ESP32 蓝牙架构

版本 1.0

关于本手册

本手册为 ESP32 的蓝牙架构简介，主要分三个章节介绍了蓝牙、经典蓝牙和蓝牙低功耗方面的整体架构。注意，本手册仅针对 ESP-IDF V2.1 及以下版本。

章	标题	内容
第 1 章	蓝牙简介	本章节介绍了 ESP32 的蓝牙概况。
第 2 章	经典蓝牙	本章节介绍了 ESP32 的经典蓝牙概况、主要协议及规范。
第 3 章	蓝牙低功耗	本章节介绍了 ESP32 的蓝牙低功耗概况、GATT、GAP 及 SMP。

发布说明

日期	版本	发布说明
2017.11	V1.0	首次发布。

文档变更通知

用户可通过乐鑫官网订阅技术文档变更的电子邮件通知。

目录

1. 蓝牙

本章节介绍了 ESP32 的基本蓝牙架构。

1.1. 概述

1.1.1. 蓝牙应用结构

蓝牙是一种短距通信系统，其关键特性包括鲁棒性、低功耗、低成本等。蓝牙系统分为两种不同的技术：经典蓝牙 (Classic Bluetooth) 和蓝牙低功耗 (Bluetooth Low Energy)。ESP32 支持双模蓝牙，即同时支持经典蓝牙和蓝牙低功耗。

从整体结构上，蓝牙可分为控制器 (Controller) 和主机 (Host) 两大部分：控制器包括了 PHY、Baseband、Link Controller、Link Manager、Device Manager、HCI 等模块，用于硬件接口管理、链路管理等等；主机则包括了 L2CAP、SMP、SDP、ATT、GATT、GAP 以及各种规范，构建了向应用层提供接口的基础，方便应用层对蓝牙系统的访问。主机可以与控制器运行在同一个宿主上，也可以分布在不同的宿主上。ESP32 可以支持上述两种方式，下图罗列了几种典型应用结构：

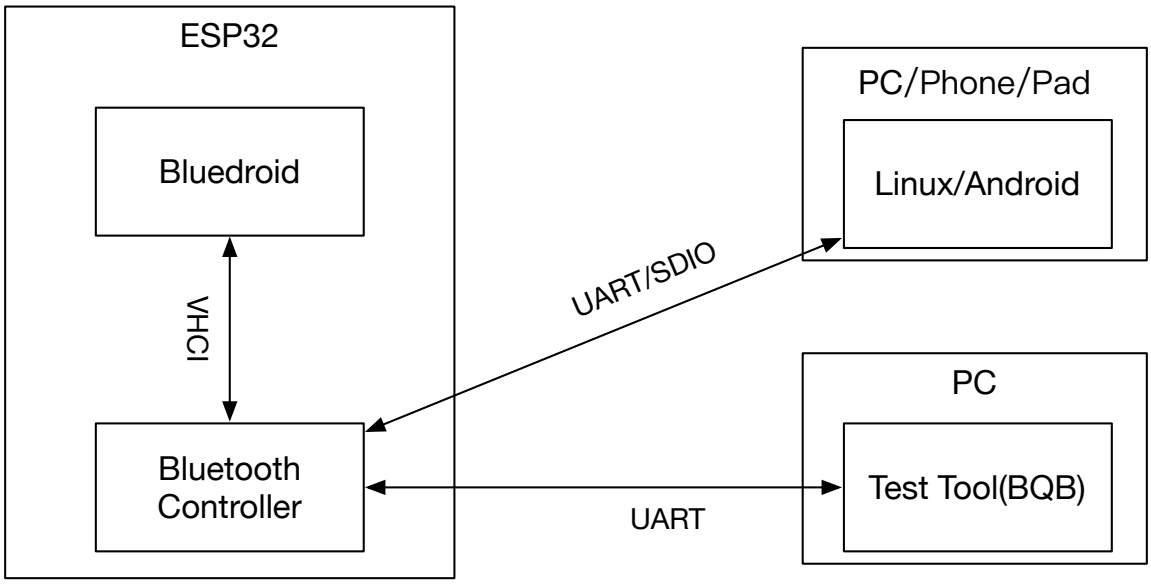

图 1-1. ESP32 蓝牙主机与控制器的关系结构图

- 场景一（ESP-IDF 默认）：在 ESP32 的系统上，选择 BLUEDROID 为蓝牙主机，并通过 VHCI（软件实现的虚拟 HCI 接口）接口，访问控制器。此场景下，BLUEDROID 和控制器都运行在同一宿主上（即 ESP32 芯片），不需要额外连接运行蓝牙主机的 PC 或其它主机设备。

- 场景二：在 ESP32 上运行控制器（此时设备将单纯作为蓝牙控制器使用），外接一个运行蓝牙主机的设备（如运行 BlueZ 的 Linux PC、运行 BLUEDROID 的 Android 等）。此场景下，控制器和主机运行在不同宿主上，与手机、PAD、PC 的使用方式比较类似。
- 场景三：此场景与场景二类似，特别之处在于，在 BQB（或其它认证）的控制器测试下，可以将 ESP32 作为 DUT，用 UART 作为 IO 接口，接上认证测试的 PC 机，即可完成认证。

1.1.2. HCI 接口选择

ESP32 上，HCI 只能同时使用一个 IO 接口，即如使用 UART，则放弃 VHCI、SDIO 等其他 IO 接口。在 ESP-IDF（V2.1 以后）中，可以在 ***menuconfig*** 中将蓝牙的 HCI IO 接口方式配置为 VHCI 或 UART，如下图：

```
--- Bluetooth
[ ]   Bluedroid Bluetooth stack enabled  --->
[ ]   HCI use UART as IO (NEW)  ----
```

图 1-2. HCI IO 接口方式配置

若选中 ***Bluedroid Bluetooth stack enabled***，则表示使用 VHCI 作为 IO 方式，那么 ***HCI use UART as IO (NEW)*** 选项会消失；若选中 ***HCI use UART as IO (NEW)***，则表示使用 UART 作为 IO 方式；目前，ESP-IDF 暂时不支持其他 IO，如需使用其他方式（如 SPI 等），则需开发 SPI-VHCI 的 bridge 模块。

选项一：

进入 ***Bluedroid Bluetooth stack enabled*** 选项时，可以看到如下配置。

```
--- 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)
```

图 1-3. VHCI 配置

如上图所示，用户可在此界面配置：

- ***Bluetooth event (callback to application) task stack size***（BTC Task 大小）；
- ***Bluedroid memory debug***（BLUEDROID 内存调试）；
- ***Classic Bluetooth***（使能经典蓝牙）；
- ***Release DRAM from Classic BT Controller***（从经典蓝牙控制器中释放 DRAM）；
- ***Include GATT server module (GATTS)***（包括 GATTS 模块）；
- ***Include GATT client module (GATTC)***（包括 GATTC 模块）；
- ***Include BLE security module (SMP)***（包括 SMP 模块）；
- ***Close the bluedroid bt stack log print***（关闭 BLUEDROID 打印）；
- ***BT/BLE MAX ACL CONNECTIONS (1~7)***（最大 ACL 连接数）等。

选项二：

进入 ***HCI use UART as IO*** 选项时，可以看到如下配置。

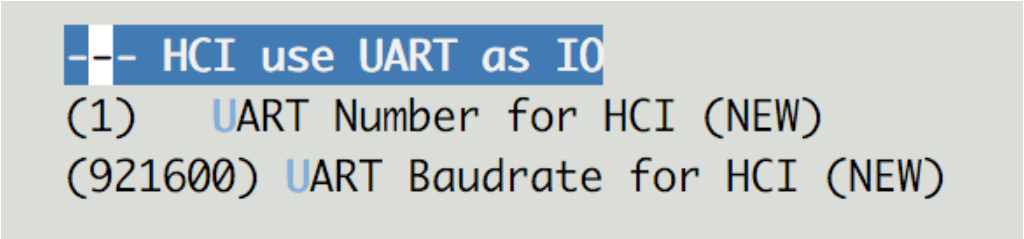

图 1-4. UART 配置

用户可在此界面配置 ***UART Number for HCI (NEW)***（UART 端口号），以及 ***UART Baudrate for HCI (NEW)***（UART 端口的波特率）。其中，UART 模式必须支持硬件流控 (CTS/RTS)。

1.1.3. 蓝牙运行环境

ESP-IDF 的默认运行环境为双核 FreeRTOS，ESP32 的蓝牙可按照功能分为多个任务 (task) 运行，不同任务的优先级也有不同，其中优先级最高的为运行控制器的任务。控制器任务对实时性的要求较高，在 FreeRTOS 系统中的优先级仅次于 IPC 任务（IPC 任务用于双核 CPU 的进程间通信）。BLUEDROID（ESP-IDF 默认蓝牙主机）共包含 4 个任务，分别运行 BTC、BTU、HCI UPWARD，及 HCI DOWNWARD。

1.2. 框架

1.2.1. 控制器

ESP32 的控制器同时支持 Classic BT 和 BLE，支持的蓝牙版本为 4.2。控制器中主要集成了 H4 协议、HCI、Link Manager、Link Controller、Device Manager、HW Interface 等功能。这些功能都以库的形式提供给开发者，并做了一些 API 用来访问控制器，具体请见 *readthedocs*。

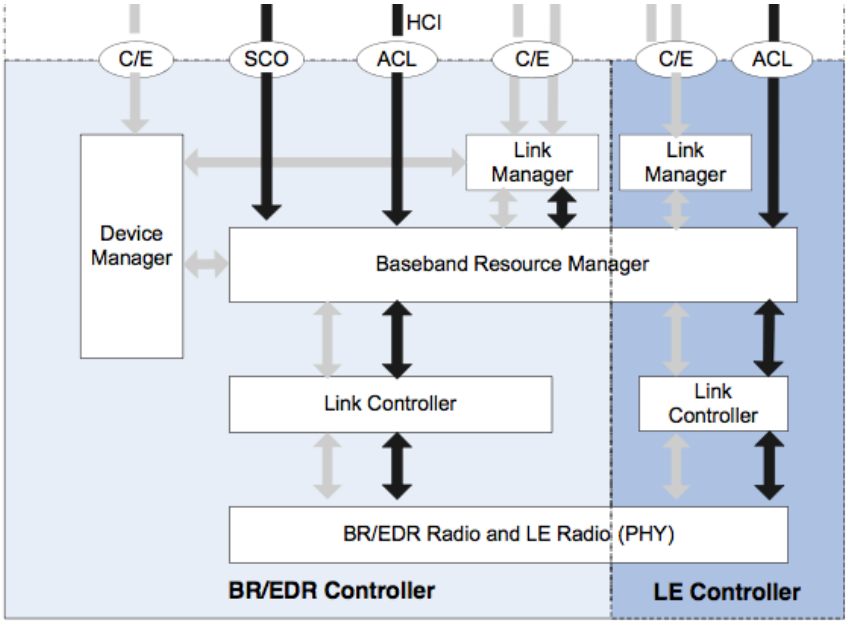

图 1-5. Classic BT & BLE 控制器架构（摘自 SIG BT CORE4.2）

1.2.2. BLUEDROID

1.2.2.1. 主机架构

在 ESP-IDF 中，使用经过大量修改后的 BLUEDROID 作为蓝牙主机 (Classic BT + BLE)。BLUEDROID 拥有较为完善的功能，支持常用的规范和架构设计，同时也较为复杂。经过大量修改后，BLUEDROID 保留了大多数 BTA 层以下的代码，几乎完全删去了 BTIF 层的代码，使用了较为精简的 BTC 层作为内置规范及 Misc 控制层。修改后的 BLUEDROID 及其与控制器之间的关系如下图：

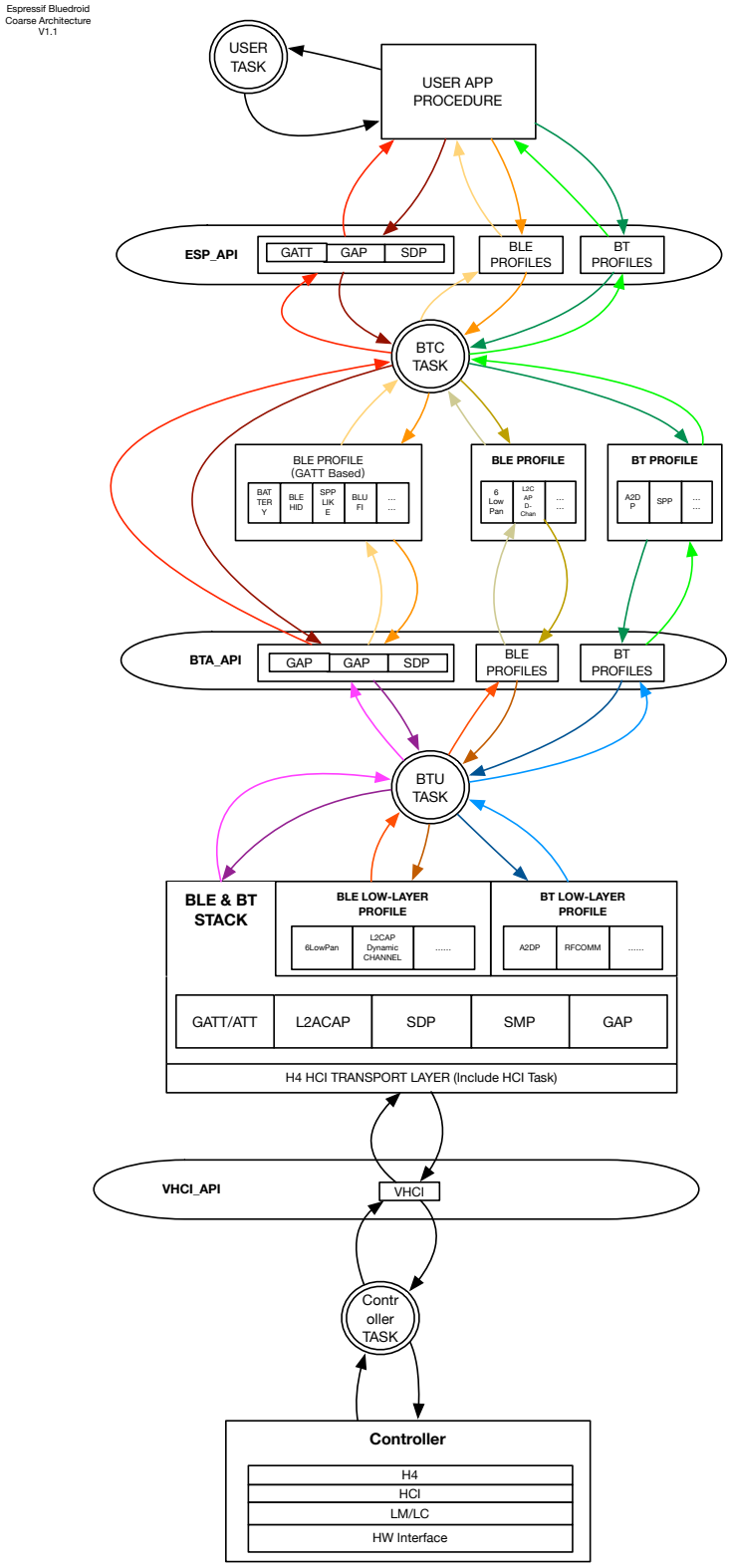

图 1-6. ESP32 BLUEDROID 层次关系图

📖 **说明：**

此架构图主要描述架构层次，部分细节（如 *HCI TASK*）已舍去。下文会较为具体地介绍各层次的内容。

从上图可以看到，BLUEDROID 内部大致分为 2 层：BTU 层和 BTC 层（除去 HCI ），每个层都有对应的任务来处理。BTU 层主要负责蓝牙主机底层协议栈的处理，包括 L2CAP、GATT/ATT、SMP、GAP 以及部分规范等，并向上提供以“bta”为前缀的接口；BTC 层主要负责向应用层提供接口支持、处理基于 GATT 的规范、处理杂项等，并向应用层提供以“esp”为前缀的接口。所有的 API 都在 ESP_API 层，开发者应当使用“esp”为前缀的蓝牙 API（特殊的除外）。

上图并未详细描述 HCI 部分，而事实上，HCI 具有 2 个任务（至少在 ESP-IDF V2.1 以前），分别处理 Downward 和 Upward 的数据。

此框架的其中一条设计思路是尽量将蓝牙相关的任务交给 BTC 来处理，从而避免和降低用户任务 (User Task) 的负载，也使结构上更加简洁。

由于历史原因和实际需求，经典蓝牙的部分规范，如 RFCOMM、A2DP 等，中层次偏协议偏底层的部分运行在 BTU 层，偏控制流程的以及需要提供 ESP-API 的运行在 BTC 层。

蓝牙低功耗的部分规范或偏底层的功能，如 6LowPan 或 Dynamic L2CAP Channel 的功能，将运行在 BTU 层，再通过 BTC 向应用层提供 ESP-API。

1.2.2.2. OS 相关适配

BLUEDROID 中有部分与系统相关的接口需要进行 OSI 适配，涉及到的功能包括 Timer (Alarm)、Task (Thread)、Future Await/Ready (Semphore)、Allocator/GKI (malloc/free) 等。

BLUEDROID 中将 FreeRTOS 的 Timer 封装成 Alarm，用于启动定时器，触发某些特定任务。

BLUEDROID 将原先的 Linux 下的 Thread 部分重新替换成 FreeRTOS 的任务，并使用 FreeRTOS 的 Queue 来触发任务的运行（唤醒）。

BLUEDROID 使用 Future Await/Ready 功能来实现阻塞，Future Lock 将 FreeRTOS 的 xSemphoreTake 包装成 `future_await` 函数，并将 xSemphoreGive 包装成 `future_ready` 函数。值得注意的是，`future_await` 和 `future_ready` 不能在同一任务中调用。

BLUEDROID 将标准库中的 malloc/free 封装成 Allocator 的申请/释放内存的函数，GKI 功能也同样使用 malloc/free 来作为 GKI_getbuf/GKI_freebuf 的核心函数。

1.2.3. 蓝牙目录

进入 ***ESP-IDF*** 的 ***component/bt*** 目录，可以看到有如下子目录和子文件：

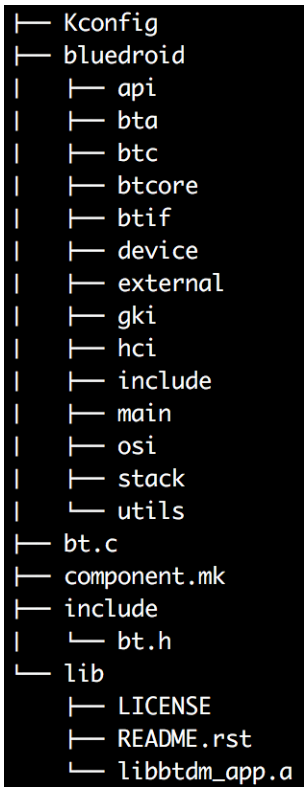

图 1-7. ESP-IDF 的 component/bt 目录

各级目录的具体说明见下表：

表 1-1. ESP-IDF 的 component/bt 目录说明

目录	说明	备注
├── *Kconfig*	实用工具函数	–
├── *bluedroid*	BLUEDROID 主目录	–
│ ├── *api*	API 目录，所有的 API（除 Controller 相关）都在此目录下	–
│ ├── *bta*	蓝牙适配层，适配一些主机底层协议的接口	–
│ ├── *btc*	蓝牙控制层，控制主机上层协议（包括规范）以及杂项的处理	–
│ ├── *btcore*	一些原始的 `feature/bdaddr` 转换函数	欲废弃
│ ├── *btif*	一些 BTA 使用 `call out` 函数	欲废弃
│ ├── *device*	与控制器设备控制相关的，如控制器基本设置的 HCI CMD 流程等	–
│ ├── *external*	与蓝牙自身无关，但又要使用的代码，如 SBC codec 软件程序等	–
│ ├── *gki*	BLUEDROID 内存常用的 buffer、queue 等管理代码	–

目录	说明	备注
│ ├── *hci*	HCI 层协议	–
│ ├── *include*	BLUEDROID 顶层的文件目录	–
│ ├── *main*	主程序目录（主要为启动、关闭流程）	–
│ ├── *osi*	OS 接口相关（包括 semphore/timer/thread 等）	–
│ ├── *stack*	主机底层协议栈（GAP/ATT/GATT/SDP/SMP 等）	–
│ └── *utils*	实用工具函数	–
├── *bt.c*	控制器相关处理文件	–
├── *component.mk*	makefile	–
├── *include*	控制器相关头文件目录	–
│ └── *bt.h*	包含控制器相关 API 的头文件	–
└── *lib*	控制器库目录	–
├── *LICENSE*	License	–
├── *README.rst*	帮助文件	–
└── *libbtdm_app.a*	控制器库	–

2. 经典蓝牙

本章介绍了 ESP-IDF 中的经典蓝牙。

2.1. 概述

ESP-IDF 中的蓝牙主机协议栈源于 BLUEDROID，后经过改良以配合嵌入式系统的应用。在底层中，蓝牙主机协议栈通过虚拟 HCI 接口，与蓝牙双模控制器进行通信；在上层中，蓝牙主机协议栈将为用户应用程序提供用于协议栈管理和规范的 API。

协议 (Protocol) 定义了完成特定功能的消息格式和过程，例如数据传输、链路控制、安全服务和服务信息交换等。另一方面，蓝牙规范 (Profile) 则定义了蓝牙系统中从 PHY 到 L2CAP 及核心规范外的其他协议所需的功能和特性。

目前，主机协议栈支持的经典蓝牙规范和协议如下。

- 规范：GAP、A2DP (SNK)、AVRCP (CT)
- 协议：L2CAP、SDP、AVDTP、AVCTP

协议模型如图 2-1 所示。

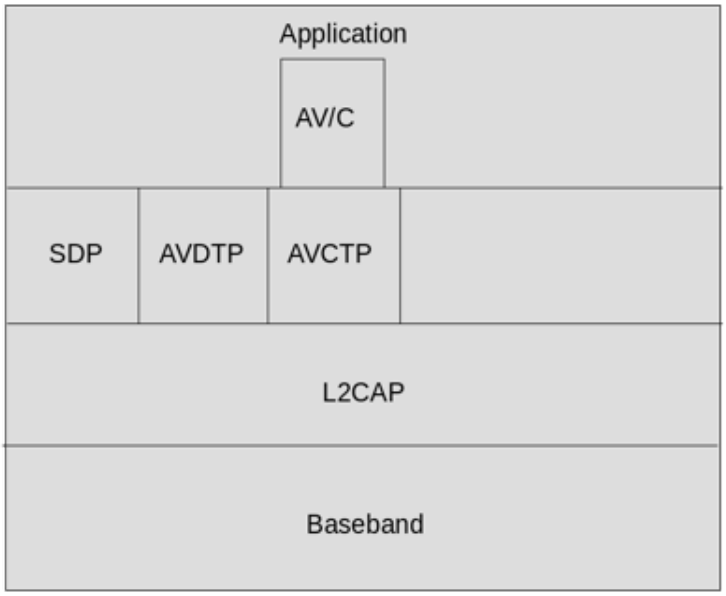

图 2-1. 蓝牙规范关系图

如图 2-1 所示，L2CAP 和 SDP 是经典蓝牙最小主机协议栈的必备组成部分，AVDTP、AV/C 和 AVCTP 并不属于核心规范，仅用于特定规范。

2.2. 协议和规范

2.2.1. L2CAP

“蓝牙逻辑链路控制和适配协议” (L2CAP) 是 OSI 2 层协议，支持上层的协议复用、分段和重组及服务质量信息的传递。L2CAP 可以让不同的应用程序共享一条 ACL-U 逻辑链路。应用程序和服务协议可通过一个面向信道的接口，与 L2CAP 进行交互，从而与其他设备上的等效实体进行连接。

L2CAP 信道共支持 6 种模式，可通过 L2CAP 信道配置过程进行选择，不同模式的应用场合不同，主要差别在于可提供的 QoS 不同。这些模式分别是：

- 基本 L2CAP 模式
- 流量控制模式
- 重传模式
- 加强重传模式
- 流模式
- 基于 LE Credit 的流量控制模式

其中，ACL-U 逻辑链路支持的操作模式包括基本 L2CAP 模式、加强重传模式和流模式。L2CAP 信道为支持的固定信道，也支持“帧校验序列” (FCS)。

2.2.2. SDP

服务发现协议 (SDP) 允许应用程序发现其他对等蓝牙设备提供的服务，并确定可用服务的特征。SDP 包含 SDP 服务器和 SDP 客户端之间的通信。服务器维护一个描述服务特性的服务记录表。客户端可通过发出 SDP 请求，从服务器维护的服务记录表中进行信息检索。

SDP 客户端和服务器都部署在主机协议栈中，该模块仅供 A2DP 和 AVRCP 等规范使用，目前并不为用户应用程序提供 API。

2.2.3. GAP

通用访问规范 (GAP) 可提供有关设备可发现性、可连接性和安全性的模式和过程描述。

目前，经典蓝牙主机协议栈仅提供少数几个 GAP API。应用程序可以将这些 API 用作“被动”设备，被对等蓝牙设备发现并连接。然而，目前暂不向客户（用户应用程序）提供用于发起问询 (Inquiry) 的 API。

在安全方面，IO 功能已经在代码中固定为“无输入，无输出”，因此只支持“蓝牙安全简单配对” (Secure Simple Pairing) 中的“Just Works”关联模型。链路密钥的存储将由主机自动完成。

未来，经典蓝牙将推出更多 GAP API，并提供功能更强大且可支持其他关联模型的安全 API，以及用于设备发现和链接策略设置的 API。

2.2.4. A2DP 和 AVRCP

“高级音频分发规范” (Advanced Audio Distribution Profile, A2DP) 定义了在 ACL 信道上，实现高质量单声道或立体声音频内容传输的协议和过程。A2DP 负责处理音频流，通常与“音频 / 视频远程控制规范” (AVRCP) （包括音频 / 视频控制功能）一起使用。图 2-2 描述了这些规范的结构和关系图[1]：

📖 **注意：**

[1]: 《高级音频分发规范》（修订版*1.3.1*）

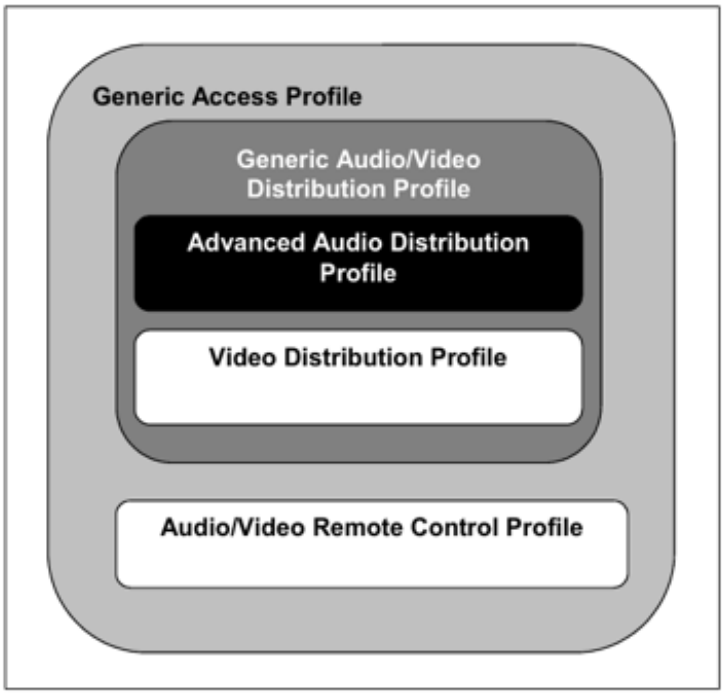

图 2-2. 规范关系图

如图 2-2 所示，A2DP 的工作基于 GAP 以及“通用音频/视频分发规范” (GAVDP)，定义了建立音频 / 视频流的过程。

A2DP 中共定义了两个角色：Source (SRC) 和 Sink (SNK)。SRC 代表数字音频流的源端，SNK 代表数字音频流的接收端。

AVRCP 中共定义了两个角色：控制器 (CT) 和目标 (TG)。控制器可通过向目标发送“命令帧”，发起事务。控制器的常见例子包括个人电脑、PDA 和移动电话等。目标可接收控制器发送的“命令帧”，并生成相应的“响应帧”。目标的常见例子包括音频播放器或耳机。目前，A2DP (SRC) 和 AVRCP (CT) 已经得到支持，设备可以作为扬声器，向音频源发送遥控信息。

在目前的 A2DP 解决方案中，SBC 是唯一支持的音频编解码器，SBC 在 A2DP 规范中是必须的编解码格式。目前方案中实现的规范、协议版本是 A2DP V1.2 和 AVDTP V1.2。

AVDTP 协议定义了蓝牙设备之间在 L2CAP 协议层上建立和传输媒体流的二进制业务。作为 A2DP 的基本传输协议，AVDTP 建立在 L2CAP 层协议之上，由“一个协商媒体流参数的信令传输实体”和“一个传输媒体流的实体”组成。

在 AVDTP 传输功能中，基本服务是 A2DP 规范中所强制要求的。根据当前的服务能力配置，基本服务中可提供“媒体传输”和“媒体编解码器”功能。

AVRCP 定义了支持音频 / 视频遥控的应用场景的各项需求。

AVRCP 中的命令主要分为三个大组：“AV / C 数字接口命令集”，其特定命令子集被采用，且通过 AVCTP 协议传输；“浏览命令”，可通过 AVCTP 浏览信道，提供浏览功能；“封面艺术命令”，用于传输与媒体项目有关的图像，通过基于OBEX 协议的“蓝牙基本图像规范” (BIP) 实现。

AVRCP 使用了其中的两套 AV/C 命令：其一包括 AV/C 规范中定义的 PASS THROUGH、UNIT INFO 和 SUBUNIT INFO 命令；其二是 AVRCP 专用 AV/C 命令，作为对 Bluetooth SIG Vendor Dependent 的扩展。AV/C 命令通过 AVCTP 控制信道发送。PASS THROUGH 命令可通过控制器上的按钮，向面板子单元传送用户操作，并提供一个简单的通用机制来控制目标。例如，PASS THROUGH 中的操作 ID 包括播放、暂停、停止、调高音量和调低音量等常用指令。

为了保证互操作性，AVRCP 将 A/V 功能分为四类：

- 播放机/录像机
- 监控器/放大器
- 调音器
- 菜单

目前的方案提供了 AVRCP V1.3和 AVCTP V1.4。AVRCP 支持功能的默认配置属于第二类，即监视器/放大器。此外，方案还提供了用于发送 PASS THROUGH 命令的 API。

A2DP 和 AVRCP 经常一起使用。在目前的解决方案中，下层主机堆栈实现了 AVDTP 和 AVCTP 逻辑，并独立为 A2DP 和 AVRCP 提供接口。然而，在堆栈上层中，两个规范组合

匹配成为 “AV” 模块。例如，BTA 层提供一个统一的“AV”接口，而在 BTC 层中，状态机将处理两种规范的事务。然而，A2DP 和 AVRCP 的 API 是分别提供的。

3. 蓝牙低功耗

本章节介绍了 ESP32 的蓝牙低功耗功能。

3.1. GAP

3.1.1. 概述

本章节主要介绍了 ESP32 BLE 通用访问规范 (GAP) 接口 API 的实现和使用流程，GAP 协议层定义了 BLE 设备的发现流程，设备管理和设备连接的建立。

BLE GAP 协议层采用 API 调用和事件 (Event) 返回的设计模式，通过事件返回来获取 API 在协议栈的处理结果。当对端设备主动发起请求时，也是通过事件返回获取对端设备的状态。BLE 设备定义了四类 GAP 角色：

- 广播者 (Broadcaster)：处于这种角色的设备通过发送广播 (Advertising) 让接收者发现自己。这种角色只能发广播，不能被连接。
- 观察者 (Observer)：处于这种角色的设备通过接收广播事件并发送扫描 (Scan) 请求。这种角色只能发送扫描请求，不能被连接。
- 外围设备 (Peripheral)：当广播者接受了观察者发来的连接请求后就会进入这种角色。当设备进入了这种角色之后，将会作为从设备 (Slave) 在链路中进行通信。
- 中央设备 (Central)：当观察者主动进行初始化，并建立一个物理链路时就会进入这种角色。这种角色在链路中同样被称为主设备 (Master)。

3.1.2. BLE 设备角色转换状态图

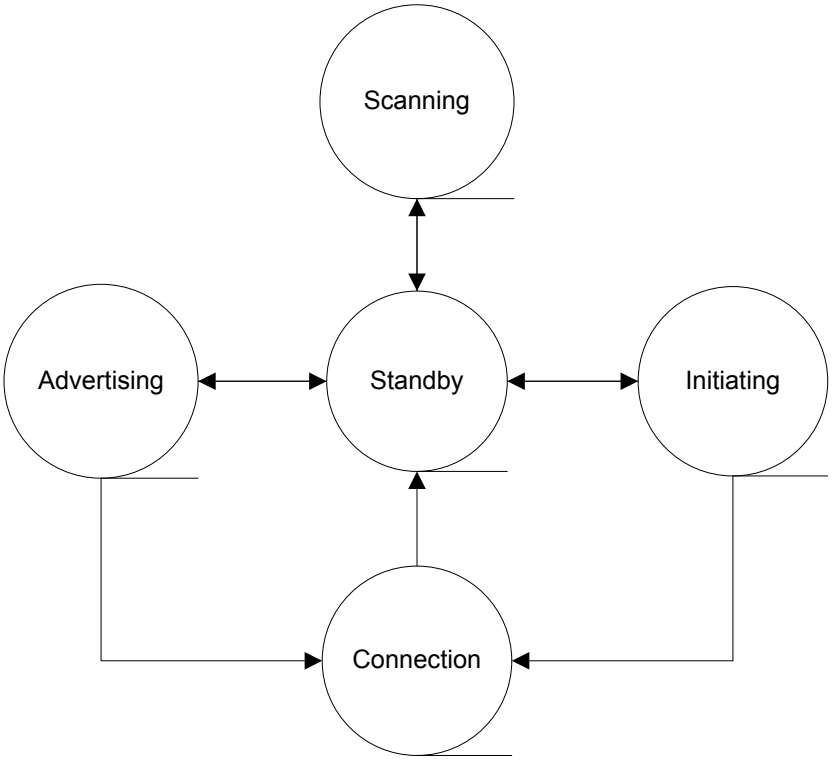

图 3-1. BLE 设备角色转换状态图

3.1.3. BLE 广播流程

3.1.3.1. 使用 public 地址进行广播

使用 public 地址进行广播时，需要将 `esp_ble_adv_params_t` 成员 `own_addr_type` 设置为 `BLE_ADDR_TYPE_PUBLIC`，广播流程图如下：

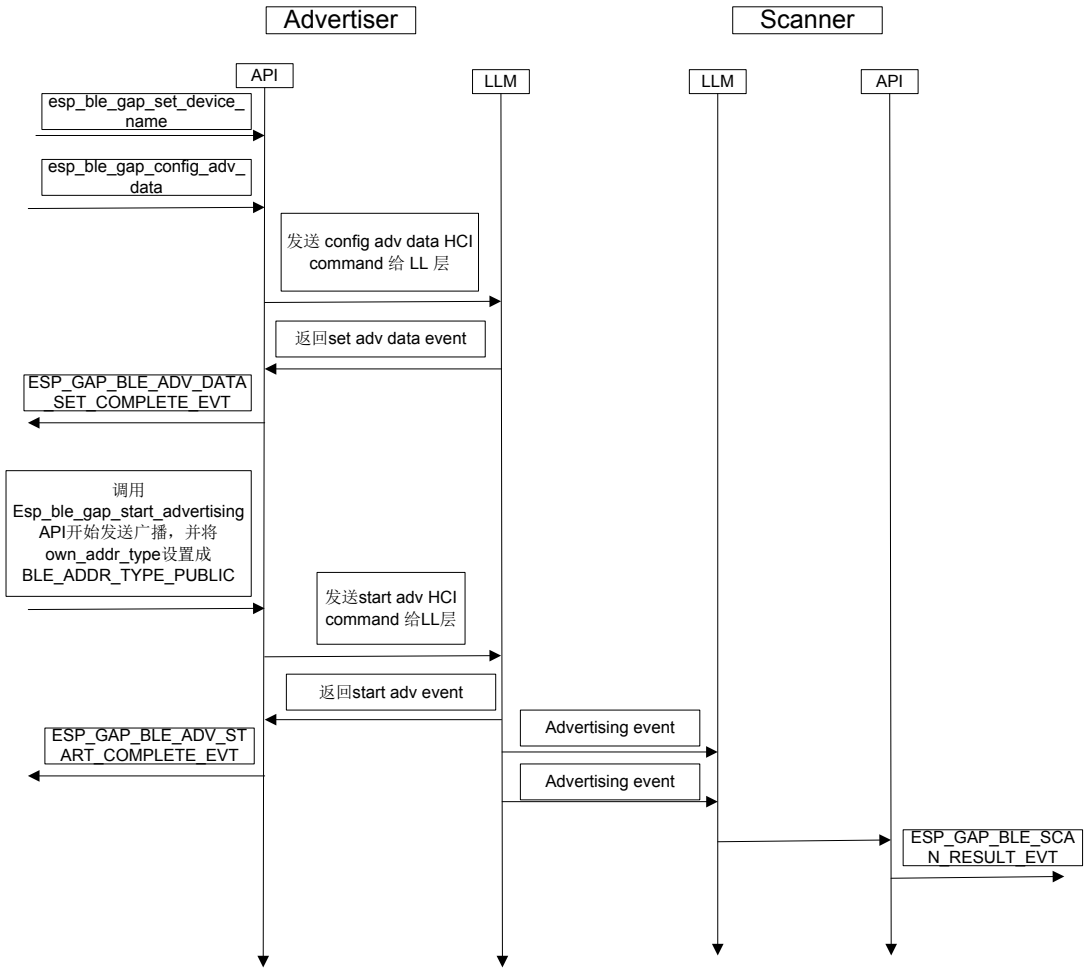

图 3-2. 广播流程图 – 使用 public 地址

3.1.3.2. 使用可解析地址进行广播

使用可解析地址进行广播时，底层协议栈会 15 分钟更新一次广播地址，需要将 esp_ble_adv_params_t 成员 own_addr_type 设置为 BLE_ADDR_TYPE_RANDOM，广播流程图如下：

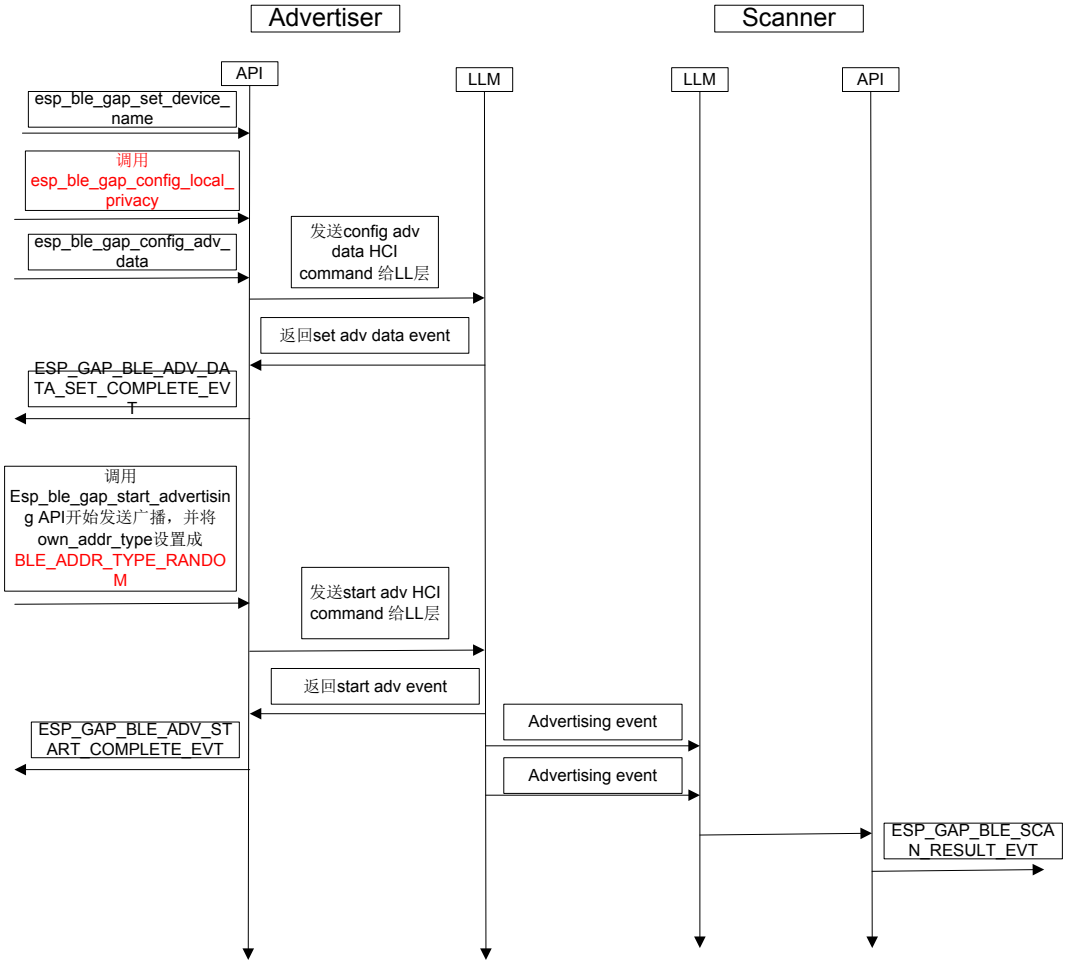

图 3-3. 广播流程图 – 使用可解析地址

> ⚠ **注意：**
>
> 当使用可解析地址进行广播时，需要等待 esp_ble_gap_config_local_privacy 事件返回后，才能开始广播，并且需要将广播参数里的 own_addr_type 类型设置为 BLE_ADDR_TYPE_RANDOM。

3.1.3.3. 使用静态随机地址进行广播

与使用可解析地址进行广播一样，使用静态随机地址进行广播也需要将 esp_ble_adv_params_t 成员 own_addr_type 设置为 BLE_ADDR_TYPE_RANDOM，广播流程图如下：

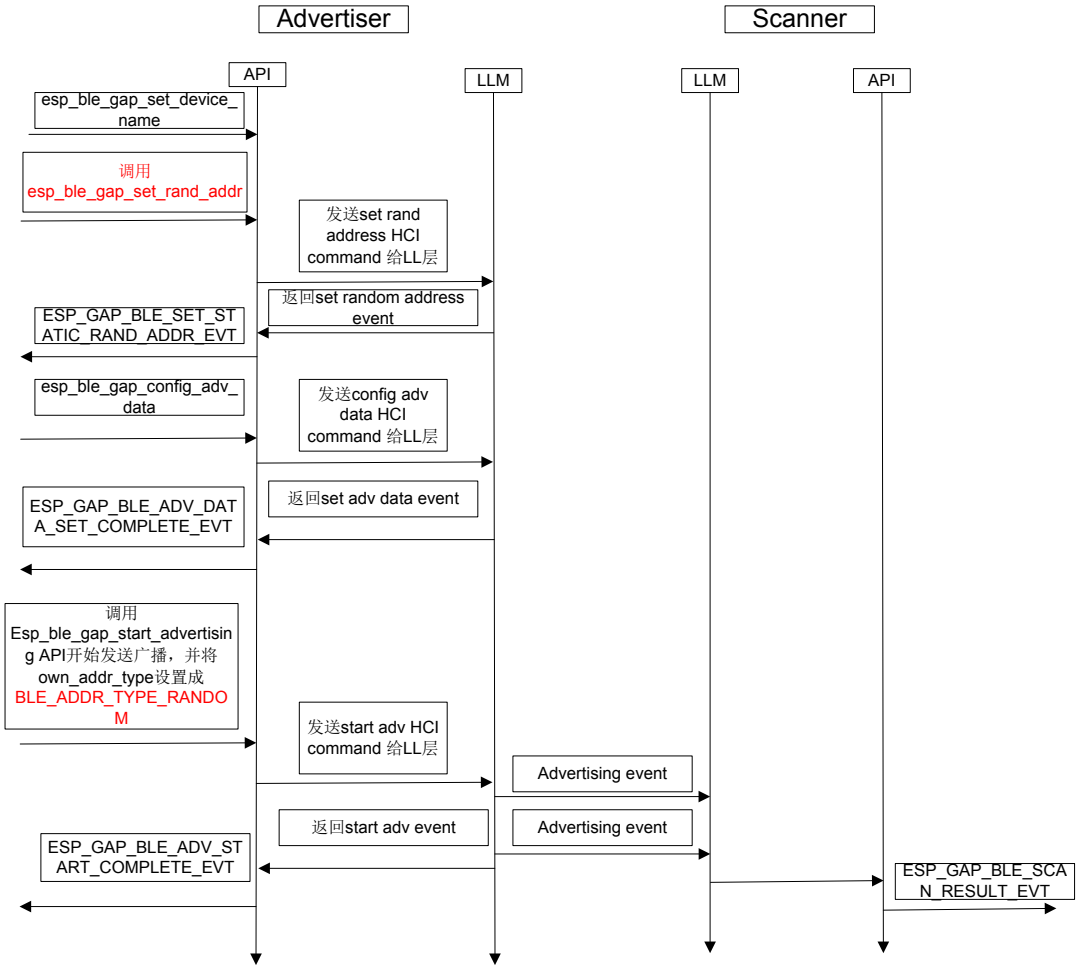

图 3-4. 广播流程图 – 使用静态随机地址

3.1.4. BLE 广播类型介绍

BLE 广播主要有 5 种类型，分别为：可连接可扫描非定向广播 (Connectable scannable undirected event type)、高占空比定向广播 (High duty cycle directed event type)、可扫描非定向广播 (Scannable undirected event type)、不可连接非定向广播 (Non-connectable undirected event type)、可连接低占空比定向广播 (Connectable low duty cycle directed event type)。

3.1.4.1. 可连接可扫描非定向广播

表 3-1.数据包结构

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

可连接可扫描非定向广播是指可被任何设备发现并可连接。可扫描是指当对端设备发送扫描请求 (Scan Request) 时，本端设备需要回复扫描应答 (Scan Response)。

如上表所示，可连接可扫描非定向广播数据包主要包括 6 个字节的广播地址和 0~31 个字节的广播数据包内容。其中，当广播地址类型设置为“静态随机地址”时，应用需要调用 `esp_ble_gap_set_rand_addr` 设置指定的地址进行广播；当广播地址类型设置为“public”或“可解析地址”时，则由协议栈自动生成。

3.1.4.2. 高占空比定向广播和可连接低占空比定向广播

表 3-2.数据包结构

Payload	
AdvA (6 octets)	InitA (6 octets)

定向广播只能被指定设备所发现和连接。

如上表所示，高占空比定向广播数据包主要包括 6 个字节的固定广播设备接收地址和 6 个字节的指定广播设备接收地址。在本广播类型下，广播参数中的 `adv_int_min` 和 `adv_int_max` 将被忽略。

在可连接低占空比定向广播类型下，广播参数中的 `adv_int_min` 和 `adv_int_max` 必须大于 100 ms (0xA0)。

> 📖 **说明：**
>
> 定向广播不携带广播数据（*Adv Data*）。

3.1.4.3. 可扫描非定向广播

可扫描非定向广播是指可被任何设备发现但是不能被连接。

表 3-3.数据包结构

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

如上表所示，与可连接可扫描非定向广播数据包一样，可扫描非定向广播数据包也包括 6 个字节的广播地址和 0~31 个字节的广播数据包内容，但是只能被设备扫描而不能被设备连接。

3.1.4.4. 不可连接非定向广播

不可连接非定向广播是指可被任何设备发现，但是既不可扫描也不可连接。不可扫描是指当对端设备发送扫描请求时不会回应扫描应答，不可连接是指不能被任何设备连接。

表 3-4.数据包结构

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

如上图所示，不可连接非定向广播数据包也包括 6 个字节的广播地址和 0~31 个字节的广播数据包内容，但是只能被设备发现不能被设备扫描也不能被连接。

3.1.5. BLE 广播过滤策略介绍

在 ESP32 的 BLE 中，通过设置 adv_filter_policy 枚举类型来实现广播过滤策略，此枚举类型中有以下 4 个值：

- `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`

这 4 个值分别对应 4 种情况，分别为：

- 可被任何设备扫描和连接（不使用白名单）
- 处理所有连接请求和只处理在白名单设备中的扫描请求
- 处理所有扫描请求和只处理在白名单中的连接请求
- 只处理在白名单中设备的连接请求和扫描请求

3.1.6. BLE 扫描流程

在 ESP32 中，扫描设备主要是通过调用 `esp_ble_gap_set_scan_params` 来设置扫描时的参数，然后调用 `esp_ble_gap_start_scanning` 开始扫描。扫描到的设备将会通过 `ESP_GAP_BLE_SCAN_RESULT_EVT` 事件返回，最后当 `duration` 超时时，会通过 `ESP_GAP_SEARCH_INQ_CMPL_EVT` 事件返回。

> ⚠ **注意：**
>
> 当 *duration* 值为 *0* 时，将会永久扫描而不产生超时。

3.1.7. BLE GAP 实现机制

ESP32 的 BLE 通用访问规范 (GAP) 采用调用 BLE gap API 相关的 API 和注册 BLE gap callback 并通过事件 (event) 返回来获取当前设备状态。

3.2. GATT

3.2.1. ATT 属性协议

BLE 里面的数据以属性 (Attribute) 方式存在，每条属性由四个元素组成：

- 属性句柄 (Attribute Handle)：正如我们可以使用内存地址查找内存中的内容一样，ATT 属性的句柄也可以协助我们找到相应的属性，例如第一个属性的句柄是 `0x0001`，第二个属性的句柄是 `0x0002`，以此类推，最大可以到 `0xFFFF`。
- 属性类型 (Attribute UUID)：每个数据有自己需要代表的意思，例如表示温度、发射功率、电池等等各种各样的信息。蓝牙组织 (Bluetooth SIG) 对常用的一些数据类型进行了归类，赋予不同的数据类型不同的标识码 (UUID)。例如 `0x2A09` 表示电池信息，`0x2A6E` 表示温度信息。UUID 可以是 16 比特的 (16-bit UUID)，也可以是 128 比特的 (128-bit UUID)。
- 属性值 (Attribute Value)：属性值是每个属性真正要承载的信息，其他 3 个元素都是为了让对方能够更好地获取属性值。有些属性的长度是固定的，例如电池属性 (Battery Level) 的长度只有 1 个字节，因为需要表示的数据仅有 0~100%，而 1 个字节足以表示 1～100 的范围；而有些属性的长度是可变的，例如基于 BLE 实现的透传模块。
- 属性许可 (Attribute Permissions)：每个属性对各自的属性值有相应的访问限制，比如有些属性是可读的、有些是可写的、有些是可读又可写的等等。拥有数据的一方可以通过属性许可，控制本地数据的可读写属性。

表 3-5. 属性结构表

属性句柄	属性类型	属性值	属性许可
0x0001	-	-	-
0x0002	-	-	-
.....	-	-	-
0xFFFE	-	-	-
0xFFFF	-	-	-

我们把存有数据（即属性）的设备叫做服务器 (Server)，而将获取别人设备数据的设备叫做客户端 (Client)。下面是服务器和客户端间的常用操作：

- **客户端给服务端发数据**，通过对服务器的数据进行写操作 (Write)，来完成数据发送工作。写操作分两种，一种是写入请求 (Write Request)，一种是写入命令 (Write Command)，两者的主要区别是前者需要对方回复响应 (Write Response)，而后者不需要对方回复响应。
- **服务端给客户端发数据**，主要通过服务端指示 (Indication) 或者通知 (Notification) 的形式，实现将服务端更新的数据发给客户端。与写操作类似，指示和通知的主要区别是前者需要对方设备在收到数据指示后，进行回复 (Confirmation)。
- **客户端也可以主动通过读操作读取服务端的数据**。

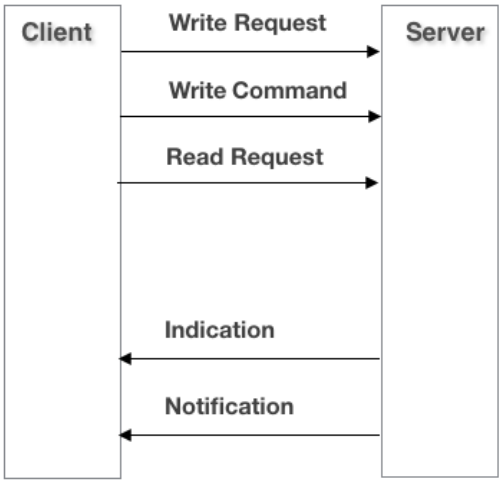

图 3-5. 客户端和服务器间的常见操作

⚠ **注意：**

关于更多的服务器和客户端之间的属性操作，参考蓝牙协议 *Core_V5.0*、*Vol3. Part F*、*Chapter 3.4 "Attribute Protocol PDUs"*。

服务器和客户端之间的交互操作都是通过上述的消息 ATT PDU 实现的。每个设备可以指定自己设备支持的最大 ATT 消息长度，我们称之为 MTU。ESP32 IDF 里面规定 MTU 可以设置的范围是 23~517 字节，对属性值的总长度没有做限制。

如果用户需要发送的数据包长度大于 (MTU-3)*，则需要调用准入写入请求 (Prepare Write Request) 来完成数据的写操作。同理，在读取一个数据时候，如果数据的长度超过 (MTU-1)，则需要通过大对象读取请求 (Read Blob Request) 来继续读取剩余的值。

> ⚠ ***注意：**
>
> 注意区分 *MTU* 和空中的单个物理数据包的最大长度 *(LE Packet Length)* 的区别。前者是针对 *Host ATT* 层，而后者针对的是物理层 *(PHY)*。简单解释，即 *MTU* 针对的是单个 *ATT Request* 包能否完全装下要发送的数据，是否需要使用类似 *Prepare Write Request* 的包来分包发送；而 *LE Packet Length* 是底层的物理层 *PHY* 在发包的时候，决定这个包是否需要分成多个物理包来发送。例如，如果 *(MTU+4)* 大于 *LE Packet Length*，则 *1* 个 *ATT* 包可能需要分成多个物理包发送；反之，如果 *(MTU+4)* 小于 *LE Packet Length*，则所有的 *ATT* 包都能通过 *1* 个物理包发送出去。这里 *MTU* 之所以需要先加 *4* 再去比较，是因为实际发送的时候，需要给包加上 *4* 个字节的 *L2CAP* 头信息。

3.2.2. GATT 规范

ATT 属性协议规定了在 BLE 中的最小数据存储单位，而 GATT 规范则定义了如何用特性值和描述符表示一个数据，如何把相似的数据聚合成服务 (Service)，以及如何发现对端设备拥有哪些服务和数据。

GATT 规范引进了特性值的概念。这是由于在某些时候，一个数据可能并不只是单纯的数值，还会带有一些额外的信息：

- 比如这个数据的单位是什么？是重量单位千克 kg、温度单位摄氏度 °C，还是其他单位；
- 比如希望具体告知对方这个数值的名称，例如同为温度属性 UUID 下，希望告知对方该数据表示“主卧温度”，另一个数据表示“客厅温度”；
- 比如在表示 230000、460000 等大数据时，可以增加指数信息，告知对方该数据的指数是 10^4，这样仅需在空中传递 23、46 即可。

上述内容仅为清楚描述一个数据众多需求中的几个例子，实际应用中还可能出现其他以各种方式表达的数据需求。为了包含这些信息，每个属性中均需要安排一大段数据空间，存储这些额外信息。然而，一个数据很有可能用不到绝大部分的额外信息，因此这种设计并这不符合 BLE “协议尽可能精简”的要求。在此背景下，GATT 规范引进了描述符的概念，每种描述符可以表达一种意思，用户可使用描述符，描述数据的额外信息。必需说明的是，每个数据和描述符并非一一对应，即一个复杂的数据可以拥有多个描述符，而一个简单的数据可以没有任何描述符。

数据本身的属性值及其可能携带的描述符，构成了特性 (Characteristic)* 的概念。数据特性包含以下几个部分：

- 特性声明 (Characteristic Declaration)：主要告诉对方此声明后面跟的内容为特性数值。从当前特性声明开始到下一个特性声明之间的所有句柄 (Handle) 将构成一个完整的特性。此外，特性声明还包括紧跟其后的特性数值的可写可读属性信息。
- 特性数值 (Characteristic Value)：特性的核心部分，一般紧跟在特性声明后面，承载特性的真正内容。
- 描述符 (Descriptor)：描述符可以对特性进行进一步描述，每个特性可以有多个描述符，也可以没有描述符。

BLE 协议中会把一些常用的功能定义成一个个的服务 (Service)*，例如把电池相关的特性和行为定义成电池服务 (Battery Service)；把心率测试相关的特性和行为定义成心跳服务 (Heart Rate Service)；把体重测试相关的特性和行为定义成体重服务 (Weight Scale Service)。

可以看到，每个服务包含若干个特性，每个特性包含若干个描述符。用户可以根据自己的应用需求选择需要的服务，并组成最终的产品应用。

一个完整服务的特性定义参考如下：

表 3-6. 服务定义示例

属性句柄	属性类型
0x0001	服务 1
0x0002	特性声明 1
0x0003	特性数值 1
0x0004	描述符 1
0x0005	特性声明 2
0x0006	特性数值 2
0x0007	描述符 2
0x0008	描述符 3
0x0009	服务 2
.....	.....

> ⚠ ***注意：**
>
> 关于不同的服务、特性值及描述符的定义，详见：
>
> - 蓝牙协议 *Core_V5.0*、*Vol3. Part G*、*Chapter 3 "Service Interoperability Requirements"*。
> - 蓝牙官网 https://www.bluetooth.com/zh-cn/specifications/gatt。

3.2.3. 基于 ESP32 IDF 建立 GATT 服务（GATT 服务器）

ESP32 IDF Release 1.0 实现了手动添加服务和特性的方法*，这种方式需要用户基于事件一条条地添加属性。所有的读写操作都会通过事件到应用层，由用户自己组包回复。这种方式对于不熟悉 BLE 协议的用户来说比较容易出错，尤其在需要添加大型 GATT 数据库的情况下，不推荐用户使用。

> ⚠ ***注意：**
>
> *ESP IDF* 仍保留了之前添加服务和特性的接口和 *Example*，用户可参考 *gatt_server* 例子程序。

在此背景之下，Release 2.0 基于先前版本，推出了通过属性表 (Attribute Table)* 添加服务和特性的功能。用户只需将要添加的服务和特性逐一填入一个表格，然后调用 `esp_ble_gatts_create_attr_tab` 函数，即可添加对应的服务和特性。此外，这种属性表添加方式还支持底层自动回复功能。这也就是说，底层可以自动回复一些请求，并判断一些错误，用户只需负责收发数据，而无需进行复杂的错误判断。

这种方式可以方便用户从其他平台移植规范到 ESP32 平台，整个过程无需重新实现全部的 BLE 规范。

> ⚠ ***注意：**
>
> 通过属性表添加服务和特性更加简单方便，不易出错，而且底层还可以对收到的数据包自动进行分析和处理，推荐客户使用，可参考 ***gatt_server_service_table*** 例子程序。

属性表的结构体规定了用户为了描述一个属性而需要初始化的元素，通过 `esp_gatts_attr_db_t` 进行定义，总结如下：

表 3-7. ESP32 IDF 结构体参数表

结构体参数	说明
`uint8_t attr_control`	定义对于类似 `write_response` 的相关回复是由底层进行自动回复，或传到应用层让用户手动回复（推荐使用 `ESP_GATT_AUTO_RSP` 自动回复模式）。
`uint16_t uuid_length`	表示 UUID 的长度，分 16-bit、32-bit、128-bit 三种。这是由于属性 UUID 是通过指针进行传递的，因此需要说明长度。

结构体参数	说明
uint8_t *uuid_p	表示当前属性的 UUID 的指针，用户根据上面的 UUID 长度，从指针里面读取指定长度的 UUID 值。
uint16_t perm	表示当前属性的读写许可。该变量为按位操作，每个比特表示一个特定的读写属性，对不同比特做或操作，可以表示多种读写属性。例如 PERM_READ | PERM_WRITE 表示这个属性既可读又可写。
uint16_t max_length	表示当前属性值的最大长度，协议栈主要根据这个变量为该属性分配内存。如果对方写入的属性值超过这里定义的最大长度，即回复写错误，错误原因是写长度超出数据最大长度。
uint16_t length	表示当前属性的当前实际长度。例如，该属性最大的长度是 512 字节，对方对其进行写操作，把值设置成了 2 个字节 “0x1122”，我们就设置当前实际长度为 2。当对方设备对这个属性进行读操作，我们可以从内存中获取该值的实际长度，仅将有实际内容的部分发送过去，而非将 512 字节全部发送给对方。
uint8_t *value	表示当前属性的属性值初始化值。由于这个参数为指针格式，因此需要从上面的 Length 获取该值的实际长度，从而从指针里获得正确的值。

3.2.4. 基于 ESP32 IDF 发现对方设备的服务信息（GATT 客户端）

GATT 客户端需要具有发现对方设备的服务和特性的功能 (Service Discovery)。不同的设备可能会使用不同的发现流程，下面以查找对方设备的 GATT 服务为例介绍一下 ESP32 IDF 使用的服务发现过程：

- 首先发现对方所有的 Service 信息，包括 Service 的 UUID 和 Handle 范围
  - GATT Service，UUID 0x1801，Handles 0x0001~0x0005
  - GAP Service，UUID 0x1800，Handles 0x0014~0x001C
- 然后在 GATT 的 Handle 范围内 (0x0001~0x0005)，继续查找所有的特性 (0x2803)
  - 找到特性 "Service Change Characteristic”，Handles 0x0002~0x0003
  - 其中 0x0002 对应的是这个特性的特性声明
  - 其中 0x0003 对应的是这个特性的特性值
  - 所以每个特性至少需要占据 2 个 Handle 的属性
- 既然 GATT Service 的 Handles 范围是 0x0001~0x0005，所以在 0x0003 后面可能跟有相应的描述符，因此继续从 0x0004 开始查找所有的描述符
  - 其中 0x0004 对应的是 “Client Characteristic Configuration” 描述符
  - 其中 0x0005 暂时没有任何信息，可能是为这个 Service 预留的 Handle

- 至此，GATT Service 的所有信息发现完毕。

3.3. SMP

本章节主要描述 ESP32 BLE 安全管理接口 API 的实现和使用流程。

3.3.1. 概述

ESP32 BLE 安全管理协议 (SMP) 相关 API 已封装在 BLE GAP 模块中，以供应用程序进行调用。

安全管理协议可以用于生成加密密钥和身份密钥，定义了一套简单的匹配和密钥分发协议，允许协议栈的其他层与其他设备进行安全链接并交换数据。这需要一个链路层链接，并对这个特定的链接有安全需求。BLE 安全管理器允许两个设备通过设置安全级别对该链路进行加密。此流程可以参考《蓝牙核心规范 4.2 版本》 (Bluetooth Core Specification version 4.2)中对安全管理章节的相关描述。为了清楚介绍 BLE 安全管理模块的实现过程，这里需要首先解释几个关键概念：

- 配对 (Pairing)：指两个设备同意双方建立一定级别的安全性连接。
- 绑定 (Bonding)：指至少一个设备向另一个设备发送某种识别或安全信息，供将来的连接使用。这些识别或安全信息可能是加密密钥 (LTK)、签名密钥 (CSRK) 或地址解析密钥 (IRK)。如果两个设备都可以绑定，则配对后会进行密钥分配，否则不会交换绑定信息。如果发送任何绑定信息，则违反协议。配对可能发生而不一定需要绑定，但在配对期间，两个设备需要通过交换特征确定对方是否支持绑定。如果两个设备都不支持绑定，则不应存储对方设备的相关安全信息。
- 认证 (Authentication)：不认证 (Unauthentication) 并不意味着链路不具备任何安全性，而是没有安全性和认证安全级别之间的中间级别。当用于链路加密的密钥具有需要双方设备确认的安全属性时，两个设备之间的关系称为认证。在使用短期密钥 (STK) 方法生成的功能下，这种安全属性在配对过程中被赋予了一个关键字。对于具有输入 / 输出和 OOB 方法的设备，之后产生和交换的所有密钥都具有认证 (MITM) 属性（使用 PIN / 较大的 OOB 密钥，这将强制执行安全性）。如果使用 Just Works 方法，所有密钥将具有未验证 (No MITM) 属性。
- 授权 (Authorization)：这是来自应用层的执行操作的许可。一些程序可能需要授权，在这种情况下，应用程被要求授权。如果没有给出授权，则该过程将失败。

3.3.2. BLE 安全管理控制器

3.3.2.1.BLE 加密流程

BLE 加密流程分为两部分：

- 当两个 BLE 设备之间未进行过绑定时，两个设备间将会通过配对步骤进行加密，并根据配对时的信息决定是否需要进行绑定；
- 当两个设备已经进行过绑定，一方设备想跟另一方进行加密时会直接走绑定流程。

Just work 模式下，Master 主动发起加密请求的流程图如下：

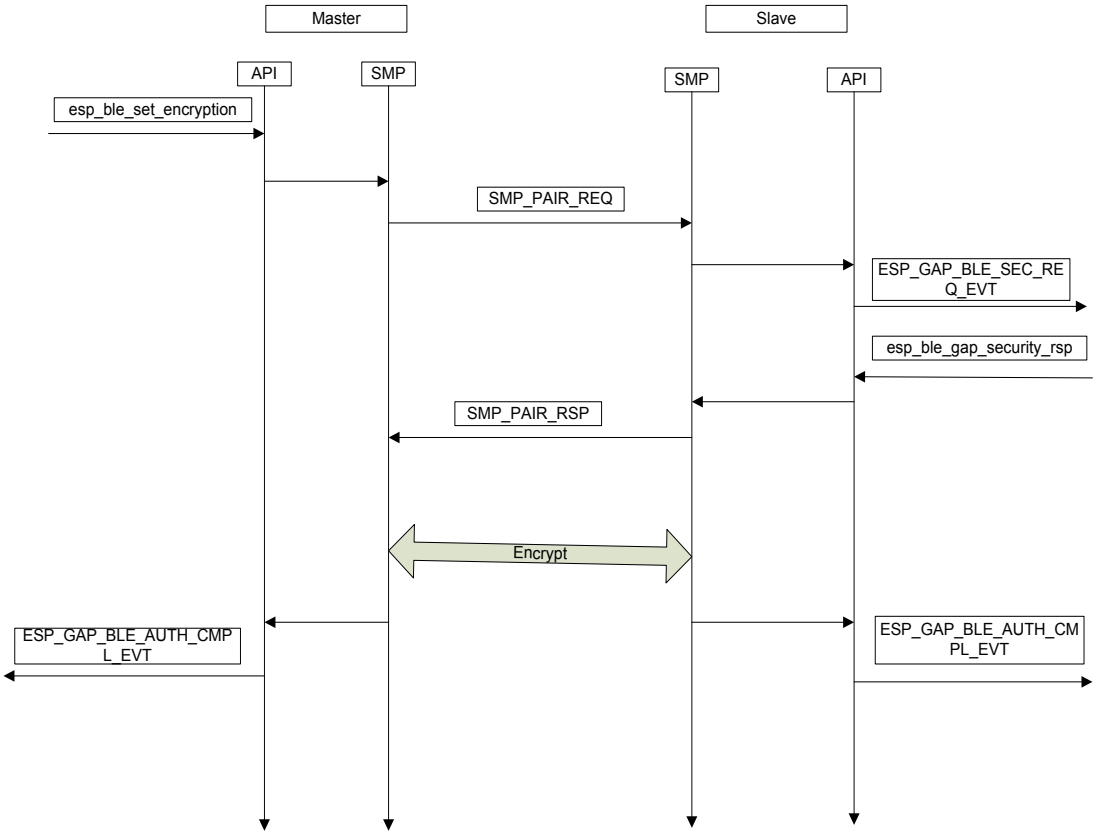

图 3-6. Just Work 加密流程图

Passkey notify 模式下，Master 主动发起加密请求的流程图如下：

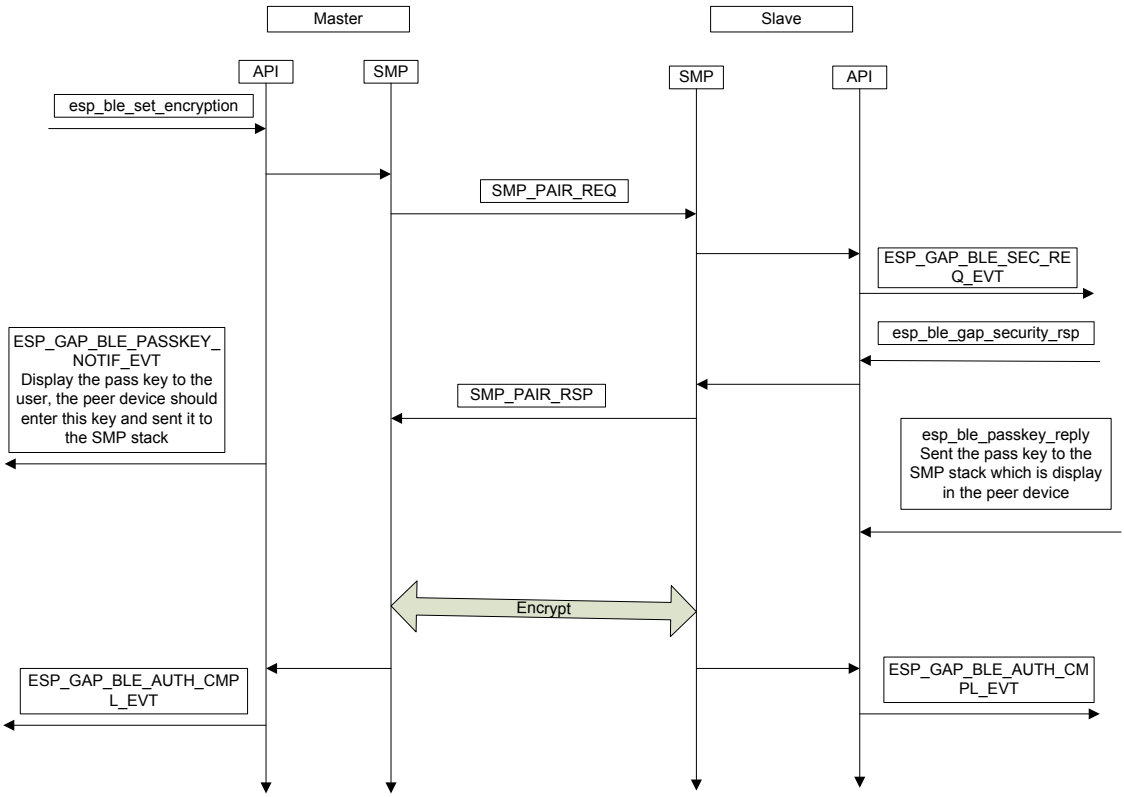

图 3-7. Passkey Notify 加密流程图

3.3.2.2.BLE 绑定流程

两个 BLE 设备间的绑定是通过调用 GAP 中的 API 实现的。根据蓝牙核心规范中的描述，绑定的目的是为了两个 BLE 设备在进行 SMP 加密后，再进行重连时能够使用相同的密钥对链路进行加密，从而能简化两个设备再连接时的加密流程。其中，两个 BLE 设备会在配对过程中交换加密密钥，并存储起来以便长期使用。绑定状态图如下：

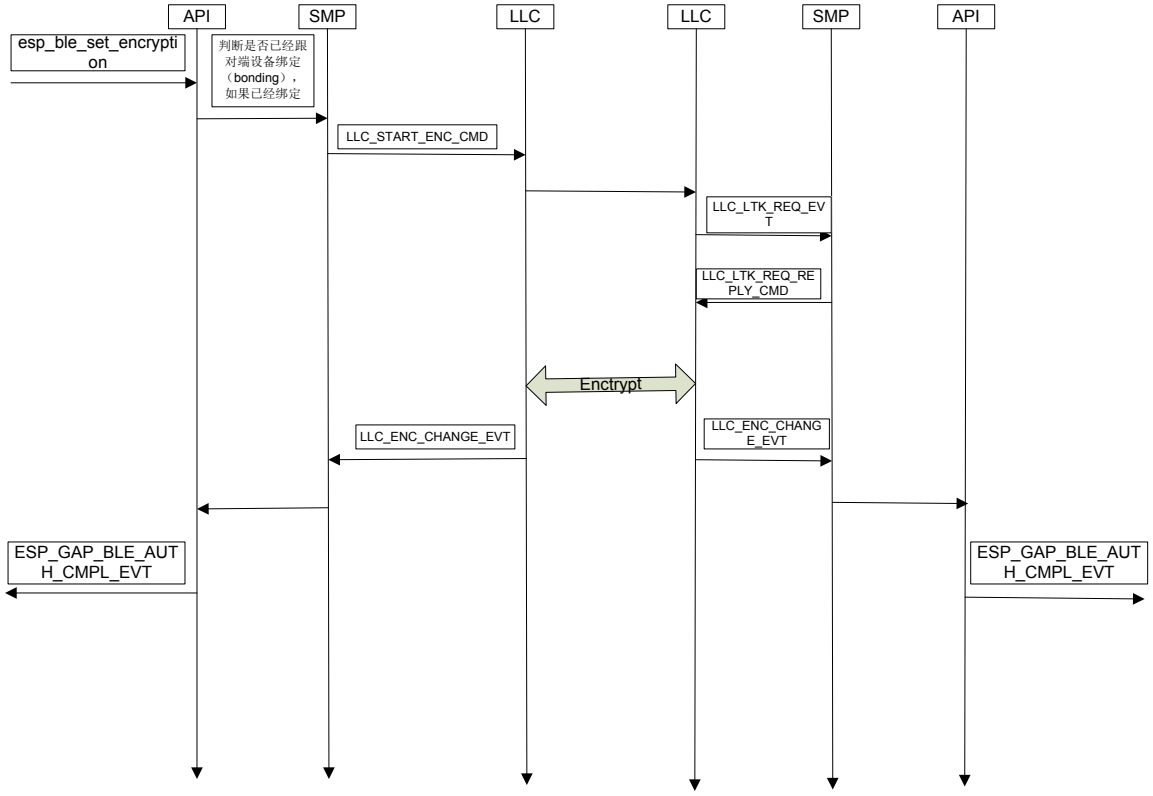

图 3-8. BLE 设备的绑定流程图

> ⚠ **注意：**
>
> 绑定必须由主设备 *(master)* 在连接时发起。

3.3.3. BLE 安全管理实现机制

BLE 安全管理是采用调用 BLE GAP 的 API 相关的加密函数，注册 BLE GAP callback，并通过事件返回获取当前的加密状态。

乐鑫 IoT 团队
www.espressif.com