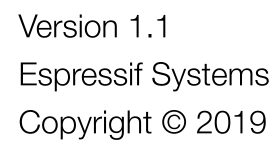
ESP32

Bluetooth Architecture



Version 1.1
Espressif Systems
Copyright © 2019

About This Guide

This document introduces the ESP32 Bluetooth® architecture, namely Bluetooth, Classic Bluetooth and Bluetooth Low Energy. Note that this user guide is only applicable to ESP-IDF V2.1 and earlier.

Release Notes

Date	Version	Release notes
2018.01	V1.0	First release.
2019.11	V1.1	• Updated document cover; • Updated an example in Section 3.24.

Documentation Change Notification

Espressif provides email notifications to keep customers updated on changes to technical documentation. Please subscribe at *https://www.espressif.com/en/subscribe*.

Certification

Download certificates for Espressif products from *https://www.espressif.com/en/certificates*.

Table of Contents

1. Bluetooth 1

1.1. Overview 1

1.1.1. Bluetooth Application Structure 1

1.1.2. Selection of the HCI Interfaces 2

1.1.3. Bluetooth Operating Environment 4

1.2. Architecture 4

1.2.1. Controller 4

1.2.2. BLUEDROID 4

1.2.2.1. Overall Architecture 4

1.2.2.2. OS-related Adaptation 6

1.2.3. Bluetooth Directory Introduction 6

2. Classic Bluetooth 9

2.1. Overview 9

2.1.1. L2CAP 10

2.1.2. SDP 10

2.1.3. GAP 10

2.1.4. A2DP and AVRCP 11

3. Bluetooth Low Energy 14

3.1. GAP 14

3.1.1. Overview 14

3.1.2. Status Transitions among GAP Roles 15

3.1.3. BLE Broadcast Procedure 16

3.1.3.1. Broadcast using a public address 16

3.1.3.2. Broadcast using a resolvable address 17

3.1.3.3. Broadcast using a static random address 18

3.1.4. BLE Modes 19

3.1.4.1. Connectable Scannable Undirected Mode 19

3.1.4.2. High Duty Cycle Directed Mode and Connectable Low Duty Cycle Directed Mode 19

3.1.4.3. Scannable Undirected Mode 19

3.1.4.4. Non-connectable Undirected Mode 20

3.1.5. BLE Broadcast Filtering Policy 20

3.1.6. BLE Scanning Procedure 20

3.1.7. BLE GAP Implementation Mechanism 21
3.2. GATT 21
3.2.1. ATT 21
3.2.2. GATT Profile...23
3.2.3. Add Gatt Services in ESP32 IDF Environment 25
3.2.4. Discover a Peer Device's Services in ESP32 IDF (GATT Client) 26
3.3. SMP 27
3.3.1. Overview...27
3.3.2. Safety Management Controller..28
3.3.2.1.BLE Encryption 28
3.3.2.2.BLE Bonding 30
3.3.3. The Implementation of SMP 30

1. Bluetooth

This chapter describes the basic Bluetooth architecture of ESP32.

1.1. Overview

1.1.1. Bluetooth Application Structure

Bluetooth is a wireless technology standard for exchanging data over short distances, with advantages including robustness, low power consumption and low cost. The Bluetooth system can be divided into two different categories: Classic Bluetooth and Bluetooth Low Energy (BLE). ESP32 supports dual-mode Bluetooth, meaning that both Classic Bluetooth and BLE are supported by ESP32.

Basically, the Bluetooth protocol stack is split into two parts: a "controller stack" and a "host stack". The controller stack contains the PHY, Baseband, Link Controller, Link Manager, Device Manager, HCI and other modules, and is used for the hardware interface management and link management. The host stack contains L2CAP, SMP, SDP, ATT, GATT, GAP and various profiles, and functions as an interface to the application layer, thus facilitating the application layer to access the Bluetooth system. The Bluetooth Host can be implemented on the same device as the Controller, or on different devices. Both approaches are supported by ESP32. Figure 1-1 describes some typical application structures:

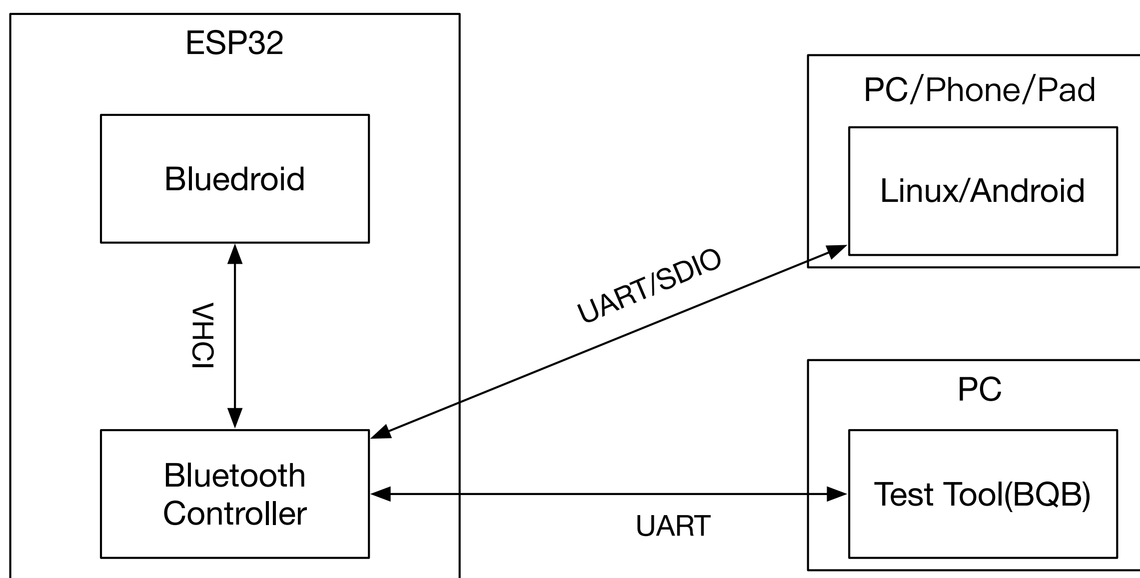


Figure 1-1.The architecture of Bluetooth host and controller in ESP-IDP

- Scenario 1 (Default ESP-IDF setting): BLUEDROID is selected as the Bluetooth Host, and VHCI (software-implemented virtual HCI interface) is used for the communication between Bluetooth Host and Controller. In this scenario, both the BLUEDROID and

the Controller are implemented on the same device (i.e. the ESP32 chip), eliminating the need for an extra PC or other host devices running the Bluetooth Host.

- Scenario 2: the ESP32 system is used only as a Bluetooth Controller, and an extra device running the Bluetooth Host is required (such as a Linux PC running BlueZ or an Android device running BLUEDROID, etc). In this scenario, Controller and Host are implemented on different devices, which is quite similar to the case of mobile phones, PADs or PCs.
- Scenario 3: This scenario is similar to Scenario 2. The difference lies in that, in the BQB controller test (or other certifications), ESP32 can be tested by connecting it to the test tools, with the UART being enabled as the IO interface.

1.1.2. Selection of the HCI Interfaces

In the ESP32 system, only one IO interface at a time can be used by HCI, meaning that if UART is enabled, other interfaces such as the VHCI and SDIO are disabled. In the ESP-IDF (V2.1 and later versions), you can configure the Bluetooth HCI IO interface as VHCI or UART in the ***menuconfig*** screen, as shown below:

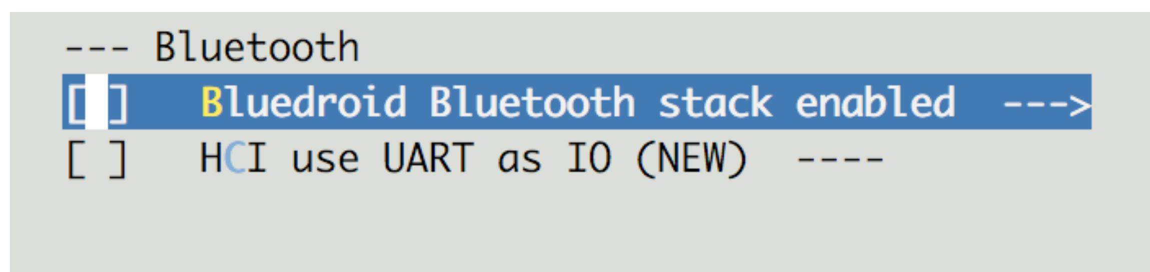


Figure 1-2.Configuration of the HCI IO interface

When the ***Bluedroid Bluetooth stack enabled*** option is selected , VHCI is enabled as the IO interface and the ***HCI use UART as IO (NEW)*** option will disappear from the menu. When the ***HCI use UART as IO (NEW)*** option is selected, UART is enabled as the IO interface. Currently, other IOs are not supported in ESP-IDF. If you want to use other IOs, such as the SPI, a SPI-VHCI bridge is required.

Option 1:

When the ***Bluedroid Bluetooth stack enabled*** option is selected, the following screen is displayed:

```
--- Bluedroid Bluetooth stack enabled
(3072) Bluetooth event (callback to application) task stack size
[ ]    Bluedroid memory debug
[ ]    Classic Bluetooth
[ ]    Release DRAM from Classic BT controller
[*]    Include GATT server module(GATTS)
[*]    Include GATT client module(GATTC)
[*]    Include BLE security module(SMP)
[ ]    Close the bluedroid bt stack log print
(4)    BT/BLE MAX ACL CONNECTIONS(1~7)
```

Figure 1-3.VHCI configuration

Here, users can configure the following items:

- ***Bluetooth event (callback to application) task stack size:*** sets the size of the BTC Task);
- ***Bluedroid memory debug:*** debugs the BLUEDROID memory;
- ***Classic Bluetooth:*** enables the Classic Bluetooth;
- ***Release DRAM from Classic BT Controller:*** releases the DRAM from the Classic Bluetooth Controller;
- ***Include GATT server module (GATTS):*** includes the GATTS module;
- ***Include GATT client module (GATTC):*** includes the GATTC module;
- ***Include BLE security module (SMP):*** includes the SMP module;
- ***Close the bluedroid bt stack log print:*** closes the BLUEDROID printing;
- ***BT/BLE MAX ACL CONNECTIONS (1~7):*** sets the maximum number of ACL connections.

Option 2:

When the ***HCI use UART as IO*** option is selected, the following screen is displayed:

```
--- HCI use UART as IO
(1)     UART Number for HCI (NEW)
(921600) UART Baudrate for HCI (NEW)
```

Figure 1-4.UART configuration

Users can configure the "***UART Number for HCI (NEW)***" (UART port number) and the "***UART Baudrate for HCI (NEW)***" (the baud rate of UART) here. It should also be mentioned here that CTS/RTS must be supported, in order to enable the UART as the HCI IO interface.

1.1.3. Bluetooth Operating Environment

The default operating environment of ESP-IDF is dual-core FreeRTOS. ESP32 Bluetooth can assign function-based tasks with different priorities. The tasks with the highest priority are the ones running the Controller. The Controller tasks, which have higher requirements of real time, have the highest priority in the FreeRTOS system except for the IPC tasks, which are mainly for the interprocess communication between the dual-core CPUs. BLUEDROID (the default ESP-IDF Bluetooth Host) contains four tasks in total, which run the BTC, BTU, HCI UPWARD, and HCI DOWNWARD.

1.2. Architecture

1.2.1. Controller

The Bluetooth Controller of ESP32 supports both the Classic BT and BLE (V4.2). The Controller has integrated a variety of functions, including H4 protocol, HCI, Link Manager, Link Controller, Device Manager, and HW Interface. These functions are provided to the developers in the form of libraries, while some APIs that can access the Controller are also provided. For details, see *readthedocs*.

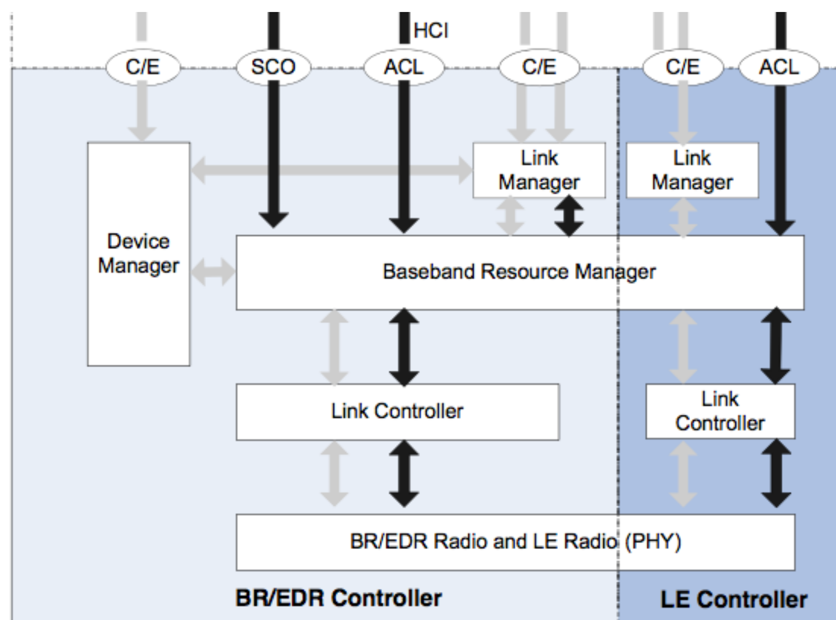


Figure 1-5. Architecture of the Classic BT & BLE controller (from the SIG BT CORE 4.2)

1.2.2. BLUEDROID

1.2.2.1. Overall Architecture

In ESP-IDF, the significantly modified BLUEDROID is used as the Bluetooth Host (Classic BT + BLE). The BLUEDROID, which has a complete set of functions and supports most of the commonly-used standards and architectural designs, is relatively complicated. However, the modified BLUEDROID retains most of the codes below the BTA layer and

almost completely deletes the BTIF layer code, using a leaner BTC layer as the built-in specification and Misc control layer. The architecture of the modified BLUEDROID and its relationship with the Controller are shown in the figure below:

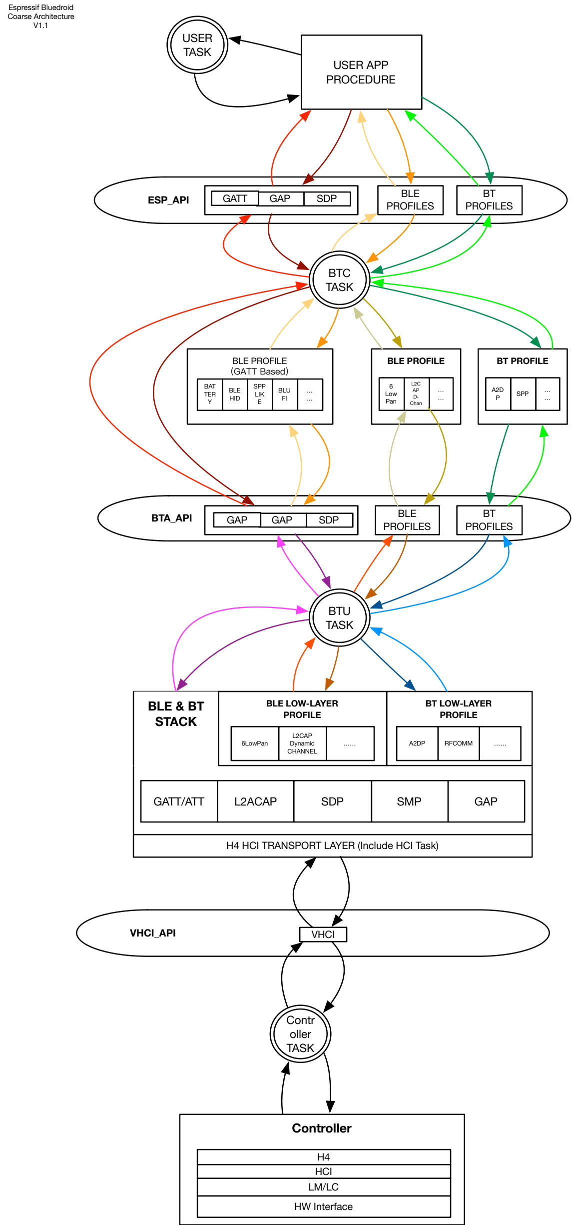


Figure 1-6.ESP32 BLUEDROID diagram

📖 ***Note:***

This diagram mainly describes the hierarchy of the BLUEDROID ~~BLUEDRIOD~~, rather than details, such as the HCI TASK. The detailed information about each layer can be seen below.

As shown in the figure above, the BLUEDROID can be divided into two layers mainly, which are the BTU layer and the BTC layer (except for HCI). Each layer is responsible for its corresponding tasks. More specifically, the BTU layer is mainly responsible for processing the bottom layer protocol stacks of the Bluetooth Host, including L2CAP, GATT/ATT, SMP,

GAP, and other profiles. The BTU layer provides an interface prefixed with "bta". The BTC layer is mainly responsible for providing a supported interface, prefixed with “esp”, to the application layer, processing GATT-based profiles and handling miscellaneous tasks. All the APIs are located in the ESP_API layer. Developers should use the Bluetooth APIs prefixed with “esp".

The figure above does not describe the HCI layer in detail. In fact, the HCI layer has two tasks which process the Downward and Upward data (in the designs of ESP-IDF V2.1 and older versions).

The design logic behind this architecture is to minimize the load on the User Tasks and streamline the Bluetooth structure by handing over the Bluetooth-related tasks to the BTC layer.

Due to legacy reasons and actual demand, some of the Classic Bluetooth profiles, such as RFCOMM and A2DP, as well as other lower layer protocols are implemented in the BTU layer, while some of the protocols that are related to procedural controls, or require the ESP-API, are implemented in the BTC layer.

Some of the profiles and lower layer functions of the Bluetooth Low Energy, such as the 6LowPan or Dynamic L2CAP Channel, will be implemented in the BTU layer, thus providing the application layer with the ESP-API through the BTC.

1.2.2.2. OS-related Adaptation

Some interfaces that are related with the system in BLUEDROID require OSI adaptation. The functions involved include `Timer (Alarm)`, `Task (Thread)`, `Future Await/Ready (Semphore)`, and `Allocator/GKI (malloc/free)`.

The FreeRTOS Timer in BLUEDROID has been packaged as an Alarm, and is used to start the timer which triggers certain tasks.

In BLUEDROID, the POSIX Thread has been replaced with the FreeRTOS tasks, and uses the FreeRTOS Queue to trigger tasks (i.e. wake up).

In BLUEDROID, the `Future Await/Ready` function is used to achieve Blocking. Future Lock packages the xSemphoreTake of FreeRTOS as the `future_await` function, and packages the xSemphoreGive as the `future_ready` function. It is worth noting that the `future_await` and `future_ready` functions cannot be called in the same task context.

In BLUEDROID, `malloc/free` in the standard library is packaged as the Allocator function that reserves (mallocs) or frees memory. Besides, the GKI function also uses `malloc/free` as the core function of `GKI_getbuf/GKI_freebuf`.

1.2.3. Bluetooth Directory Introduction

In the ***component/bt*** screen of the ***ESP-IDF***, you can see the following sub-folders and sub-files:

```
├── Kconfig
├── bluedroid
|   ├── api
|   ├── bta
|   ├── btc
|   ├── btcore
|   ├── btif
|   ├── device
|   ├── external
|   ├── gki
|   ├── hci
|   ├── include
|   ├── main
|   ├── osi
|   ├── stack
|   └── utils
├── bt.c
├── component.mk
├── include
|   └── bt.h
└── lib
    ├── LICENSE
    ├── README.rst
    └── libbtdm_app.a
```

Figure 1-7. Component/bt in ESP-IDF

The detailed description of each sub-folder and sub-file can be found in the table below:

Table 1-1. Description of component/bt in ESP-IDF

Dictionary	Description	Remarks
├── *Kconfig*	Menuconfig files	–
├── *bluedroid*	BLUEDROID home entry	–
│ ├── *api*	The API directory, which includes all the APIs (except for those that are related to the Controller)	–
│ ├── *bta*	The Bluetooth adaptation layer, which is suitable for the interface of some bottom layer protocols in the host.	–
│ ├── *btc*	The Bluetooth control layer, which controls the upper-layer protocols (including profiles) and miscellaneous items in the host.	–
│ ├── *btcore*	Some of the original `feature/bdaddr` conversion functions	To be abandoned
│ ├── *btif*	Some `call out` functions used by the BTA layer	To be abandoned
│ ├── *device*	Related to the device control of the Controller, e.g. the basic set of HCI CMD controller processes	–

Dictionary	Description	Remarks
│ ├── *external*	Codes that are not directly related to the Bluetooth, but are still usable, e.g. the SBC codec software programs	–
│ ├── *gki*	The management codes that are commonly used by the BLUEDROID memory, e.g. the buffer and queue.	–
│ ├── *hci*	HCI layer protocols	–
│ ├── *include*	The top-layer BLUEDROID directory	–
│ ├── *main*	Main program (mainly to start or halt the process)	–
│ ├── *osi*	OS interfaces (including semaphore/timer/thread, etc.)	–
│ ├── *stack*	The bottom layer protocol stacks in the Host (GAP/ATT/GATT/SDP/SMP, etc.)	–
│ └── *utils*	Practical utilities	–
├── *bt.c*	Controller-related processing files	–
├── *component.mk*	makefile	–
├── *include*	Controller-related header file directory	–
│ └── *bt.h*	Header files that contain the controller-related APIs	–
└── *lib*	Controller library directory	–
├── *LICENSE*	License	–
├── *README.rst*	Readme files	–
└── *libbtdm_app.a*	Controller library	–

2. Classic Bluetooth

This chapter introduces the Classic Bluetooth in ESP-IDF.

2.1. Overview

The Bluetooth Host Stack in ESP-IDF originates from BLUEDROID and has been adapted to embedded applications. At the lower layer, the Bluetooth Host Stack communicates with the Bluetooth dual-mode Controller via the virtual HCI interface. At the upper layer, the Bluetooth Host stack provides the profiles and APIs for stack management to the user applications.

Protocols define the message formats and the procedures aimed to accomplish specific functions, e.g. data transportation, link control, security service and service information exchange. Bluetooth profiles, on the other hand, define the functions and features required of each layer in the Bluetooth system, from PHY to L2CAP, and any other protocols outside the core specification.

Below are the Classic BT profiles and protocols currently supported in the Host Stack.

- Profiles: GAP, A2DP(SNK), AVRCP(CT)
- Protocols: L2CAP, SDP, AVDTP, AVCTP

The protocol model is depicted in Figure 2-1.

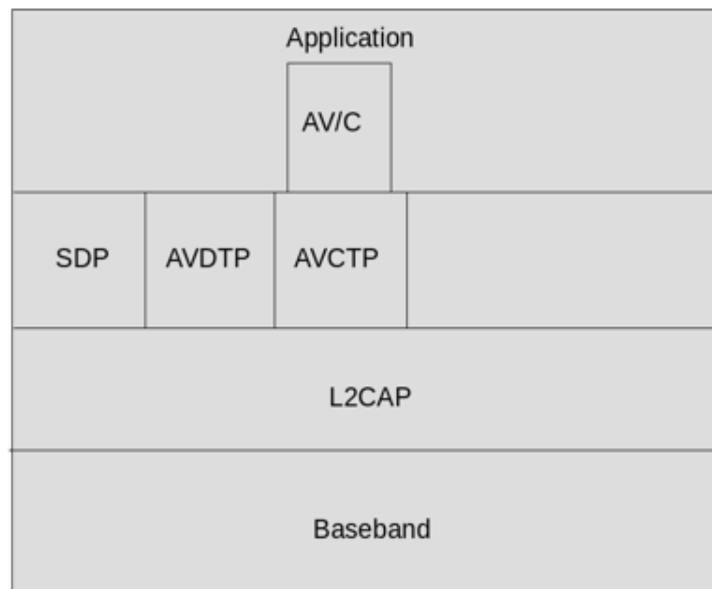


Figure 2-1. Profile Dependencies

In Figure 2-1, L2CAP and SDP are necessary in a minimal Host Stack for Classic Bluetooth. AVDTP, AV/C and AVCTP are outside the core specification and are used by specific profiles.

2.1.1. L2CAP

As an OSI layer 2 Bluetooth protocol, the Bluetooth Logical Link Control and Adaptation Protocol (L2CAP) supports higher level protocol multiplexing, packet segmentation and reassembly, as well as the delivery of service information quality. L2CAP makes it possible for different applications to share an ACL-U logical link. Applications and service protocols interface with L2CAP, using a channel-oriented interface, to create connections to equivalent entities on other devices.

L2CAP channels may operate in one of the six modes selected through the L2CAP channel configuration procedure. The operation modes are distinguished from the QoS that they can provide, and are utilized in different application conditions. These modes are:

- Basic L2CAP Mode
- Flow Control Mode
- Retransmission Mode
- Enhanced Retransmission Mode
- Streaming Mode
- LE Credit-Based Flow Control Mode

For ACL-U logical links, the supported operation modes are the Basic L2CAP Mode, Enhanced Retransmission Mode and Streaming Mode. For other features, the L2CAP Signaling channel is the supported fixed channel, while the Frame Check Sequence (FCS) is also a supported option.

2.1.2. SDP

The Service Discovery Protocol (SDP) provides a means for applications to discover services offered by a peer Bluetooth device, as well as to determine the characteristics of the available services. The SDP involves communication between an SDP server and an SDP client. A server maintains a list of service records that describe the characteristics of services associated with the server. A client can retrieve this information by issuing an SDP request.

Both SDP client and server are implemented in the Host stack, and this module is only used by profiles, such as A2DP and AVRCP, and does not provide APIs for user applications at the moment.

2.1.3. GAP

The Generic Access Profile (GAP) provides a description of the modes and procedures in device discoverability, connection and security.

For the time being, only a limited number of GAP APIs are provided to the Classic Bluetooth Host Stack. An application can make use of these APIs, as if they were a "passive device" which could be discovered by and connected to peer Bluetooth devices. However, APIs used for initiating the inquiry procedure are not currently provided to customers (user applications).

As for the security aspect, the IO capability is hard-coded as "No Input, No Output". Therefore, only the "Just Works" association model in the Secure Simple Pairing is supported. The storage of the link key is done automatically in the Host.

More GAP APIs for Classic Bluetooth are coming up next. Security APIs that are more powerful and supportive of other association models will be provided in the near future. APIs for device discovery and link policy settings will also be given at a later stage.

2.1.4. A2DP and AVRCP

The Advanced Audio Distribution Profile (A2DP) defines the protocols and procedures that realize the distribution of high-quality audio content in mono or stereo on ACL channels. A2DP handles audio streaming and is often used together with the Audio/Video Remote Control Profile (AVRCP), which includes the audio/video control functions. Figure 2-2 depicts the structure and dependencies of the profiles[1]:

> 📖 ***Note*:**
>
> *[1]: Advanced Audio Distribution Profile Specification, Revision 1.3.1.*

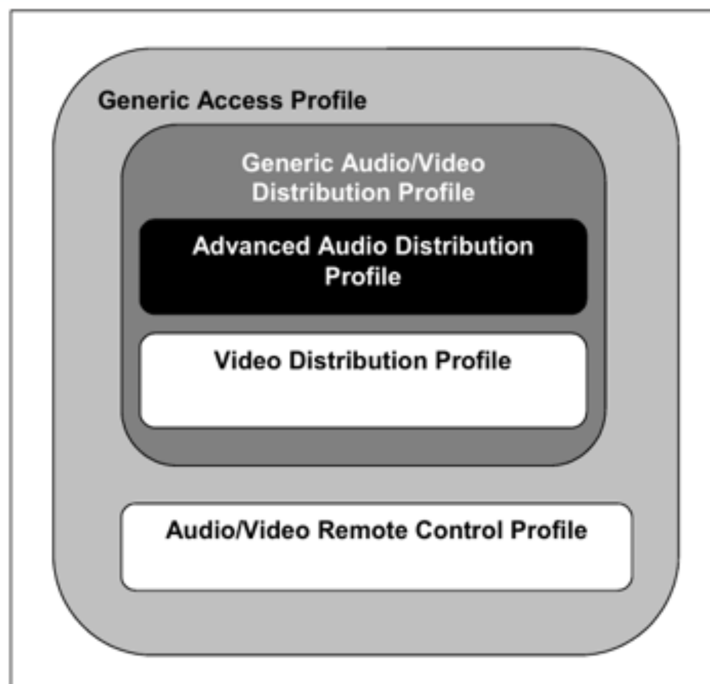


Figure 2-2. Profile Dependencies

As indicated in Figure 2-2, the A2DP is dependent upon the GAP, as well as the Generic Audio/Video Distribution Profile (GAVDP), which defines procedures required to set up an audio/video streaming.

Two roles are defined in A2DP: Source (SRC) and Sink (SNK). SRC functions as a source of a digital audio stream and SNK functions as a sink of a digital audio stream delivered from the SRC.

Two roles defined in AVRCP are Controller (CT) and Target (TG). CT is a device that initiates a transaction by sending a command frame to a target. Examples of CT include personal computers, PDAs and mobile phones. TG is a device that receives a command frame and accordingly generates a response frame. Audio players or headphones are examples of TG. For the time being, A2DP (SRC) and AVRCP (CT) are supported and the device can work as a loudspeaker which can also send remote control messages to the audio source.

In the current A2DP solution, the only audio codec supported is SBC, which is mandated in the A2DP specification. A2DP Version 1.2 and AVDTP Version 1.2 are implemented.

Audio/Video Distribution Transport Protocol (AVDTP) defines the binary transactions between Bluetooth devices for a streaming set-up, and media streaming for audio and video using L2CAP. As the basic transport protocol for A2DP, AVDTP is built upon the L2CAP layer and consists of a signaling entity for negotiating streaming parameters and a transport entity that handles the streaming itself.

The basic service of AVDTP transport capabilities is mandated by the A2DP specification. According to the configuration of current service capabilities, Media Transport and Media Codec in the basic service capability are provided.

AVRCP defines the requirements necessary for the support of the Audio/Video remote control use case.

The commands used in AVRCP fall into three main categories. The first one is the AV/C Digital Interface Command Set, which is applied only on certain occasions and is transported with the Audio/Video Control Transport Protocol (AVCTP). Browsing commands are included in the second category, which provides browsing functionality over another transport channel called the AVCTP browsing channel. The third category, Cover Art Commands, is used to transmit images associated with media items, and is provided through the protocol defined in the Bluetooth Basic Imaging Profile (BIP) with the OBEX protocol.

Two sets of AV/C commands are used in AVRCP. The first one includes the PASS THROUGH command, UNIT INFO command and SUBUNIT INFO command, which are defined in the AV/C specification. The second set includes AVRCP-specific AV/C commands which are defined as a Bluetooth SIG Vendor-Dependent extension. AV/C commands are sent over the AVCTP control channel. A PASS THROUGH command is used to transfer a user operation via a button from the Controller to the panel subunit, which provides a simple and common mechanism to control the target. For example, the operation IDs in PASS THROUGH include common instructions such as Play, Pause, Stop, Volume Up and Volume down.

AVRCP arranges the A/V functions in four categories to ensure interoperability:

- Player/Recorder
- Monitor/Amplifier

- Tuner
- Menu

In the current implementation, AVRCP Version 1.3 and AVCTP Version 1.4 are provided. The default configuration for AVRCP-supported features is Category 2: Monitor/Amplifier. Also, APIs for sending PASS THROUGH commands are provided.

A2DP and AVRCP are often used together. In the current solution, the lower Host stack implements AVDTP and the AVCTP logic, while providing interfaces for A2DP and AVRCP independently. In the upper layer of the stack, however, the two profiles combined make up the "AV" module. The BTA layer, for example, provides a unified "AV" interface, and in BTC layer there is a state machine that handles the events for both profiles. The APIs, however, are provided separately for A2DP and AVRCP.

3. Bluetooth Low Energy

This chapter describes the architecture of the Bluetooth Low Energy in ESP32.

3.1. GAP

3.1.1. Overview

This section mainly introduces the implementation and use of the BLE GAP APIs in ESP32. The GAP (the Generic Access Profile) defines the discovery process, device management and the establishment of device connection between BLE devices.

The BLE GAP is implemented in the form of API calls and Event returns. The processing result of API calls in the protocol stack is returned by Events. When a peer device initiates a request, the status of that peer device is also returned by an Event.

There are four GAP roles defined for a BLE device:

- Broadcaster: A broadcaster is a device that sends advertising packets, so it can be discovered by the observers.This device can only advertise, but cannot be connected.
- Observer: An observer is a device that scans for broadcasters and reports this information to an application. This device can only send scan requests, but cannot be connected.
- Peripheral: A peripheral is a device that advertises by using connectable advertising packets and becomes a slave once it gets connected.
- Central: A central is a device that initiates connections to peripherals and becomes a master once a physical link is established.

3.1.2. Status Transitions among GAP Roles

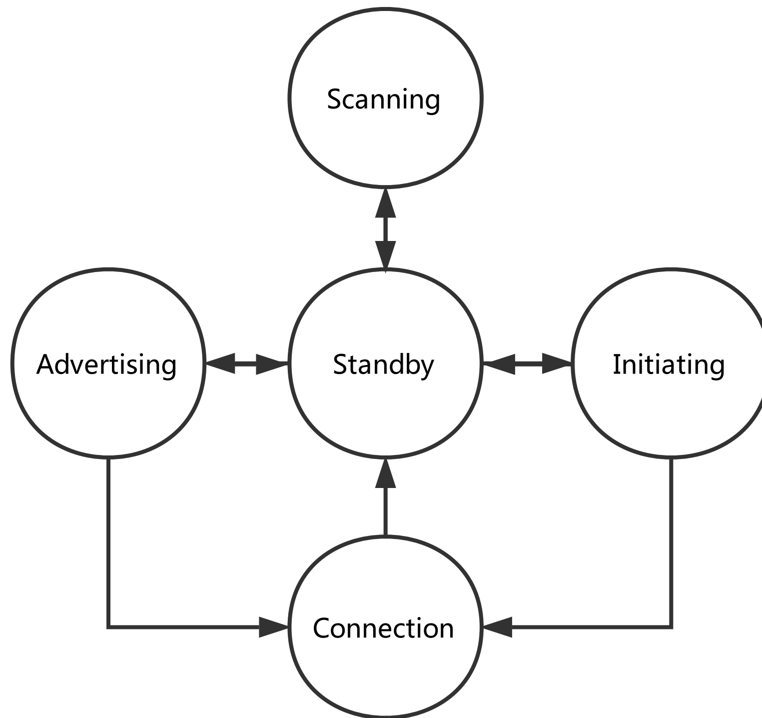


Figure 3-1.The status transitions among GAP roles

3.1.3. BLE Broadcast Procedure

3.1.3.1. Broadcast using a public address

When a public address is used for broadcasting, the `own_addr_type` of `esp_ble_adv_params_t` must be set to `BLE_ADDR_TYPE_PUBLIC`. The broadcast flowchart is as follows:

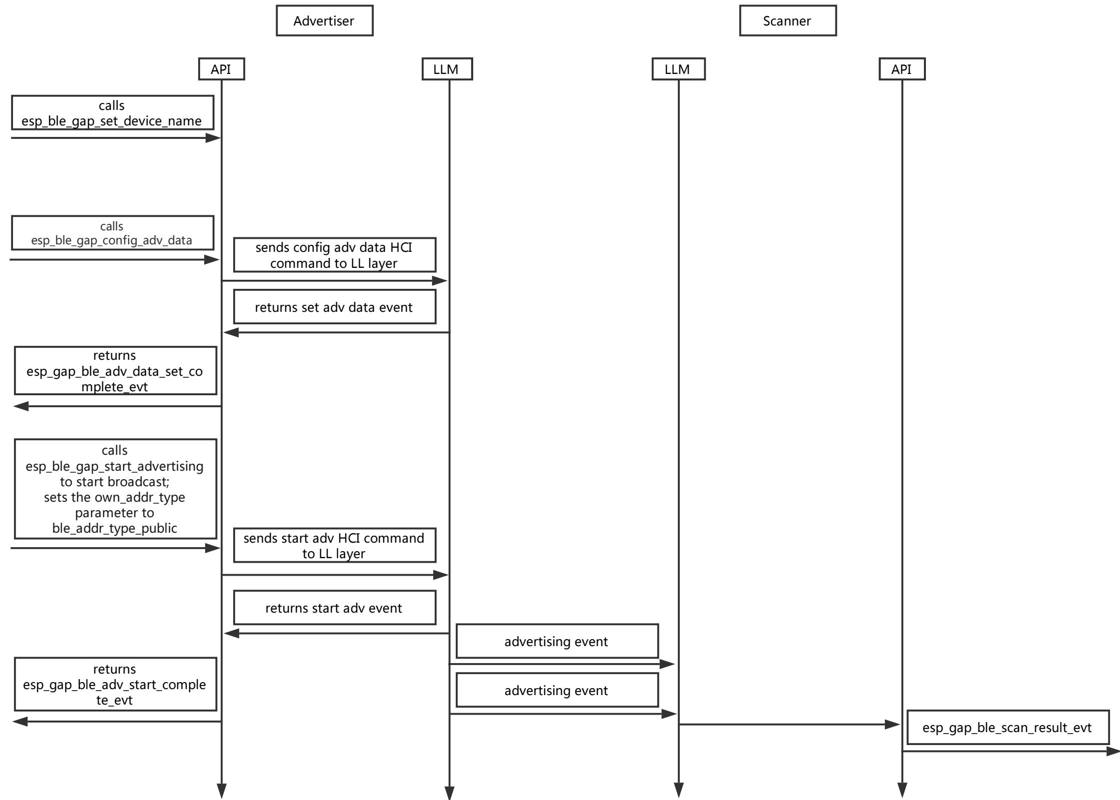


Figure 3-2.Broadcast using a public address

3.1.3.2. Broadcast using a resolvable address

When a resolvable address is used for broadcasting, the underlying protocol stack updates the broadcast address every 15 minutes, and the `own_addr_type` of `esp_ble_adv_params_t` must be set to `BLE_ADDR_TYPE_RANDOM`. The broadcast flowchart is as follows:

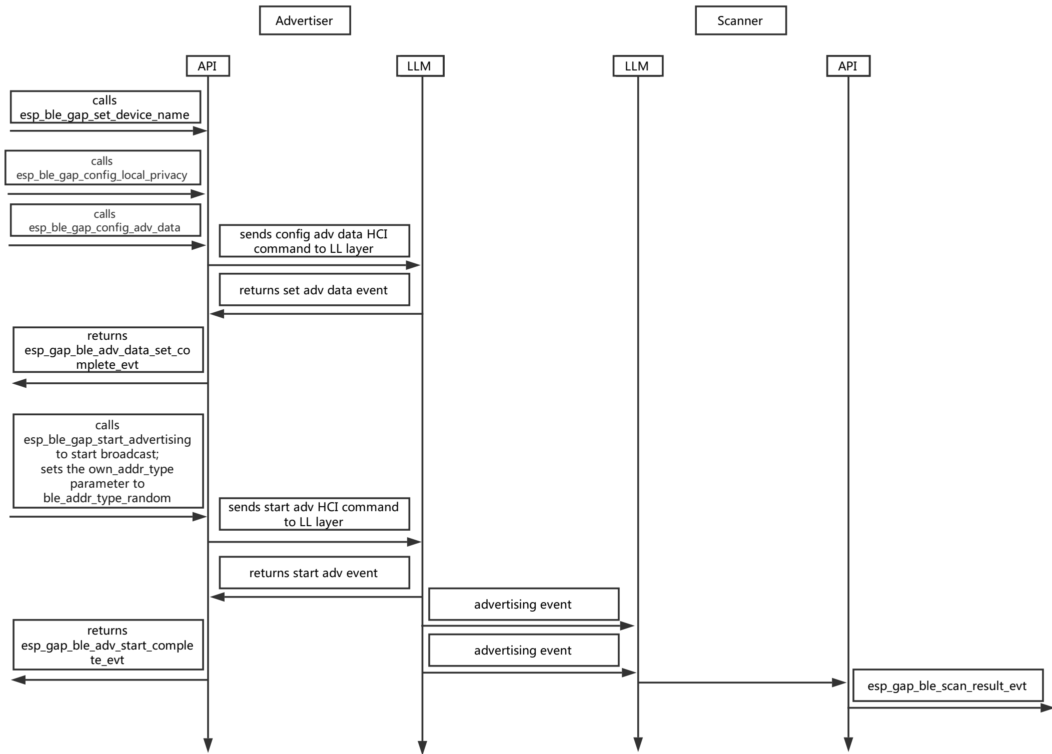


Figure 3-3.Broadcast using a resolvable address

> ⚠ ***Note:***
>
> *When a resolvable address is used for broadcasting, the broadcast only starts after the* `esp_ble_gap_config_local_privacy` *event is returned, and the* `own_addr_type` *type, a broadcast parameter, must be set to* `BLE_ADDR_TYPE_RANDOM`*.*

3.1.3.3. Broadcast using a static random address

When a static random address is used for broadcasting, the `own_addr_type` of the `esp_ble_adv_params_t` must be set to `BLE_ADDR_TYPE_RANDOM`, which is similar to the case of broadcasting using a resolvable address. The broadcast flowchart is as follows:

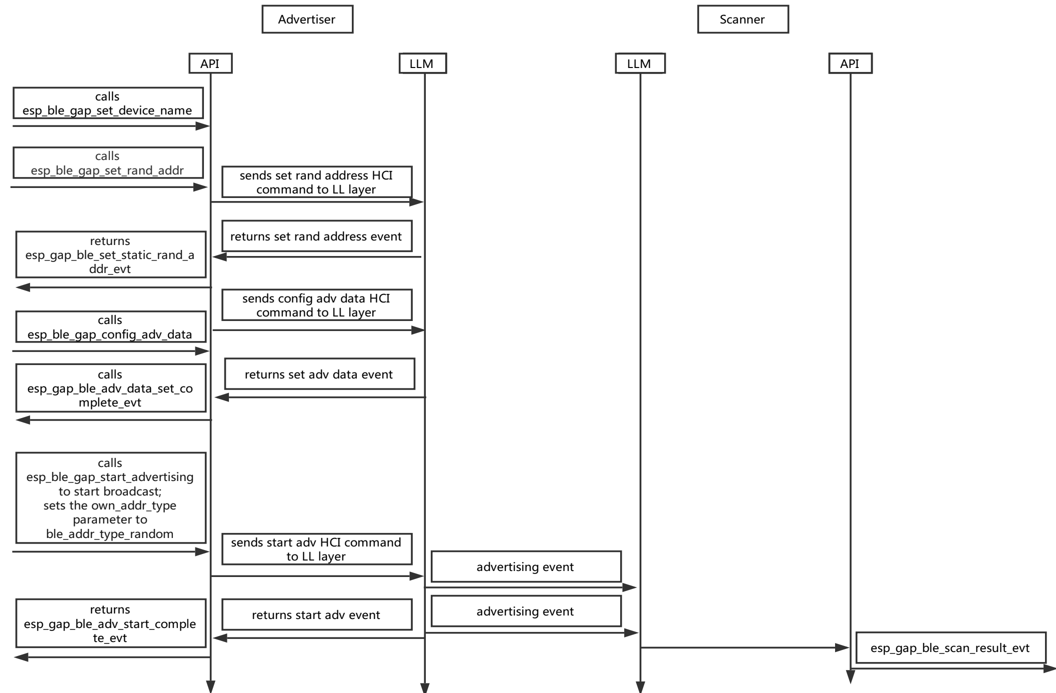


Figure 3-4. Broadcast using a static random address

3.1.4. BLE Modes

Five modes are defined for the BLE broadcasts: Connectable Scannable Undirected mode, High Duty Cycle Directed mode, Scannable Undirected mode, Non-connectable Undirected mode, and Connectable Low Duty Cycle Directed mode.

3.1.4.1. Connectable Scannable Undirected Mode

Table 3-1.Packet structure

Payload	
AdvA (6 octets)	AdvData (0~31 octets)

A device in the Connectable Scannable Undirected mode can be discovered by and connected to any device. Scannability indicates that the local device needs to reply with a scan response, when a peer device sends a scan request.

As shown in the table above, a Connectable Scannable Undirected broadcast packet includes 6 bytes of the broadcast address and 0 ~ 31 bytes of the broadcast packet data. When a static random address is used for broadcasting, the broadcast address is specified by calling `esp_ble_gap_set_rand_addr`. When a public address or a resolvable address is used for broadcasting, the broadcast address is generated automatically by the protocol stack.

3.1.4.2. High Duty Cycle Directed Mode and Connectable Low Duty Cycle Directed Mode

Table 3-2.Packet structure

Payload	
AdvA (6 octets)	InitA (6 octets)

The IP directed broadcasts can only be discovered by and connected to the designated devices.

As shown in the table above, the High Duty Cycle Directed Broadcast packet includes 6 bytes of the broadcasting device's address and 6 bytes of the receiving device's address. In this mode, the broadcast parameters `adv_int_min` and `adv_int_max` are ignored.

In the Connectable Low Duty Cycle Directed mode, the broadcast parameters `adv_int_min` and `adv_int_max` must be greater than 100 ms (0xA0).

> 📖 ***Note:***
>
> *IP directed broadcasts do not carry Adv Data.*

3.1.4.3. Scannable Undirected Mode

In the Scannable Undirected mode, a device can be discovered by any other device, but it cannot get connected to it.

Table 3-3.Packet structure

Payload	
AdvA (6 octets)	AdvData (0~31 octets)

As shown in the table above, a Scannable Undirected packet includes 6 bytes of a broadcast address and 0~31 bytes of the broadcast packet data, which is the same structure as in the Connectable Scannable Undirected packet. A device in this mode can only be scanned by any device, but it cannot be connected to it.

3.1.4.4. Non-connectable Undirected Mode

In the Non-connectable Undirected mode, a device can be discovered by any device, but it can neither be scanned by, nor connected to any other devices. An unscannable device is one that will not reply with a scan response, when a peer device sends a scan request. A disconnectable device is one that cannot be connected to any device.

Table 3-4.Packet structure

Payload	
AdvA (6 octets)	AdvData (0~31 octets)

As shown in the table above, a Non-connectable Undirected broadcast packet also includes 6 bytes of broadcast address and 0~31 bytes of broadcast packet data. In this mode, a device can be discovered but cannot be scanned nor be connected by other devices.

3.1.5. BLE Broadcast Filtering Policy

In ESP32's BLE architecture, the broadcast filtering policy is implemented by setting the `adv_filter_policy` enumeration type, which has the following four values:

- `ADV_FILTER_ALLOW_SCAN_ANY_CON_ANY`
- `ADV_FILTER_ALLOW_SCAN_WLST_CON_ANY`
- `ADV_FILTER_ALLOW_SCAN_ANY_CON_WLST`
- `ADV_FILTER_ALLOW_SCAN_WLST_CON_WLST`

The four values correspond to 4 cases, respectively:

- Can be scanned and connected to any device (no white list)
- Handles all of the connection requests, and only the scan requests in the whitelist
- Handles all of the scan requests, and only the connection requests in the whitelist
- Handles the connection requests and scan requests in the whitelist

3.1.6. BLE Scanning Procedure

In ESP32, the scanning device sets the parameters of the scan mainly by calling esp_ble_gap_set_scan_params, and then starts the scan by calling

esp_ble_gap_start_scanning. The information of the scanned device will be returned by the ESP_GAP_BLE_SCAN_RESULT_EVT event, or by the ESP_GAP_SEARCH_INQ_CMPL_EVT event when the duration times out.

> ⚠️ ***Notice:***
>
> *When the value of the duration is 0, the device will be scanned permanently without timeout.*

3.1.7. BLE GAP Implementation Mechanism

ESP32 calls the BLE GAP APIs, registers BLE GAP Callback and obtains the status of the current device by the returned value of the Event.

3.2. GATT

3.2.1. ATT

The data inside the BLE architecture exists in the form of Attributes that consist of four basic elements:

- Attribute Handle: an Attribute Handle can help us locate any Attribute, which is similar to using an address to locate data in the memory. For example, the Handle of the first attribute is 0x0001 and the second one is 0x0002, and so on, up to 0xFFFF.
- Attribute Type: Each data set exposes a certain type of information, such as temperature, transmit power, battery level and so on. The type of the data that is exposed is called *attribute type*. Given the different possible types of data that can be exposed, a 16-bit or 128-bit number, also known as UUID, is used to identify the type of the attribute. For example, UUID `0x2A09` is for Battery Level and UUID `0x2A6E` is for Temperature.
- Attribute value: the attribute value is the key information of each attribute, while the other three elements (handle, type and permission) are added so the attribute value can be better understood. The length of the attribute value for different attribute types can be fixed or variable. For example, the length of the attribute value in Battery Level is only 1 byte, which is enough to cover all the possible values of a Battery Level attribute, i.e. 0-100, while the attribute length of a BLE-enabled pass-through module is variable.
- Attribute Permission: Each attribute contains information that can only be read or written. To facilitate these restrictions upon access, each and every attribute has its own attribute permissions. The party that owns the data can control the read/write access of its local data through the attribute permissions.

Table 3-5. Attribute Structure Table

Attribute Handle	Attribute Type	Attribute Value	Attribute Permission
`0x0001`	-	-	-
`0x0002`	-	-	-

Attribute Handle	Attribute Type	Attribute Value	Attribute Permission
………	-	-	-
0xFFFE	-	-	-
0xFFFF	-	-	-

The device that holds the data (i.e. the attributes) is defined as a server, and the device that obtains the data from the server is defined as a client. The common operations between a server and a client are listed below:

- **A client sends data to a server** by writing data into the server. Both the Write Request and Write Command can be used to write an attribute value. However, a Write Response is only prompted when a Write Request is used.
- **A server sends data to a client** by sending an Indication or Notification to the client. The only difference between an Indication and a Notification is that a Confirmation is prompted only when an Indication is used. This is similar to the difference between a Write Request and a Write Command.
- **A client can also obtain data from the server by initiating a Read Request.**

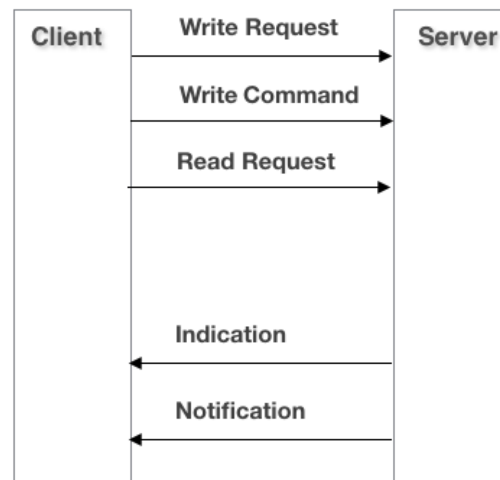


Figure 3-5. Common operations between a server and a client

Notice:

For detailed information regarding the common operations between a server and a client, please refer to Core_V5.0, Vol3. Part F, Chapter 3.4 "Attribute Protocol PDUs".

The common operations between a server and a client are achieved by using ATT PDU. Each device can specify its supported MTU, which is the maximum length of an ATT message. In ESP32 IDF, the MTU can be 23 - 517 bytes, and the total length of an attribute value is not limited.

If the length of the user's packet is greater than (MTU-3), a Prepare Write Request is required to write data. Similarly, if the length of the packet is greater than (MTU-1), a Read Blob Request is required to read the remaining data.

> ⚠️ ***Notice:***
>
> *The difference between the MTU and the LE Packet Length of a single physical packet should be highlighted here. The MTU, which is about the Host ATT layer in this case, determines whether all the "data to be sent" can fit into a single ATT Request and also whether a Prepare Write Request is required to send the data. Meanwhile, the LE Packet Length, which is about the PHY layer, determines whether the packet needs to be divided and transmitted in several packages. For example, if (MTU + 4) is greater than the LE Packet Length, this ATT packet needs to be divided and transmitted in more than one physical packets. On the contrary, if (MTU+4) is shorter than the LE Packet Length, only one physical packet is required to send the whole ATT packet. The reason we add 4 bytes to the MTU is because a 4-byte L2CAP header will be added during the transmission.*

3.2.2. GATT Profile

The ATT specifies the minimum data storage unit in the BLE architecture, while the GATT defines how to represent a data set using attribute values and descriptors, how to aggregate similar data into a service, and how to find out what services and data a peer device owns.

The GATT introduces the concept of Characteristics, which are about information that is not purely numerical, as in the cases outlined below:

- The unit of a given value, for example, weight measured in kilograms (kg), temperature measured in Celsius (°C), and so on.
- The name of a given value. For example, for characteristics with the same UUID, e.g. temperature attribute, the name of the value informs the peer device that this value indicates "the temperature in the master bedroom", while the other one indicates "the temperature in the living room".
- The exponent of excessive data numbers, such as 230,000 and 460,000. Given that the exponent is already specified as 10^4, transmitting only "23" and "46" is enough to represent 230,000 and 460,000.

These are just a few examples of the many existing requirements for describing data accurately in actual applications. In order to provide more nuanced information, a large piece of data space should be reserved to store this additional information in each characteristic. However, in many cases, most of the extra space reserved will not be used. Such a design, then, will not comply with BLE's prerequisite to have as concise as possible protocols. In cases like this, the GATT specification introduces the concept of descriptors to outline this additional information. It must be noted that each piece of data and descriptor do not have a one-to-one correspondence, that is, complex data can have multiple descriptors, while simple data can have no descriptors at all.

A characteristic is composed of three basic elements:

- Characteristic Declaration: a declaration is the start of a characteristic, informing a peer device that the content following the declaration is the characteristic value. All

the handles between two declarations compose a complete characteristic. The write and read properties are also included in a declaration.

- Characteristic Value: a characteristic value is the main part of a characteristic, which carries the most important information of a characteristic.
- Descriptor: Descriptors can further describe characteristics (e.g. providing configuration information) and a characteristic can have multiple or no descriptors.

In BLE, the GAP groups similar functions together in the form of Services. For example, all of the characteristics and behaviors related to the battery can be defined as a Battery Service; all of the characteristics and behaviors related to the heart rate can be defined as a Heart Rate Service; and all of the characteristics and behaviors related to the weight scale can be defined as a Weight Scale Service.

A Service typically includes one or more characteristics and each characteristic includes zero or many descriptors. Users can select the required services based on their own application requirements, and form the final application.

A completed service definition table is shown below:

Table 3-6. The definition table of services

Attribute Handle	Attribute Type
`0x0001`	Service 1
`0x0002`	Characteristic Declaration 1
`0x0003`	Characteristic Value 1
`0x0004`	Descriptor 1
`0x0005`	Characteristic Declaration 2
`0x0006`	Characteristic Value 2
`0x0007`	Descriptor 2
`0x0008`	Descriptor 3
`0x0009`	Service 2
………	………

> ⚠️ ***Notices:***
>
> *For other definitions of services, characteristics and descriptors, please refer to:*
>
> - *Chapter 3 "Service Interoperability Requirements", Part G, Vol3., Core_V5.0;*
> - *https://www.bluetooth.com/zh-cn/specifications/gatt*

3.2.3. Add Gatt Services in ESP32 IDF Environment

Users can add services and characteristics manually, through events, one by one in ESP32 IDF Release 1.0. All of the read-write operations will reach the application layer through events and users can respond to them with packages. This approach is prone to errors by users who are not familiar with the BLE protocol, especially when adding services in a large GATT database. So adding services manually one by one is not recommended.

> ⚠️ ***Notice:***
>
> *The interface and examples of adding services and characteristics manually are still reserved in ESP IDF. For more information, please refer to the gatt_service example.*

In this context, adding services and characteristics with an attribute table is introduced in ESP32 IDF Release 2.0. Users can add new services and characteristics by simply entering them in an attribute table and, then, calling `esp_ble_gatts_create_attr_tab`. Additionally, a lower layer response is also supported in this case, meaning the lower layer is able to respond to some requests and identify errors automatically, so the users can focus on receiving and sending data.

In this way, users can port profiles to the ESP32 platform from other platforms easily, without the need to implement the BLE specifications all over again.

> ⚠️ ***Notice:***
>
> *We recommend that users add services and characteristics through the attribute table, which is much easier, less error-prone, and supports low-layer responses. For details, please refer to the* ***gatt_server_service_table*** *examples.*

The structure of an attribute table defines all of the parameters that require initialization to describe an attribute through `esp_gatts_attr_db_t:`

Table 3-7. The structure parameters of ESP32 IDF

Parameter	Description
`uint8_t attr_control`	Defines some responses, such as the `write_response,` which are given by the lower layer automatically or passed to the application layer, so that users can respond manually. The `ESP_GATT_AUTO_RSP` automatic response mode is recommended.
`uint16_t uuid_length`	Indicates that the length of UUID is 16 bits, 32 bits or 128 bits. Since the attribute UUID is transmitted by pointers, the length of the UUID has to be specified.

Parameter	Description
`uint8_t *uuid_p`	Indicates the pointer of the UUID of the current attribute. Users can read a certain length of the UUID value, based on the length information specified in the uuid_lenght parameter.
`uint16_t perm`	Indicates the write and read permissions of the current attribute. This parameter is bitwise-operated. Each bit represents a specific write and read permission. Operating a certain bit can change the write and read permission of the corresponding attribute. For example, PERM_READ | PERM_WRITE means an attribute that can be read and written.
`uint16_t max_length`	Indicates the maximum length of the current attribute value. The protocol stack allocates memory to the attribute, based on this parameter. If the length of the attribute value that a peer device has written exceeds the maximum length defined in this parameter, a write error is returned, indicating that the error is due to the length of the write operation exceeding the maximum length of the data.
`uint16_t length`	Indicates the actual length of the current attribute. For example, in case the maximum length of the attribute is 512 bits and a peer device wants to write "0x1122" into this attribute, we will set the current length of the attribute to 2. When a peer device, then, reads this attribute, we can obtain the actual length of this attribute from the memory, sending only the part with the actual values, instead of the whole 512 bits.
`uint8_t  *value`	Indicates the initialized values of the current attribute value. Since the format of this parameter is a pointer, the actual length of this parameter should be obtained first by the `length` parameter, in order to get the correct value from the pointer.

3.2.4. Discover a Peer Device's Services in ESP32 IDF (GATT Client)

The Discovering Service can help a GATT Client to discover a peer device's services and characteristics. The discovery procedure can be different for different devices. The discovery procedure of ESP32 IDF is introduced here, along with an example of how to discover a peer device's GATT service.

- Firstly, discover all of the peer devices' services information, including the service UUID and the range of the attribute handle.
  - GATT Service, UUID 0x1801, Handles 0x0001~0x0005
  - GAP Service, UUID 0x1800, Handles 0x0014~0x001C
- Then, discover all of the peer devices' characteristics within the handle range of a GATT service (0x0001~0x0005).
  - Find "Service Change Characteristic", Handles 0x0002~0x0003
  - 0x0002 represents the characteristic declaration
  - 0x0003 represents the characteristic value
  - So each characteristic has the attributes of at leasts two handles.

- Given that the handles of a GATT Service lie in the range of 0x0001~0x0005, 0x0003, for example, should be followed by the corresponding descriptor. All descriptors, then, should be sought from 0x0004 onwards.
  - 0x0004 represents the descriptor of the Client Characteristic Configuration
  - 0x0005 does not have any information at the moment, and could be a handle reserved for this service.
- At this point, the discovery procedure of GATT Service is complete.

3.3. SMP

This chapter mainly introduces the implementation and use of the ESP32 BLE SMP (Security Management Protocol).

3.3.1. Overview

The SMP-related APIs have been packaged in ESP32 BLE's GAP module.

The SMP generates encryption keys and identity keys, defines a convenient protocol for pairing and key distribution, and allows the other layers in the protocol stack to connect and exchange data with other devices safely. A connection in the data link layer and certain security standards are required in this process. The GAP SMP allows two devices encrypting their connection in the data link layer by setting such security levels as those in the SMP chapter of the Bluetooth Core Specification version. Before introducing the implementation of the GAP SMP, we should clarify the following concepts:

- **Pairing** indicates that two devices have agreed to establish a connection with certain security levels.
- **Bonding** indicates that at least one device has sent some kind of indication or security information, which could be an LTK, CSRK or IRK, to another device for future connections. If these two devices can bond with each other, the key distribution occurs after the pairing, otherwise no bonding information will be exchanged. Bonding is not a prerequisite for pairing. However, during pairing the two devices exchange their characteristics to determine whether the peer device is open for bonding. If neither of these two devices is open for bonding, no security information of the peer devices should be stored.
- **Authentication** indicates the security of a link. However, a deauthentication link does not necessarily mean this link is not secure at all. When the key for the link encryption has the security attributes that have been confirmed by both devices, these two devices are considered authenticated. When STK is used for the authentication, a keyword is generated during the pairing. For devices with input/output and OOB functions, all the keys generated and exchanged have the MITM attributes (PIN/larger OOB keys are used, which enforces security). If Just Works is used, all the keys generated and exchanged have the No MITM attributes.
- **Authorization** is defined as the assignment of permission to perform an operation from the application layer. Some applications may require authorization, in which case

the application must be granted permission before being used. If no permission is given, the whole process will fail.

3.3.2. Safety Management Controller

3.3.2.1. BLE Encryption

The encryption of a BLE device can be achieved with two basic methods:

- When no bonding is established between two BLE devices, these devices are encrypted through the pairing procedure, while bonding (or not bonding) is determined according to the specific pairing information of these BLE devices.
- Two bonded devices: initiate the encryption through the bonding procedure. When two devices have already bonded, encryption is initiated with one device resorting to the original bonding process.

The way in which a master initiates an encryption request in Just Works mode can be seen in the flow chart below:

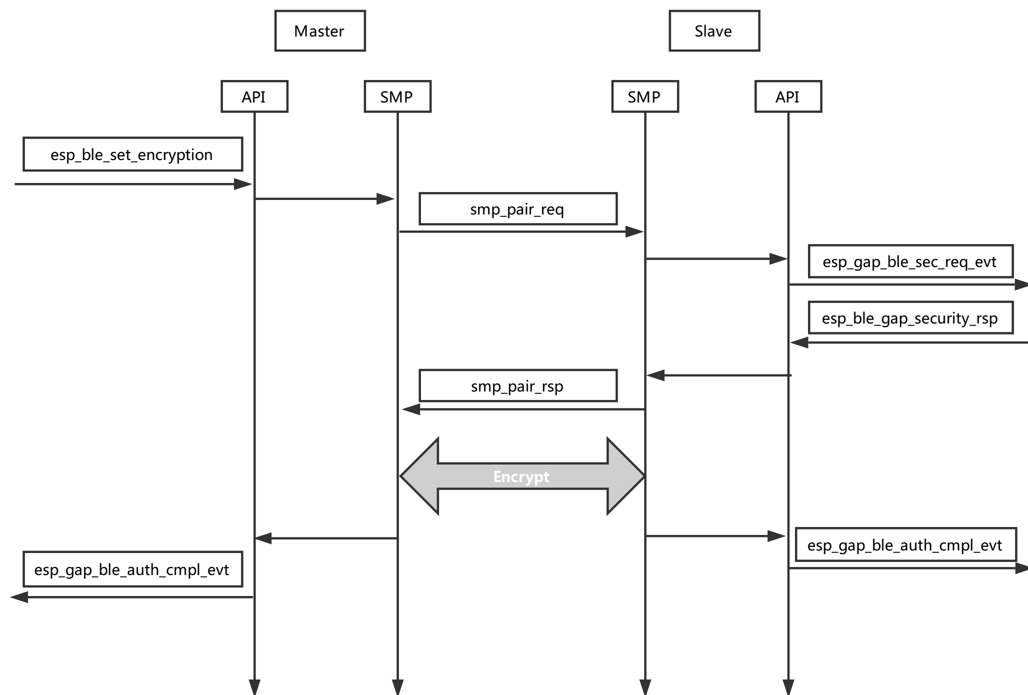


Figure 3-6. The flow chart of encryption in Just Works mode

The way in which a master initiates an encryption request in Passkey Notify mode can be seen in the flow chart below:

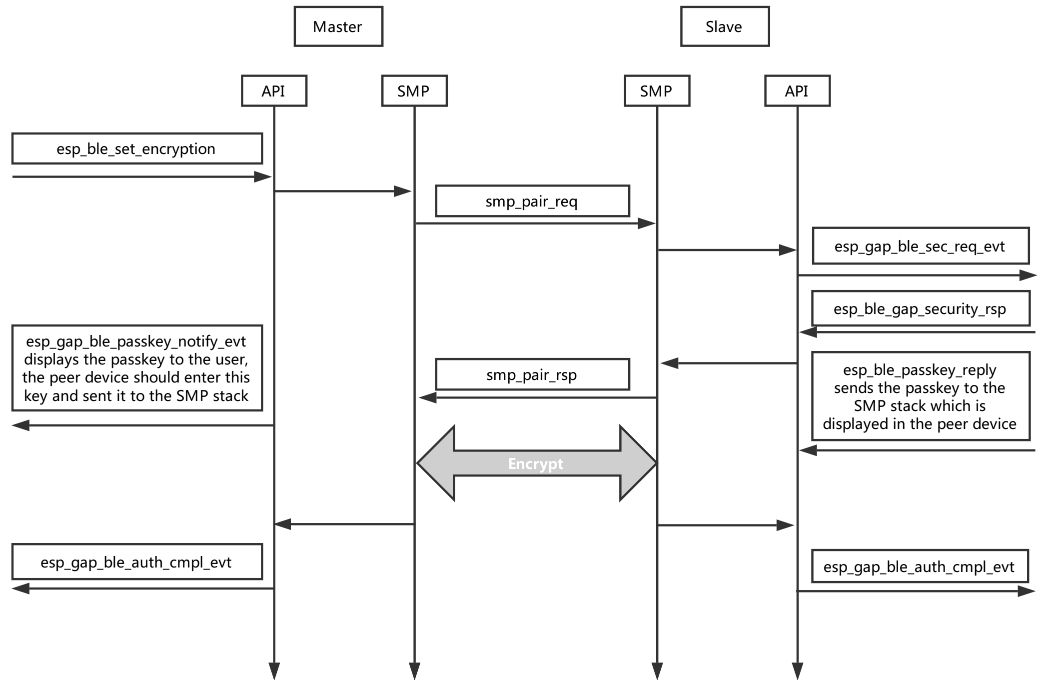


Figure 3-7. The flow chart of encryption in Passkey Notify mode

3.3.2.2.BLE Bonding

The bonding between two BLE devices is achieved by calling a GAP API. According to the description in the Bluetooth Core Specification, the purpose of bonding is that two BLE devices, which have been encrypted by SMP, are able to use the same keys to encrypt a link when they reconnect with each other, thus simplifying the reconnection process. These two BLE devices exchange encryption keys during their pairing, and store them for long-term use. The bonding process can be seen in the flow chart below:

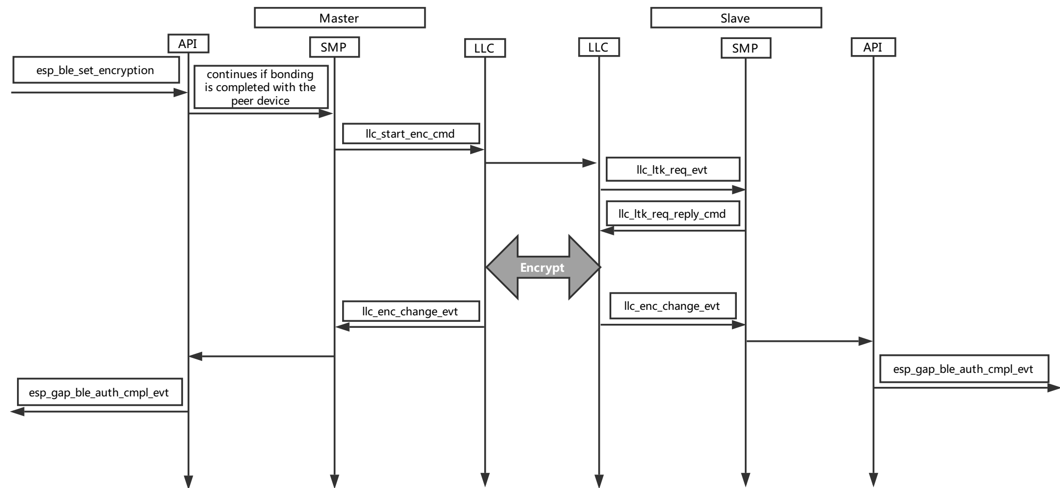


Figure 3-8. The flow chart of BLE bonding process

⚠ ***Notice:***

The bonding process must be initiated by a master device during the connection.

3.3.3. The Implementation of SMP

The BLE SMP calls encryption APIs in BLE GAP, registers the BLE GAP callbacks, and obtains the current encryption status through the return values of events.

Espressif IoT Team
www.espressif.com

Disclaimer and Copyright Notice

Information in this document, including URL references, is subject to change without notice.

THIS DOCUMENT IS PROVIDED AS IS WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

All liability, including liability for infringement of any proprietary rights, relating to use of information in this document is disclaimed. No licenses express or implied, by estoppel or otherwise, to any intellectual property rights are granted herein.

The Wi-Fi Alliance Member logo is a trademark of the Wi-Fi Alliance. The Bluetooth logo is a registered trademark of Bluetooth SIG.

All trade names, trademarks and registered trademarks mentioned in this document are property of their respective owners, and are hereby acknowledged.

Copyright © 2019 Espressif Inc. All rights reserved.