Airoha IoT SDK Bluetooth Developer's Guide

Version:	1.9
Release date:	25 Sep 2019

Document Revision History

Revision	Date	Description
1.0	2 September 2016	Initial release
1.1	4 November 2016	Added description for multi-advertising support, see section 4.1.8, "Multiple advertising".
1.2	13 January 2017	Added description for AWS Added Bluetooth panic mechanism, see section 5.4, “Bluetooth panic mechanism”
1.3	5 May 2017	Refine the issues in table of contents (TOC).
1.4	30 June 2017	• Updated description for AWS. • Updated AVRCP version from 1.3 to 1.6, added section 3.5.2.4, “Absolute volume”.
1.5	15 September 2017	• Added description to get play status, added section 3.5.2.7, “Get play status”. • Added description for get capabilities, see section 3.5.2.8, “Get capabilities”. • Added description for A2DP source and sink, see section 3.4, “A2DP”
1.6	27 October 2018	• Added AVRCP Browsing channel, see section 3.5 • Removed AWS • Added AWS_MCE • Added HSP, added section 3.8 and section 1.1.1 • Upgrade A2DP to V1.3, see section 3.4 • Added description for SDP client support, see section 3.2.3, “Use client API” • Added DI, see section 3.9, “DI” • Add description for GAP cancel connection, see section Cancel connection 3.1.1.5, “Cancel connection”
1.7	25 March 2019	• Added unsupported chips note for BR/EDR features, see section 3, “The Bluetooth BR/EDR Protocol and Profiles”. • Replaced “AWS_MCE” with “MCSync”, see section 3.8 and section 1.1.1.
1.8	20 May 2019	• Removed the MCSync, see section 1.1.1
1.9	25 Sep 2019	• Fixed some link errors

Table of Contents

Lists of Tables and Figures

1. Overview

Airoha IoT development platform provides Bluetooth and Bluetooth Low Energy (LE) connectivity support for IoT and Wearable's applications. Bluetooth standard offers basic rate (BR) or enhanced data rate (EDR) and Bluetooth Low Energy (LE) support. Devices that can support BR/EDR and Bluetooth LE are referred to as dual-mode devices. Typically, in a Bluetooth system, a mobile phone or laptop computer acts as a dual-mode device. Devices that only support Bluetooth LE are referred to as single-mode devices where the low power consumption is the primary concern for application development, such as those that run on coin cell batteries.

This document guides you through:

- Support for Bluetooth with the library description and supported reference examples.
- Detailed description of the BR/EDR profiles.
- Bluetooth LE profiles.
- Custom application development and debugging logs.

1.1. Bluetooth protocol stack

The Bluetooth protocol stack consists of two sections, the controller and the host, derived from the original Bluetooth core specification where the two sections were often implemented separately.

1.1.1. BR/EDR protocol stack

The BR/EDR protocol stack architecture is illustrated in Figure 1.

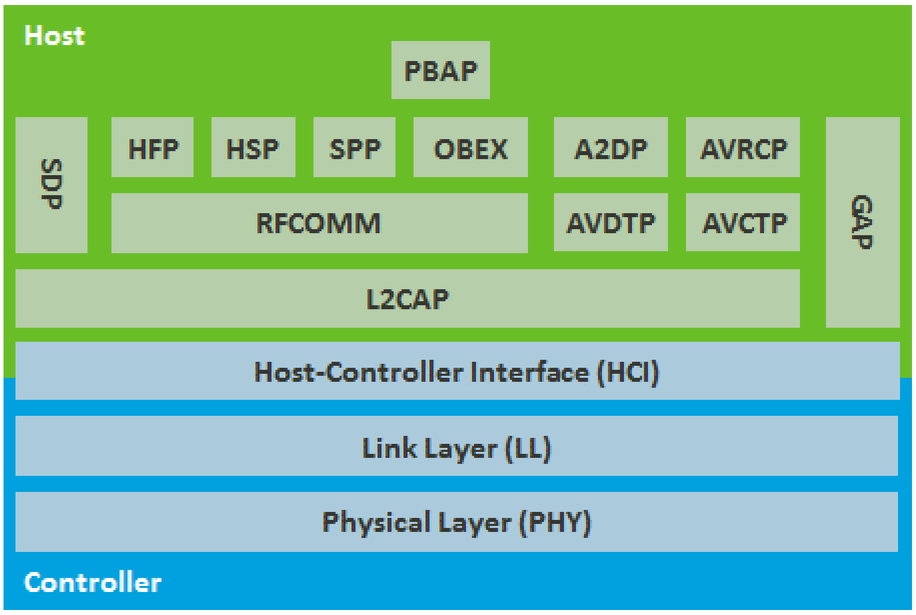

Figure 1. BR/EDR protocol stack

The controller section includes the Physical Layer (PHY), Link Layer (LL), and Host-Controller Interface (HCI).

1) The physical layer (PHY) is the lowest layer of the Bluetooth protocol stack to manage physical channels and links.

2) The link layer (LL) is used to control the radio link between two devices, handling matters such as link establishment, querying device abilities and providing power control.
3) The host control interface (HCI) provides a means of communication between the host and controller using a standardized interface.

The host includes various communication protocols, as described below.

1) The logical link control and adaption protocol (L2CAP) provides data encapsulation services to the upper layers that enable logical end-to-end data communication.
2) The radio frequency communication (RFCOMM) is a set of transport protocols on top of the L2CAP protocol, providing emulated RS-232 serial ports.
3) The generic access profile (GAP) defines the generic procedures related to discovery of Bluetooth devices and link management aspects of connecting to Bluetooth devices.
4) The service discovery protocol (SDP) is used to allow devices to discover the supported services, and parameters to use to connect to them.
5) The hands-free profile (HFP) allows hands-free kits to control the mobile phone of calling functions and provide the voice connection between devices.
6) The object exchange (OBEX) is a communications protocol that facilitates the exchange of binary objects between devices.
7) The phonebook access profile (PBAP) defines the procedures and protocols to exchange phonebook objects between devices.
8) The Advanced Audio Distribution Profile (A2DP) defines how multimedia audio is streamed from one device to another via Bluetooth connection.
9) The audio/video distribution transport protocol (AVDTP) is used by the advanced audio distribution profile to stream music to stereo headsets over the L2CAP channel intended for video distribution profile in the Bluetooth transmission.
10) The audio/video control transport protocol (AVCTP) is used to transport the command/response messages exchanged for the control of distant A/V devices over point-to-point connections.
11) The audio/video remote control profile (AVRCP) defines the features and procedures required in order to ensure interoperability between Bluetooth devices with audio/video control functions in the audio/video distribution scenarios.
12) The serial port profile (SPP) emulates a serial cable to provide a simple substitute for RS-232.
13) The Headset Profile (HSP) defines the protocols and procedures that are used by devices which must have a full-duplex audio connection combined with minimal device control command.

1.1.2. Bluetooth LE protocol stack

The Bluetooth LE protocol stack architecture is shown in Figure 2.

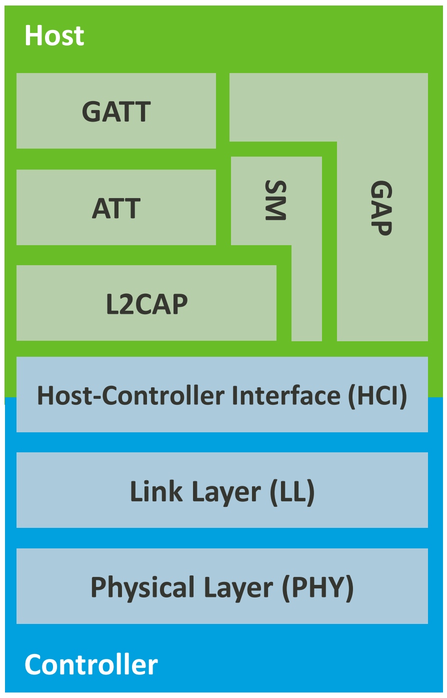

Figure 2. Bluetooth Low Energy protocol stack

2. Support for Bluetooth

The platform includes Airoha IoT SDK v4 that can be extended to develop Bluetooth applications. The controller, host and application are implemented as a true single chip solution to enable cost effective and application development with low power consumption.

The SDK is designed to support custom Bluetooth application development with protocol-layer access to the following:

- GAP
- SDP
- HFP
- A2DP
- AVRCP
- PBAP
- SPP
- GAP LE
- SM
- GATT

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

The SDK includes the following content along with this document:

- Binary libraries
- C header files
- Example applications
- API reference guides

2.1. Airoha IoT SDK library

The SDK provides a library file interface to 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 Bluetooth

Module	File Name	Location	Function
Bluetooth	`libbt.a`	`/middleware/bluetooth/lib/`	BR/EDR and Bluetooth LE stack library
	`libbtdriver_[chip].a`		Bluetooth driver library
	`libbt_hfp.a`		HFP library
	`libbt_hsp.a`		HSP library
	`libbt_a2dp.a`		A2DP library

Module	File Name	Location	Function
	`libbt_avrcp.a`		AVRCP library, including the PASS THROUGH command.
	`libbt_avrcp_enhance.a`		AVRCP library, including all VENDOR DEPENDENT commands.
	`libbt_pbapc.a`		PBAP library
	`libbt_spp.a`		SPP library
	`bt_platform.h`	`/middleware/bluetooth/inc/`	Interface for Bluetooth tasks
	`bt_type.h`		Common data types
	`bt_system.h`		Interface for the system, such as power on or off, memory initiation and callback APIs for event handling.
	`bt_uuid.h`		Interface for the UUID
	`bt_spp.h`		Interface for the SPP
	`bt_a2dp.h`		Interface for the A2DP
	`bt_codec.h`		Interface for the codec
	`bt_hfp.h`		Interface for the HFP
	`bt_hsp.h`		Interface for the HSP
	`bt_avrcp.h`		Interface for the AVRCP
	`bt_pbapc.h`		Interface for the PBAP client
	`bt_gap.h`		Interface for the GAP
	`bt_sdp.h`		Interface for the SDP
	`bt_gap_le.h`		Interface for the GAP for Bluetooth LE support
	`bt_hci_le.h`		HCI structures and events for Bluetooth LE support
	`bt_gatt.h`		GATT UUID
	`bt_gattc.h`		Interface for the GATT client
	`bt_gatts.h`		Interface for the GATT server
	`bt_att.h`		Interface for the Attribute protocol
	`bt_os_layer_api.h`		Wrapper APIs for RTOS, memory, advanced encryption standard (AES) and rand
	`bt_debug.h`		Encapsulated debugging interface
	`bt_hci_log.h`		Encapsulated interface for the HCI logging
	`bt_os_layer_api.c`	1 `/middleware/bluetooth/src/`	Encapsulated interface for system, memory or AES. Developers can replace the implementation when porting to other platforms.
	`bt_debug.c`		Encapsulated debugging interface. Developers can replace the

Module	File Name	Location	Function
			implementation when porting to other platforms.
	`bt_hci_log.c`		Encapsulated interface for the HCI logging. Developers can replace the implementation when porting to other platforms.
	`bt_task.c`		The default Bluetooth task entry function.
	`bt_log.c`		The definition for the debugging string used in BT Stack library.

3. The Bluetooth BR/EDR Protocol and Profiles

The Bluetooth support is provided as a collection of binary library files. This section provides details on the GAP, SDP, HFP, A2DP, AVRCP, SPP and PBAP profiles for which application has direct access to the APIs.

Note: BR/EDR Protocol and profiles don't support on MT76x7 and AW7698.

3.1. GAP

GAP services include device discovery, connection modes, security, authentication, association models and service discovery. Implement a user-defined API `bt_gap_get_local_configuration()` in your application when turning the Bluetooth on. The GAP profile then calls the function `bt_gap_get_local_configuration()` to get the local configuration from the application.

The purpose of the inquiry is to discover other Bluetooth-enabled devices in general or limited discoverable modes. The purpose of the bonding procedure is to create a connection between two Bluetooth-enabled devices based on a common link key. The link key will be used for future authentication.

The GAP profile provides a user interface, sends related commands to the controller and receives events from the controller, as shown in Figure 3.

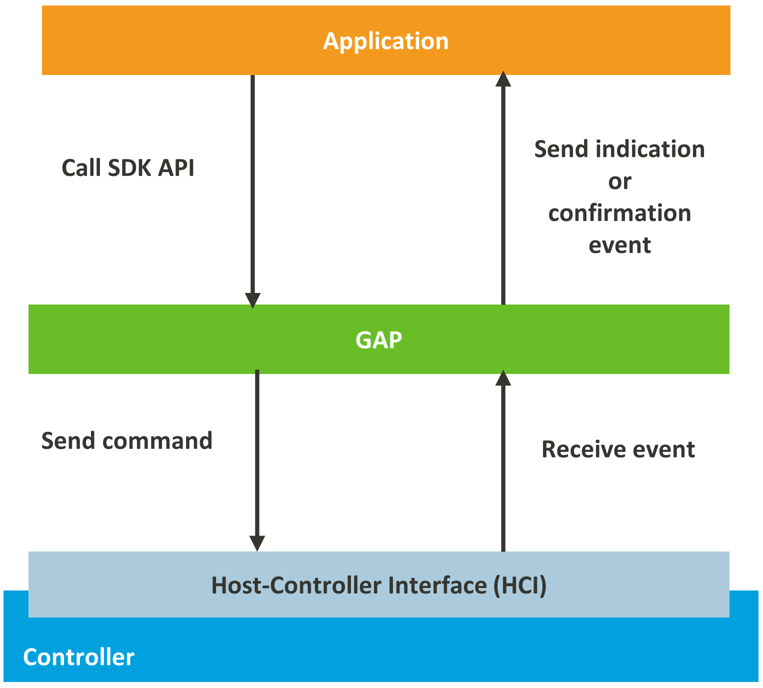
Application
Call SDK API
Send indication or confirmation event
GAP
Send command
Receive event
Host-Controller Interface (HCI)
Controller

Figure 3. The GAP abstraction layout

3.1.1. The GAP message sequences

This section introduces typical message sequences to provide more details about the events and procedures. For more details, refer to the `bt_gap.h`.

3.1.1.1. Set the scan mode

The scan mode controls whether the device can be discovered or connected by other Bluetooth-enabled devices, as shown in Figure 4. If set the scan mode is enabled, the device can be discovered by other Bluetooth devices. If the page scan is enabled, the device can be connected by other Bluetooth devices.

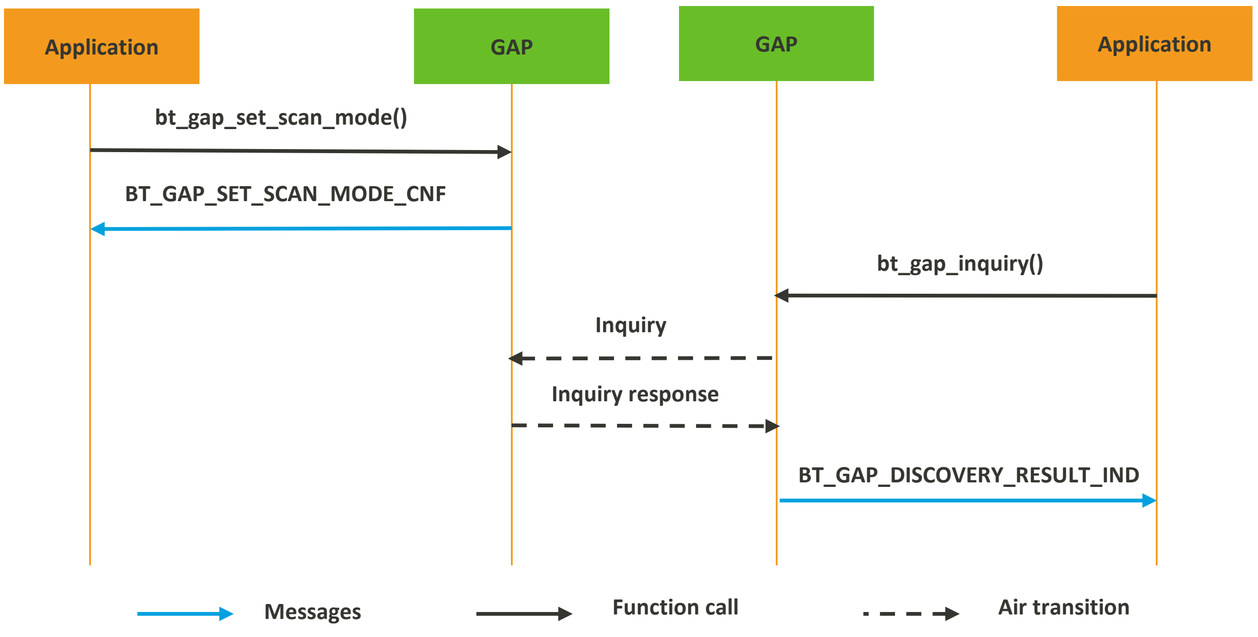

Figure 4. Set the scan mode message sequence

3.1.1.2. Inquiry

The purpose of the inquiry is to discover other Bluetooth devices in general or limited discoverable mode, as shown in Figure 5.

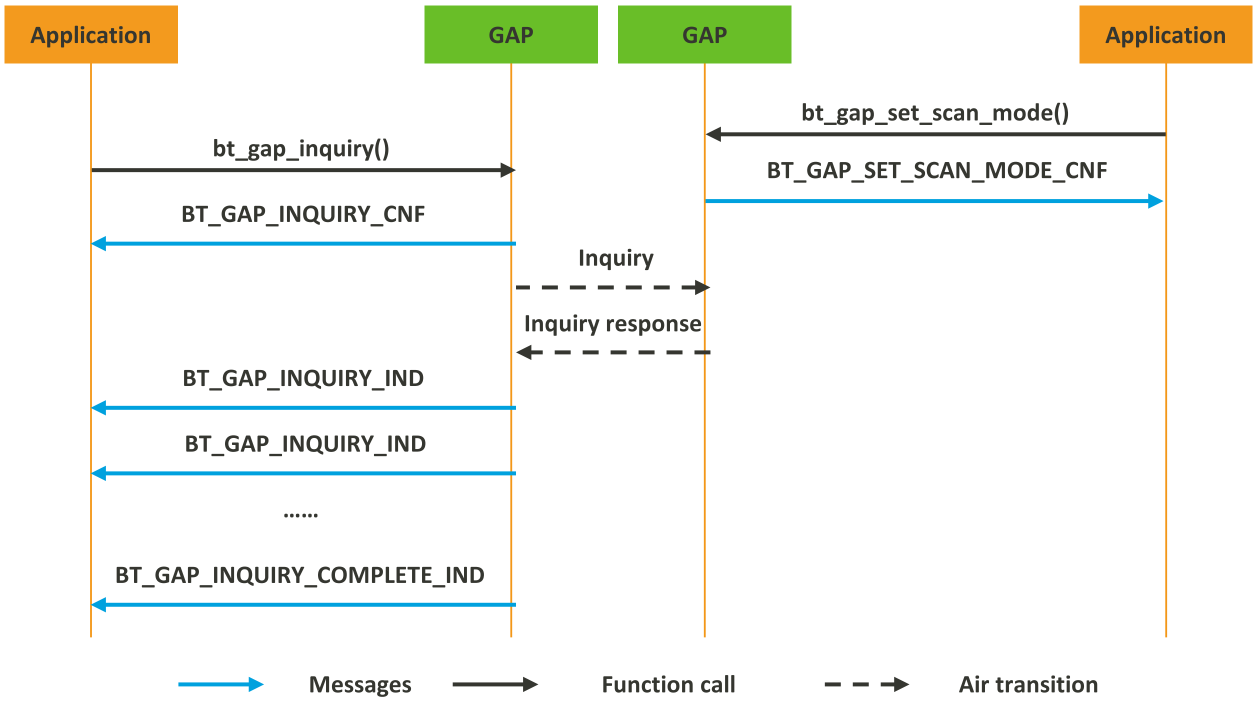

Figure 5. GAP inquiry message sequence

3.1.1.3. Bonding

The purpose of bonding procedure is to create a connection between two Bluetooth-enabled devices based on a common link key. The application should implement the user-defined API `bt_gap_get_link_key()`. After the bonding procedure starts, the GAP profile calls the function `bt_gap_get_link_key()` to get the link key from the application. If the link key already exists, proceed to authentication. Otherwise, the application receives BT_GAP_BONDING_START_IND event and begins the pairing. After the pairing is complete, the application can store the link key by handling the `BT_GAP_LINK_KEY_NOTIFICATION_IND` event, and BT_GAP_BONDING_COMPLETE_IND event will be sent to notify the bonding result. There is no bonding API to start a bonding procedure, it starts when connecting devices with authentication requirement. There are four bonding modes: auto-confirmation, numeric comparison, passkey entry and pin code.

1) The auto-confirmation model is applied where the local I/O capability is either BT_GAP_IO_CAPABILITY_NO_INPUT_NO_OUTPUT or BT_GAP_IO_CAPABILITY_DISPLAY_ONLY and the remote I/O capability isn't BT_GAP_IO_CAPABILITY_KEYBOARD_ONLY, as shown in Figure 6.

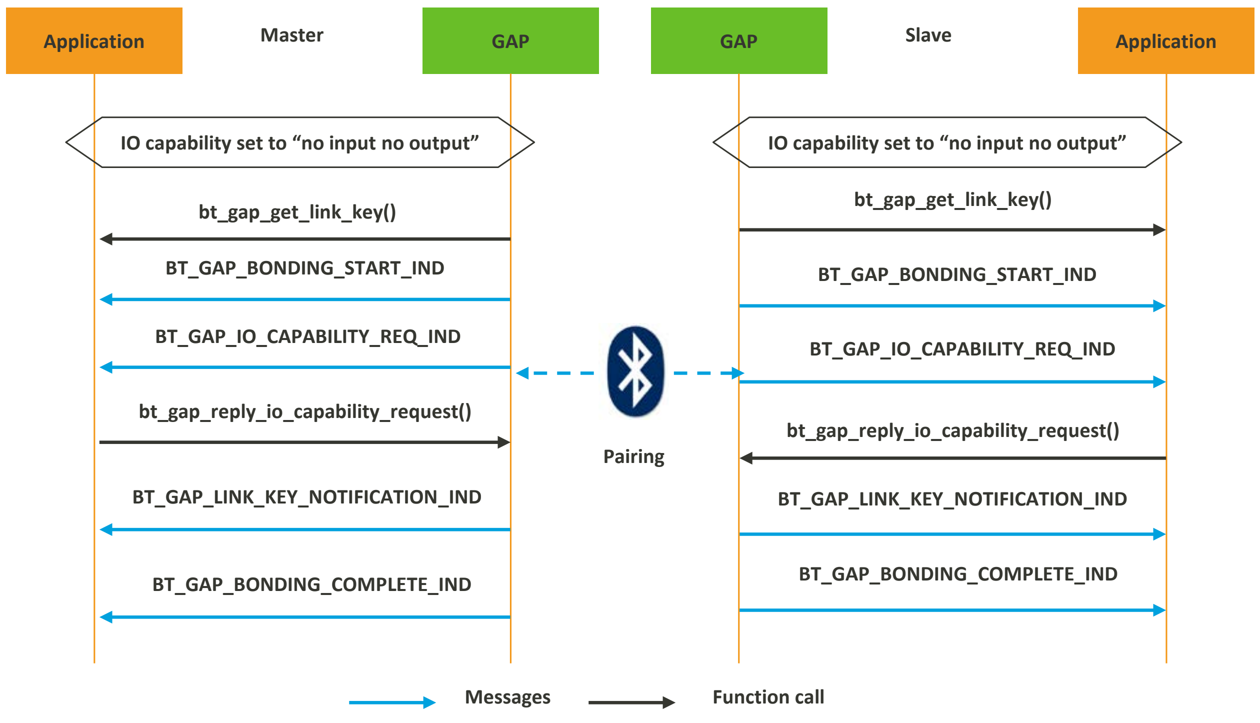
Application
Master
GAP
GAP
Slave
Application
IO capability set to "no input no output"
IO capability set to "no input no output"
bt_gap_get_link_key()
BT_GAP_BONDING_START_IND
BT_GAP_IO_CAPABILITY_REQ_IND
bt_gap_reply_io_capability_request()
BT_GAP_LINK_KEY_NOTIFICATION_IND
BT_GAP_BONDING_COMPLETE_IND
Pairing
Messages
Function call

Figure 6. Bonding using auto-confirmation message sequence

2) The numeric comparison model is applied when the local I/O capability is BT_GAP_IO_CAPABILITY_DISPLAY_YES_NO, as shown in Figure 7.

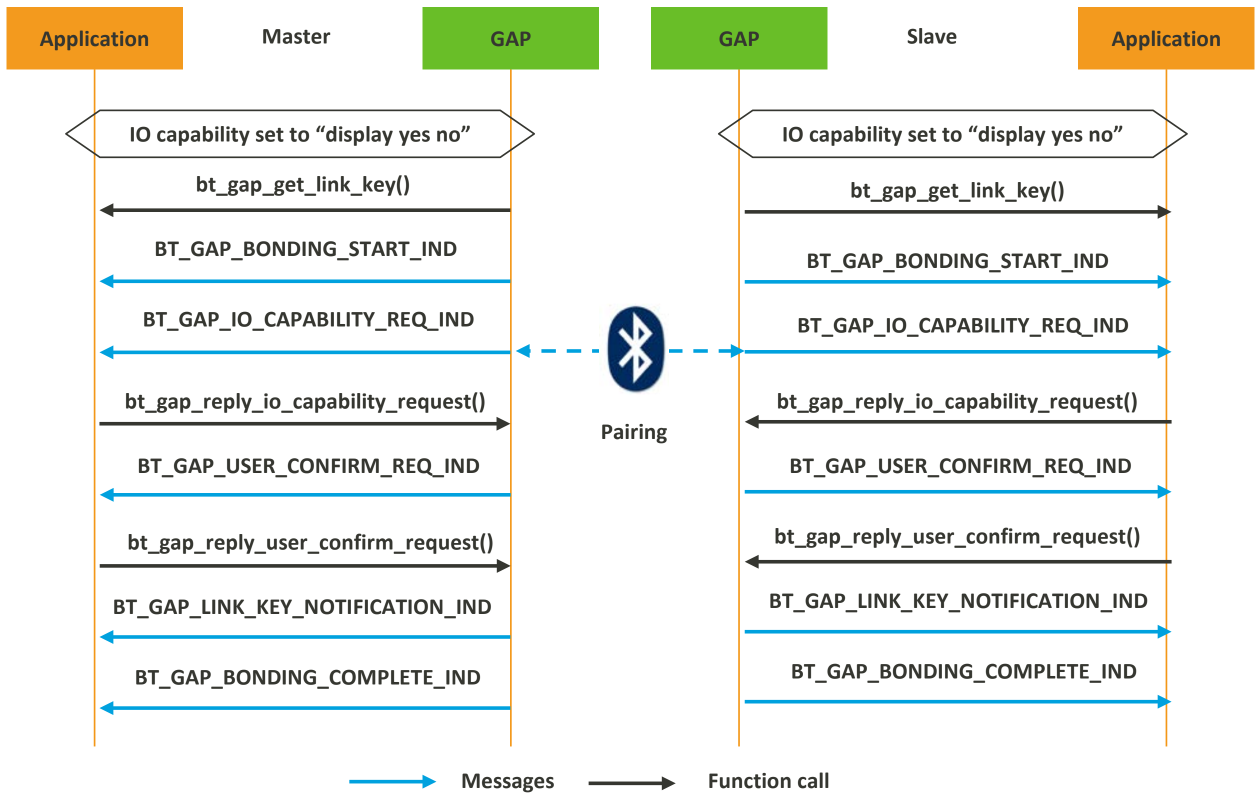
Application
Master
GAP
GAP
Slave
Application
IO capability set to "display yes no"
IO capability set to "display yes no"
bt_gap_get_link_key()
BT_GAP_BONDING_START_IND
BT_GAP_IO_CAPABILITY_REQ_IND
bt_gap_reply_io_capability_request()
BT_GAP_USER_CONFIRM_REQ_IND
bt_gap_reply_user_confirm_request()
BT_GAP_LINK_KEY_NOTIFICATION_IND
BT_GAP_BONDING_COMPLETE_IND
Pairing
Messages
Function call

Figure 7. Bonding using numeric comparison message sequence

3) The passkey entry model is used when one device has input capability but does not have the capability to display six digits and the other device has output capabilities, as shown in Figure 8.

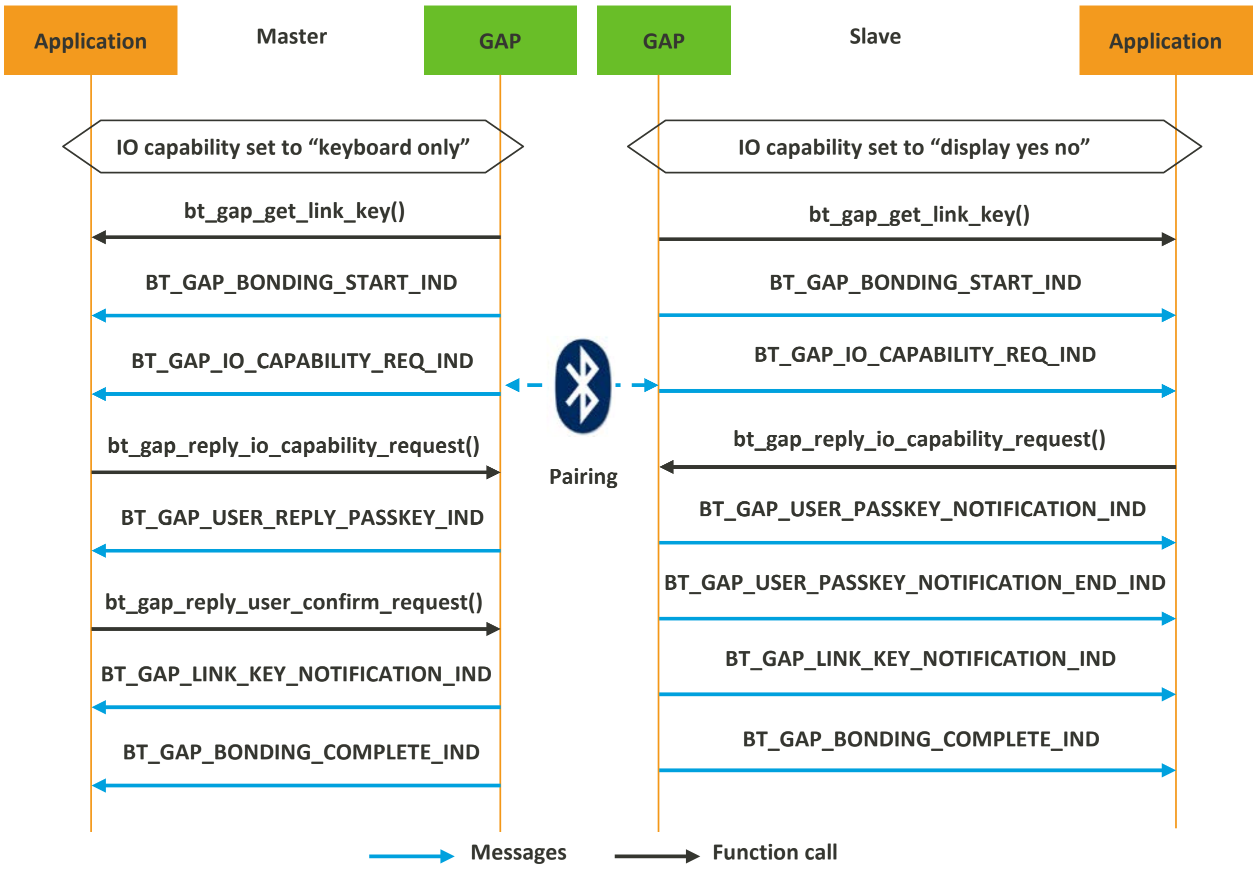
Application
Master
GAP
GAP
Slave
Application
IO capability set to "keyboard only"
IO capability set to "display yes no"
bt_gap_get_link_key()
BT_GAP_BONDING_START_IND
BT_GAP_IO_CAPABILITY_REQ_IND
bt_gap_reply_io_capability_request()
BT_GAP_USER_REPLY_PASSKEY_IND
bt_gap_reply_user_confirm_request()
BT_GAP_LINK_KEY_NOTIFICATION_IND
BT_GAP_BONDING_COMPLETE_IND
bt_gap_get_link_key()
BT_GAP_BONDING_START_IND
BT_GAP_IO_CAPABILITY_REQ_IND
bt_gap_reply_io_capability_request()
BT_GAP_USER_PASSKEY_NOTIFICATION_IND
BT_GAP_USER_PASSKEY_NOTIFICATION_END_IND
BT_GAP_LINK_KEY_NOTIFICATION_IND
BT_GAP_BONDING_COMPLETE_IND
Pairing
Messages
Function call

Figure 8. Bonding using passkey entry message sequence

4) The pin code model is used when one of the devices doesn't support SSP. The default pin code is "0000", this value can be modified by calling a user-defined API `bt_gap_get_pin_code()`, as shown in Figure 9.

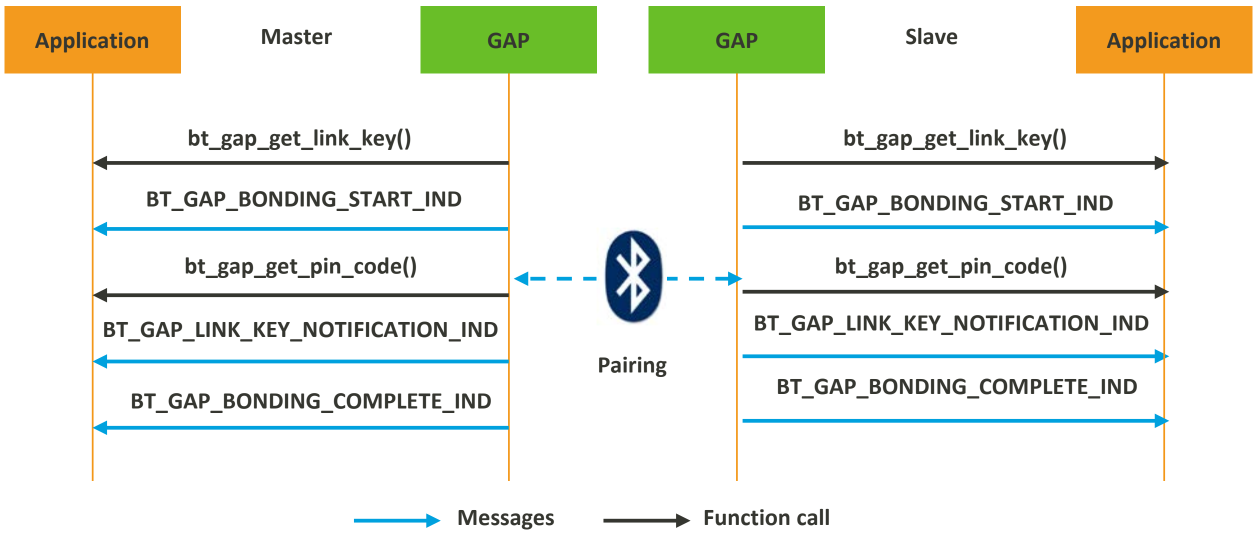

Figure 9. Bonding using pin code message sequence

3.1.1.4. Disconnect

The disconnect procedure terminates an existing ACL link, as shown in Figure 10.

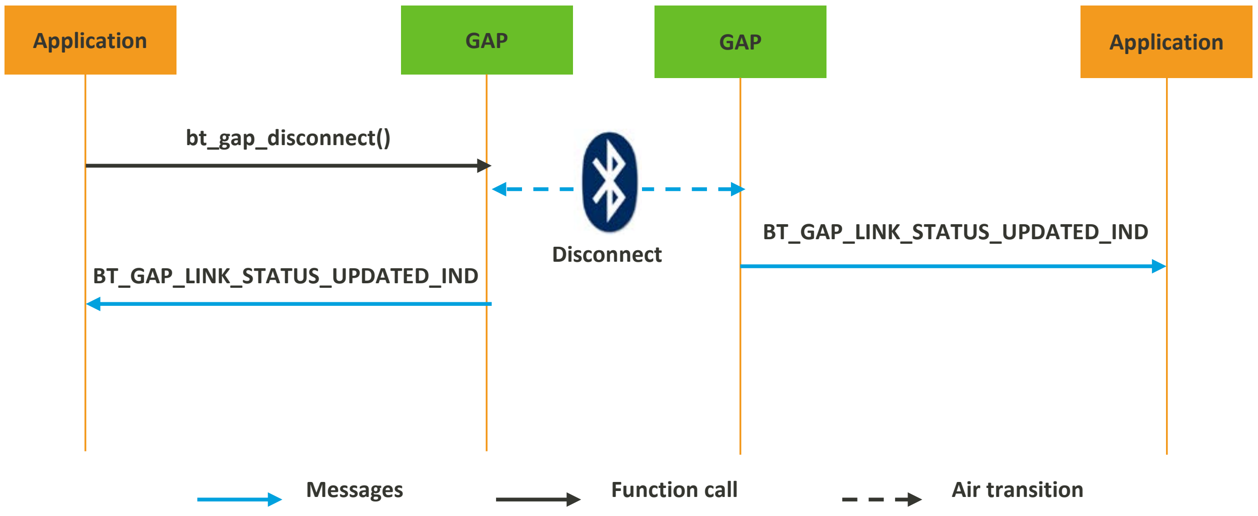

Figure 10. GAP disconnect message sequence

3.1.1.5. Cancel connection

The cancel connection procedure tries to cancel the creation of an ACL link, as shown in Figure 11.

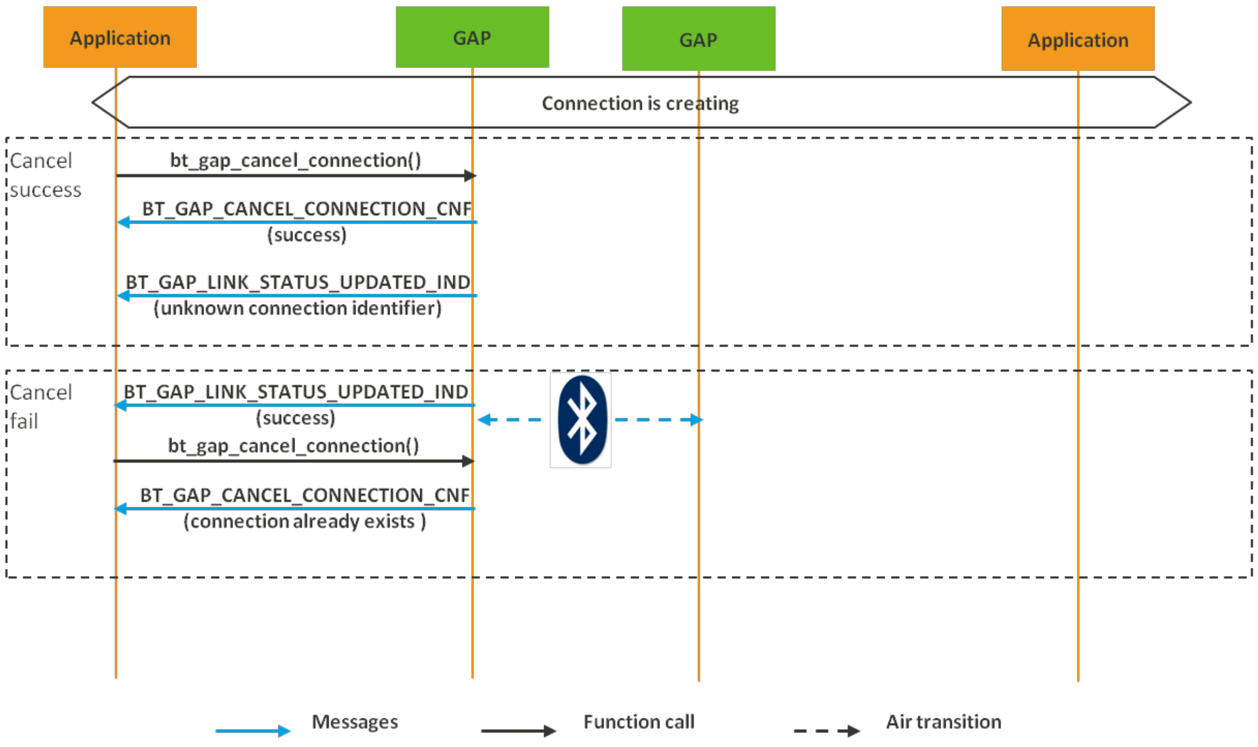
Application
GAP
GAP
Application
Connection is creating
Cancel success
bt_gap_cancel_connection()
BT_GAP_CANCEL_CONNECTION_CNF (success)
BT_GAP_LINK_STATUS_UPDATED_IND (unknown connection identifier)
Cancel fail
BT_GAP_LINK_STATUS_UPDATED_IND (success)
bt_gap_cancel_connection()
BT_GAP_CANCEL_CONNECTION_CNF (connection already exists)
Messages
Function call
Air transition

Figure 11. GAP cancels connection message sequence

3.1.2. Using the GAP APIs

This section describes how to use the GAP profile APIs. An example implementation is shown below.

1) Implement the required `bt_app_event_callback()` function to handle the GAP events, see section 3.1.2.1, "Full source code for `bt_app_event_callback()`".

2) Implement the required user-defined API `bt_gap_get_link_key()` to provide the link key stored during the last pairing procedure.

```
// The application may store the key in RAM, NVDM or permanent storage.
bt_gap_link_key_notification_ind_t edr_key;
void bt_gap_get_link_key(bt_gap_link_key_notification_ind_t*
key_information)
{
    if (memcmp(key_information->address, edr_key.address, 6) == 0) {
        memcpy(key_information, &edr_key,
sizeof(bt_gap_link_key_notification_ind_t));
    } else {
        // No link key, Do nothing
    }
}
```

3) Implement the required user-defined API `bt_gap_get_local_configuration()` to provide the GAP profile configuration.

```
static const bt_gap_config_t bt_config_default = {
    .inquiry_mode = 2, // Inquiry result with RSSI format or Extended
Inquiry Result (EIR) format.
    .io_capability = BT_GAP_IO_CAPABILITY_NO_INPUT_NO_OUTPUT,
    .cod = 0x240404, // Audio device type.
    .device_name = {"HB Duo device"},
};

// Define GAP configuration callback, it's invoked when Bluetooth is
powered on.
const bt_gap_config_t* bt_gap_get_local_configuration(void)
{
    return &bt_config_default; // Must return a global variable address.
}
```

4) Optional, implement the required user-defined API `bt_gap_get_pin_code()` to provide the pin code (default value is "0000").

```
// If local or remote device doesn't support SSP, the pin code pairing is
used.
// Default pin code is "0000", to replace it,  implement the function
bt_gap_get_pin_code().
static const bt_gap_pin_code_information_t bt_gap_my_pin_code =
{
    .pin_len = 4,
    .pin_code = {"1234"},
};

const bt_gap_pin_code_information_t* bt_gap_get_pin_code(void)
{
    return &bt_gap_my_pin_code; // Change the pin code from ”0000” to
"1234".
}
```

5) Call the function `bt_gap_set_scan_mode()` to set the device's scan mode in order to let other devices to discover it.

```
bt_gap_set_scan_mode(BT_GAP_MODE_GENERAL_ACCESSIBLE);
```

6) Call the function `bt_gap_inquiry()` to set the device in the inquiry mode to discover the nearby devices.

```
bt_gap_inquiry(10, 0);
```

7) Call the function `bt_gap_cancel_inquiry()` to cancel and exit the inquiry mode.

```
bt_gap_cancel_inquiry();
```

8) After the link is established, call related `bt_gap_get_remote_address()` to get the address.

```
const bt_bd_addr_t* address = bt_gap_get_remote_address(link_handle);
```

9) Call the function `bt_gap_read_remote_name()` to get the name of the remote device.

```
const bt_bd_addr_t address = {0xF1,0x24,0x0,0x22,0x48,0xFe};
bt_gap_read_remote_name(&address);
```

10) Call the function `bt_gap_disconnect()` to disconnect the link.

```
bt_gap_disconnect(link_handle);
```

3.1.2.1. Full source code for `bt_app_event_callback()`

```
void bt_app_event_callback(bt_msg_type_t msg, bt_status_t status, void
*buff)
{
    switch (msg) {
        case BT_GAP_INQUIRY_CNF:
        {
            if (status == BT_STATUS_SUCCESS) {
                      // Inquiry command executes successfully.
            } else {
                      // Inquiry command failed.
            }
            break;
        }
        case BT_GAP_INQUIRY_IND:
        {
            // Find a nearby device.
            bt_gap_inquiry_ind_t* device = (bt_gap_inquiry_ind_t*) buff;
            // Handle the event, example connecting to the device.
            break;
        }
        case BT_GAP_INQUIRY_COMPLETE_IND:
        {
            // Inquiry is complete.
            break;
        }
        case BT_GAP_SET_SCAN_MODE_CNF:
        {
                  if (status == BT_STATUS_SUCCESS) {
                      // bt_gap_set_scan_mode() executes successfully, it
should be found by nearby devices.
                  } else {
                      // Setting the scan mode has failed.
                  }
            break;
        }
        case BT_GAP_LINK_STATUS_UPDATED_IND:
        {
            bt_gap_link_status_updated_ind_t* param =
(bt_gap_link_status_updated_ind_t*) buff;
            // Handle link status update event.
            if (param->link_status == BT_GAP_LINK_STATUS_DISCONNECTED) {
                // The link is disconnected and it cannot re-connect in
this function callstack.
            } elseif (param->link_status >=
BT_GAP_LINK_STATUS_CONNECTED_0) {
                // The link is connected.
            }
            break;
        }
        case BT_GAP_BONDING_START_IND:
        {
            bt_gap_connection_handle_t* link_handle =
(bt_gap_connection_handle_t*) buff;
            // link_handle is going to bond.
            break;
        }
        case BT_GAP_IO_CAPABILITY_REQ_IND:
        {
            // This event will be received after BT_GAP_BONDING_START_IND
and both devices are support SSP.
            bt_gap_connection_handle_t handle;
            handle = (bt_gap_connection_handle_t)buff;
            // Call bt_gap_reply_io_capability_request() to accept
bongding, or call bt_gap_reject_io_capability_request() to terminate
bonding.
            break;
        }
        case BT_GAP_USER_CONFIRM_REQ_IND:
        {
            // This event will be received after
BT_GAP_IO_CAPABILITY_REQ_IND and IO Capability of both devices are
BT_GAP_IO_CAPABILITY_DISPLAY_YES_NO.
            // Call bt_gap_reply_user_confirm_request(true) to accept
bongding, or call bt_gap_reply_user_confirm_request(false) to terminate
bonding.
            break;
        }
        case BT_GAP_USER_PASSKEY_NOTIFICATION_IND:
        {
            // This event will be received after
BT_GAP_IO_CAPABILITY_REQ_IND, and local IO Capability is
BT_GAP_IO_CAPABILITY_DISPLAY_ONLY or BT_GAP_IO_CAPABILITY_DISPLAY_YES_NO
and remote IO Capability is BT_GAP_IO_CAPABILITY_KEYBOARD_ONLY
            // User should display the passkey on the screen.
            uint32_t* passkey = (uint32_t*) buff;
            // Call GDI APIs to show the passkey.
            break;
        }
        case BT_GAP_USER_PASSKEY_NOTIFICATION_END_IND:
        {
            // This event will be received after
BT_GAP_USER_PASSKEY_NOTIFICATION_IND.
            // User should stop showing the passkey.
```

```
                break;
            }
            case BT_GAP_USER_REPLY_PASSKEY_IND:
            {
                // This event will be received after
  BT_GAP_IO_CAPABILITY_REQ_IND, and remote IO Capability is
  BT_GAP_IO_CAPABILITY_DISPLAY_ONLY or BT_GAP_IO_CAPABILITY_DISPLAY_YES_NO
  and local IO Capability is BT_GAP_IO_CAPABILITY_KEYBOARD_ONLY
                // Call bt_gap_reply_passkey_request() to replay the passkey
  before passkey timeout (25 seconds).
                break;
            }
            case BT_GAP_LINK_KEY_NOTIFICATION_IND:
            {
                // This event will be received before
  BT_GAP_BONDING_COMPLETE_IND and bonding success, or the old link key is
  phased out.
                bt_gap_link_key_notification_ind_t * key_info =
  (bt_gap_link_key_notification_ind_t *) buff;
                if (key_info->key_type == BT_GAP_LINK_KEY_TYPE_INVALID) {
                          // The old link key is phased out, delete it from
  link key database.
                } else {
                    // Save the new link key, and key_type to link key data
  base.
                }
                break;
            }
            case BT_GAP_BONDING_COMPLETE_IND:
            {
                // Bonding is complete.
                if (status == BT_STATUS_SUCCESS) {
                    // Bonding completed successfully.
                } else {
                    // An error occurred.
                }
                break;
            }
            default:
            break;
        }
}
```

3.2. SDP

Service Discovery Protocol (SDP) is used to locate the available services of the remote device and manage the local available services, as shown in Figure 12.

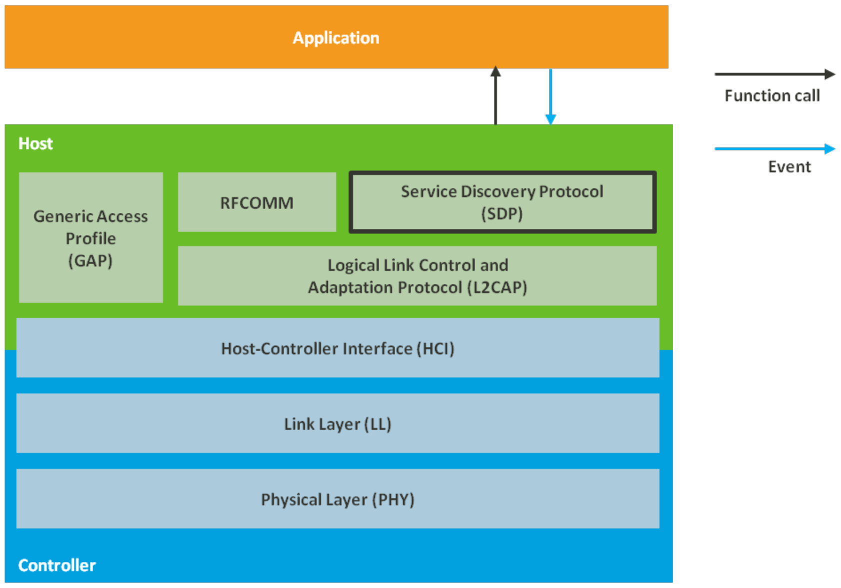
Application
Function call
Event
Host
Generic Access Profile (GAP)
RFCOMM
Service Discovery Protocol (SDP)
Logical Link Control and Adaptation Protocol (L2CAP)
Host-Controller Interface (HCI)
Link Layer (LL)
Physical Layer (PHY)
Controller

Figure 12. SDP abstraction layout

The SDP client-server architecture is shown in Figure 13. The client only supports two PDU types: SDP_ServiceSearchRequest and SDP_ServiceAttributeRequest. The user can customize service records on the server.

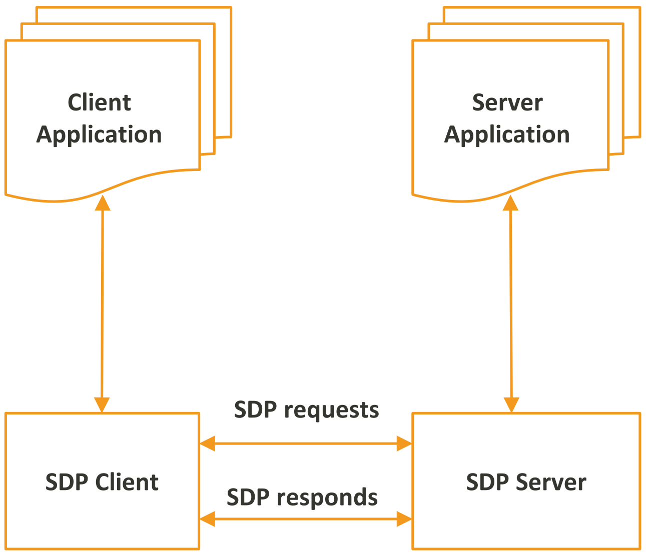

Figure 13. SDP client-server architecture

A service record on the SDP server contains a list of service attributes, as shown in Figure 14.

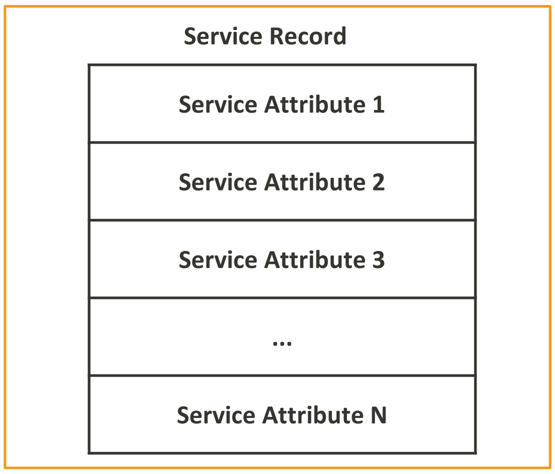

Figure 14. SDP service record

3.2.1. The SDP message sequences

The SDP API headers can be found in the bt_sdp.h. It’s mandatory to implement the function bt_sdps_get_customized_record() if any customized service record needs to be discovered by a remote device.

The application on the local device might check VID/PID or if the specific profile is supported on a peer device by sending an SDP query. In this case, the SDP acts as a client and calls the functions bt_sdpc_search_service() and bt_sdpc_search_attribute(), as shown in Figure 15.

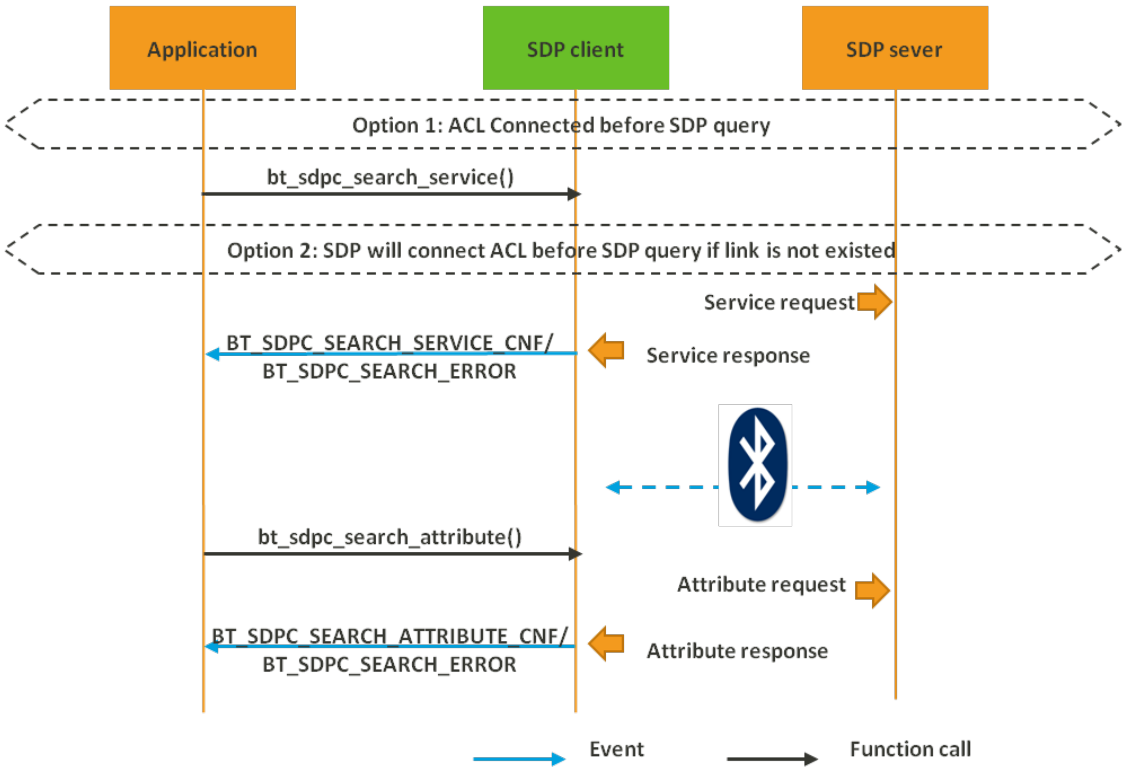
Application
SDP client
SDP sever
Option 1: ACL Connected before SDP query
bt_sdpc_search_service()
Option 2: SDP will connect ACL before SDP query if link is not existed
Service request
BT_SDPC_SEARCH_SERVICE_CNF/
BT_SDPC_SEARCH_ERROR
Service response
bt_sdpc_search_attribute()
Attribute request
BT_SDPC_SEARCH_ATTRIBUTE_CNF/
BT_SDPC_SEARCH_ERROR
Attribute response
Event
Function call

Figure 15. SDP client role

The application on the remote device usually checks if the specific profile is supported on a peer device by sending an SDP query before it connects. In this case, the SDP acts as a server and calls bt_sdps_get_customized_record() from the Bluetooth stack to check if there is any customized record, as shown in Figure 16.

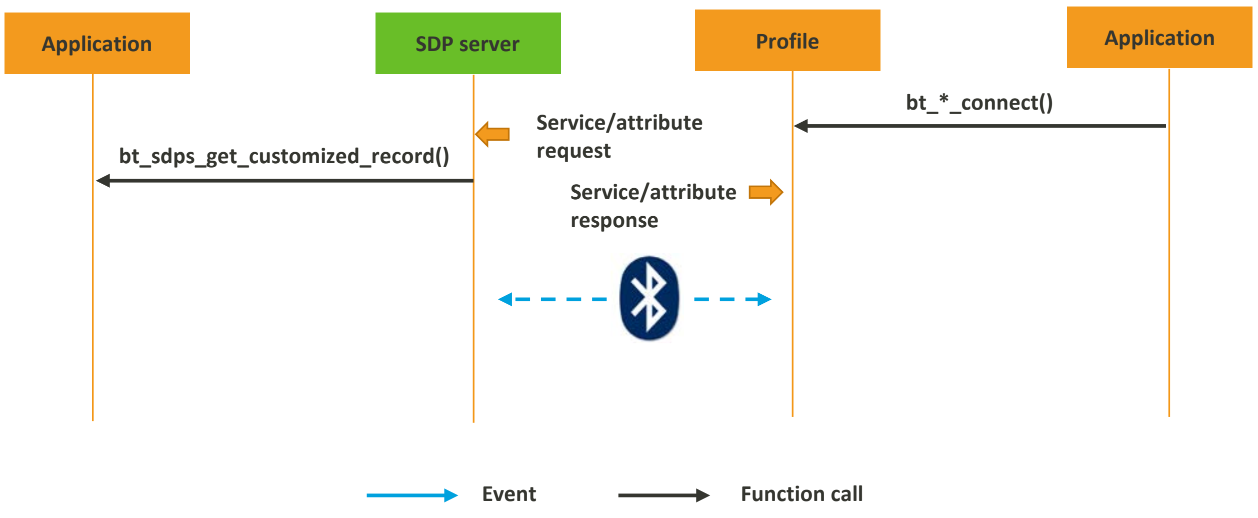
Application
SDP server
Profile
Application
bt_*_connect()
Service/attribute request
bt_sdps_get_customized_record()
Service/attribute response
Event
Function call

Figure 16. SDP server role

3.2.2. Add a customized record

Here is an example (see section 3.2.2.1, "Full source code to add a custom record") to add a custom record that contains the service name, language list, service class ID list and protocol description list for a sample profile. Add your own record according to the SDP profile specification. For more information about the service attributes, refer to the Bluetooth core specifications version 4.2 [VOL 3, part B, chapter 5]. More details about the API can be found in the header file bt_sdp.h.

Details of an example are explained below.

1) Define a service name with a given length.

```
static const uint8_t bt_sample_service_name[] =
{
    BT_SDP_TEXT_8BIT(7), // The string length of the service name.
    'S', 'a', 'm', 'p', 'l', 'e','\0'
};
```

2) Define a sample language for the profile.

   a) In this example, each item is 3 bytes and the total length of the list items is 9 bytes.

   b) The language is defined as "en". A language identifier represents natural language. Please refer to the ISO 639:1988 (E/F): "Code for the representation of names of languages" and ISO 639 Codes (Names of Languages).

   c) The encoding is defined as UTF-8, a character encoding identifier that can be found in IANA'S database and the Character Sets.

   d) Define a base attribute ID for the natural language in the service record.

```
static const uint8_t bt_sample_language_list[] =
{
    BT_SDP_ATTRIBUTE_HEADER_8BIT(9),
    BT_SDP_UINT_16BIT(0x656E),
    BT_SDP_UINT_16BIT(0x006A),
    BT_SDP_UINT_16BIT(0x0100)
};
```

3) Define the service class ID. In this example, each item is 3 bytes and the total length of the following list items is 6 bytes (see `BT_SDP_ATTRIBUTE_HEADER_8BIT(6)`).

```
static const uint8_t bt_sample_service_class_id_list[] =
{
  BT_SDP_ATTRIBUTE_HEADER_8BIT(6),
  BT_SDP_UUID_16BIT(BT_SDP_SERVICE_CLASS_HANDSFREE),
  BT_SDP_UUID_16BIT(BT_SDP_SERVICE_CLASS_GENERIC_AUDIO)
};
```

4) Define the attribute header for a protocol RFCOMM.

```
static const uint8_t bt_sample_protocol_description_list[] =
{
  BT_SDP_ATTRIBUTE_HEADER_8BIT(12), // The total length of all list items
is 12 bytes.

  BT_SDP_ATTRIBUTE_HEADER_8BIT(3),  // The total length of the following
list item is 3 bytes.
  BT_SDP_UUID_16BIT(BT_SDP_PROTOCOL_L2CAP),


  BT_SDP_ATTRIBUTE_HEADER_8BIT(5),  // The total length of the following
two list items is 5 bytes.
  BT_SDP_UUID_16BIT(BT_SDP_PROTOCOL_RFCOMM),
  BT_SDP_UINT_8BIT(0x01)
};
```

5) Define the SDP protocol attributes.

In this example the service name must add a base attribute ID, default value is 0x0100 (see `BT_SDP_ATTRIBUTE(BT_SDP_ATTRIBUTE_ID_SERVICE_NAME+0x0100, bt_sample_service_name)`) depending on your profile.

```
static const bt_sdps_attribute_t bt_sample_sdp_attributes[] =
{
  BT_SDP_ATTRIBUTE(BT_SDP_ATTRIBUTE_ID_SERVICE_CLASS_ID_LIST,
bt_sample_service_class_id_list),
  BT_SDP_ATTRIBUTE(BT_SDP_ATTRIBUTE_ID_PROTOCOL_DESC_LIST,
bt_sample_protocol_description_list),
  BT_SDP_ATTRIBUTE(BT_SDP_ATTRIBUTE_ID_LANG_BASE_ID_LIST,
bt_sample_language_list),
  BT_SDP_ATTRIBUTE(BT_SDP_ATTRIBUTE_ID_SERVICE_NAME+0x0100,
bt_sample_service_name)
};
```

6) Create an attribute list with records.

```
static const bt_sdps_record_t bt_sample_sdp_record =
{
  .attribute_list_length = sizeof(bt_sample_sdp_attributes),
  .attribute_list = bt_sample_sdp_attributes,
};
```

7) Define an SDP sample record.

```
static const bt_sdps_record_t *sdps_sample_record[]= {
  &bt_sample_sdp_record,
  };
```

8) Get a customized record.

This API invoked by the SDK process should be implemented by the application. If no records to be found by the remote device, set the `record_list` to NULL and return 0.

```
  uint8_t bt_sdps_get_customized_record(const bt_sdps_record_t ***
  record_list)
  {
    *record_list = &sdps_sample_record;
    return sizeof(sdps_sample_record)/sizeof(bt_sdps_record_t*);
}
```

3.2.2.1. Full source code to add a custom record

```
  static const uint8_t bt_sample_service_name[] =
  {
      BT_SDP_TEXT_8BIT(7), // The string length of the service name.
      'S', 'a', 'm', 'p', 'l', 'e','\0'
  };

  static const uint8_t bt_sample_language_list[] =
  {
      BT_SDP_ATTRIBUTE_HEADER_8BIT(9),
      BT_SDP_UINT_16BIT(0x656E),
      BT_SDP_UINT_16BIT(0x006A),
      BT_SDP_UINT_16BIT(0x0100)
  };

  static const uint8_t bt_sample_service_class_id_list[] =
  {
    BT_SDP_ATTRIBUTE_HEADER_8BIT(6),
    BT_SDP_UUID_16BIT(BT_SDP_SERVICE_CLASS_HANDSFREE),
    BT_SDP_UUID_16BIT(BT_SDP_SERVICE_CLASS_GENERIC_AUDIO)
  };

  static const uint8_t bt_sample_protocol_description_list[] =
  {
    BT_SDP_ATTRIBUTE_HEADER_8BIT(12),
    BT_SDP_ATTRIBUTE_HEADER_8BIT(3),
    BT_SDP_UUID_16BIT(BT_SDP_PROTOCOL_L2CAP),
    BT_SDP_ATTRIBUTE_HEADER_8BIT(5),
    BT_SDP_UUID_16BIT(BT_SDP_PROTOCOL_RFCOMM),
    BT_SDP_UINT_8BIT(0x01)
  };

  static const bt_sdps_attribute_t bt_sample_sdp_attributes[] =
  {
    BT_SDP_ATTRIBUTE(BT_SDP_ATTRIBUTE_ID_SERVICE_CLASS_ID_LIST,
  bt_sample_service_class_id_list),
    BT_SDP_ATTRIBUTE(BT_SDP_ATTRIBUTE_ID_PROTOCOL_DESC_LIST,
  bt_sample_protocol_description_list),
    BT_SDP_ATTRIBUTE(BT_SDP_ATTRIBUTE_ID_LANG_BASE_ID_LIST,
  bt_sample_language_list),
    BT_SDP_ATTRIBUTE(BT_SDP_ATTRIBUTE_ID_SERVICE_NAME+0x0100,
  bt_sample_service_name)
  };

  static const bt_sdps_record_t bt_sample_sdp_record =
  {
    .attribute_list_length = sizeof(bt_sample_sdp_attributes),
    .attribute_list = bt_sample_sdp_attributes,
  };

  static const bt_sdps_record_t *sdps_sample_record[]= {
    &bt_sample_sdp_record,
  };

  uint8_t bt_sdps_get_customized_record(const bt_sdps_record_t ***
  record_list)
  {
    *record_list = &sdps_sample_record;
    return sizeof(sdps_sample_record)/sizeof(bt_sdps_record_t*);
  }
```

3.2.3. Use client API

Here is an example (see section 3.2.2.1, "Full source code to add a custom record client API") to use client API to get Vendor ID of remote device. For more information about the client protocol, refer to the Bluetooth core specifications version 4.2 [VOL 3, part B, chapter 5]. More details about the API can be found in the header file `bt_sdp.h`.

Details of an example are explained below.

1) Define a search pattern for service search.
   a) In this example, only one UUID item and it is 3 bytes and the total length of the UUID list items is 3 bytes.
   b) The UUID is 16 bit and defined as "PNP information".
   c) Define the maximum handle number returned by remote device.

```
static const uint8_t bt_di_search_service_pattern[] =
{
    BT_SDP_ATTRIBUTE_HEADER_8BIT(3), //Data element sequence, each
UUID below is 3 bytes.
      BT_SDP_UUID_16BIT(BT_DI_SDP_SERVICE_CLASS_PNP_INFORMATION),
//UUID for PnP information in big endian.
    0x02,0x00 //Maximum number of handles to be returned is 0x0200.
};
```

2) Define a search pattern for attribute search.
   a) Define the maximum number of bytes of attribute data returned by remote device.
   b) In this example, each attribute item is 3 bytes and the total length of the attribute id list items is 6 bytes.
   c) The attribute data of "vendor ID" of DI profile is expected to be returned.

d) The attribute data of “product ID” of DI profile is expected to be returned.

```
static const uint8_t bt_di_search_attribute_pattern[] =
{
    0x00, 0x64, //Maximum number of bytes of attribute data to be
returned is 0x0064.
    BT_SDP_ATTRIBUTE_HEADER_8BIT(6), //Data element sequence,
each attribute ID below is 3 bytes.
    BT_SDP_UINT_16BIT(BT_DI_SDP_ATTRIBUTE_VENDOR_ID),
    BT_SDP_UINT_16BIT(BT_DI_SDP_ATTRIBUTE_PRODUCT_ID),
};
```

3) Use API bt_sdpc_search_service() to query the service handle, and wait for event BT_SDPC_SEARCH_SERVICE_CNF or BT_SDPC_SEARCH_ERROR in bt_app_event_callback().

```
void bt_sample_get_di_vendor_id(bt_bd_addr_t * addr)
{
    bt_sdpc_service_request_t request = {{0}};
    memcpy(di_profile.peer_address, addr, sizeof(bt_bd_addr_t));
    di_profile.state = 1;//query is ongoing
    request.search_pattern = bt_di_search_service_pattern;
    request.pattern_length = sizeof(bt_di_search_service_pattern);
    request.address = addr;
    request.user_data = (void *)&di_profile;
    result = bt_sdpc_search_service(&request);
}
```

4) After service handles are returned, use API bt_sdpc_search_attribute to query the service attributes, and wait for event BT_SDPC_SEARCH_ATTRIBUTE_CNF or BT_SDPC_SEARCH_ERROR in bt_app_event_callback().

```
...........
if (msg == BT_SDPC_SEARCH_SERVICE_CNF) {
        bt_sdpc_service_cnf_t *cnf = (bt_sdpc_service_cnf_t*)buff;
        if (cnf->user_data == &di_profile && di_profile.state) {
            bt_profile_user_data_t *user_data =
(bt_profile_user_data_t *)cnf->user_data;
            bt_status_t result;

            if(cnf->handle_number > 0) {
                uint32_t query_service_handle =
bt_sdp_get_32bit(cnf->handle_data);

                bt_sdpc_attribute_request_t request = {{0}};

                request.attribute_pattern =
bt_di_search_attribute_pattern;
                request.pattern_length =
sizeof(bt_di_search_attribute_pattern);
                request.search_handle = query_service_handle;
                request.user_data = user_data;
                request.address = user_data.peer_address;
                result = bt_sdpc_search_attribute(&request);
............
            }
        }
}
```

3.2.3.1. Full source code to add a custom record client API

```
static const uint8_t bt_di_search_service_pattern[] =
{
    BT_SDP_ATTRIBUTE_HEADER_8BIT(3),                            //Data
Element Sequence, 3 bytes
    BT_SDP_UUID_16BIT(BT_DI_SDP_SERVICE_CLASS_PNP_INFORMATION), // UUID
for PnP information in Big Endian
    0x02,0x00                                                   //max
returned handle number
};

static const uint8_t bt_di_search_attribute_pattern[] =
{
    0x00, 0x64, //0x0064, max handle for attribute return
    BT_SDP_ATTRIBUTE_HEADER_8BIT(6),
    BT_SDP_UINT_16BIT(BT_DI_SDP_ATTRIBUTE_VENDOR_ID),
    BT_SDP_UINT_16BIT(BT_DI_SDP_ATTRIBUTE_PRODUCT_ID),
};

typedef struct {
    bt_bd_addr_t peer_address;
    uint8_t state;
}bt_profile_user_data_t;

static bt_profile_user_data_t di_profile;

void bt_sample_get_di_vendor_id(bt_bd_addr_t * addr)
{
    bt_sdpc_service_request_t request = {{0}};
    memcpy(di_profile.peer_address, addr, sizeof(bt_bd_addr_t));
    di_profile.state = 1;//query is ongoing
    request.search_pattern = bt_di_search_service_pattern;
    request.pattern_length = sizeof(bt_di_search_service_pattern);
    request.address = addr;
    request.user_data = (void *)&di_profile;
    result = bt_sdpc_search_service(&request);
}
static void bt_app_event_callback(bt_msg_type_t msg, bt_status_t status,
void *buff)
{
    if (msg == BT_SDPC_SEARCH_SERVICE_CNF) {
        bt_sdpc_service_cnf_t *cnf = (bt_sdpc_service_cnf_t*)buff;
        if (cnf->user_data == &di_profile && di_profile.state) {
            bt_profile_user_data_t *user_data = (bt_profile_user_data_t
*)cnf->user_data;
            bt_status_t result;

            if(cnf->handle_number > 0) {
                uint32_t query_service_handle = bt_sdp_get_32bit(cnf-
>handle_data);

                bt_sdpc_attribute_request_t request = {{0}};

                request.attribute_pattern =
bt_di_search_attribute_pattern;
                request.pattern_length =
sizeof(bt_di_search_attribute_pattern);
```

```
                request.search_handle = query_service_handle;
                request.user_data = user_data;
                request.address = user_data.peer_address;
                result = bt_sdpc_search_attribute(&request);

                if (result == BT_STATUS_OUT_OF_MEMORY ||
                    result == BT_STATUS_BUSY) {
                    //retry again later
                    return;
                }
            } else {
                BT_LOGI("APP", "no di is found\n");
            }
        }
    } else if (msg == BT_SDPC_SEARCH_ATTRIBUTE_CNF) {
        uint8_t *attr_value;
        uint8_t *supported_value;
        uint16_t value_len;
        bt_status_t result;

        bt_sdpc_attribute_cnf_t *attr_query_result =
(bt_sdpc_attribute_cnf_t *)buff;
        if (attr_query_result->user_data == &di_profile &&
di_profile.state) {
            if (attr_query_result->length > 0) {
                result = bt_sdpc_parse_attribute(
                    &attr_value,
                    &value_len,
                    BT_DI_SDP_ATTRIBUTE_VENDOR_ID,
                    0,
                    attr_query_result->length,
                    attr_query_result->attribute_data);
                if (result != 0) {
                    BT_LOGI("APP", "no di vid is found\n");
                    return;
                }
                result = bt_sdpc_parse_next_value(
                    &supported_value,
                    &value_len,
                    attr_value,
                    value_len);

                if (value_len && !result) {
                    uint16_t vid;
                    if (value_len != 2) {
                        assert(0);
                        return;
                    }
                    vid = bt_sdp_get_16bit(supported_value);
                    BT_LOGI("APP", "di vid is %x\n", vid);
                } else {
                    BT_LOGI("APP", "no di vid is found\n");
                }
            } else {
                BT_LOGI("APP", "no match avrcp attribute is found\n");
            }
            di_profile.state = 0;
        }
    } else if (msg == BT_SDPC_SEARCH_ERROR) {
        bt_sdpc_search_error_t *error = (bt_sdpc_search_error_t *)buff;
        if (attr_query_result->user_data == &di_profile &&
di_profile.state) {
            BT_LOGI("APP", "sdp query error %x\n", status);
        }
        di_profile.state = 0;
    }

}
```

3.3. HFP

The profile defines details on how two devices supporting HFP interact on a point-to-point basis. An implementation of the HFP typically enables a headset or an embedded hands-free unit to connect wirelessly to a cellular phone for the purposes of acting as the cellular phone's audio input and output and allowing typical telephony functions to be performed without access to the actual phone.

The following roles are defined for this profile:

- **Audio Gateway (AG)** — a device that acts as input and output gateway of the audio. Typical devices acting as AG are cellular phones.
- **Hands-Free unit (HF)** — a device that acts as the AG's remote audio input and output. It also provides remote control capabilities.

Note: Only the HF role is supported in the Airoha IoT SDK v4.

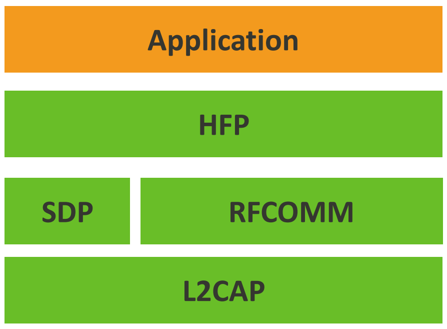

Figure 17. HFP abstraction layout

The HFP depends on RFCOMM, which defines procedures required to send and receive files between two Bluetooth-enabled devices. HFP APIs are called in the **Application** to implement the features related to the HFP.

The HF is responsible for HFP connection management, audio connection management and call related action handling.

3.3.1. HFP message sequences

The HFP procedure can be established using the message sequence. The message sequence for each process is described below.

- Connection establishment

- Connection release
- Initiate an outgoing voice call
- Audio connection setup
- Audio connection release
- Enable or disable the voice recognition
- Incoming call actions
- Three way call handling
- Remote speaker volume control
- Remote microphone volume control
- Transmit DTMF codes
- Remote speaker volume control

3.3.1.1. Connection establishment

Apply this process to establish HFP connection between devices, as shown in Figure 18. The SDK provides two different message sequences; either application driven or remote device driven. For more details, refer to the header file `bt_hfp.h`.

```
bt_status_t status;
uint32_t hfp_handle;

status = bt_hfp_connect(&hfp_handle, bt_addr);

bt_status_t bt_app_event_callback(bt_msg_type_t msg, bt_status_t status,
void *buff)
{
    switch (msg) {
        case BT_HFP_SLC_CONNECTING_IND:
        {
            break;
        }
        case BT_HFP_SLC_CONNECTED_IND:
        {
            break;
        }
    }
    return BT_STATUS_SUCCESS;
}
```

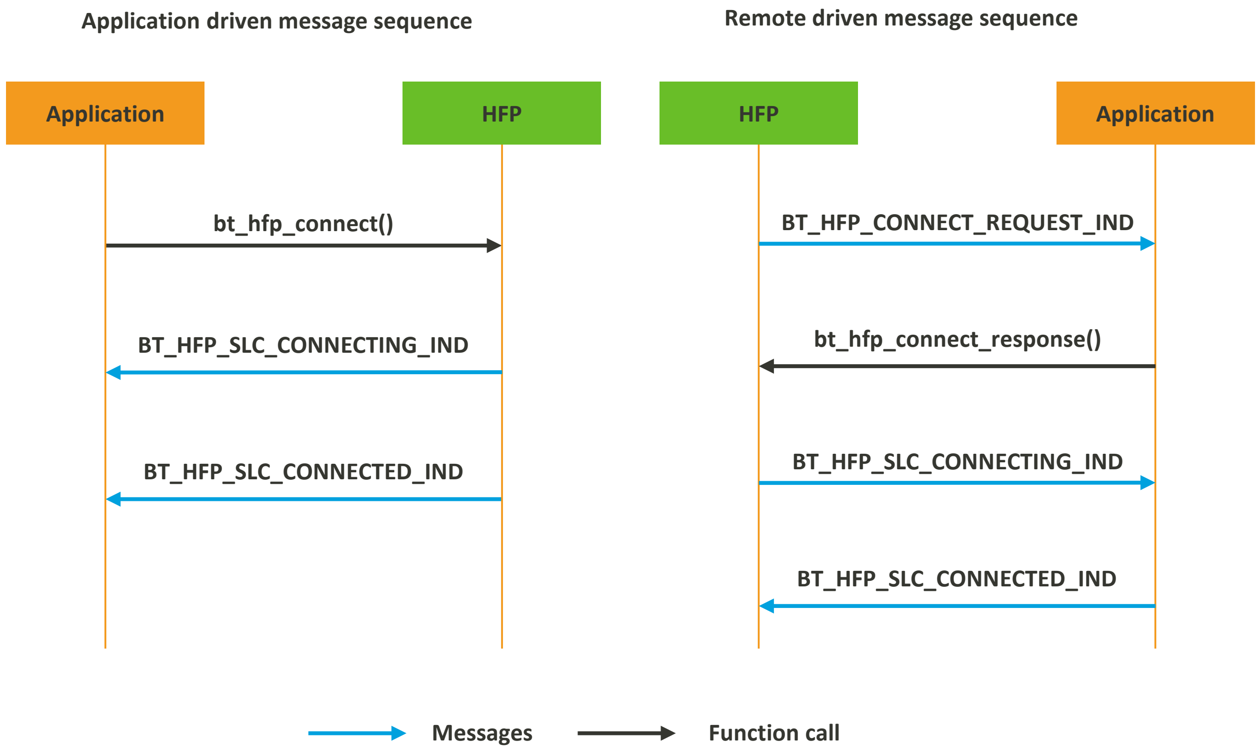
Application driven message sequence
Remote driven message sequence
Application
HFP
HFP
Application
bt_hfp_connect()
BT_HFP_SLC_CONNECTING_IND
BT_HFP_SLC_CONNECTED_IND
BT_HFP_CONNECT_REQUEST_IND
bt_hfp_connect_response()
BT_HFP_SLC_CONNECTING_IND
BT_HFP_SLC_CONNECTED_IND
Messages
Function call

Figure 18. HFP connection establishment message sequence

3.3.1.2. Connection release

Apply this procedure to release the HFP connection, as shown in Figure 19. The SDK provides two different message sequences; either application driven or remote device driven. For more details, refer to the `bt_hfp.h`.

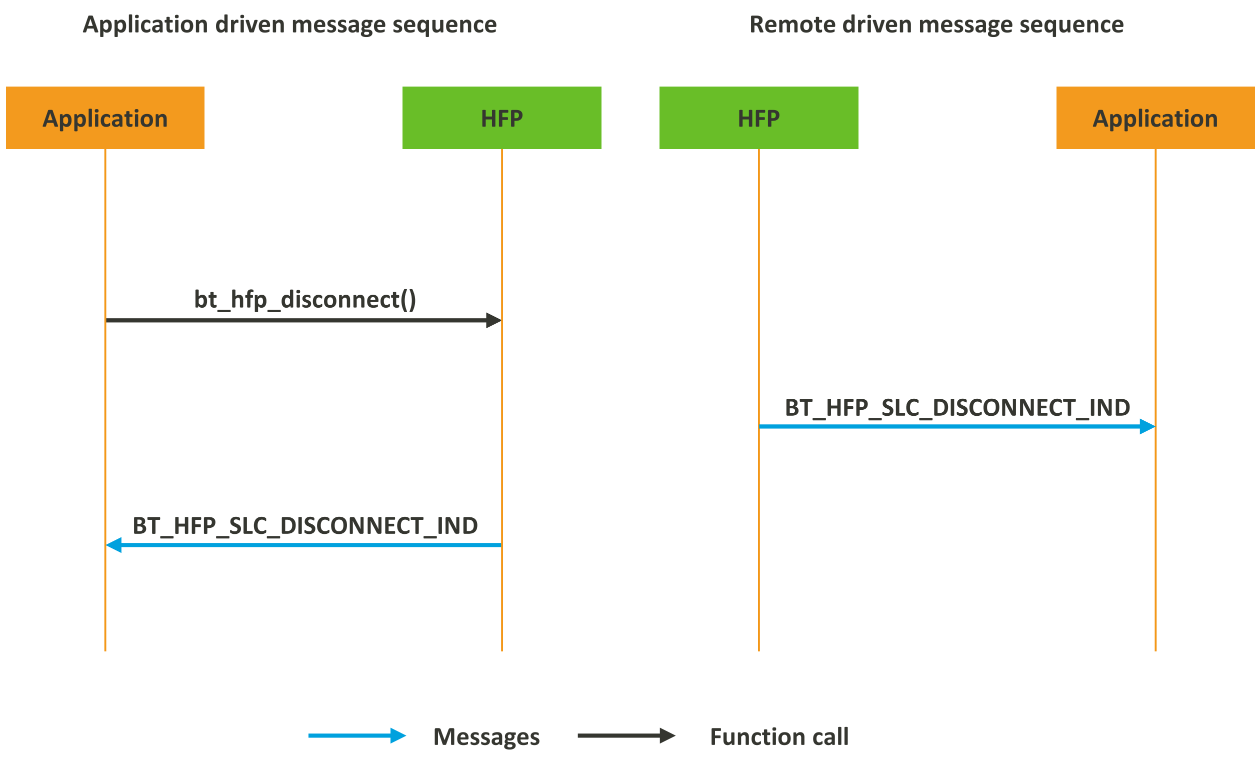
Application driven message sequence
Remote driven message sequence
Application
HFP
HFP
Application
bt_hfp_disconnect()
BT_HFP_SLC_DISCONNECT_IND
BT_HFP_SLC_DISCONNECT_IND
Messages
Function call

Figure 19. HFP connection release message sequence

3.3.1.3. Initiate an outgoing voice call

Apply this procedure to initiate an outgoing voice call from the HF. Application uses the API `bt_hfp_send_command()` and passes the command "ATDnnn", (where nnn is the phone number to dial) to make an outgoing call, as shown in Figure 20. For more details, refer to the `bt_hfp.h` and HFP specification version 1.6.

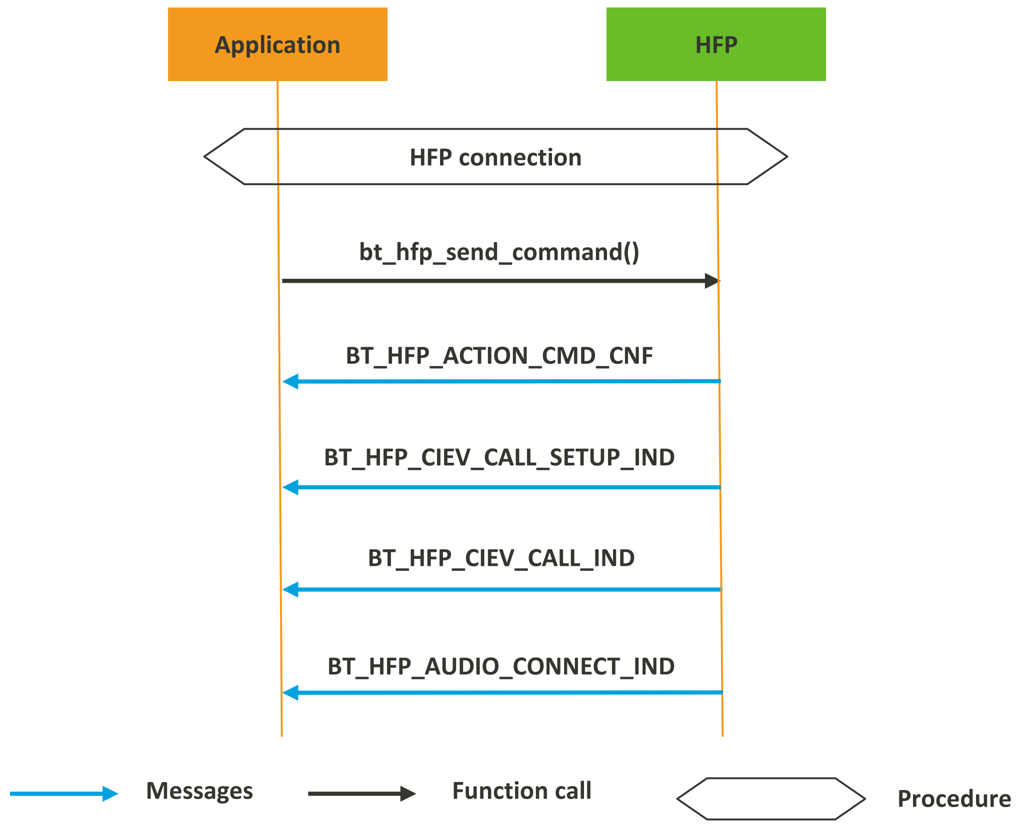

Figure 20. Initiate an outgoing voice call message sequence

3.3.1.4. Audio connection setup

Apply this procedure to create an SCO link, as shown in Figure 21. The SDK provides two different message sequences; either application driven or remote device driven. The application driven message sequence uses the API `bt_hfp_audio_transfer()` and passes the director parameter of "BT_HFP_AUDIO_TO_HF" to create audio connection. For more details, refer to the `bt_hfp.h` and HFP specification version 1.6.

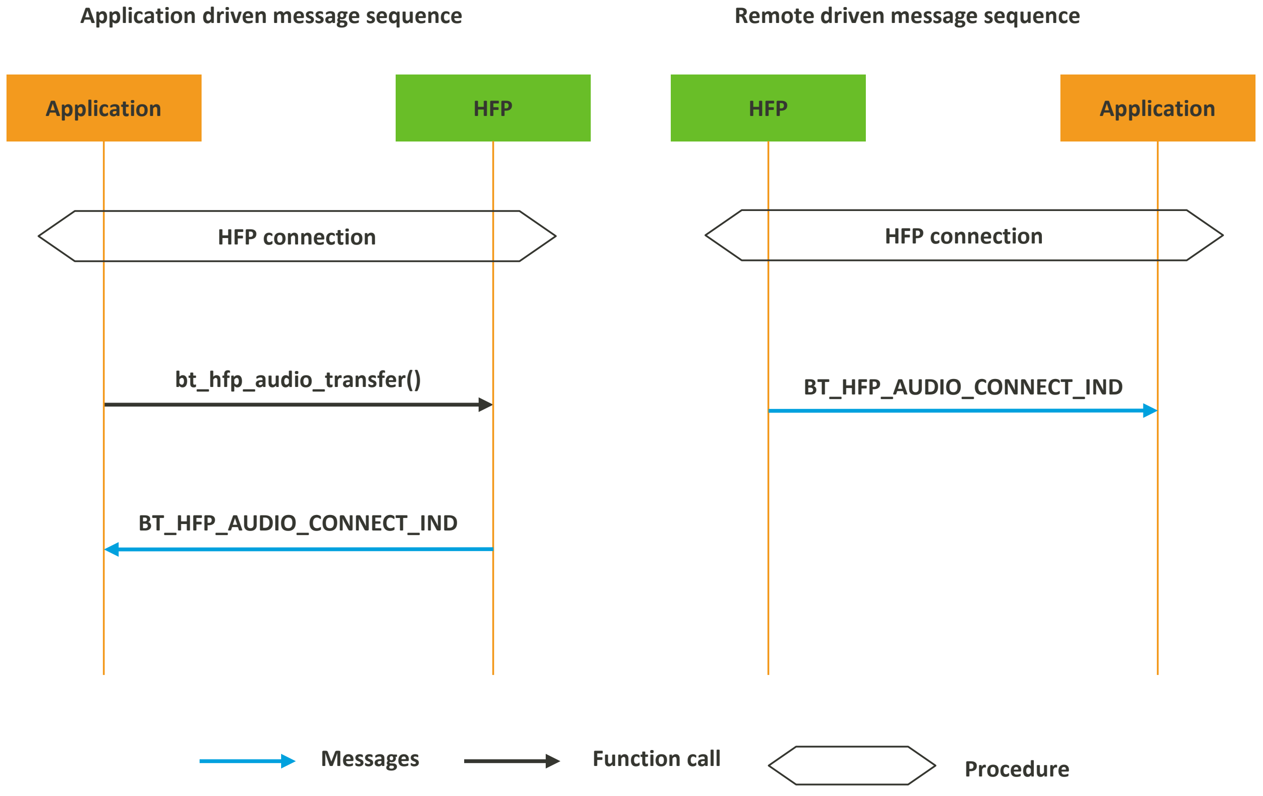

Figure 21. Audio connection setup message sequence

3.3.1.5. Audio connection release

Apply this process to release the SCO link, as shown in Figure 22. The SDK provides two different message sequences; either application driven or remote device driven. The application driven message sequence uses the API `bt_hfp_audio_transfer()` and passes the director parameter of "BT_HFP_AUDIO_TO_AG" to release the audio connection. For more details, refer to the `bt_hfp.h` and HFP specification version 1.6.

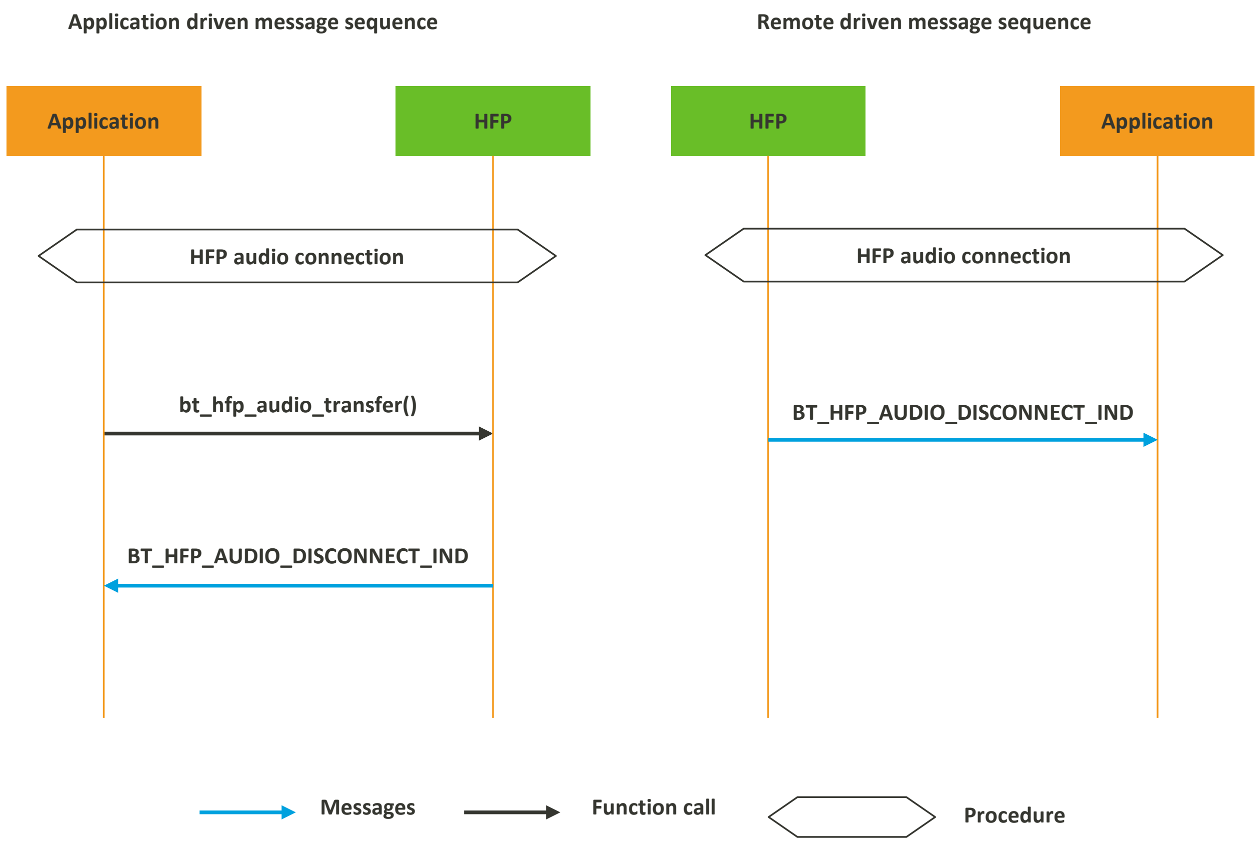

Figure 22. Audio connection release message sequence

3.3.1.6. Enable or disable the voice recognition

Apply this process to enable or disable voice recognition, as shown in Figure 23. The SDK provides two different message sequences; either application driven or remote device driven. The application driven message sequence uses the API `bt_hfp_send_command()` and passes the command "AT+BVRA=1" or "AT+BVRA=0" to enable or disable the voice recognition. For more details, refer to the `bt_hfp.h` and HFP specification version 1.6.

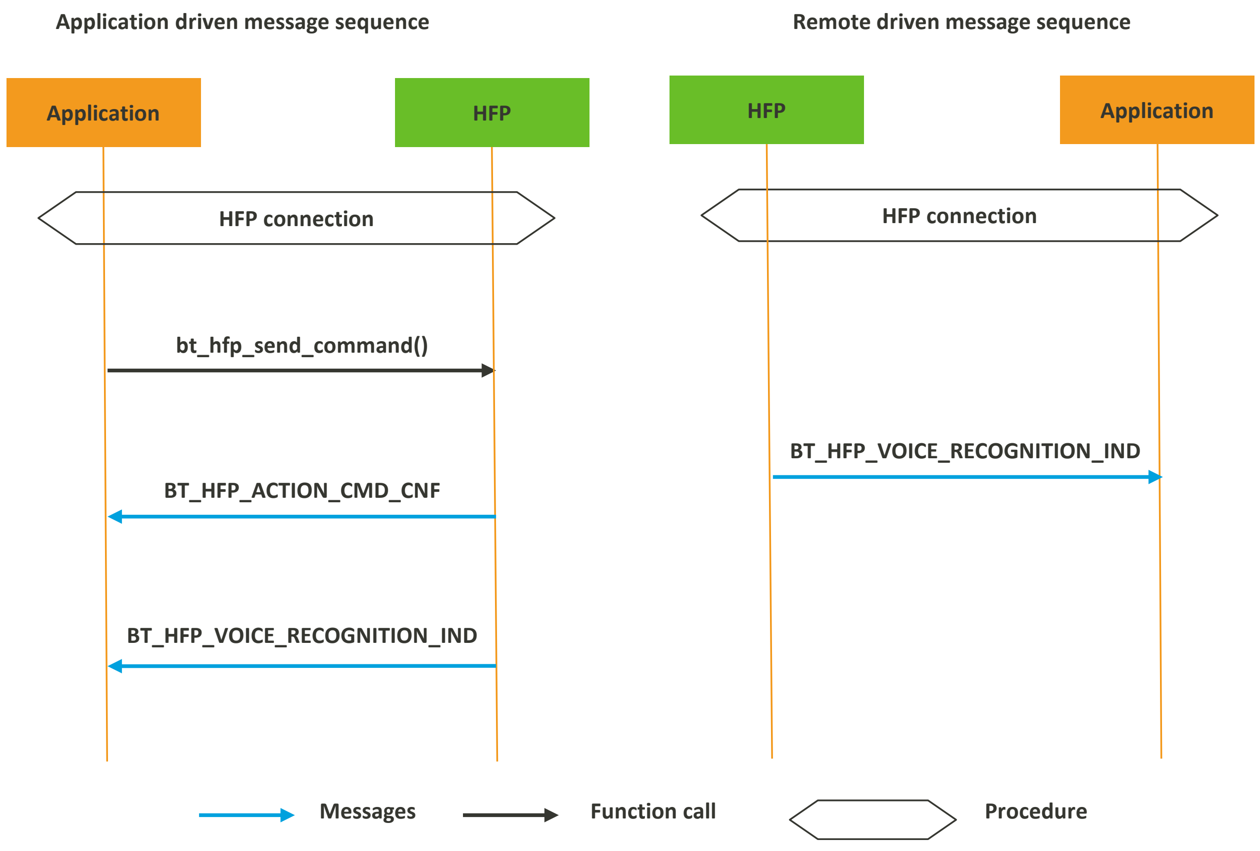

Figure 23. Enable or disable the voice recognition message sequence

3.3.1.7. Incoming call actions

Apply this procedure to accept or reject an incoming call, as shown in Figure 24. Application uses the API `bt_hfp_send_command()` and passes the command "ATA" or "AT+CHUP" to accept or reject the incoming call. For more details, refer to the `bt_hfp.h` and HFP specification version 1.6.

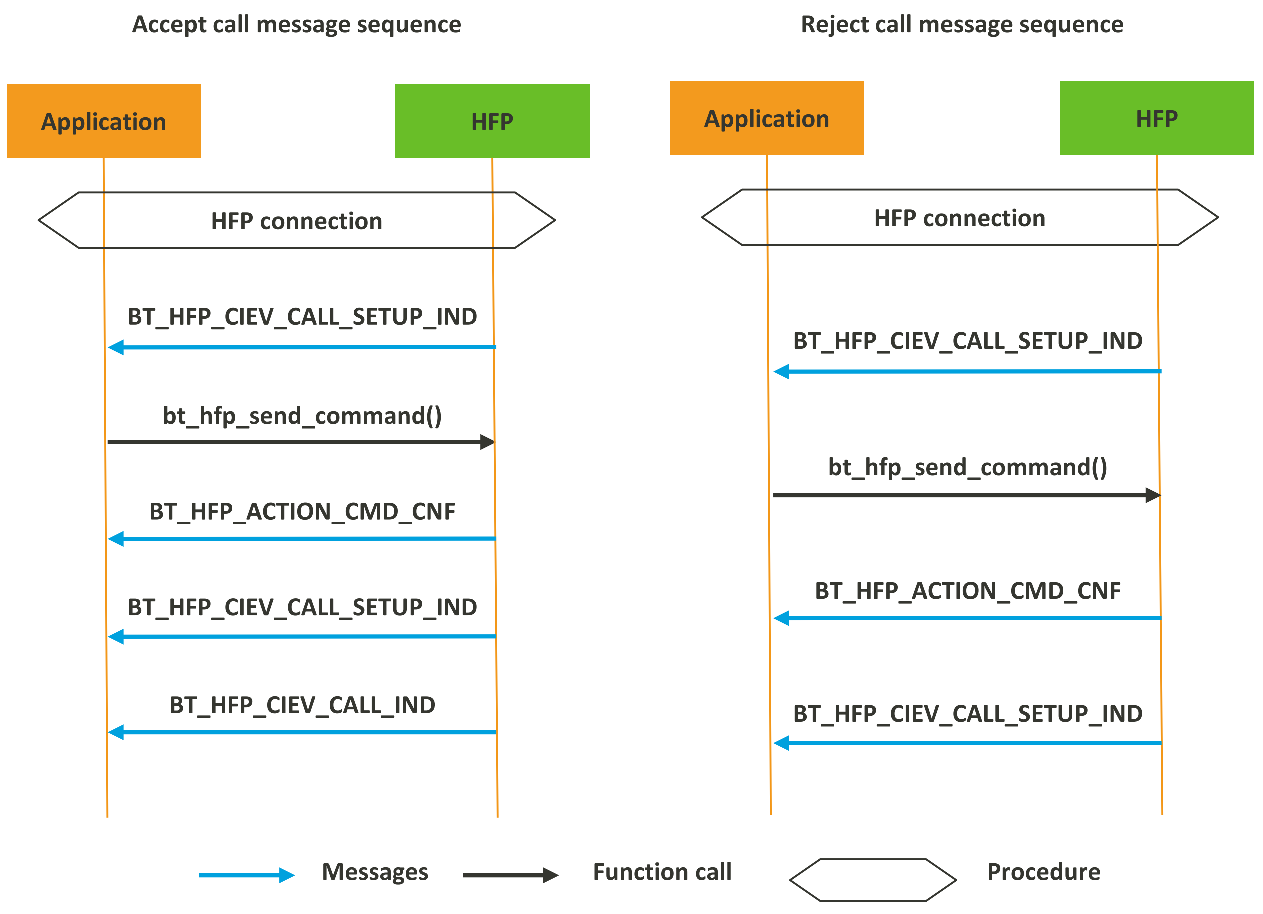

Figure 24. The actions of an incoming call message sequence

3.3.1.8. Three way call handling

Apply this process to hold the call, as shown in Figure 25. The application calls `bt_hfp_send_command()` and passes the command "AT+CHLD=0" or "AT+CHLD=1" or any other command to make the call hold actions. For more details, refer to the `bt_hfp.h` and HFP specification version 1.6.

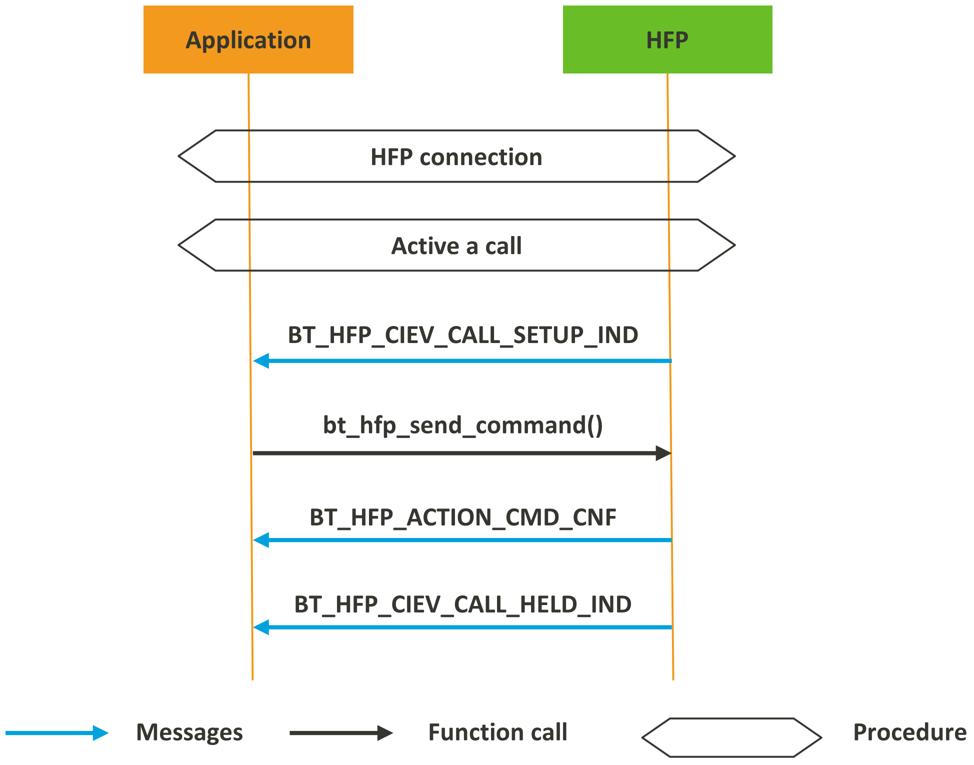

Figure 25. The three way call handling message sequence

3.3.1.9. Remote speaker volume control

Apply this procedure to synchronize the speaker volume on a remote device, as shown in Figure 26. There are two different message sequences; either application driven or remote device driven. The application driven message sequence calls `bt_hfp_send_command()` and passes the command “AT+VGS=XX” (XX is the volume value to set) to synchronize the speaker volume. For more details, refer to the `bt_hfp.h` and HFP specification version 1.6.

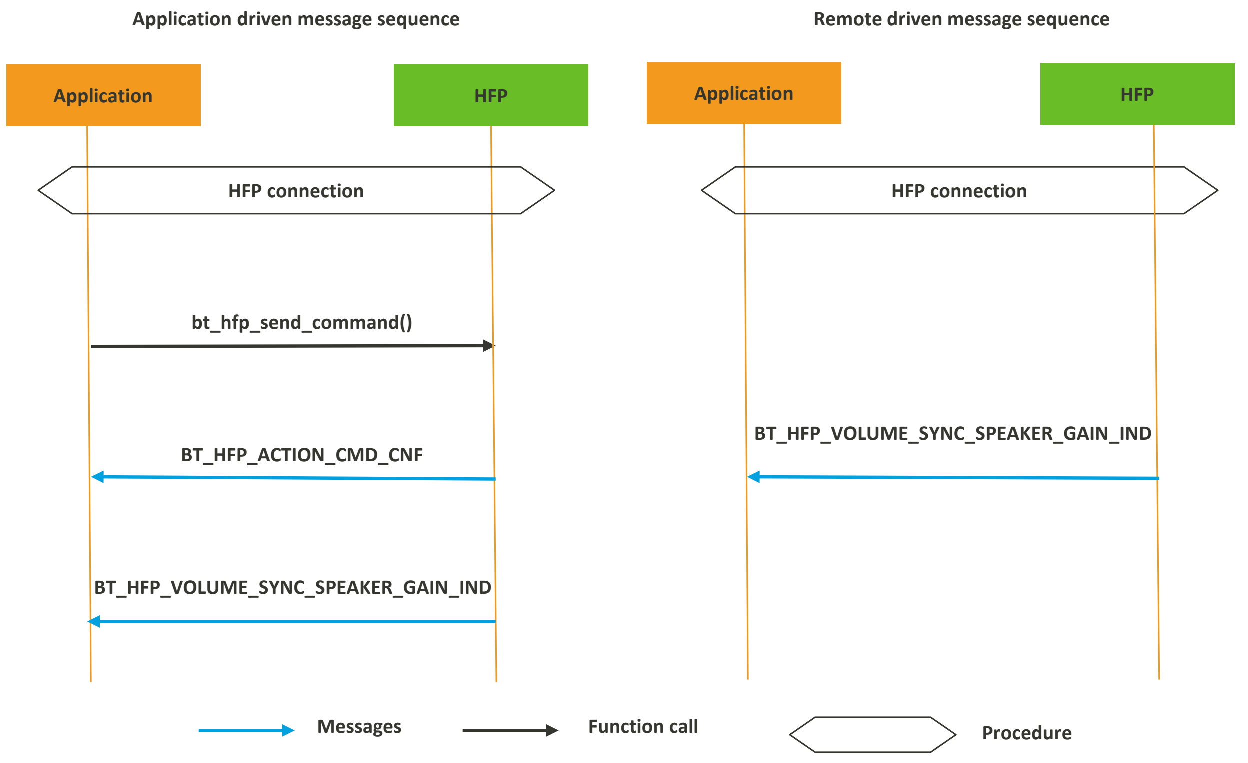
Application driven message sequence
Application
HFP
HFP connection
bt_hfp_send_command()
BT_HFP_ACTION_CMD_CNF
BT_HFP_VOLUME_SYNC_SPEAKER_GAIN_IND
Remote driven message sequence
Application
HFP
HFP connection
BT_HFP_VOLUME_SYNC_SPEAKER_GAIN_IND
Messages
Function call
Procedure

Figure 26. The remote speaker volume control message sequence

3.3.1.10. Remote microphone volume control

Apply this process to synchronize the microphone volume with a remote device, as shown in Figure 27. There are two different message sequences; either application driven or remote device driven. The application driven message sequence calls `bt_hfp_send_command()` and passes the command "AT+VGM=XX" (XX is the volume value to set) to synchronize the microphone volume. For more details, refer to the `bt_hfp.h` and HFP specification version 1.6.

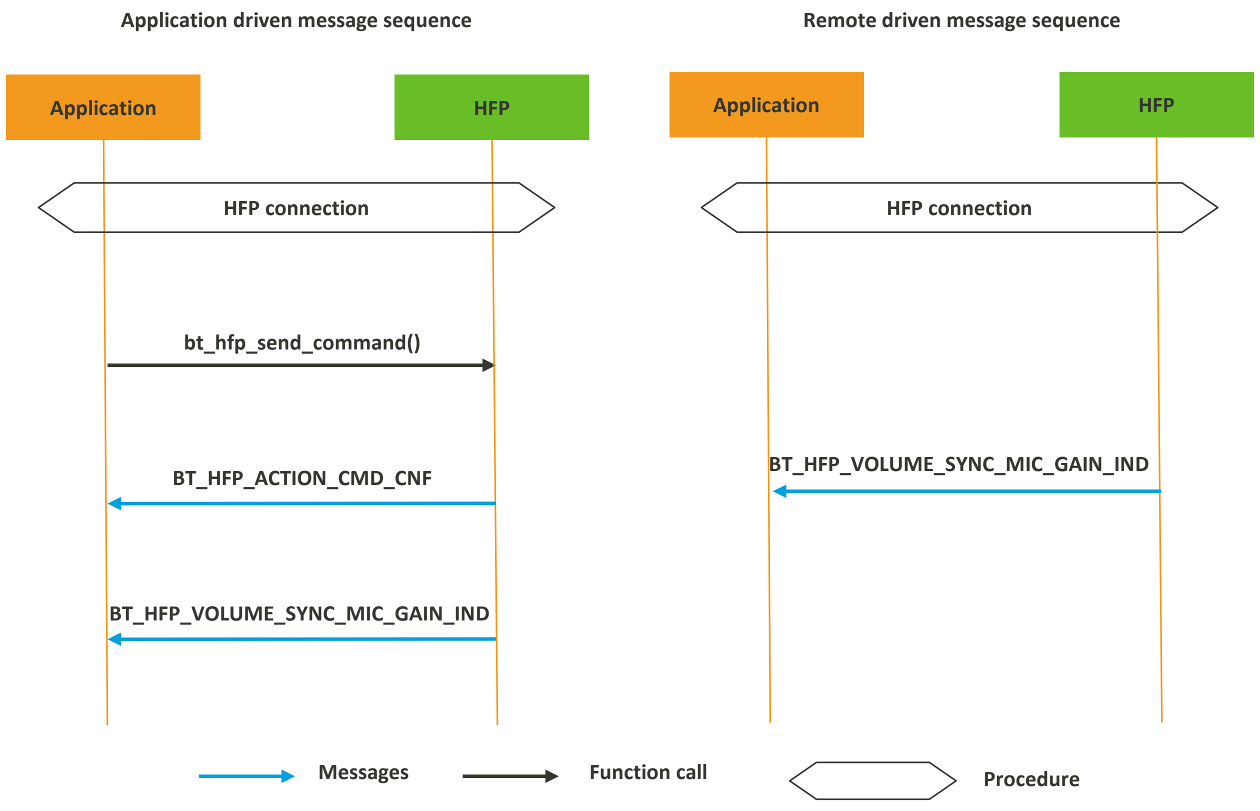

Figure 27. The remote microphone volume control message sequence

3.3.1.11. Transmit DTMF codes

Apply this process to transmit DTMF codes through AG using the HDK, as shown in Figure 28. Application uses the API `bt_hfp_send_command()` and passes the command "AT+VTS=X" (X is the DTMF code to send) to send the DTMF code. For more details, refer to the `bt_hfp.h` and HFP specification version 1.6.

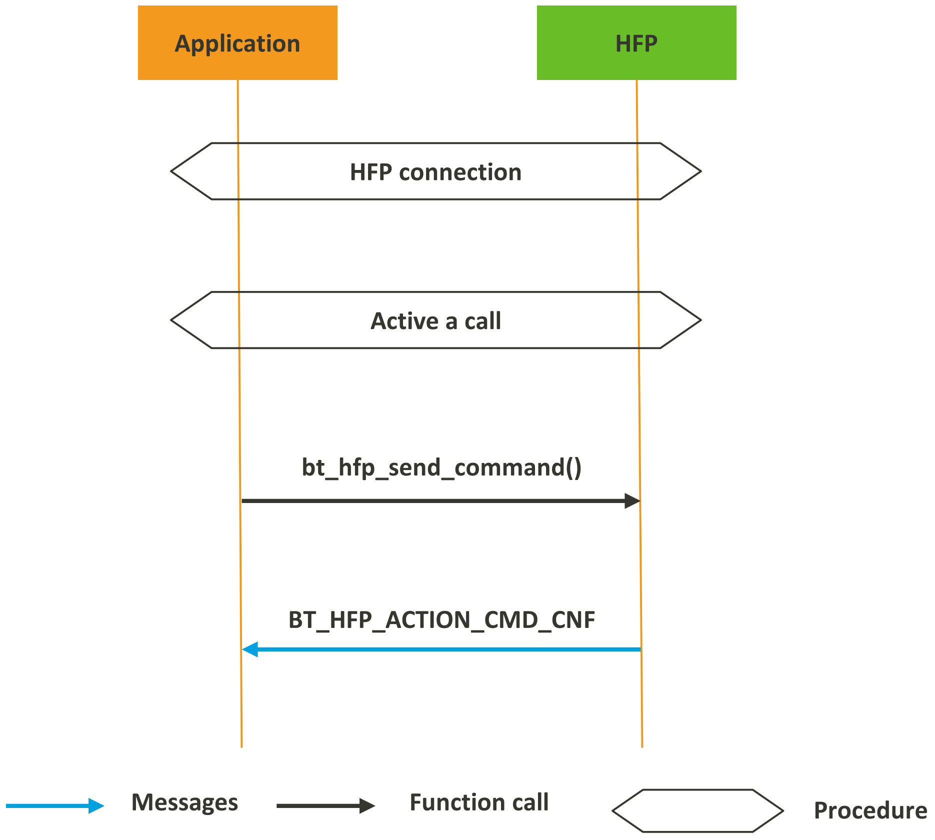

Figure 28. Transmit DTMF codes message sequence

3.3.1.12. Query a list of current calls

Apply this process to query the calls' detailed information, as shown in Figure 29. Application uses the API `bt_hfp_send_command()` and passes the command "AT+CLCC" to fetch current calls' detailed information. For more details, refer to the `bt_hfp.h` andHFP specification version 1.6.

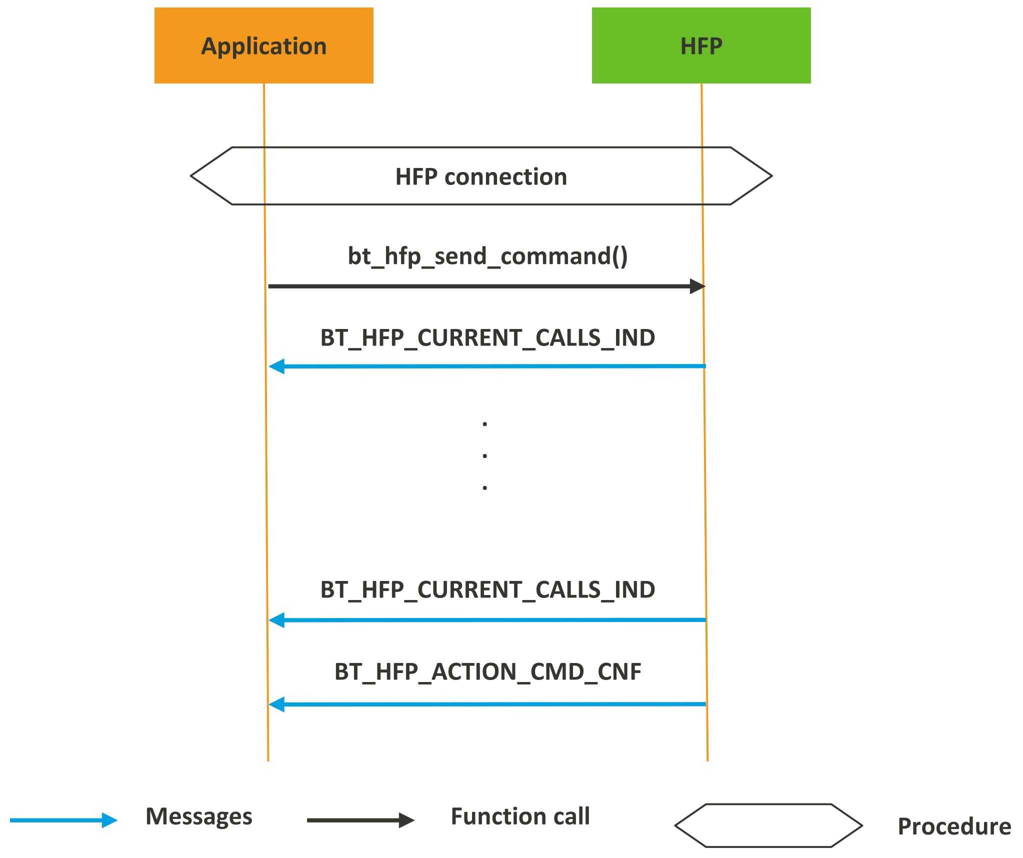

Figure 29. Query a list of current calls message sequence

3.3.2. Using the HFP APIs

This section describes how to use the HFP APIs for an application development. The source code of the HFP API is implemented in a binary library code but the headers can be found in `bt_hfp.h`.

To use the HFP APIs:

1) Implement the API `bt_hfp_get_init_params()`. This function is called when the HFP channel is initialized and the HFP will create SLC connection with a remote device according to the initialization parameters. If this function is not implemented or the API's return value is not BT_STATUS_SUCCESS, the HFP channel cannot be created.

```
bt_status_t bt_hfp_get_init_params(bt_hfp_init_params_t *param);
```

2) Implement the API `bt_app_event_callback()`. This function is called to provide notifications to the application. If a remote device sent a connection request to the HDK, this function will be called with event ID BT_HFP_CONNECT_REQUEST_IND, and the application should call `bt_hfp_connect_response()` API to accept or reject the connection.

```
void bt_app_event_callback(bt_event_t event_id, const void *param);
```

- Call `bt_hfp_connect()` to connect to a remote device, the return value is the connection result:
  - BT_STATUS_SUCCESS, the command is successfully sent.
  - BT_STATUS_FAIL, the connection request has failed.

- BT_STATUS_OUT_OF_MEMORY, the connection request has failed as there is not enough memory to complete the operation. To solve the out of memory issue, refer to section 5.1.2, "Out of Memory (OOM) ".
- If the connection is complete, the application will receive BT_HFP_SLC_CONNECTED_IND with the connection result.

To send the HFP command to a remote device:

3) Connect the HFP to a remote device and wait until connection is established, get the event ID BT_HFP_SLC_CONNECTED_IND.
4) Use the API `bt_hfp_send_command()` to send an HFP command to a remote device. The return value is the result of sending a command.
   a) BT_STATUS_SUCCESS, the command is successfully sent.
   b) BT_STATUS_FAIL, sending a command has failed.
   c) BT_STATUS_OUT_OF_MEMORY, sending a command has failed, as there is not enough memory to complete the operation. To solve the out of memory issue, refer to section 5.1.2, "Out of Memory (OOM) ".
5) Application gets the event ID BT_HFP_ACTION_CMD_CNF to get the response from a remote device.
6) Call `bt_hfp_disconnect()` to disconnect from a remote device and then get the event BT_HFP_DISCONNECT_IND with the disconnect result. If the remote device initiates the disconnection, the application will directly get the event BT_HFP_DISCONNECT_IND result.

3.4. A2DP

The A2DP provides protocols and procedures to implement distribution of high-quality audio content on ACL channels. For more information about this profile, refer to the A2DP specification version 1.3.

According to the A2DP specification, there are two roles (Sink and Source) defined for smart devices. The Source (SRC) acts as a digital audio streaming source that is delivered to the Sink (SNK) of the piconet. The SNK acts as a sink of a digital audio streaming delivered from the SRC on the same piconet.

Note: Only SNK is supported on the Airoha IoT SDK v4.

Whichever role the device is configured for, there are certain states to operate on, as described in the state machine diagram shown in Figure 30.

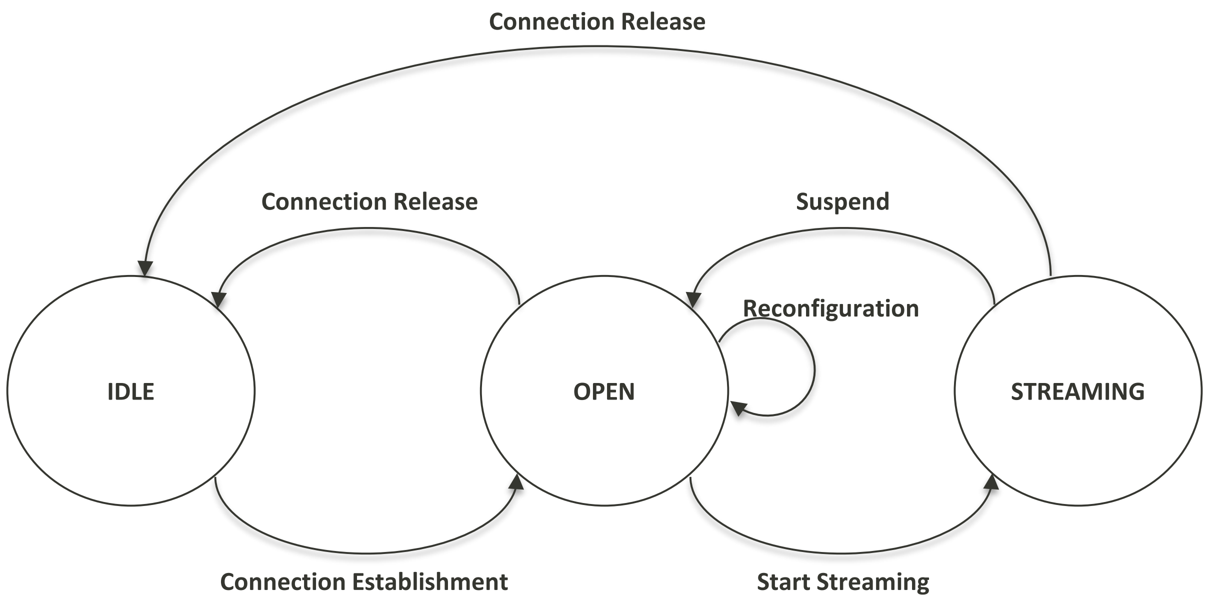

Figure 30. The A2DP state diagram

- **IDLE** — the initial state where the A2DP is not connected with or A2DP is disconnected from a remote device.
- **OPEN** — the A2DP is connected with a remote device, but both devices are not ready for streaming.
- **STREAMING** — both devices are ready for streaming.

The SDK supports two types of A2DP audio codecs, SBC and AAC. If Bluetooth is enabled and powered on, the application is requested to configure the codec information with `bt_a2dp_get_init_params()`.

The A2DP stack architecture layout is shown in Figure 31.

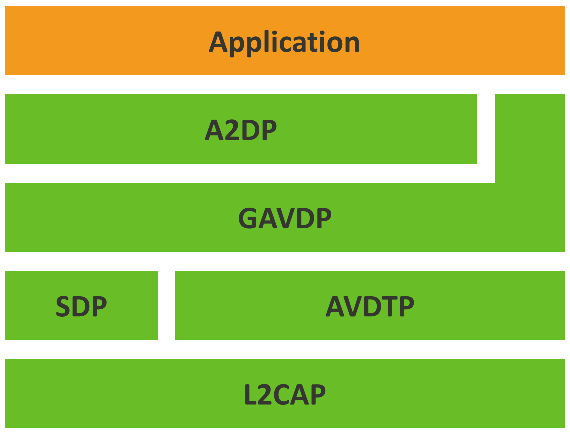

Figure 31. The A2DP abstraction layout

The A2DP depends on the Generic Audio/Video Distribution Profile (GAVDP), which defines procedures required to set up an audio/video streaming. Compared to GAVDP, there are fewer procedures involved in the application, and messages are simplified.

3.4.1. The A2DP message sequences

There are five operations available for the A2DP; connection establishment, connection release, start streaming, suspend steaming and reconfiguration. The message sequence for each procedure is described in the following sections.

3.4.1.1. Connection establishment

Apply this process to establish a streaming connection with another device. The initial state of both devices is **IDLE**, and it becomes **Open** after connection is established, as shown in Figure 32. The SDK provides two different message sequences for application from a different initiator that initiates a signaling procedure. For more details, refer to the `bt_a2dp.h`.

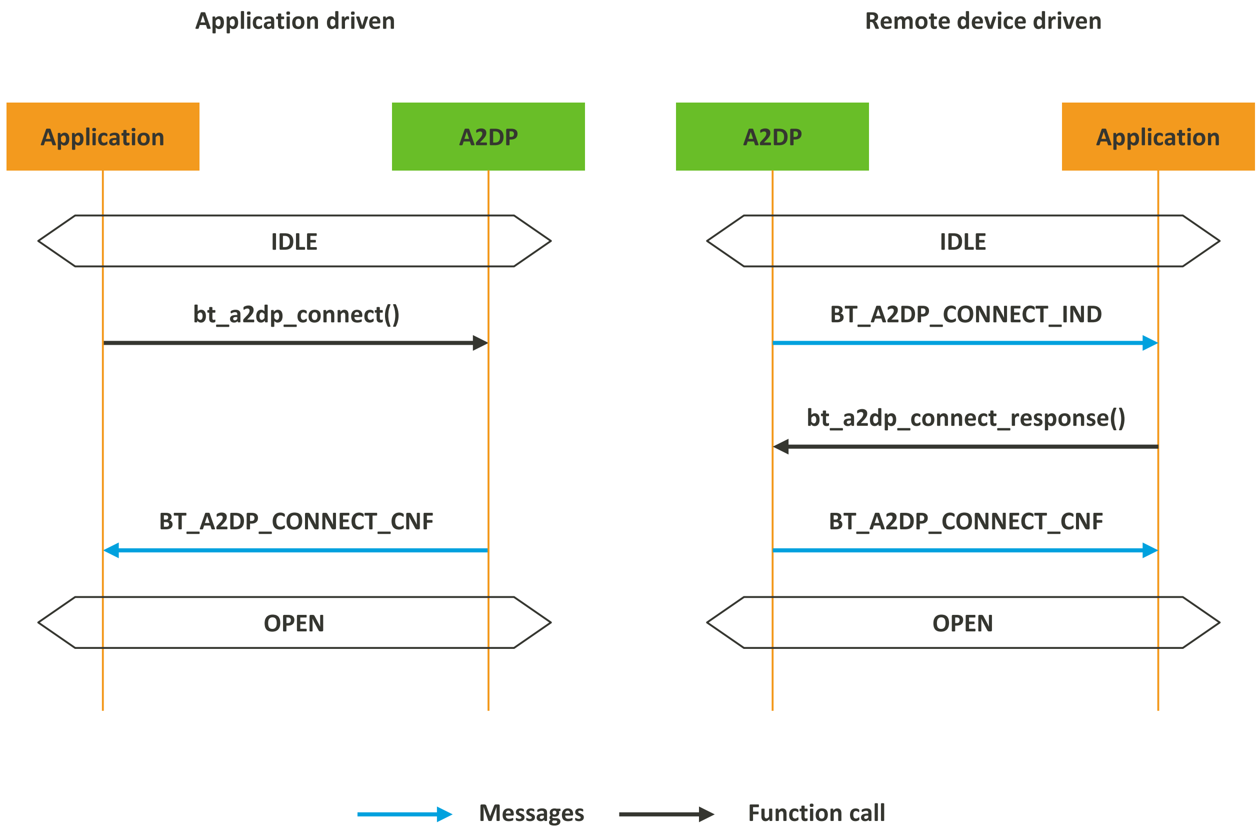

Figure 32. The A2DP connection establishment message sequence

3.4.1.2. Connection release

The A2DP connection is disconnected if a connection is released. It could be applied to **OPEN** and **STREAMING** states. The current state is set to **IDLE** for both devices, as shown in Figure 33. The HDK can initiate A2DP disconnection and accept the remote device's request to release the connection. For more details, refer to the `bt_a2dp.h`.

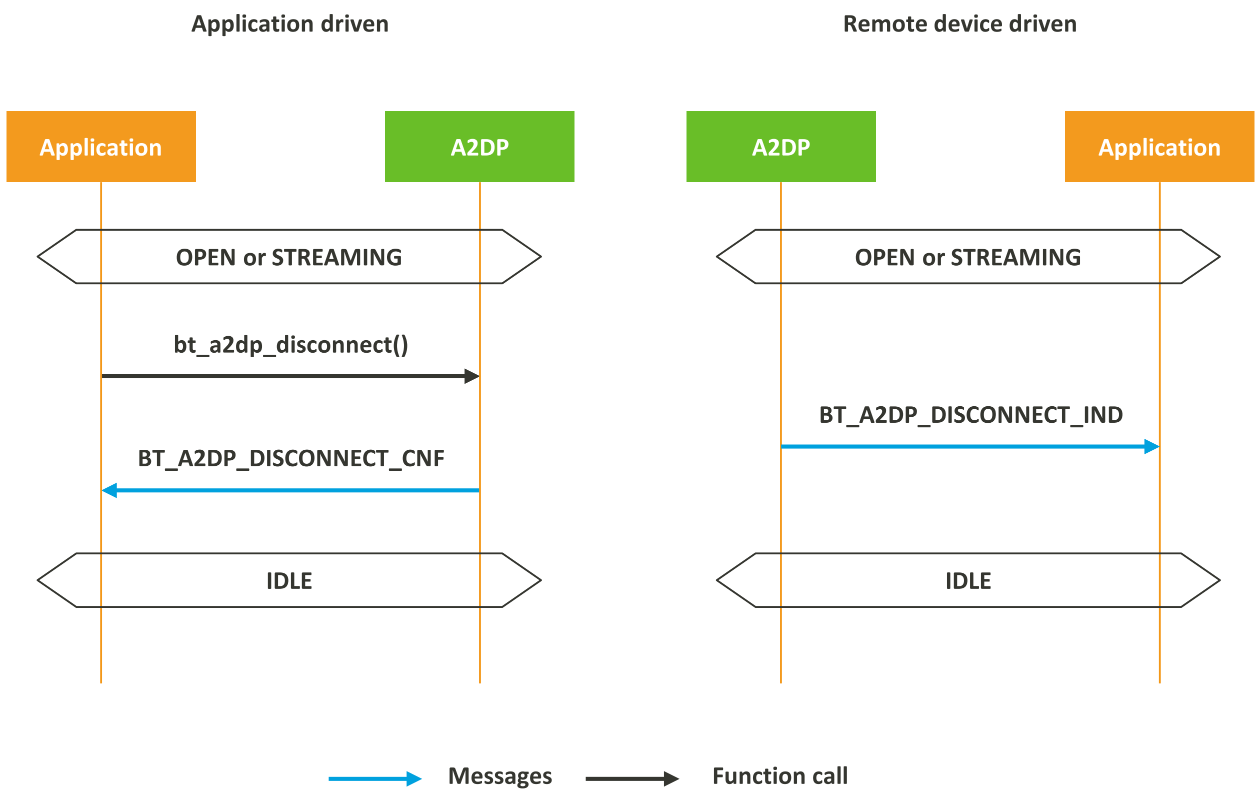

Figure 33. The A2DP connection release message sequence

3.4.1.3. Start streaming

To start or resume the audio streaming, initiate start streaming to change the state from **OPEN** to **STREAMING**, as shown in Figure 34. The device plays both acceptor and initiator role to accept the start streaming request and initiate the streaming. For more details, refer to the `bt_a2dp.h`.

In the **STREAMING** state, the application interacts with audio module to open the codec configured previously and send streaming data to audio or send SBC encoded data to the A2DP. For more details, see section 3.4.2, “Interaction with audio”.

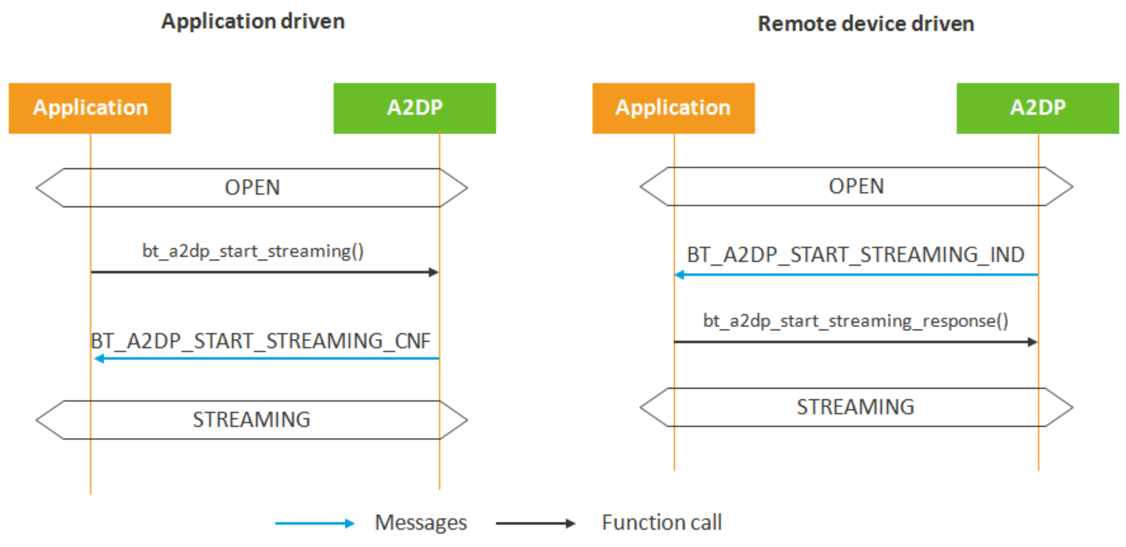
Application driven
Remote device driven
Application
A2DP
OPEN
bt_a2dp_start_streaming()
BT_A2DP_START_STREAMING_CNF
STREAMING
BT_A2DP_START_STREAMING_IND
bt_a2dp_start_streaming_response()
Messages
Function call

Figure 34. The A2DP start streaming message sequence

3.4.1.4. Suspend streaming

Apply this process to suspend the audio streaming and change the state from **STREAMING** to **OPEN**, as shown in Figure 35. In the meantime, the application notifies the audio module to close the codec and pause the audio so that the audio resource is released. For more details, refer to the `bt_a2dp.h`.

The SRC initiates streaming suspend by a user initiated action or an internal event.

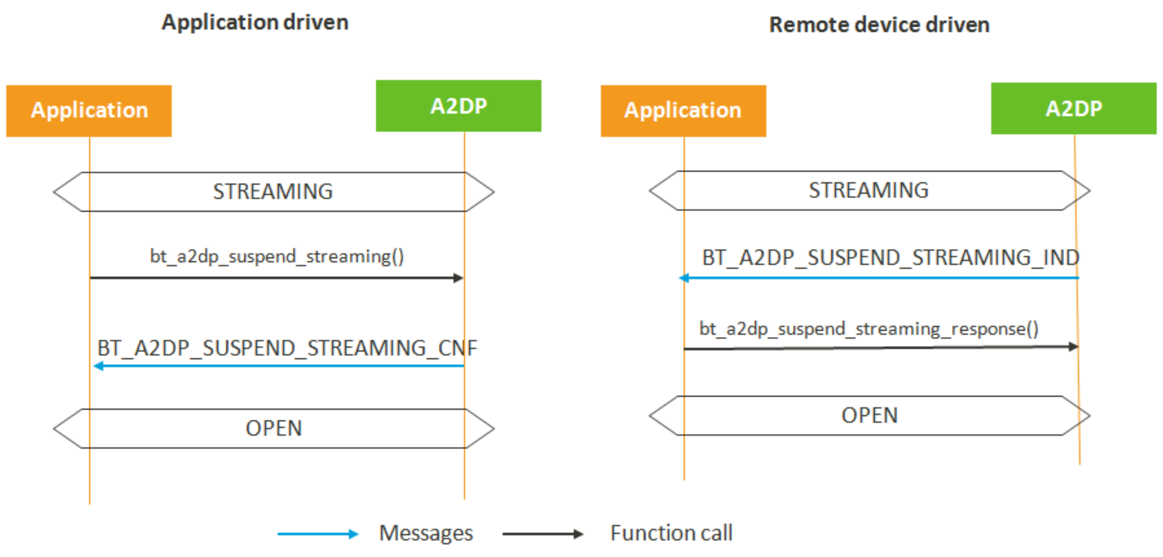
Application driven
Remote device driven
Application
A2DP
STREAMING
bt_a2dp_suspend_streaming()
BT_A2DP_SUSPEND_STREAMING_CNF
OPEN
BT_A2DP_SUSPEND_STREAMING_IND
bt_a2dp_suspend_streaming_response()
Messages
Function call

Figure 35. The A2DP suspend streaming message sequence

3.4.1.5. Reconfiguration

Apply this process, if the codec information is changed for a special audio streaming at SRC, before the streaming is sent to SNK. To process the reconfiguration, both devices should be in **OPEN** state. Execute suspend operation, if the state is **STREAMING** and change the state to **OPEN** for both devices, as shown in Figure 36. For more details, refer to the `bt_a2dp.h`.

At the end of this process, the device's state remains **OPEN**. Initiate Start streaming procedure to resume the streaming see section 3.4.2, "Interaction with audio".

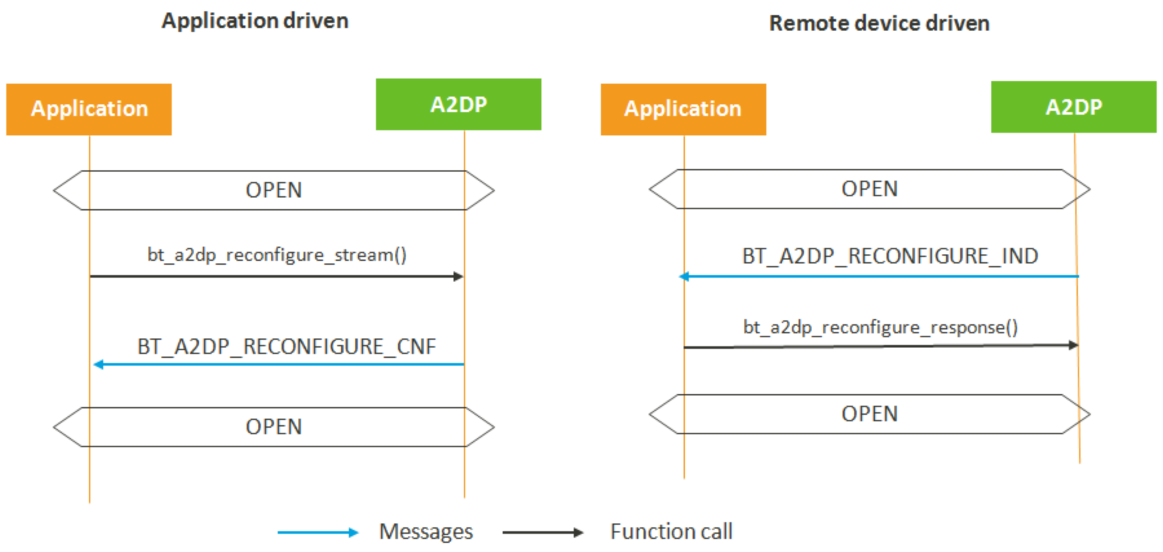

Figure 36. The A2DP reconfiguration message sequence

You've now successfully processed the A2DP message sequences.

3.4.2. Interaction with audio

To establish A2DP connection, both SRC and SNK devices need to negotiate and configure service parameters, such as the media transport, and the A2DP audio codec. The media codec enables to encode or decode digital data stream using Bluetooth protocol. The received data is decoded and passed through digital-to-analog converter (DAC).

Once the A2DP connection is established and Start streaming procedure is executed, the SRC and SNK switch to **STREAMING** state. At the SRC, the audio module encodes the raw audio data with a codec. This data is then passed to the SNK using A2DP over Bluetooth. At the SNK, the received data passes through the audio module and the raw data is decoded using the same codec. The A2DP audio codec must be opened before media data is streaming, and be closed after suspended. There are four main operations during the interaction between A2DP streaming and the audio codec; open codec, play, get data, close codec. For more details, refer to the `bt_codec.h`.

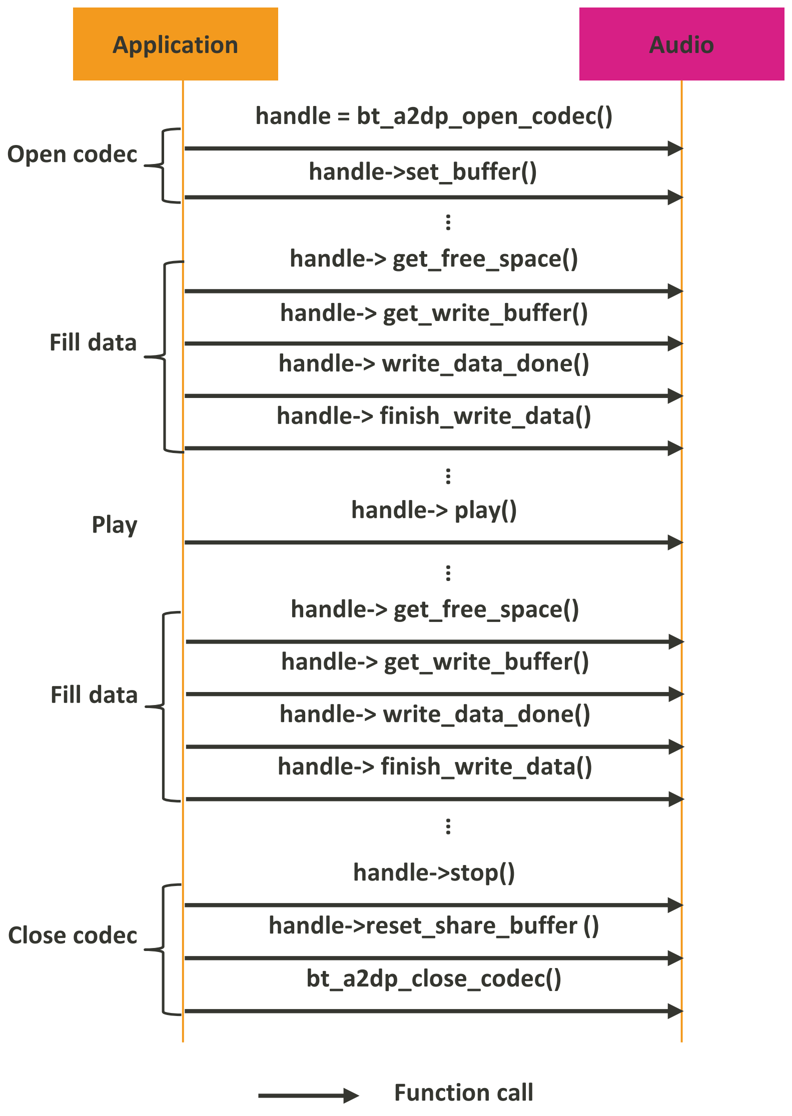
Application
Audio
Open codec
handle = bt_a2dp_open_codec()
handle->set_buffer()
Fill data
handle-> get_free_space()
handle-> get_write_buffer()
handle-> write_data_done()
handle-> finish_write_data()
Play
handle-> play()
Fill data
handle-> get_free_space()
handle-> get_write_buffer()
handle-> write_data_done()
handle-> finish_write_data()
Close codec
handle->stop()
handle->reset_share_buffer ()
bt_a2dp_close_codec()
Function call

Figure 37. The A2DP codec operation message sequence (Sink)

3.4.2.1. Open codec

Bluetooth parameters, such as callback function and codec related parameters must be sent to configure the codec properties during open codec operation. If requested capability is available, a handle is returned and functions to share the buffer are provided. The Get data function sets the A2DP codec driver after opening the codec.

For A2DP sink operation, Bluetooth data is held before it is consumed. Codec driver fetches Bluetooth data with get data function for decoding (see Figure 37).

For A2DP source operation, the audio data is encoded with encoder, wrapped with Bluetooth header and then sent to the remote device (see Figure 38).

3.4.2.2. Play and Get data

For A2DP sink, it’s recommended to keep enough Bluetooth data before playing the audio. The application will provide a callback function to fetch the data. To start the Bluetooth audio, call the play function (see Figure 37).

For A2DP source, application will be notified to get audio data after codec play is called, and the function `get_payload_done()` should be called after application gets and sends audio data.

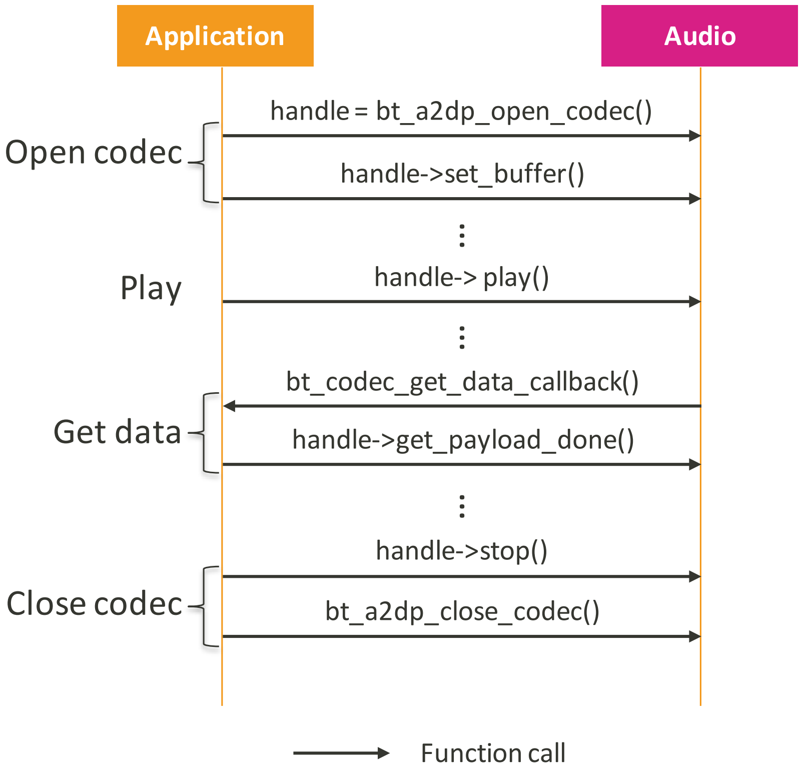

Figure 38. The A2DP codec operation message sequence (Source)

3.4.2.3. Stop and Close the codec

Apply this operation if the audio is close to an end or an error occurred during playback. If the Bluetooth audio link is no longer needed, the codec can be closed (see Figure 37).

3.4.3. Using the A2DP APIs

This section describes how to use the A2DP APIs in your application. The functionality of the A2DP APIs is implemented in the library code but header functions can be found in `bt_a2dp.h`.

1) Implement `bt_a2dp_get_init_params()` to configure the codec information and role when the Bluetooth is powered on by calling `bt_gap_power_on()`.

```
bt_a2dp_codec_capability_t cap[2]; // Support 2 codec.
int32_t bt_a2dp_get_init_params(bt_a2dp_init_params_t* params) {
(A2DP SINK)
    params->codec_number = 2;
    // Fill supported audio codec capability in init_codec, and then pass
to params-> codec_list.
    params->codec_list = &init_codec;
(A2DP SOURCE)
    params->codec_number = 1;
    // Fill supported audio codec capability in init_codec, and then pass
to params-> codec_list.
    params->codec_list = &init_codec;

    return Bluetooth_STATUS_A2DP_OK;
    }
```

2) The application connects to a remote device by calling the `bt_a2dp_connect()` function, and then the event BT_A2DP_CONNECT_CNF is expected to be received, as shown in `bt_app_event_callback(bt_msg_type_t msg_id, bt_status_t status, void *buff)`.

```
(A2DP SINK)
ret = bt_a2dp_connect(&a2dp_handle, address, BT_A2DP_SINK);
(A2DP SOURCE)
ret = bt_a2dp_connect(&a2dp_handle, address, BT_A2DP_SOURCE);
```

3) The HDK disconnects from a remote device using the following API and then the event BT_A2DP_DISCONNECT_CNF is expected to be received, as shown in `bt_app_event_callback (bt_msg_type_t msg_id, bt_status_t status, void *buff)`.

```
bt_a2dp_disconnect(a2dp_handle);
```

4) (A2DP SINK and A2DP Media Payload)

- A2DP SINK

5) Implement a utility function to hold the media data node and fetch data callback. If media data is received from the A2DP SDK layer when playing music, the data node is needed to keep in a buffer and locate the valid media data. Fetch data callback is called to fill the media data from data node in the buffer when the codec driver starts decoding. Remember to release the data node if the media data is fully copied.

    a) Hold the media data node.

```
#define MEDIA_PKT_HEADER_LEN 12 // Media packet header 12 Bytes.
typedef struct _bt_media_packet_node{
    _bt_media_packet_node *node;
    uint16_t        packet_length;
    uint16_t        offset;
} bt_media_packet_node_t;
void bt_a2dp_keep_media_data_node(bt_media_packet_node_t *data_node,
int16_t offset, int16_t total_len)
{
    void* p_payload = data_node + offset + MEDIA_PKT_HEADER_LEN;
    *payload_len = total_len – offset - MEDIA_PKT_HEADER_LEN;
    // Search the payload start pointer for AAC.
    if (g_a2dp_codec.type == BT_A2DP_CODEC_AAC) {
        uint8_t payload_offset = 9;
        uint8_t media_start_pos = 0;
        do {
            media_start_pos = ((uint8_t *) p_payload)[payload_offset];
            payload_offset++;
        } while (media_start_pos  == 0xFF);
        *payload_len -= payload_offset;
        p_payload += payload_offset;
    }
    data_node->offset = p_payload – (void *)data_node;
    data_node->packet_length = total_len;
    // Insert the data_node at the end of data node buffer list.
    bt_a2dp_hold_media_data_node(data_node);
}
```

    b) Implement fetch data callback.

Fetch the data node from the data buffer list to `data_node` and copy the media data to `dsp_buff`. If the media data length in data node is less than `dsp_buff` length, then

        i) Remove the data node from the list and release it;

ii) Fetch the data node and copy again, till the buffer is empty or the `dsp_buffer` is full.

If the media length is larger than `dsp_buff` length, then

i) Do not remove `data_node` before it's filled with received data.

ii) Modify the remaining media offset only.

```
int32_t bt_sink_codec_get_data_callback(volatile uint16_t *dsp_buff,
uint32_t len)
{
    bt_media_packet_node_t *data_node = NULL;
    int16_t media_len;
    uint32_t filled_length;
    // Fetch the bt media packet node and fill the received data in the
node to dsp_buffer.
    // Note: The data should be filled in units of Word.
    return filled_length;
}
```

- A2DP SOURCE

6) Implement `bt_app_event_callback()` in your application.

```
bt_status_t bt_app_event_callback (bt_msg_type_t msg_id, bt_status_t
status, void *buff)
{
    switch (msg_id) {
        case BT_A2DP_START_STREAMING_IND: {
                // Open a2dp codec according to the specified codec, and
then set the fetch data function callback for codec driver.
                g_media_handle = bt_codec_a2dp_open(bt_audio_callback,
&a2dp_codec);
                g_media_handle->
set_get_data_function(handle,bt_sink_codec_get_data_callback);
                bt_a2dp_start_streaming_response(params->conn_id, true);
            }
            break;
        case BT_A2DP_SUSPEND_STREAMING_IND: {
                // Stop music, close codec.
                g_media_handle->stop(g_media_handle);
                bt_a2dp_close_codec(g_media_handle);
                bt_a2dp_suspend_streaming_response(params->conn_id, true);
            }
            break;
        case BT_A2DP_STREAMING_RECEIVED_IND: {
                bt_a2dp_keep_media_data_node(params->data_node, params->
media_offset, params->total_length);
                // Call g_media_handle->play(); if this is the streaming
data received first time after BT_A2DP_START_STREAMING_IND.
            }
            break;
      }
}
```

3.5. AVRCP

The Audio/Video Remote Control Profile (AVRCP) defines the features and procedures to ensure interoperability between Bluetooth devices with audio/video control functions. For more information about this profile, refer to the AVRCP specification version 1.6.1.

There are two roles configured in AVRCP: controller (CT) and target (TG). The CT transmits AV/C (the AV/C Digital Interface Command Set) command to a remote Bluetooth device (TG). The CT must initiate a transaction by sending a command frame to a TG. The TG receives the command frame and generates a corresponding response frame.

AVRCP does not manage A/V streaming. Refer to the A2DP profile for information about managing streaming. Please note that smartphones may not support the AVRCP connection without A2DP. We strongly recommend connecting both profiles.

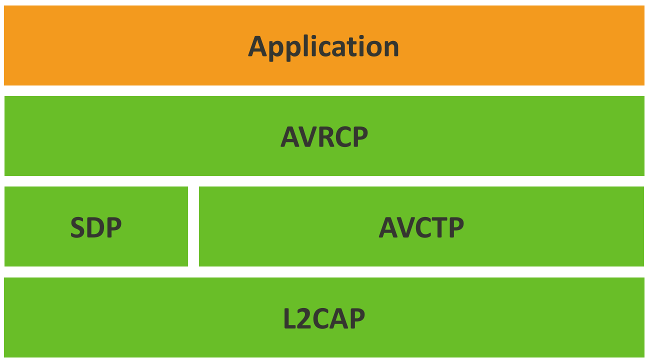

Figure 39. The AVRCP abstraction layout

The packets sent between the CT and TG are transported by the Audio/Video Control Transport Protocol (AVCTP), as shown in Figure 39.

3.5.1. The AVRCP procedure and message sequences

3.5.1.1. Connection establishment

AVRCP establishes an L2CAP connection for the AVCTP control channel or the AVCTP browsing channel that can be initiated by CT or TG. At the same time, there must be only one AVCTP control channel and one AVCTP browsing channel per ACL link. The new connections are rejected if the channel already exists. The AVCTP control channel is always established before the AVCTP browsing channel. This process is shown in Figure 40.

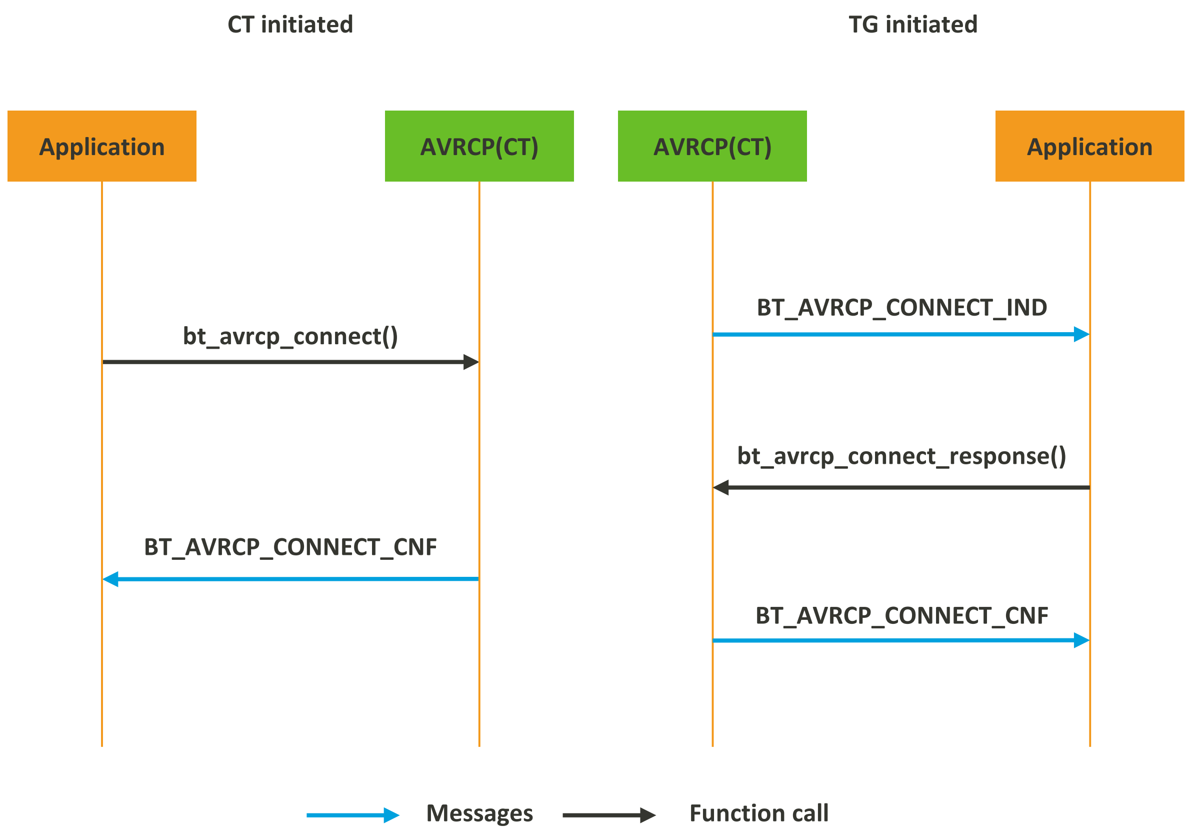

Figure 40. AVRCP connection establishment

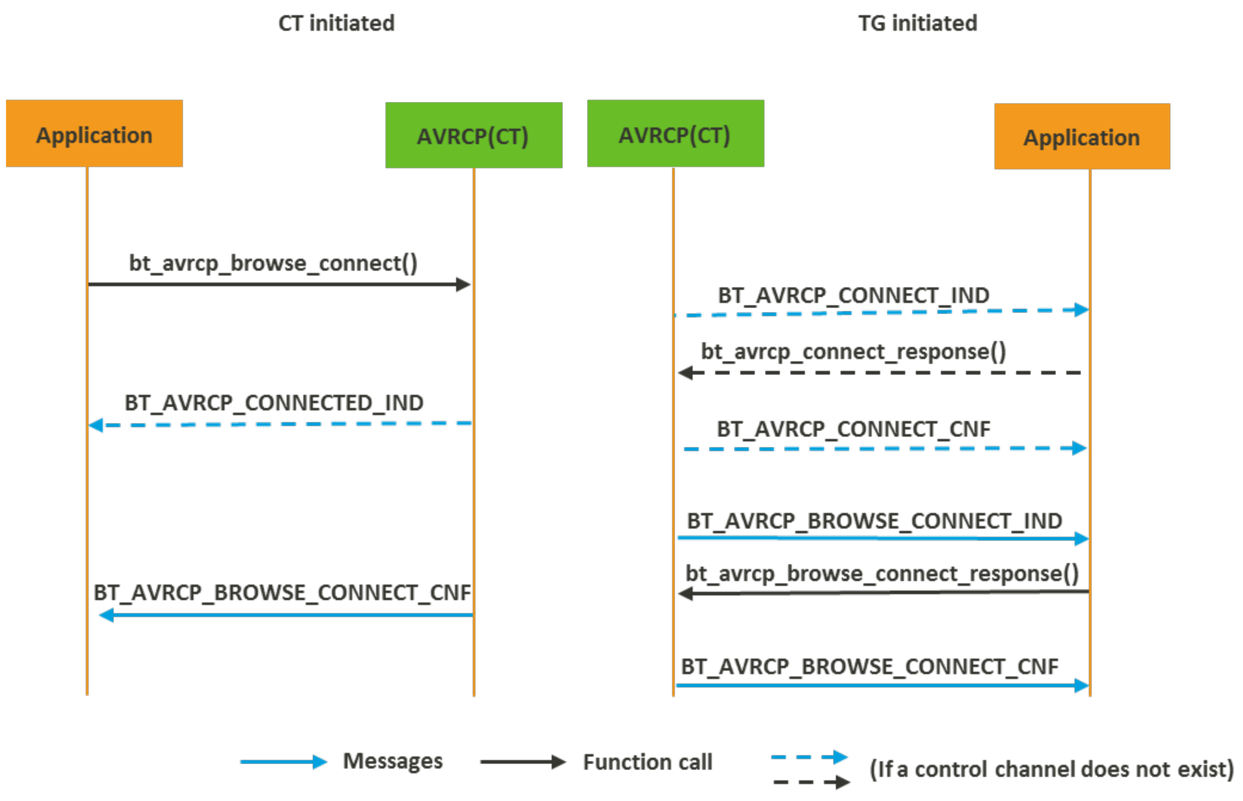
CT initiated
TG initiated
Application
AVRCP(CT)
AVRCP(CT)
Application
bt_avrcp_browse_connect()
BT_AVRCP_CONNECTED_IND
BT_AVRCP_BROWSE_CONNECT_CNF
BT_AVRCP_CONNECT_IND
bt_avrcp_connect_response()
BT_AVRCP_CONNECT_CNF
BT_AVRCP_BROWSE_CONNECT_IND
bt_avrcp_browse_connect_response()
BT_AVRCP_BROWSE_CONNECT_CNF
Messages
Function call
(If a control channel does not exist)

Figure 41. AVRCP browsing connection establishment

3.5.1.2. Connection release

The CT or the TG can both initiate the process to release AVCTP connection. The process is shown in Figure 42.

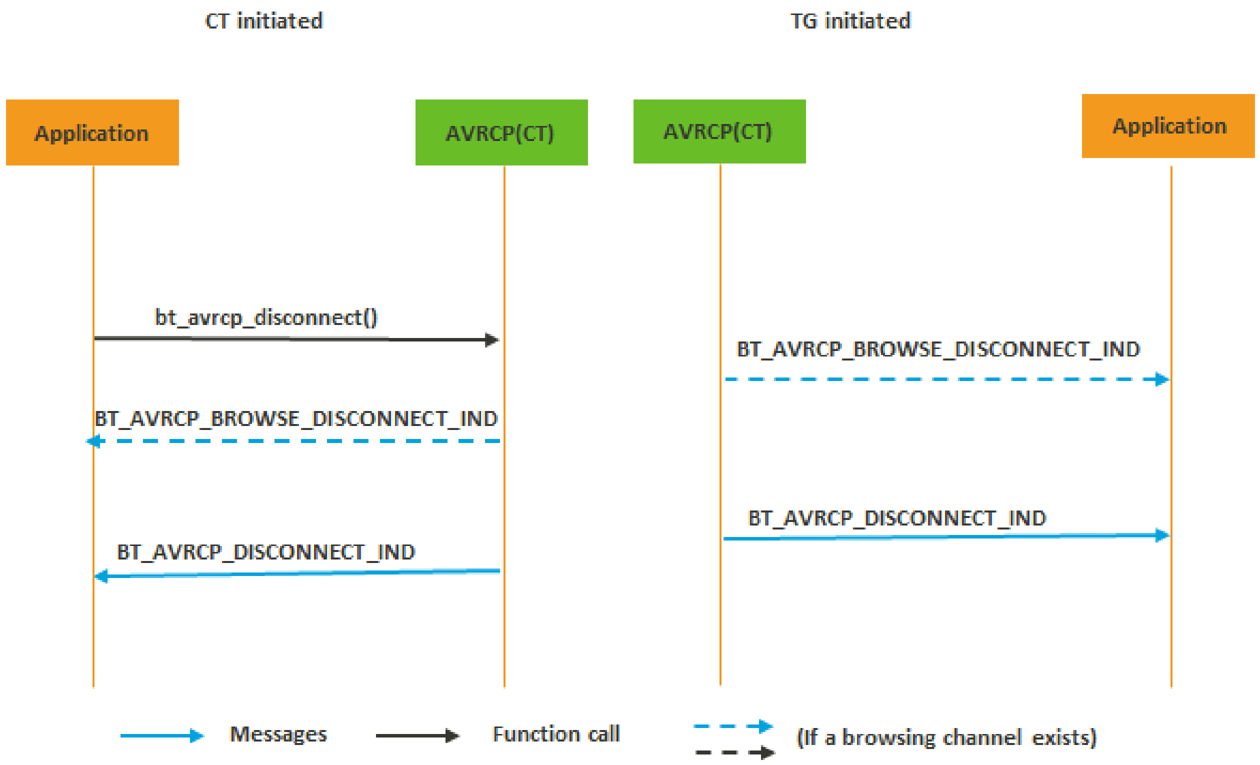

Figure 42. AVRCP connection release

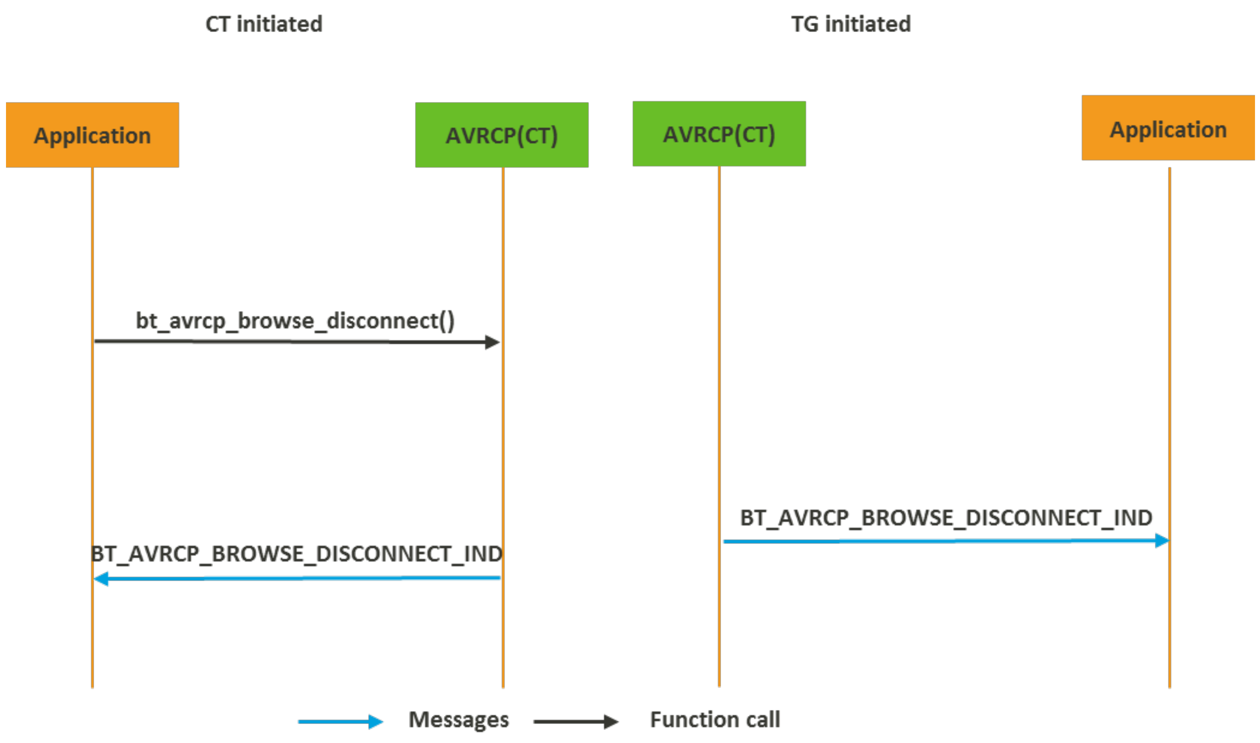
CT initiated
TG initiated
Application
AVRCP(CT)
AVRCP(CT)
Application
bt_avrcp_browse_disconnect()
BT_AVRCP_BROWSE_DISCONNECT_IND
BT_AVRCP_BROWSE_DISCONNECT_IND
Messages
Function call

Figure 43. AVRCP browsing connection release

3.5.1.3. AV/C command procedure

The CT can send the AV/C command when the connection is established. The process is shown in Figure 44. AVRCP supports two types of commands: PASS THROUGH and VENDOR DEPENDENT. Each command receives a response from TG. Only VENDOR DEPENDENT commands receive an interim response. Note, the application should only send one command and wait for its response. AVRCP returns a busy status if a command is sent without waiting for the arrival of the previous command's response.

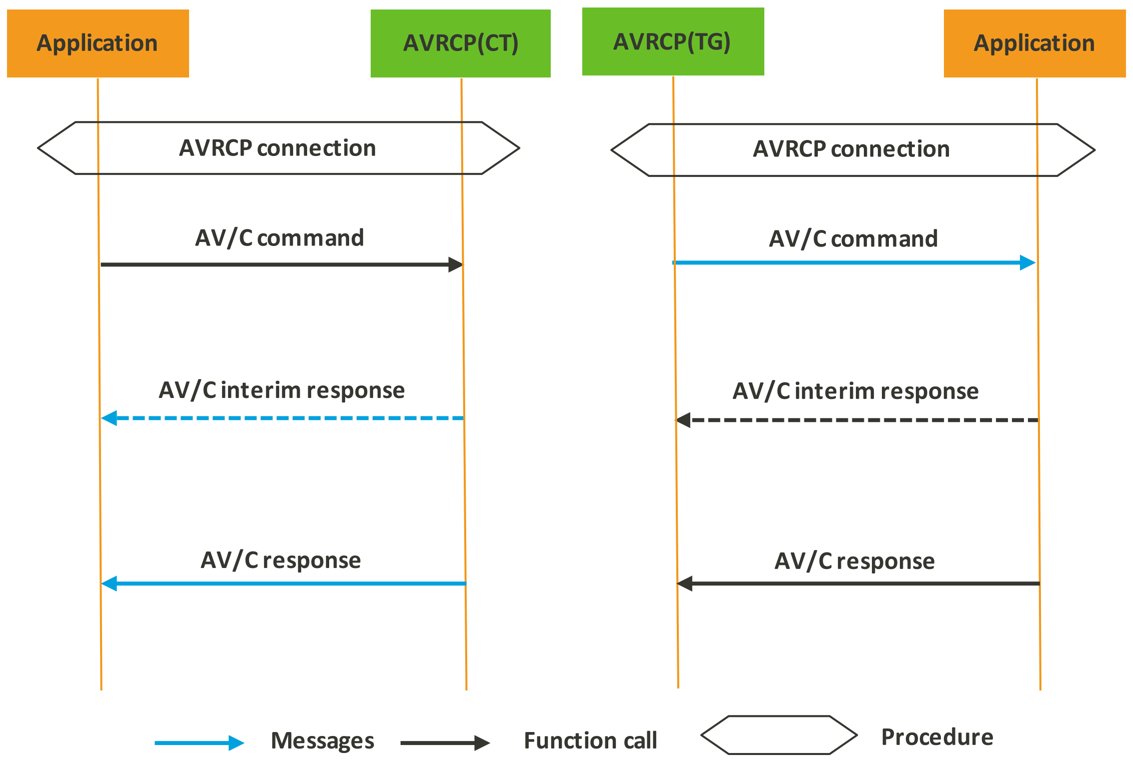

Figure 44. AV/C command procedure

3.5.1.4. Browsing command procedure

The procedure of the AVRCP Browsing commands defines the commands that are used directly over AVCTP. AV/C is not used. The AVCTP Browsing Channel does not use fragmentation. The application receives the parameter `list_mtu` of BT_AVRCP_BROWSE_CONNECT_CNF event, representing the TG sends the maximum size in parameter `size_of_folder_path` of `bt_avrcp_browse_set_browsed_player_response()`, `size_of_item_list` of `bt_avrcp_browse_get_folder_items_response()`, `size_of_attribute_list` of `bt_avrcp_browse_get_item_attributes_response()`. Note, the application should only send one browsing command and wait for its response. The AVRCP returns a busy status if a browsing command is sent without waiting for the arrival of the previous command’s response.

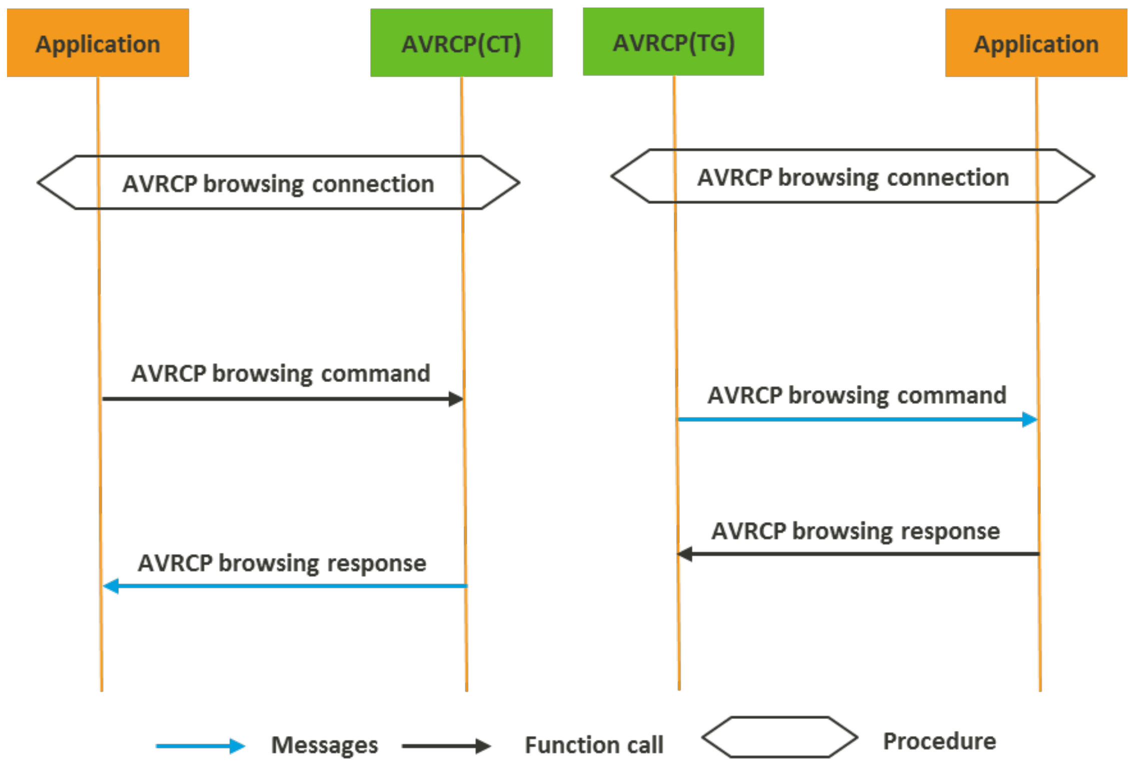

Figure 45. Browsing command procedure

3.5.2. AV/C commands

This section specifies two types of commands: PASS THROUGH and VENDOR DEPENDENT. The library `libbt_avrcp.a` provides only PASS THROUGH type of functions. You must include `libbt_avrcp_enhence.a` library in your application to use the VENDOR DEPENDENT type of commands. VENDOR DEPENDENT commands are mainly used for transferring metadata (e.g., to get the TG player application settings or metadata attributes).

3.5.2.1. PASS THROUGH

The PASS THROUGH command is mainly used to transfer a user operation from CT device to a TG device. To send a user operation from CT device, application can call `bt_avrcp_send_pass_through_command()` with operation ID and state. The operation ID specifies the given operation and the state defines the operation key status (see `bt_avrcp.h`). Since the TG may reject the PASS THROUGH command, application should check the response of the command. On TG device, the application receives the event BT_AVRCP_PASS_THROUGH_COMMAND_IND with operation ID and state from CT. Application should call `bt_avrcp_send_pass_through_response()` immediately after receiving the event, to send a response to the CT device (accepted or rejected).

Note, when sending the key, the key status can be either push or release and the TG device can respond to either one of them. We strongly recommend that the application tries to both push and release when sending the PASS THROUGH command.

3.5.2.2. Player application settings

CT provides commands for the application to query the player application attribute settings and get and set the attribute values according to the attributes on the TG. All available settings on the TG are in the `<attribute, value>` pair. Therefore, each setting has a unique attribute ID and a corresponding value.

Note, TG can accept the application attribute value setting commands sent from CT, but, depending on the implementation of TG, it might not take any action with the attribute values.

The related attributes and values are listed in Table 2. For more details, please refer to Appendix F of AVRCP specification version 1.6.1.

Table 2. Player application settings and values

Attribute ID	Attribute Description	Supported Value	Value Description
BT_AVRCP_SETTING_ATTRIBUTE_EQUALIZER	Equalizer status	0x01	OFF
		0x02	ON
BT_AVRCP_ SETTING_ATTRIBUTE _REPEAT_MODE	Repeat status	0x01	OFF
		0x02	Single track repeat
		0x03	All tracks repeat
		0x04	Group repeat
BT_AVRCP_ SETTING_ATTRIBUTE _SHUFFLE_MODE	Shuffle status	0x01	OFF
		0x02	All tracks shuffle
		0x03	Group shuffle
BT_AVRCP_ SETTING_ATTRIBUTE _SCAN_MODE	Scan status	0x01	OFF
		0x02	All tracks scan
		0x03	Group Scan

3.5.2.3. Metadata attributes

CT enables access to metadata attributes for the current media track by calling the Get Element attribute API. When TG receives a request from CT, TG responds with the attribute values.

The related media attributes are listed in Table 3. For more details, please refer to Appendix E of AVRCP specification version 1.6.1.

Table 3. Media element attributes

Attribute ID	Value	Description
BT_AVRCP_MEDIA_ATTRIBUTE_TITLE	0x01	Title
BT_AVRCP_ MEDIA_ATTRIBUTE _ARTIST_NAME	0x02	Artist Name
BT_AVRCP_ MEDIA_ATTRIBUTE _ALBUM_NAME	0x03	Album Name
BT_AVRCP_ MEDIA_ATTRIBUTE _MEDIA_NUMBER	0x04	Current media number
BT_AVRCP_ MEDIA_ATTRIBUTE _TOTAL_MEDIA_NUMBER	0x05	Total number of media
BT_AVRCP_ MEDIA_ATTRIBUTE _GENRE	0x06	Genre category
BT_AVRCP_ MEDIA_ATTRIBUTE _PLAYING_TIME	0x07	Length of the audio file in milliseconds

3.5.2.4. Absolute volume

The absolute audio volume is applied to adjust the volume (in percentage) between CT and TG devices. The application uses `bt_avrcp_set_absolute_volume()` to set TG volume from CT. Meanwhile, the application should call an event notification function to register the absolute volume change at TG device.

The TG device should use `bt_avrcp_send_set_absolute_volume_response()` after receiving the event BT_AVRCP_SET_ABSOLUTE_VOLUME_CNF. And also, the event notification response API with absolute volume change event if local volume in TG device changes.

Note, the AVRCP CT role is with A2DP sink role, while TG is with A2DP source role. However, the CT role of absolute volume is with A2DP source role, while TG is with A2DP sink role.

3.5.2.5. Event notification

CT enables the application to register for an event to detect whether the attribute value on TG has changed. CT receives an interim response from TG to get the current value of the attribute. If the value has changed, the CT receives the final response from TG for the updated value. For the CT device, the application must register for additional events (see Table 4), after the last event receives an interim response. If the application needs to listen to the same attribute repeatedly, it can register for the same event again after receiving the final response. For the TG device, the application must send an interim response with the current value for which the CT device is registered. The TG device must send the updated value to the CT device anytime the value changes.

The related event IDs are listed in Table 4. For more details, please refer to Appendix H of the AVRCP specification version 1.6.1.

Table 4. List of notification events

Event ID	Description
BT_AVRCP_EVENT_PLAYBACK_STATUS_CHANGED	Change in the playback status of the current track
BT_AVRCP_EVENT_TRACK_CHANGED	Change of current track
BT_AVRCP_EVENT_TRACK_REACHED_END	Reached end of a track
BT_AVRCP_EVENT_TRACK_REACHED_START	Reached start of a track
BT_AVRCP_EVENT_PLAYBACK_POS_CHANGED	Change in playback position. Return the playback position after a specified playback interval in seconds.
BT_AVRCP_EVENT_BATT_STATUS_CHANGED	Change in the battery status
BT_AVRCP_EVENT_SYSTEM_STATUS_CHANGED	Change in the system status
BT_AVRCP_EVENT_PLAYER_APP_SETTING_CHANGED	Change in the player application setting
BT_AVRCP_EVENT_VOLUME_CHANGED	Change the absolute volume

3.5.2.6. Continuation packets

All metadata transfers related to VENDOR DEPENDENT commands support continuation packets. That means CT can receive more than one packet for a command to avoid sending large packets. Each response for these commands has a packet type parameter, which indicates whether the packet is a continuation of a previous packet.

On the CT device, the application must continue calling `bt_avrcp_request_continuing_response()` to get more packets until the data transfer is complete, or call `bt_avrcp_abort_continuing_response()` to abort the operation.

Note, TG does not allow CT to send another metadata transfer command when it is receiving continuation packets.

3.5.2.7. Get play status

CT provides commands to get the status of the currently playing media. The TG device responds with the corresponding status when it receives a request from CT.

3.5.2.8. Get capabilities

CT provides commands to get the capabilities supported by the TG device. There are two capabilities to query: company and event supported. TG responds back with its capability.

3.5.2.9. Play item

CT provides commands to start playing an item indicated by the UID. The TG device responds with the corresponding status when it receives a request from the CT.

3.5.3. Browsing commands

To use browsing commands, including libbt_avrcp_br.a library in your application. Browsing commands enable CT to navigate the media on the TG device, then perform operations on a specific media item.

3.5.3.1. Set Browsed Player

To route browsing commands with the scope `BT_AVRCP_SCOPE_MEDIA_PLAYER_VIRTUAL_FILESYSTEM` or BT_AVRCP_SCOPE_SEARCH is the browsed player. CT provides a `bt_avrcp_browse_set_browsed_player()` command to set browsed player. The application can retrieve a list of available players by executing a `bt_avrcp_browse_get_folder_items()` command with the scope of `BT_AVRCP_SCOPE_MEDIA_PLAYER_LIST`.

The TG device must use `bt_avrcp_browse_set_browsed_player_response()` after receiving the event `BT_AVRCP_SET_BROWSED_PLAYER_IND`.

3.5.3.2. Get Folder Items

CT provides a bt_avrcp_browse_get_folder_items() command to retrieve a listing of the contents of a folder with the scope BT_AVRCP_SCOPE_MEDIA_PLAYER_LIST, BT_AVRCP_SCOPE_MEDIA_PLAYER_VIRTUAL_FILESYSTEM, BT_AVRCP_SCOPE_SEARCH or BT_AVRCP_SCOPE_NOW_PLAYING.

The TG device must use `bt_avrcp_browse_get_folder_items_response()` after receiving the event `BT_AVRCP_GET_FOLDER_ITEMS_IND`.

3.5.3.3. Get Item Attributes

CT provides a `bt_avrcp_browse_get_item_attributes()` command to retrieve the metadata attributes for a specific media element item or folder item.

The TG device must use `bt_avrcp_browse_get_item_attributes_response()` after receiving the event `BT_AVRCP_GET_ITEM_ATTRIBUTES_IND`.

3.5.4. Using the AVRCP APIs

This section describes details on using the AVRCP APIs for application development. The AVRCP APIs are implemented in the library code but header functions can be found in `bt_avrcp.h`.

1) Initiate a connection to a TG device. The BT_AVRCP_CONNECT_CNF event is sent to `bt_app_event_callback()` to indicate the connection result.

```
bt_status_t bt_avrcp_connect(uint32_t *handle, const bt_bd_addr_t *address);
```

2) Initiate a connection of AVCTP browsing channel to the remote device. The BT_AVRCP_BROWSE_CONNECT_CNF event is sent to `bt_app_event_callback()` to indicate the connection result. If there is no AVCTP control channel with the device, it will create a connection of AVCTP control channel and then create a connection of AVCTP browsing channel. The BT_AVRCP_CONNECTED_IND event is sent to `bt_app_event_callback()` to indicate the control channel connection result after connection establishment of AVCTP control channel.

```
bt_status_t bt_avrcp_browse_connect(uint32_t *handle, const bt_bd_addr_t
*address);
```

3) Release a connection of AVCTP control channel and AVCTP browsing channel which is present to the remote device. The browsing channel is released before the control channel. The BT_AVRCP_BROWSE_DISCONNECT_IND event and BT_AVRCP_DISCONNECT_IND event is sent to `bt_app_event_callback() by sequence`. If there is no browsing channel, only the control channel is released and only the BT_AVRCP_DISCONNECT_IND event is sent to `bt_app_event_callback().`

```
bt_status_t bt_avrcp_disconnect(uint32_t *handle);
```

4) Release a connection of AVCTP browsing channel to the remote device. The BT_AVRCP_BROWSE_DISCONNECT_IND event is sent to `bt_app_event_callback()`.

```
bt_status_t bt_avrcp_browse_disconnect(uint32_t *handle);
```

5) Send a pass-through command to a TG device. Call this API after connection is established. The BT_AVRCP_PASS_THROUGH_CNF event is sent to `bt_app_event_callback()` to indicate the result of the command.

```
bt_status_t bt_avrcp_send_pass_through_command(
   uint32_t handle,
   bt_avrcp_operation_id_t op_id,
   t_avrcp_operation_state_t op_state);
```

6) Respond to the pass through to CT device. Call this API after receiving BT_AVRCP_PASS_THROUGH_COMMAND_IND.

```
bt_status_t bt_avrcp_send_pass_through_response(
   uint32_t handle,
   bt_avrcp_response_t response,
   bt_avrcp_operation_id_t op_id,
   t_avrcp_operation_state_t op_state);
```

7) List player application setting attribute IDs of a TG device. Call this API after connection is established. The BT_AVRCP_LIST_APP_SETTING_ATTRIBUTES_CNF event is sent to `bt_app_event_callback()` to indicate the result of the command.

```
bt_status_t bt_avrcp_list_app_setting_attributes(uint32_t handle);
```

8) Get player application setting attribute value for a specified value ID. Call this API after connection is established. The BT_AVRCP_GET_APP_SETTING_VALUE_CNF event is sent to `bt_app_event_callback()` to indicate the result of the command.

```
bt_status_t bt_avrcp_get_app_setting_value(
   uint32_t handle,
   uint16_t attribute_size,
   bt_avrcp_get_app_setting_value_t *attribute_list);
```

9) Set player application setting attribute value for a specified value ID. Call this API after connection is established. The BT_AVRCP_SET_APP_SETTING_VALUE_CNF event is sent to `bt_app_event_callback()` to indicate the result of the command.

```
bt_status_t bt_avrcp_set_app_setting_value(
   uint32_t handle,
   uint16_t attribute_size,
   bt_avrcp_app_setting_value_t *attribute_value_list);
```

10) Get metadata attribute of a TG device. Call this API after connection is established. The BT_AVRCP_GET_ELEMENT_ATTRIBUTES_CNF event is sent to `bt_app_event_callback()` to indicate the result of the command.

```
bt_status_t bt_avrcp_get_element_attributes(
   uint32_t handle,
   uint16_t attribute_size,
   bt_avrcp_get_element_attributes_t *attribute_list);
```

11) Set absolute volume to a TG device. Call this API at volume change in CT device. The BT_AVRCP_SET_ABSOLUTE_VOLUME_CNF event is sent to `bt_app_event_callback()` to indicate the result of the command.

```
bt_status_t bt_avrcp_set_absolute_volume(
   uint32_t handle,
   uint8_t volume);
```

12) Send the set absolute volume to a CT device. Call this API after receiving BT_AVRCP_SET_ABSOLUTE_VOLUME_COMMAND_IND.

```
bt_status_t bt_avrcp_set_absolute_volume_response(
   uint32_t handle,
   uint8_t volume);
```

13) Play item to a TG device. Call this API at volume change in CT device. The BT_AVRCP_PLAY_ITEM_CNF event is sent to `bt_app_event_callback()` to indicate the result of the command.

```
bt_status_t bt_avrcp_play_item(
   uint32_t handle,
   bt_avrcp_play_item_t *param);
```

14) Send the play item response to a CT device. Call this API after receiving BT_AVRCP_PLAY_ITEM_COMMAND_IND.

```
bt_status_t bt_avrcp_play_item_response(
   uint32_t handle,
   bt_avrcp_err_code_t status);
```

15) Send the play item response to a CT device. Call this API after receiving BT_AVRCP_SET_ADDRESSED_PLAYER_COMMAND_IND.

```
bt_status_t  bt_avrcp_send_set_addressed_player_response(
    uint32_t handle,
    bt_avrcp_err_code_t status);
```

16) Register a notification event of a TG device. Call this API after connection is established. The `BT_AVRCP_EVENT_NOTIFICATION_IND` event is sent to `bt_app_event_callback()` to indicate the result of the command.

```
  bt_status_t bt_avrcp_register_notification(
    uint32_t handle,
    bt_avrcp_event_t event_id,
    uint32_t play_back_interval);
```

17) Request a TG device to send next packet when received metadata response indicates the packet is not complete. Call this API after connection is established and if the received packet is not the last one. The corresponding metadata event is sent to `bt_app_event_callback()` to indicate the result of the next packet.

```
  bt_status_t bt_avrcp_request_continuing_response(
    uint32_t handle,
    bt_avrcp_pdu_id_t pdu_id);
```

18) Abort a TG device to send next packet when received metadata response indicates the packet is not complete. Call this API after connection is established and if the received packet is not the last one. The `BT_AVRCP_ABORT_CONTINUING_CNF` event is sent to `bt_app_event_callback()` to indicate the result of the command.

```
  bt_status_t bt_avrcp_abort_continuing_response(
    uint32_t handle,
    bt_avrcp_pdu_id_t pdu_id);
```

19) Set browsed player of a TG device. The `BT_AVRCP_BROWSE_SET_BROWSED_PLAYER_CNF` event is sent to `bt_app_event_callback()` to indicate the result of the command.

```
  bt_status_t bt_avrcp_browse_set_browsed_player(
    uint32_t handle,
    uint16_t player_id);
```

20) Respond the set browsed player command to CT device. Call this API after receiving `BT_AVRCP_BROWSE_SET_BROWSED_PLAYER_IND`.

```
  bt_status_t bt_avrcp_browse_set_browsed_player_response(
    uint32_t handle,
    bt_avrcp_browse_set_browsed_player_response_t *param);
```

21) Get folder items of a TG device. The `BT_AVRCP_BROWSE_GET_FOLDER_ITEMS_CNF` event is sent to `bt_app_event_callback()` to indicate the result of the command.

```
  bt_status_t bt_avrcp_browse_get_folder_items(
    uint32_t handle,
    bt_avrcp_browse_get_folder_items_t *param);
```

22) Respond the get folder items command to CT device. Call this API after receiving `BT_AVRCP_BROWSE_GET_FOLDER_ITEMS_IND`.

```
  bt_status_t bt_avrcp_browse_get_folder_items_response(
    uint32_t handle,
    bt_avrcp_browse_get_folder_items_response_t *param);
```

23) Change path of a TG device. The BT_AVRCP_BROWSE_CHANGE_PATH_CNF event is sent to bt_app_event_callback() to indicate the result of the command.

```
bt_status_t bt_avrcp_browse_change_path(
   uint32_t handle,
   bt_avrcp_browse_change_path_t *param);
```

24) Respond the change path command to CT device. Call this API after receiving BT_AVRCP_BROWSE_CHANGE_PATH_IND.

```
bt_status_t bt_avrcp_browse_change_path_response(
   uint32_t handle,
   bt_avrcp_browse_change_path_response_t *param);
```

25) Get item attributes of a TG device. The BT_AVRCP_BROWSE_GET_ITEM_ATTRIBUTES_CNF event is sent to bt_app_event_callback() to indicate the result of the command.

```
bt_status_t bt_avrcp_browse_get_item_attributes(
   uint32_t handle,
   bt_avrcp_browse_get_item_attributes_t *param);
```

26) Respond the get item attributes command to CT device. Call this API after receiving BT_AVRCP_BROWSE_GET_ITEM_ATTRIBUTES_IND.

```
bt_status_t bt_avrcp_browse_get_item_attributes_response(
   uint32_t handle,
   bt_avrcp_browse_get_item_attributes_response_t *param);
```

27) Respond the get total number of items command to CT device. Call this API after receiving BT_AVRCP_BROWSE_GET_TOTAL_NUMBER_OF_ITEMS_IND.

```
bt_status_t bt_avrcp_browse_get_total_number_of_items_response(
   uint32_t handle,
   bt_avrcp_browse_get_total_number_of_items_response_t *param);
```

3.6. PBAP

The PBAP defines the protocols and procedures to retrieve phonebook objects on smart devices. For more information about this profile, refer to PBAP specification.

The following roles are defined for this profile:

- **PBAP Server (PBAPS)** — provides the source phone book objects.
- **PBAP Client (PBAPC)** — pulls the phone book objects.

Note: The term PBAPC is used in the rest of this document to designate the role of the PCE. Only PBAPC role is supported in the SDK.

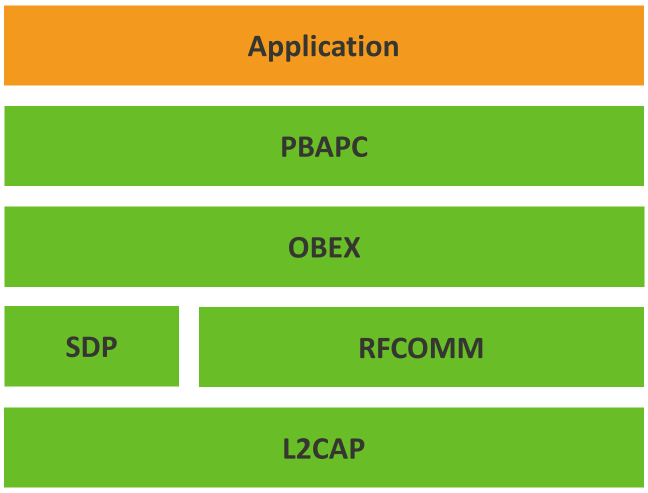

Figure 46. PBAPC abstraction layout

The PBAPC depends on OBEX, as shown in Figure 46. It defines procedures required to send and receive files between two Bluetooth-enabled devices. Similar features are shared between these two profiles, such as connect, get (download and browsing) and disconnect.

3.6.1. The PBAPC features and message sequences

3.6.1.1. PBAP connection

This feature establishes PBAP connection with or without authentication using PBAP Client as an initiator.

1) Establish connection without authentication.

The message sequence is shown in Figure 47. For more details, see `bt_pbapc.h`.

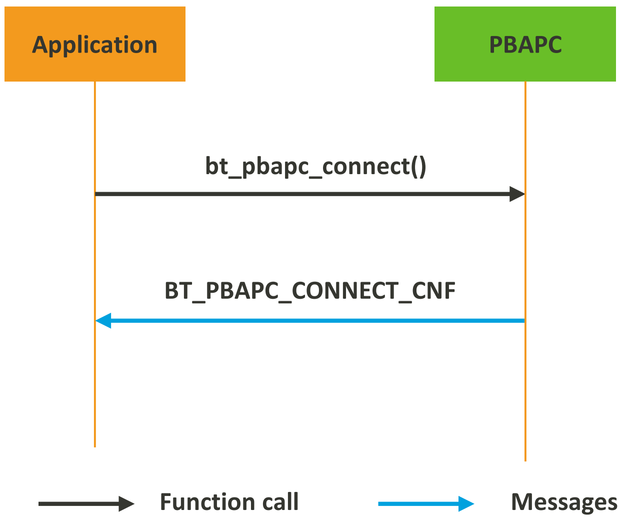

Figure 47. PBAPC connection establishment without authentication message sequence

3.6.1.2. Get phonebook objects

This feature is used to pull phonebook object’s content. The data is in vCard format and the attribute of vCard depends on the PBAP Client’s request parameter, such as the version number of the vCard (2.1 or 3.0) or the offset in the vCard. vCard 3.0 is used in Airoha IoT SDK v4.0.0. The message sequence is shown in Figure 48 . For more details, see `bt_pbapc.h`.

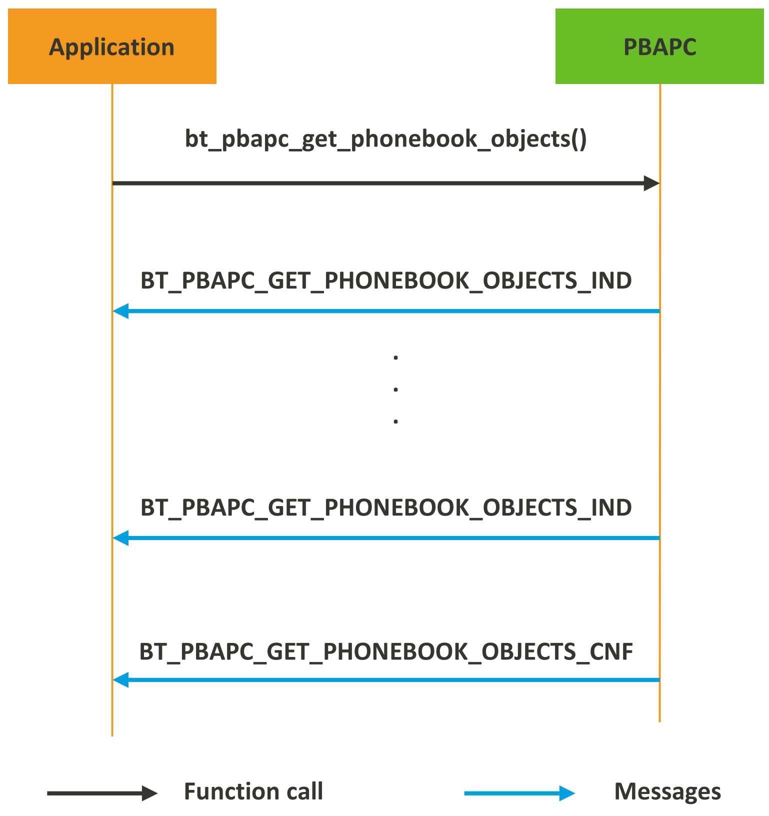
Application
PBAPC
bt_pbapc_get_phonebook_objects()
BT_PBAPC_GET_PHONEBOOK_OBJECTS_IND
BT_PBAPC_GET_PHONEBOOK_OBJECTS_IND
BT_PBAPC_GET_PHONEBOOK_OBJECTS_CNF
Function call
Messages

Figure 48. PBAPC get phone book object message sequence

3.6.1.3. Get the number of phonebook objects by `PullPhonebook`

This feature is used to obtain the number of phonebook objects of type “Missed Calls History” or “Main Phone Book” from the PBAP Server by calling the `PullPhonebook` function after connection is established. For more information about this feature, refer to PBAP specification. The message sequence is shown in Figure 49. For more details, see `bt_pbapc.h`.

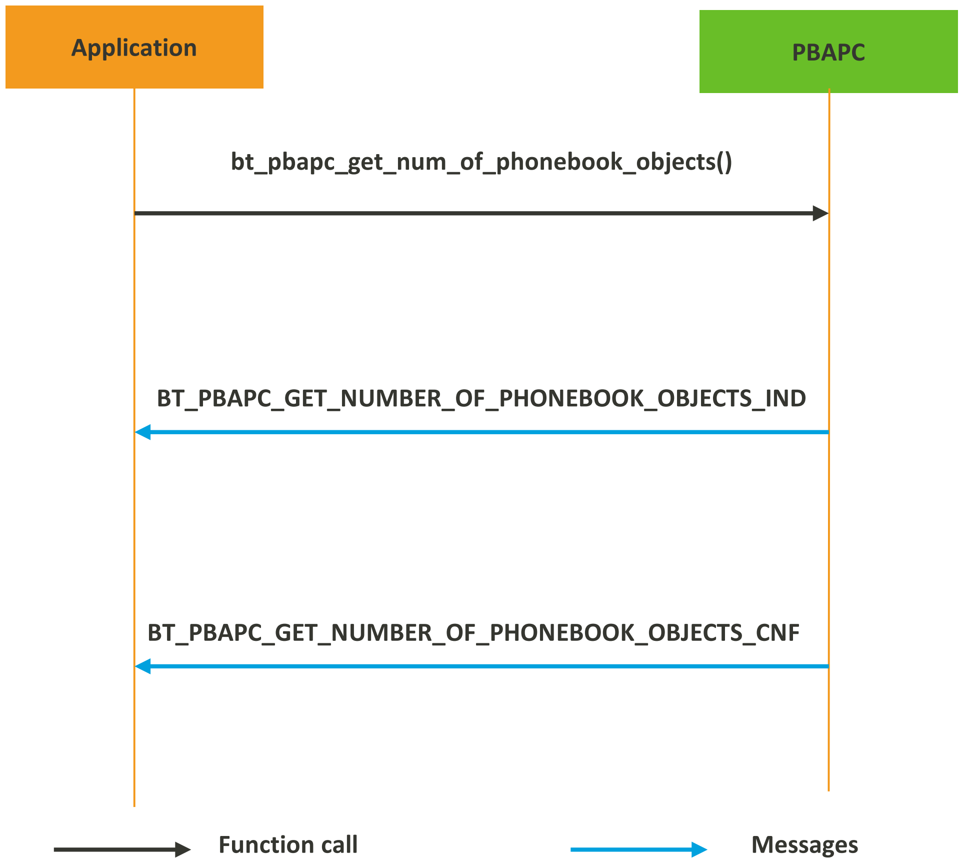
Application
PBAPC
bt_pbapc_get_num_of_phonebook_objects()
BT_PBAPC_GET_NUMBER_OF_PHONEBOOK_OBJECTS_IND
BT_PBAPC_GET_NUMBER_OF_PHONEBOOK_OBJECTS_CNF
Function call
Messages

Figure 49. PBAPC get the number of phonebook objects message sequence

3.6.1.4. Get the caller name by number

This feature is used to get the caller name by number by calling the `PullvCardListing` function. First, get the phonebook listing object from the PBAP Server. Then get the name field by parsing an XML file. The sequence is shown in Figure 50. For more details, see `bt_pbapc.h`.

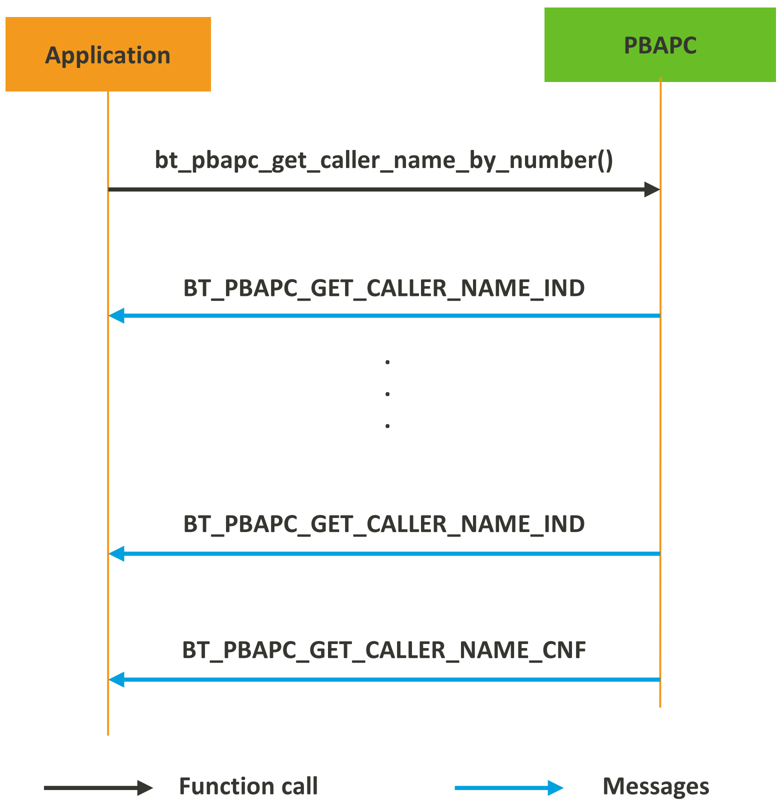
Application
PBAPC
bt_pbapc_get_caller_name_by_number()
BT_PBAPC_GET_CALLER_NAME_IND
BT_PBAPC_GET_CALLER_NAME_IND
BT_PBAPC_GET_CALLER_NAME_CNF
Function call
Messages

Figure 50. PBAPC get caller name by number message sequence

3.6.1.5. `PullvCardEntry` Function

This function is used to retrieve a specific vCard from the PBAP Server. The sequence is shown in Figure 51. For more details, refer to `bt_pbapc.h`.

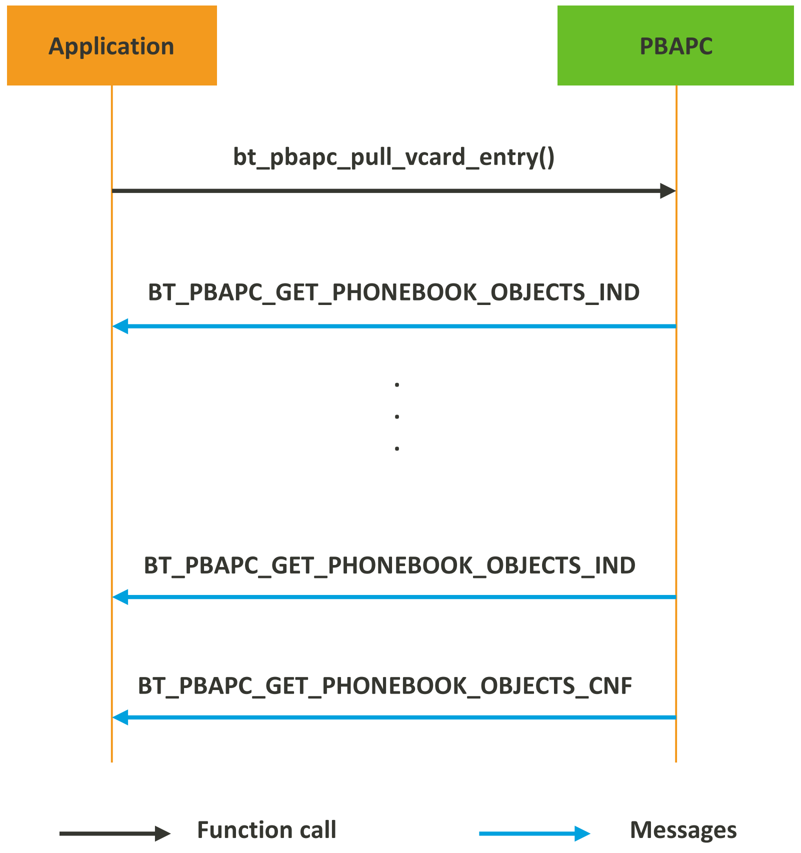
Application
PBAPC
bt_pbapc_pull_vcard_entry()
BT_PBAPC_GET_PHONEBOOK_OBJECTS_IND
BT_PBAPC_GET_PHONEBOOK_OBJECTS_IND
BT_PBAPC_GET_PHONEBOOK_OBJECTS_CNF
Function call
Messages

Figure 51. PBAPC `PullvCardEntry` message sequence

3.6.1.6. Connection release

This feature disconnects the PBAPC connection (see Figure 52). For more details, refer to `bt_pbapc.h`.

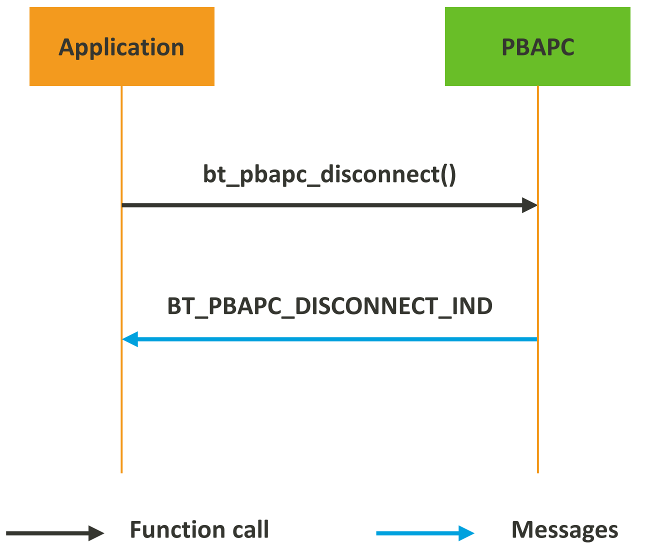

Figure 52. Disconnect message sequence

3.6.2. Using the PBAPC APIs

This section describes the PBAPC API usage for your application development. The source code of the PBAPC APIs is implemented in the library code but header functions can be found in `bt_pbapc.h`.

1) Initiate a connection to PBAP server from the PBAPC, call the APIs as follows:

   a) Call bt_pbapc_connect() API to send a connect request.

   b) The application gets the event BT_PBAPC_CONNECT_CNF to indicate the connection request result.

2) Implement the `bt_app_event_callback` () API. The API is called for events to notify the application or perform a task in this callback.

```
void bt_app_event_callback(bt_msg_type_t event_id, bt_status_t status,
 const void *param);
```

3) Get the number of phonebook objects by calling the following function.

```
bt_status_t bt_pbapc_get_num_of_phonebook_objects(uint32_t handle,
bt_pbapc_phonebook_type_t type);
```

   a) The BT_PBAPC_GET_NUMBER_OF_PHONEBOOK_OBJECTS_IND event will notify application callback function with the data including the number of phonebook objects received from the remote device.

   b) The BT_PBAPC_GET_NUMBER_OF_PHONEBOOK_OBJECTS_CNF event will notify the application callback function to indicate the result of the request.

4) Retrieve vCard objects from the remote device by calling the following function.

```
bt_status_t bt_pbapc_get_phonebook_objects(uint32_t handle, uint16_t
offset, bt_pbapc_phonebook_type_t type);
```

   a) The BT_PBAPC_GET_PHONEBOOK_OBJECTS_IND event will notify application callback function to indicate there is data received from a remote device. It will include name field, name length, number field and number length.

   b) The BT_PBAPC_GET_PHONEBOOK_OBJECTS_CNF event will notify the application callback function to indicate the result of the request.

5) Get caller name by number with the following function.

```
bt_status_t bt_pbapc_get_phonebook_objects(uint32_t handle,
uint8_t*number);
```

a) The BT_PBAPC_GET_CALLER_NAME_IND event will notify application callback function to indicate there is data received from the remote device, it will include name field, name length.

b) The BT_PBAPC_GET_CALLER_NAME_CNF event will notify the application callback function to indicate the result of the request.

6) Pull the specific vCard with the following function.

```
bt_status_t bt_pbapc_pull_vcard_entry(uint32_t handle, uint8_t index);
```

   a) The BT_PBAPC_GET_PHONEBOOK_OBJECTS_IND event will notify application callback function to indicate there is data received from a remote device, it will include name field, name length, number field and number length.

   b) The BT_PBAPC_GET_PHONEBOOK_OBJECTS_CNF event will notify the application callback function to indicate the result of the request.

7) To disconnect the current connection, call this function in the application.

```
bt_status_t bt_pbapc_disconnect(uint32_t handle);
```

a) The BT_PBAPC_DISCONNECT_IND event will notify the application callback function to indicate the result of the request.

3.7. SPP

The SPP provides protocols and procedures for devices using Bluetooth for RS232 (or similar) serial cable emulation.

The SPP defines two roles, SPP client (Device A) and SPP server (Device B).

- **SPP client** — a device that takes the initiates connection with another device.
- **SPP server** — a device that waits for another device to take the initiative to connect.

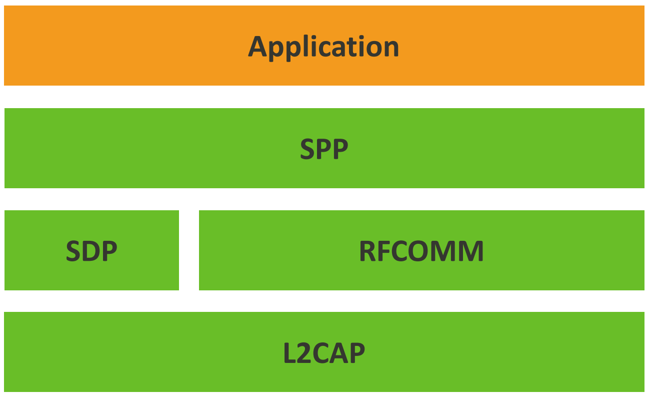

Figure 53. SPP abstraction layout

The SPP is dependent upon the RFCOMM and SDP, as shown in Figure 53. RFCOMM is the Bluetooth adaptation of GSM TS 07.10, providing a transport protocol for serial port emulation. SDP is the Bluetooth Service Discovery Protocol.

3.7.1. The SPP message sequences

SPP enables the following functionality.

1) Connection establishment
2) Connection release
3) Data transfer

The message sequence for each procedure will be described in the upcoming sections.

3.7.1.1. Connection establishment

According to the SPP specification, SPP connection is initiated by the SPP client. The SPP server must implement an SDP record for an emulated serial port before the connection is initiated. For more details about SPP record implementation, refer to `bt_spp.h`.

The connection establishment procedure between the SPP client and the SPP server is shown in Figure 54. For more details, refer to `bt_spp.h`.

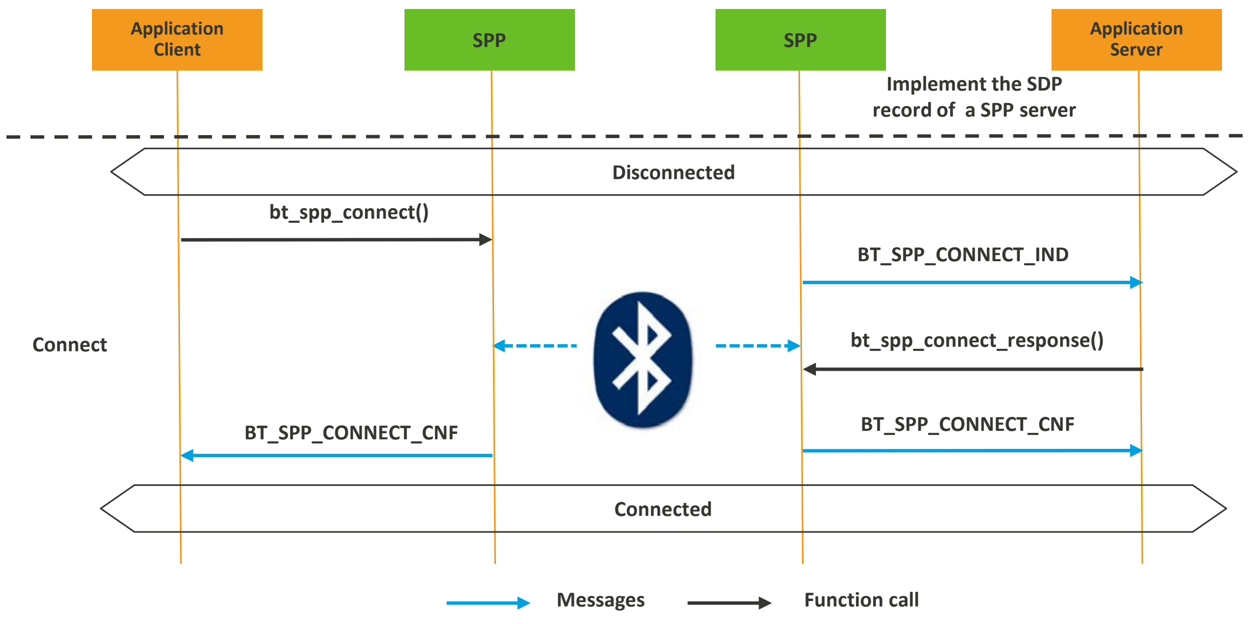

Figure 54. SPP connection establishment message sequence

3.7.1.2. Connection release

The SPP connection release can be initiated by both the SPP client and the SPP server, as shown in Figure 55. More details can be found in the header file `bt_spp.h`.

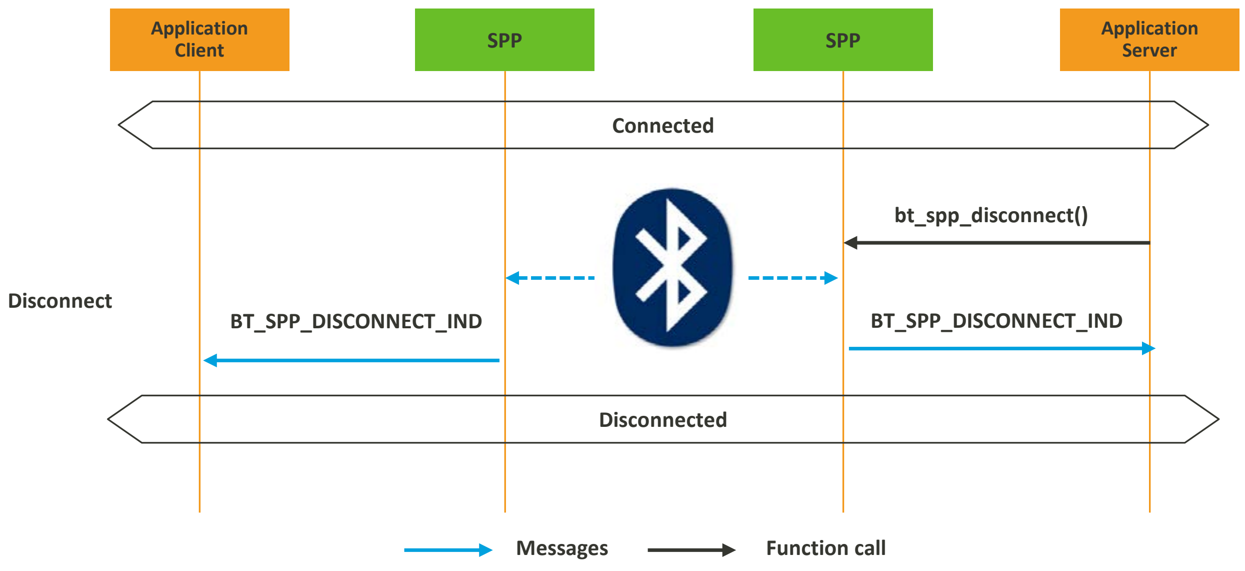

Figure 55. SPP connection release message sequence

3.7.1.3. Data transfer

During the SPP connection, the application data can be transferred between the SPP server and the SPP client, as shown in Figure 56. There is no order restriction, both the SPP server and the SPP client are able to initiate to send and receive data. More details can be found in the header file `bt_spp.h`.

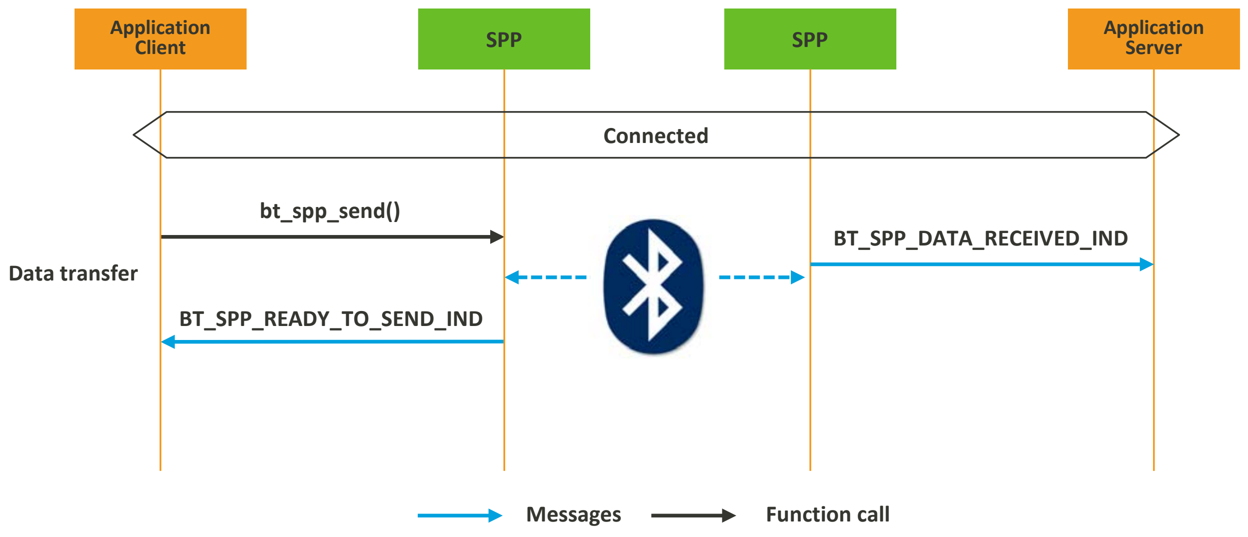

Figure 56. SPP data transfer message sequence

3.7.2. Using the SPP APIs

This section describes how to use the SPP APIs for the application development. The source code of the SPP APIs is implemented in the binary library code but the headers can be found in `bt_spp.h`.

The SDK supports client and server roles and all SPP APIs can be classified into three categories.

1) Common APIs
2) Server-specific APIs
3) Client-specific APIs

3.7.2.1. Common APIs

1) Implement the API `bt_app_event_callback()`. All SPP events will be generated to trigger this function to notify the application.

```
void bt_app_event_callback(bt_event_t event_id, const void *param);
```

2) Disconnect the SPP connection from the SPP server or client. The `BT_SPP_DISCONNECT_IND` event notifies `bt_app_event_callback()` to indicate the result of the disconnection.

```
bt_status_t bt_spp_disconnect(uint32_t handle);
```

3) Application can send data to the remote device using the SPP connection.

```
bt_status_t bt_spp_send(
uint32_t handle,
uint8_t *packet,
uint16_t packet_length);
```

4) Application call hold the SPP data to the Bluetooth RX buffer if it is no free buffer to save the SPP data from the `BT_SPP_DATA_RECEIVED_IND`.

```
void bt_spp_hold_data(uint8_t *data);
```

5) Call this function to release the data once it’s saved in the application.

```
void bt_spp_release_data(uint8_t *data);
```

3.7.2.2. Server-specific APIs

1) Implement the API `bt_sdps_get_customized_record()` to provide the SPP record and record number. More details can be found in the header file `bt_sdp.h`. More details about SPP record implementation can be found in the header file `bt_spp.h`.

```
uint8_t bt_sdps_get_customized_record(const bt_sdps_record***
record_list);
```

2) If the BT_SPP_CONNECT_IND event is generated to trigger `bt_app_event_callback()`, that means the remote SPP client has initiated a connection request to the SPP server. Then the SPP server can respond (accept or reject) this connection request. Finally, the BT_SPP_CONNECT_CNF event is generated to trigger `bt_app_event_callback().` It then notifies the application with the result of this SPP connection establishment.

```
bt_status_t bt_spp_connect_response(uint32_t handle, bool accept);
```

3.7.2.3. Client-specific APIs

Initiate a connection to a remote SPP server using the SPP client. The BT_SPP_CONNECT_CNF event is generated to trigger `bt_app_event_callback()`. Then it notifies the application with the result of this SPP connection establishment.

```
bt_status_t bt_spp_connect(
    uint32_t *handle,
    const bt_bd_addr_t *address,
    const uint8_t *uuid128);
```

3.8. HSP

The Headset Profile (HSP) defines how two devices supporting HSP interact on a point-to-point basis. An implementation of HSP usually enables a headset to wirelessly connect to a smartphone for the purpose of acting as the smartphone‘s audio input and output source.

Two roles are defined for this profile; only the HS role is supported.

Audio Gateway (AG) – This is the device that is the gateway of the audio for input and output. Smartphones usually act as the AG.

Headset (HS) – This is the device that acts as the AG’s remote audio input and output source.

As shown in Figure 57, HSP depends on RFCOMM which defines the procedures that are necessary for sending and receiving user data between two Bluetooth devices. HSP APIs are called in the application to implement the features related to HSP.

The Bluetooth HSP for HS is responsible for HSP connection management, audio connection management, and remote audio volume control.

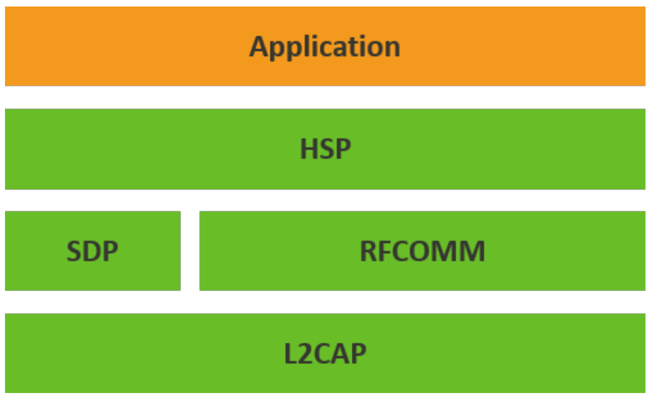

Figure 57. HSP abstraction layout

3.8.1. HSP message sequences

The HSP procedure can be established using the message sequence. The message sequence for each procedure is described in the following sections.

- Connection establishment.
- Connection release.
- Incoming audio connection establishment.
- Outgoing audio connection establishment.
- Audio connection release.
- Audio connection transfer.
- The remote speaker volume control.
- The remote microphone volume control.

3.8.1.1. Connection establishment

Apply this process to establish HSP connection between devices, as shown in Figure 58. There are two different message sequences either application driven or remote device driven. For more details, refer to the header file bt_hsp.h and Handset Profile specification.

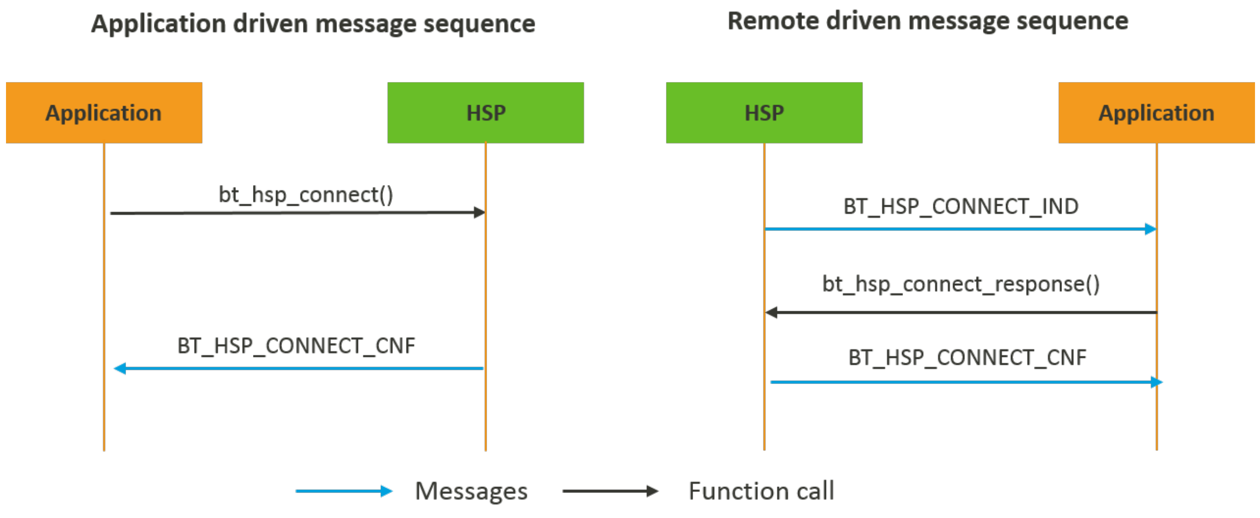

Figure 58. HSP connection establishment message sequence

3.8.1.2. Connection release

Apply this procedure to release the HSP connection from the application, as shown in Figure 59. There are two different message sequences either application driven or remote device driven. For more details, refer to the bt_hsp.h and Handset Profile specification.

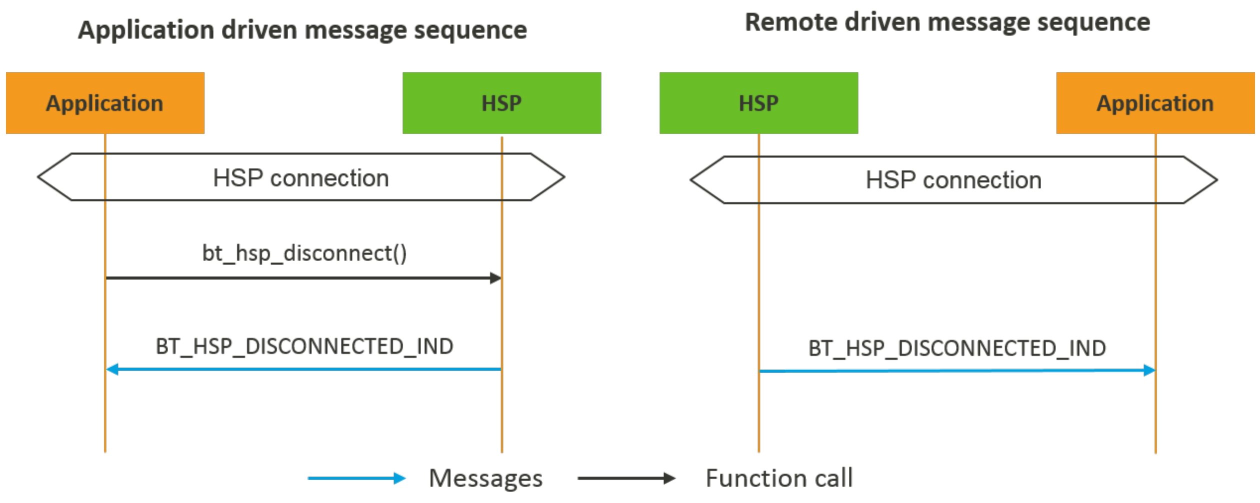

Figure 59. HSP connection release message sequence

3.8.1.3. Incoming audio connection establishment

Apply this procedure to create an SCO link, as shown in Figure 60. The AG will repeatedly send the RING unsolicited result code to the HS for a time period decided by the A. Application will send the command ‘AT+CKPD=200’ to the AG to create an SCO link. For more details, refer to the bt_hsp.h and Handset Profile specification.

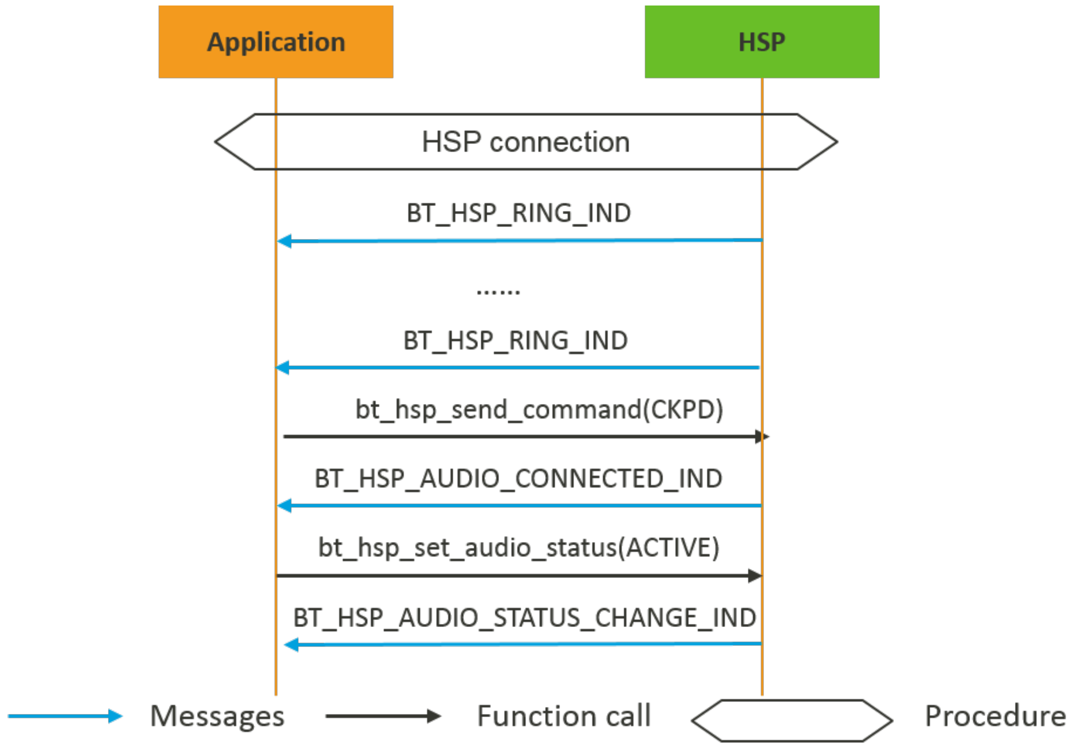
Application
HSP
HSP connection
BT_HSP_RING_IND
......
BT_HSP_RING_IND
bt_hsp_send_command(CKPD)
BT_HSP_AUDIO_CONNECTED_IND
bt_hsp_set_audio_status(ACTIVE)
BT_HSP_AUDIO_STATUS_CHANGE_IND
Messages
Function call
Procedure

Figure 60. Incoming audio connection establishment message sequence

3.8.1.4. Outgoing audio connection establishment

Apply this procedure to create an SCO link, as shown in Figure 61. There are 2 different message sequences either application driven or remote device driven. As the application driven, application will send the command 'AT+CKPD=200' to AG to create the audio connection. For more details, refer to the bt_hsp.h and Handset Profile specification.

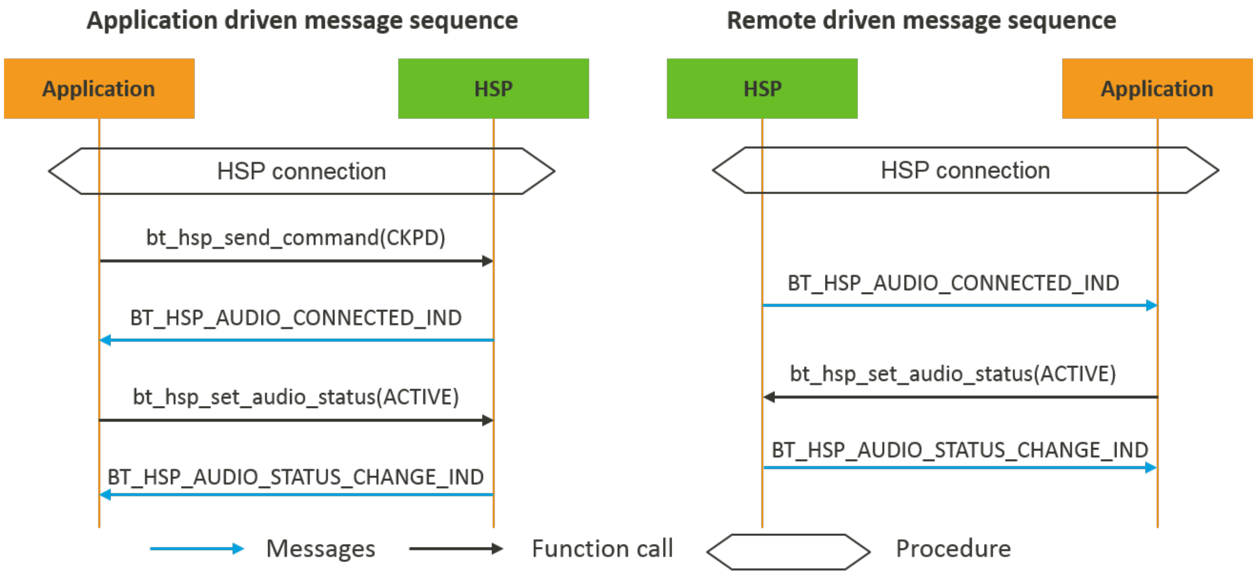
Application driven message sequence
Remote driven message sequence
Application
HSP
HSP
Application
HSP connection
HSP connection
bt_hsp_send_command(CKPD)
BT_HSP_AUDIO_CONNECTED_IND
bt_hsp_set_audio_status(ACTIVE)
BT_HSP_AUDIO_STATUS_CHANGE_IND
BT_HSP_AUDIO_CONNECTED_IND
bt_hsp_set_audio_status(ACTIVE)
BT_HSP_AUDIO_STATUS_CHANGE_IND
Messages
Function call
Procedure

Figure 61. Outgoing audio connection establishment message sequence

3.8.1.5. Audio connection release

Apply this procedure to transfer an SCO link, as shown in Figure 62. There are 2 different message sequences either application driven or remote device driven. As the application driven, if audio connection is ongoing, application will send the command 'AT+CKPD=200' to AG to release the audio connection. For more details, refer to the bt_hsp.h and Handset Profile specification.

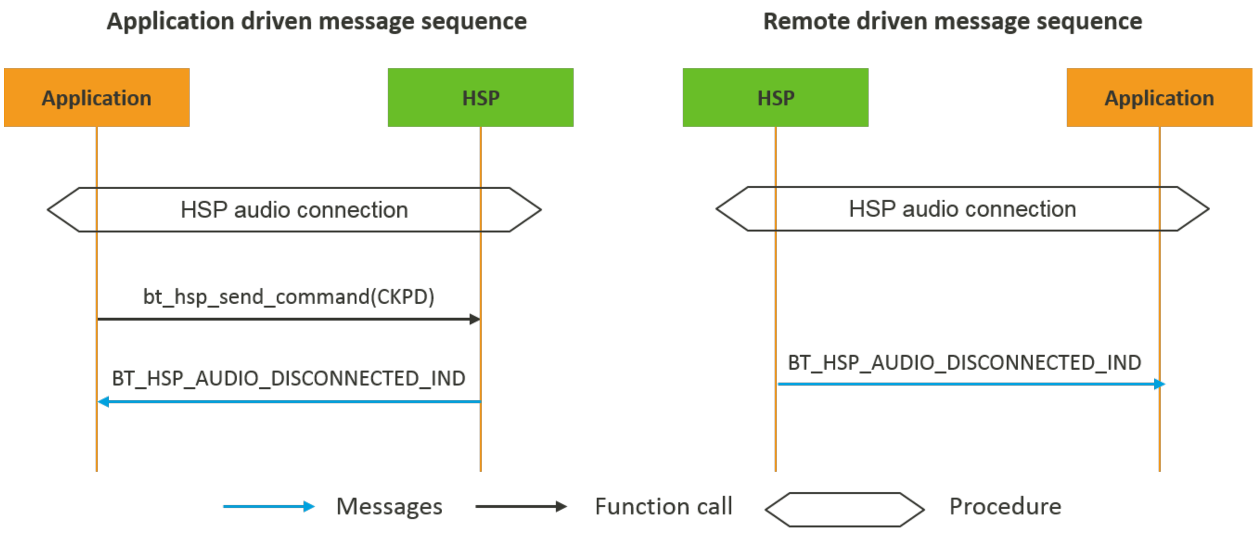
Application driven message sequence
Remote driven message sequence
Application
HSP
HSP
Application
HSP audio connection
HSP audio connection
bt_hsp_send_command(CKPD)
BT_HSP_AUDIO_DISCONNECTED_IND
BT_HSP_AUDIO_DISCONNECTED_IND
Messages
Function call
Procedure

Figure 62. Audio connection release message sequence

3.8.1.6. Audio connection transfer

Apply this procedure to transfer an audio connection from AG to HS or from HS to AG, as shown in Figure 63. There are 2 different message sequences either application driven or remote device driven. As the application driven, application will send the command 'AT+CKPD=200' to AG to establish or release the audio connection. For more details, refer to the bt_hsp.h and Handset Profile specification.

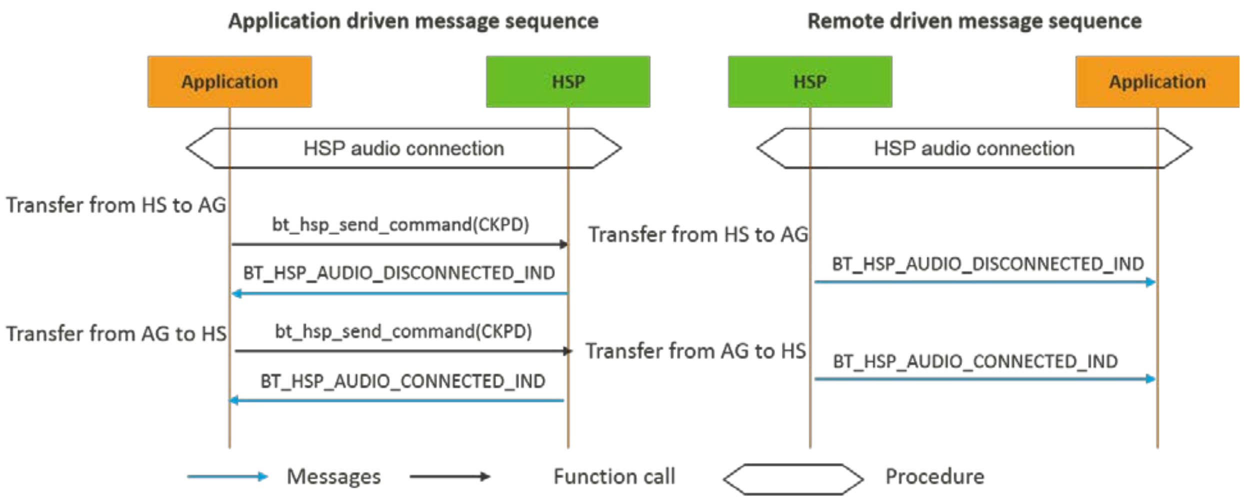
Application driven message sequence
Remote driven message sequence
Application
HSP
HSP
Application
HSP audio connection
HSP audio connection
Transfer from HS to AG
bt_hsp_send_command(CKPD)
BT_HSP_AUDIO_DISCONNECTED_IND
Transfer from AG to HS
bt_hsp_send_command(CKPD)
BT_HSP_AUDIO_CONNECTED_IND
Transfer from HS to AG
BT_HSP_AUDIO_DISCONNECTED_IND
Transfer from AG to HS
BT_HSP_AUDIO_CONNECTED_IND
Messages
Function call
Procedure

Figure 63. Audio connection transfer message sequence

3.8.1.7. The remote speaker volume control

Apply this process to control the remote speaker volume as shown in Figure 64. There are 2 different message sequences either application driven or remote device driven. As the application driven, application will send the command 'AT+VGS=XX' (XX is the volume value need to set) to synchronize the speaker volume. For more details, refer to the bt_hsp.h and Handset Profile specification.

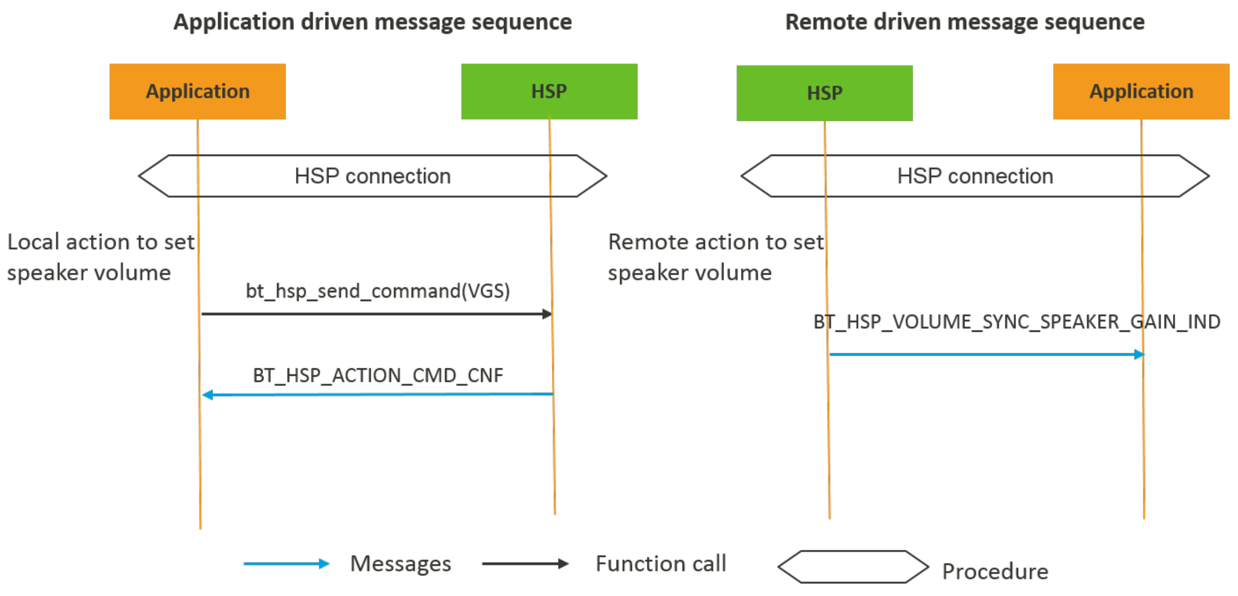
Application driven message sequence
Application
HSP
HSP connection
Local action to set speaker volume
bt_hsp_send_command(VGS)
BT_HSP_ACTION_CMD_CNF
Remote driven message sequence
HSP
Application
HSP connection
Remote action to set speaker volume
BT_HSP_VOLUME_SYNC_SPEAKER_GAIN_IND
Messages
Function call
Procedure

Figure 64. The remote speaker volume control message sequence

3.8.1.8. The remote microphone volume control

Apply this process to synchronize the microphone volume as shown in Figure 65. There are 2 different message sequences either application driven or remote device driven. As the application driven, application will send the command 'AT+VGM=XX' (XX is the volume value need to set) to synchronize the microphone volume. For more details, refer to the bt_hsp.h and Hands-free Profile specification.

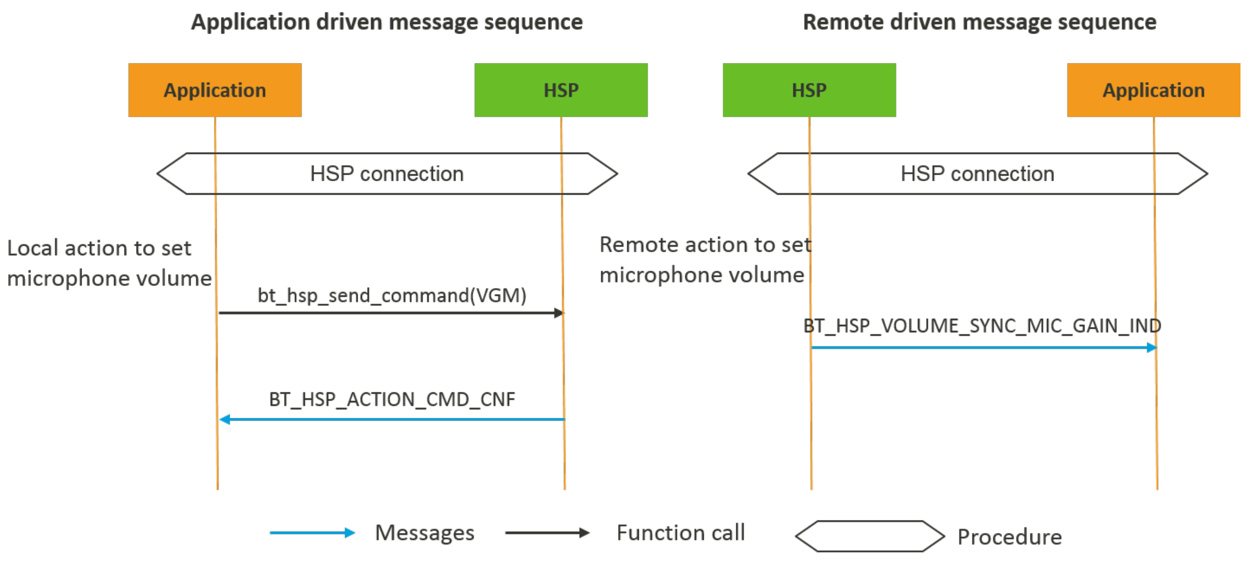
Application driven message sequence
Application
HSP
HSP connection
Local action to set microphone volume
bt_hsp_send_command(VGM)
BT_HSP_ACTION_CMD_CNF
Remote driven message sequence
HSP
Application
HSP connection
Remote action to set microphone volume
BT_HSP_VOLUME_SYNC_MIC_GAIN_IND
Messages
Function call
Procedure

Figure 65. The remote microphone volume control message sequence

3.8.2. How to use the HSP APIs

This section describes how to use the HSP APIs for the application development. The source code of the HSP APIs is implemented in the binary library code but the headers can be found in bt_hsp.h.

To use the HSP APIs in your application:

1) Check the HSP SDP service record is enabled or not. The HSP SDP service record default enabled if HSP feature is enabled. If the API's returned value is false, The HSP SDP service record is disabled so that the remote device initiating the HSP connection will fail.

```
bool bt_hsp_check_service_record(void);
```

2) Enable or disable the HSP SDP service record. If the API's parameter is true, The HSP SDP service record will be enabled, or else it will be disabled.

```
void bt_hsp_enable_service_record(bool enable);
```

3) Implement the API bt_app_event_callback(). All HSP events will be generated to trigger this function to notify the application.

```
void bt_app_event_callback(bt_event_t event_id, const void *param);
```

4) Initiate a HSP connection to a remote device. The BT_HSP_CONNECT_CNF event is generated to trigger `bt_app_event_callback()`. Then it notifies the application with the result of this HSP connection establishment.

```
bt_status_t bt_hsp_connect(
uint32_t *handle,
    const bt_bd_addr_t *address);
```

5) If the BT_HSP_CONNECT_IND event is generated to trigger `bt_app_event_callback()`, it means that the remote device has initiated a connection request to the HS. Then the HS can respond (accept or reject) this connection request. Finally, the BT_HSP_CONNECT_CNF event is generated to trigger `bt_app_event_callback()`. Then it notifies the application with the result of this HSP connection establishment.

```
bt_status_t bt_hsp_connect_response(uint32_t handle, bool accept);
```

6) To send the HSP command to remote device. The return value is the result of the command sending. If it returns BT_STATUS_SUCCESS, the command send success, if it returns BT_STATUS_FAIL, the command send fail, if it returns BT_STATUS_OUT_OF_MEMORY, the command send fail due to there's not enough memory to send, about how to handle this out of memory case, please refer to 5.1.2. Application gets the event ID BT_HSP_ACTION_CMD_CNF to get the response of remote device.

```
bt_status_t bt_hsp_send_command(uint32_t handle, const char*command);
```

7) To disconnect HSP connection from the HS and then get the event BT_HSP_DISCONNECTED_IND with the disconnect result. If the remote device initiates the disconnection, the application will directly get the event BT_HFP_DISCONNECT_IND result.

```
bt_status_t bt_ hsp_disconnect(uint32_t handle);
```

3.9. DI

The DI profile defines a method via Bluetooth SDP records for providing information about the Bluetooth devices which can be used by peer Bluetooth devices to support different features, as shown in Figure 66.

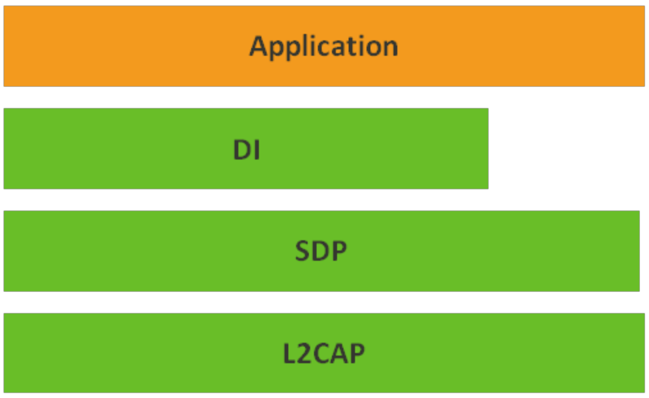

Figure 66. DI abstraction layout

3.9.1. The DI message sequences

The application on the local device might have one or more Device ID Service Records, and is identified by a unique UUID called PNP Information. It might specify the brand and device information (such as manufacture and product identifier) in the records. Peer device can query this information by SDP, as shown in Figure 67.

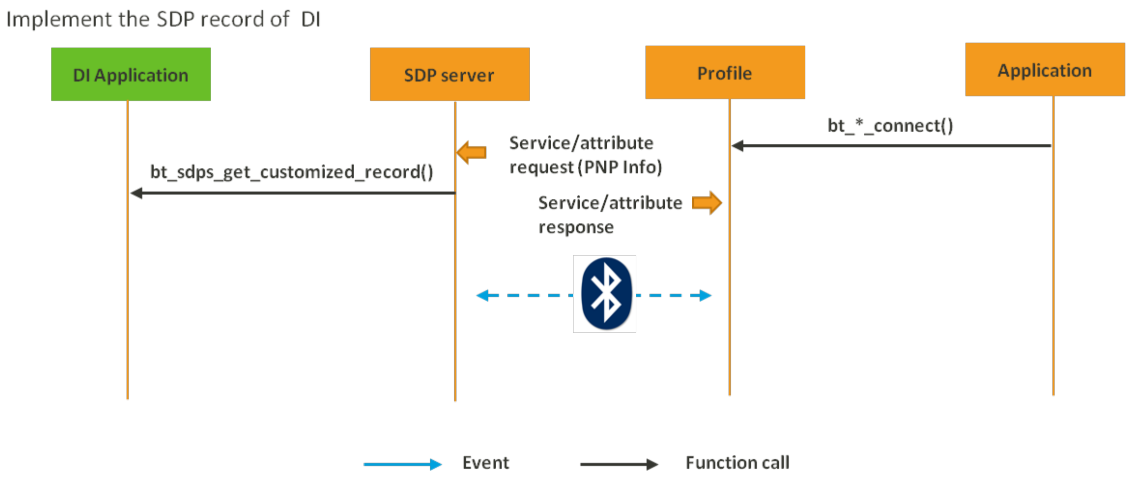
Implement the SDP record of DI
DI Application
SDP server
Profile
Application
bt_*_connect()
Service/attribute request (PNP Info)
bt_sdps_get_customized_record()
Service/attribute response
Event
Function call

Figure 67. DI record query message sequence

3.9.2. How to use the DI APIs

Here is an example (see section 3.9.2.1, “Full source code to add a DI record”) to add an example DI record that contains the service class ID list, the specification ID, the vendor ID, the product ID, the version, the primary record and the vendor ID source for a sample profile. Add your own record according to the DI profile specification. For more information about the service attributes, refer to the Device ID profile. More details about the API can be found in the header file `bt_di.h`.

Details of an example are explained below.

1) Define the mandatory attributes.

```
static const bt_sdps_attribute_t bt_di_sdp_attributes[] = {
    BT_SDP_ATTRIBUTE(BT_SDP_ATTRIBUTE_ID_SERVICE_CLASS_ID_LIST, bt_di_service_class_id_list),
    BT_SDP_ATTRIBUTE(BT_DI_SDP_ATTRIBUTE_SPECIFICATION_ID, bt_di_specification_id),
    BT_SDP_ATTRIBUTE(BT_DI_SDP_ATTRIBUTE_VENDOR_ID, bt_di_vendor_id),
    BT_SDP_ATTRIBUTE(BT_DI_SDP_ATTRIBUTE_PRODUCT_ID, bt_di_product_id),
    BT_SDP_ATTRIBUTE(BT_DI_SDP_ATTRIBUTE_VERSION, bt_di_version),
    BT_SDP_ATTRIBUTE(BT_DI_SDP_ATTRIBUTE_PRIMARY_RECORD, bt_di_primary_record),
    BT_SDP_ATTRIBUTE(BT_DI_SDP_ATTRIBUTE_VENDOR_ID_SOURCE, bt_di_vendor_id_source)
};
```

2) Implement the API `bt_sdps_get_customized_record()` to provide the DI record and record number.

```
static const bt_sdps_record_t bt_di_sdp_record = {
    .attribute_list_length = sizeof(bt_di_sdp_attributes),
    .attribute_list = bt_di_sdp_attributes,
};

static const bt_sdps_record_t *my_sdps_record[] = {
    &bt_di_sdp_record
};

uint8_t bt_sdps_get_customized_record(const bt_sdps_record_t *** record_list)
{
    *record_list = my_sdps_record;
    return sizeof(my_sdps_record) / sizeof(bt_sdps_record_t *);

}
```

3.9.2.1. Full source code to add a DI record

```
#define MY_BT_DI_VENDOR_ID        0x1234   // The vendor ID definition.
#define MY_BT_DI_PRODUCT_ID       0x0004   // A value defined for product ID and it should be managed and
assigned by the vendor.
#define MY_BT_DI_VERSION        0x0100   // A value defined for the version number, in this example it's 1.0.0.

static const uint8_t bt_di_service_class_id_list[] = {
    BT_DI_SERVICE_CLASS_ID_LIST
};

static const uint8_t bt_di_specification_id[] = {
    BT_DI_SPECIFICATION_ID
};

static const uint8_t bt_di_vendor_id[] = {
    BT_DI_VENDOR_ID(MY_BT_DI_VENDOR_ID)    // The device ID.
};

static const uint8_t bt_di_product_id[] = {
    BT_DI_PRODUCT_ID(MY_BT_DI_PRODUCT_ID)    // The product ID managed by the vendor to distinguish itself
from different products.
};

static const uint8_t bt_di_version[] = {
    BT_DI_VERSION(MY_BT_DI_VERSION)    // The version number assigned by the vendor to identify the device's
release number.
};

static const uint8_t bt_di_primary_record[] = {
    BT_DI_PRIMARY_RECORD(BT_DI_PRIMARY_RECORD_TRUE)    // It should be set with
#BT_DI_PRIMARY_RECORD_TRUE, when there is only one Device ID service record in the service.
 };

 static const uint8_t bt_di_vendor_id_source[] = {
    BT_DI_VENDOR_ID_SOURCE(BT_DI_VENDOR_ID_SOURCE_USB_FORUM_ASSIGNED)
 };

 static const bt_sdps_attribute_t bt_di_sdp_attributes[] = {
    BT_SDP_ATTRIBUTE(BT_SDP_ATTRIBUTE_ID_SERVICE_CLASS_ID_LIST, bt_di_service_class_id_list),
    BT_SDP_ATTRIBUTE(BT_DI_SDP_ATTRIBUTE_SPECIFICATION_ID, bt_di_specification_id),
    BT_SDP_ATTRIBUTE(BT_DI_SDP_ATTRIBUTE_VENDOR_ID, bt_di_vendor_id),
    BT_SDP_ATTRIBUTE(BT_DI_SDP_ATTRIBUTE_PRODUCT_ID, bt_di_product_id),
    BT_SDP_ATTRIBUTE(BT_DI_SDP_ATTRIBUTE_VERSION, bt_di_version),
    BT_SDP_ATTRIBUTE(BT_DI_SDP_ATTRIBUTE_PRIMARY_RECORD, bt_di_primary_record),
    BT_SDP_ATTRIBUTE(BT_DI_SDP_ATTRIBUTE_VENDOR_ID_SOURCE, bt_di_vendor_id_source)
 };

 static const bt_sdps_record_t bt_di_sdp_record = {
    .attribute_list_length = sizeof(bt_di_sdp_attributes),
    .attribute_list = bt_di_sdp_attributes,
 };

 static const bt_sdps_record_t *my_sdps_record[] = {
    &bt_di_sdp_record
 };

 //Implements the function #bt_sdps_get_customized_record().
 uint8_t bt_sdps_get_customized_record(const bt_sdps_record_t *** record_list)
 {
    *record_list = my_sdps_record;
    return sizeof(my_sdps_record) / sizeof(bt_sdps_record_t *);
 }
```

4. The Bluetooth Low Energy Protocol or Profiles

The addition to the Airoha IoT SDK for Bluetooth support is provided as a collection of binary library files. This section provides details on the GAP, SM and GATT profiles.

4.1. GAP

The Bluetooth LE GAP is responsible for connection and device management, including device configuration, device discovery, and connection establishment and termination, as shown in Figure 68.

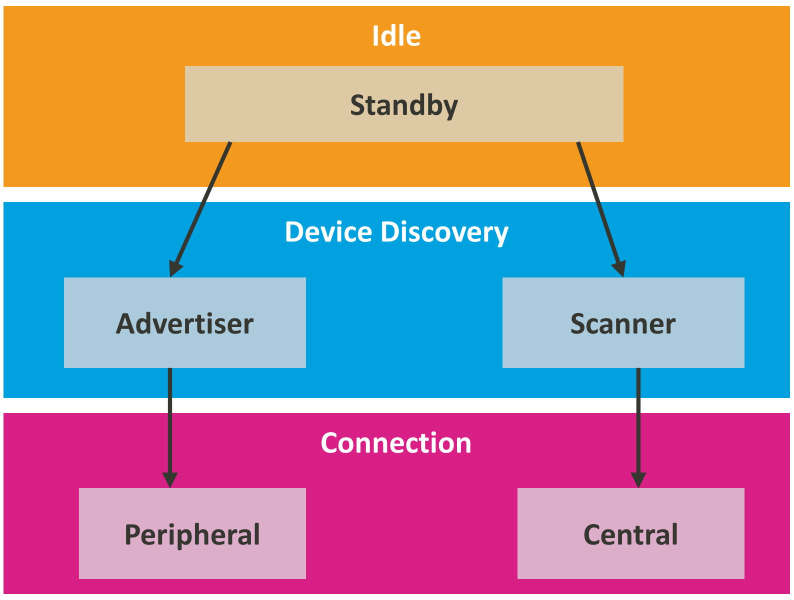

Figure 68. GAP state diagram

- **Standby** — initial idle state when the Bluetooth powers on.
- **Advertiser** — the device is advertising with specific data to provide information to nearby devices about the advertiser, including the device address, advertising type and advertising data.
- **Scanner** — the device searches for nearby devices, based on the type of scanning and advertisement, it may send a scan request to the advertiser. The scanner can get more information in the scan response from the advertiser. Once the discovery is complete, the scanner can initiate a connection or not with the advertiser based on the advertising type.
- **Peripheral/Central** — once the scanner initiates a connection to the advertiser, the advertiser device will function as a peripheral device and the scanner device will be a central device.

The application and profiles can directly invoke Bluetooth LE GAP API functions to perform Bluetooth LE related functions, such as scan or connect, as shown in Figure 69. More details can be found in the header file `bt_gap_le.h`.

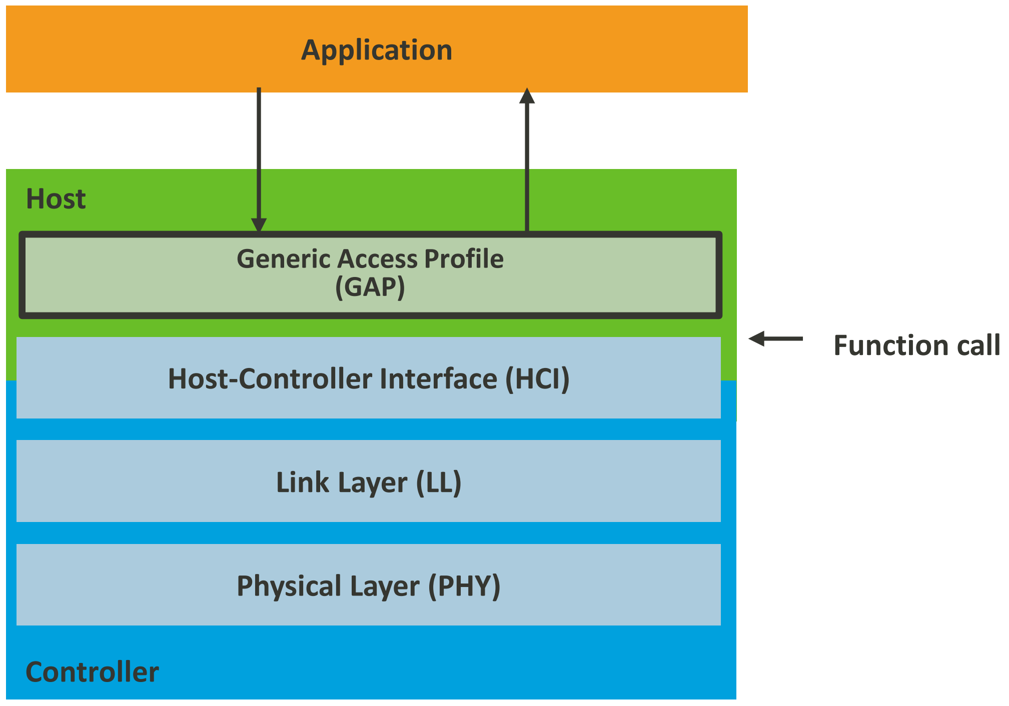

Figure 69. GAP abstraction layout

4.1.1. Power on

Call the `bt_power_on()` function in the `main()` function to enable the Bluetooth. The BT_POWER_ON_CNF event is received after successful completion, as shown in Figure 70. More details can be found in the header file `bt_system.h`.

```
bt_status_t bt_app_event_callback(bt_msg_type_t msg, bt_status_t status,
void *buff)
{
    switch (msg) {
        case BT_POWER_ON_CNF:
        {
            break;
        }
    }
    return BT_STATUS_SUCCESS;
}
```

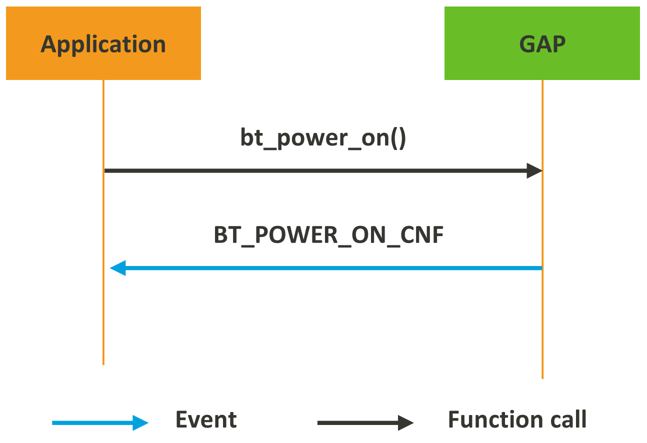

Figure 70. Powering on the Bluetooth message sequence

4.1.2. Start advertising

Start advertising to enable device discovery and connectivity, as shown below.

```
void app_fun(void)
{
    bt_hci_cmd_le_set_advertising_enable_t adv_enable = {
        .advertising_enable = BT_HCI_ENABLE,
    };
    bt_hci_cmd_le_set_advertising_parameters_t adv_para = {
        .advertising_interval_min = 0x0800,
        .advertising_interval_max = 0x0800,
        .advertising_type = BT_HCI_ADV_TYPE_CONNECTABLE_UNDIRECTED,
        .advertising_channel_map = 7,
        .advertising_filter_policy = 0
    };
    bt_hci_cmd_le_set_advertising_data_t adv_data = {0};
    char dev_name[] = ”My Device”;

    adv_data.advertising_data_length = strlen(dev_name) + 2;
    adv_data.advertising_data[0] = strlen(dev_name) + 1;
    adv_data.advertising_data[1] = BT_GAP_LE_AD_TYPE_NAME_COMPLETE;
    memcpy(adv_data.advertising_data + 2, dev_name, strlen(dev_name));

    bt_gap_le_set_advertising(&adv_enable, &adv_para, &adv_data, NULL);

    return;
}
```

4.1.2.1. Advertising and scan response data format

The format of advertising data can be found in the Bluetooth Core Specifications version 4.2 [VOL 3, Part C, 11], Bluetooth GAP and in the Specification of the Bluetooth system.

4.1.3. Searching the nearby devices

The device scans for the nearby devices, as shown in Figure 71 and Figure 72. More details can be found in the header file `bt_gap_le.h`.

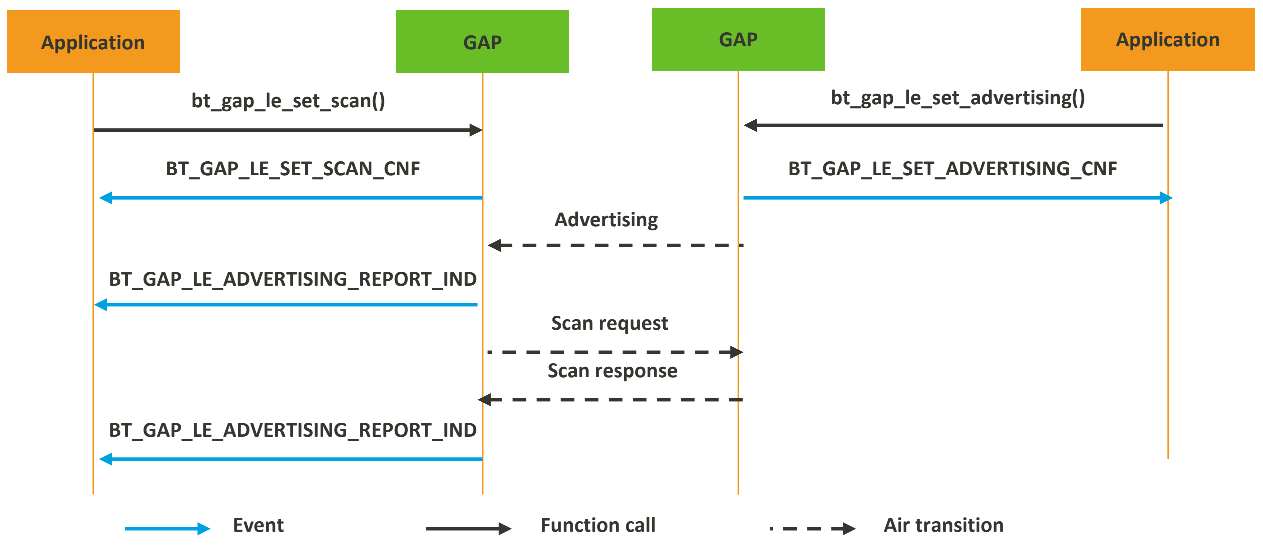

Figure 71. GAP active scan message sequence

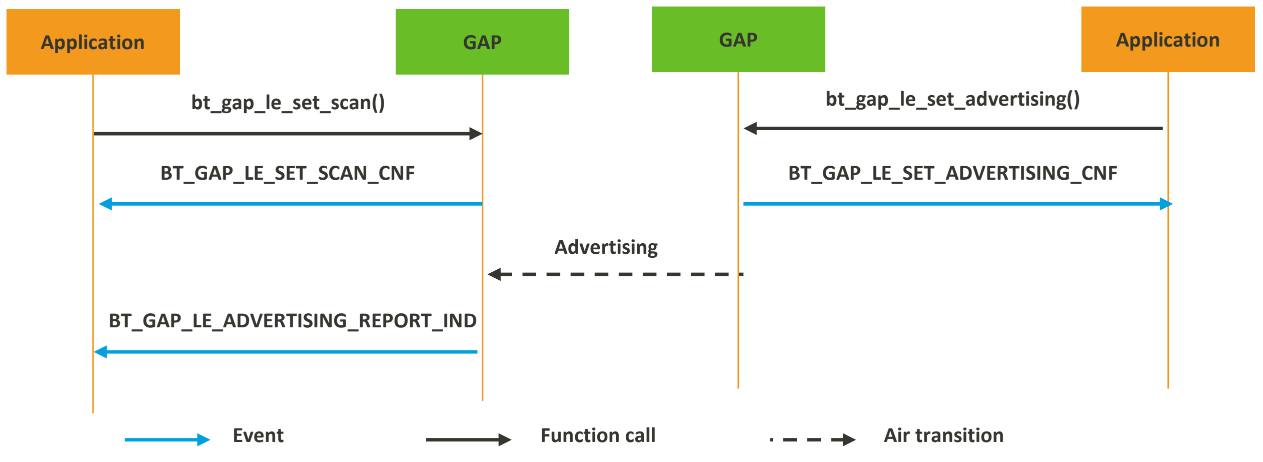

Figure 72. GAP passive scan message sequence

An example code to provide GAP active scan (see Figure 71) is shown below:

```
void app_fun(void)
{
    bt_hci_cmd_le_set_scan_parameters_t scan_params;

    //Set scan parameters.
    scan_params.le_scan_type = BT_HCI_SCAN_TYPE_PASSIVE;
    scan_params.own_address_type = BT_HCI_SCAN_ADDR_PUBLIC;
    scan_params.le_scan_interval = 0x0024;
    scan_params.le_scan_window = 0x0011;
    scan_params.scanning_filter_policy = 0x00;

    bt_hci_cmd_le_set_scan_enable_t enalbe;
    enable.le_scan_enable = BT_HCI_ENABLE;
    enable.filter_duplicates = BT_HCI_ENABLE;

    //Start the scanner.
    bt_gap_set_scan(&enable, &param);
}
```

```
bt_status_t bt_app_event_callback(bt_msg_type_t msg, bt_status_t status,
void *buff)
{
    switch (msg) {
        case BT_GAP_LE_ADVERTISING_REPORT_IND:
        {
            const bt_gap_le_advertising_report_ind_t *report =
                (bt_gap_le_advertising_report_ind_t *)buff;
            // Log advertising report information.
        }
    }
    return BT_STATUS_SUCCESS;
}
```

4.1.4. Connecting

There are two ways to create a Bluetooth LE connection, a general connection and auto connection.

1) General connection. It enables the host to establish a connection with a specified device, as shown in Figure 73. More details can be found in the header file `bt_gap_le.h`.

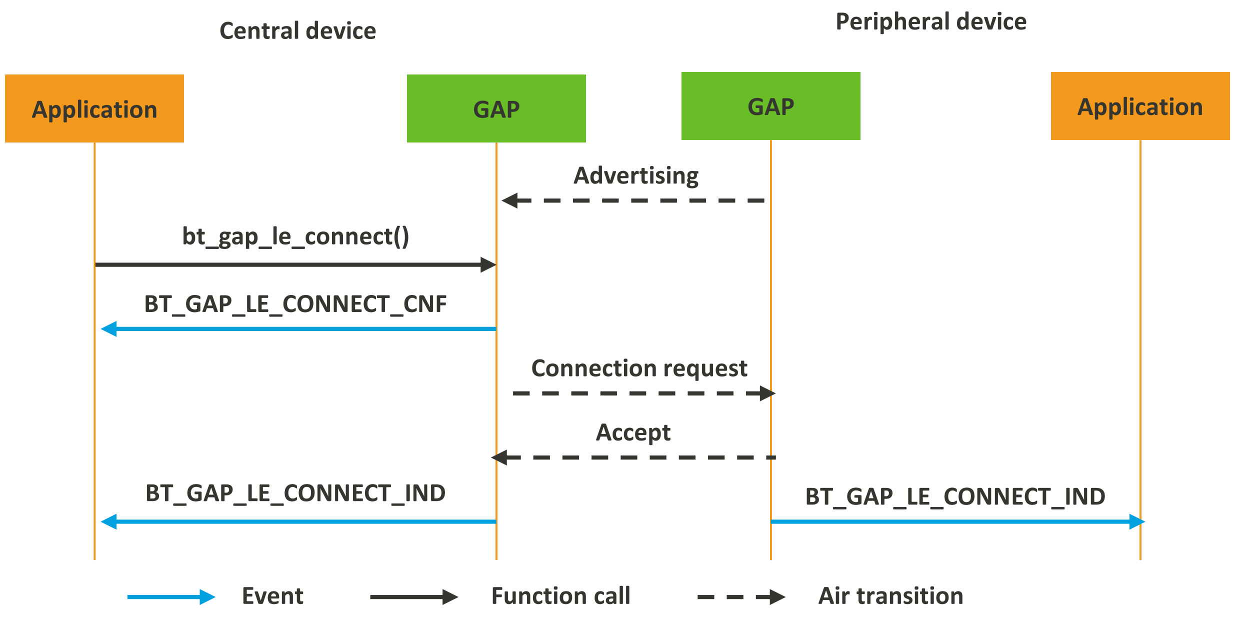

Figure 73. GAP general connection message sequence

An example code to provide general connection (see Figure 73) is shown below.

```
void app_fun(void)
{
    bt_hci_cmd_le_create_connection_t conn_params;
    uint8_t addr[6] = {0xC0, 0x74, 0x3A, 0x2A, 0x09, 0xFF};

    //Set connection parameters.
    conn_params.le_scan_interval = 0x0060;
    conn_params.le_scan_window = 0x0030;
    conn_params.initiator_filter_policy =
BT_HCI_CONN_FILTER_ASSIGNED_ADDRESS;
    conn_params.peer_address.type = 0x00;
    memcpy(conn_params.peer_address.addr, addr,sizeof(addr));
    conn_params.own_address_type = BT_ADDR_RANDOM;
    conn_params.conn_interval_min = 0x0018;
    conn_params.conn_interval_max = 0x0028;
    conn_params.conn_latency = 0x0000;
    conn_params.supervision_timeout = 0x0C80;
    conn_params.minimum_ce_length = 0x0060;
      conn_params.maximum_ce_length = 0x0140;
    //Start connecting.
    bt_gap_le_connect(&conn_params);

    return;
}
```

2) Auto-connection. It enables the host to configure the controller to establish a connection automatically with one or more devices. A white list in the controller stores device addresses. The controller automatically establishes a connection with devices in the white list, as shown in Figure 74. More details can be found in the header file `bt_gap_le.h`.

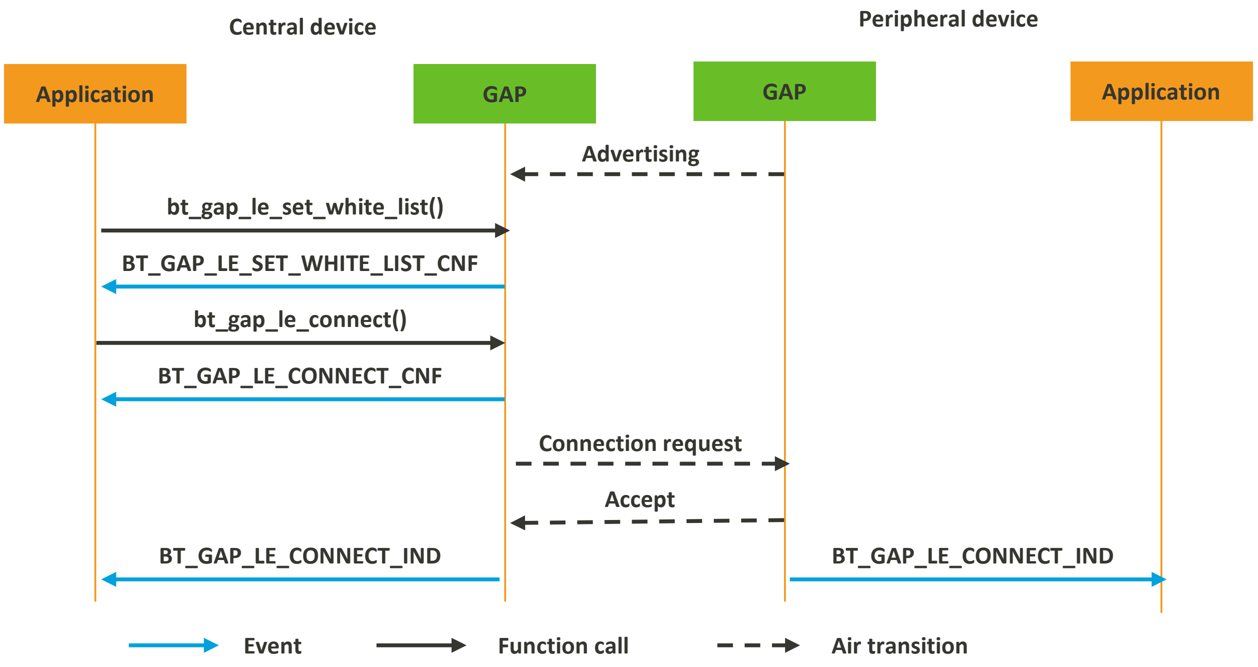

Figure 74. GAP auto connection message sequence

```
void app_fun(void)
{
    uint8_t addr[6] = {0xC0, 0x74, 0x3A, 0x2A, 0x09, 0xFF};
    bt_addr_t device;

    device.type = 0x00;
    memcpy(device.addr, addr, sizeof(addr));

    //Add the device to the white list
    bt_gap_le_set_white_list(BT_GAP_LE_ADD_TO_WHITE_LIST, &device);
    return;
}

bt_status_t bt_app_event_callback(bt_msg_type_t msg, bt_status_t status,
void *buff)
{
    switch (msg) {
        case BT_GAP_LE_ADVERTISING_REPORT_IND:
        {
            bt_hci_cmd_le_create_connection_t conn_params;

            //Set connection parameters.
            conn_params.le_scan_interval = 0x0060;
            conn_params.le_scan_window = 0x0030;
            conn_params.initiator_filter_policy =
                BT_HCI_CONN_FILTER_WHITE_LIST_ONLY;
            conn_params.own_address_type = BT_ADDR_RANDOM;
            conn_params.conn_interval_min = 0x0018;
            conn_params.conn_interval_max = 0x0028;
            conn_params.conn_latency = 0x0000;
            conn_params.supervision_timeout = 0x0C80;
            conn_params.minimum_ce_length = 0x0060;
              conn_params.maximum_ce_length = 0x0140;
            //Start connecting.
            bt_gap_le_connect(&conn_params);

            break;
        }
    }

    return BT_STATUS_SUCCESS;
}
```

Calling the `bt_gap_le_connect()` function again, while the GAP is connecting, will result in failure (`BT_STATUS_FAIL`). In this case, the application should call `bt_gap_le_cancel_connection()` to cancel the connection and then call `bt_gap_le_connect()` to re-connect.

`BT_GAP_LE_CONNECT_IND` event is received, when connecting or being connected successfully.

```
bt_status_t bt_app_event_callback(bt_msg_type_t msg, bt_status_t status,
void *buff)
{
    switch (msg) {
        case BT_GAP_LE_CONNECT_IND:
        {
            bt_gap_le_connect_ind_t *event;
            event = (bt_gap_le_connect_ind_t *)buff;
            //Handle connected event.
            break;
        }
    }
    return;
}
```

4.1.4.1. Connection parameters

The connection parameters in this section affect the power consumption and performance. They are sent by the central device with the connection request and can be modified by a peripheral device, once the connection is established.

1) Connection Interval. During the connection interval, the two devices send and receive data only in Bluetooth LE connection state. If there is no data to send, the two devices will still keep sending link layer data to notify the peer device that it is still alive. The current increases if there is data to send in the

connection event, as shown in Figure 75. The range of connection interval is from 7.5ms to 4s. The larger the connection interval, the lower the power consumption and the throughput are. The ideal throughput is defined as $\frac{(\text{conn\_interval}/0.625\text{ms}-3)\text{slot} * 23 \text{ bytes}/1.1\text{slot}}{\text{conn\_interval}}$, where 23 bytes are useful bytes in the PDU. Three slots deducted in the expression are reserved in each connection interval and 23 bytes per 1.1 slot is the maximum BLE baseband capacity for Bluetooth 4.1.

More details can be found in the header file `bt_hci_le.h`.

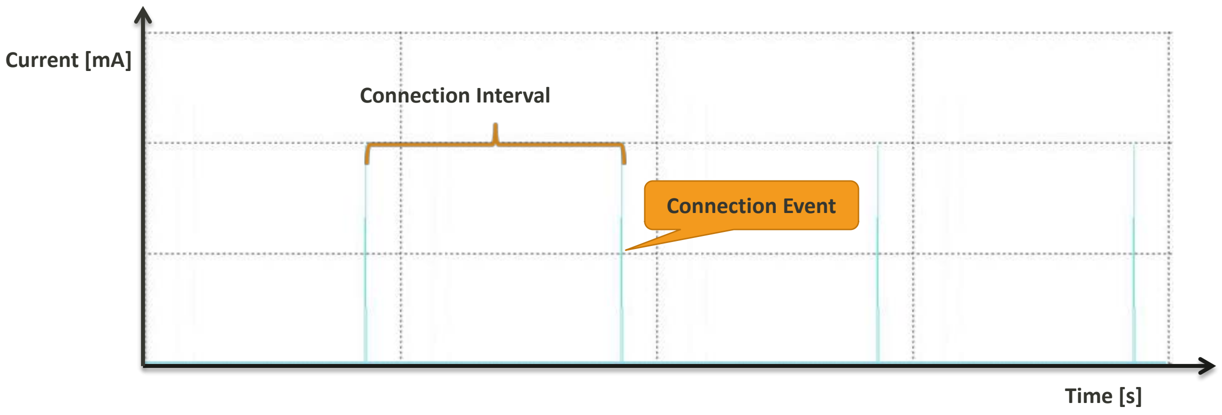

Figure 75. Connection interval and event

2) Connection Latency. Enables the peripheral device to skip a number of connection events. If the peripheral device has no data to send. It can skip no more than connection latency of connection events and stay asleep, as shown in Figure 76. The range of connection latency is from 0 to $\frac{conn\_timeout}{2*conn\_interval} - 1$. The value of the latency should be less than 311.875ms.

More details can be found in the header file `bt_hci_le.h`.

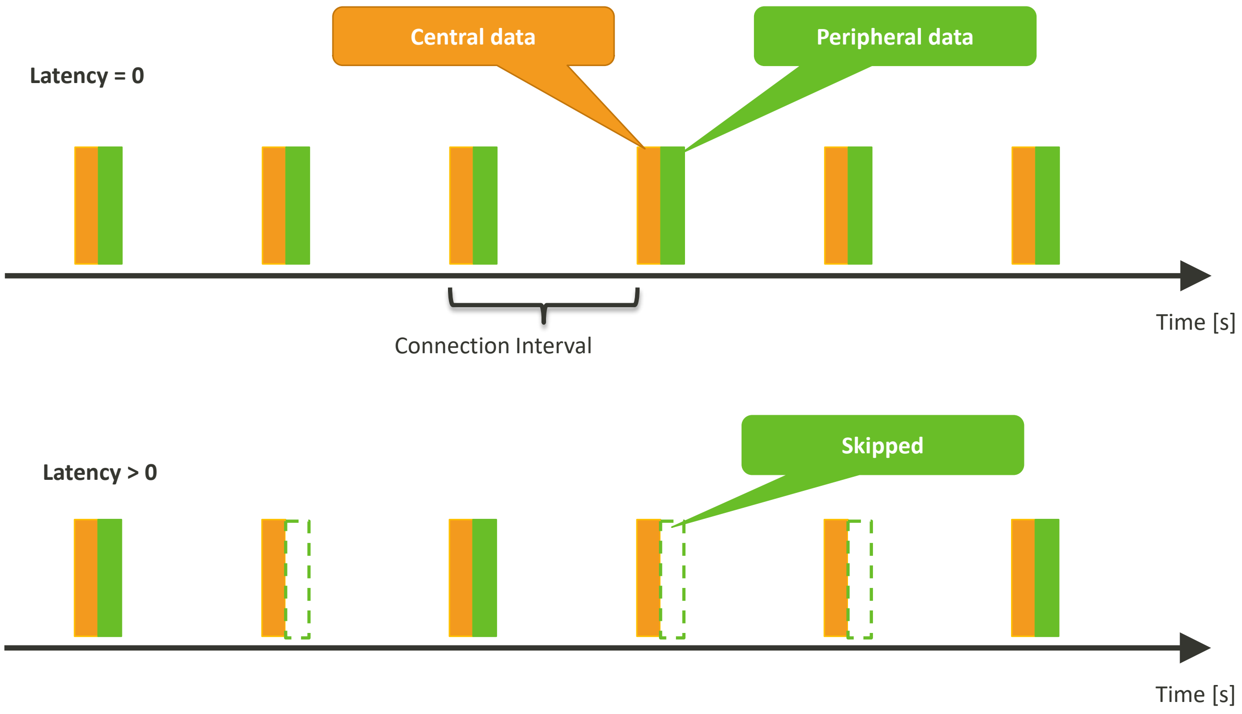

Figure 76. Slave latency

3) Connection Timeout. A connection might fail with an error code of `-0x08` due to not getting connection event data from the peer device before the connection times out. The range of the connection timeout is from 100ms to 32s. It defines the maximum time between two received connection data events before the connection is considered as failed, as shown in Figure 77.

More details can be found in the header file `bt_hci_le.h`.

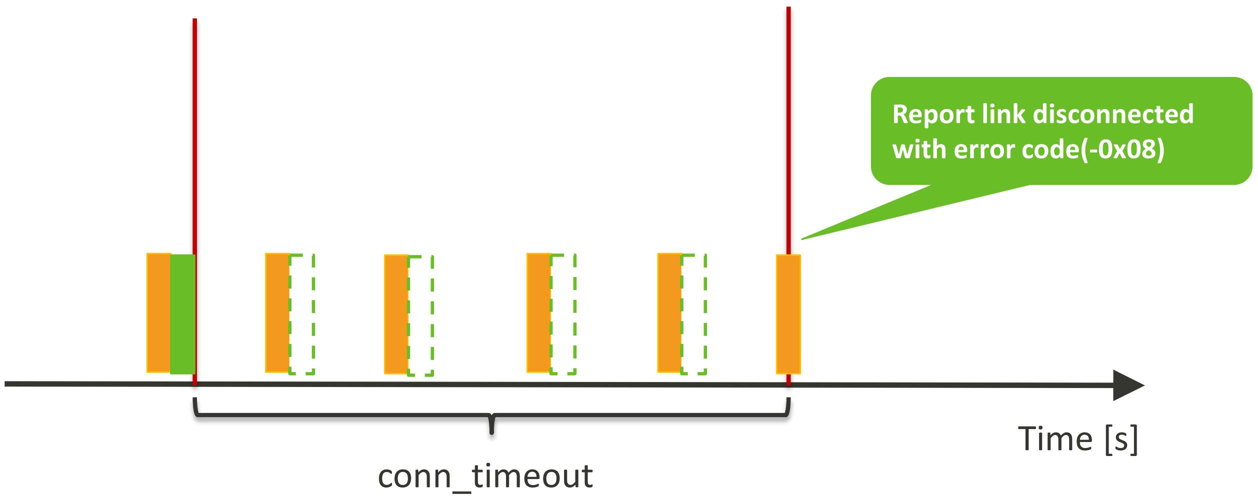

Figure 77. Connection timeout error occurred (error code -0x08)

If the master fails to receive six continuous packet data units (PDU) after the connection is created, the connection is considered lost. This enables fast termination of connections that fail to establish, as shown in Figure 78.

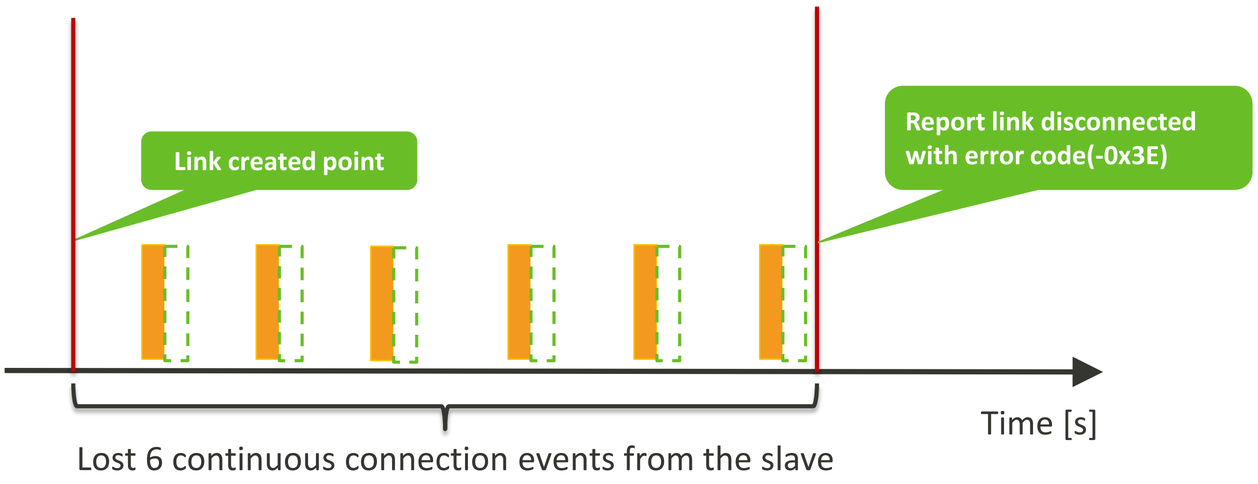
Link created point
Report link disconnected with error code(-0x3E)
Time [s]
Lost 6 continuous connection events from the slave

Figure 78. Connection timeout error occurred (error code -0x3E)

4.1.5. Resolvable private address resolution

This feature makes it more difficult for an attacker to track a device over a certain period of time. The user should enable address resolution and add the device to the resolving list in the controller.

4.1.5.1. Enable address resolution

Call the function `bt_gap_le_set_address_resolution_enable()` to enable the resolution of the resolvable private addresses in the controller.

4.1.5.2. Set resolvable private address timeout

Call the function `bt_gap_le_set_resolvable_private_address_timeout()` to set the resolvable private address timeout in the controller. The controller will generate a new resolvable private address and use it at timeout.

4.1.5.3. Set resolving list

Call the function `bt_gap_le_set_resolving_list()` to add a device identity to the resolving list, remove a device identity from the resolving list or clear the resolving list. The device identity contains the peer's identity address and local and peer's IRK pair.

```
bt_status_t app_add_dev_to_resolving_list(const bt_gap_le_bonding_info_t
*bonded_info)
{
    bt_status_t st = BT_STATUS_SUCCESS;
    if (BT_ADDR_TYPE_UNKNOW != bonded_info->identity_addr.address.type) {
        bt_hci_cmd_le_add_device_to_resolving_list_t dev;
        dev.peer_identity_address = bonded_info->identity_addr.address;
        memcpy(dev.peer_irk, &(bonded_info->identity_info),
               sizeof(dev.peer_irk));
        memcpy(dev.local_irk, &(local_key_req.identity_info),
               sizeof(dev.local_irk));
        st =
bt_gap_le_set_resolving_list(BT_GAP_LE_ADD_TO_RESOLVING_LIST,
                                          (void*)&dev);
    }
    return st;
}

bt_status_t app_delete_dev_from_resolving_list(const
bt_gap_le_bonding_info_t *bonded_info)
{
    bt_status_t st = BT_STATUS_SUCCESS;
    if (BT_ADDR_TYPE_UNKNOW != bonded_info->identity_addr.address.type) {
        bt_hci_cmd_le_remove_device_from_resolving_list_t dev;
        dev.peer_identity_address = bonded_info->identity_addr.address;
        st =
bt_gap_le_set_resolving_list(BT_GAP_LE_REMOVE_FROM_RESOLVING_LIST,
                                          (void*)&dev);
    }
    return st;
}

bt_status_t app_clear_resolving_list()
{
    bt_status_t st = BT_STATUS_SUCCESS;
    st = bt_gap_le_set_resolving_list(BT_GAP_LE_CLEAR_RESOLVING_LIST,
NULL);
    return st;
}
```

4.1.6. Read the RSSI

Call the function `bt_gap_le_read_rssi()` to read the Received Signal Strength Indication (RSSI) of a given connection. More details can be found in the Bluetooth Core Specifications version 4.2 [vol2, Part E, 7.5.4], Bluetooth GAP.

4.1.7. Update data length

Call the function `bt_gap_le_update_data_length()` to suggest data length for a given connection. More details can be found in the Bluetooth Core Specifications version 4.2 [vol2, Part E, 7.8.33], Bluetooth GAP.

4.1.8. Multiple advertising

The multiple advertising is supported on the MT7697 chip only; the objectives of multiple advertising are the following:

- Ability to support multiple non-connectable advertising
- Different advertising content and address
- An individualized response for each advertising
- Configurable transmission power (the range is from -70 dBm to 20 dBm)
- Ability to perform single (connectable) and multi-advertising (non-connectable), simultaneously, by calling the functions `bt_gap_le_set_advertising()` and `bt_gap_le_start_multiple_advertising()`, respectively.

With multiple advertising, different applications can start advertising without frequent modification of the advertising content. After it starts, the host can remain in sleep mode to save power. The application can start multiple advertising by using the multiple advertising APIs.

4.1.8.1. Limitations

- Multiple advertising does not support connectable type advertising.
- The `BT_HCI_ADV_TYPE_CONNECTABLE_DIRECTED_HIGH` and `BT_HCI_ADV_TYPE_CONNECTABLE_DIRECTED_HIGH` types of advertising are no longer supported by `bt_gap_le_set_advertising()`.

4.1.8.2. Get maximum advertising instances

Call the function `bt_gap_le_get_max_multiple_advertising_instances()` to get the maximum number of supported advertising instances, as shown below.

```
uint8_t max_instances =
bt_gap_le_get_max_multiple_advertising_instances();
```

4.1.8.3. Start multiple advertising

Set different advertising parameters and data for each advertising instance and start multiple advertising, as shown below. Multiple instances can start concurrently.

```
void app_start_instance_1(void)
{
    bt_bd_addr_t adv_addr = {0x13, 0x71, 0xDA, 0x7D, 0x1A, 0x0};
    bt_hci_cmd_le_set_advertising_parameters_t adv_para = {
        .advertising_interval_min = 0x0800,
        .advertising_interval_max = 0x0800,
        .advertising_type = BT_HCI_ADV_TYPE_SCANNABLE_UNDIRECTED,
        .advertising_channel_map = 7,
        .advertising_filter_policy = 0
    };
    bt_hci_cmd_le_set_advertising_data_t adv_data = {
        .advertising_data = {2, 1, 0x1A, 8, 9, 'a', 'd', 'v', 'o', 'n',
’e’,0},
        .advertising_data_length = 12,
        .};
    bt_hci_cmd_le_set_scan_response_data_t scan_res_data = {
        .scan_response data = {2, 1, 0x1A},
        .scan_response_data_length = 3,
        .};

    // Start advertising on instance 1. Each advertising instance needs to
start individually.
    bt_gap_le_start_multiple_advertising(
           1, /* Instance 1. The range is from 1 to (max_instance - 1). */
           5,
           (bt_bd_addr_ptr_t)adv_addr,
           &adv_para,
           &adv_data,
           &scan_res_data
           );
    return;
  }

void app_start_instance_2(void)
{
    bt_bd_addr_t adv_addr = {0x11, 0x11, 0x11, 0x11, 0x11, 0x11};
    bt_hci_cmd_le_set_advertising_parameters_t adv_para = {
        .advertising_interval_min = 0x0800,
        .advertising_interval_max = 0x0800,
        .advertising_type = BT_HCI_ADV_TYPE_SCANNABLE_UNDIRECTED,
        .advertising_channel_map = 7,
        .advertising_filter_policy = 0
    };
    bt_hci_cmd_le_set_advertising_data_t adv_data = {
        .advertising_data = {2,1,0x1A, 8, 9, ‘a’, ‘d’, ’v’,
’t’,’w’,’0’,0},
        .advertising_data_length = 12,
        .};
    bt_hci_cmd_le_set_scan_response_data_t scan_res_data = {
        .scan_response data = {2, 1, 0x1A},
        .scan_response_data_length = 3,
        .};

    // Start advertising on instance 2. Each advertising instance needs to
start individually.
    bt_gap_le_start_multiple_advertising(
           2, // Instance 2. The range is from 1 to (max_instance - 1).
           5,
           (bt_bd_addr_ptr_t)adv_addr,
           &adv_para,
           &adv_data,
           &scan_res_data
           );
    return;
  }

bt_status_t bt_app_event_callback(bt_msg_type_t msg, bt_status_t status,
void *buff)
{
    switch (msg) {
        case BT_GAP_LE_START_MULTIPLE_ADVERTISING_CNF:
        {
            bt_gap_le_start_multiple_advertising_cnf_t *event;
            event = (bt_gap_le_start_multiple_advertising_cnf_t *)buff;
            //Check the result of starting multi-advertising instance.
            break;
        }
    }
    return;
}
```

4.1.8.4. Stop multiple advertising

Call the function bt_gap_le_stop_multiple_advertising () to stop the advertising instance, as shown below. Multiple instances can stop concurrently.

```
void app_stop_instance(void)
{
    // Stop advertising instance 1. Each advertising instance needs to be
stopped individually.
    bt_gap_le_stop_multiple_advertising(1);
    return;
  }

bt_status_t bt_app_event_callback(bt_msg_type_t msg, bt_status_t status,
void *buff)
{
    switch (msg) {
        case BT_GAP_LE_START_MULTIPLE_ADVERTISING_CNF:
        {
            bt_gap_le_stop_multiple_advertising_cnf_t *event;
            event = (bt_gap_le_stop_multiple_advertising_cnf_t *)buff;
            //Check the result of stopping multi-advertising instance.
            break;
        }
    }
    return;
}
```

4.2. SM

The SM manages pairing, authentication and encryption between Bluetooth LE devices.

Pairing process has three phases. The first two are mandatory while the third one is optional.

1) Pairing Feature Exchange. Pairing feature includes I/O capability, Out-of-Band data flag, authentication requirements (bonding flag, man-in-the-middle (MITM) flag and Secure Connections (SC) flag), maximum encryption key size, initiator key distribution and responder key distribution. At this phase, only the I/O capability, Out-of-Band data flag and authentication requirements are configurable.
   a) The Bonding flag is set to false to delete the keys (described in step (d)), after disconnecting. If it's set to true, store the keys to the NAND flash so that they can be used again, when the two devices reconnect.
   b) The MITM flag and the SC flag are used to determine whether to use the MITM protection and LE SC Pairing separately or not.
   c) Maximum encryption key size is set to be 16 octets.

d) The key distribution in the pairing request from the initiator can be one of the following:

  i) LTK, EDIV, and Rand (only for legacy pairing)

  ii) IRK

  iii) CSRK

The responder will set the key distribution to be the minimum subset of Central and Peripheral devices.

  i) Short Term Key (STK) Generation for LE legacy pairing or LTK Generation for LE SC. The devices determine which of the following methods to use after exchanging I/O capabilities.

  ii) Just Works

  iii) Numeric Comparison (only for LE SC)

2) Passkey Entry. It varies depending on the I/O capability. One device displays the passkey and the other inputs the passkey. The passkey is generated randomly but can be modified to be fixed or variable before display.

3) Out-of-Band. The keys are not exchanged over the air, but rather over some other source such as serial port or near field communication (NFC). For more information about how to determine the method, refer to the Bluetooth core specifications version 4.2 [VOL 3, part H, 2.3.5.1].

4) Transport Specific Key Distribution. Phase 3 is optional to distribute specific keys, such as the Identity Address information and IRK. Phase 1 and Phase 3 are identical regardless of the method used in Phase 2.

Phase 3 is only performed on an encrypted link using the STK or LTK generated in Phase 2. Phase 1 and Phase 2 can be performed on a link with no encryption.

The keys distributed in Phase 3 need to be stored in the application. If the keys aren't stored, the pairing starts, otherwise the pairing is skipped and only the encryption is performed. Once the keys are stored for each device, the resolvable address needs to be considered when searching for the keys for that device. The resolvable address can be modified at any time. Keys are stored for a remote device with their resolvable addresses. The keys might change when the two devices are reconnected. In that case, during the search for keys, if both resolvable addresses can be resolved to the same identity address, the keys are found and can be used to encrypt the link so that the pairing process can be skipped.

Note. The Out-of-Band method is only supported in the LE legacy pairing, but not in the LE SC.

Encryption is implemented when the STK or LTK is generated. Usually the link is encrypted with the LTK again when reconnecting to the paired remote device. If the local device only initiates the encryption but the LTK doesn't exist on the remote device, then it's suggested to apply the pairing again.

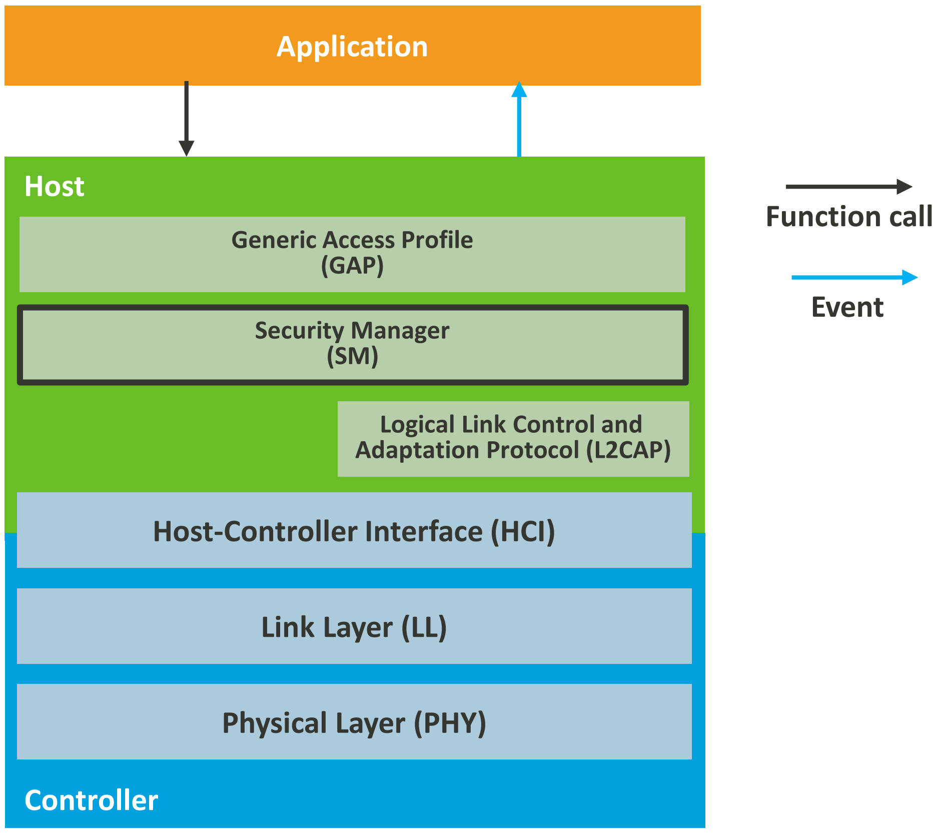
Application
Host
Generic Access Profile (GAP)
Security Manager (SM)
Logical Link Control and Adaptation Protocol (L2CAP)
Host-Controller Interface (HCI)
Link Layer (LL)
Physical Layer (PHY)
Controller
Function call
Event

Figure 79. SM abstraction layout

It's possible for the application and profiles to directly call GAP API functions to perform authentication and encryption after connection is established. GAP is based on SM, and the SM is based on L2CAP and HCI. GAP manages links and encapsulates the SM code, and the SM uses an L2CAP channel to send data to the remote device and receive data from the remote device, as shown in Figure 79.

4.2.1. The SM message sequences

This section introduces typical message sequences to provide more details about the events and procedures of the SM.

4.2.1.1. Just Works

The central role can initiate the pairing by calling the function `bt_gap_le_bond()`, and Just Works is used if the Out-of-Band flag of both devices is "`OOB not present`" and the I/O capability of one of these two devices is "`no input no output`" or the MITM flag of both devices is "`non-MITM`". An example where the I/O capabilities of both devices are "`no input no output`" is shown in Figure 80.

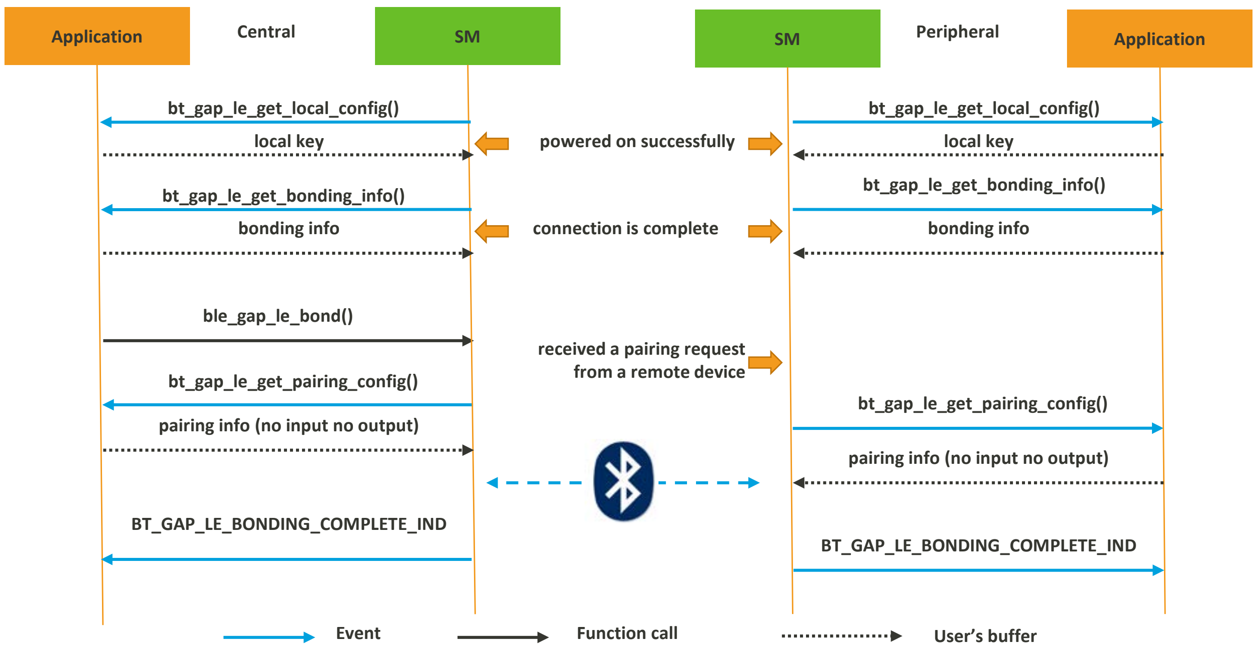
Application
Central
SM
SM
Peripheral
Application
bt_gap_le_get_local_config()
local key
powered on successfully
bt_gap_le_get_local_config()
local key
bt_gap_le_get_bonding_info()
bonding info
connection is complete
bt_gap_le_get_bonding_info()
bonding info
ble_gap_le_bond()
received a pairing request from a remote device
bt_gap_le_get_pairing_config()
pairing info (no input no output)
bt_gap_le_get_pairing_config()
pairing info (no input no output)
BT_GAP_LE_BONDING_COMPLETE_IND
BT_GAP_LE_BONDING_COMPLETE_IND
Event
Function call
User's buffer

Figure 80. Just Works (central role)

1) The peripheral role can request the central role to initiate pairing by calling the function `bt_gap_le_bond()`, and Just Works is used if the Out-of-Band flag of both devices is "`OOB not present`" and the I/O capability of one of these two devices is "`no input no output`" or the MITM flag of both devices is "`non-MITM`". An example where the I/O capabilities of both devices are "`no input no output`" is shown in Figure 81.

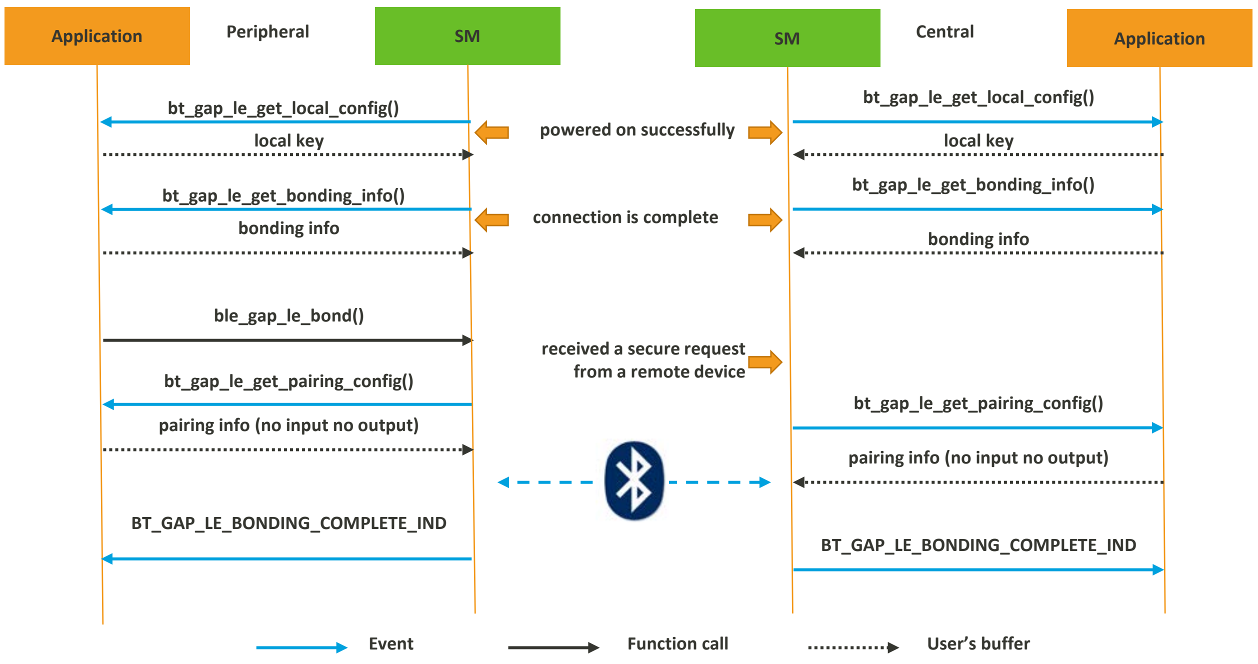
Application
Peripheral
SM
SM
Central
Application
bt_gap_le_get_local_config()
local key
powered on successfully
bt_gap_le_get_local_config()
local key
bt_gap_le_get_bonding_info()
bonding info
connection is complete
bt_gap_le_get_bonding_info()
bonding info
ble_gap_le_bond()
received a secure request from a remote device
bt_gap_le_get_pairing_config()
pairing info (no input no output)
bt_gap_le_get_pairing_config()
pairing info (no input no output)
BT_GAP_LE_BONDING_COMPLETE_IND
BT_GAP_LE_BONDING_COMPLETE_IND
Event
Function call
User's buffer

Figure 81. Just Works (peripheral role)

4.2.1.2. Numeric Comparison (only for LE SC)

The central role can initiate the pairing by calling the function `bt_gap_le_bond()`, and passkey entry is used if the Out-of-Band flag of both devices is "`OOB not present`", the MITM flag of at least one device is "`has MITM`" and the I/O capability of one of these two devices is "`Display Yes No`". An example is shown in Figure 82.

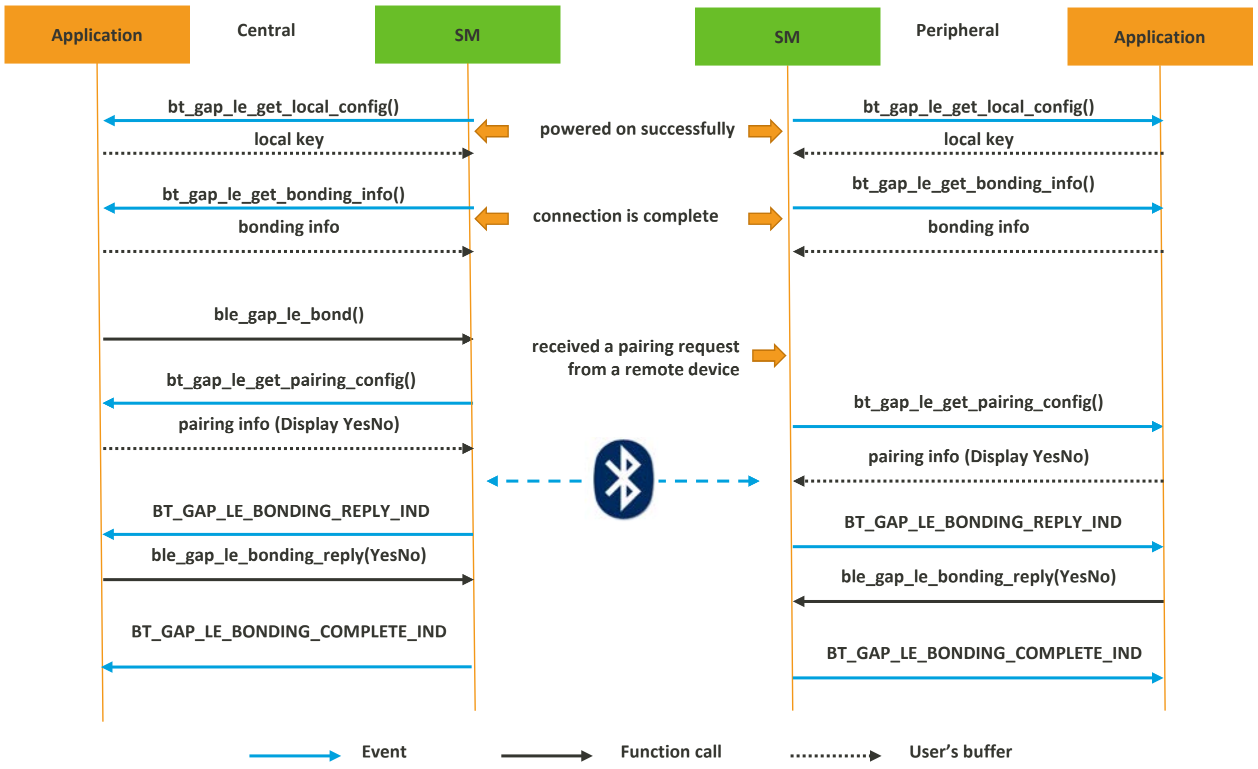
Application
Central
SM
SM
Peripheral
Application
bt_gap_le_get_local_config()
local key
powered on successfully
bt_gap_le_get_local_config()
local key
bt_gap_le_get_bonding_info()
bonding info
connection is complete
bt_gap_le_get_bonding_info()
bonding info
ble_gap_le_bond()
received a pairing request from a remote device
bt_gap_le_get_pairing_config()
pairing info (Display YesNo)
bt_gap_le_get_pairing_config()
pairing info (Display YesNo)
BT_GAP_LE_BONDING_REPLY_IND
ble_gap_le_bonding_reply(YesNo)
BT_GAP_LE_BONDING_REPLY_IND
ble_gap_le_bonding_reply(YesNo)
BT_GAP_LE_BONDING_COMPLETE_IND
BT_GAP_LE_BONDING_COMPLETE_IND
Event
Function call
User's buffer

Figure 82. Numeric Comparison (central role)

The peripheral role can request the central role to initiate the pairing by calling the function `bt_gap_le_bond()` and passkey entry is used if the Out-of-Band flag of both devices is "`OOB not present`", the MITM flag of at least one of the devices is "`has MITM`", the I/O capability of one of these two devices is "`Display Yes No`".

An example is shown in Figure 83.

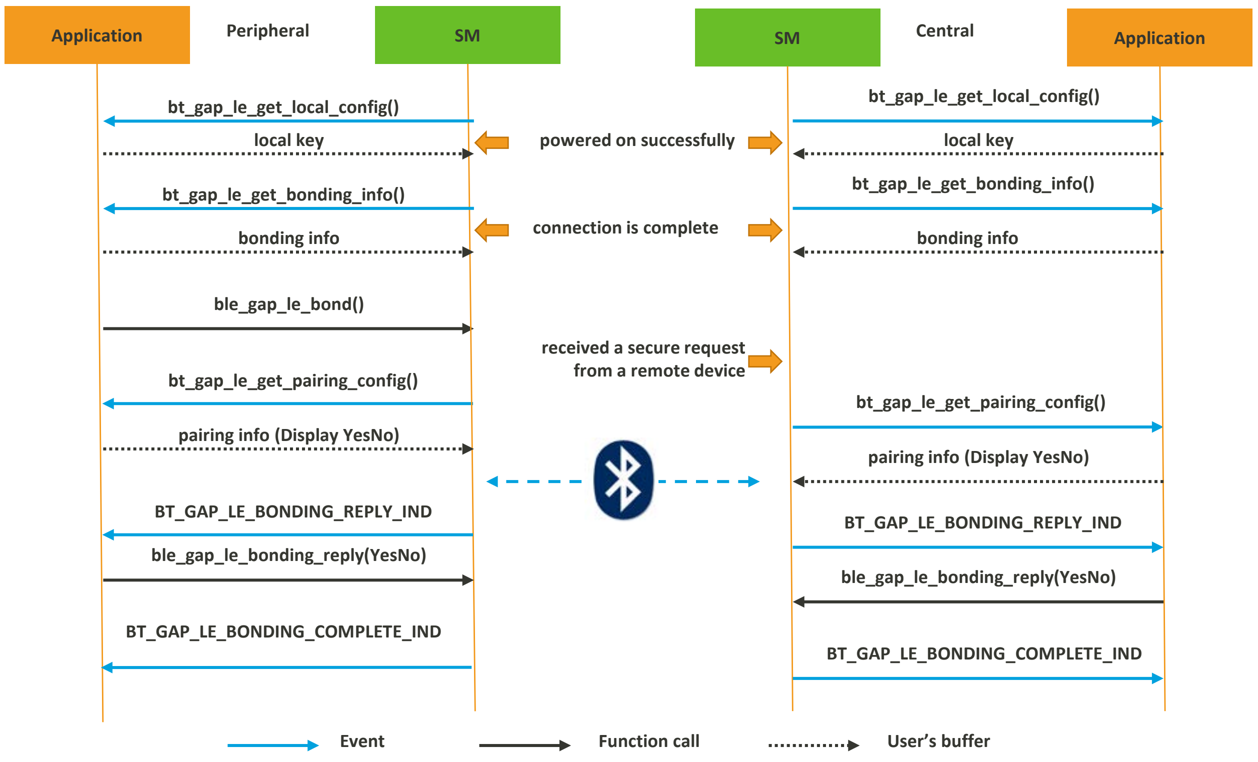
Application
Peripheral
SM
SM
Central
Application
bt_gap_le_get_local_config()
local key
powered on successfully
bt_gap_le_get_bonding_info()
bonding info
connection is complete
ble_gap_le_bond()
received a secure request from a remote device
bt_gap_le_get_pairing_config()
pairing info (Display YesNo)
BT_GAP_LE_BONDING_REPLY_IND
ble_gap_le_bonding_reply(YesNo)
BT_GAP_LE_BONDING_COMPLETE_IND
Event
Function call
User's buffer

Figure 83. Numeric Comparison (peripheral role)

4.2.1.3. Passkey entry

1) The central role can initiate the pairing by calling the function `bt_gap_le_bond()`, and passkey entry is used if the Out-of-Band flag of both devices is "`OOB not present`", the MITM flag of at least one device is "`has MITM`" and the I/O capabilities of these two devices are any except "`no input no output`" and "`display only`". An example where the I/O capabilities of both devices are "`keyboard only`" is shown in Figure 84.

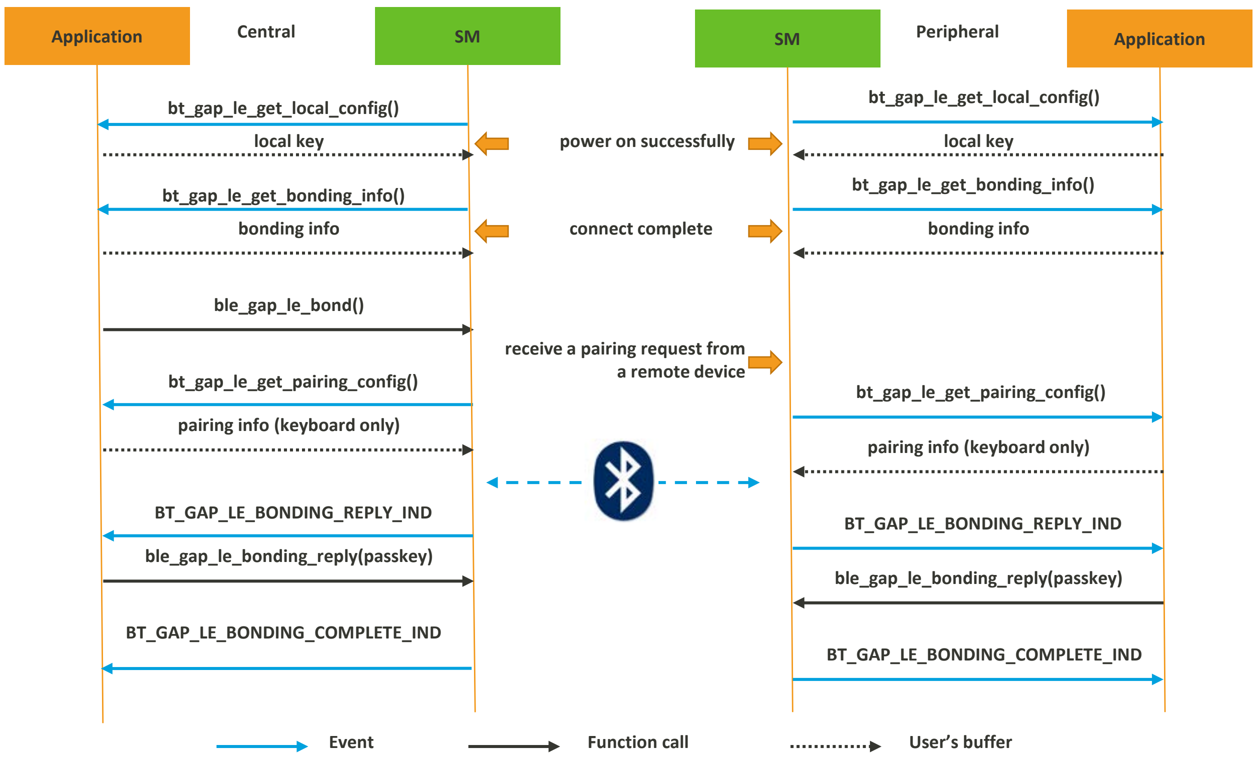
Application
Central
SM
SM
Peripheral
Application
bt_gap_le_get_local_config()
local key
power on successfully
bt_gap_le_get_local_config()
local key
bt_gap_le_get_bonding_info()
bonding info
connect complete
bt_gap_le_get_bonding_info()
bonding info
ble_gap_le_bond()
receive a pairing request from a remote device
bt_gap_le_get_pairing_config()
pairing info (keyboard only)
bt_gap_le_get_pairing_config()
pairing info (keyboard only)
BT_GAP_LE_BONDING_REPLY_IND
ble_gap_le_bonding_reply(passkey)
BT_GAP_LE_BONDING_REPLY_IND
ble_gap_le_bonding_reply(passkey)
BT_GAP_LE_BONDING_COMPLETE_IND
BT_GAP_LE_BONDING_COMPLETE_IND
Event
Function call
User's buffer

Figure 84. Passkey entry (central role)

2) The peripheral role can request the central role to initiate pairing by calling the function `bt_gap_le_bond()` and passkey entry is used if the Out-of-Band flag of both devices is “`OOB not present`”, the MITM flag of at least one of the devices is “`has MITM`”, the I/O capabilities of these two devices are anything except “`no input no output`” and “`display only`”. An example where the I/O capabilities of both devices are “`keyboard only`” is shown in Figure 85.

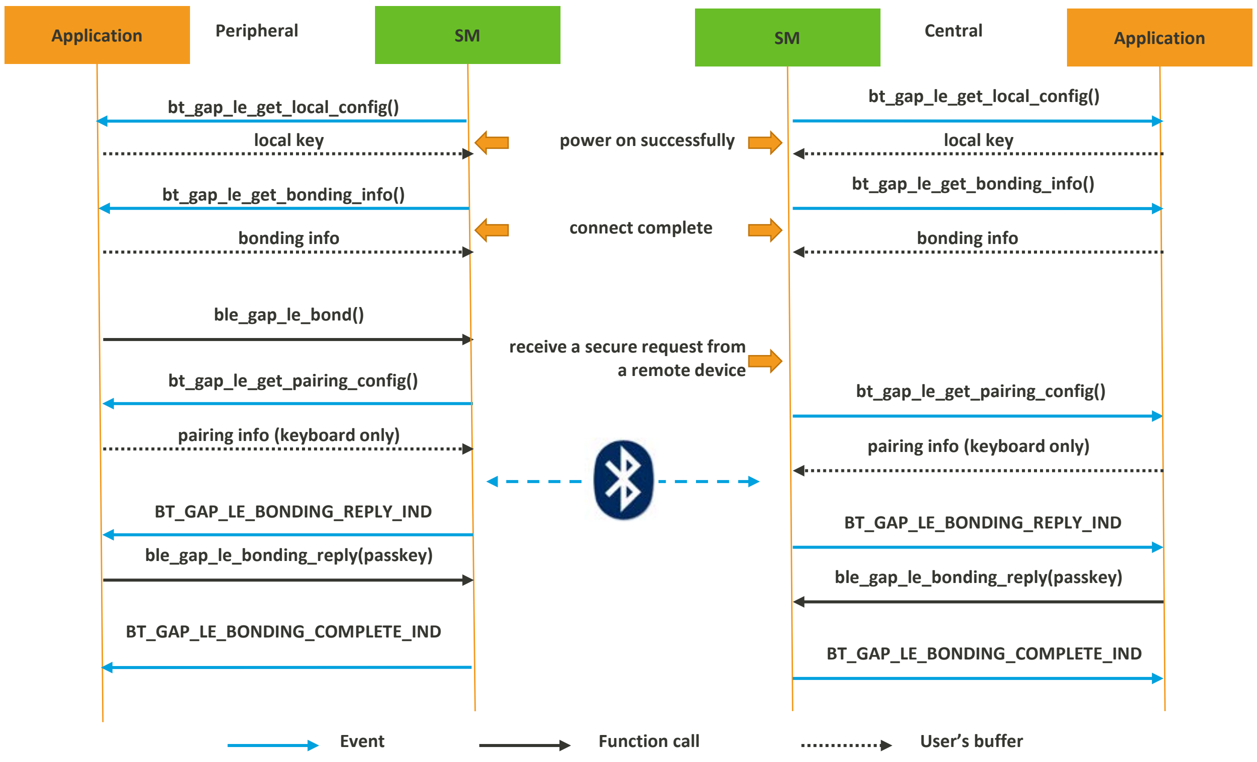
Application
Peripheral
SM
SM
Central
Application
bt_gap_le_get_local_config()
local key
power on successfully
bt_gap_le_get_bonding_info()
bonding info
connect complete
ble_gap_le_bond()
receive a secure request from a remote device
bt_gap_le_get_pairing_config()
pairing info (keyboard only)
BT_GAP_LE_BONDING_REPLY_IND
ble_gap_le_bonding_reply(passkey)
BT_GAP_LE_BONDING_COMPLETE_IND
Event
Function call
User's buffer

Figure 85. Passkey entry (peripheral role)

4.2.1.4. Encryption

1) The central role requests to initiate the encryption by calling the function `bt_gap_le_bond()`. After the pairing is complete, the link is already encrypted with the STK in LE legacy pairing or LTK in LE S C pairing. Usually the encryption with the LTK is performed if higher security level is required or after reconnecting. An example where the LTK is used to encrypt the link is shown in Figure 86.

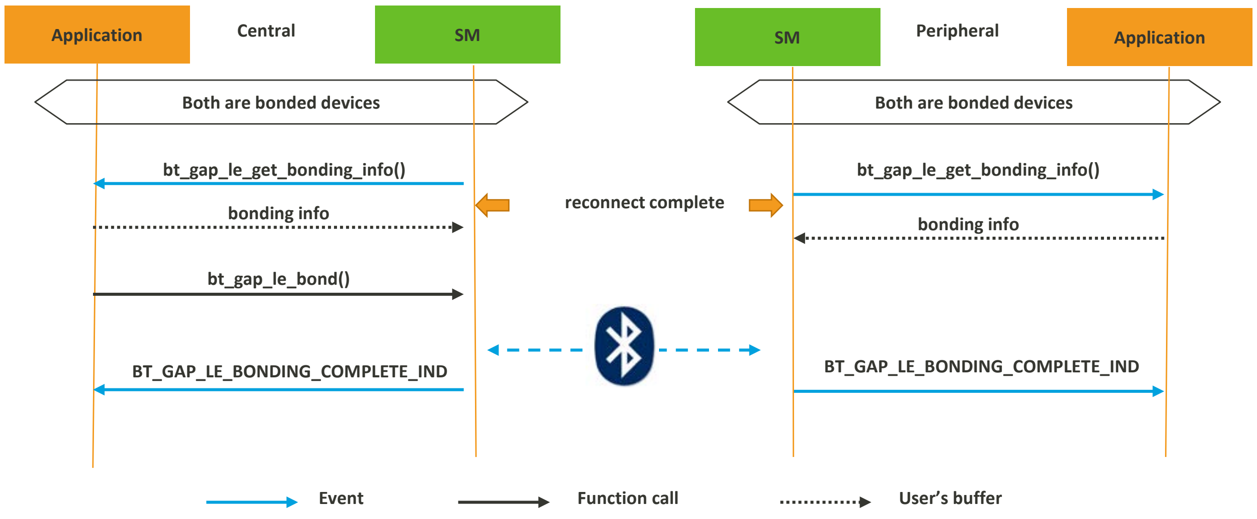

Figure 86. Encryption (central role)

2) The peripheral role initiates the encryption by calling the function `bt_gap_le_bond()`. After the pairing is complete, the link is already encrypted with STK in LE legacy pairing or LTK in LE SC pairing. Usually the

encryption with the LTK is performed if higher security level is required or after reconnecting. An example where the LTK is used for link encryption is shown in Figure 87.

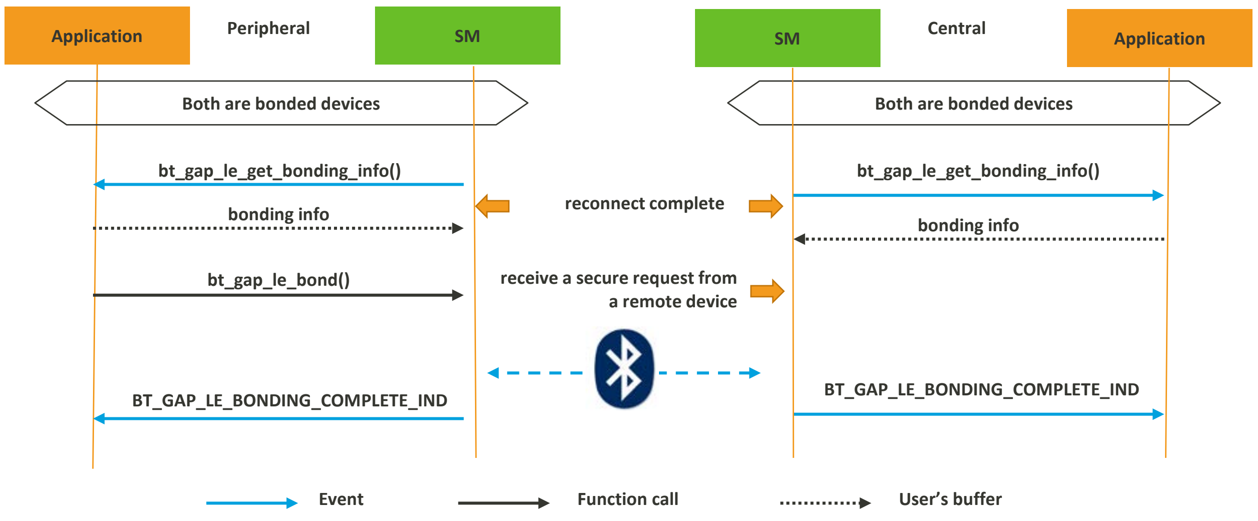

Figure 87. Encryption (peripheral role)

4.2.2. Using the SM APIs

The SM API headers can be found in the `bt_gap_le.h` header file with only two interface APIs for pairing, `bt_gap_le_bond()` and `bt_gap_le_bonding_reply()`. Apply the following function calls to use the SM in your application development.

1) Optional, call the `bt_gap_le_bond()` API to pair with a remote device and perform the encryption once the pairing is finished. Also call the `bt_gap_le_bond()` API to encrypt only, if these two devices are still bonded, which is usually performed after reconnecting with the bonded remote device.

```
bt_status_t status;
bt_smp_pairing_config_t pairing_config = {//mitm, bond, oob
.maximum_encryption_key_size = 16,
.io_capability = BT_SMP_DISPLAY_YES_NO,
.auth_req = BT_SMP_AUTH_REQ_MITM | BT_SMP_AUTH_REQ_SECURE_CONNECTION,
};
uint32_t connection_handle = 0x0200;
status = bt_gap_le_bond(connection_handle, & pairing_config);
```

2) Optional, call the `bt_gap_le_bonding_reply()` API to send the Out-of-Band data for the Out-of-Band pairing method or passkey for the Passkey Input pairing method (see 4.2.1.3, "Passkey entry"), or Numeric Comparison result (see 4.2.1.2, "Numeric Comparison (only for LE SC)"), when `BT_GAP_LE_BONDING_REPLY` event is received.

```
bt_status_t status;
bt_gap_le_bonding_reply_t rsp = {.passkey = 123456};
uint32_t connection_handle = 0x0200;
status = bt_gap_le_bonding_reply(connection_handle, &rsp);
```

3) Mandatory, implement the `bt_app_event_callback ()` API to handle the events sent by the stack.

```
bt_status_t bt_app_event_callback(bt_msg_type_t msg, bt_status_t status,
void *buff)
{
    bt_status_t status;
    switch (msg) {
        case BT_GAP_LE_BONDING_REPLY_IND: {
            //Reply OOB data or Passkey or Numeric Comparison result, call
bt_gap_le_bonding_reply() API.
        }
        case BT_GAP_LE_BONDING_COMPLETE_IND: {
            // indicates a pairing procedure has finished.
        }

        default:
            break;
    }
    return status;
}
```

4) Mandatory, implement the `bt_gap_le_get_local_config()` API to return the local configuration field to stack. All configuration fields must be a global buffer.

```
static bt_gap_le_local_config_req_ind_t g_app_local_config;
bt_gap_le_local_key_t local_key_req = {
    .encryption_info = {0};
    .master_id = {0};
    .identity_info =
{0x12,0x34,0x56,0x78,0x9a,0xbc,0xde,0xf0,0x19,0x28,0x55,0x33,0x68,0x33,0x5
6,0xde};
    .signing.info = {0};
};
Bool sc_omly = false;
bt_gap_le_local_config_req_ind_t bt_gap_le_get_local_config(void)
{
    g_app_local_config.local_key_req = &local_key_req;
    g_app_local_config.sc_only_mode_req = sc_only;

    return &g_app_local_config;
}
```

5) Mandatory, implement the `bt_gap_le_get_bonding_info()` API to return the bonding info field to stack. Bonding info field must be a global buffer.

```
static bt_gap_le_bonding_info_t bongding_info = {0};
bt_gap_le_bonding_info_t bt_gap_le_get_bonding_info(const bt_addr_t
remote_addr)
{
    return &bongding_info;
}
```

6) Mandatory, implement the `bt_gap_le_get_pairing_config()` API to return the pairing info field to stack. Pairing info field must be a global buffer.

```
static bt_gap_le_smp_pairing_config_req = {
    .auth_req = BT_GAP_LE_SMP_AUTH_REQ_BONDING,
    .maximum_encryption_key_size = 16,
    .io_capability = BT_GAP_LE_SMP_NO_INPUT_NO_OUTPUT,
};
bt_status_t bt_gap_le_get_pairing_config(bt_gap_le_bonding_start_ind_t
*ind)
{
    ind->pairing_config_req = pairing_config_req;
    return BT_STATUS_SUCCESS;
}
```

4.3. GATT

GATT describes a service framework using the Attribute Protocol (ATT) to discover, read, write, notify and indicate characteristics, as well as to configure the broadcast of characteristics. GATT is designed for an application or another profile to communicate with a peer device. There are two roles defined for devices that implement this profile, as shown in Figure 88.

- **Client** — the device that sends commands and requests to the server and can receive responds, indications and notifications from the server.
- **Server** — the device that accepts commands and requests from the client and sends responds, indications and notifications to a client.

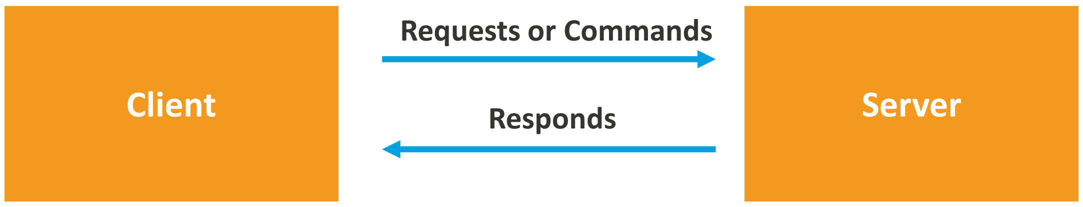

Figure 88. GATT Client and Server

The ATT request-response or indication-confirmation pair is considered as a single transaction. The two roles are not fixed to the device, but are determined when a device initiates a transaction, and are released when the transaction ends. A device can act in both roles at the same time.

A profile should have one or more services. The service is composed of one or more characteristics or included services. Each characteristic contains a value and may contain optional information known as descriptors of the value, as shown in Figure 89. All information (service, characteristic and the components of the characteristic) is stored in **Attributes** on the **Server**.

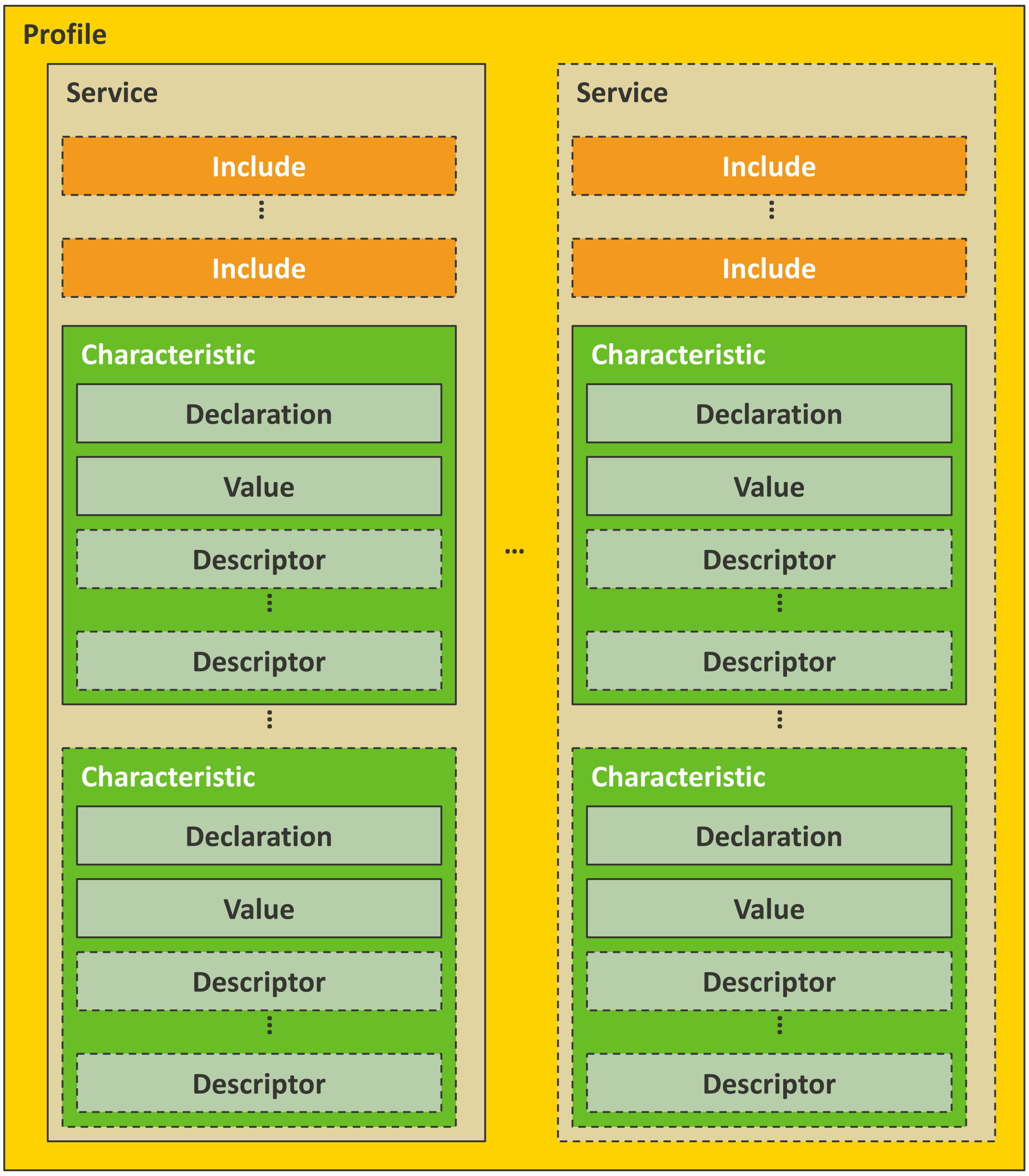

Figure 89. GATT profile hierarchy

The **Attribute** consists of: **Attribute Handle**, **Attribute Type**, **Attribute Value** and **Attribute Permissions**, as shown in Figure 90.

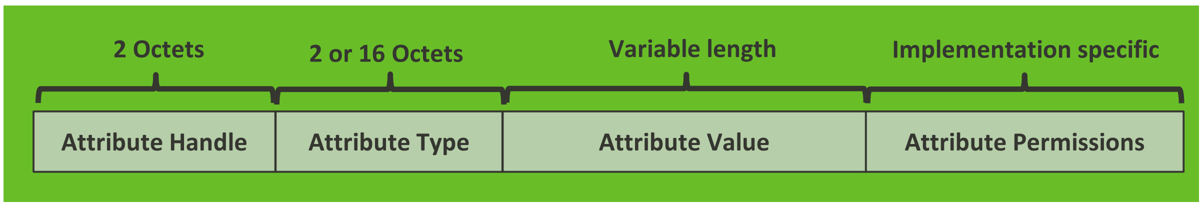

Figure 90. Logical representation of the attribute

The **Attribute Handle** is an index corresponding to a specific **Attribute**. The **Attribute Type** is a UUID that describes the **Attribute Value**. The **Attribute Value** is the data described by the **Attribute Type** and indexed by the **Attribute Handle**. The **Attributes** are ordered by ascending **Attribute Handle** values. The **Attribute Handle** values are in a range from 0x0001 to 0xFFFF. **Attribute Permissions** are part of the **Attribute** that cannot be read from or written to using ATT. The permissions are used by the **Server** to determine whether read or write access is permitted for a given **Attribute**.

4.3.1. Create a GATT Server

The GATT Server stores and manages an application-wide attribute table. This section describes the process to create a user-defined GATT Server.

GATT Server's database is divided into three levels, from top to bottom, the server, the service and the record. Each server can have more than one service. Each service can have more than one record.

An example of a GATT Server that has two services, `bt_if_gap_service` with four records and `bt_if_gatt_service` with five records, is shown below.

```
|server
    |service
        |handle|record|
|bt_if_gatt_server
    |bt_if_gap_service
        |0x0001|bt_if_gap_primary_service|
        |0x0002|bt_if_gap_included_manufacturer_service|
        |0x0003|bt_if_gap_char4_dev_name|
        |0x0004|bt_if_gap_dev_name|
    |bt_if_gatt_service
        |0x0011|bt_if_gatt_primary_service|
        |0x0012|bt_if_gatt_char4_service_changed|
        |0x0013|bt_if_gatt_service_changed|
        |0x0014|bt_if_gatt_client_config|
        |0x0015|bt_if_gatt_server_config|
```

Follow the steps below to implement GATT Server with the defined services and records.

1) Implement a callback is required to handle the device name read and write requests when creating a device name characteristic, as shown below.

```
char gatts_device_name[256]={"MTKHB"};
static uint32_t bt_if_gap_dev_name_callback (const uint8_t rw, uint16_t
handle, void *data, uint16_t size, uint16_t offset)
{
    uint32_t str_size = strlen(gatts_device_name);
    uint32_t buf_size = sizeof(gatts_device_name);
    uint32_t copy_size;
    switch (rw) {
        case BT_GATTS_CALLBACK_READ:
            /* To handle read request. */
            copy_size = (str_size> offset)?(str_size-offset):0;
            if (size==0){
                return str_size;
            }
            copy_size = (size > copy_size)? copy_size:size;
            memcpy(data, gatts_device_name+offset, copy_size);
            return copy_size;
        case BT_GATTS_CALLBACK_WRITE:
            /* To handle write request. */
            copy_size = (size > buf_size)? buf_size:size;
            memcpy(gatts_device_name, data, copy_size);
            return copy_size;
        default:
            return BT_STATUS_SUCCESS;
    }
}
```

2) Create the four (4) records of the GAP service and store them in the bt_if_gap_service_rec, as shown below.

```
/* To create records of GAP primary service. */
enum {
    GAP_HANDLE_START = 0x0001,
    GAP_HANDLE_PRIMARY_SERVICE = GAP_HANDLE_START,
    GAP_HANDLE_INCLUDED_MANUFACTURER_SERVICE,
    GAP_HANDLE_CHAR4_DEV_NAME,
    GAP_HANDLE_DEV_NAME,
    GAP_HANDLE_END
};
/* The record of primary service. */
BT_GATTS_NEW_PRIMARY_SERVICE_16(bt_if_gap_primary_service,
    BT_GATT_UUID16_GAP_SERVICE);
/* The record of included service. */
BT_GATTS_NEW_INCLUDED_SERVICE_128(bt_if_gap_included_manufacturer_service,
                                 0x0500, 0x0522);
/* The record of device name characteristic declaration. */
BT_GATTS_NEW_CHARC_16_WRITABLE(bt_if_gap_char4_dev_name,
     BT_GATT_CHARC_PROP_READ, GAP_HANDLE_DEV_NAME,
     BT_SIG_UUID16_DEVICE_NAME);
/* The record of device name characteristic value. */
BT_GATTS_NEW_CHARC_VALUE_CALLBACK(bt_if_gap_dev_name,
     BT_SIG_UUID_DEVICE_NAME,
     BT_GATTS_REC_PERM_READABLE|BT_GATTS_REC_PERM_WRITABLE,
     bt_if_gap_dev_name_callback);
/*  The bt_if_gap_service_rec to hold all records of GAP service. */
static const bt_gatts_service_rec_t *bt_if_gap_service_rec[] = {
    (const bt_gatts_service_rec_t*) &bt_if_gap_primary_service,
    (const bt_gatts_service_rec_t*)
&bt_if_gap_include_manufacturer_service,
    (const bt_gatts_service_rec_t*) &bt_if_gap_char4_dev_name,
    (const bt_gatts_service_rec_t*) &bt_if_gap_dev_name
};
```

3) Create a GAP service named as bt_if_gap_service and stored in bt_if_gatt_server, as shown below.

```
/* The value of the starting_handle is assigned to the first record:
bt_if_gap_primary_service.
 The value of the ending_handle is assigned to the last record:
bt_if_gap_dev_name.
(ending_handle - starting_handle + 1) should equal to the number of
records.
The required_encryption_key_size defines the encryption key size this
service requires. If record's perm is set as
BT_GATTS_REC_PERM_READABLE_ENCRYPTION or
BT_GATTS_REC_PERM_WRITABLE_ENCRYPTION, GATTS will check if the channel is
encrypted and the encryption key size is greater than or equal to
required_encryption_key_size. */
static const bt_gatts_service_t bt_if_gap_service = {
    .starting_handle = GAP_HANDLE_START,
    .ending_handle = GAP_HANDLE_END - 1,
    .required_encryption_key_size = 7,
     /* If record needs channel encryption, the size of encryption key
should be equal to or greater than 7. */
    .records = bt_if_gap_service_rec
  };
```

4) Implement a callback is required to handle the client characteristic configuration read and write requests when creating the client characteristic configuration descriptor.

```
static uint16_t bt_if_gatt_notify=0; //Client Characteristic Configuration
static uint32_t bt_if_gatt_client_config_callback (const uint8_t rw,
                  uint16_t handle, void *data, uint16_t size, uint16_t
offset)
{
    if (rw == BT_GATTS_CALLBACK_WRITE){
        /* To handle write request. */
        if (size != sizeof(bt_if_gatt_notify)){ //Size check
            return 0;
        }
        bt_if_gatt_notify = *(uint16_t*)data;
    } else {
        /* To handle read request. */
        if (size!=0){
            uint16_t *buf = (uint16_t*) data;
            *buf = bt_if_gatt_notify;
        }
    }
    return sizeof(bt_if_gatt_notify);
}
```

5) Create five (5) records of the GATT service stored in `bt_if_gatt_service_rec`, as shown below.

```
/* To create records of GATT primary service. */
enum {
    GATT_HANDLE_START = 0x0011,
    GATT_HANDLE_PRIMARY_SERVICE = GATT_HANDLE_START,
    GATT_HANDLE_CHAR4_SERVICE_CHANGED,
    GATT_HANDLE_SERVICE_CHANGED,
    GATT_HANDLE_CLIENT_CONFIG,
    GATT_HANDLE_END
};
BT_GATTS_NEW_PRIMARY_SERVICE_16(bt_if_gatt_primary_service,
    BT_GATT_UUID16_GATT_SERVICE);
/* The record of service changed characteristic declaration. */
BT_GATTS_NEW_CHARC_16(bt_if_gatt_char4_service_changed,
    BT_GATT_CHARC_PROP_READ|BT_GATT_CHARC_PROP_NOTIFY|
    BT_GATT_CHARC_PROP_INDICATE,GATT_HANDLE_SERVICE_CHANGED,
    BT_SIG_UUID16_SERVICE_CHANGED);
/* The record of service changed characteristic value. */
BT_GATTS_NEW_CHARC_VALUE_UINT32_WRITABLE(bt_if_gatt_service_changed,
    BT_SIG_UUID_SERVICE_CHANGED, 0x2, 0x0001050F);
/* The record of client configuration descriptor. */
BT_GATTS_NEW_CLIENT_CHARC_CONFIG(bt_if_gatt_client_config,
    BT_GATTS_REC_PERM_READABLE|BT_GATTS_REC_PERM_WRITABLE,
    bt_if_gatt_client_config_callback);

static const bt_gatts_service_rec_t *bt_if_gatt_service_rec[] = {
     (const bt_gatts_service_rec_t*) &bt_if_gatt_primary_service,
     (const bt_gatts_service_rec_t*) &bt_if_gatt_char4_service_changed,
     (const bt_gatts_service_rec_t*) &bt_if_gatt_service_changed,
     (const bt_gatts_service_rec_t*) &bt_if_gatt_client_config
};
```

6) Create a GATT service named as `bt_if_gatt_service` and store it in `bt_if_gatt_server`, as shown below.

```
/* Value of starting_handle will be assigned to first record:
bt_if_gatt_primary_service.
 Value of ending_handle will be assigned to end record:
bt_if_gatt_client_config.
(ending_handle - starting_handle + 1) should equal to number of records.
The required_encryption_key_size is used to define encryption key size
this  service requires. If record's perm was setted with
BT_GATTS_REC_PERM_READABLE_ENCRYPTION or
BT_GATTS_REC_PERM_WRITABLE_ENCRYPTION,  GATTS will check if the channel is
encrypted and the encryption key size is greater than or equal to
required_encryption_key_size. */

static const bt_gatts_service_t bt_if_gatt_service = {
     .starting_handle = GATT_HANDLE_START,
     .ending_handle = GATT_HANDLE_END - 1,
     .required_encryption_key_size = 9,
     /* If record need channel encryption, the size of encryption key
should be equal to or greater than 9. */
     .records = bt_if_gatt_service_rec
};
```

7) Create a GATT Server named as `bt_if_gatt_server` to collect all services and implement

```
/* - Create bt_if_gatt_server and collect all services. */
// Server collects all service.
const bt_gatts_service_t * bt_if_gatt_server[] = {
    &bt_if_gap_service,
    &bt_if_gatt_service_ro,
    NULL
};
/* This server should be provided to GATTS via bt_get_gatt_server(). */
const bt_gatts_service_t** bt_get_gatt_server()
{
    return bt_if_gatt_server;
}
```

`bt_get_gatt_server()` to return the final constructed server.

8) If service or characteristic want to use 128 bit UUID, please refer to the following sample code.

```
/* - How to Add characteristic with 128 bit UUID. */
/* If uuid is 16 bit, convert it as 128 bit. */
bt_uuid_t uuid128_xxx = BT_UUID_INIT_WITH_UUID16(uuid16_xxx);
/* Define 128 UUID characteristic, and add to service as before. */
BT_GATTS_NEW_CHARC_128(name, prop, _handle, uuid128_xxx);

/* - How to add a secondary service with 128bit UUID. */
/* Define a secondary service with 128 bit UUID, and some characteristic
for this service */
#define BT_SIG_UUID_VENDOR_SERVICE
{{0x12,0x34,0x56,0x78,0x90,0x13,0x57,0x91,0x24,0x68,0x82,0x11,0x22,0x33,0x
44,0x55}}
BT_GATTS_NEW_SECONDARY_SERVICE_128(bt_if_vendor_secondary_service,BT_SIG_U
UID_VENDOR_SERVICE);
BT_GATTS_NEW_CHARC_16(bt_if_vendor_char1_manufacture_name, ...) ;
```

```
......
/* Add service records array into gatt server "bt_if_gatt_server[]" */
static const bt_gatts_service_rec_t *bt_if_vendor_service_rec[] = {
    (const bt_gatts_service_rec_t*) &bt_if_vendor_secondary_service,
    (const bt_gatts_service_rec_t*) &bt_if_vendor_char1_manufacture_name,
    ......
};
static const bt_gatts_service_t bt_if_vendor_service = {
    .starting_handle = 0x0500,
    .ending_handle = 0x0522,
    .required_encryption_key_size = 16,
    .records = bt_if_vendor_service_rec
};
const bt_gatts_service_t * bt_if_gatt_server[] = {
    &bt_if_gap_service,//0x0001
    &bt_if_gatt_service_ro,//0x0010
    &bt_if_bas_service,//0x0020
    &bt_if_vendor_service,//0x500
    NULL
};
/* Define include service which start/end handle same with secondary
service, and add include service into primary service. */
BT_GATTS_NEW_INCLUDED_SERVICE_128(bt_if_included_vendor_secondary_service,
0x0500, 0x0522);
static const bt_gatts_service_rec_t *bt_if_bas_service_rec[] = {
  (const bt_gatts_service_rec_t*) &bt_if_bas_primary_service,
  (const bt_gatts_service_rec_t*)&bt_if_included_vendor_secondary_service,
   ......
};
```

4.3.1.1. GATT Server callbacks

- Implement the `bt_gatts_get_authorization_check_result()` API, if the GATT Server requires to check for an attribute authorization. The API confirms with a user whether to give authorization to a specified attribute. If the server accepts peer access to this attribute, the returned value is BT_STATUS_SUCCESS; otherwise, the returned value is BT_STATUS_UNSUPPORTED.
- If the GATT server has a long attribute, `bt_gatts_get_execute_write_result()` should be implemented, to confirm with the server whether execute write request completed successfully. If the execute write request completed successfully, the returned value is BT_STATUS_SUCCESS; otherwise, the returned value is BT_ATT_ERRCODE_XXX.

4.3.2. Using the GATT APIs

This section describes how to use the GATT APIs in your application development. The APIs can be found in `bt_gattc.h` and `bt_gatts.h` header files.

Most of the GATT APIs require a connection handle as the first parameter obtained from listening to the BLE_GAP_CONNECT_IND event from the `bt_app_event_callback()` callback function.

4.3.2.1. Exchanging MTU

Initialize the Exchange MTU only once during a connection. The message flow for Exchange MTU is shown in Figure 91. More details can be found in the `bt_gattc.h` header file.

The default MTU size is 23. Neither client nor server should use MTU size of less than 23. When the client supports a value greater than the default MTU, the client should inform the server of the client's maximum receive MTU size by calling the function `bt_gattc_exchange_mtu()`.

The server can set the maximum MTU size it can support by calling `bt_gatts_set_max_mtu()`. Once the messages are exchanged, the final MTU between the two devices is set to the minimum of the client receive MTU and server receive MTU values. The GATT client or server can call the function `bt_gattc_get_mtu()` to get the current MTU.

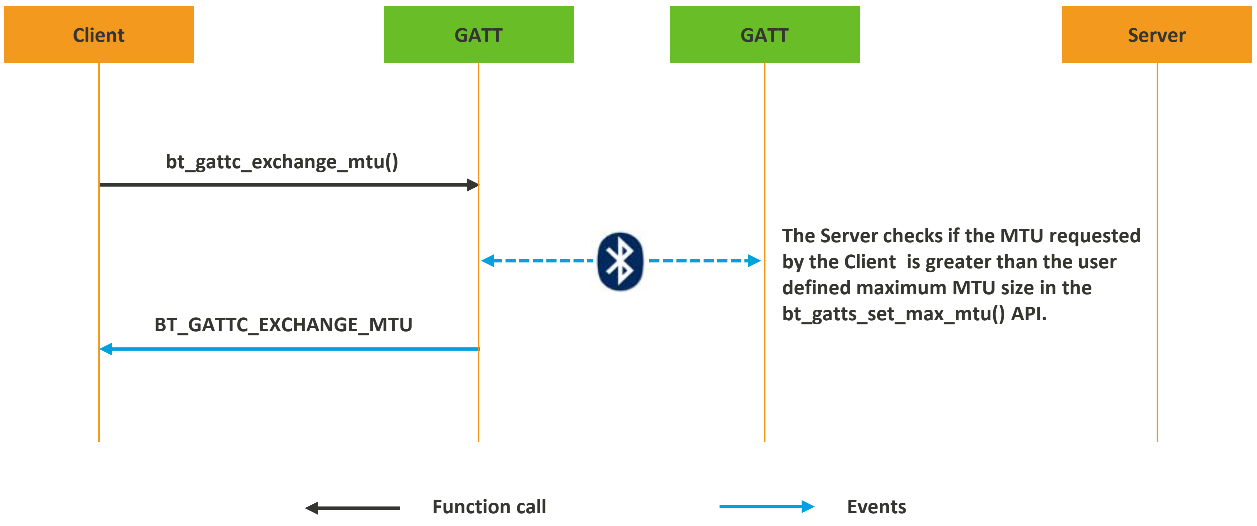
Client
GATT
GATT
Server
bt_gattc_exchange_mtu()
The Server checks if the MTU requested by the Client is greater than the user defined maximum MTU size in the bt_gatts_set_max_mtu() API.
BT_GATTC_EXCHANGE_MTU
Function call
Events

Figure 91. Exchange MTU event sequence

An example implementation to Exchange MTU:

```
/* BLE devices is connected, and conn_handle has been obtained from
BLE_GAP. */
BT_GATTC_NEW_EXCHANGE_MTU_REQ(req);
bt_gattc_exchange_mtu(conn_handle, &req);

void bt_app_event_callback(bt_msg_type_t msg, bt_status_t status, void
*buff)
{
  switch (msg) {
  case BT_GATTC_EXCHANGE_MTU: {
    if (status != BT_STATUS_SUCCESS) {
      printf ("BT_GATTC_EXCHANGE_MTU FINISHED!!");
      break;
    }
    bt_gatt_exchange_mtu_rsp_t *rsp = (bt_gatt_exchange_mtu_rsp_t *)buff;
    printf("mtu=%d", rsp->server_rx_mtu);
    break;
  }
}
```

4.3.2.2. Primary service discovery

The Primary service discovery message flow is shown in Figure 92. More details can be found in the `bt_gattc.h` header file.

There are two APIs `bt_gattc_discover_primary_service()` and `bt_gattc_discover_primary_service_by_uuid()` used by a client to discover primary services on a server.

Once the primary services are discovered, additional information about the primary services can be accessed using other APIs.

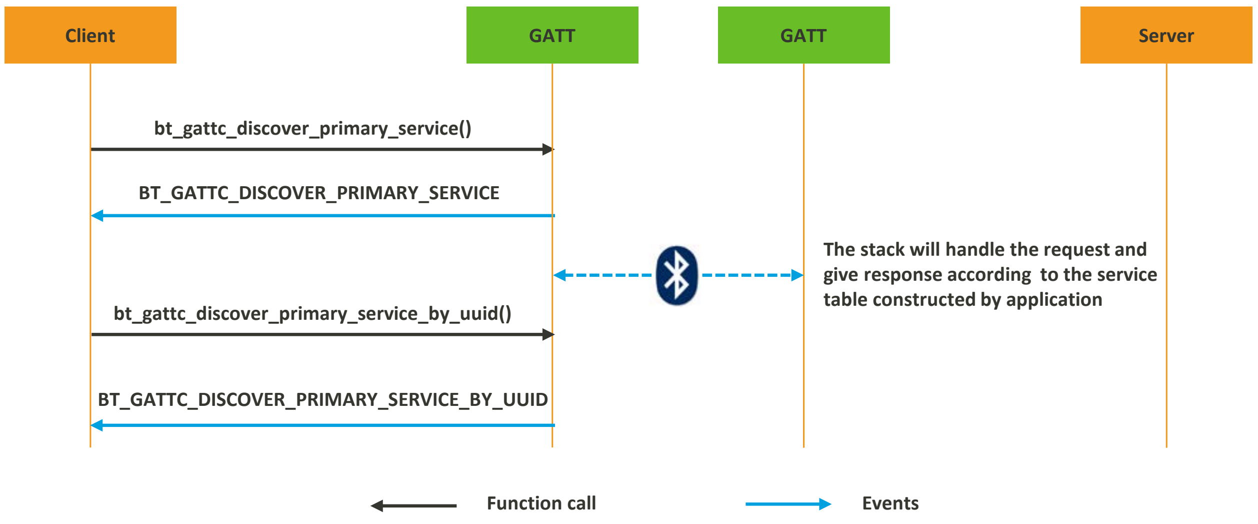
Client
GATT
GATT
Server
bt_gattc_discover_primary_service()
BT_GATTC_DISCOVER_PRIMARY_SERVICE
The stack will handle the request and give response according to the service table constructed by application
bt_gattc_discover_primary_service_by_uuid()
BT_GATTC_DISCOVER_PRIMARY_SERVICE_BY_UUID
Function call
Events

Figure 92. Primary service discovery event sequence

These two APIs will discover all primary services between the given starting handle and ending handle and then notify the upper layer about each received response.

An example implementation to discover a primary service:

```
/* BLE devices is connected, and conn_handle has been obtained from
BLE_GAP. */
BT_GATTC_NEW_DISCOVER_PRIMARY_SERVICE_REQ(req);
bt_gattc_discover_primary_service(conn_handle, &req);

void bt_app_event_callback(bt_msg_type_t msg, bt_status_t status, void
*buff)
{
  switch (msg) {
  case BT_GATTC_DISCOVER_PRIMARY_SERVICE: {
    if (status != BT_STATUS_SUCCESS && status != BT_ATT_ERRCODE_CONTINUE)
{
      printf("BT_GATTC_DISCOVER_PRIMARY_SERVICE FINISHED!!");
      break;
    }
    bt_gattc_read_by_group_type_rsp_t *rsp =
(bt_gattc_read_by_group_type_rsp_t *)buff;
    uint16_t end_group_handle = 0, starting_handle = 0, uuid = 0;
    bt_uuid_t uuid128;
    uint8_t *attribute_data_list = rsp->att_rsp->attribute_data_list;
    uint8_t num_of_data = (rsp->length - 2) / rsp->att_rsp->length;
    int i, j;
    for (i = 0; i < num_of_data; i++) {
      memcpy(&starting_handle,
             attribute_data_list + i * rsp->att_rsp->length, 2);
      memcpy(&end_group_handle,
             attribute_data_list + i * rsp->att_rsp->length + 2, 2);
      if (rsp->att_rsp->length == 6) {
        memcpy(&uuid, attribute_data_list + i * rsp->att_rsp->length + 4,
               rsp->att_rsp->length - 4);
      } else {
        memcpy(&uuid128.uuid, attribute_data_list+i*rsp->att_rsp->length +
4,
                   rsp->att_rsp->length - 4);
      }
    }
    break;
  }
}
```

To discover a primary service by the UUID:

```
/* BLE devices is connected, and conn_handle has been obtained from
BLE_GAP. */
BT_GATTC_NEW_DISCOVER_PRIMARI_SERVICE_BY_UUID16_REQ(req, 0xABCD);
bt_gattc_discover_primary_service_by_uuid(conn_handle, &req);

void bt_app_event_callback(bt_msg_type_t msg, bt_status_t status, void
*buff)
{
  switch (msg) {
  case BT_GATTC_DISCOVER_PRIMARY_SERVICE_BY_UUID: {
    if (status != BT_STATUS_SUCCESS && status != BT_ATT_ERRCODE_CONTINUE)
{
      printf("BT_GATTC_DISCOVER_PRIMARY_SERVICE BY UUID FINISHED!!");
      break;
    }
    bt_gattc_find_by_type_value_rsp_t *rsp =
                        (bt_gattc_find_by_type_value_rsp_t *)buff;
    uint16_t attribute_handle = 0, end_group_handle = 0;
    uint8_t *handles_info_list = rsp->att_rsp->handles_info_list;
    uint8_t num_of_info = (rsp->length - 1) / 4;
    int i;
    for (i = 0 ; i < num_of_info; i++) {
      memcpy(&attribute_handle, handles_info_list + 4 * i, 2);
      memcpy(&end_group_handle, handles_info_list + 4 * i + 2, 2);
    }
    break;
  }
}
```

When the server receives a Primary service discovery request, the stack handles this message and sends a response to the client. The server only constructs a profile as described in section 4.3.1, “Create a GATT Server”.

4.3.2.3. Find included services

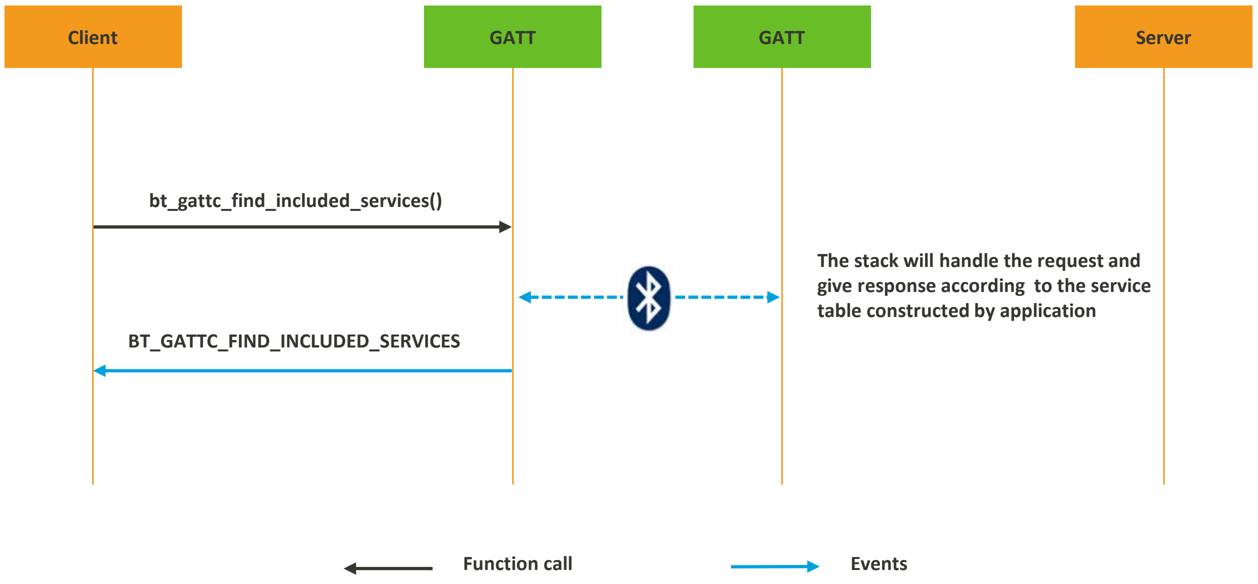
Client
GATT
GATT
Server
bt_gattc_find_included_services()
BT_GATTC_FIND_INCLUDED_SERVICES
The stack will handle the request and give response according to the service table constructed by application
Function call
Events

Figure 93. Find included services event sequence

The Find included services message flow is shown in Figure 93. More details can be found in the `bt_gattc.h` header file.

Call the function `bt_gattc_find_included_services()` to find the included services. The response is in `BT_GATTC_FIND_INCLUDED_SERVICES`.

An example implementation to find included services:

```
/* BLE devices is connected, and conn_handle has been obtained from
BLE_GAP. */
BT_GATTC_NEW_FIND_INCLUDED_SERVICE_REQ(req, 0x0001, 0x0010);
bt_gattc_find_included_services(conn_handle, &req);

void bt_app_event_callback(bt_msg_type_t msg, bt_status_t status, void
*buff)
{
  switch (msg) {
  case BT_GATTC_FIND_INCLUDED_SERVICES: {
    if (status != BT_STATUS_SUCCESS && status != BT_ATT_ERRCODE_CONTINUE)
{
      printf("BT_GATTC_FIND_INCLUDED_SERVICES FINISHED!!");
      break;
    }
    bt_gattc_read_by_type_rsp_t *rsp = (bt_gattc_read_by_type_rsp_t
*)buff;
    uint8_t *attribute_data_list = rsp->att_rsp->attribute_data_list;
    uint16_t attribute_handle = 0, starting_handle = 0, ending_handle = 0;
    uint16_t uuid = 0;
    bt_uuid_t uuid128;
    int i;

    if (rsp->att_rsp->length <= 8) {
      uint8_t num_of_data = (rsp->length - 2) / rsp->att_rsp->length;
      for (i = 0 ; i < num_of_data; i++) {
        memcpy(&attribute_handle,
               attribute_data_list + rsp->att_rsp->length*i, 2);
        memcpy(&starting_handle,
               attribute_data_list + rsp->att_rsp->length*i + 2, 2);
        memcpy(&ending_handle,
               attribute_data_list + rsp->att_rsp->length*i + 4, 2);
        memcpy(&uuid, attribute_data_list + rsp->att_rsp->length*i + 6,
2);
      }
    } else {
      memcpy(&attribute_handle, attribute_data_list, 2);
      memcpy(&starting_handle, attribute_data_list + 2, 2);
      memcpy(&ending_handle, attribute_data_list + 4, 2);
      memcpy(&uuid128.uuid, attribute_data_list + 6, 16);
    }
    break;
  }
}
```

When the server receives Find included services request, the stack handles this message and responds to the client according to the profile constructed by the application, as described in section 4.3.1, “Create a GATT Server”.

4.3.2.4. Characteristic discovery

The Characteristic discovery message flow is shown in Figure 94. More details can be found in the `bt_gattc.h` header file.

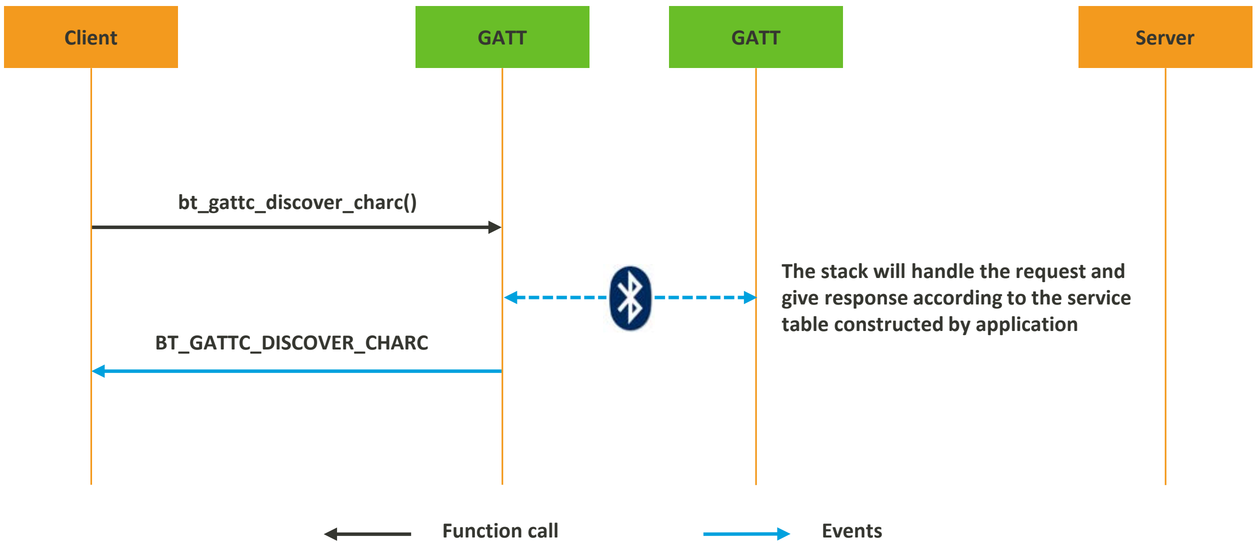
Client
GATT
GATT
Server
bt_gattc_discover_charc()
BT_GATTC_DISCOVER_CHARC
The stack will handle the request and give response according to the service table constructed by application
Function call
Events

Figure 94. Discover characteristic event sequence

Call the function `bt_gattc_discover_charc()` for characteristic discovery. The response is in the BT_GATTC_DISCOVER_CHARC.

An example implementation for the Characteristic discovery:

```
   /* BLE devices is connected, and conn_handle has been obtained from
   BLE_GAP. */
   BT_GATTC_NEW_DISCOVER_CHARC_REQ(req, 0x0001, 0x0010);
   bt_gattc_discover_charc(conn_handle, &req);

   void bt_app_event_callback(bt_msg_type_t msg, bt_status_t status, void
   *buff)
   {
     switch (msg) {
     case BT_GATTC_DISCOVER_CHARC: {
       if (status != BT_STATUS_SUCCESS && status != BT_ATT_ERRCODE_CONTINUE)
   {
         printf("BT_GATTC_DISCOVER_CHARC FINISHED!!");
         break;
       }
       bt_gattc_read_by_type_rsp_t *rsp = (bt_gattc_read_by_type_rsp_t
   *)buff;
       uint8_t *attribute_data_list = rsp->att_rsp->attribute_data_list;
       uint16_t attribute_handle = 0;
       uint8_t attribute_value[30] = {0};
       uint8_t num_of_data = (rsp->length - 2) / rsp->att_rsp->length;
       bt_uuid_t uuid128;
       int i, j;
       if (rsp->att_rsp->length < 20) {
         for (i = 0 ; i < num_of_data; i++) {
           memcpy(&attribute_handle,
                  attribute_data_list + i * rsp->att_rsp->length, 2);
           memcpy(&attribute_value, attribute_data_list+i*rsp->att_rsp-
   >length+2,
                  rsp->att_rsp->length - 2);
         }
       } else {
         memcpy(&attribute_handle, attribute_data_list, 2);
         memcpy(&attribute_value, attribute_data_list + 3, 2);
         memcpy(&uuid128.uuid, attribute_data_list + 5, 16);
       }
       break;
     }
   }
```

When the server receives a Characteristic discovery request, the stack handles this message and responds to the client according to the profile constructed by the application, as described in section 4.3.1, “Create a GATT Server”.

4.3.2.5. Characteristic descriptor discovery

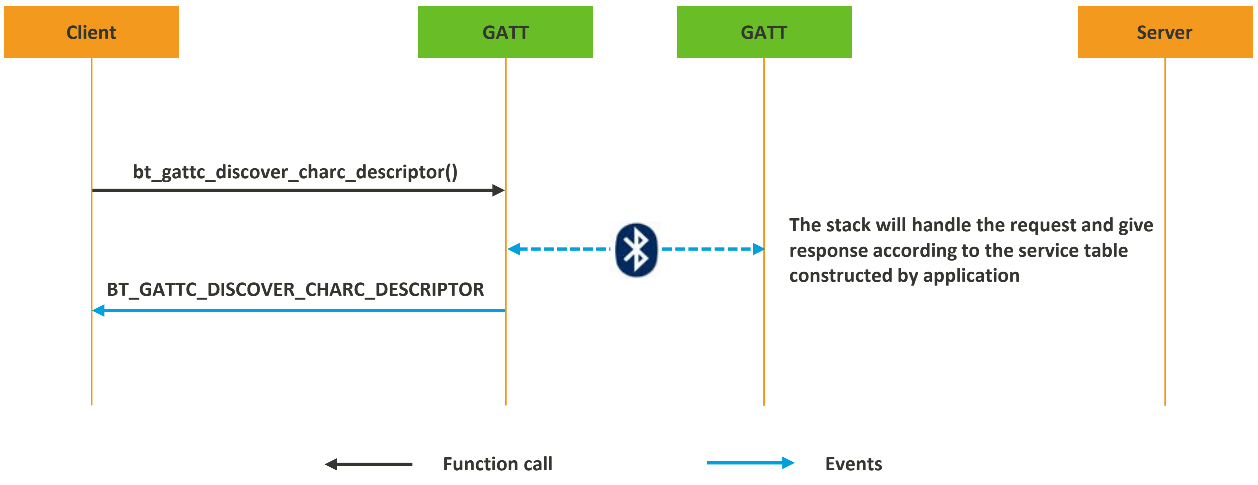
Client
GATT
GATT
Server
bt_gattc_discover_charc_descriptor()
BT_GATTC_DISCOVER_CHARC_DESCRIPTOR
The stack will handle the request and give response according to the service table constructed by application
Function call
Events

Figure 95. Characteristic descriptor discovery event sequence

The Characteristic descriptor discovery message flow is shown in Figure 95. More details can be found in the header file `bt_gattc.h`.

`bt_gattc_discover_charc_descriptor()` is used by a client to find all the characteristic descriptors' **Attribute Handles** and **Attribute Types** within a characteristic definition when only the characteristic handle range is known. The start handle should be set to the handle of the specified characteristic value + 1, and the end handle should be set to the ending handle of the specified characteristic or service. The response will be stored in BT_GATTC_DISCOVER_CHARC_DESCRIPTOR.

An example implementation for the Characteristic descriptor discovery:

```
/* BLE devices is connected, and conn_handle has been obtained from
BLE_GAP. */
BT_GATTC_NEW_DISCOVER_CHARC_DESCRIPTOR_REQ(req, 0x0003, 0x0005);
bt_gattc_discover_charc_descriptor(conn_handle, &req);

void bt_app_event_callback(bt_msg_type_t msg, bt_status_t status, void
*buff)
{
  switch (msg) {
  case BT_GATTC_DISCOVER_CHARC_DESCRIPTOR: {
    if (status != BT_STATUS_SUCCESS && status != BT_ATT_ERRCODE_CONTINUE)
{
      printf("BT_GATTC_DISCOVER_CHARC_DESCRIPTOR FINISHED!!");
      break;
    }
    bt_gattc_find_info_rsp_t *rsp = (bt_gattc_find_info_rsp_t *)buff;
    uint8_t format = rsp->att_rsp->format;
    uint16_t attribute_handle = 0, attribute_value = 0;
    uint8_t attribute_length = 0;
    uint8_t num_of_attribute;
    bt_uuid_t uuid128;
    int i, j;
    if ( format == 0x02 ) {
      /* 128-bit UUID. */
      attribute_length = 18;
      num_of_attribute = (rsp->length - 2) / attribute_length;
      for (i = 0; i < num_of_attribute; ++i) {
        memcpy(&attribute_handle,
               rsp->att_rsp->info_data + i * attribute_length, 2);
        memcpy(&uuid128,
               rsp->att_rsp->info_data + i * attribute_length + 2, 16);
      }
    } else {
      /* 16-bit UUID. */
      attribute_length = 4;
      num_of_attribute = (rsp->length - 2) / attribute_length;
      memcpy(&attribute_handle,
              rsp->att_rsp->info_data+(num_of_attribute-
1)*attribute_length, 2);
      memcpy(&attribute_value,
             rsp->att_rsp->info_data+(num_of_attribute-
1)*attribute_length+2, 2);
    }
    break;
  }
}
```

When the server receives the Characteristic descriptor discovery request, the stack handles this message and sends a response to the client according to the profile constructed by the application, as described in section 4.3.1, "Create a GATT Server".

4.3.2.6. Read the characteristic value

There are four APIs to read a characteristic value, `bt_gattc_read_charc()`, `bt_gattc_read_long_charc()`, `bt_gattc_read_using_charc_uuid()` and `bt_gattc_read_multi_charc_values()`.

1) Call the function `bt_gattc_read_charc()` to read a characteristic value from the server, if the client knows the characteristic value handle only. The handle parameter should be set to the characteristic value handle. The BT_GATTC_READ_CHARC returns the characteristic value.
2) Call the function `bt_gattc_read_long_charc()` to read the characteristic value, if the client knows the characteristic value handle and the length of the characteristic value is more than MTU – 1. The Bluetooth stack will continuously send read blob request until all the values of the characteristic are read. It will provide a response in BT_GATTC_READ_LONG_CHARC one by one, user is responsible to merge them together.
3) Call the function `bt_gattc_read_using_charc_uuid()` to read a characteristic value, if the client knows only the characteristic UUID and does not know the handle of the characteristic.
4) Call the function `bt_gattc_read_multi_charc_values()` to read multiple characteristic values in one request, if the handle parameter is set to the characteristic value handles.

The message flow to read the characteristic value is shown in Figure 96. More details can be found in the `bt_gattc.h` header file.

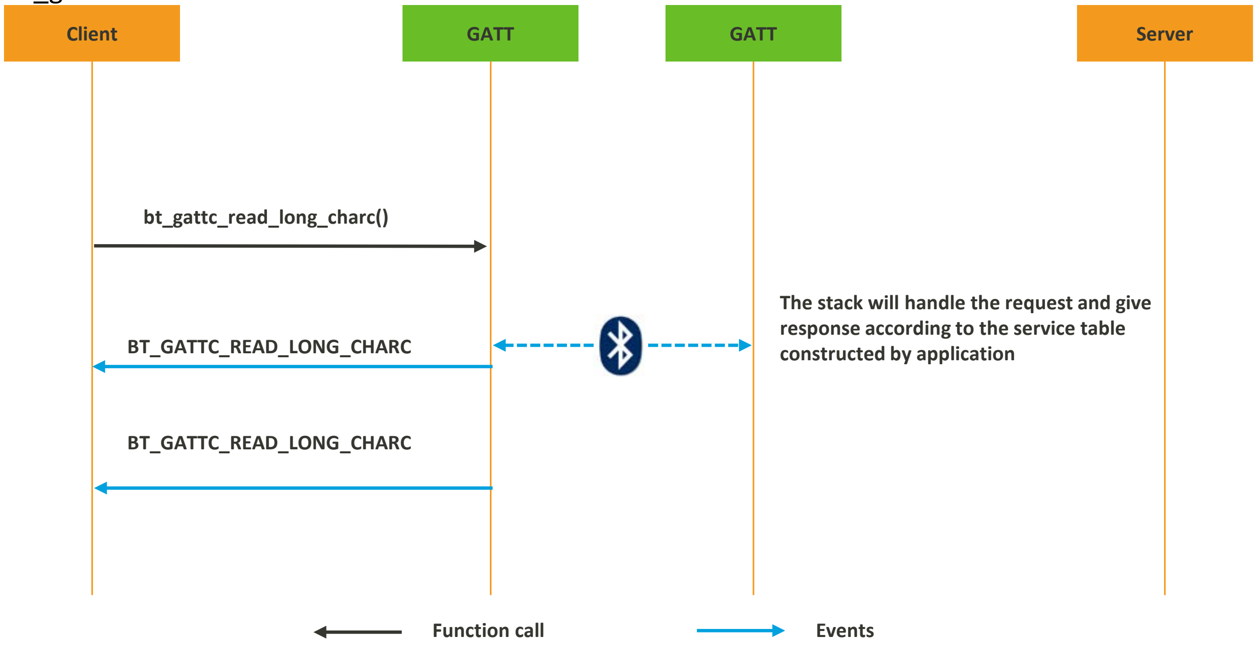
Client
GATT
GATT
Server
bt_gattc_read_long_charc()
BT_GATTC_READ_LONG_CHARC
The stack will handle the request and give response according to the service table constructed by application
BT_GATTC_READ_LONG_CHARC
Function call
Events

Figure 96. Characteristic value read event sequence

An example implementation to apply the function `bt_gattc_read_long_charc()`:

```
/* BLE devices is connected, and conn_handle has been obtained from
BLE_GAP. */
BT_GATTC_NEW_READ_LONG_CHARC_REQ(req, 0x0008, 0);
bt_gattc_read_long_charc(conn_handle, &req);
void bt_app_event_callback(bt_msg_type_t msg, bt_status_t status, void
*buff)
{
  switch (msg) {
  case BT_GATTC_READ_LONG_CHARC: {
    if (status != BT_STATUS_SUCCESS) {
      printf("BT_GATTC_READ_LONG_CHARC FINISHED!!");
      break;
    }
    bt_gattc_read_blob_rsp_t *rsp = (bt_gattc_read_blob_rsp_t*)buff;
    uint8_t length = rsp->length - 1;
    int i;
    for (i = 0; i < length ; i++)
      printf("0x%02x ", rsp->att_rsp->attribute_value[i]);
    break;
  }
}
```

The message flow to read the characteristic value by UUID is shown in Figure 97. More details can be found in the `bt_gattc.h` header file.

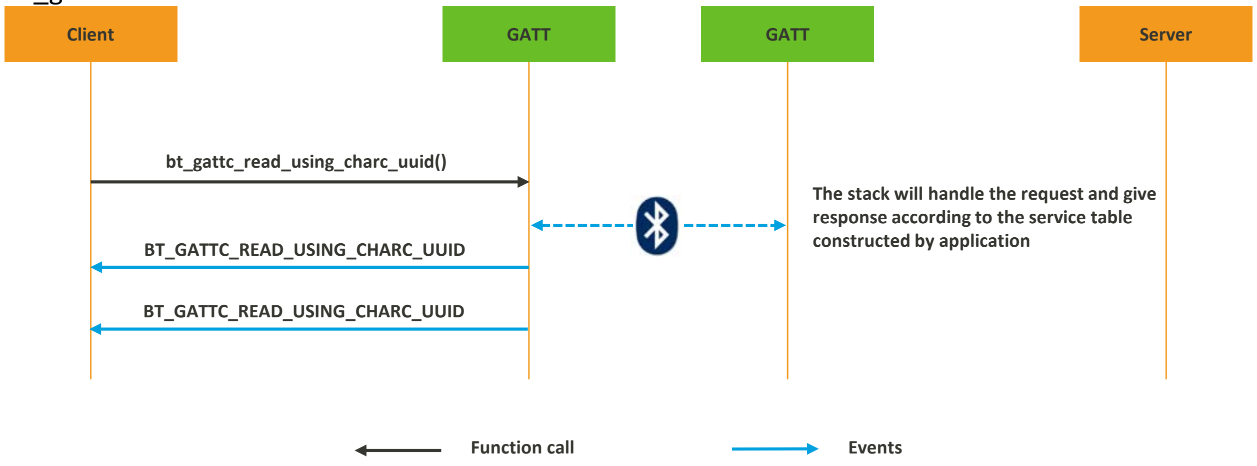
Client
GATT
GATT
Server
bt_gattc_read_using_charc_uuid()
BT_GATTC_READ_USING_CHARC_UUID
BT_GATTC_READ_USING_CHARC_UUID
The stack will handle the request and give response according to the service table constructed by application
Function call
Events

Figure 97. Read characteristic value using UUID event sequence

An example implementation to apply the function `bt_gattc_read_using_charc_uuid()`:

```
/* BLE devices is connected, and conn_handle has been obtained from
BLE_GAP. */
uint8_t uuid16[2] = {0x2A, 0x01};
BT_GATTC_NEW_READ_USING_CHARC_UUID_REQ(req, 0x0001, 0xFFFF, uuid16, 2);
bt_gattc_read_using_charc_uuid(conn_handle, &req);

void bt_app_event_callback(bt_msg_type_t msg, bt_status_t status, void
*buff)
{
  switch (msg) {
  case BT_GATTC_READ_USING_CHARC_UUID: {
    if (status != BT_STATUS_SUCCESS) {
      printf("BT_GATTC_READ_USING_CHARC_UUID FINISHED!!");
      break;
    }
    bt_gattc_read_by_type_rsp_t *rsp = (bt_gattc_read_by_type_rsp_t
*)buff;
    uint8_t *attribute_data_list = rsp->att_rsp->attribute_data_list;
    uint16_t attribute_handle = 0;
    uint8_t attribute_value[30] = {0};
    int i, j;
```

```
      uint8_t num_of_data = (rsp->length - 2) / rsp->att_rsp->length;
      for (i = 0 ; i < num_of_data; i++) {
        memcpy(&attribute_handle,
               attribute_data_list + i * rsp->att_rsp->length, 2);
        memcpy(&attribute_value, attribute_data_list + i*rsp->att_rsp-
  >length + 2,
               rsp->att_rsp->length - 2);
        for (j = 0; j < rsp->att_rsp->length - 2; j++) {
          printf("0x%02x ", attribute_value[j]);
        }
      }
      break;
    }
  }
```

The message sequence to read multiple characteristic values is shown in Figure 98. More details can be found in the `bt_gattc.h` header file.

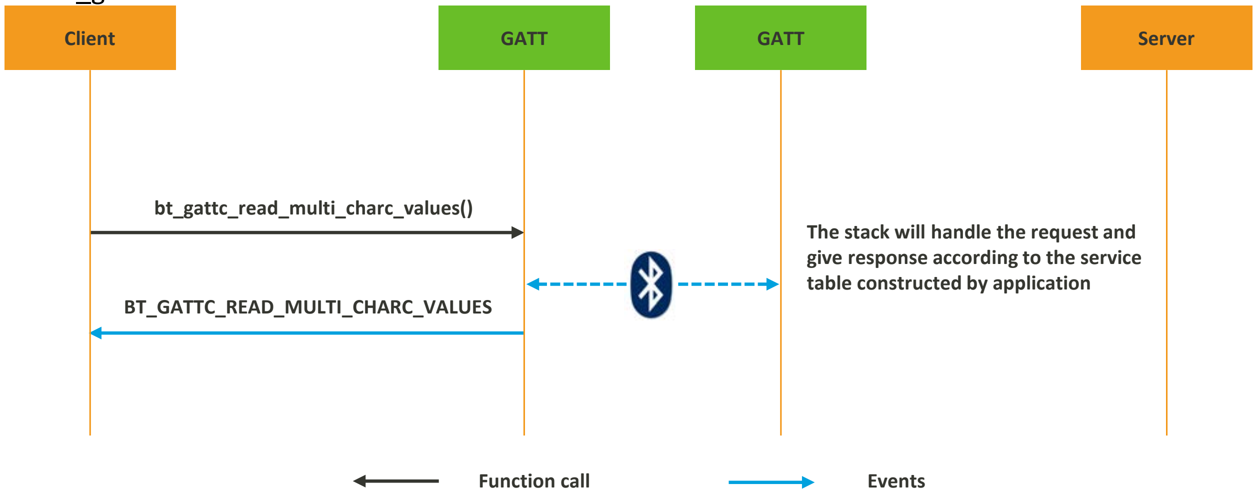
Client
GATT
GATT
Server
bt_gattc_read_multi_charc_values()
BT_GATTC_READ_MULTI_CHARC_VALUES
The stack will handle the request and give response according to the service table constructed by application
Function call
Events

Figure 98. Read multiple characteristic value event sequence

An example implementation to apply the function `bt_gattc_read_multi_charc_values()`:

```
  /* BLE devices is connected, and conn_handle has been obtained from
  BLE_GAP. */
  bt_gattc_read_multi_charc_values_req_t req;
  req.handle_length = 4;     /* The length of value handles. 4 means there
  are two handles. */
  uint8_t buffer[5] = {0};     /* Buffer length = 1 + handle_length. */
  req.att_req = (bt_att_read_multiple_req_t *)buffer;
  req.att_req->opcode = BT_ATT_OPCODE_READ_MULTIPLE_REQUEST;
  req.att_req->set_of_handles[0] = 0x0003;
  req.att_req->set_of_handles[1] = 0x0005;
  bt_gattc_read_multi_charc_values(conn_handle, &req);

  void bt_app_event_callback(bt_msg_type_t msg, bt_status_t status, void
  *buff)
  {
    switch (msg) {
    case BT_GATTC_READ_MULTI_CHARC_VALUES: {
      if (status != BT_STATUS_SUCCESS) {
        printf("BT_GATTC_READ_MULTI_CHARC_VALUES FINISHED!!");
        break;
      }
      bt_gattc_read_multiple_rsp_t *rsp = (bt_gattc_read_multiple_rsp_t
  *)buff;
      uint8_t op_code = rsp->att_rsp->opcode;
      uint8_t length = rsp->length - 1;
      int i;
      for (i = 0; i < length; i++) {
        printf("0x%02x ", rsp->att_rsp->set_of_values[i]);
      }
      break;
    }
  }
```

4.3.2.7. Characteristic value write

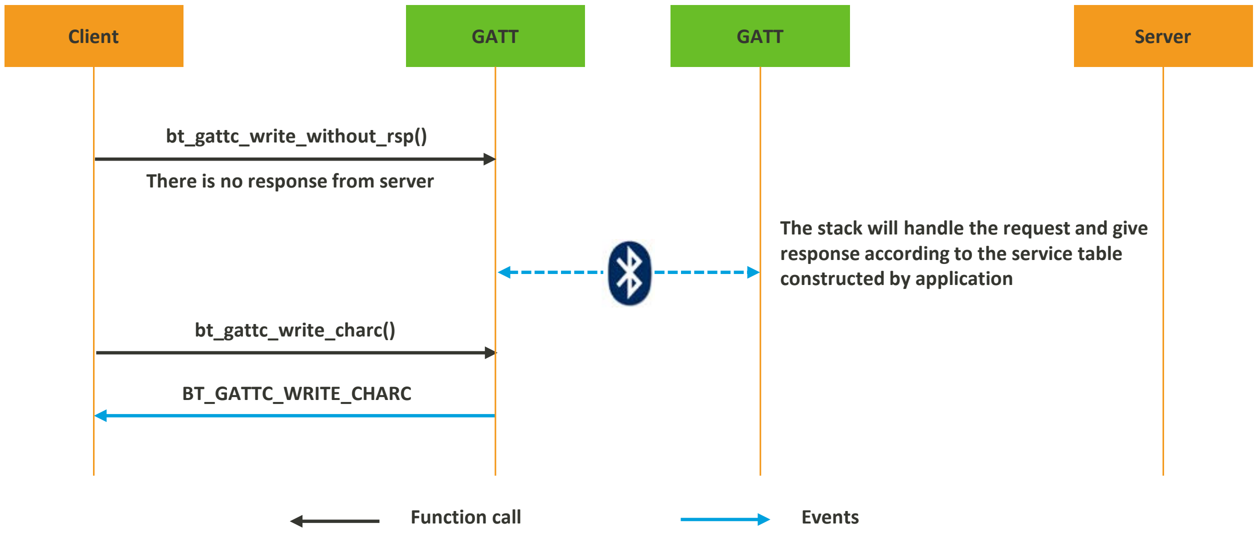
Client
GATT
GATT
Server
bt_gattc_write_without_rsp()
There is no response from server
The stack will handle the request and give response according to the service table constructed by application
bt_gattc_write_charc()
BT_GATTC_WRITE_CHARC
Function call
Events

Figure 99. Characteristic value write event sequence

There are two types for write characteristic value requests to write a characteristic value no longer than the MTU, as shown in Figure 99. More details can be found in the `bt_gattc.h` header file.

1) Call the function `bt_gattc_wrote_without_rsp()`, if the client knows the characteristic value handle and does not need an acknowledgement that the write was successfully performed.
2) Call the function `bt_gattc_write_charc()`, if the client requires the status of a write request.

An example implementation of these two APIs:

```
  /* BLE devices is connected, and conn_handle has been obtained from
  BLE_GAP. */
  static uint8_t buff[40] = {0};
  static uint8_t *value = "TestWriteCmd";
  BT_GATTC_NEW_WRITE_WITHOUT_RSP_REQ(req, buff, 0, 0x0008, value,
  strlen(value));
  /* 0 means it is WRITE_COMMAND, 1 means SIGNED_WRITE_COMMAND. */
  bt_gattc_write_without_rsp(conn_handle, 0, &req);
  /* There is no response to handle for this request. */

  static uint8_t buff2[40] = {0};
  BT_GATTC_NEW_WRITE_CHARC_REQ(wreq, buff2, 0x0011, value, strlen(value));
```

```
bt_gattc_write_charc(conn_handle, &wreq);

void bt_app_event_callback(bt_msg_type_t msg, bt_status_t status, void
*buff)
{
  switch (msg) {
  case BT_GATTC_WRITE_CHARC: {
    printf("bt_gattc_write_charc, status = %d!!", status);
    break;
  }
}
```

4.3.2.8. Long characteristic value or descriptor write

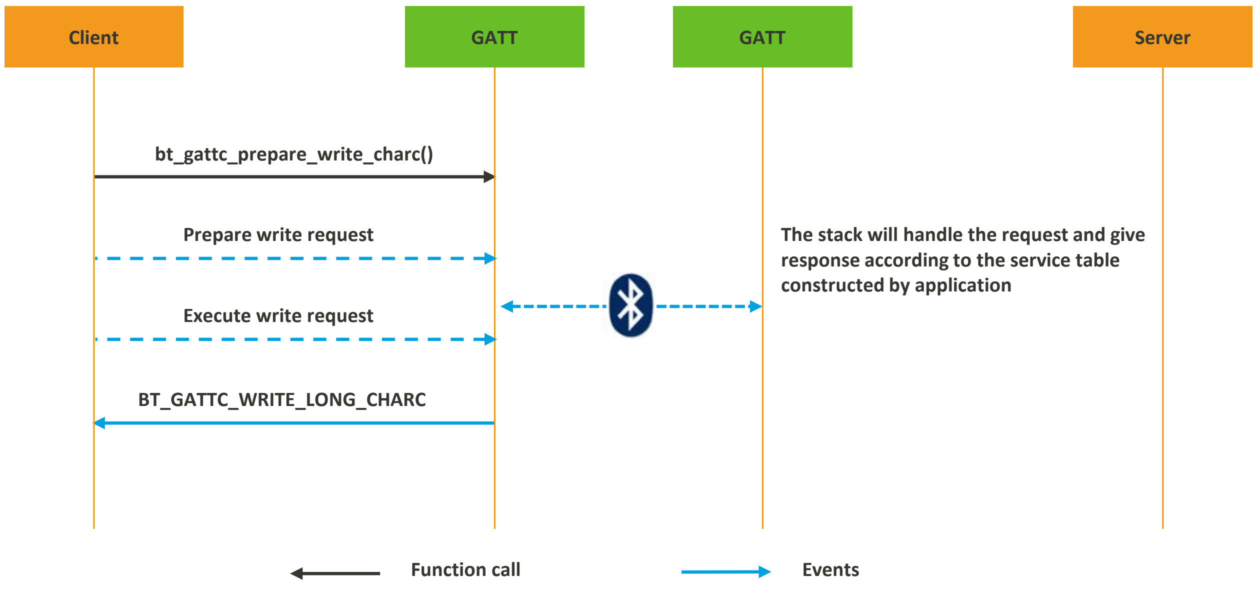
Client
GATT
GATT
Server
bt_gattc_prepare_write_charc()
Prepare write request
Execute write request
BT_GATTC_WRITE_LONG_CHARC
The stack will handle the request and give response according to the service table constructed by application
Function call
Events

Figure 100. Long value write event sequence

Apply the function `bt_gattc_prepare_write_charc()`, if the length of a characteristic value or descriptor is greater than the MTU size, as shown in Figure 100. More details can be found in the `bt_gattc.h` header file.

To write the complete characteristic value or descriptor, the offset should be set to 0. The Bluetooth stack will update the offset and send a subsequent **Prepare write** request. The **Prepare write** request is repeated until the characteristic value or descriptor is completely transferred, after which the **Execute write** request is used to write the complete value.

The second parameter of `bt_gattc_prepare_write_charc()` is to decide whether it is a reliable write or not. If set to 1, the Bluetooth stack will check if the value in the **Prepare write** response is the same as **Prepare write** request, if so, it will send a subsequent **Prepare write** request, else, it will send **Cancel the execute write request** to abort the write long procedure.

An example implementation is shown below:

```
/* BLE devices is connected, and conn_handle has been obtained from
BLE_GAP. */
static uint8_t *gatt_long_charc =
"ABCDEF1234567890FEDCBA09876543210413454556353 5";
BT_GATTC_NEW_PREPARE_WRITE_REQ(req, 0x0022, 0, gatt_long_charc,
strlen(gatt_long_charc));
```

```
bt_gattc_prepare_write_charc(conn_handle, 0, 0, &req);

void bt_app_event_callback(bt_msg_type_t msg, bt_status_t status, void
*buff)
{
  switch (msg) {
  case BT_GATTC_WRITE_LONG_CHARC:
  case BT_GATTC_RELIABLE_WRITE_CHARC: {
    printf("bt_gattc_prepare_write_charc, status = %d!!", status);
    break;
  }
}
```

4.3.2.9. Characteristic value indication

Call the function `bt_gatts_send_charc_value_notification_indication()`, if the server requires to indicate a characteristic value to a client and expects an acknowledgement. The message flow is shown in Figure 101. More details can be found in the `bt_gatts.h` header file.

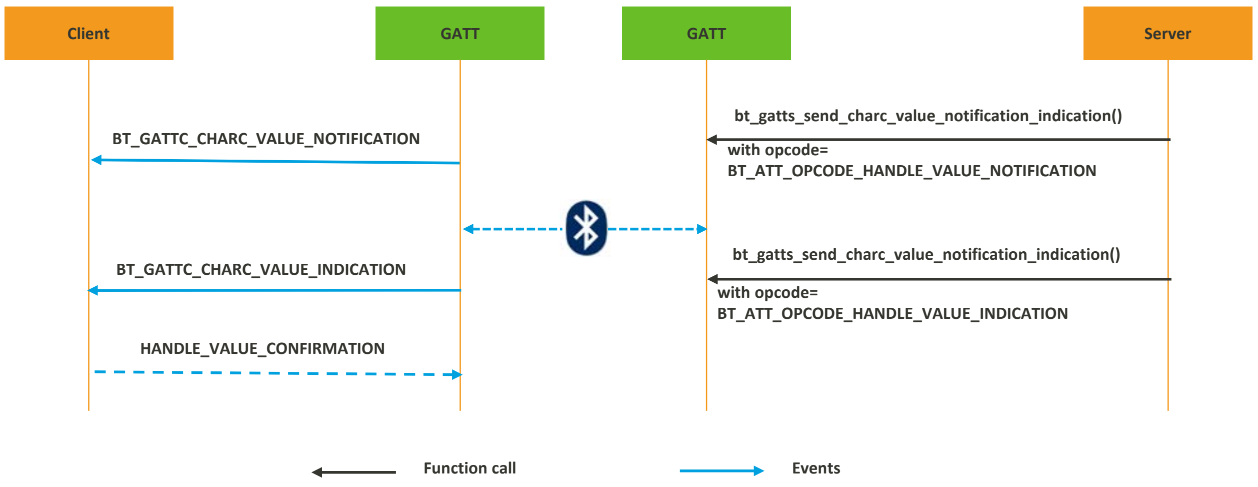
Client
GATT
GATT
Server
bt_gatts_send_charc_value_notification_indication()
BT_GATTC_CHARC_VALUE_NOTIFICATION
with opcode=
BT_ATT_OPCODE_HANDLE_VALUE_NOTIFICATION
bt_gatts_send_charc_value_notification_indication()
BT_GATTC_CHARC_VALUE_INDICATION
with opcode=
BT_ATT_OPCODE_HANDLE_VALUE_INDICATION
HANDLE_VALUE_CONFIRMATION
Function call
Events

Figure 101. Characteristic value indication event sequence

An example implementation for the server to notify and indicate a characteristic value:

```
/* BLE devices is connected, and conn_handle has been obtained from
BLE_GAP. */
static uint8_t buf[64] = {0};
static uint8_t *noti_value = “notification”;  /* The value to notify. */
bt_gattc_charc_value_notification_indication_t *bas_noti =
(bt_gattc_charc_value_notification_indication_t *)buf;
bas_noti->att_req.opcode = BT_ATT_OPCODE_HANDLE_VALUNE_NOTIFICATION;
bas_noti->att_req.handle = 0x0013; /* The value handle to notify. */
bas_noti->attribute_value_length = strlen(noti_value);
memcpy((void *)(bas_noti->att_req.attribute_value), noti_value,
strlen(noti_value));
bt_gatts_send_charc_value_notification_indication(conn_handle, bas_noti);
static uint8_t *ind_value = “indication”; /* The value to indicate. */
bt_gattc_charc_value_notification_indication_t *bas_ind =
(bt_gattc_charc_value_notification_indication_t *)buf;
bas_ind->att_req.opcode = BT_ATT_OPCODE_HANDLE_VALUNE_INDICATION;
bas_ind ->att_req.handle = 0x0015; /* The value handle to indicate. */
bas_ind ->attribute_value_length = strlen(ind_value);
```

```
memcpy((void *)( bas_ind ->att_req.attribute_value), ind_value,
strlen(ind_value));
bt_gatts_send_charc_value_notification_indication(conn_handle, bas_ind);
```

An example implementation for the client to handle a characteristic value notification and indication:

```
void bt_app_event_callback(bt_msg_type_t msg, bt_status_t status, void
*buff)
{
  switch (msg) {
  case BT_GATTC_CHARC_VALUE_NOTIFICATION:
  case BT_GATTC_CHARC_VALUE_INDICATION: {
    bt_gatt_handle_value_notification_t *rsp =
                               (bt_gatt_handle_value_notification_t
*)buff;
    uint8_t op_code = rsp->att_rsp->opcode;
    uint16_t attribute_handle = rsp->att_rsp->handle;
    uint8_t length = rsp->length - 3;
    int i;
    printf("op_code = 0x%08x", op_code);
    printf("attribute_handle = 0x%08x", attribute_handle);
    for (i = 0; i < length ; i++)
      printf("0x%02x ", rsp->att_rsp->attribute_value[i]);
    printf("\n");
    break;
  }
}
```

5. Creating a Custom Bluetooth Application

This section guides you through creating your own Bluetooth project including:

- Memory management for the SDK according to the application requirements.
- Creating a task. The task provides access to the system resources such as CPU and queues. The application can adjust task priorities to optimize the CPU usage.
- How to interact with the SDK stack task — a special, high priority task.
- Bluetooth panic mechanism for restoring Bluetooth function when Bluetooth works no normal.

5.1. Memory management

5.1.1. Memory configuration

Memory configuration of the application should be applied before the Bluetooth powers on. There are two types of memory resources:

1) TX/RX buffer.
   - `BT_MEMORY_TX_BUFFER` — Bluetooth transmit buffer. The buffer size determines the transmit speed and probability of out of memory (OOM) occurrence.
   - `BT_MEMORY_RX_BUFFER` — Bluetooth receive buffer. The buffer size determines the speed and probability of the OOM occurrence.
2) Fixed size control blocks are shown in Table 5.

Table 5. Fixed size control blocks

Control block	Description
`BT_MEMORY_CONTROL_BLOCK_TIMER`	Control block for the timer. The maximum allocated timer value is ACL Link number added with 3 and normally it's the same as the value for `BT_MEMORY_CONTROL_BLOCK_LE_CONNECTION` and `BT_MEMORY_CONTROL_BLOCK_EDR_CONNECTION`
`BT_MEMORY_CONTROL_BLOCK_LE_CONNECTION`	Control block for LE connection. Each ACL link allocates a portion of this memory.
`BT_MEMORY_CONTROL_BLOCK_EDR_CONNECTION`	Control block for EDR connection. Each ACL link allocates a portion of this memory.
`BT_MEMORY_CONTROL_BLOCK_RFCOMM`	Control block for RFCOMM
`BT_MEMORY_CONTROL_BLOCK_AVRCP`	Control block for AVRCP
`BT_MEMORY_CONTROL_BLOCK_PBAPC`	Control block for PBAPC
`BT_MEMORY_CONTROL_BLOCK_A2DP_SEP`	Control block for A2DP SEP
`BT_MEMORY_CONTROL_BLOCK_A2DP`	Control block for A2DP
`BT_MEMORY_CONTROL_BLOCK_HFP`	Control block for HFP
`BT_MEMORY_CONTROL_BLOCK_SPP`	Control block for SPP

The application should configure the memory for the control block, if the module is enabled. To improve memory utilization, the SDK enables an OOM mechanism. If OOM occurs, the device operation may fail or the throughput may slow down. To increase the performance of Bluetooth stack, application should ensure there is enough memory allocated in TX/RX buffers.

An example implementation to configure the memory size for each memory block is shown below.

```
/*To configure the max EDR ACL link number, please refer to the datasheet
chapter 1.3 for the maximum link number.*/
#define BT_CONNECTION_MAX 2
#define BT_CONNECTION_BUF_SIZE \
                   (BT_CONNECTION_MAX*
BT_CONTROL_BLOCK_SIZE_OF_EDR_CONNECTION)
static char bt_connection_cb_buf[BT_CONNECTION_BUF_SIZE];

/*To configure the timer, the suggested value is between BT_MAX_ACL_LINK
to
    BT_MAX_ACL_LINK+3 */
#define BT_TIMER_NUM 5
#define BT_TIMER_BUF_SIZE (BT_TIMER_NUM * BT_CONTROL_BLOCK_SIZE_OF_TIMER)
static char timer_cb_buf[BT_TIMER_BUF_SIZE];

/*To configure the transmit packet buffer,  the vaule depends on the ACL
link and the GATT MTU size. Usually it’s difficult to send data in All ACL
links. The suggest TX buffer size is about 2~3 times the  MTU size.*/
#define BT_TX_BUFFER_SIZE 256 * BT_MAX_ACL_LINK
static tx_buf[BT_TX_BUFFER_SIZE];

/*To configure the receive packet buffer,  the value depends on the ACL
link and the GATT MTU size. Receive buffer may recive data from all ACL
links and advertising data. The suggested RX buffer size is more than
1Kbytes  or ACL link num * MTU size) */
#define BT_RX_BUFFER_SIZE 1024 * BT_MAX_ACL_LINK
static rx_buf[BT_RX_BUFFER_SIZE];

/*To configure the max LE link number, please refer to the datasheet
chapter 1.3 for the maximum link number. */
  #define BT_LE_CONNECTION_MAX 10
#define BT_LE_CONNECTION_BUF_SIZE \
            (_LE_CONNECTION_MAX * BT_CONTROL_BLOCK_SIZE_OF_LE_CONNECTION)
static char le_conn_cb_buf[BT_LE_CONNECTION_BUF_SIZE];

/*To configure the rfcomm buffer size, the value of
BT_RFCOMM_TOTAL_LINK_NUM is the supported rfcomm link number*/
#define BT_RFCOMM_TOTAL_LINK_NUM 2
#define BT_RFCOMM_BUF_SIZE \
                 (BT_RFCOMM_TOTAL_LINK_NUM *
BT_CONTROL_BLOCK_SIZE_OF_RFCOMM)
static char bt_rfcomm_cb_buf[BT_RFCOMM_BUF_SIZE];

/*To configure the hfp buffer size, the value of BT_HFP_TOTAL_LINK_NUM is
the supported hfp link number*/
#define BT_HFP_TOTAL_LINK_NUM 2
#define BT_HFP_BUF_SIZE \
                  (BT_HFP_TOTAL_LINK_NUM * BT_CONTROL_BLOCK_SIZE_OF_HFP)
static char bt_hfp_cb_buf[BT_HFP_BUF_SIZE];

/*To configure the avrcp buffer size, the value of BT_AVRCP_TOTAL_LINK_NUM
```

```
is the supported avrcp link number*/
#define BT_AVRCP_TOTAL_LINK_NUM 2
#define BT_AVRCP_BUF_SIZE \
                  (BT_AVRCP_TOTAL_LINK_NUM *
BT_CONTROL_BLOCK_SIZE_OF_AVRCP)
static char bt_avrcp_cb_buf[BT_AVRCP_BUF_SIZE];

/* To configure the a2dp streaming end point(sep) buffer size, the value
of BT_A2DP_SINK_CODEC_NUM is the supported sink codec number. The value of
BT_A2DP_SEP_TOTAL_NUM is the supoported a2dp link number *
BT_A2DP_SINK_CODEC_NUM. */
#define BT_A2DP_SINK_CODEC_NUM (2)
#define BT_A2DP_SEP_TOTAL_NUM (2 * BT_A2DP_SINK_CODEC_NUM)
#define BT_A2DP_SEP_BUF_SIZE (BT_A2DP_SEP_TOTAL_NUM *
BT_CONTROL_BLOCK_SIZE_OF_A2DP_SEP) static char
bt_a2dp_sep_cb_buf[BT_A2DP_SEP_BUF_SIZE];

/*To configure the a2dp buffer size, the value of BT_A2DP_TOTAL_LINK_NUM
is the supported a2dp link number*/
#define BT_A2DP_TOTAL_LINK_NUM 2
#define BT_A2DP_BUF_SIZE \
                   (BT_A2DP_TOTAL_LINK_NUM *
BT_CONTROL_BLOCK_SIZE_OF_A2DP)
static char bt_a2dp_cb_buf[BT_A2DP_BUF_SIZE];

/*To configure the spp buffer size, the value of
BT_SPP_TOTAL_CONNECTION_NUM is the supported spp total connection number*/
#define BT_SPP_TOTAL_CONNECTION_NUM  5
#define BT_SPP_BUF_SIZE \
                   (BT_SPP_TOTAL_CONNECTION_NUM *
BT_CONTROL_BLOCK_SIZE_OF_SPP)
static char bt_spp_cb_buf[BT_SPP_BUF_SIZE];

/*To configure the pbapc buffer size, the value of
BT_PBAPC_TOTAL_CONNECTION_NUM  is the supported pbapc link number*/
#define BT_PBAPC_TOTAL_CONNECTION_NUM  2
#define BT_PBAPC_BUF_SIZE \
               (BT_PBAPC_TOTAL_CONNECTION_NUM *
BT_CONTROL_BLOCK_SIZE_OF_PBAPC)
static char bt_pbapc_cb_buf[BT_PBAPC_BUF_SIZE];

void bt_mm_init(void)
{
    bt_memory_init_packet(BT_MEMORY_TX_BUFFER, tx_buf, BT_TX_BUFFER_SIZE);
    bt_memory_init_packet(BT_MEMORY_TX_BUFFER, tx_buf, BT_TX_BUFFER_SIZE);
    bt_memory_init_control_block(BT_MEMORY_CONTROL_BLOCK_TIMER,
timer_cb_buf,
                                 BT_TIMER_BUF_SIZE);
    bt_memory_init_control_block(BT_MEMORY_CONTROL_BLOCK_LE_CONNECTION,
                                 le_conn_cb_buf,
BT_LE_CONNECTION_BUF_SIZE);
    bt_memory_init_control_block(BT_MEMORY_CONTROL_BLOCK_EDR_CONNECTION,
                                 bt_connection_cb_buf,
BT_CONNECTION_BUF_SIZE);
    bt_memory_init_control_block(BT_MEMORY_CONTROL_BLOCK_RFCOMM,
                                 bt_rfcomm_cb_buf, BT_RFCOMM_BUF_SIZE);
    bt_memory_init_control_block(BT_MEMORY_CONTROL_BLOCK_HFP,
                                 bt_hfp_cb_buf, BT_HFP_BUF_SIZE);
    bt_memory_init_control_block(BT_MEMORY_CONTROL_BLOCK_AVRCP,
                                 bt_avrcp_cb_buf, BT_AVRCP_BUF_SIZE);
    bt_memory_init_control_block(BT_MEMORY_CONTROL_BLOCK_A2DP_SEP,
                                 bt_a2dp_sep_cb_buf,
BT_A2DP_SEP_BUF_SIZE);
    bt_memory_init_control_block(BT_MEMORY_CONTROL_BLOCK_A2DP,
                                 bt_a2dp_cb_buf, BT_A2DP_BUF_SIZE);
    bt_memory_init_control_block(BT_MEMORY_CONTROL_BLOCK_SPP,
                                 bt_spp_cb_buf, BT_SPP_BUF_SIZE);
    bt_memory_init_control_block(BT_MEMORY_CONTROL_BLOCK_PBAPC,
                                 bt_pbapc_cb_buf, BT_PBAPC_BUF_SIZE);
}

int main(void)
{
    /*Configure the system */

    /*Configure memory of the SDK.*/
    bt_mm_init();

}
```

5.1.2. Out of Memory (OOM) handle

Reasons OOM occurs:

- The control block memory is not enough. If the connect request API returns OOM, there may be not enough memory allocated for the control block memory. The application should enlarge the control block memory.
- The TX memory is not enough. User can check the system log for a message, such as <`TX OOM: "[MM]should not come here, please enlarge TX memory. "`>. If TX OOM occurs rarely, the application can delay for a while and retry the TX request. If the error message persists, the application should enlarge the TX buffer.
- The RX memory is not enough. User can check the system log for a message, such as <`RX OOM: "[MM]should not come here, please enlarge Rx memory."`>. If the RX performance is poor due to RX OOM, the application should enlarge the RX buffer.

5.2. Create an application task

Airoha IoT SDK enables to create a task at runtime. The application can create a queue or semaphore to communicate with other tasks, as shown below.

```
void app_task(void *arg)
{
    /*Application can implment its code here */
}

void bt_task(void * arg)
{
    uint32_t i;
    bt_bd_addr_t random_addr;
    for(i = 0; i < 6; i++) {
        * (((uint8_t*)random_addr) + i) =
(uint8_t)(bt_os_layer_generate_random() & 0xFF);
    }
    * (((uint8_t*)random_addr) + 5) |= 0xC0;
    bt_task_semaphore = bt_os_layer_create_semaphore();
    //bt_os_layer_sleep_task(500);
    if (arg != NULL) {
        // If arg is NULL means user does not want to power on BT
        bt_power_on((bt_bd_addr_ptr_t)arg,
(bt_bd_addr_ptr_t)&random_addr);
    }

    /* main loop */
    do {
        bt_os_layer_take_semaphore(bt_task_semaphore);

        /* handle events */
        if (BT_STATUS_SUCCESS != bt_handle_interrupt()) {
            break;
        }
    } while (1);
}

int main(void)
{
/*first need configure hardware and system*/

    /*Task API usage please refer RTOS dev guider*/
    xTaskCreate(app_task, "app_t", 512, NULL, 1, NULL);

    /*bt_task is the main function implmented in SDK that application need
use it
      as entry function. After bt task running, bt will power on
automaticlly.  */
    xTaskCreate(bt_task, "bt_task", 1024, local_pulblic_addr, 5, NULL);

    /*All task will truely construtor after this function executed*/
    vTaskStartScheduler();

}
```

The default implementation of Bluetooth task entry function `bt_task()` is provided in `bt_platform.c`. A user-defined entry function implementation can also be provided in that source file. Call the function `bt_task()` in the `bt_platform.h` as a task entry function to create a stack task. The stack size of `bt_task()` should be no less than 1024 words. Call the API `bt_handle_interrupt()` to handle the Bluetooth stack interrupt, such as timer and events in stack task.

5.3. Interaction with the Bluetooth host stack

The application task can call Bluetooth APIs directly. Callback functions are invoked in the Bluetooth host stack to process the events that are input as parameter to the callback (see Figure 102). Note, that the stack task needs to be created by the application and the entry function should handle the Bluetooth stack interrupt.

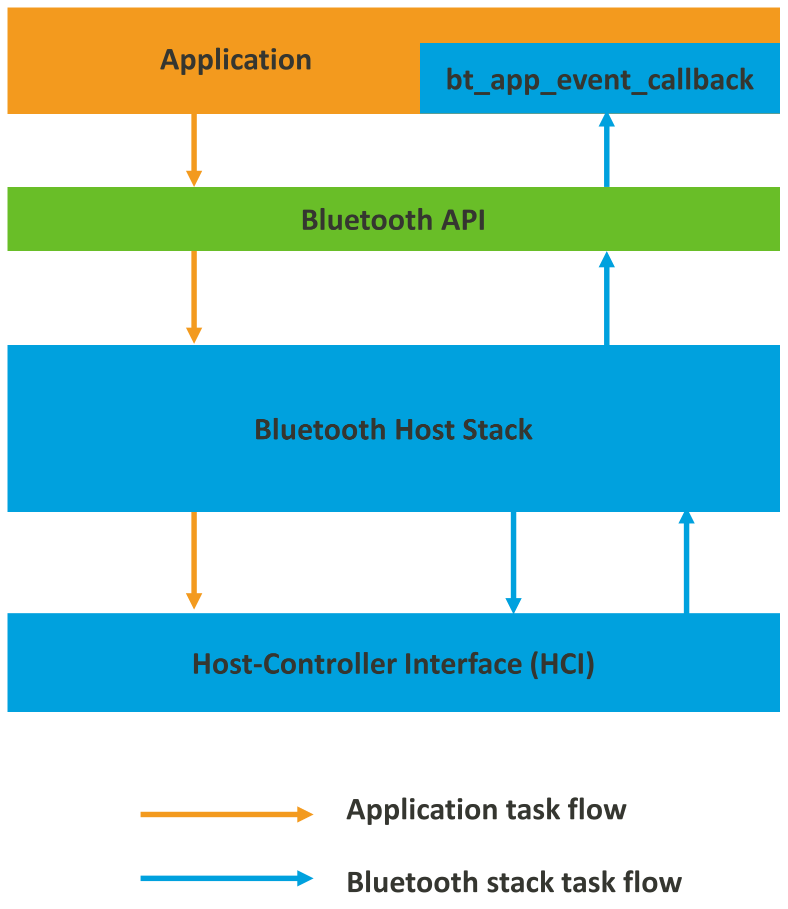

Figure 102. Application interaction with the Bluetooth API

5.3.1. Calling the APIs

The Airoha IoT SDK provides a set of thread-safe APIs for the application task. The application can call the SDK APIs in any task even including the stack task self. Note, that the SDK APIs run in the caller task.

```
void app_task(void *arg)
{
    /*Application can implement the code here */
    while(1) {
    /*Application should make task non-block, like receive queue here*/
        {
            bt_bd_addr_t addr[] = {0xC3,0x01,0x02,0x03,0x05,0x11};
            /*set random address API run in app task*/
            bt_gap_le_set_random_address(addr);
        }
    }
}
```

5.3.2. Processing the events

The SDK will send an event to `bt_app_event_callback()` which needs to be implemented in the application. The callback will be called in the SDK task, so the application has to consider which task will process the event. It's recommended to send the event to its own task. Note, that if processing an event in the stack task takes too long, it may cause Bluetooth connection failure or reduce the data transmission rate. The application can use queue or semaphore in RTOS to switch task from stack to its own task.

An example implementation on getting the event from Bluetooth stack task is shown below.

```
typedef struct {
    uint16_t event_id
    void *parameter;
} bt_app_event_t;

PRIVILEGED_DATA static QueueHandle_t bt_app_queue_p;

bt_status_t bt_app_event_callback(bt_msg_type_t msg, bt_status_t status,
void *buff){
    bt_app_event_t event;
    switch(msg) {
        /*Handle event and convert to message sending to application
task*/
        xQueueSend(bt_app_queue_p, &event, 0);
    }
}

void app_task(void *arg)
{
    /*Application can implment its code here */
    bt_app_queue_p = xQueueCreate(20,  sizeof(bt_app_event_t));

    while(1) {
    /*Application should make task non-block, like receive queue*/

      if (xQueueReceive(bt_app_queue_p, &queue_event, portMAX_DELAY) ==
pdPASS){
             /*Handle message here*/
        }
    }
}
```

5.4. Bluetooth panic mechanism

The application will receive Bluetooth panic event from `bt_app_event_callback()`, in case Bluetooth host or controller malfunction. The application should reboot the Bluetooth with the `bt_power_off()` and `bt_power_on()` APIs to run the module successfully, as shown in the code below. It’s also required to clear unused data or reset the state machine of the application to release all resources and reset the system to the default state.

An example implementation to restore Bluetooth when Bluetooth panic event is received is shown below.

```
bool is_panic = false;
bt_status_t bt_app_event_callback(bt_msg_type_t msg, bt_status_t status,
void *buff){
    switch(msg) {
        case BT_PANIC: {
            is_panic = true;
            if (((bt_panic_t *)buff)->source == BT_PANIC_SOURCE_HOST) {
                /* The panic event source is Bluetooth host */
                /* The application should */
                /* clear unused data or reset state machine of the
application, and reboot Bluetooth (or reboot device). */
                bt_power_off();
            } else {
                /* The panic event source is Bluetooth controller */
                /* The application should */
                /* clear unused data or reset state machine of the
application, and reboot Bluetooth (or reboot device). */
                bt_power_off();
            }
        }
        case BT_POWER_OFF_CNF: {
            if (is_panic == true) {
                bt_power_on(NULL,NULL);
            }
        }
        case BT_POWER_ON_CNF: {
            if (is_panic == true) {
                is_panic = false;
             /*reboot Bluetooth success, start to run application again.*/
            }
        }
    }
}
```

6. Debugging and Porting Layer

6.1. Debugging

The Airoha IoT SDK provides two ways to debug the application:

- System log — provides the host information.
- HCI log — provides the HCI communication log between host and controller.

6.1.1. System log

The Bluetooth stack calls `bt_debug_log(), bt_debug_log_msgid_d(), bt_debug_log_msgid_i(), bt_debug_log_msgid_w() or bt_debug_log_msgid_e()` in the `bt_debug.c` source file to provide the host information. Application can set a filter or disable logging in this function. The application can also output these logs to any UART or USB port.

6.1.2. HCI log

The Airoha IoT SDK provides HCI information by calling the `bt_hci_log()` function in the `bt_hci_log.c` source file. The HCI information follows the standard HCI data format in Bluetooth core specification. The application can receive this information in the `bt_hci_log()` function and then send the output to UART. Users can analyze the data using standard HCI data or parse these data to a file that commercial tools like Frontline can recognize.

6.2. Porting layer

The Bluetooth stack has flexibility to integrate with third-party RTOS APIs. These APIs can be implemented in the source file `bt_os_layer_api.c`.

7. Appendix A: Acronyms and Abbreviations

The acronyms and abbreviations used in this developer's guide are listed in Table 6.

Table 6. Acronyms and abbreviations

Abbreviation/Term	Expansion/Definition
GAP	Generic Access Profile defines the generic procedures related to the discovery of Bluetooth devices and link management aspects of connecting to Bluetooth devices. It also defines procedures related to use of different security levels.
SM	Security Manager defines the protocol and behavior to manage pairing, authentication, and encryption between LE devices.
GATT	Generic Attribute Profile defines a service framework using the Attribute Protocol for discovering services and for reading and writing characteristic value on a remote device.
MTU	The Maximum Transmission Unit (MTU) is the size of the largest protocol data unit that the communications layer can pass onwards.
LTK	Long Term Key, is a 128-bit key used to generate the contributory session key for an encrypted connection.
MITM	Man-in-the-middle attack is an attack where the attacker secretly relays and possibly alters the communication between two parties who believe they are directly communicating with each other.
STK	Short Term Key, is a 128-bit key used to encrypt a connection following pairing.
EDIV	Encrypted Diversifier is a 16-bit stored value used to identify the LTK distributed during the low energy legacy pairing.
IRK	Identity Resolving Key is a 128-bit key used to generate and resolve random addresses.
CSRK	Connection Signature Resolving Key is a 128-bit key that signs data and verifies signatures on the receiving device.
Out-Of-Band	The model is primarily designed for scenarios where an Out of Band mechanism is used to discover the devices and to exchange or transfer cryptographic numbers used in the pairing process.
AES	Advanced Encryption Standard