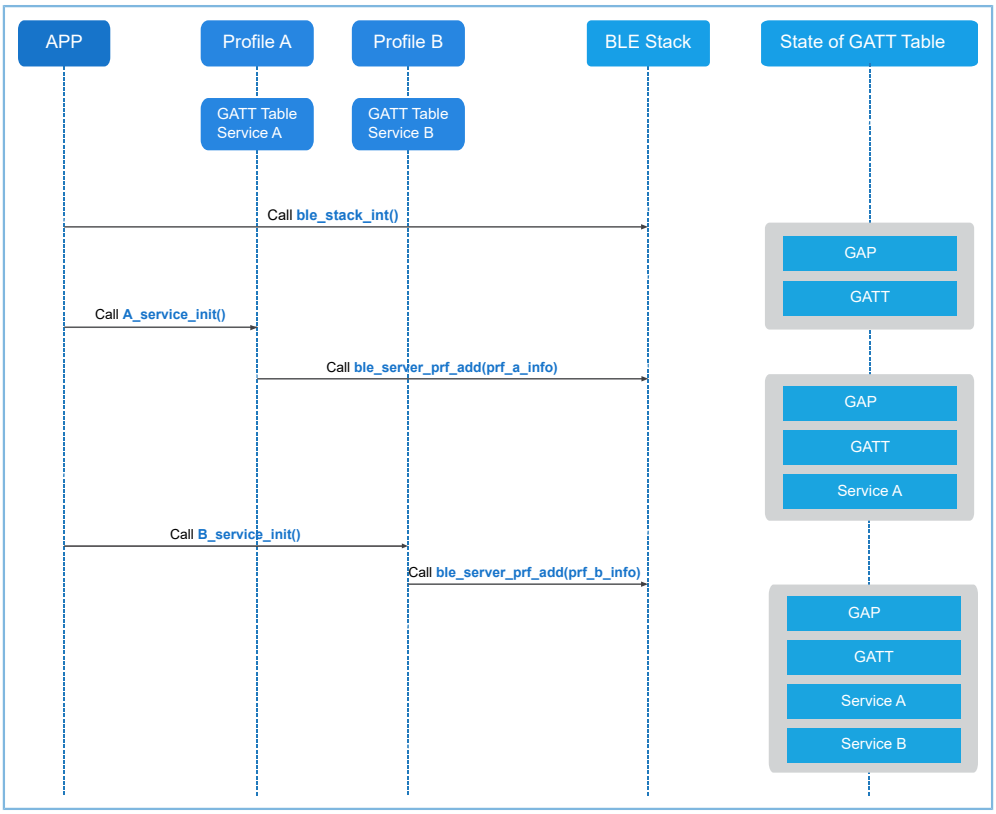

Figure 3-3 Procedures to create an attribute table

3.4 Fundamental GATT Procedures

GATT defines sub-procedures for communications between the GATT Server and the GATT Client. Procedures for the GATT Client to initiate communications and for the GATT Server to process requests are described below.

3.4.1 GATT Client

The GATT Client sends commands and requests to the GATT Server, and receives responses, indications, and notifications from the GATT Server.

The GATT Client has no attribute table or profiles. It obtains attribute information, instead of providing attribute information and services. Most of the interactions with the GATT layer are initiated directly by the application layer. In this case, GATT API functions can be used directly. These API functions are mainly used by applications of the GATT Client. For most of these API functions, when they are called, results are returned from callback functions registered by users, including attribute values which have been read, write state, and indications. For more information about API functions, see *GR551x API Reference*.

3.4.1.1 Client-initiated Communications

This section describes procedures for applications at the application layer to serve as the GATT Client to initiate communications. Related function declarations of the GATT Client are in SDK_Folder\components\sdk\ble_sdk_api\ble_gattc.h.

When a device serves as the GATT Client to initiate communications, interactions between the applications and BLE Stack are shown in Figure 3-4.

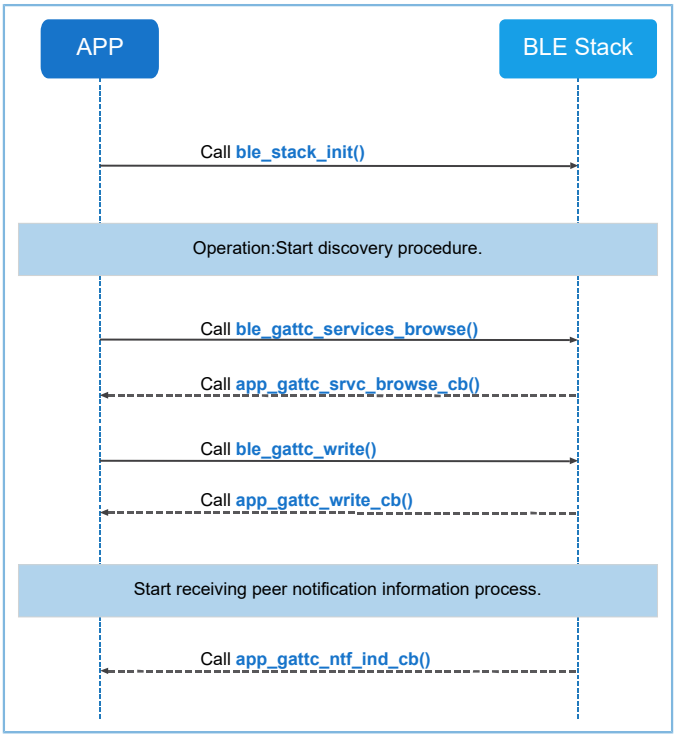

Figure 3-4 App-BLE-Stack interactions of the GATT Client (sample)

Follow the steps below to initiate communications from the GATT Client:

Note:

Code snippets in the steps below are extracted from the GATT Client procedure example: ble_app_gatt_client (SDK_Folder\projects\ble\ble_basic_example\).

---

1. Implement callback function sets of the GATT Client.

   Users need to implement callback function sets of the GATT Client, including app_gattc_callback.

   The pointer members contained in the app_gattc_callback callback function set of the GATT Client include:

```
/**@brief GATTC Event callback Structures. */
typedef struct
{
/**< Primary Service Discovery Response callback. */
void (*app_gattc_srvc_disc_cb)(uint8_t conn_idx,
                               uint8_t status, const ble_gattc_srvc_disc_t *
 p_prim_srvc_disc);
/**< Relationship Discovery Response callback. */
void (*app_gattc_inc_srvc_disc_cb)(uint8_t conn_idx,
                                   uint8_t status, const ble_gattc_incl_disc_t *
 p_inc_srvc_disc);
```

```
/**< Characteristic Discovery Response callback. */
void (*app_gattc_char_disc_cb)(uint8_t conn_idx,
                               uint8_t status, const ble_gattc_char_disc_t * p_char_disc);
/**< Descriptor Discovery Response callback. */
void (*app_gattc_char_desc_disc_cb)(uint8_t conn_idx,
                                    uint8_t status,const ble_gattc_char_desc_disc_t
 *p_char_desc_disc);
/**< Read Response callback. */
void (*app_gattc_read_cb)(uint8_t conn_idx, uint8_t status,
                          const ble_gattc_read_rsp_t *p_read_rsp);
/**< Write complete callback. */
void (*app_gattc_write_cb)(uint8_t conn_idx, uint8_t status, uint16_t handle);
/**< Handle Value Notification/Indication Event callback. */
void (*app_gattc_ntf_ind_cb)(uint8_t conn_idx,
                             const ble_gattc_ntf_ind_t *p_ntf_ind);
/**< Service found callback during browsing procedure. */
void app_gattc_srvc_browse_cb(uint8_t conn_idx,
                              uint8_t status, const ble_gattc_browse_srvc_t *p_browse_srvc);
```

Note:

- The ble_gattc_primary_services_discover() API function discovers primary services of the peer device. When primary services are discovered, app_gattc_srvc_disc_cb is called.
- The ble_gattc_included_services_discover() API function discovers included services of the peer device. When included services are discovered, app_gattc_inc_srvc_disc_cb is called.
- The ble_gattc_char_discover() API function discovers characteristics of the peer device. When characteristic declarations are discovered, app_gattc_char_disc_cb is called.
- The ble_gattc_char_desc_discover() API function discovers characteristic descriptors of the peer device. When characteristic descriptors are discovered, app_gattc_char_desc_disc_cb is called.
- The API functions, ble_gattc_read(), ble_gattc_read_by_uuid(), and ble_gattc_read_multiple() read attribute values of the peer device. After attribute values are read, app_gattc_read_cb is called.
- When the host receives notifications and indications from the peer device, app_gattc_ntf_ind_cb is called.
- The ble_gattc_services_browse() API function discovers attributes in one or all services of the peer device. When all the attributes are discovered, app_gattc_srvc_browse_cb is called.

2. Register callback functions of the GATT Client.

```
static app_callback_t s_app_ble_callback =
{
    .app_ble_init_cmp_callback = ble_init_cmp_callback,
    .app_gap_callbacks         = &app_gap_callbacks,
    .app_gatt_common_callback  = NULL,
    .app_gattc_callback        = &app_gattc_callback,
};
```

Note:

Code path:

`SDK_Folder\projects\ble\ble_basic_example\ble_app_gatt_client\Src\user\main.c`

3. Implement write procedures of the GATT Client.

   Browse CCCDs in the discovered characteristics, and write CCCD characteristic configuration values to the peer device, to enable notification for the peer device.

```
//write cccd to enable notification for peer server
uint16_t cccd_value = 0x0001;
if ((BLE_GATTC_BROWSE_ATTR_DESC  ==
      p_browse_srvc->info[fnd_att].attr_type) && (BLE_ATT_DESC_CLIENT_CHAR_CFG == *(uint16_t
 *)
      (p_browse_srvc->info[fnd_att].attr.uuid)))
{
    APP_LOG_DEBUG("[%s] Char Description: attr handle = %04X\n",
                  __func__, (p_browse_srvc->start_hdl + fnd_att + 1));
    if (ble_gattc_write(conn_idx, p_browse_srvc->start_hdl + fnd_att + 1,
                        0, sizeof(uint16_t),(uint8_t *)&cccd_value) == SDK_SUCCESS)
    {
        APP_LOG_DEBUG("[%s] Send write cccd value command!\n", __func__);
}
}
```

4. Applications receive and process responses and notifications from the GATT Server.

   The GATT Client calls the app_gattc_write_cb function registered by users after receiving the ATT_WRITE_RSP data package from the GATT Server.

```
static void app_gattc_write_cb(uint8_t conn_idx, uint8_t status, uint16_t handle)
{
    APP_LOG_DEBUG("[%s]GATT Client Write Completed!" , __func__);
}
```

Note:

Code path:

`SDK_Folder\projects\ble\ble_basic_example\ble_app_gatt_client\Src\user_callback\user_gattc_callback.c`

The GATT Client also receives notifications from the GATT Server, and returns notifications to the application layer by the app_gattc_ntf_ind_cb function registered by users.

```
static void app_gattc_ntf_ind_cb(uint8_t conn_idx, const ble_gattc_ntf_ind_t *p_ntf_ind)
{
    APP_LOG_DEBUG("[%s]enter!" , __func__);

    char *notify_indicate[2] =
    {
        "GATTC_OP_NOTIFICATION",
        "GATTC_OP_INDICATION",
    };

    if (BLE_GATT_NOTIFICATION == p_ntf_ind->type)
    {
        APP_LOG_DEBUG("[%s]type = %s, ", __func__, notify_indicate[0]);
    }
    else if (BLE_GATT_INDICATION == p_ntf_ind->type)
    {
        APP_LOG_DEBUG("[%s]type = %s, ", __func__, notify_indicate[1]);
```

```
    }

    APP_LOG_DEBUG("Attribute handle = %04X, Attribute Value = ", p_ntf_ind->handle);

    for (uint16_t i = 0; i < p_ntf_ind->length; i++)
    {
        if (i == p_ntf_ind->length - 1)
        {
            APP_LOG_DEBUG("%02X", p_ntf_ind->p_value[i]);
        }
        else
        {
            APP_LOG_DEBUG("%02X:" , p_ntf_ind->p_value[i]);
        }
    }

    /* send confirm pdu */
    if (BLE_GATT_INDICATION == p_ntf_ind->type)
    {
        ble_gattc_indicate_cfm(conn_idx, p_ntf_ind->handle);
    }
}
```

Note:

Code path:

`SDK_Folder\projects\ble\ble_basic_example\ble_app_gatt_client\Src\user_callback\user_gattc_callback.c`

3.4.2 GATT Server

The GATT Server receives commands and requests from the peer device (GATT Client), and responds to or sends notifications or indications to the peer device based on the commands and requests received. Functions of the GATT Server are implemented with support from profiles.

The following sections introduce how to add profiles and register profile callback functions in the GATT Server, and how the GATT Server processes read/write requests from the GATT Client.

3.4.2.1 Add Profiles

It is required to add profiles supported by applications during application initialization. Each profile has an API function, which enables applications to add the profile. Two profiles added to BLE Stack are GAP profile and GATT profile.

The following part describes how to add a profile by taking the API function hrs_service_init(hrs_init_t *p_hrs_init) in `SDK_Folder\components\profiles\hrs\hrs.c` as an example.

```
memcpy(&s_hrs_env.hrs_init, p_hrs_init, sizeof(hrs_init_t));
return ble_server_prf_add(&hrs_prf_info);
```

Call the API function ble_server_prf_add() to BLE Stack to register the profile. Information about the initialization function and the callback function of the profile is set to BLE Stack by this API function.

3.4.2.2 Read/Write Callback Functions of Profiles

To handle read/write profile attributes from the peer device (GATT Client), profiles need to define read/write callback functions. These callback functions are registered by running ble_server_prf_add().

Profiles pass information to applications by sending events and receive event handlers registered by users. The registration is completed during application initialization.

For example, events in SDK_Folder\components\profiles\hrs\hrs.h are defined as follows:

```
/**@brief Heart Rate Service event types. */
typedef enum
{
    HRS_EVT_NOTIFICATION_ENABLED,    /**< Heart Rate value notification has been enabled.*/
    HRS_EVT_NOTIFICATION_DISABLED,   /**< Heart Rate value notification has been disabled.*/
    HRS_EVT_RESET_ENERGY_EXPENDED,   /**< The peer device requests to reset Energy
 Expended.*/
    HRS_EVT_READ_BODY_SEN_LOCATION,  /**< The peer device read Body Sensor Location
 characteristic.*/
} hrs_evt_type_t;
/** @} */

/**
 * @defgroup HRS_STRUCT Structures
 * @{
 */
/**@brief Heart Rate Service event. */
typedef struct
{
    uint8_t         conn_idx;    /**< Index of connection. */
    hrs_evt_type_t  evt_type;    /**< Heart Rate Service event type. */
} hrs_evt_t;
/** @} */

/**
 * @defgroup HRS_TYPEDEF Typedefs
 * @{
 */
/**@brief Heart Rate Service  event handler type. */
typedef void (*hrs_evt_handler_t)(hrs_evt_t *p_evt);
/** @} */
```

3.4.2.3 Handle Read Requests from GATT Client

Interactions between the GATT Server and the GATT Client in handling read requests from the GATT Client are shown in Figure 3-5 by taking the Heart Rate Profile as an example:

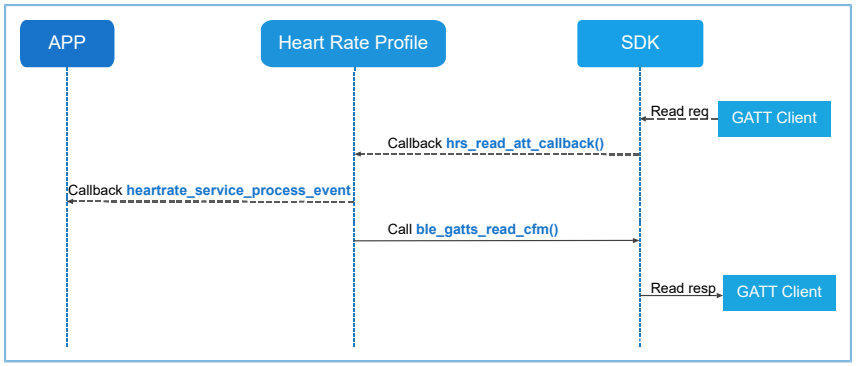

Figure 3-5 GATT Server-Client interactions to handle a read request from the GATT Client

1. When receiving a request for reading an attribute from the GATT Client, BLE Stack first verifies whether the attribute is readable.
2. If the attribute is readable, and the attribute value is stored in user space, BLE Stack calls a read callback function of the profile (the read callback function should be defined already; otherwise, a read timeout event occurs on the peer device).
3. The read callback function of the profile calls an event handler of applications on demand.
4. The read callback function of the profile calls the API function ble_gatts_read_cfm to pass the attribute value to an SDK, and then the attribute value is sent to the GATT Client through BLE Stack.

   The code snippet of the read callback function in SDK_`Folder\components\profiles\hrs\hrs.c` is shown as follows:

```
switch (tab_index)
{
    case HRS_IDX_HR_MEAS_VAL:
        cfm.length = HRS_MEAS_MAX_LEN;
        fm.value  = m_hrs_env.hr_meas.hr_meas_value;
        break;

    case HRS_IDX_HR_MEAS_NTF_CFG:
        cfm.length = sizeof(uint16_t);
        cfm.value  = (uint8_t *)(&(m_hrs_env.ntf_cfg[conn_idx]));
        break;

    case HRS_IDX_BODY_SENSOR_LOC_VAL:
        if (s_hrs_env.hrs_init.evt_handler)
        {
            evt.conn_idx = conn_idx;
            evt.evt_type = HRS_EVT_READ_BODY_SEN_LOCATION;
            s_hrs_env.hrs_init.evt_handler(&evt);
        }
        cfm.length = sizeof(uint8_t);
        cfm.value  = (uint8_t *)(&s_hrs_env.hrs_init.sensor_loc);
        break;

    default:
        cfm.length = 0;
        cfm.status = BLE_ATT_INVALID_HANDLE;
        break;
}

return ble_gatts_read_cfm(conn_idx, &cfm);
```

3.4.2.4 Handle Write Requests from GATT Client

Interactions between the GATT Server and the GATT Client in handling write requests from the GATT Client are shown in Figure 3-6 by taking the heart rate profile as an example:

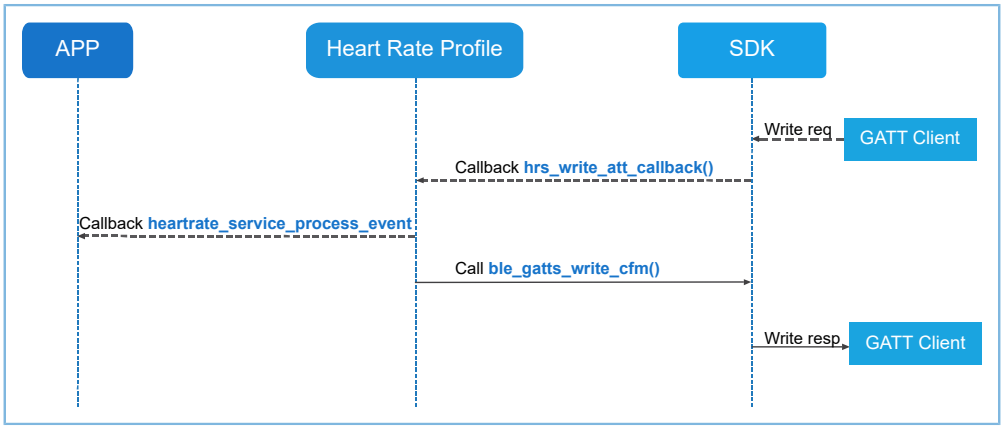

Figure 3-6 GATT Server-Client interactions to handle a write request from the GATT Client

1. When receiving a request to write to an attribute value from the GATT Client, BLE Stack first verifies whether the attribute is writeable.
2. If the attribute is writeable, BLE Stack calls a write callback function (if any) of the profile.
3. The write callback function of the profile calls an event handler of applications on demand. The event handler stores characteristic values to be written. If a CCCD is to be written, the profile should start the notify or indicate process.
4. The write callback function of the profile calls the API function ble_gatts_write_cfm to pass the execution state of the write action to an SDK, and then the state is sent to the GATT Client through BLE Stack.

The code snippet of the write callback in SDK_Folder\components\profiles\hrs\hrs.c is shown as follows:

```
switch (tab_index)
{
 case HRS_IDX_HR_MEAS_VAL:
  cfm.length = HRS_MEAS_MAX_LEN;
  cfm.value  = s_hrs_env.hr_meas.hr_meas_value;
  break;

 case HRS_IDX_HR_MEAS_NTF_CFG:
  cfm.length = sizeof(uint16_t);
  cfm.value  = (uint8_t *)(&(s_hrs_env.ntf_cfg[conn_idx]));
  break;

 case HRS_IDX_BODY_SENSOR_LOC_VAL:
  if (s_hrs_env.hrs_init.evt_handler)
  {
   evt.conn_idx = conn_idx;
   evt.evt_type = HRS_EVT_READ_BODY_SEN_LOCATION;
   s_hrs_env.hrs_init.evt_handler(&evt);
  }
  cfm.length = sizeof(uint8_t);
  cfm.value  = (uint8_t *)(&s_hrs_env.hrs_init.sensor_loc);
  break;

 default:
  cfm.length = 0;
  cfm.status = BLE_ATT_ERR_INVALID_HANDLE;
```

```
  break;
}

ble_gatts_read_cfm(conn_idx, &cfm);
```

3.5 GATT Security

The GATT Server defines permissions for each characteristic independently. It can permit any client to access a certain characteristic, or permit an authenticated or authorized client to access a certain characteristic. The characteristic permission is generally defined as part of the upper-layer profile specification. For a custom profile, users can select proper permissions on demand.

For more information about GATT security, see “Security considerations (Vol 3, Part G)” in the *Bluetooth Core Spec*.

3.5.1 Authentication

To access characteristics which require authentication, the GATT Client shall first finish authenticated pairing. BLE Stack handles permission verification and access control, which does not involve the application layer. However, the application layer needs to declare access authentication requirements of related service attributes during service registration.

Take the service in SDK_`Folder\components\profiles\hrs\hrs.c` as an example. Change the permission of the Heart Rate Measurement characteristic value of the service from “NOTIFY_PERM_UNSEC” (notify-permitted) to “READ_PERM(AUTH)” (readable with authentication). Updated code is provided below:

```
// HR Measurement Characteristic - Value
[HRS_IDX_HR_MEAS_VAL] = {BLE_ATT_CHAR_HEART_RATE_MEAS,
                                READ_PERM(AUTH),
ATT_VAL_LOC_USER,
HRS_MEAS_MAX_LEN},
```

When an unauthenticated client attempts to read the characteristic value, the GATT Server automatically rejects the request and returns an error code BLE_ATT_ERR_INSUFF_AUTHEN(0x05), and does not call the user-defined read callback function.

When an authenticated client attempts to read the characteristic value, the read request is passed to and then handled by the read callback function of the profile.

3.5.2 Authorization

Authorization is a type of permission through which users control attribute access. Users can decide which attributes are accessible. Authorized access to attributes is implemented at the application layer by users. After BLE Stack verifies permissions configured by users in the attribute table (if the verification is successful), it passes read/write requests to read/write callback functions defined by users for handling.

If users do not grant read/write permission to a certain attribute of the GATT Client, it is necessary to set an error code BLE_ATT_ERR_INSUFF_AUTHOR(0x08) in the callback function to indicate insufficient authorization.

4 Security Manager (SM)

The Security Manager (SM) protocol manages Bluetooth security by defining the processes of pairing and key distribution. The figure below shows the modules composing an SM.

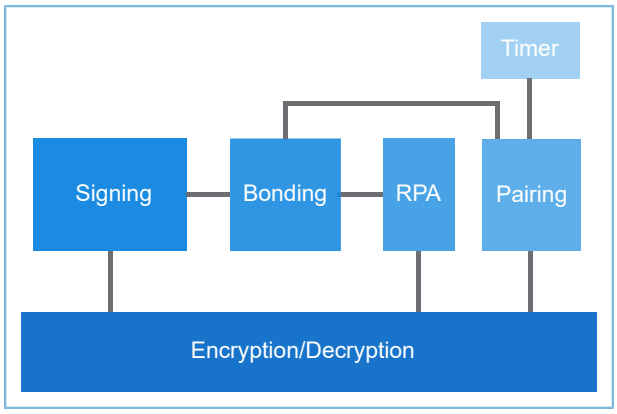

Figure 4-1 SM structure

4.1 Pairing

Pairing enables to establish keys, during which two devices conduct key negotiations and reach an agreement.

Pairing completes in three phases:

1. Exchange pairing information.
2. Generate encryption keys at the link layer.
3. Distribute information of other designated keys on encrypted links, and decide whether it is necessary to store the information of the distributed keys in the security database, based on whether the device can be bonded to.

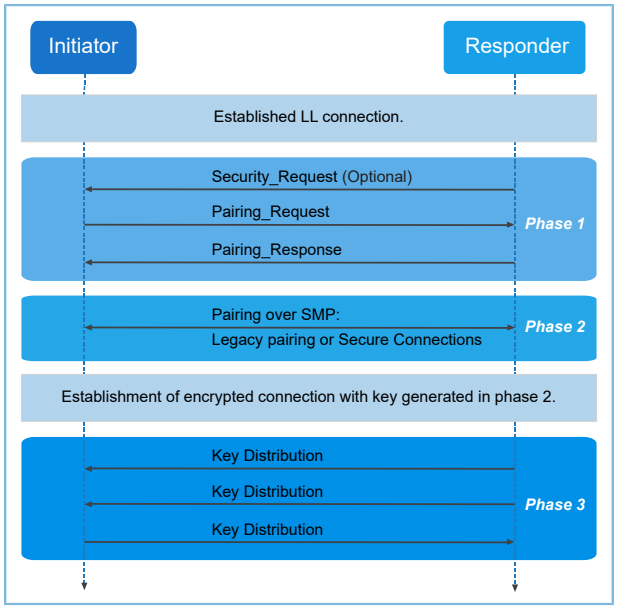

Figure 4-2 Pairing of Bluetooth LE devices

4.1.1 Choose a Pairing Method

Bluetooth Core Specification v4.2 and later versions introduce Secure Connections (SC) pairing, which guarantees more secured operations; in v4.1 and earlier versions, Legacy (LE) pairing is applied. The difference between the two methods is that SC pairing introduces the key agreement protocol, Elliptic Curve Diffie-Hellman, which is not applied in Legacy pairing. Choose a suitable pairing method among the following four, based on the security features of the devices.

- Just Works (Secure Connections or Legacy)
- Passkey Entry (Secure Connections or Legacy)
- Numeric Comparison (Secure Connections)
- Out of Band (Secure Connections or Legacy)

Users can choose a pairing method by following the rules provided below:

- If both the devices support Secure Connections pairing, choose a method according to the rules in the figure below.

		Initiator			
		OOB Set	**OOB Not Set**	**MITM Set**	**MITM Not Set**
Responder	**OOB Set**	Use OOB	Use OOB		
	OOB Not Set	Use OOB	Check MITM		
	MITM Set			Use IO Capabilities	Use IO Capabilities
	MITM Not Set			Use IO Capabilities	Use Just Works

Figure 4-3 Rules for Secure Connections pairing

- If one or more than one of the two devices does not support Secure Connections pairing, choose a method according to the rules in the figure below.

		Initiator			
		OOB Set	**OOB Not Set**	**MITM Set**	**MITM Not Set**
Responder	**OOB Set**	Use OOB	Check MITM		
	OOB Not Set	Check MITM	Check MITM		
	MITM Set			Use IO Capabilities	Use IO Capabilities
	MITM Not Set			Use IO Capabilities	Use Just Works

Figure 4-4 Rules for Legacy pairing

The figure below shows the mapping relationships between I/O capabilities and pairing methods.

	Initiator				
Responder	**DisplayOnly**	**Display YesNo**	**Keyboard Only**	**NoInput NoOutput**	**Keyboard Display**
Display Only	Just Works Unauthenticated	Just Works Unauthenticated	Passkey Entry: responder displays, initiator inputs Authenticated	Just Works Unauthenticated	Passkey Entry: responder displays, initiator inputs Authenticated
Display YesNo	Just Works Unauthenticated	Just Works (For LE Legacy Pairing) Unauthenticated Numeric Comparison (For LE Secure Connections) Authenticated	Passkey Entry: responder displays, initiator inputs Authenticated	Just Works Unauthenticated	Passkey Entry (For LE Legacy Pairing): responder displays, initiator inputs Authenticated Numeric Comparison (For LE Secure Connections) Authenticated
Keyboard Only	Passkey Entry: initiator displays, responder inputs Authenticated	Passkey Entry: initiator displays, responder inputs Authenticated	Passkey Entry: initiator and responder inputs Authenticated	Just Works Unauthenticated	Passkey Entry: initiator displays, responder inputs Authenticated
NoInput NoOutput	Just Works Unauthenticated	Just Works Unauthenticated	Just Works Unauthenticated	Just Works Unauthenticated	Just Works Unauthenticated
Keyboard Display	Passkey Entry: initiator displays, responder inputs Authenticated	Passkey Entry (For LE Legacy Pairing): initiator displays, responder inputs Authenticated Numeric Comparison (For LE Secure Connections) Authenticated	Passkey Entry: responder displays, initiator inputs Authenticated	Just Works Unauthenticated	Passkey Entry (For LE Legacy Pairing): initiator displays, responder inputs Authenticated Numeric Comparison (For LE Secure Connections) Authenticated

Figure 4-5 Mapping relationships between I/O capabilities and pairing methods

4.1.2 Configure Pairing Methods

This section introduces how to adopt pairing methods by configuring security parameters.

Note:

By default, the pairing process examples (ble_app_sm_initiator and ble_app_sm_responder, available in SDK_Folder\projects\ble\ble_basic_example\) adopt Just Works. Users can modify the security parameter s_sec_param in the two example projects to adopt other pairing methods. For details about modifying parameters, refer to the code snippet in the corresponding section.

4.1.2.1 Just Works Pairing

When neither of the two devices requires authentication of man-in-the-middle (MITM) protection, users can choose Just Works pairing. Just Works pairing does not require MITM authentication, and therefore devices choosing this method cannot resist MITM attacks. Just Works pairing can be Legacy pairing or Secure Connections pairing. Users only need to configure the security parameters during initialization, before launching a pairing process. No interaction from users is required.

```
static sec_param_t s_sec_param =
{
    .level = SEC_MODE1_LEVEL1,
    .io_cap = IO_DISPLAY_ONLY,
    .oob = false,
    .auth = AUTH_NONE,
    .key_size = 16,
    .ikey_dist = KDIST_ENCKEY,
    .rkey_dist = KDIST_ENCKEY,
};
ble_sec_params_set(&s_sec_param);
```

Note:

Code paths:

- SDK_Folder\projects\ble\ble_basic_example\ble_app_sm_initiator\Src\user\user_app.c
- SDK_Folder\projects\ble\ble_basic_example\ble_app_sm_responder\Src\user\user_app.c

4.1.2.2 Passkey Entry Pairing

Passkey Entry pairing requires MITM protection authentication and supports Legacy pairing and Secure Connections pairing. Users can start Passkey Entry pairing by configuring the security parameters below and are required to enter the six-digit password in decimal in the pairing process.

```
static sec_param_t s_sec_param =
{
    .level = SEC_MODE1_LEVEL1,
    .io_cap = IO_KEYBOARD_ONLY#
    .oob = false#
    .auth = AUTH_MITM#
    .key_size = 16#
    .ikey_dist = KDIST_ENCKEY#
    .rkey_dist = KDIST_ENCKEY#
};
```

```
ble_sec_params_set#&s_sec_param#;
```

Note:

Code paths:

- `SDK_Folder\projects\ble\ble_basic_example\ble_app_sm_initiator\Src\user\user_app.c`
- `SDK_Folder\projects\ble\ble_basic_example\ble_app_sm_responder\Src\user\user_app.c`

4.1.2.3 Numeric Comparison Pairing

Numeric Comparison pairing only exists for Secure Connections pairing. Users can choose Numeric Comparison pairing when both devices support Secure Connections pairing, display and input (enabled by the I/O capabilities), and are authenticated with MITM protection. An example of configuring security parameters for Numeric Comparison is provided below:

```
static sec_param_t s_sec_param =
{
    .level = SEC_MODE1_LEVEL1,
    .io_cap = IO_DISPLAY_YES_NO,
    .oob = false,
    .auth = AUTH_BOND | AUTH_MITM | AUTH_SEC_CON,
    .key_size = 16,
    .ikey_dist = KDIST_ENCKEY,
    .rkey_dist = KDIST_ENCKEY,
};
ble_sec_params_set#&s_sec_param#;
```

Note:

Code paths:

- `SDK_Folder\projects\ble\ble_basic_example\ble_app_sm_initiator\Src\user\user_app.c`
- `SDK_Folder\projects\ble\ble_basic_example\ble_app_sm_responder\Src\user\user_app.c`

4.1.2.4 Pairing Disabling

Users can disable pairing by setting the pairing value to false in the SDK API function. After pairing is disabled, the BLE Stack rejects all packets requesting pairing.

```
ble_gap_pair_enable_set#false#;
```

Note:

Code paths:

- `SDK_Folder\projects\ble\ble_basic_example\ble_app_sm_initiator\Src\user\user_app.c`
- `SDK_Folder\projects\ble\ble_basic_example\ble_app_sm_responder\Src\user\user_app.c`

4.2 Bonding

Bonded key information is used for link re-encryption, signature authentication on data, and address parsing after devices are re-connected. In re-encryption after reconnection, for devices that have been bonded to each other before, the bonded key information is used for link encryption; for devices that have not, the pairing process will be launched.

4.2.1 Enable Bonding

After bonding is enabled, the interactions between applications and BLE Stack are shown in the figure below.

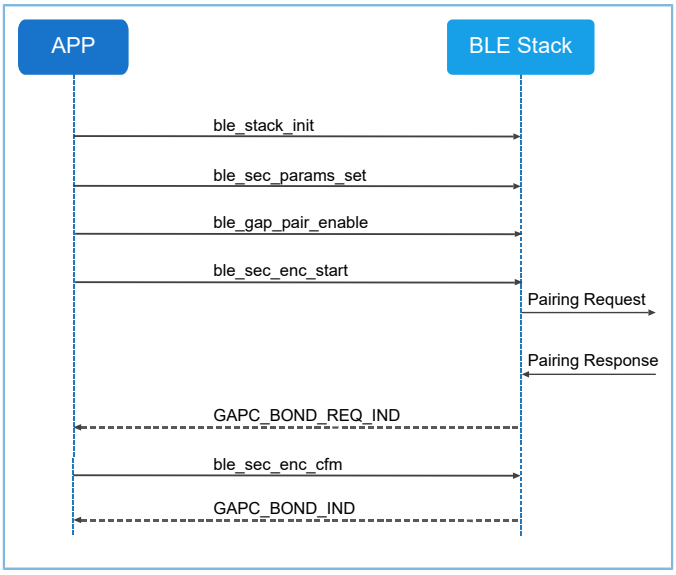

Figure 4-6 Interactions after bonding is enabled

Steps to enable bonding:

Note:

Code snippets in the following steps are extracted from the pairing process examples: ble_app_sm_initiator and ble_app_simple_sm_responder (in `SDK_Folder\projects\ble\ble_basic_example\`).

1. Configure the security parameters, including the I/O capabilities, the bonding status, MITM protection, and keys to be distributed for the devices.

```
//set the default security parameters.
```



```
static sec_param_t s_sec_param =
{
    .level = SEC_MODE1_LEVEL1,
    .io_cap = IO_DISPLAY_ONLY,
    .oob = false,
    .auth = AUTH_BOND,
    .key_size = 16,
    .ikey_dist = KDIST_ENCKEY,
    .rkey_dist = KDIST_ENCKEY,
};
ble_sec_params_set#&s_sec_param#;
```

Note:

Code paths:

- `SDK_Folder\projects\ble\ble_basic_example\ble_app_sm_initiator\Src\user\user_app.c`
- `SDK_Folder\projects\ble\ble_basic_example\ble_app_sm_responder\Src\user\user_app.c`

2. Enable pairing.

```
ble_gap_pair_enable#true#;
```

Note:

Code paths:

- `SDK_Folder\projects\ble\ble_basic_example\ble_app_sm_initiator\Src\user_callback\user_sm_callback.c`
- `SDK_Folder\projects\ble\ble_basic_example\ble_app_sm_responder\Src\user_callback\user_sm_callback.c`

3. Implement and register relevant callback functions.

   (1). Implement callback functions.

```
static void app_sec_rcv_enc_req_cb(uint8_t conn_idx, sec_enc_req_t *p_enc_req)
{
    APP_LOG_DEBUG("rcv enc req cb\n");

    const uint32_t tk = 123456;
    sec_cfm_enc_t cfm_enc;

    memset((uint8_t *)&cfm_enc, 0, sizeof(sec_cfm_enc_t));

    if (NULL == p_enc_req)
    {
        return;
    }
   switch (p_enc_req->req_type)
   {
       case PAIR_REQ:
```

```
        {
            APP_LOG_DEBUG("pair req\n");
            cfm_enc.req_type = PAIR_REQ;
            cfm_enc.accept = true;
            break;
         }
         case TK_REQ:
         {
             APP_LOG_DEBUG("tk req\n");
             cfm_enc.req_type = TK_REQ;
             cfm_enc.accept = true;
             memset(cfm_enc.data.tk.key, 0, 16);
             cfm_enc.data.tk.key[0] = (uint8_t)((tk & 0x000000FF) >> 0);
             cfm_enc.data.tk.key[1] = (uint8_t)((tk & 0x0000FF00) >> 8);
             cfm_enc.data.tk.key[2] = (uint8_t)((tk & 0x00FF0000) >> 16);
             cfm_enc.data.tk.key[3] = (uint8_t)((tk & 0xFF000000) >> 24);
             break;
         }
         case OOB_REQ:
         {
             APP_LOG_DEBUG("oob req\n");
             break;
         }
         case NC_REQ:
         {
             APP_LOG_DEBUG("nc req\n");
             uint32_t num = *(uint32_t *)(p_enc_req->data.nc_data.value);
             APP_LOG_DEBUG("num=%d\n", num);
             cfm_enc.req_type = NC_REQ;
             cfm_enc.accept = true;
             break;
         }
         default:
             break;
     }
     ble_sec_enc_cfm(conn_idx, &cfm_enc);
}
…
const sec_cb_fun_t app_sec_callback = {
.app_sec_enc_req_cb = app_sec_enc_req_cb#
.app_sec_enc_ind_cb = app_sec_enc_ind_cb#
.app_sec_keypress_notify_cb = NULL
};
```

Note:

Code paths:

- `SDK_Folder\projects\ble\ble_basic_example\ble_app_sm_initiator\Src\user_callback\user_sm_callback.c`
- `SDK_Folder\projects\ble\ble_basic_example\ble_app_sm_responder\Src\user_callback\user_sm_callback.c`

(2). Register callback functions.

```
static app_callback_t m_app_ble_callback =
{
    .app_ble_init_cmp_callback = ble_init_cmp_callback,
```

```
    .app_gap_callbacks          = &app_gap_callbacks,
    .app_sec_callback           = &app_sec_callback,
};
// Initialize ble stack.
ble_stack_init(&s_app_ble_callback, &heaps_table);
```

Note:

Code paths:

- `SDK_Folder\projects\ble\ble_basic_example\ble_app_sm_initiator\Src\user\main.c`
- `SDK_Folder\projects\ble\ble_basic_example\ble_app_sm_responder\Src\user\main.c`

4. Encrypt the link in callback functions where devices are connected.

```
static void app_gap_connect_cb(uint8_t conn_idx, uint8_t status,const gap_conn_cmp_t
 *p_conn_param)
{
APP_LOG_DEBUG("Enter connect complete cb, conidx=%d\n", conn_idx);
ble_sec_enc_start(conn_idx);
}
```

Note:

Code path:

`SDK_Folder\projects\ble\ble_basic_example\ble_app_sm_initiator\Src\user_callback\user_gap_callback.c`

4.3 Privacy Management

In Bluetooth LE technology, according to the privacy policy, authenticated devices can track and identify the target devices, whereas unauthenticated devices cannot. Privacy management enables authenticated devices to connect to and communicate with target devices, and prevents tracking from unauthenticated devices and malicious devices.

4.3.1 Enable Privacy Management

During initialization, a Bluetooth LE device automatically loads the address parsing lists of bonded devices and configures the addresses for protocol stacks. Users shall enable privacy management and set the time for address update and set the identity address of the peer device in advertising parameters, which facilitates protocol stacks to identify the relevant addressing parsing list based on the identity address, so as to generate a resolvable private address (RPA).

Follow the steps below to enable privacy management:

Note:

Code snippets in the steps below are extracted from the privacy management examples: ble_app_privacy_slave and ble_app_privacy_master (in SDK_Folder\projects\ble\ble_basic_example\).

1. Connect and bond the master and the slave. During bonding, the two devices exchange the identity resolving keys (IRKs) and identity addresses; after bonding, disconnect the master from the slave. See "Section 4.2 Bonding" for details about the bonding process.
2. Configure the slave privacy policy.

   (1). Set the privacy parameters, and set the update interval of RPA as 150 seconds.

```
ble_gap_privacy_params_set(150#true);
```

   (2). Obtain the identity address of the peer device from the address parsing list, such as the identity address of the first device in the list.

```
// get bond list
bond_dev_list_t bond_list;
memset(&bond_list, 0, sizeof(bond_dev_list_t));
ble_gap_bond_devs_get(&bond_list);

APP_LOG_DEBUG("bond list size = %d\n", bond_list.num);
APP_LOG_DEBUG("addr_type = %d\n", bond_list.items[0].addr_type);
for (uint8_t i = 0; i < 6; i++)
{
    APP_LOG_DEBUG("addr[%d] = 0x%x ", i,
                  bond_list.items[0].gap_addr.addr[i]);
}
APP_LOG_DEBUG("\n");
```

   (3). Set the identity address of the peer device in the advertising parameters.

```
// set peer identity addr
memcpy(g_gap_adv_param.peer_addr.gap_addr.addr,
bond_list.items[0].gap_addr.addr, 6);
g_gap_adv_param.peer_addr.addr_type = bond_list.items[0].addr_type;
```

   (4). Reset the advertising parameters, and start advertising.

```
// set adv param and start adv again
ble_gap_adv_param_set(0, BLE_GAP_OWN_ADDR_STATIC, &g_gap_adv_param);
ble_gap_adv_data_set(0, BLE_GAP_ADV_DATA_TYPE_DATA, s_adv_data_set,
                        sizeof(s_adv_data_set));
ble_gap_adv_data_set(0, BLE_GAP_ADV_DATA_TYPE_SCAN_RSP,
                        s_adv_rsp_data_set, sizeof(s_adv_rsp_data_set));
ble_gap_adv_start(0, &g_gap_adv_time_param);
```

Note:

Code path for steps (1) to (4):

`SDK_Folder\projects\ble\ble_basic_example\ble_app_privacy_slave\Src\user_callback\user_gap_callback.c`

3. Configure the master privacy policy.

   (1). Obtain the identity address of the peer device from the address parsing list, such as the identity address of the first device in the list.

```
// get bond list
bond_dev_list_t bond_list;
memset(&bond_list, 0, sizeof(bond_dev_list_t));
ble_gap_bond_devs_get(&bond_list);
APP_LOG_DEBUG("bond list size = %d\n", bond_list.num);
APP_LOG_DEBUG("addr_type = %d\n", bond_list.items[0].addr_type);
for (uint8_t i = 0; i < 6; i++)
{
    APP_LOG_DEBUG("addr[%d] = 0x%x ", i,
    bond_list.items[0].gap_addr.addr[i]);
}
APP_LOG_DEBUG("\n");
// save peer identity addr
memcpy(g_peer_iden_addr.gap_addr.addr,
          bond_list.items[0].gap_addr.addr, 6);
g_peer_iden_addr.addr_type = bond_list.items[0].addr_type;
```

Note:

Code path:

`SDK_Folder\projects\ble\ble_basic_example\ble_app_privacy_master\Src\user_callback\user_sm_callback.c`

   (2). Set scanning parameters, and enable privacy management.

```
// set privacy params
ble_gap_privacy_params_set(150, true);

// start scan
gap_scan_param_t scan_param;
scan_param.scan_type = GAP_SCAN_ACTIVE;
scan_param.scan_mode = GAP_SCAN_GEN_DISC_MODE;
scan_param.scan_dup_filt = GAP_SCAN_FILT_DUPLIC_EN;
scan_param.use_whitelist = 1;
scan_param.interval= 15;
scan_param.window= 15;
scan_param.timeout = 0;
ble_gap_scan_param_set(BLE_GAP_OWN_ADDR_STATIC, &scan_param);
```

   (3). Start scanning.

```
ble_gap_scan_start();
```

(4). Set the connection parameters in the advertising callback function, and initiate requests for connection.

```
static void app_gap_adv_report_ind_cb(const gap_ext_adv_report_ind_t  *p_adv_report)
{
    if (memcmp(p_adv_report->broadcaster_addr.gap_addr.addr,
         g_peer_iden_addr.gap_addr.addr, 6) == 0)
    {
        APP_LOG_DEBUG("scan success\n");
        // connect peer device again
        gap_init_param_t conn_param;
        memcpy(conn_param.peer_addr.gap_addr.addr,
                 g_peer_iden_addr.gap_addr.addr, 6);
        conn_param.peer_addr.addr_type = g_peer_iden_addr.addr_type;
        conn_param.type = GAP_INIT_TYPE_DIRECT_CONN_EST;
        conn_param.interval_min = 6;
        conn_param.interval_max = 10;
        conn_param.slave_latency = 1;
        conn_param.sup_timeout = 100;
         ble_gap_connect(BLE_GAP_OWN_ADDR_STATIC, &conn_param);
    }
}
```

Note:

Code path for steps (2) to (4):

`SDK_Folder\projects\ble\ble_basic_example\ble_app_privacy_master\Src\user_callback\user_gap_callback.c`

4.3.2 Address Configuration Description

When devices start advertising, scanning, and establishing connections, the controller can send air interface packets through different addresses, based on the specific configuration. This section describes how to configure the addresses used for air interfaces, by taking advertising as an example.

1. Use the ble_gap_addr_set API function to set the identity address as public or static. If the function has not been called before and default public addresses are unavailable in eFuse/NVDS, by default, a static address is generated and set as the identity address, based on the UUID of the chip.
2. Configure the address passed from host layer to the controller through own_addr_type, the parameter of the ble_gap_adv_param_set API function.
   (1). BLE_GAP_OWN_ADDR_STATIC: Sends the identity address (configured in Step 1) to the controller.
   (2). BLE_GAP_OWN_ADDR_GEN_RSLV: Sends the RPA generated at the host layer to the controller.
   (3). BLE_GAP_OWN_ADDR_GEN_NON_RSLV: Sends the non-RPA generated at the host layer to the controller.
3. Enable or disable privacy management by configuring enable_flag, a parameter in ble_gap_privacy_params_set.
   (1). If privacy management is not enabled, the air interfaces use the addresses configured for the controller (as described in Step 1).

(2). If privacy management is enabled, the controller searches for the address parsing list based on the peer addresses passed through advertising parameters. If the searching fails, the air interfaces use the addresses configured at the host layer for the controller.

(3). If privacy management is enabled, and the controller finds out the address parsing list based on the peer addresses passed through advertising parameters, use the RPA generated automatically from the controller based on the IRKs of the peer address.

5 Logical Link Control and Adaptation Protocol (L2CAP)

The Logical Link Control and Adaptation Protocol (L2CAP) plays a key role in Bluetooth communications. It permits transmission, receiving, reorganizing, and unpacking of Asynchronous Connectionless (ACL) packets. It also allows creating connection-oriented channels (COCs) by sending signaling packets through L2CAP. The figure below is a block diagram of L2CAP.

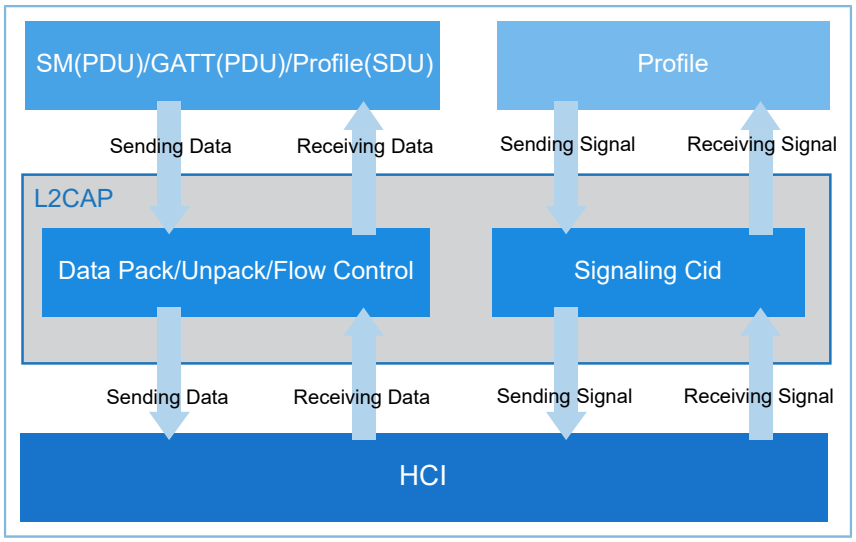

Figure 5-1 L2CAP structure

5.1 L2CAP Data Packet Structure

A service data unit (SDU) refers to a data packet transmitted from upper layers to underlying protocols. Such data packets target at the application layer, and are mainly applied to dynamic channel services created through COCs. A protocol data unit (PDU) refers to a data packet at the L2CAP layer. An SDU can be unpacked into one or more PDUs at the L2CAP layer. Each PDU has a 32-bit header at the payload front-end. Therefore, the length of a data packet shall be stored in the header, so as to identify the end of the data packet.

The data packet structure of a PDU is shown in the figure below:

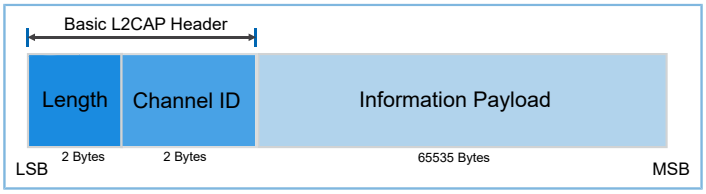

Figure 5-2 Data packet structure of a PDU

The data packet structure of an SDU is shown in the figure below:

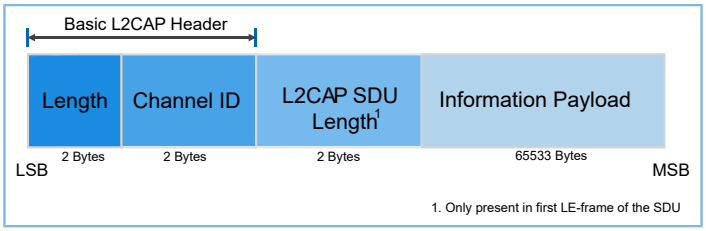

Figure 5-3 Data packet structure of an SDU

A header contains a 2-byte length field and a 2-byte channel ID. The length field shows the length of information payload in byte following the header. It should be noted that in an SDU, the first two bytes after the header in the first frame shows the payload length of the SDU.

5.2 Maximum Transmission Unit (MTU)

L2CAP permits higher-level protocols (such as SM and GATT) to transmit upper-layer data packets, and the maximum size of data packets that L2CAP layer entities can accept is known as the maximum transmission unit (MTU). If a data packet transmitted by higher-level protocols exceeds the MTU, the data packet shall be unpacked on air interfaces.

A PDU is divided into fragments based on the length from the controller, as defined in ACL_Data_Packet_Length. The figure below describes the fragmentation of a PDU:

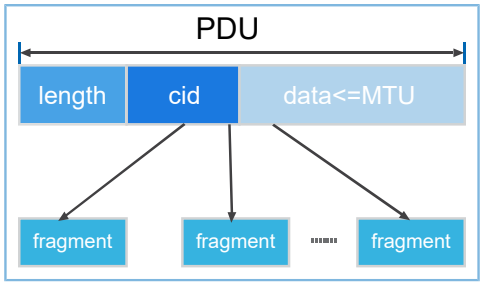

Figure 5-4 PDU fragmentation

An SDU is first divided into segments, based on the maximum PDU payload size (MPS). Each segment is then divided into fragments based on the length defined in ACL_Data_Packet_Length. The following figure shows the segmentation of an SDU:

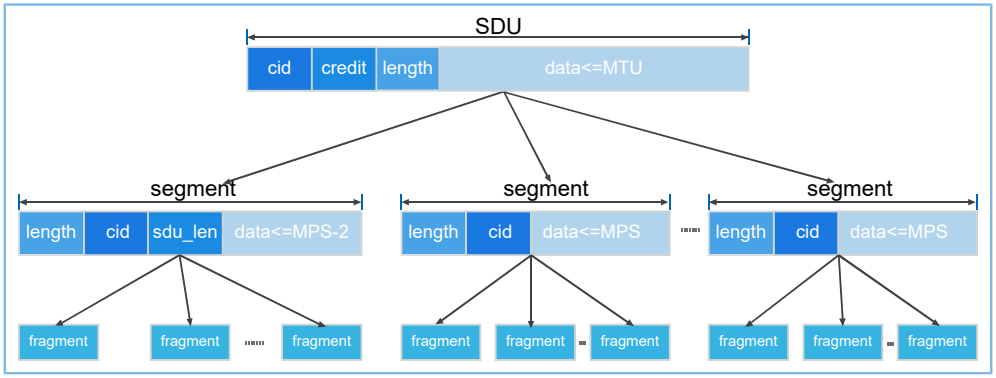

Figure 5-5 SDU segmentation

Sample code to configure MTUs and MPSs is provided below:

```
// mtu:23#2048# mps:23#mtu# lecb_conn_num: 0x00#0x20
error_code = ble_gap_l2cap_params_set(512, 250, 10);
APP_ERROR_CHECK(error_code);
```

Note:

Code path:

`SDK_Folder\projects\ble\ble_basic_example\ble_app_l2cap_coc_server\Src\user\user_app.c`

5.3 L2CAP Channels

Bluetooth LE has fixed channels (from 0x0004 to 0x0006), which exist the moment two devices are connected and do not require parameter configuration, as well as COCs, which are dynamically created.

The table below lists the channel identifiers, which are 16-bit numbers. The channel identifier 0x0000 is reserved and shall not be used. The channel identifier 0x0001 is the fixed channel ID for Bluetooth Classic signaling.

Table 5-1 L2CAP channel identifiers

Channel Identifier	Description
0x0000	Shall not be used
0x0001 - 0x0003	Reserved, can be used in future
0x0004	ATT
0x0005	Low-power signaling channel
0x0006	SM protocol
0x0007 – 0x001F	Reserved, can be used in future
0x0020 – 0x003E	Assigned by Bluetooth SIG
0x003F	Reserved, can be used in future
0x0040 – 0x007F	COC

5.4 Connection-oriented Channel (COC)

A connection-oriented channel (COC) is a major feature of an L2CAP controller. A COC allows an LE service to create a specific channel on designated links. A COC channel should be created before a server exchanges data with a client. COCs stand out for allowing application layers to send large data packets by configuring the MTU and the MPS, so as to boost the throughput of systems. Internet Protocol Support Profile (IPSP) and Object Transfer Profile (OTP) are two typical application scenarios.

When creating a COC, the client initiates a request for creating a COC based on the designated protocol/service multiplexer (PSM). To ensure the server can accept the request, users must register the PSM at the application layer, because any request based on an unregistered PSM is ignored on the server. PSM values can be fixed or dynamic, as shown in the table below.

Table 5-2 PSM value description

Range	Type	Description
0x0001 – 0x007F	The PSM values assigned by Bluetooth SIG	For existing standard services
0x0080 – 0x00FF	Dynamically allocated user-defined PSM values	Designated by user-defined services
0x0100 – 0xFFFF	Reserved	Reserved

When registering PSMs, users can also designate the authentication rights of services.

```
gap_lepsm_register_t param;
param.le_psm = 0x25;
param.sec_lvl = 0x00;
param.mks_flag = false;
error_code = ble_gap_lepsm_register(&param);
APP_ERROR_CHECK(error_code);
```

Note:

Code path:

`SDK_Folder\projects\ble\ble_basic_example\ble_app_l2cap_coc_server\Src\user\user_app.c`

5.4.1 Process for Creating a COC

The SDK provides APIs to create L2CAP COCs, to enable two-way transmission of data between two devices on which the function is supported.

Interactions between applications and BLE Stack when a COC is created are described in the figure below.

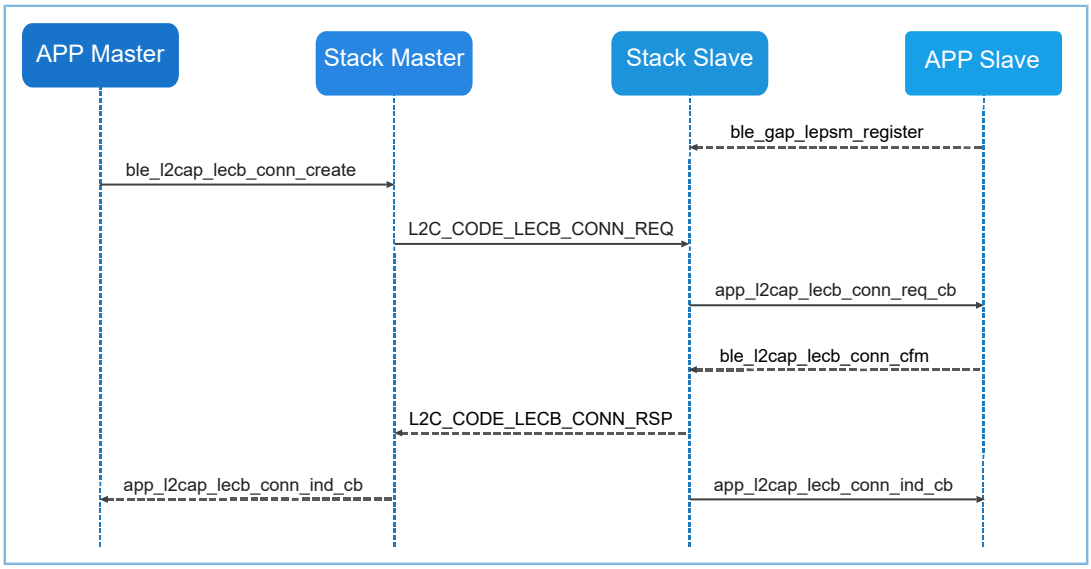

Figure 5-6 Process for creating a COC

Steps:

1. During initialization, applications on the server shall first register the PSM on GAP. Code snippets in the steps are extracted from the SDK example, ble_app_l2cap_coc_server (in SDK_Folder\projects\ble\ble_basic_example\), which explains process of creating a COC.

```
gap_lepsm_register_t param;
param.le_psm = 0x25;
param.sec_lvl = 0x00;
param.mks_flag = false;
error_code = ble_gap_lepsm_register(&param);
APP_ERROR_CHECK(error_code);
```

Note:

Code path:

SDK_Folder\projects\ble\ble_basic_example\ble_app_l2cap_coc_server\Src\user\user_app.c

2. After the link connection is established, the client initiates a request for creating a COC based on the designated PSM.

```
lecb_conn_req_t conn_req;
conn_req.le_psm = psm;
conn_req.local_credits = 0xffff;
conn_req.local_cid = 0;
conn_req.mtu = 512;
conn_req.mps = 230;
ble_l2cap_lecb_conn_create(0, &conn_req);
```

Note:

Code path:

SDK_Folder\projects\ble\ble_basic_example\ble_app_l2cap_coc_client\Src\user_callback\user_gap_callback.c

3. The server retransmits the request to the application layer after receiving the request.
4. The application layer decides whether to accept the request, and sends a response to the peer device. A COC will be created if the request is accepted.

```
static void app_l2cap_lecb_conn_req_cb(uint8_t conn_idx,
 lecb_conn_req_ind_t *p_conn_req)
{
    APP_LOG_DEBUG("app rcv lecb con req\n");
    APP_LOG_DEBUG("peer_mtu = %d, peer_mps = %d\n", p_conn_req->peer_mtu,
                      p_conn_req->peer_mps);
    lecb_cfm_conn_t cfm_conn;
    cfm_conn.accept = true;
    cfm_conn.peer_cid = p_conn_req->peer_cid;
    cfm_conn.local_credits = 0xffff;
    cfm_conn.local_cid = 0;
    cfm_conn.mtu = 512;
    cfm_conn.mps = 230;
```

```
    ble_l2cap_lecb_conn_cfm(conn_idx, &cfm_conn);
}
```

Note:

Code path:

`SDK_Folder\projects\ble\ble_basic_example\ble_app_l2cap_coc_server\Src\user_callback\user_l2cap_callback.c`

6 Glossary and Abbreviations

Table 6-1 Glossary and abbreviations

Name	Description
ACL	Asynchronous Connectionless
ATT	Attribute Protocol
Bluetooth LE	Bluetooth Low Energy
CCCD	Client Characteristic Configuration Descriptor
COC	Connection-oriented Channel
GAP	Generic Access Profile
GATT	Generic Attribute Profile
GFSK	Gaussian Frequency-Shift Keying
HAL	Hardware Abstraction Layer
HCI	Host-Controller Interface
IPSP	Internet Protocol Support Profile
L2CAP	Logical Link Control and Adaptation Protocol
LE	Legacy
LECB	LE Credit Based Connection
LL	Link Layer
LSB	Least Significant Bit
MIC	Message Integrity Check
MITM	Man-in-the-Middle
MSB	Most Significant Bit
MTU	Maximum Transmission Unit
NVDS	Non-volatile Data Storage
OTP	Object Transfer Profile
PDU	Protocol Data Unit
PHY	Physical Layer
SC	Secure Connections
SDK	Software Development Kit
SDU	Service Data Unit
SM	Security Manager
SoC	System on Chip
UUID	Universally Unique Identifier