# Airoha IoT SDK LE Audio Developers Guide

Version: 1.3

Release date: 30 Aug 2022

## Document revision history

| Revision | Date | Description |
|---|---|---|
| 1.0 | 25 December 2020 | Initial release |
| 1.1 | 18 October 2021 | Add synchronization to BMS |
| 1.2 | 6 May 2022 | Add section 1.4 Supported Profiles |
| 1.3 | 30 Aug 2022 | Update supported profiles |

## Table of contents

## Lists of tables and figures

# 1. Introduction

This document introduces the Bluetooth Low Energy Audio (LE Audio) that is supported by the Airoha IoT SDK and helps developers to understand the design and code structure of Airoha IoT SDK LE Audio It also provides details about the methods to do customizations.

This document guides you through:

- Support for LE Audio with the library description and supported reference examples.
- Detailed descriptions LE Audio services and profiles.
- Custom application development and debugging logs.

## 1.1. Architecture

The LE Audio architecture is shown in Figure 1.

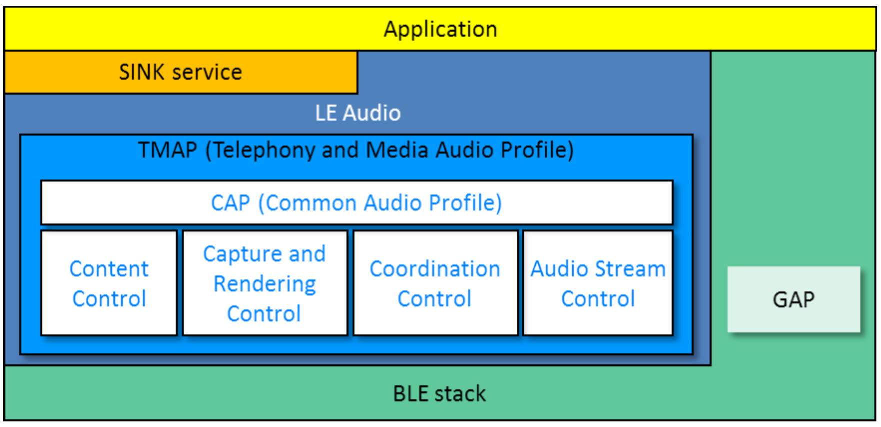

***Figure 1. Bluetooth LE Audio stack architecture***

The BLE stack is described in the *Airoha_IoT_SDK_Bluetooth_Developers_Guide.docx* document. Application layer communicates with BLE stack to let the device connected with smart device (i.e. ADV setting and link establishment). After the connecting process is complete, LE Audio layer communicates with BLE stack and sink service for ACL data transfer and audio stream control. Bluetooth Telephony and Media Audio features are defined in Telephony and Media Audio Profile (TMAP), and these features are enabled by setting the TMAP role. Please refer to 1.2 Configurations and Roles for supported TMAP roles. Common Audio Profile (CAP) specifies procedures to control Audio Stream, volume and device input by using the corresponded profiles and services.

- Content Control: A set of Generic Attribute Profile (GATT)-based profiles used to control audio content, such as Media Control Profile (MCP), and Call Control Profile (CCP).
- Capture and Rendering Control: A set of GATT-based services used to control volume and microphone, such as Volume Control Service (VCS), Microphone Control Service (MICS), Volume Offset Control Service (VOCS), and Audio Input Control Service (AICS).
- Audio Stream Control: A set of GATT-based services used to control the Audio Streams that carry that content to devices, including coordinated sets of devices, such as Published Audio Capabilities Service (PACS), Audio Stream Control Service (ASCS), and Coordinated Set Identification Service (CSIS).

For more information on the LE Audio specifications, refer to the Bluetooth Special Interest Group website.

## 1.2. Configurations and Roles

TMAP defines the following roles for LE Audio device.

- Call Gateway (CG): A CG device has the connection to the call network infrastructure, such as smartphones, laptops, tablets, and PCs.
- Call Terminal (CT): Headset type devices in telephony or VoIP applications, which have wireless headsets, speakers, and microphones that can participate in conversational audio through CG device.
- Unicast Media Sender (UMS): UMS devices send media audio content to a sink device in a Unicast Audio Stream, such as smartphones, media players, TVs, laptops, tablets, and PCs.
- Unicast Media Receiver (UMR): UMR devices receive media audio content from a source device in a Unicast Audio Stream, such as headphones, earbuds, and wireless speakers.
- Broadcast Media Sender (BMS): BMS devices broadcast audio stream in encrypted or non-encrypted packets, such as smartphones, media players, TVs, laptops, tablets, and PCs.
- Broadcast Media Receiver (BMR): BMR devices receive audio stream by synchronizing to the BMS, such as the headphones, earbuds, and wireless speakers.

The example implementations that use TMAP roles are shown in Figure 2.

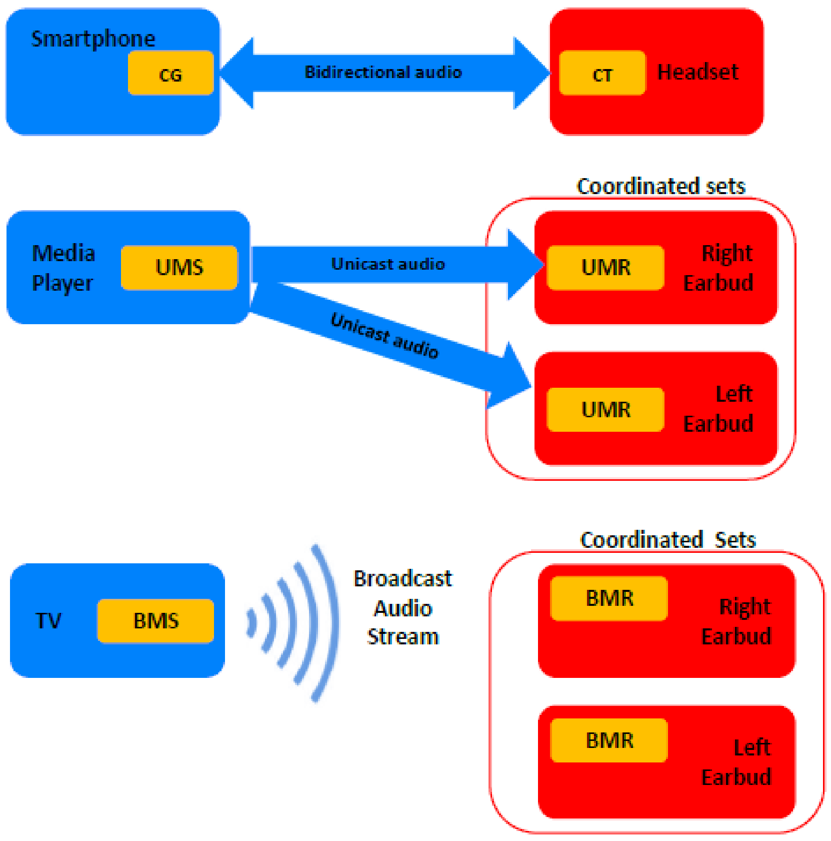

*Figure 2. TMAP roles examples*

## 1.3. Related SDK library requested

Airoha LE Audio is run on the Airoha IoT SDK for Bluetooth platform with the requested library files to interface the Bluetooth with C source and header files related to the platform, as shown in Table 1.

***Table 1. Airoha IoT SDK library support for LE Audio***

| Module | File Name | Location | Function |
|---|---|---|---|
| Bluetooth | libbt_la.a | /prebuilt/middleware/MTK/le_audio/lib/ | BR/EDR and Bluetooth LE stack library (Dual mode) |
| | libble_la.a | | Bluetooth LE stack library (LE only) |
| | libpka_leaudio.a | | Bluetooth controller library (with LE Audio functions) |
| | libbt_leaudio.a | | LE Audio library |
| | bt_gap_le.h | /mcu/prebuilt/middleware/MTK/bluetooth/inc | Interface for the GAP for Bluetooth LE support |
| | bt_gatt.h | | GATT UUID |
| | bt_gatts.h | | Interface for the GATT server |
| | bt_type.h | | Common data types |
| | bt_gap_le_audio.h | /mcu/prebuilt/middleware/MTK/le_audio/inc | Interface for the GAP for LE Audio support |
| | bt_le_audio_sink.h | /mcu/middleware/MTK/le_audio/inc | Interface for the LE Audio sink device |
| | ble_ascs.h | /mcu/middleware/MTK/le_audio/inc/ascs | Interface for the ASCS |
| | ble_ascs_def.h | | Common data types for the ASCS |
| | ble_bap.h | /mcu/middleware/MTK/le_audio/inc/bap | Interface for the BAP |
| | ble_bass.h | /mcu/middleware/MTK/le_audio/inc/bass | Interface for the BASS |
| | ble_ccp.h | /mcu/middleware/MTK/le_audio/inc/ccp | Interface for the CCP |
| | ble_ccp_discovery.h | | Interface for CCP setting TBS record |
| | ble_csis.h | /mcu/middleware/MTK/le_audio/inc/csip | Interface for the CSIS |
| | ble_csis_def.h | | Common data types for the CSIS |
| | ble_mcp.h | /mcu/middleware/MTK/le_audio/inc/mcp | Interface for the MCP |
| | ble_mcp_discovery.h | | Interface for MCP setting MCS record |
| | ble_mics.h | /mcu/middleware/MTK/le_audio/inc/micp | Interface for the MICS |
| | ble_mics_def.h | | Common data types for the MICS |
| | ble_pacs.h | /mcu/middleware/MTK/le_audio/inc/pacs | Interface for the PACS |
| | ble_pacs_def.h | | Common data types for the PACS |
| | ble_tmas_def.h | /mcu/middleware/MTK/le_audio/inc/tmap | Interface for the TMAS |
| | ble_aics.h | /mcu/middleware/MTK/le_aud | Interface for the AICS |

| Module | File Name | Location | Function |
|---|---|---|---|
| | `ble_aics_def.h` | io/inc/vcp | Common data types for the AICS |
| | `ble_vcs.h` | | Interface for the VCS |
| | `ble_vcs_def.h` | | Common data types for the VCS |
| | `ble_vocs.h` | | Interface for the VOCS |
| | `ble_vocs_def.h` | | Common data types for the VOCS |
| | `bt_le_audio_def.h` | /mcu/middleware/MTK/le_audio/inc/util | Common data types for the LE Audio |
| | `bt_le_audio_type.h` | | Common data types for the LE Audio |
| | `ble_ascs_service.c` | /mcu/middleware/MTK/le_audio/src/ascs | Service record for the ASCS |
| | `ble_csis_service.c` | /mcu/middleware/MTK/le_audio/src/csip | Service record for the CSIS |
| | `ble_mics_service.c` | /mcu/middleware/MTK/le_audio/src/ | Service record for the MICS |
| | `ble_pacs_service.c` | /mcu/middleware/MTK/le_audio/src/pacs | Service record for the PACS |
| | `ble_aics_service.c` | /mcu/middleware/MTK/le_audio/src/vcp | Service record for the AICS |
| | `ble_vcs_service.c` | | Service record for the VCS |
| | `ble_vocs_service.c` | | Service record for the VOCS |

## 1.4. Supported Profiles

The supported LE Audio profiles are shown in Table 2. Supported LE Audio profiles. For more details of LE Audio specifications, please refer to https://www.bluetooth.com/specifications/specs/ .

***Table 2. Supported LE Audio profiles***

| Profile | Description | Version |
|---|---|---|
| AICS | Audio Input Control Service | 1.0 |
| ASCS | Audio Stream Control Service | 1.0 |
| BAP | Basic Audio Profile | 1.0.1 |
| BASS | Broadcast Audio Scan Service | 1.0 |
| CCP | Call Control Profile | 1.0 |
| CAP | Common Audio Profile | 1.0 |
| CAS | Common Audio Service | 1.0 |
| CSIP | Coordinated Set Identification Profile | 1.0.1 |
| CSIS | Coordinated Set Identification Service | 1.0.1 |
| GMCS | Generic Media Control Service | 1.0 |
| GTBS | Generic Telephone Bearer Service | 1.0 |
| HAP | Hearing Access Profile | 1.0 |
| HAS | Hearing Access Service | 1.0 |

| Profile | Description | Version |
|---|---|---|
| MCP | Media Control Profile | 1.0 |
| MCS | Media Control Service | 1.0 |
| MICP | Microphone Control Profile | 1.0 |
| MICS | Microphone Control Service | 1.0 |
| PBP | Public Broadcast Profile | 1.0 |
| PACS | Published Audio Capabilities Service | 1.0 |
| TBS | Telephone Bearer Service | 1.0 |
| TMAP | Telephony and Media Audio Profile | 1.0 |
| VCP | Volume Control Profile | 1.0 |
| VCS | Volume Control Service | 1.0 |
| VOCS | Volume Offset Control Service | 1.0 |

# 2. Implementation LE Audio Application

This chapter provides information about how to implement a LE Audio application with TMAP CT and/or UMR role, set the advertising data, and set up LE Audio services. The following sample codes can be found in Airoha LE Audio reference design source code.

## 2.1. Initialization

This section describes how to use LE Audio APIs for application development. The functionality of the LE Audio APIs is implemented in the library, but related APIs can be found in `bt_le_audio_sink.h`.

1) Call `bt_le_audio_sink_init()` to initialize the LE Audio functions with the parameter of TMAP role, callback function, and the maximum link number that LE AUDIO sink supported to. Please refer to `ble_tmap_def.h` for the TMAP role.

```
    bt_le_audio_sink_init((BLE_TMAP_ROLE_MASK_CT|BLE_TMAP_ROLE_MASK_UMR),
    app_le_audio_event_callback, max_link_num);
```

Note: Airoha sink service has the function of LE Audio sink device with CT and UMR role. Developers can skip this section and call `le_sink_srv_init()` to initialize sink service instead.

2) Implement callback function to handle the LE AUDIO sink events, such as status changed and caller information, etc.

```
    static void app_le_audio_event_callback(uint16_t event_id, void *p_msg)
    {
        switch (event_id) {
            case BT_LE_AUDIO_SINK_EVENT_CONNECTED: {
                /* LE Audio link connected */
                break;
            }
            case BT_LE_AUDIO_SINK_EVENT_DISCONNECTED: {
                /* LE Audio link disconnected */
                break;
            }
            case BT_LE_AUDIO_SINK_EVENT_CALL_SERVICE_READY: {
                /* call control service discovery complete */
                break;
            }
            ...
        }
    }
```

## 2.2. Advertising data setting

This section describes how to set the advertising data for a LE Audio sink device.

1) Call `multi_ble_adv_manager_add_ble_adv()` to register for adding LE Audio advertising data.

```
    multi_ble_adv_manager_add_ble_adv(MULTI_ADV_INSTANCE_DEFAULT,
    app_le_audio_get_adv_data);
```

2) Implement the callback function for the advertising manager to get the advertising data.

The advertising data include the following information:

- Random Set Identifier (RSI)
- Service data with ASCS UUID and available audio contexts
- Tx power
- Appearance
- Service data with BASS UUID

```
static uint32_t app_le_audio_get_adv_data(multi_ble_adv_info_t *adv_data)
{
    /* ADV DATA */
    if ((NULL != adv_data->adv_data) && (NULL != adv_data->adv_data->data)) {
        uint16_t sink_conent, source_conent;
        uint8_t rsi[6];
        uint8_t len = 0;

        adv_data->adv_data->data[len] = 2;
        adv_data->adv_data->data[len + 1] = BT_GAP_LE_AD_TYPE_FLAG;
        adv_data->adv_data->data[len + 2] =
BT_GAP_LE_AD_FLAG_BR_EDR_NOT_SUPPORTED |
BT_GAP_LE_AD_FLAG_GENERAL_DISCOVERABLE;
        len += 3;

        /* adv_data: RSI */
        adv_data->adv_data->data[len] = 7;
        adv_data->adv_data->data[len + 1] =  0x2E ;
        ble_csis_get_rsi(rsi);
        memcpy(&adv_data->adv_data->data[len + 2], rsi, sizeof(rsi));
        len += 8;

        /* adv_data: AD_TYPE_SERVICE_DATA */
        adv_data->adv_data->data[len] = 7;
        adv_data->adv_data->data[len + 1] = BT_GAP_LE_AD_TYPE_SERVICE_DATA;
        /* ASCS UUID: 2 bytes */
        adv_data->adv_data->data[len + 2] = (BT_GATT_UUID16_ASCS_SERVICE &
0x00FF);
        adv_data->adv_data->data[len + 3] = ((BT_GATT_UUID16_ASCS_SERVICE &
0xFF00) >> 8);        /*  content  availability : 4 bytes */
        ble_pacs_get_available_audio_contexts(&sink_conent, &source_conent);
        memcpy(&adv_data->adv_data->data[len + 4], &sink_conent, 2);
        memcpy(&adv_data->adv_data->data[len + 6], &source_conent, 2);
        len += 8;

        /* adv_data: TX_POWER */
        adv_data->adv_data->data[len] = 2;
        adv_data->adv_data->data[len + 1] = BT_GAP_LE_AD_TYPE_TX_POWER;
        adv_data->adv_data->data[len + 2] = 0x7F;
        len += 3;

        /* adv_data: AD_TYPE_APPEARANCE (TMAP) */
        adv_data->adv_data->data[len] = 3;
        adv_data->adv_data->data[len + 1] = BT_GAP_LE_AD_TYPE_APPEARANCE;
        /* value: 2 bytes */
        adv_data->adv_data->data[len + 2] =  0x41 ;
```

```
            adv_data->adv_data->data[len + 3] =  0x09 ;
            len += 4;

            /* adv_data: AD_TYPE_SERVICE_DATA (BASS)*/
            adv_data->adv_data->data[len] = 3;
            adv_data->adv_data->data[len + 1] = BT_GAP_LE_AD_TYPE_SERVICE_DATA;
            /* BASS UUID: 2 bytes */
            adv_data->adv_data->data[len + 2] = (BT_SIG_UUID16_BASS & 0x00FF);
            adv_data->adv_data->data[len + 3] = ((BT_SIG_UUID16_BASS & 0xFF00) >>
    8);
            len += 4;

            adv_data->adv_data->data_length = len;
        }

        /* ADV Parameter */
        if (NULL != adv_data->adv_param) {
            ...
        }

        /* SCAN RSP */
        if (NULL != adv_data->scan_rsp) {
            ...
        }
        return 0;
    }
```

Note: The maximum length of advertising data is 31 bytes. Developers may use scan response to fill in more information such as device name.

## 2.3. Service setting

All LE Audio service data are managed in LE Audio library. Airoha IoT SDK LE Audio provides developers to customize own LE Audio services. Here are the services that can be customized.

- AICS
- ASCS
- CSIS
- MICS
- PACS
- VCS
- VOCS

The way to establish GATT services is described in *Airoha_IoT_SDK_Bluetooth_Developers_Guide.docx* document. This section will introduce how to initialize the service data, customize multiple characteristics and multiple service instances, and distribute the request from remote device to LE Audio library.

### 2.3.1. Service data initialization

This section describes how to initialize the data in each LE Audio services.

### 2.3.1.1. Coordinated set setting

A coordinated set is defined as a group of devices such as a pair of earbuds. The coordinated set size is the number of the devices in the group. Airoha IoT SDK LE Audio provides CSIS APIs for developers to customize own coordinated set. In the following is an example to set a device to a coordinated set which has size is equal to 2. The sample code can be found in `ble_csis_service.c` in Airoha LE Audio reference design.

1) Define coordinated set size.

```
#define BLE_CSIS_DEFAULT_SIZE   2
```

2) Define a Set Identity Resolving Key (SIRK). The SIRK is associated with the Coordinated Set.

```
bt_key_t sirk = {0x12, 0x34, 0x56, 0x78, 0x9a, 0xbc, 0xde, 0xf0, 0x19, 0x28,
0x55, 0x33, 0x68, 0x33, 0x56, 0xde};
```

3) Implement the user defined function `ble_csis_init_parameter()`. This function is invoked by the LE Audio library in the state of CSIS initialization. API `ble_csis_set_coordinated_set_size()` and `ble_csis_set_sirk()` are used to set coordinated set size and SIRK. These APIs are defined in `ble_csis.h`.

```
void ble_csis_init_parameter(void)
{
    ble_csis_set_sirk(sirk);
    ble_csis_set_coordinated_set_size(BLE_CSIS_DEFAULT_SIZE);
}
```

4) Add Private Set Random Identifier (RSI) in the Advertising data by using API `ble_csis_get_rsi()` to get RSI. More advertising data setting information is describes in chapter 2.2

```
uint8_t rsi[6];
ble_csis_get_rsi(rsi);
```

### 2.3.1.2. Volume setting

This section describes how to set the default volume value in VCS. The sample code can also be found in `ble_vcs_service.c` in Airoha LE Audio reference design.

1) Implement the user defined function `ble_vcs_init_parameter()`. This function is invoked by the LE Audio library in the state of VCS initialization. API `ble_vcs_set_default_volume()` is used to set default volume value and is defined in `ble_vcs.h`.

```
void ble_vcs_init_parameter(void)
{
    ble_vcs_set_default_volume(BLE_VCS_DEFAULT_VOLUME);
}
```

### 2.3.1.3. PACS initialization

This section describes how to set Published Audio Capability (PAC), audio location, audio context in PACS. The PACS APIs are defined in `ble_pacs.h` and the sample codes are shown in `ble_pacs_service.c` in Airoha LE Audio reference design.

1) Implement the user defined function `ble_pacs_init_parameter()`. This function is invoked by the LE Audio library in the state of PACS initialization.

```
void ble_pacs_init_parameter(void)
{
    ...

    /* set sink audio location */
    ble_pacs_set_audio_location(AUDIO_DIRECTION_SINK, (channel ==
AUDIO_CHANNEL_NONE) ? AUDIO_LOCATION_NONE : (channel == AUDIO_CHANNEL_R) ?
AUDIO_LOCATION_FRONT_RIGHT : AUDIO_LOCATION_FRONT_LEFT);

    /* set source audio location */
    ble_pacs_set_audio_location(AUDIO_DIRECTION_SOURCE,
AUDIO_LOCATION_FRONT_LEFT);

    /* set available audio contexts */

ble_pacs_set_available_audio_contexts(AUDIO_CONTENT_TYPE_RINGTONE|AUDIO_CONTENT
_TYPE_CONVERSATIONAL|AUDIO_CONTENT_TYPE_MEDIA,
                                            AUDIO_CONTENT_TYPE_CONVERSATIONAL);

    /* set supported audio contexts */

ble_pacs_set_supported_audio_contexts(AUDIO_CONTENT_TYPE_RINGTONE|AUDIO_CONTENT
_TYPE_CONVERSATIONAL|AUDIO_CONTENT_TYPE_MEDIA,
                                            AUDIO_CONTENT_TYPE_CONVERSATIONAL);

    /* set PAC */
    ble_pacs_set_pac(AUDIO_DIRECTION_SINK, BLE_PACS_SINK_PAC_1,
&g_pacs_sink_pac_1);

    ble_pacs_set_pac(AUDIO_DIRECTION_SOURCE, BLE_PACS_SOURCE_PAC_1,
&g_pacs_source_pac_1);
}
```

2) In Airoha LE Audio reference design, four PAC records are provided as shown in Table 3. More details and the method for setting each PAC record are shown below.

***Table 3. PAC record list***

| Audio direction | codec | Sampling Frequency (kHz) | Frame Durations (ms) | Octets Per Codec Frame |
|---|---|---|---|---|
| SINK/ SOURCE | LC3 | 16 | 7.5 and 10 | 30 ~ 40 |
| SINK | LC3 | 24 | 7.5 and 10 | 45 ~ 60 |
| SINK/ SOURCE | LC3 | 32 | 7.5 and 10 | 60 ~ 80 |
| SINK | LC3 | 48 | 7.5 and 10 | 75 ~ 155 |

```
/*  codec_capabilities (16kHz) */
static uint8_t g_pacs_codec_capabilities_16k[] =
{
    CODEC_CAPABILITY_LEN_SUPPORTED_SAMPLING_FREQUENCY,
    CODEC_CAPABILITY_TYPE_SUPPORTED_SAMPLING_FREQUENCY,
    (uint8_t)SUPPORTED_SAMPLING_FREQ_16KHZ,
(uint8_t)(SUPPORTED_SAMPLING_FREQ_16KHZ >> 8),
    CODEC_CAPABILITY_LEN_SUPPORTED_FRAME_DURATIONS,
    CODEC_CAPABILITY_TYPE_SUPPORTED_FRAME_DURATIONS,
    SUPPORTED_FRAME_DURATIONS_7P5_MS|SUPPORTED_FRAME_DURATIONS_10_MS,

    CODEC_CAPABILITY_LEN_AUDIO_CHANNEL_COUNTS,
    CODEC_CAPABILITY_TYPE_AUDIO_CHANNEL_COUNTS,
    AUDIO_CHANNEL_COUNTS_1,

    CODEC_CAPABILITY_LEN_SUPPORTED_OCTETS_PER_CODEC_FRAME,
    CODEC_CAPABILITY_TYPE_SUPPORTED_OCTETS_PER_CODEC_FRAME,
    (uint8_t)SUPPORTED_OCTETS_PER_CODEC_FRAME_30_40,
(uint8_t)(SUPPORTED_OCTETS_PER_CODEC_FRAME_30_40 >> 8),
    (uint8_t)(SUPPORTED_OCTETS_PER_CODEC_FRAME_30_40 >> 16),
(uint8_t)(SUPPORTED_OCTETS_PER_CODEC_FRAME_30_40 >> 24),

    CODEC_CAPABILITY_LEN_MAX_SUPPORTED_LC3_FRAMES_PER_SDU,
    CODEC_CAPABILITY_TYPE_MAX_SUPPORTED_LC3_FRAMES_PER_SDU,
    MAX_SUPPORTED_LC3_FRAMES_PER_SDU_1,
};

/*  codec_capabilities (24kHz) */
static uint8_t g_pacs_codec_capabilities_24k[] =
{
    CODEC_CAPABILITY_LEN_SUPPORTED_SAMPLING_FREQUENCY,
    CODEC_CAPABILITY_TYPE_SUPPORTED_SAMPLING_FREQUENCY,
    (uint8_t)SUPPORTED_SAMPLING_FREQ_24KHZ,
(uint8_t)(SUPPORTED_SAMPLING_FREQ_24KHZ >> 8),

    CODEC_CAPABILITY_LEN_SUPPORTED_FRAME_DURATIONS,
    CODEC_CAPABILITY_TYPE_SUPPORTED_FRAME_DURATIONS,
    SUPPORTED_FRAME_DURATIONS_7P5_MS|SUPPORTED_FRAME_DURATIONS_10_MS,

    CODEC_CAPABILITY_LEN_AUDIO_CHANNEL_COUNTS,
    CODEC_CAPABILITY_TYPE_AUDIO_CHANNEL_COUNTS,
    AUDIO_CHANNEL_COUNTS_1,

    CODEC_CAPABILITY_LEN_SUPPORTED_OCTETS_PER_CODEC_FRAME,
    CODEC_CAPABILITY_TYPE_SUPPORTED_OCTETS_PER_CODEC_FRAME,
    (uint8_t)SUPPORTED_OCTETS_PER_CODEC_FRAME_45_60,
(uint8_t)(SUPPORTED_OCTETS_PER_CODEC_FRAME_45_60 >> 8),
    (uint8_t)(SUPPORTED_OCTETS_PER_CODEC_FRAME_45_60 >> 16),
(uint8_t)(SUPPORTED_OCTETS_PER_CODEC_FRAME_45_60 >> 24),

    CODEC_CAPABILITY_LEN_MAX_SUPPORTED_LC3_FRAMES_PER_SDU,
    CODEC_CAPABILITY_TYPE_MAX_SUPPORTED_LC3_FRAMES_PER_SDU,
    MAX_SUPPORTED_LC3_FRAMES_PER_SDU_1,
};

/*  codec_capabilities (32kHz) */
static uint8_t g_pacs_codec_capabilities_32k[] = {
    CODEC_CAPABILITY_LEN_SUPPORTED_SAMPLING_FREQUENCY,     /* length */
    CODEC_CAPABILITY_TYPE_SUPPORTED_SAMPLING_FREQUENCY,    /* type */
    (uint8_t)SUPPORTED_SAMPLING_FREQ_32KHZ,
(uint8_t)(SUPPORTED_SAMPLING_FREQ_32KHZ >> 8),  /* value */
```

```
        CODEC_CAPABILITY_LEN_SUPPORTED_FRAME_DURATIONS,
        CODEC_CAPABILITY_TYPE_SUPPORTED_FRAME_DURATIONS,
        SUPPORTED_FRAME_DURATIONS_7P5_MS|SUPPORTED_FRAME_DURATIONS_10_MS,

        CODEC_CAPABILITY_LEN_AUDIO_CHANNEL_COUNTS,
        CODEC_CAPABILITY_TYPE_AUDIO_CHANNEL_COUNTS,
        AUDIO_CHANNEL_COUNTS_1,

        CODEC_CAPABILITY_LEN_SUPPORTED_OCTETS_PER_CODEC_FRAME,
        CODEC_CAPABILITY_TYPE_SUPPORTED_OCTETS_PER_CODEC_FRAME,
        (uint8_t)SUPPORTED_OCTETS_PER_CODEC_FRAME_60_80,
    (uint8_t)(SUPPORTED_OCTETS_PER_CODEC_FRAME_60_80 >> 8),
        (uint8_t)(SUPPORTED_OCTETS_PER_CODEC_FRAME_60_80 >> 16),
    (uint8_t)(SUPPORTED_OCTETS_PER_CODEC_FRAME_60_80 >> 24),

        CODEC_CAPABILITY_LEN_MAX_SUPPORTED_LC3_FRAMES_PER_SDU,
        CODEC_CAPABILITY_TYPE_MAX_SUPPORTED_LC3_FRAMES_PER_SDU,
        MAX_SUPPORTED_LC3_FRAMES_PER_SDU_1,
    };

    /*  codec_capabilities (48kHz) */
    static uint8_t g_pacs_codec_capabilities_48k[] =
    {
        CODEC_CAPABILITY_LEN_SUPPORTED_SAMPLING_FREQUENCY,
        CODEC_CAPABILITY_TYPE_SUPPORTED_SAMPLING_FREQUENCY,
        (uint8_t)SUPPORTED_SAMPLING_FREQ_48KHZ,
    (uint8_t)(SUPPORTED_SAMPLING_FREQ_48KHZ >> 8),

        CODEC_CAPABILITY_LEN_SUPPORTED_FRAME_DURATIONS,
        CODEC_CAPABILITY_TYPE_SUPPORTED_FRAME_DURATIONS,
        SUPPORTED_FRAME_DURATIONS_7P5_MS|SUPPORTED_FRAME_DURATIONS_10_MS,

        CODEC_CAPABILITY_LEN_AUDIO_CHANNEL_COUNTS,
        CODEC_CAPABILITY_TYPE_AUDIO_CHANNEL_COUNTS,
        AUDIO_CHANNEL_COUNTS_1,

        CODEC_CAPABILITY_LEN_SUPPORTED_OCTETS_PER_CODEC_FRAME,
        CODEC_CAPABILITY_TYPE_SUPPORTED_OCTETS_PER_CODEC_FRAME,
        (uint8_t)SUPPORTED_OCTETS_PER_CODEC_FRAME_75_155,
    (uint8_t)(SUPPORTED_OCTETS_PER_CODEC_FRAME_75_155 >> 8),
        (uint8_t)(SUPPORTED_OCTETS_PER_CODEC_FRAME_75_155 >> 16),
    (uint8_t)(SUPPORTED_OCTETS_PER_CODEC_FRAME_75_155 >> 24),

        CODEC_CAPABILITY_LEN_MAX_SUPPORTED_LC3_FRAMES_PER_SDU,
        CODEC_CAPABILITY_TYPE_MAX_SUPPORTED_LC3_FRAMES_PER_SDU,
        MAX_SUPPORTED_LC3_FRAMES_PER_SDU_1,
    };

    /*  metadata (48kHz) */
    static uint8_t g_pacs_metadata[] = {0x03, 0x01, 0x01, 0x00};

    /*  PAC list (48kHz) */
    static ble_pacs_pac_record_t g_pacs_pac_1[] = {
        {
            CODEC_ID_LC3,
            PACS_CODEC_CAPABLITIES_LEN,
            &g_pacs_codec_capabilities_16k[0],
            PACS_METADATA_LEN,
            &g_pacs_metadata[0],
        },
        {
            CODEC_ID_LC3,
            PACS_CODEC_CAPABLITIES_LEN,
            &g_pacs_codec_capabilities_32k[0],
            PACS_METADATA_LEN,
            &g_pacs_metadata[0],
        },
        {
            CODEC_ID_LC3,
            PACS_CODEC_CAPABLITIES_LEN,
            &g_pacs_codec_capabilities_24k[0],
            PACS_METADATA_LEN,
            &g_pacs_metadata[0],
        },
        {
            CODEC_ID_LC3,
            PACS_CODEC_CAPABLITIES_LEN,
            &g_pacs_codec_capabilities_48k[0],
            PACS_METADATA_LEN,
            &g_pacs_metadata[0],
        },
    };

    /* SINK PAC */
    ble_pacs_pac_t g_pacs_sink_pac_1 = {
        PACS_SINK_PAC_1_RECORD_NUM,
        &g_pacs_pac_1[0],
    };

    /* SOURCE PAC */
    ble_pacs_pac_t g_pacs_source_pac_1 = {
        PACS_SOURCE_PAC_1_RECORD_NUM,
        &g_pacs_pac_1[0],
    };
```

### 2.3.1.4. Audio input description setting

This section describes how to set the audio input description in AICS if AICS is declared in this device. The sample code can be found in `ble_aics_service.c` in Airoha LE Audio reference design.

1) Implement the user defined function `ble_aics_init_parameter()`. This function is invoked by the LE Audio library in the state of AICS initialization. `ble_aics_set_audio_input_description_by_channel()` is used to set audio input description and is defined in `ble_aics.h`.

```
    void ble_aics_init_parameter(void)
    {
        /* BLE_AICS_1 */
        ble_aics_set_audio_input_description_by_channel(BLE_AICS_1,
    input_description, INPUT_DESCRIPTION_LENGTH);
    }
```

#### 2.3.1.5. Audio output description setting

This section describes how to set the audio output description in VOCS, if VOCS is declared in this device. The sample code can be found in `ble_vocs_service.c` in Airoha LE Audio reference design.

1) Implement the user defined function `ble_vocs_init_parameter()`. This function is invoked by the LE Audio library in the state of VOCS initialization. `ble_vocs_set_audio_output_description_by_channel()` is used to set audio output description and is defined in `ble_vocs.h`.

```
void ble_vocs_init_parameter(void)
{
    /* BLE_VOCS_CHANNEL_1 */
    ble_vocs_set_audio_output_description_by_channel(BLE_VOCS_CHANNEL_1,
(uint8_t *)vocs_audio_output_desc_left, VOCS_AUDIO_OUTPUT_DESC_SIZE_LEFT);

    /* BLE_VOCS_CHANNEL_2 */
    ble_vocs_set_audio_output_description_by_channel(BLE_VOCS_CHANNEL_2,
(uint8_t *)vocs_audio_output_desc_right, VOCS_AUDIO_OUTPUT_DESC_SIZE_RIGHT);
}
```

### 2.3.2. Multiple characteristics

In some LE Audio services, multiple characteristics in a single service instance are permitted, such as ASCS and PACS. Airoha IoT SDK LE Audio provides developer to customize these LE Audio services. This section describes how to implement a LE Audio GATT service with multiple characteristics. The following is an example to implement an ASCS with three Audio Stream Endpoint (ASE) characteristics. These sample codes can be found in Airoha reference design in `ble_ascs_service.c`.

1) Define the ASCS attribute handles including the ASCS service start handle, service end handle, and characteristic value handles. These attribute handles are also used in establishing GATT service. And 0x1105, 0x1108, and 0x110B are the attribute value handle of the ASEs.

```
#define ASCS_START_HANDLE       0x1103 /**< ASCS service start handle.*/
#define ASCS_VALUE_HANDLE_ASE_1 0x1105 /**< ASE_1 characteristic handle.*/
#define ASCS_VALUE_HANDLE_ASE_2 0x1108 /**< ASE_2 characteristic handle.*/
#define ASCS_VALUE_HANDLE_ASE_3 0x110B /**< ASE_3 characteristic handle.*/
#define ASCS_VALUE_HANDLE_ASE_CONTROL_POINT 0x110E /**< ASE Control Point
characteristic handle.*/
#define ASCS_END_HANDLE         0x110F /**< ASCS service end handle.*/
```

2) Define a characteristic ASE index table.

```
typedef enum {
    BLE_ASCS_ASE_1,          /**< ASE_1. */
    BLE_ASCS_ASE_2,          /**< ASE_2. */
    BLE_ASCS_ASE_3,          /**< ASE_3. */
    BLE_ASCS_ASE_MAX_NUM,    /**< Total number of ASE. */
} ble_ascs_charc_ase_t;
```

3) Define a UUID type and attribute handle mapping table. ASCS UUID type is referred in `ble_ascs_def.h`. The first record shall be ASCS service UUID type with service start handle and the last record shall be `BLE_ASCS_UUID_TYPE_INVALID` with service end handle. And for the three ASEs,

BLE_ASCS_UUID_TYPE_ASE is used as the UUID type and the attribute handle are their attribute value handles.

```
static ble_ascs_attribute_handle_t g_ascs_att_handle_tbl[] = {
    {BLE_ASCS_UUID_TYPE_ASCS_SERVICE,        ASCS_START_HANDLE},
    /* BLE_ASCS_ASE_1 */
    {BLE_ASCS_UUID_TYPE_ASE,                 ASCS_VALUE_HANDLE_ASE_1},
    /* BLE_ASCS_ASE_2 */
    {BLE_ASCS_UUID_TYPE_ASE,                 ASCS_VALUE_HANDLE_ASE_2},
    /* BLE_ASCS_ASE_3 */
    {BLE_ASCS_UUID_TYPE_ASE,                 ASCS_VALUE_HANDLE_ASE_3},
    /* ASE Control Point */
    {BLE_ASCS_UUID_TYPE_ASE_CONTROL_POINT,
ASCS_VALUE_HANDLE_ASE_CONTROL_POINT},
    {BLE_ASCS_UUID_TYPE_INVALID,             ASCS_END_HANDLE},
};
```

4) Implement the user defined function `ble_ascs_get_attribute_handle_tbl()`. This function is used in LE Audio library get the UUID type and attribute handle mapping table that defined above.

```
ble_ascs_attribute_handle_t *ble_ascs_get_attribute_handle_tbl(void)
{
    return g_ascs_att_handle_tbl;
}
```

5) Implement the user defined function `ble_ascs_get_ase_number()`. This function is used in LE Audio library to get the number of ASEs. The number of ASEs is defined in the ASE index table.

```
uint8_t ble_ascs_get_ase_number(void)
{
    return BLE_ASCS_ASE_MAX_NUM;
}
```

6) Call `ble_ascs_gatt_request_handler()` to distributes the GATT requests from remote device to the LE Audio library. The Parameter `charc_idx` is used to indicate that which ASE is the target of the GATT request.

```
uint32_t ble_ascs_gatt_request_handler(ble_ascs_gatt_request_t type,
bt_handle_t handle, uint8_t charc_idx, void *data, uint16_t size, uint16_t
offset)
```

For example, remote device requests to read BLE_ASCS_ASE_1 content.

```
static uint32_t ble_ascs_ase_value_callback_1(const uint8_t rw, uint16_t
handle, void *data, uint16_t size, uint16_t offset)
{
    if (handle != BT_HANDLE_INVALID && rw == BT_GATTS_CALLBACK_READ) {
        return ble_ascs_handle_gatt_callback(BLE_ASCS_READ_ASE, handle,
BLE_ASCS_ASE_1, data, size, offset);
    }

    return 0;
}
```

### 2.3.3. Multiple service instances

In some LE Audio services, multiple service instances are permitted such as VOCS and AICS. Airoha IoT SDK LE Audio provides developer to customize these LE Audio services. This section describes how to implement a LE Audio GATT service with multiple service instances. In the following is an example to implement two VOCS service instances. These sample codes can be found in Airoha reference design in `ble_vocs_service.c`.

1) Define a service instance index table. There are two instances in this example `BLE_VOCS_CHANNEL_1` and `BLE_VOCS_CHANNEL_2`.

```
typedef uint8_t ble_vocs_channel_t;
#define BLE_VOCS_CHANNEL_1          0x00 /**< VOCS channel 1. */
#define BLE_VOCS_CHANNEL_2          0x01 /**< VOCS channel 2. */
#define BLE_VOCS_MAX_CHANNEL_NUM    0x02 /**< The maximum number of VOCS
channel. */
```

2) Define a UUID type and attribute handle mapping table for each service instances. VOCS UUID type is referred in `ble_vocs_def.h`. The first record shall be VOCS service UUID type with service start handle and the last record shall be `BLE_VOCS_UUID_TYPE_INVALID` with service end handle. The others are the characteristic UUID types with the corresponding attribute value handles.

```
/* BLE_VOCS_CHANNEL_1 */
static ble_vocs_attribute_handle_t g_vocs_att_handle_tbl_1[] = {
    {BLE_VOCS_UUID_TYPE_VOLUME_OFFSET_CONTROL_SERVICE,
BLE_VOCS_START_HANDLE_CHANNEL_1},
    {BLE_VOCS_UUID_TYPE_OFFSET_STATE,
BLE_VOCS_VALUE_HANDLE_OFFSET_STATE_CHANNEL_1},
    {BLE_VOCS_UUID_TYPE_AUDIO_LOCATION,
BLE_VOCS_VALUE_HANDLE_AUDIO_LOCATION_CHANNEL_1},
    {BLE_VOCS_UUID_TYPE_VOLUME_OFFSET_CONTROL_POINT,
BLE_VOCS_VALUE_HANDLE_CONTROL_POINT_CHANNEL_1},
    {BLE_VOCS_UUID_TYPE_AUDIO_OUTPUT_DESCRIPTION,
BLE_VOCS_VALUE_HANDLE_AUDIO_OUTPUT_DESCRIPTION_CHANNEL_1},
    {BLE_VOCS_UUID_TYPE_INVALID,
BLE_VOCS_END_HANDLE_CHANNEL_1},
};

/* BLE_VOCS_CHANNEL_2 */
static ble_vocs_attribute_handle_t g_vocs_att_handle_tbl_2[] = {
    {BLE_VOCS_UUID_TYPE_VOLUME_OFFSET_CONTROL_SERVICE,
BLE_VOCS_START_HANDLE_CHANNEL_2},
    {BLE_VOCS_UUID_TYPE_OFFSET_STATE,
BLE_VOCS_VALUE_HANDLE_OFFSET_STATE_CHANNEL_2},
    {BLE_VOCS_UUID_TYPE_AUDIO_LOCATION,
BLE_VOCS_VALUE_HANDLE_AUDIO_LOCATION_CHANNEL_2},
    {BLE_VOCS_UUID_TYPE_VOLUME_OFFSET_CONTROL_POINT,
BLE_VOCS_VALUE_HANDLE_CONTROL_POINT_CHANNEL_2},
    {BLE_VOCS_UUID_TYPE_AUDIO_OUTPUT_DESCRIPTION,
BLE_VOCS_VALUE_HANDLE_AUDIO_OUTPUT_DESCRIPTION_CHANNEL_2},
    {BLE_VOCS_UUID_TYPE_INVALID,
BLE_VOCS_END_HANDLE_CHANNEL_2},
};
```

3) Implement the user defined function `ble_vocs_get_attribute_handle_tbl()`. This function is used in LE Audio library get the UUID type and attribute handle mapping table that defined above.

```
ble_vocs_attribute_handle_t *ble_vocs_get_attribute_handle_tbl(uint8_t channel)
```

```
{
    switch (channel) {
        case BLE_VOCS_CHANNEL_1: {
            return g_vocs_att_handle_tbl_1;
        }
        case BLE_VOCS_CHANNEL_2: {
            return g_vocs_att_handle_tbl_2;
        }
        default:
            break;
    }
    return NULL;
}
```

4) Implement the user defined function `ble_vocs_get_channel_number()`. This function is used in LE Audio library get the number of VOCS service instances. The number of VOCS service instances is defined in service instance index table.

```
uint8_t ble_vocs_get_channel_number(void)
{
    return BLE_VOCS_MAX_CHANNEL_NUM;
}
```

5) Call `ble_vocs_gatt_request_handler()` to distributes the GATT requests from remote device to the LE Audio library. The Parameter `channel` is used to indicate that which VOCS service instance is the target of the GATT request.

```
uint16_t ble_vocs_gatt_request_handler(ble_vocs_gatt_request_t type,
bt_handle_t handle, uint8_t channel, void *data, uint16_t size)
```

For example, remote device requests to read the volume offset state in BLE_VOCS_CHANNEL_1.

```
static uint32_t ble_vocs_offset_state_callback_channel_1(const uint8_t rw,
uint16_t handle, void *data, uint16_t size, uint16_t offset)
{
    if (handle != BT_HANDLE_INVALID && rw == BT_GATTS_CALLBACK_READ) {
        return ble_vocs_gatt_request_handler(BLE_VOCS_READ_VOLUME_OFFSET_STATE,
handle, BLE_VOCS_CHANNEL_1, data, size);
    }

    return 0;
}
```

## 2.4. Synchronization to BMS

This section describes how to synchronize to the BMS as the BMR. The BMR is able to listen to the encrypted or non-encrypted stream that BMS is broadcasting by synchronizing to a specific BMS.

Below is the sample code for synchronizing to the stream with the left earbud:

```
bt_sink_srv_cap_stream_bmr_scan_param_t scan_param = {0};

scan_param.audio_channel_allocation = AUDIO_LOCATION_FRONT_LEFT;
scan_param.bms_address = NULL; /* does not specify a BMS*/
scan_param.duration = DEFAULT_SCAN_TIMEOUT;
```

```
    status = bt_sink_srv_cap_stream_scan_broadcast_source(&scan_param);
```

Below is the audio channel allocation parameter for synchronizing with right earbud:

```
    scan_param.audio_channel_allocation =  AUDIO_LOCATION_FRONT_RIGHT;
```

If the BMR intends to synchronizing to a specific BMS, the BMS address can be assigned in the parameter.

```
    bt_bd_addr_t addr = {0x11, 0x22, 0x33, 0x44, 0x55, 0x66};

    scan_param.bms_address = (bt_bd_addr_t*)&addr[0];
```

Below is the API for the BMR intends to stop to synchronize to the BMS:

```
    bt_bd_addr_t addr = {0x11, 0x22, 0x33, 0x44, 0x55, 0x66};

    scan_param.bms_address = (bt_bd_addr_t*)&addr[0];
```

If the BMS advertises more than one BIS, the BMR can also specify a BIS index to synchronize to:

```
    uint8_t bis_indices[1] = {1};

    bt_sink_srv_cap_stream_set_big_sync_info(big_handle, 1, bis_indices);
```

# 3. Profile and service discovery

## 3.1. Call Control

For a CT device, Call Control Profile (CCP) acts as a client role to interact with a remote device that implements the Telephone Bearer Service (TBS) and/or the Generic Telephone Bearer Service (GTBS). This section describes how to use the CCP APIs to set the TBS and/or GTBS service table to CCP. The related APIs can be found in `ble_ccp_discovery.h`.

1) Call the `ble_ccp_set_service_attribute()` after discovering each TBS or GTBS service instance to set the result to CCP with the characteristic and characteristic descriptor information in the parameter.

```
    bt_status_t ble_ccp_set_service_attribute(bt_handle_t handle,
    ble_ccp_set_service_attribute_parameter_t *params)
```

For example, a GTBS service instance is found in remote device.

```
    ble_ccp_set_service_attribute_parameter_t param;

    /* set the charc information */
    param.charc = p_charc_list;

    /* set start attribute handle of the service */
    param.start_handle = g_ccp_service.start_handle;

    /* set end attribute handle of the service */
    param.end_handle = g_ccp_service.end_handle;

    /* set the charc number of the service */
    param.charc_num = charc_num;

    /* indicate the service is GTBS or not */
    param.is_gtbs = true;

    /* indicate is last service instance or not, set true if no more service
    instances will be found. */
    param.is_complete = true;

    /* callback is invoked when LE Audio Library finish setting attribute */
    param.callback = app_ccp_set_attribute_callback;

    ble_ccp_set_service_attribute(event->conn_handle, &param);
```

Note: There can be zero or multiple instances of TBS and a single instance of GTBS in a remote device, therefore `ble_ccp_set_service_attribute()` may be called more than one time. API `ble_ccp_set_service_attribute()` must be used to inform CCP event if TBS or GTBS is not found.

In the following is an example with TBS is not found in the remote device.

```
    ble_ccp_set_service_attribute_parameter_t param;

    /* set the charc information */
    param.charc = NULL;
```

```
/* set start attribute handle of the service */
param.start_handle = 0;

/* set end attribute handle of the service */
param.end_handle = 0;

/* set the charc number of the service */
param.charc_num = 0;

/* indicate the service is GTBS or not */
param.is_gtbs = false;

/* indicate is last service instance or not, set true if no more service
instances will be found. */
param.is_complete = true;

/* callback is invoked when LE Audio Library finish setting attribute */
param.callback = app_ccp_set_attribute_callback;

ble_ccp_set_service_attribute(event->conn_handle, &param);
```

2) Implement the callback function.

```
static void app_ccp_set_attribute_callback(void)
{
    /* This function will be invoked when library finish setting service
attribute, and application layer is able to discover next service instance if
any. */
}
```

3) After all TBS and GTBS discovery complete and these services exist in remote device, call control functions are able to be used. Call control APIs are defined in `bt_le_audio_sink.h`. In the following is an example to use call control API to accept call.

```
bt_le_audio_sink_call_action_param_t le_param = {
    .service_idx = BLE_CCP_GTBS_INDEX,
    .length = 2,
};
le_param.call_control_point = (ble_ccp_call_control_point_t*)buf;
buf[0] = BLE_TBS_CALL_CONTROL_OPCODE_TYPE_ACCEPT;
buf[1] = 1; /* call index */
result = bt_le_audio_sink_send_action(handle,
BT_LE_AUDIO_SINK_ACTION_CALL_ACCEPT, &le_param);
```

Note: `service_idx` in bt_le_audio_sink_call_action_param_t is used to indicate to control which TBS instance, and use BLE_CCP_GTBS_INDEX instead to control GTBS.

## 3.2. Media Control

For a UMR device, Media Control Profile (MCP) is act as a client role to interact with a remote device that implements the Media Control Service (MCS) and/or the Generic Media Control Service (GMCS). This section describes how to use the MCP APIs to set the MCS and/or GMCS service table to MCP. The related APIs can be found in `ble_mcp_discovery.h`.

1) Call the `ble_mcp_set_service_attribute()` after discovering each MCS or GMCS service instance to set the result to MCP with the characteristic and characteristic descriptor information in the parameter.

```
bt_status_t ble_mcp_set_service_attribute(bt_handle_t handle,
ble_mcp_set_service_attribute_parameter_t *params)
```

For example, a GMCS service instance is found in remote device.

```
ble_mcp_set_service_attribute_parameter_t param;

/* set the charc information */
param.charc = p_charc_list;

/* set start attribute handle of the service */
param.start_handle = g_ccp_service.start_handle;

/* set end attribute handle of the service */
param.end_handle = g_ccp_service.end_handle;

/* set the charc number of the service */
param.charc_num = charc_num;

/* indicate the service is GMCS or not */
param.is_gmcs = true;

/* indicate is last service instance or not, set true if no more service
instances will be found. */
param.is_complete = true;

/* callback is invoked when LE Audio Library finish setting attribute */
param.callback = app_mcp_set_attribute_callback;

ble_mcp_set_service_attribute(event->conn_handle, &param);
```

Note: There can be zero or multiple instances of MCS and a single instance of GMCS in a remote device, therefore `ble_mcp_set_service_attribute()` may be called more than one time. API `ble_mcp_set_service_attribute()` must be used to inform MCP event if MCS or GMCS is not found.

In the following is an example with MCS is not found.

```
ble_mcp_set_service_attribute_parameter_t param;

/* set the charc information */
param.charc = NULL;

/* set start attribute handle of the service */
param.start_handle = 0;

/* set end attribute handle of the service */
param.end_handle = 0;

/* set the charc number of the service */
param.charc_num = 0;

/* indicate the service is GMCS or not */
```

```
    param.is_gmcs = false;

    /* indicate is last service instance or not, set true if no more service
    instances will be found. */
    param.is_complete = true;

    /* callback is invoked when LE Audio Library finish setting attribute */
    param.callback = app_mcp_set_attribute_callback;

    ble_mcp_set_service_attribute(event->conn_handle, &param);
```

2) Implement the callback function.

```
    static void app_mcp_set_attribute_callback(void)
    {
        /* This function will be invoked when library finish setting service
    attribute, and application layer is able to discover next service instance if
    any. */
    }
```

3) After all MCS and GMCS discovery complete and these services exist in remote device, media control functions are able to be used. Media control APIs are defined in `bt_le_audio_sink.h`. In the following is an example to use media control API to play music.

```
    bt_le_audio_sink_action_param_t le_param = {
        .service_idx = BLE_MCP_GMCS_INDEX,
    };
    bt_le_audio_sink_send_action(handle, BT_LE_AUDIO_SINK_ACTION_MEDIA_PLAY,
    &le_param);
```

Note: `service_idx` in bt_le_audio_sink_call_action_param_t is used to indicate to control which MCS instance, and use BLE_MCP_GMCS_INDEX instead to control GMCS.

## 3.3. Volume Control

A CT and/or UMR device is also a Volume Renderer device which has Volume Control Service (VCS). The volume control APIs are defined in `bt_le_audio_sink.h`. This section describes how to set volume mute.

```
    bt_le_audio_sink_send_action(handle, BT_LE_AUDIO_SINK_ACTION_VOLUME_MUTE,
    NULL);
```