Tina Linux 蓝牙
开发指南

版本号: 4.0
发布日期: 2022-03-21

版本历史

版本号	日期	制/修订人	内容描述
1.0	2019.03.03	AWA1423	创建
1.1	2020.05.12	AWA1423	1. 简述蓝牙协议 2. 增加 A2DP Source/HFP/GATT Server API 说明
1.2	2021.04.10	AWA1381	增加 R528 平台
4.0	2022.03.15	BT team	1. 更新 API 到 btmanager4.0 版本 2. 完善整个文档

目　录

插　图

1 概述

1.1 编写目的

介绍如何在 Allwinner 平台上 Bluetooth 开发。

1.2 适用范围

Allwinner 软件平台 Tina v3.5 版本以上，btmanager 版本在 4.0 版本以上；Allwinner 硬件平台支持 bluez 协议栈的蓝牙模组，包括但不限于 R11，R16，R18，R328，R329，R528，R818，R818B，MR813，MR813B，H133，V853...

1.3 相关人员

适用 Tina 平台的广大客户以及对蓝牙开发感兴趣的同事。

2 Bluetooth 简介

蓝牙技术发展至今已经迭代多个版本，截至目前 SIG Bluetooth 规范已经到 V5.2。蓝牙主要分为两种不同的技术: 经典蓝牙 (Classic Bluetooth，简称 BT) 和蓝牙低功耗（Bluetooth Low Energy，简称 BLE）。

蓝牙的工作频率范围是 2400MHz~2483.5MHz, 在经典蓝牙中，将其分为 79 个频道（每个频道 1MHz），而在蓝牙低功耗中，分为 40 个频道（每个频道 2MHz）。

蓝牙协议从结构上可以分为控制器 (Controller) 和主机 (Host) 两大部分，如下图：

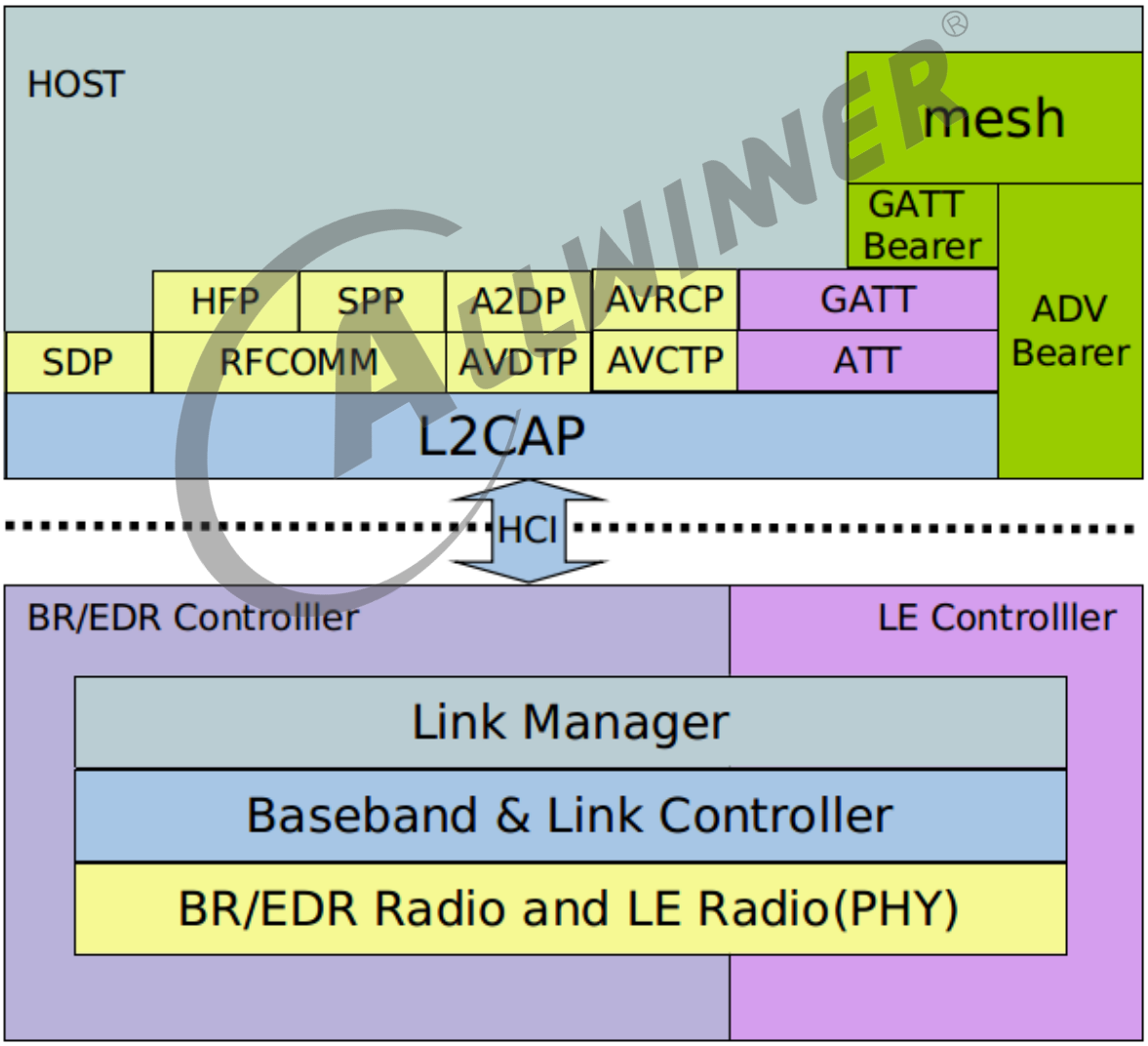

图 2-1: 协议结构图

Controller 和 Host 大部分情况下是运行在两个不同的芯片上，比如 Controller 运行在 XR829，而 HOST 运行在 R328，两个芯片通过硬件通信接口（如 UART，USB，SDIO）进行连接和通信，双方的通信协议称为 HCI 协议。

2.1 Controller 介绍

Controller 分为 BR/EDR Controller 和 LE controller，两者的在 PHY 层的信道划分是不一样的，两部分可以认为是独立的。

2.1.1 BR/EDR Controller

BR/EDR 采用跳频技术，数据传输时，并不是固定的占用 79 个信道中的某一个，而是一定规律的跳动，这个跟 wifi 的固定信道传输不同；在链路层可以下图几个状态：

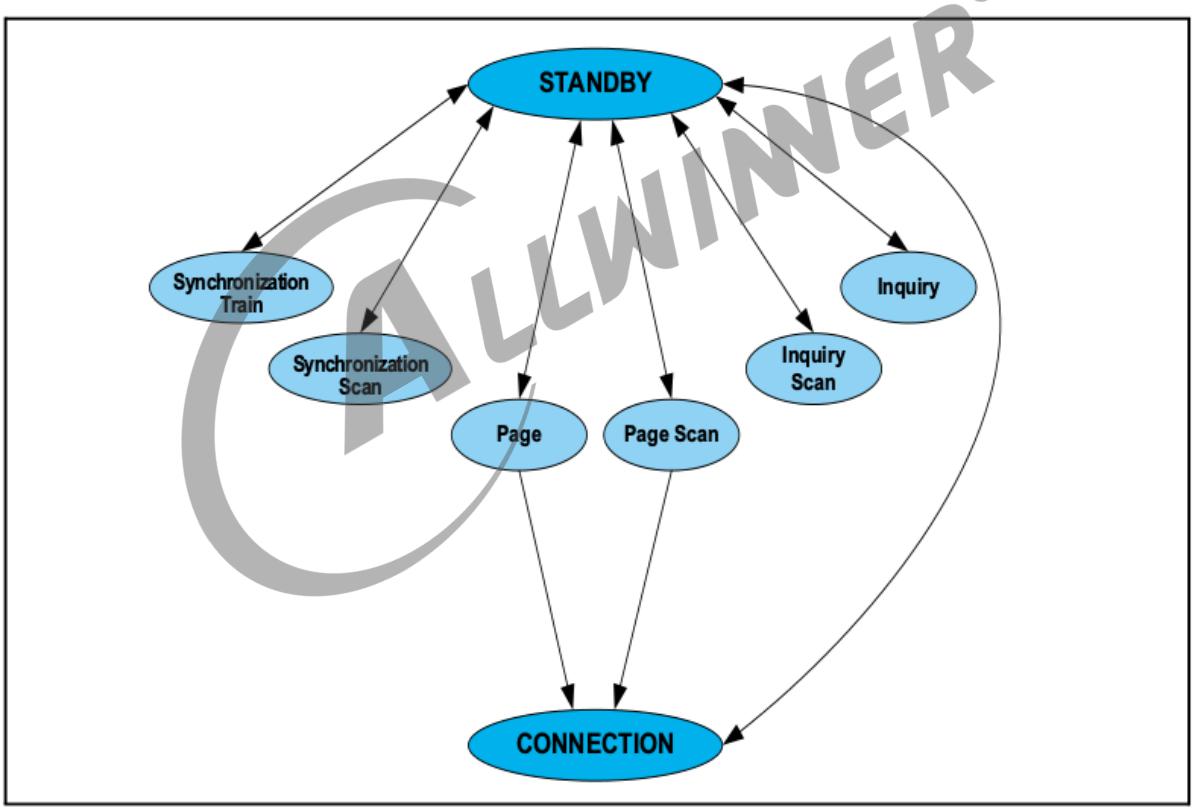

图 2-2: BR/EDR 链路状态

其中 synchronization train, synchronization scan 基本不用，

- STANDBY: 一个设备的默认状态, 可以认为是初始的状态。
- CONNECTION: 也就是处于连接的状态, 可以进行数据的交互。我们可以认为它是正常工作的状态。

- Page: 这个子状态就是我们通常称为的连接，进行连接/激活对应的 Slave 的操作我们就称为 Page。
- Page Scan: 这个子状态是和 Page 对应的，它就是等待被 Page 的 Slave 所处的状态，换句话说，若想被 Page 到，我们就要处于 Page Scan 的状态。
- Inquiry: 这就是我们通常所说的扫描状态，这个状态的设备就是去扫描周围的设备。
- Inquiry Scan: 这就是我们通常看到的可被发现的设备。体现在上层就是我们在 Android 系统中点击设备可被周围什么发现，那设备就处于这样的状态。

2.1.2 LE Controller

LE 在 40 个信道中又可以分为两种信道：连接信道和广播信道。

连接信道用于处于连接状态的蓝牙设备直接通信，与 BR/EDR 一样，都是采用跳频，只不过是在 37 个信道上跳频。

广播信道是用于设备之间进行无连接的广播通信，这些广播通信可以用于蓝牙设备的发现、连接等操作。LE 可以分为下图的几个状态。

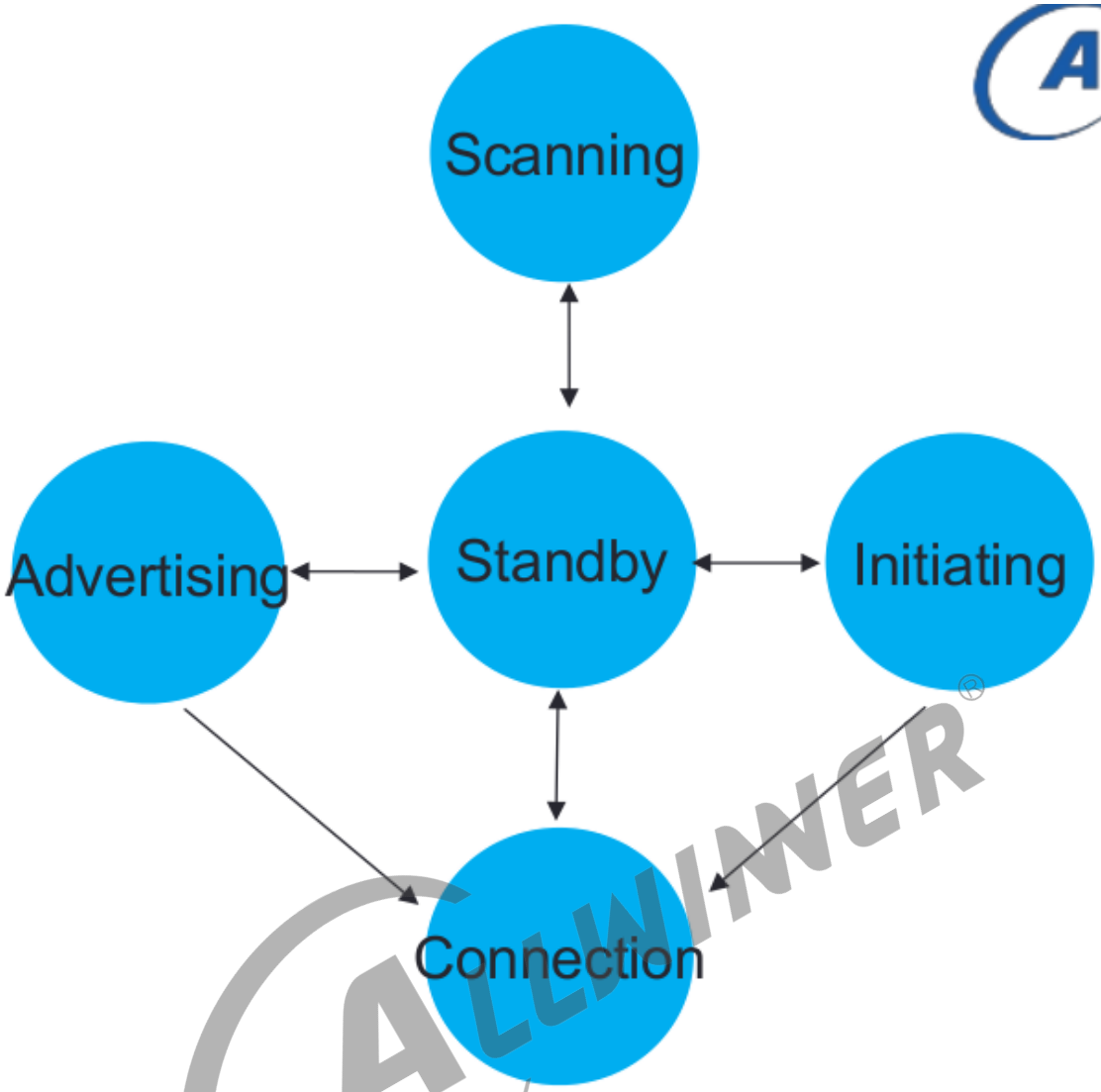

图 2-3: LE 链路状态

- Standby：链路层不收发报文。
- Advertising：链路层发送广播信道报文，并可能监听以及响应有这些广播信道报文触发的回应报文。
- Scanning：链路层监听广播者发送的广播信道报文。
- Initiating：链路层监听并响应从特定设备发起的广播信道报文。
- Connection：分为主从设备，有发起态进入连接态的设备为主设备，由广播态进入连接态的为从设备。

2.1.2.1 LE Device address

LE Device address 可以分为两种类型：Public device address 和 Random device address。

（1）Public device address

在通信系统中，设备地址是用来唯一识别一个物理设备的，如 TCP/IP 网络中的 MAC 地址、传统蓝牙中的蓝牙地址等。对设备地址而言，一个重要的特性，就是唯一性（或者说一定范围内的唯一），否则很有可能造成很多问题。蓝牙通信系统也不例外。对经典蓝牙（BR/EDR）来说，其设备地址是一个 48bits 的数字，称作 “48-bit universal LAN MAC addresses(和电脑的 MAC 地址一样)“。正常情况下，该地址需要向 IEEE 申请（其实是购买）。当然，这种地址分配方式，在 BLE 中也保留下来了，就是 Public Device Address。Public Device Address 由 24-bit 的 company_id 和 24-bit 的 company_assigned 组成，具体可参考蓝牙 Spec 中相关的说明（Core_v5.2.pdf：[Vol 2] Part B,Section 1.2）。

（2）Random device address

Random device address 又分为 Static Device Address 和 Private Device Address 两类。在 BLE 时代，只有 Public Device Address 还不够，主要 3 个原因：首先 Public Device Address 需要向 IEEE 购买。虽然不贵，但在 BLE 时代，相比 BLE IC 的成本，还是不小的一笔开销；其次：Public Device Address 的申请与管理是相当繁琐、复杂的一件事情，再加上 BLE 设备的数量众多（和传统蓝牙设备不是一个数量级的），导致维护成本增大；最后，安全因素。BLE 很大一部分的应用场景是广播通信，这意味着只要知道设备的地址，就可以获取所有的信息，这是不安全的。因此固定的设备地址，加大了信息泄漏的风险。为了解决上述问题，BLE 协议新增了一种地址：Random Device Address，即设备地址不是固定分配的，而是在设备设备启动后随机生成的。根据不同的目的，Random Device Address 分为 Static Device Address 和 Private Device Address 两类。

（a）Static Device Address

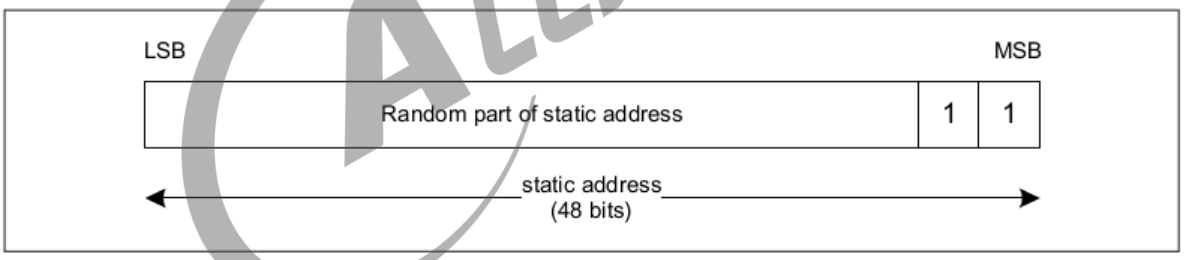

图 2-4: Format of static address

Static Device Address 是设备在上电时随机生成的地址，格式如上。46bits 的随机数，可以很好地解决 “设备地址唯一性” 的问题，因为两个地址相同的概率很小。地址随机生成，可以解决 Public Device Address 申请所带来的费用和维护问题。

特征可以总结为：

- 最高两个 bit 为 “11”。
- 剩余的 46bits 是一个随机数，不能全部为 0，也不能全部为 1。
- 在一个上电周期内保持不变。
- 下一次上电的时候可以改变。但不是强制的，因此也可以保持不变。如果改变，上次保存的连接等信息，将不再有效。

（b）Private Device Address

Static Device Address 通过地址随机生成的方式，解决了部分问题，Private Device Address 则更进一步，通过定时更新和地址加密两种方法，提高蓝牙地址的可靠性和安全性。根据地址是否加密，Private Device Address 又分为两类，Non-resolvable private address 和 Resolvable private address。下面我们分别描述。

Non-resolvable private address

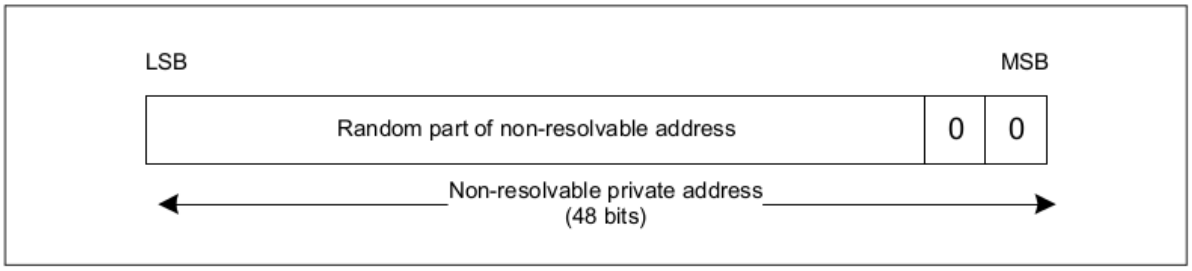

图 2-5: Format of non-resolvable private address

Non-resolvable private address 和 Static Device Address 类似。其格式如上，不同之处在于，Non-resolvable private address 会定时更新。更新的周期称是由 GAP 规定的，称作 T_GAP(private_addr_int) ，建议值是 15 分钟。特征可以总结为：

- 最高两个 bit 为 “00”。
- 剩余的 46bits 是一个随机数，不能全部为 0，也不能全部为 1。
- 以 T_GAP(private_addr_int) 为周期，定时更新。

Resolvable private address

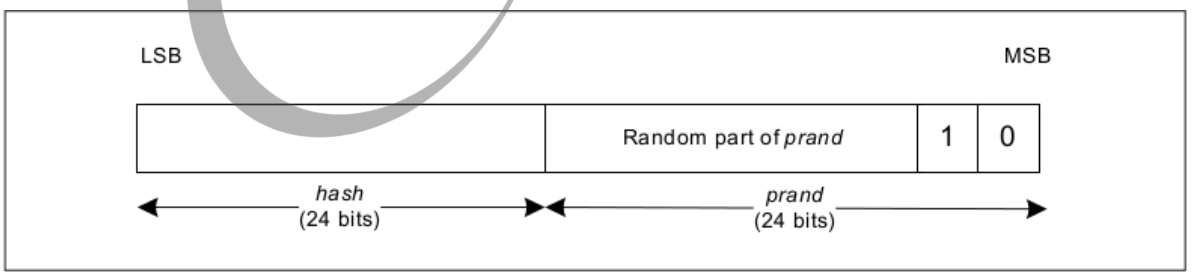

图 2-6: Format of resolvable private address

Resolvable private address 比较有用，格式如上，它通过一个随机数和一个称作 identity resolving key (IRK) 的密码生成，因此只能被拥有相同 IRK 的设备扫描到，可以防止被未知设备扫描和追踪。

- 由两部分组成：高位 24bits 是随机数部分，其中最高两个 bit 为 “10”，用于标识地址类型；低位 24bits 是随机数和 IRK 经过 hash 运算得到的 hash 值，运算的公式为 hash = ah(IRK, prand)。

- 当对端 BLE 设备扫描到该类型的蓝牙地址后，会使用保存在本机的 IRK，和该地址中的 prand，进行同样的 hash 运算，并将运算结果和地址中的 hash 字段比较，相同的时候，才进行后续的操作。这个过程称作 resolve（解析），这也是 Non-resolvable private address/Resolvable private address 命名的由来。
- 以 T_GAP(private_addr_int) 为周期，定时更新。哪怕在广播、扫描、已连接等过程中，也可能改变。
- Resolvable private address 不能单独使用，因此需要使用该类型的地址的话，设备要同时具备 Public Device Address 或者 Static Device Address 中的一种。

2.1.2.2 Physical channel

PHY Channel	RF Center Frequency	Channel Index	Physical Channel Type	
			Primary Advertising	All others
0	2402 MHz	37	●	
1	2404 MHz	0		●
2	2406 MHz	1		●
…	…	…	…	…
11	2424 MHz	10		●
12	2426 MHz	38	●	
13	2428 MHz	11		●
14	2430 MHz	12		●
…	…	…	…	…
38	2478 MHz	36		●
39	2480 MHz	39	●	

图 2-7: Mapping of PHY channel to physical channel index and channel type

BLE 的信道划分为 0~39，其中 channel 37,38,39 为广播信道，其它为数据信道。

2.1.2.3 LE 广播通信

从图 3 中我们知道 LE 链路有 5 个状态，其中 Advertising 和 Scanning 是 LE 非常重要的两个状态，它对蓝牙在未建立连接之前至关重要。

广播通信的数据格式如下：

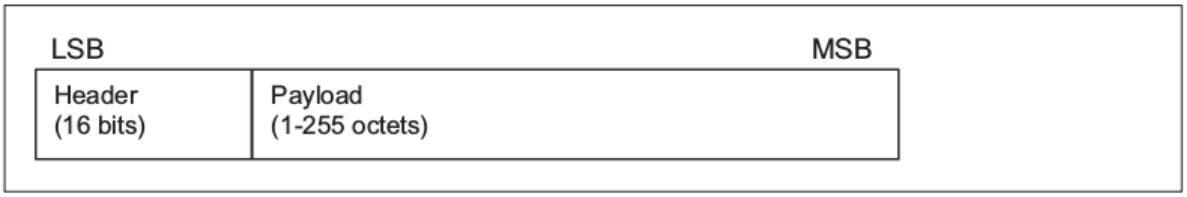

Figure 2.4: Advertising physical channel PDU

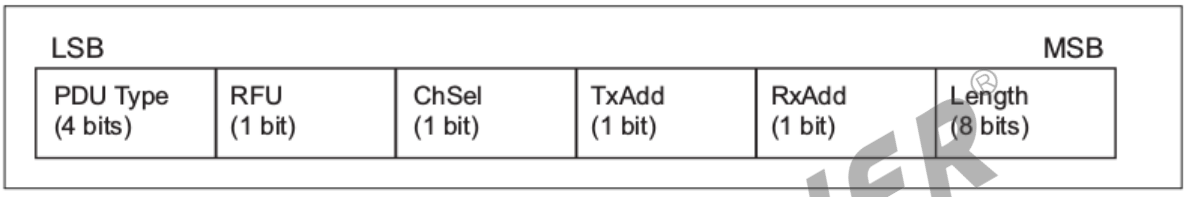

Figure 2.5: Advertising physical channel PDU header

图 2-8: Advertising physical channel PDU

根据 PDU 中的 Type 字段，我们可以分为以下几个类型：

状态	PDU类型	PDU格式	说明
Advertising	ADV_IND	AdvA(6 octets) AdvData(0~31 octets)	可被连接，可被扫描
	ADV_DRECT_IND	AdvA(6 octets) TargetA(6 octets)	可被指定的设备连接，不可被扫描
	ADV_NONCONN_IDN	AdvA(6 octets) AdvData(0~31 octets)	不可以被连接，不可以被扫描
	ADV_SCAN_IND	AdvA(6 octets) AdvData(0~31 octets)	不可以被连接，可以被扫描
Scanning	SCAN_REQ	ScanA(6 octets) AdvA(6 octets)	当接收到ADV_IND或者ADV_SCAN_IND类型的广播数据的时候，可以通过该PDU，请求广播者广播更多的信息：
	SCAN_RSP	AdvA(6 octets) ScanRspData(0~31 octets)	广播者收到SCAN_REQ请求后，通过该PDU响应，把更多的数据传送给接受者。

图 2-9: Advertising type

上图中，重点关注 AdvA，AdvData，ScanRspData。

- AdvA AdvA 字段包含广播者的地址，可以是 public address 也可以是 random address。

- AdvData 包含了广播者广播的数据内容，长度为 31 字节；
- ScanRspData 包含的是广播者收到 SCAN_REQ 之后回复的广播数据内容。

2.1.2.4 AdvData 和 ScanRspData 格式

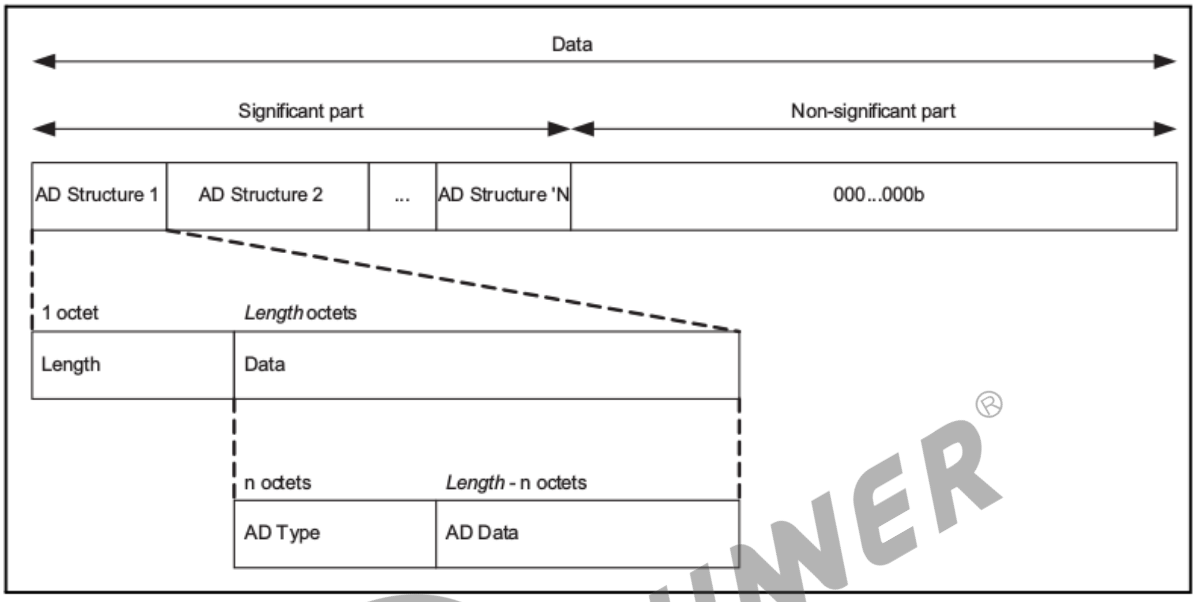

图 2-10: Advertising and Scan Response data format

如上图所示，AdvData 和 ScanRspData 格式内容由多个 AD Structure 组成，每个 AD structure 又细分为 length 和 Data，length 为 AD structure 的长度大小，Data 为数据内容。数据内容又为为 AD Type 和 AD data。AD Type 和 AD data 有详细内容可参考 https://www.Bluetooth.com/specifications/assigned-numbers/generic-access-profile/ 和 [Core Specification Supplement.pdf]。

Data type	Context				
	EIR	AD	SRD	ACAD	OOB
Service UUID	O	O	O	O	O
Local Name	C1	C1	C1	X	C1
Flags	C1	C1	X	X	C1
Manufacturer Specific Data	O	O	O	O	O
TX Power Level	O	O	O	X	O
Secure Simple Pairing OOB	X	X	X	X	O
Security Manager OOB	X	X	X	X	O
Security Manager TK Value	X	X	X	X	O
Slave Connection Interval Range	X	O	O	X	O
Service Solicitation	X	O	O	X	O
Service Data	X	O	O	O	O
Appearance	X	C2	C2	X	C1
Public Target Address	X	C2	C2	X	C1
Random Target Address	X	C2	C2	X	C1
Advertising Interval	X	C1	C1	X	C1
LE Bluetooth Device Address	X	X	X	X	C1
LE Role	X	X	X	X	C1
Uniform Resource Identifier	O	O	O	X	O
LE Supported Features	X	C1	C1	X	C1
Channel Map Update Indication	X	X	X	C1	X
BIGInfo	X	X	X	C1	X
Broadcast_Code	X	X	X	X	O

Table 1.1: Permitted usages for data types

O: Optional in this context (may appear more than once in a block).

C1: Optional in this context; shall not appear more than once in a block.

C2: Optional in this context; shall not appear more than once in a block and shall not appear in both the AD and SRD of the same extended advertising interval.

图 2-11: Permitted usages for data types

2.2 HOST 介绍

一般情况下蓝牙协议栈的 controller 运行在无线模组上，而 HOST 运行在主控芯片上，所以从用户的角度我们着重关注 HOST 端。在我们日常生活中，会碰到非常多的使用场景，比如蓝牙播放音乐，蓝牙鼠标，蓝牙传输文件，蓝牙语音通话，蓝牙 mesh 灯，通过蓝牙定位等等。根据这些不同的场景需求，SIG 定义了不同的规范（Profile）来支持这些场景下的需求。

根据不同的场景需求定义了不同用户规范（Profile），而 HOST 与 Controller 直接的传输是只有一个接口线，同时对于 controller 只需要关心数据的收发，不需要关心用户的实际场景，所以在 HOST 端有了 L2CAP 规范，这样就能屏蔽上层不同用户的协议，达到协议复用的功能，类似 TCP/IP 协议中的传输层。

L2CAP 之上有很多 profile，profile 之间有些是相辅相成的，有些则是完全独立的。根据这些 profile，我们大致可以将其分为 3 大类（参考图 2-1）。

- 经典蓝牙部分（黄色部分）。
- 蓝牙低功耗部分（紫色部分）。
- mesh 部分（绿色部分）。

下面我们对常用的 profile 进行简单的介绍，分为经典蓝牙和低功耗蓝牙两部分。

2.2.1 经典蓝牙 Prifile 介绍

2.2.1.1 GAP

GAP（Generic Access Profile）是一个基础的蓝牙 profile，用于提供蓝牙设备的通用访问功能，包括设备的发现、连接、鉴权、服务发现等等。

GAP 是所有其它应用模型的基础，它定义了在 Bluetooth 设备间建立基带链路的通用方法。还定义了一些通用的操作，这些操作可供引用 GAP 的应用模型以及实施多个应用模型的设备使用。GAP 确保了两个蓝牙设备（不管制造商和应用程序）可以通过 Bluetooth 技术交换信息，以发现彼此支持的应用程序。

2.2.1.2 A2DP

为了利用蓝牙异步无连接链路传输高质量的音频数据，蓝牙 SIG 发布了高级音频分发规范（Advanced Audio Distribution Profile, A2DP）。A2DP 典型的应用是音乐播放器将音频数据发送耳机或者音箱。当前 A2DP 仅仅定义了点对点的音频分发，没有定义广播式的音频分发。

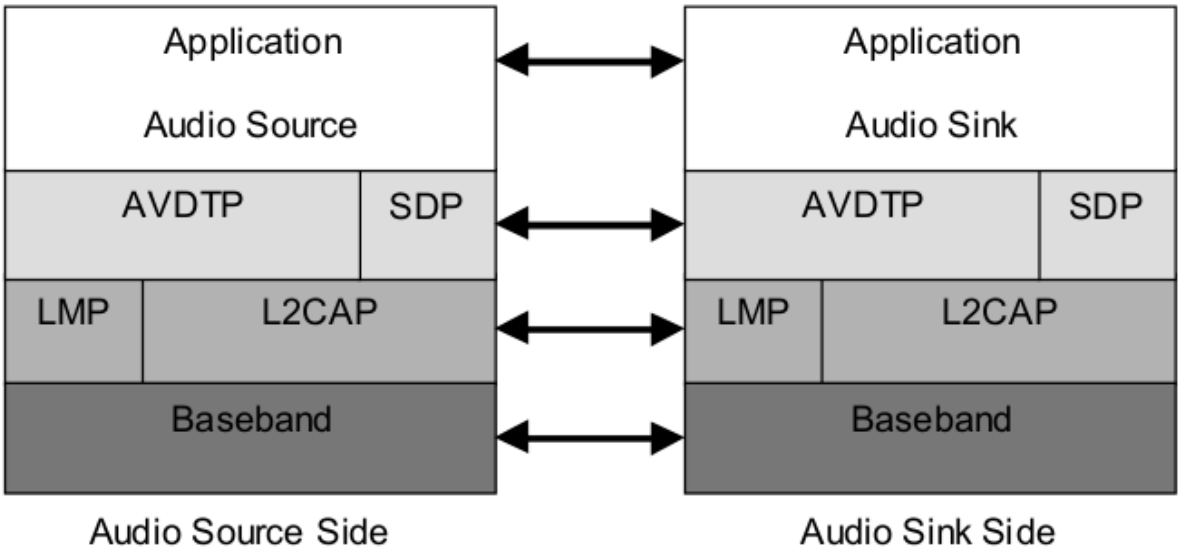

图 2-12: A2DP 传输结构

发送音频数据那一端我们称为 Souce 端（比如手机），接收音频的那一端我们称为 Sink 端（比如蓝牙音箱）。A2DP 是建立在 AVDTP 之上的，AVDTP 实现实体通过 L2CAP 分组进行 audio 数据流的传输和 audio 信令的交换，信令提供数据刘的发现、配置、建立和传输控制等功能，可以理解为 AVDTP 是 A2DP 更基础的协议。

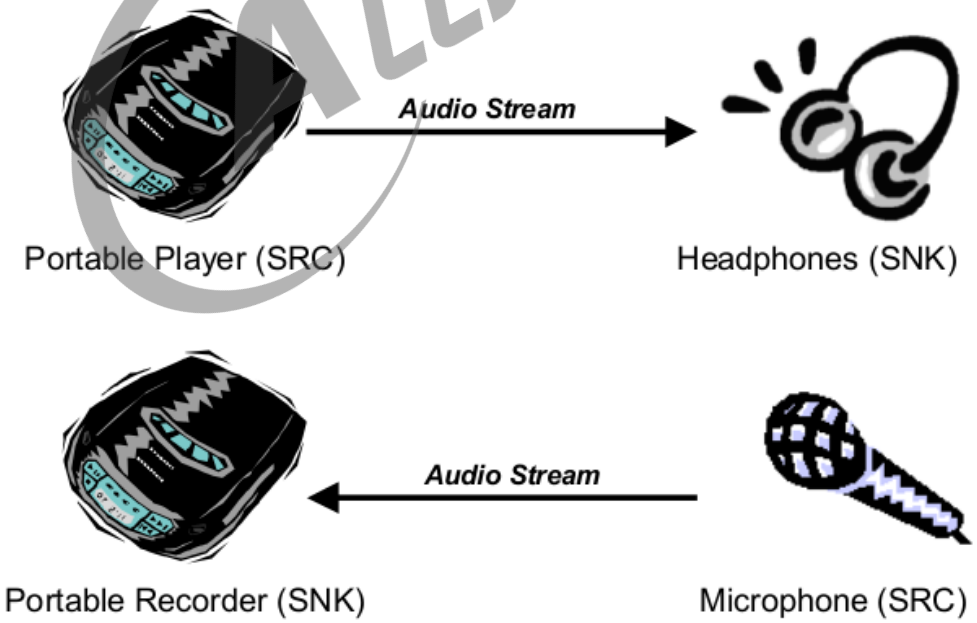

图 2-13: A2DP 传输例子

A2DP 可以分为 A2DP Source 和 A2DP Sink，音频发送端称为 A2DP Souce，数据接收端称

为 A2DP Sink。

2.2.1.3 AVRCP

AVRCP 是蓝牙音频实现蓝牙无线遥控功能的规范。

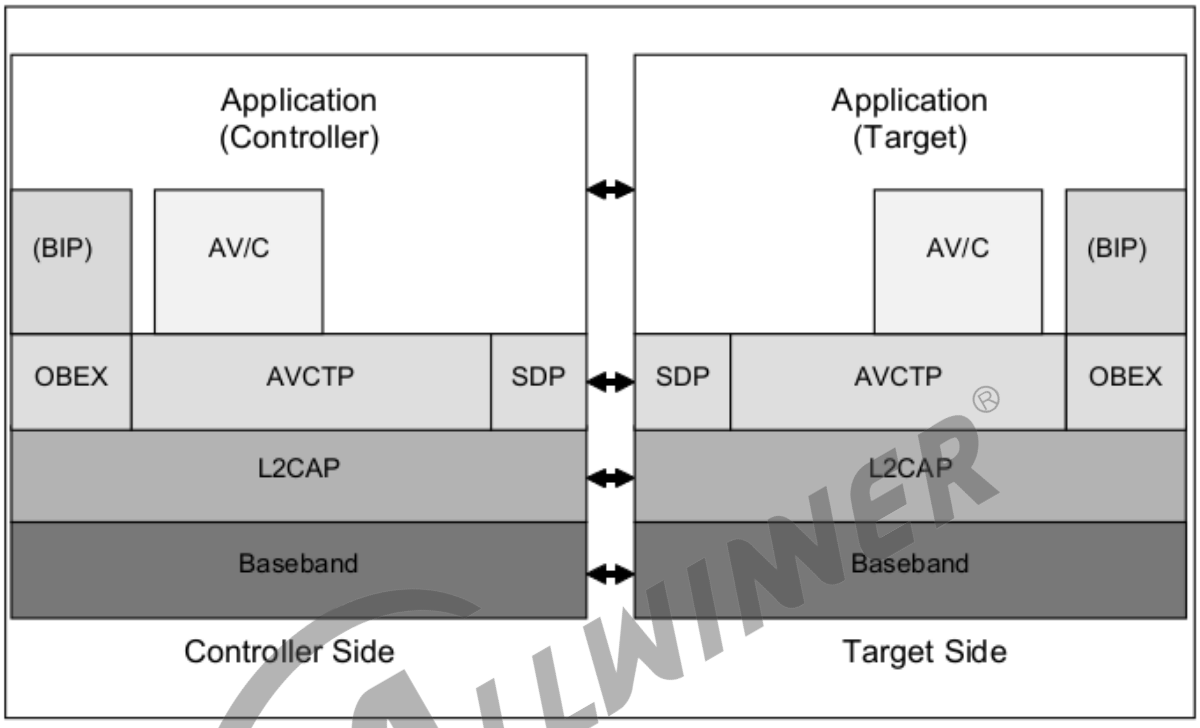

图 2-14: AVRCP 框架

AVRCP 中定义了两种设备角色：Controller（控制器，CT）、Target（目标机，TG）。CT 是发起命令传输给到 TG 的宿主，比如个人电脑、PDA、手机等。TG 是接收命令的宿主，比如蓝牙耳机、TV 等。

图 2-15: AVRCP 示例

AVRCP 中分为四种指令：

- Unit info: 用来获取 AV/C 设备的整体信息。
- Subunit info：用来获取 AV/C 设备的子设备信息。
- Vendor Dependent：厂商自定义的 AV/C 指令。
- Pass Through：音频设备使用最多的命令，如播放、暂停、快进、快退、下一曲、上一曲。

2.2.1.4 HFP

HFP 可以用做蓝牙语音通话，蓝牙语音通话实际上我们只需要重点关注两个方面：一个是语音通话的音频走哪里（over pcm 还是 over sco）。另外一个是蓝牙语音通话的指令，称为 AT 指令（比如电话的接听，挂断，拨号，获取手机信息等等）。

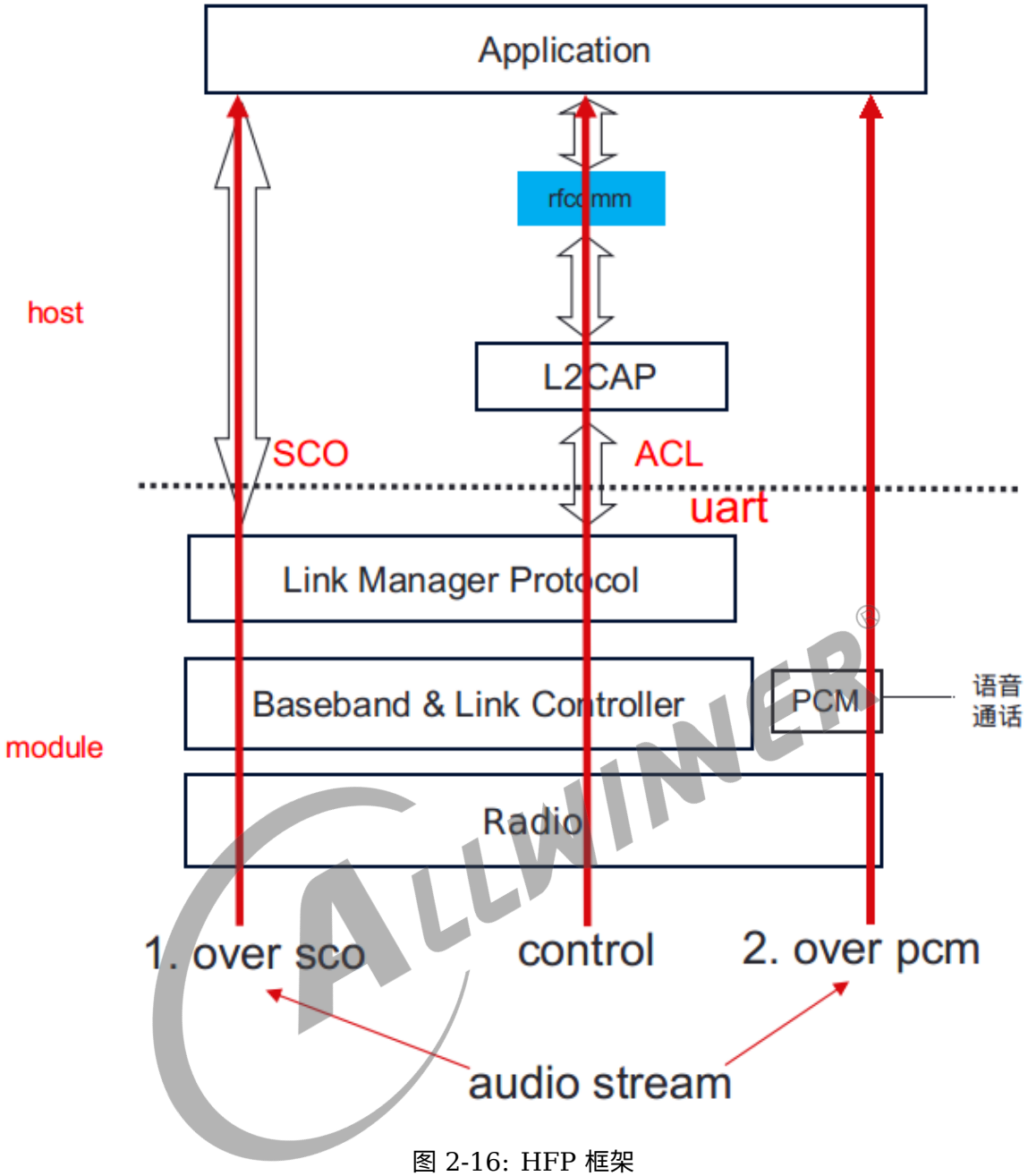

图 2-16: HFP 框架

（1）蓝牙语音通话音频数据

如上图所示，蓝牙语音通话的音频可以通过 HCI（SCO），也可以直接通过 PCM。

蓝牙语音通话数据走 HCI，数据流通过程是数据从模组端通过 uart 传输给主控，到 host 端后通过 SCO 链路传输给到上层应用。走 SCO 链路，蓝牙语音通话将与其他 profile 同时占用 hci，这样对多个 profile 同时存在时有极高要求。

蓝牙语音通话走 PCM，在模组端有单独的 pcm 接口，可以通过 I2S 与主控直接进行连接，蓝牙语音通话数据就就不需要再通过 HCI，占用带宽。

蓝牙链路层可以分为 ACL（面向无连接），SCO（面向连接）。大部分都是使用 ACL 链路，只

有蓝牙语音通话用 SCO 链路。而同时当前市面上的模组还支持蓝牙语音通话数据直接过 PCM（即也不经过 HCI SCO）。

当前我们的 btmanager 主要支持的方式是 hfp audio stream over pcm，没有走 SCO。

2.2.1.5 SPP

SPP(Serial Port Profile) 定义了使用蓝牙进行 RS232 （或类似）串行电缆仿真的设备应使用的协议和过程。简单来说就是在蓝牙设备之间建立虚拟的串口进行数据通信。SPP 底层基于 RFCOMM 协议，搭配 SDP 服务实现。

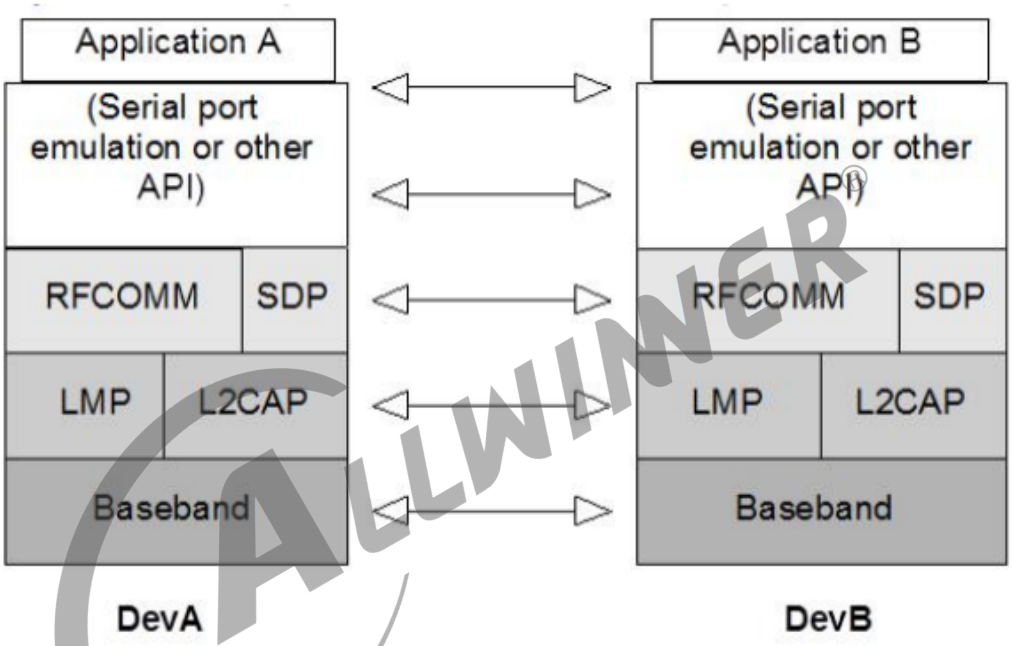

图 2-17: SPP 示例

2.2.2 低功耗蓝牙 Prfile 介绍

蓝牙低功耗对应的 profile 是 GATT（Generic Attribute profile），从第 2 章节图 1 中我们知道 GATT 规范是基于 ATT 协议（Attribute Protocol）实现的。ATT 的通信模型遵循 C/S 模型，包括 Server 与 Client。

2.2.2.1 Attribute

一台设备如果作为 gatt server 端，在 server 端可以有很多服务，比如心率服务，血压服务，电量服务等等，而服务的基本组成单元是 Attribute（属性）。

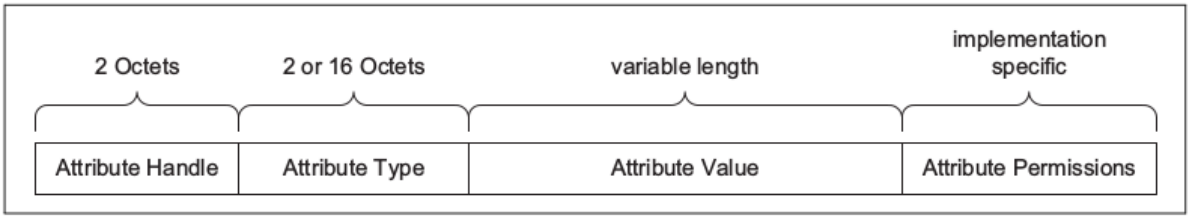

图 2-18: Attribute

属性（Attribute）是服务的基石，Attribute 的数据包类型如上图，包含了四种元素：

- Attribute Handle
- Attribute Type
- Attribute Value
- Attribute Permissions

2.2.2.1.1 Attribute Type Attribute Type 由 UUID 唯一标识，SIG 蓝牙联盟规定一些 UUID 代表特定的类型，比如 0x180D 代表 Heart Rate，0x1810 代表 Blood Pressure 等等（可参考：https://www.Bluetooth.com/specifications/gatt/services/）。

128 位的 UUID 相当长，设备间为了识别数据类型需要发送长达 16 字节的数据，为了提高效率，SIG 定义了“蓝牙 UUID 基数”的 128 位通用唯一标示码，结合一个较短的 16 位数使用，因此在实际传输的时候是 16 位的 uuid，在收发后补上蓝牙 UUID 基数即可。

蓝牙 UUID 基数如下：

```
00000000-0000-1000-8000-00805F9B34FB
```

需要发送的 16 位识别码为 0x2A01，完整的 128 位 UUID 便是：

```
00002A01-0000-1000-8000-00805F9B34FB
```

UUID 可以分为以下几组：

- 0x1800 ~ 0x26FF 用作服务类型通用唯一识别码
- 0x2700～0x27FF 用作标示计量单位
- 0x2800 ~ 0x28FF 用于区分属性类型
- 0x2900～0x29FF 用作特性描述
- 0x2A00～0X7FFF 用于区分特性类型

2.2.2.1.2 Attribute Handle 设备中有许多服务，而服务有许多属性组成，比如温度传感器服务包含温度属性、设备名称属性、电池电量属性等等，这些属性似乎可以通过 Attribute Type 来作于区分，但是如果温度属性有分为室内温度属性和室外温度属性，这样就没法通过 Attribute Type 来进行区分了，为了解决这个问题引入了 Attribute Handle，属性句柄。有效的属性句柄取值范围 0x0001~0xFFFF。

2.2.2.1.3 Attribute Value Attribute Value 是实际属性的值，比如温度传感器服务中温度属性温度是多少度。

2.2.2.1.4 Attribute Permissions Attribute 具有一组与之关联的权限值。权限值指定了关联属性是否具备读写、安全权限。一般有以下几种类型：

- Readable
- Writeable
- Readable and writable
- Encryption required
- No encryption required
- Authentication Required
- No Authentication Required

以上主要是关于属性四种元素的介绍，总结下 GATT profile 常见的属性定义，如下：

Attribute Type	UUID	Description
«Primary Service»	0x2800	Primary Service Declaration
«Secondary Service»	0x2801	Secondary Service Declaration
«Include»	0x2802	Include Declaration
«Characteristic»	0x2803	Characteristic Declaration
«Characteristic Extended Properties»	0x2900	Characteristic Extended Properties
«Characteristic User Description»	0x2901	Characteristic User Description Descriptor
«Client Characteristic Configuration»	0x2902	Client Characteristic Configuration Descriptor
«Server Characteristic Configuration»	0x2903	Server Characteristic Configuration Descriptor
«Characteristic Presentation Format»	0x2904	Characteristic Presentation Format Descriptor
«Characteristic Aggregate Format»	0x2905	Characteristic Aggregate Format Descriptor

图 2-19: GATT Profile attribute types

2.2.2.2 GATT

GATT 是基于 ATT 协议规范，所以 GATT 遵循 C/S 通信模型，包括 GATT server 和 GATT client。双方数据的传输方式分为以下 4 类：

- Client Request read
- Client Request write
- Server Notify
- Server Indication

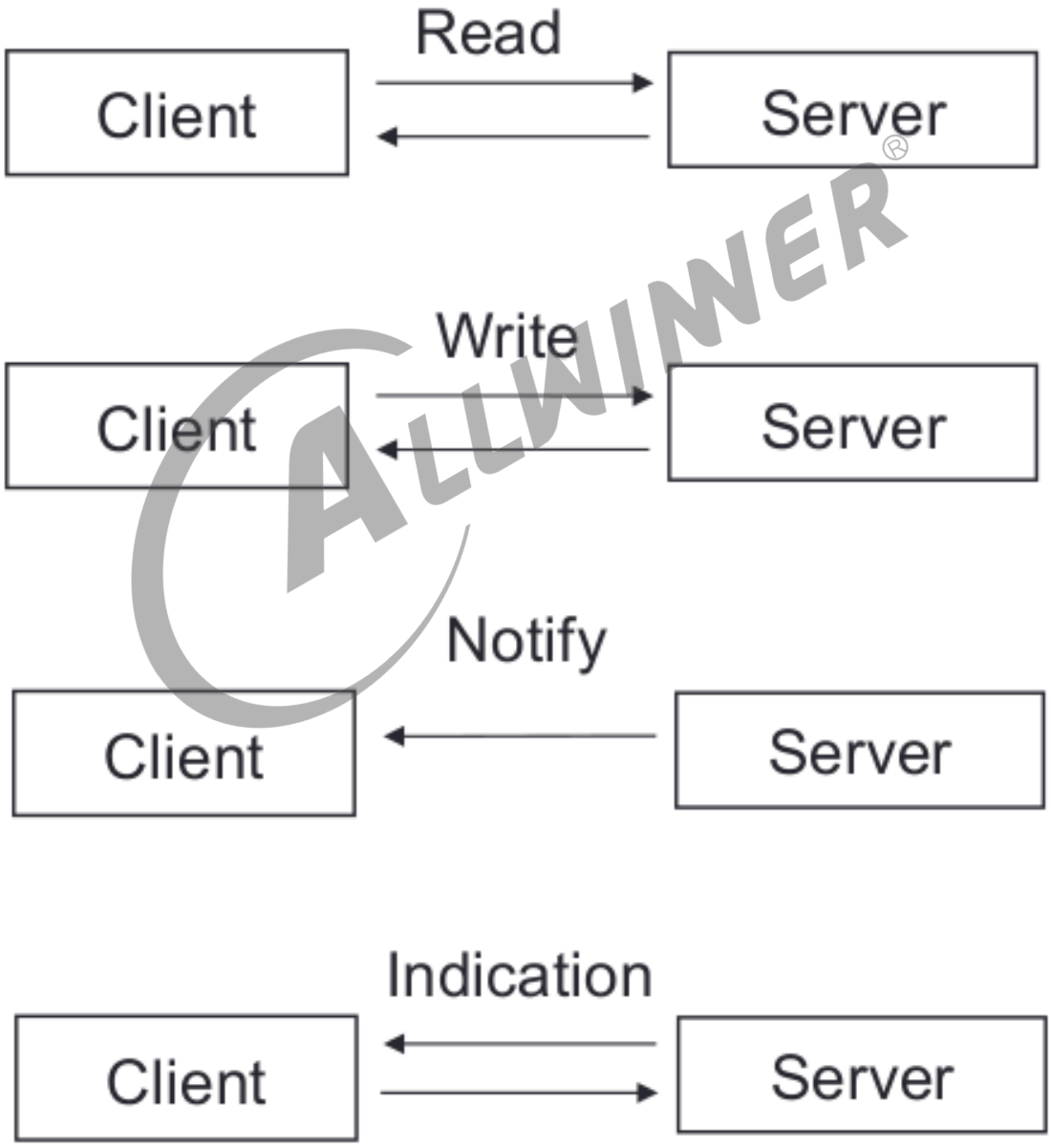

图 2-20: gatt 通信模型

其中 Server Notify 和 Server Indication 的区别：

- Notify：server 发送数据给 client 端不需要 client 回复；
- Indication ： server 发送数据给 client 端需要 client 回复；

2.2.2.3 GATT Server

前面说了，一个设备中可能有很多个服务，而服务一般具备一定的格式，服务的内容由属性（Attribute）构成。一个设备的服务结构组成如下图：

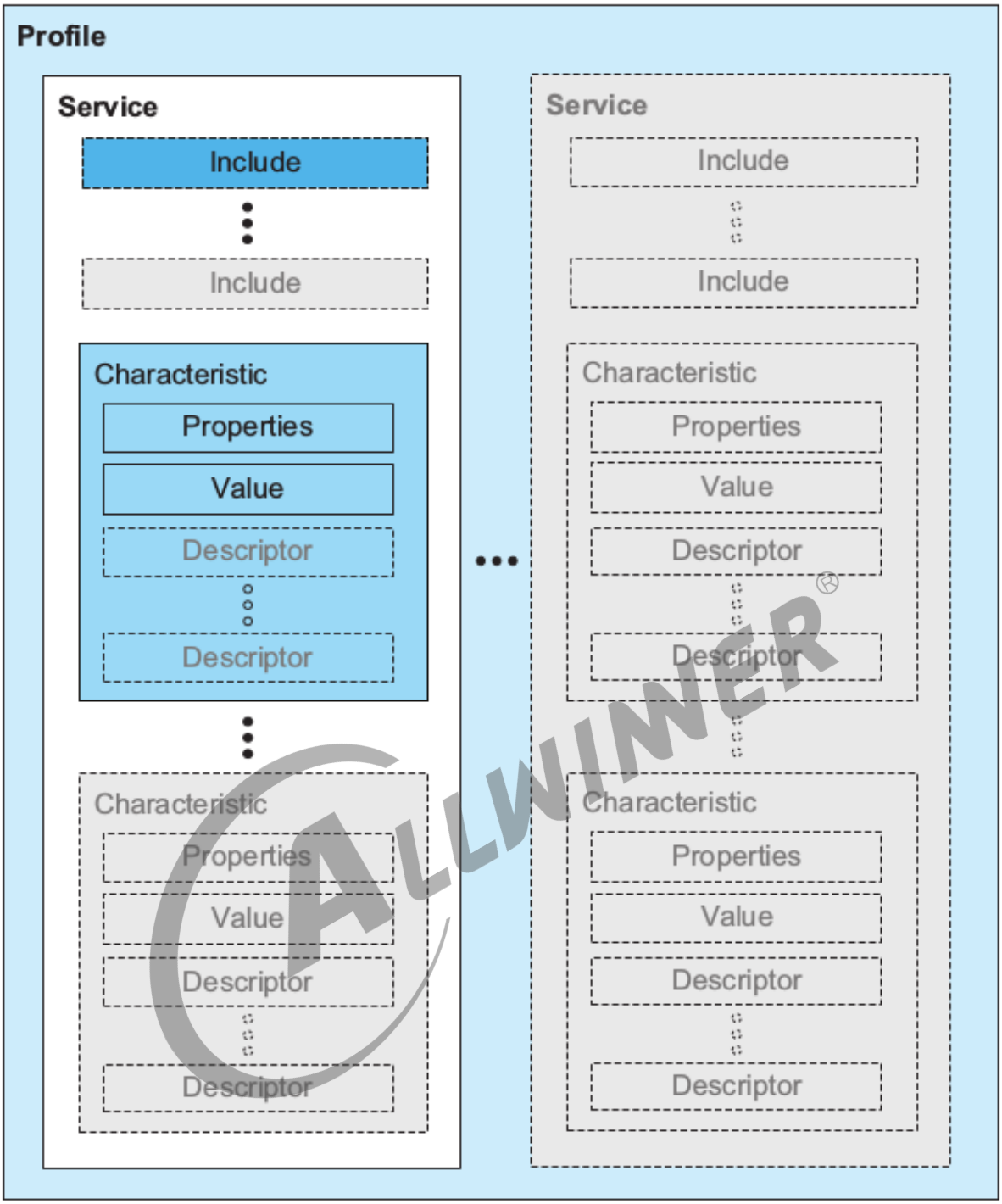

图 2-21: gatt server 模型

GATT profile 的层次结构依次为 Profile->Service->Characteristic，“profile” 是基于 GATT 所派生的真正 profile, 位于 GATT profile hierarchy 最顶层, 有一个或者多个和某一应用的场景有关的 service 组成。

GATT server 是一系列数据和相关行为组成的集合，为了完成某个功能或特性。一个 service 包含一个或者多个 Characteristic, 也可以通过 include 的方式, 包含其他 service. 所有一个 server 的属性类型可以分为以下几类：

- Primary Service
- Secondary Service
- Include
- Characteristic

大部分情况下，我们可能只会用到 Primary Service 和 Characteristic。Primary service 是用于区分不同的 service，比如上图中有两个 service。一个 service 开头的 uuid 一般固定为 0x2800，其 value 值将用于表征这是那一类 service，同时将结束于下一个 0x2800。

Characteristic 则是 GATT profile 最基本的数据单位, 由一个 properties, 一个 value, 一个或者个 Description 组成。

- Characteristic Properties 定义了 Characteristic 的 value 如何被使用, 以及 Characteristic 的 descriptor 如何被访问。
- Characteristic value 是特征的实际值, 例如一个温度特征, 就是温度值。
- Characteristic descriptor 则保存了一些和 Characteristic value 相关的信息。比如温度的单位是什么表征的。

注意：server 中的每一个定义,service,Characteristic,Characteristic Properties, Characteristic value,Characteristic descriptor 等等, 都是通过 Attribute 来进行表征的。

下图是实际一个 service 的例子：

Handle	Attribute Type	Attribute Value
0x0001	«Primary Service»	«GAP Service»
0x0004	«Characteristic»	{0x02, 0x0006, «Device Name»}
0x0006	«Device Name»	“Example Device”
0x0010	«Primary Service»	«GATT Service»
0x0011	«Characteristic»	{0x26, 0x0012, «Service Changed»}
0x0012	«Service Changed»	0x0000, 0x0000
0x0100	«Primary Service»	«Battery State Service»
0x0106	«Characteristic»	{0x02, 0x0110, «Battery State»}
0x0110	«Battery State»	0x04
0x0200	«Primary Service»	«Thermometer Humidity Service»
0x0201	«Include»	{0x0500, 0x0504, «Manufacturer Service»}
0x0202	«Include»	{0x0550,0x0568}
0x0203	«Characteristic»	{0x02, 0x0204, «Temperature»}
0x0204	«Temperature»	0x028A
0x0205	«Characteristic Presentation Format»	{0x0E, 0xFE, «degrees Celsius», 0x01, «Outside»}
0x0206	«Characteristic User Description»	“Outside Temperature”
0x0210	«Characteristic»	{0x02, 0x0212, «Relative Humidity»}
0x0212	«Relative Humidity»	0x27
0x0213	«Characteristic Presentation Format»	{0x04, 0x00, «Percent», «Bluetooth SIG», «Outside»}
0x0214	«Characteristic User Description»	“Outside Relative Humidity”
0x0280	«Primary Service»	«Weight Service»
0x0281	«Include»	0x0505, 0x0509, «Manufacturer Service»}
0x0282	«Characteristic»	{0x02, 0x0283, «Weight kg»}

Table A.1: Examples of attribute server attributes

图 2-22: weight service

3 Tina 蓝牙协议栈介绍

tina 系统当前使用的是开源的 bluez 协议栈，目前已经完全适配 RTL8723DS，XR829 等模组，如用户需要再使用其他模组，可以重新适配模组相关的硬件驱动即可，如 bt hci uart 驱动。当前有些模组厂商提供自己私有的协议栈另说。bluez 协议栈的软件结构图如下：

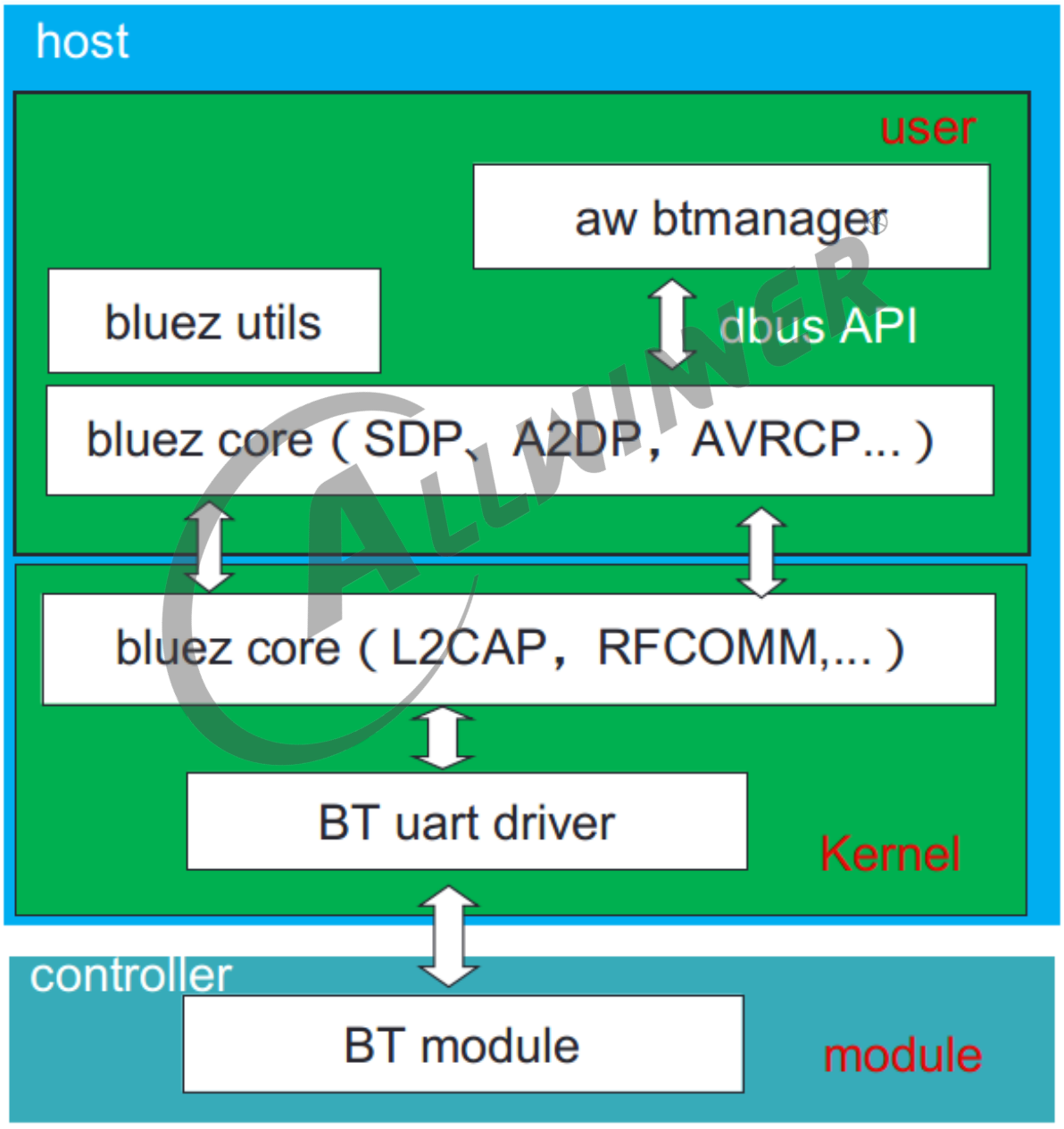

图 3-1：tina 蓝牙协议栈结构图

如上图所示，蓝牙规范的 controller 主要是在模组端实现，host 端主要是在主控端实现，模组与

主控通过 uart 进行连接通信。主控芯片（如 R328）主要实现包括 bt uart 驱动，L2CAP，以及 L2CAP 之上的各种 profile，其中 bt uart 驱动，L2CAP，rfcomm 等基本核心协议主要是在内核空间实现，其他主要在用户空间。由于开源的 bluez 协议栈主要是实现了基本的 profile，缺少一些必要组件，用户进行开发可能还需再次进行二次开发如对应蓝牙音乐，bluez 仅仅实现了 profile 部分，没有实现音频播放、解码部分，由此 allwinner 为了客户方便，开发完整功能，集成了 btmanager，并提供 c 语言的 API。

3.1 运行 tina 蓝牙协议栈

tina 蓝牙协议栈运行起来，主要是以下 4 个步骤。

- 蓝牙上电
- 下载 firmware
- 启动 bluez 协议栈
- 启动 btmanager

蓝牙协议栈应用的运行，我们这里是有一个对应的脚本 bt_init.sh，对应 tina 的文件路径如下：

```
tina3.5之前:
target/allwinner/方案/base-files/etc/Bluetooth/bt_init.sh
tina3.5及之后:
tina/package/allwinner/wireless/btmanager4.0/config/bt_init.sh
```

bt_init.sh 的内容解析详见**配置文件介绍**章节。

3.1.1 蓝牙上电

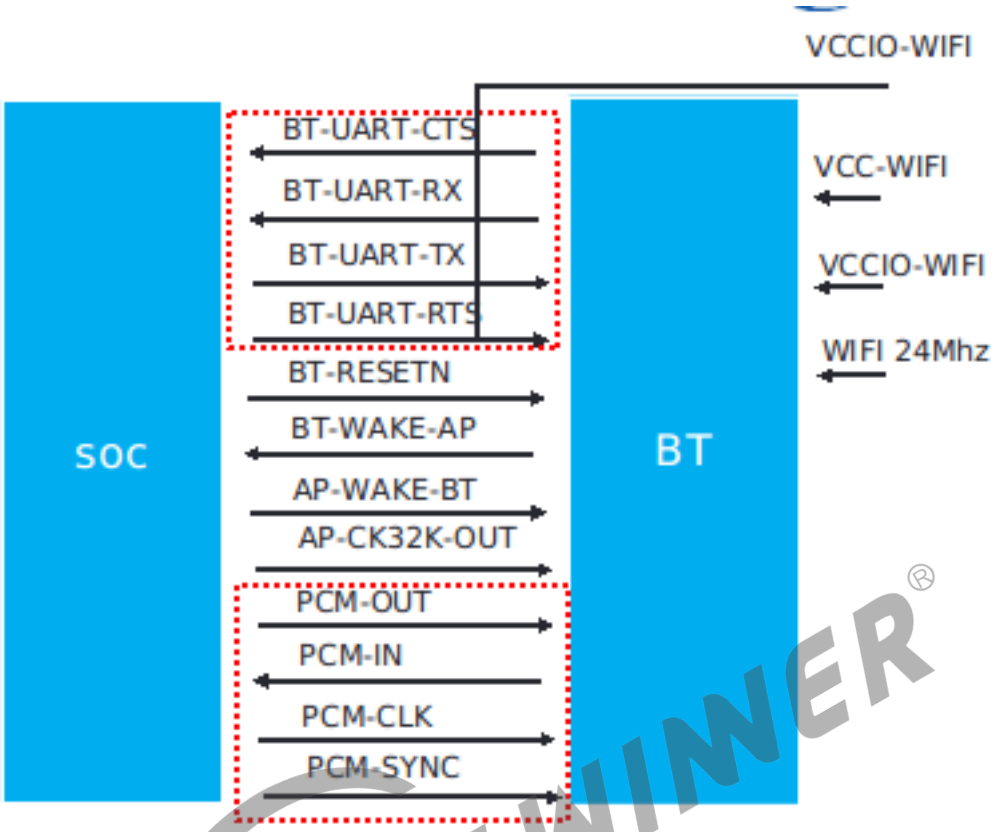

图 3-2：主控与 bt 硬件连接简图

bt 工作需要满足以下几个条件。

- 供电：蓝牙供电一般需要两路电源，一路为主电源，另一路用于 IO 上拉电源。
- 复位：需要对 BT-RESETN 进行复位操作。
- AP-WAKE-BT：主要用于使 bt 进行休眠，当 bt 正常工作时，需要输出高电平。
- 接口：主控与 bt 大部分数据通信都是通过 uart 接口，而部分模组蓝牙语音通话走 pcm 接口。
- 24/26MHz 时钟信号。
- 32.768KHz 信号：根据模组而定，有些模组内部通过（5）中的输入的 clk 进行分频得到，有些需要外部单独输入该信号。

软件上，Bluetooth 需要配置的是供电，AP-WAKE-BT 拉高，BT-RESETN 可进行复位，输出 32khz 信号。关于供电部分，大部分的模组都是 wifi，bt 一体，所以大部分操作同 wifi 一致，详情可参考《wifi 开发指南》。

以下是 linux 4.9 sysconfig.fex Bluetooth 相关的配置

```
[uart1]
uart1_used      = 1
uart1_type      = 4
```

```
uart1_tx          = port:PG06<2><1><default><default>
uart1_rx          = port:PG07<2><1><default><default>
uart1_rts         = port:PG08<2><1><default><default>
uart1_cts         = port:PG09<2><1><default><default>
[uart1_suspend]
uart1_tx          = port:PG06<7><1><default><default>
uart1_rx          = port:PG07<7><1><default><default>
uart1_rts         = port:PG08<7><1><default><default>
uart1_cts         = port:PG09<7><1><default><default>

[bt]
bt_used           = 1
compatible        = "allwinner,sunxi-bt"
clocks            = "losc_out"
bt_rst_n          = port:PE04<1><default><default><0>

[btlpm]
btlpm_used        = 1
compatible        = "allwinner,sunxi-btlpm"
uart_index        = 1
bt_wake           = port:PE03<1><default><default><1>
bt_hostwake       = port:PE00<6><default><default><0>
```

```
uart字段：主要配置uart rx，rx，ctx，rtx所使用的gpio pin。
bt字段：主要是配置bt 复位所使用的gpio pin。
btlmp字段：  主要是配置host休眠与唤醒bt，bt唤醒host所使用的gpio pin。
```

3.2 bluez 协议栈配置

tina 目前已经适配了 RTL8723DS，XR829 的模组，只需要在 kernel_menuconfig 和 menuconfig 选上对应的配置即可。

以下列出各个模组 kernel_menuconfig 以及 menuconfig 的选项。

（1）公用配置内核部分：make kernel_menuconfig

```
[*] Networking support  --->
  <*>   Bluetooth subsystem support  --->
    [*]   Bluetooth Classic (BR/EDR) features
    //如果需要支持hfp，需要选上下面两个选项
    <*>     RFCOMM protocol support
    [*]       RFCOMM TTY support
  <*>   RF switch subsystem support  --->
    [ ]   RF switch input support //这个不能选
    <*>   GPIO RFKILL driver
```

用户空间部分：make menuconfig 配置

```
  Utilities  --->
    <*> bluez-daemon..................................... Bluetooth daemon
    <*> bluez-utils...................................... Bluetooth utilities

  Allwinner  --->
    wireless  --->
      <*> btmanager-v4.0................................. bluetooth manager core
```

```
      [*]   Enable btmanager demo support
      -*- wirelesscommon............................ Allwinner Wi-Fi/BT Public lib
```

（2）XR829 模组

make kernel_menuconfig 配置

```
[*] Networking support  --->
  <*>   Bluetooth subsystem support  --->
    Bluetooth device drivers  --->
      [*]   UART (H4) protocol support
      <*> Xradio Bluetooth sleep driver support
      <*> Xradio Bluetooth farmware debug interface support
      [*] Xradio protocol support
      [*] Hfp audio over pcm
```

make menuconfig 配置

```
Kernel modules--->
  Wireless Drivers--->
    <*> kmod-net-xr829.................................. xr829 support (staging)
    <*> kmod-net-xrbtlpm........................ xradio bt lpm support (staging)
Firmware--->
     <*> xr829-firmware..................................... Xradio xr829 firmware
     [ ] xr829 with 40M sdd  //如果是40M晶振，需要选择上。
```

（2）RTL8723DS 模组

make kernel_menuconfig 配置

```
[*] Networking support  --->
  <*>   Bluetooth subsystem support  --->
    Bluetooth device drivers  --->
      [*] Realtek Three-wire UART (H5) protocol support
      <*> HCI UART driver
      [*]   UART (H4) protocol support
      [*] Hfp audio over pcm //如果支持hfp over pcm选上
```

make menuconfig 配置

```
Kernel modules--->
  Wireless Drivers--->
    <*> kmod-net-rtl8723ds........................... RTL8723DS support (staging)

Firmware--->
  <*> r8723ds-firmware.............................. RealTek RTL8723DS firmware

Utilities  --->
  rtk_hciattach  --->
    <*> rtk_hciattach................... Realtek BT HCI UART initialization tools
```

4 经典蓝牙开发介绍

开源的 bluez 协议栈，并不能满足用户的需求，它还是缺少众多组件，因而 btmanager 应用而生。本章节开始重点介绍 btmanager 经典蓝牙部分 API 使用，当前支持情况如下：

- GAP
- A2DP Source
- A2DP sink
- AVRCP
- HFP HF(针对 HFP over pcm)
- SPP

btmanager 代码位置：

```
tina/package/allwinner/wireless/btmanager4.0
```

使用示例：

```
tina/package/allwinner/wireless/btmanager4.0/demo
```

4.1 回调函数

4.1.1 btmg_callback_t

btmg_callback_t 规定了 btmanager 的总回调结构体，用户在使用的时候只需要定义一个 btmg_callback_t 类型的函数指针，通过调用 bt_manager_preinit() 可对该指针进行初始化并分配相应的空间，根据实际需求填充对应的子项，再通过 bt_manager_init() 函数注册到 btmanager 内部；

详细使用方法请参考 demo 源码的 bt_test.c 的 _bt_init() 函数；

```
typedef struct btmg_callback_t {
    btmg_adapter_callback_t btmg_adapter_cb;
    btmg_gap_callback_t btmg_gap_cb;
    btmg_agent_callback_t btmg_agent_cb;
    btmg_a2dp_sink_callback_t btmg_a2dp_sink_cb;
    btmg_a2dp_source_callback_t btmg_a2dp_source_cb;
    btmg_avrcp_callback_t btmg_avrcp_cb;
    btmg_hfp_callback_t btmg_hfp_cb;
    btmg_spp_server_callback_t btmg_spp_server_cb;
    btmg_spp_client_callback_t btmg_spp_client_cb;
```

```
    btmg_gatt_server_cb_t btmg_gatt_server_cb;
    btmg_gatt_client_cb_t btmg_gatt_client_cb;
} btmg_callback_t;
```

- btmg_adapter_cb：本地蓝牙设备状态的回调；
- btmg_gap_cb：GAP 相关的回调；
- btmg_agent_cb：Agent 的回调；
- btmg_a2dp_sink_cb：A2DP Sink 的回调；
- btmg_a2dp_source_cb：A2DP Source 的回调；
- btmg_avrcp_cb：AVRCP 的回调；
- btmg_hfp_cb：HFP 的回调；
- btmg_spp_server_cb：SPP Server 的回调；
- btmg_spp_client_cb：SPP Clinet 的回调；
- btmg_gatt_server_cb：GATT Server 的回调；
- btmg_gatt_client_cb：GATT Client 的回调；

以上的回调函数会在本文后续有更深入的介绍；

4.1.2 btmg_gap_callback_t

btmg_gap_callback_t 主要是 GAP 的相关回调函数，主要是关于扫描、绑定、设备增加、信号强度的；

```
typedef struct {
    bt_gap_scan_status_cb gap_scan_status_cb; /*used for return scan status of BT*/
    bt_gap_dev_device_add_cb gap_device_add_cb; /*used for device found event*/
    bt_gap_dev_device_remove_cb gap_device_remove_cb; /*used for device found event*/
    bt_gap_le_scan_report_cb gap_le_scan_report_cb;
    bt_gap_update_rssi_cb gap_update_rssi_cb; /*update rssi for scan and bonded devices*/
    bt_gap_bond_state_cb gap_bond_state_cb; /*used for bond state event*/
} btmg_gap_callback_t;
```

- gap_scan_status_cb：扫描状态的回调，如开始扫描，停止扫描；
- gap_device_add_cb：扫描到新设备的回调；
- gap_device_remove_cb：移除扫描缓存设备的回调；
- gap_le_scan_report_cb：BLE 扫描的实际结果；
- gap_update_rssi_cb：扫描时缓存设备的信号强度上报的回调；
- gap_bond_state_cb：配对状态的回调；

4.2 通用数据结构

4.2.1 btmg_log_level_t

log 控制等级

```
typedef enum {
  BTMG_LOG_LEVEL_NONE = 0,        //关闭任何打印
  BTMG_LOG_LEVEL_ERROR,           //只打印错误信息
  BTMG_LOG_LEVEL_WARNG,           //打印错误和警告信息
  BTMG_LOG_LEVEL_INFO,            //打印提示信息
  BTMG_LOG_LEVEL_DEBUG            //打开调试信息
}btmg_log_level_t;
```

- NONE：关闭任何打印
- ERROR：只打印错误信息
- WARNG：打印错误和警告信息
- INFO：打印提示信息
- DEBUG：打开调试信息

4.2.2 btmg_adapter_state_t

BT 状态

```
typedef enum {
    BTMG_ADAPTER_OFF,
    BTMG_ADAPTER_TURNING_ON,
    BTMG_ADAPTER_ON,
    BTMG_ADAPTER_TURNING_OFF,
} btmg_adapter_state_t;
```

数据结构 btmg_adapter_state_t 指示 BT 处于的状态。bt_manager_get_adapter_state() 可主动获取当前 BT 的状态。如果注册了 bt_adapter_state_cb() 回调函数，BT 状态改变时，会立即回调返回当前状态。

4.2.3 btmg_scan_mode_t

BT Scan 模式

```
typedef enum {
    BTMG_SCAN_MODE_NONE,                    //设备不可被发现、不可被连接
    BTMG_SCAN_MODE_CONNECTABLE,             //设备可被连接、不可被发现
    BTMG_SCAN_MODE_CONNECTABLE_DISCOVERABLE,//设备可被发现、可被连接
}btmg_scan_mode_t;
```

- NONE：设备不可被发现、不可被连接

- CONNECTABLE：设备可被连接、不可被发现
- DISCOVERABLE：设备可被发现、可被连接

以上的三种模式可以通过 hciconfig 命令查看，在设备 adb 或者串口中输入 hciconfig -a，可以获得如下信息：

```
root@TinaLinux:/# hciconfig -a
hci0:   Type: Primary  Bus: UART
        BD Address: 7C:A7:B0:26:50:65  ACL MTU: 1021:8  SCO MTU: 255:12
        UP RUNNING PSCAN ISCAN
        RX bytes:86155 acl:41 sco:0 events:453 errors:0
        TX bytes:6847 acl:37 sco:0 commands:61 errors:0
        Features: 0xff 0xff 0xff 0xfe 0xdb 0xfd 0x7b 0x87
        Packet type: DM1 DM3 DM5 DH1 DH3 DH5 HV1 HV2 HV3
        Link policy: RSWITCH HOLD SNIFF PARK
        Link mode: SLAVE ACCEPT
        Name: 'aw-bt-test-50-65'
        Class: 0x080000
        Service Classes: Capturing
        Device Class: Miscellaneous,
        HCI Version: 4.1 (0x7)  Revision: 0xaa7a
        LMP Version: 4.1 (0x7)  Subversion: 0xdd92
        Manufacturer: Realtek Semiconductor Corporation (93)
```

PSCAN 表示可被连接，ISCAN 表示可被发现。如上信息中，这两个状态都有，表示此时设备可被发现、可被连接；

- 可被连接不可被发现模式下，已经配设备之间可以直接连接
- 通过 bt_manager_enable(true) 打开蓝牙，默认是可被发现、可被连接模式;

4.2.4 btmg_io_capability_t

BT IO 能力

```
typedef enum {
    BTMG_IO_CAP_KEYBOARDDISPLAY = 0,
    BTMG_IO_CAP_DISPLAYONLY,
    BTMG_IO_CAP_DISPLAYYESNO,
    BTMG_IO_CAP_KEYBOARDONLY,
    BTMG_IO_CAP_NOINPUTNOOUTPUT,
} btmg_io_capability_t;
```

蓝牙配对时需要交换本机和目标机的 IO 能力，IO 能力包含 input 和 output；

- KEYBOARDDISPLAY：设备可以输入 0-9，确认键和 YES/NO 的能力，也有输出显示的能力
- DISPLAYONLY：设备只有输出显示的能力
- DISPLAYYESNO：设备有输入 YES 和 NO 的能力，也能够输出显示
- KEYBOARDONLY：设备可以输入 0-9，确认键和 YES/NO 的能力
- NOINPUTNOOUTPUT：设备没有输入和输出的能力

4.2.5 btmg_bond_state_t

BT 绑定状态

```
typedef enum {
    BTMG_BOND_STATE_UNBONDED,
    BTMG_BOND_STATE_BONDING,
    BTMG_BOND_STATE_BONDED,
    BTMG_BOND_STATE_BOND_FAILED,
} btmg_bond_state_t;
```

btmg_bond_state_t 规定了 BT 处于的配对状态，通过注册的回调函数 gap_bond_state_cb() 返回配对状态。

4.2.6 btmg_bt_device_t

BT device 信息

```
typedef struct {
    char *remote_address;
    char *remote_name;
    char *icon;
    bool paired;
    bool trusted;
    bool blocked;
    bool connected;
    int16_t rssi;
    uint32_t r_class;
} btmg_bt_device_t;
```

btmg_bt_device_t 描述的是对端的蓝牙设备信息，用于记录扫描到、已连接、已配对的蓝牙设备信息等，该结构体会经常使用到。

- remote_address：设备 mac 地址信息
- remote_name：设备名称
- icon：设备图标类型，由 bluez 协议栈上报，简单指示是哪种设备类型
- paired：指示配对状态
- trusted：指示是否受信任
- blocked：指示是否可以建立连接，如果为 true，将会拒绝该设备发起的连接
- connected：指示连接状态
- rssi：指示设备信号强度
- r_class：设备的 COD 类型

4.2.7 btmg_scan_type_t

BT 扫描设备类型

```
typedef enum {
    BTMG_SCAN_AUTO,
    BTMG_SCAN_BR_EDR,
    BTMG_SCAN_LE,
} btmg_scan_type_t;
```

btmg_scan_type_t 规定了调用 bt_manager_start_discovery() 扫描的设备类型：

- BTMG_SCAN_AUTO：自动，不过滤的情况下默认为自动；
- BTMG_SCAN_BR_EDR：扫描经典蓝牙设备；
- BTMG_SCAN_LE：扫描 BLE 蓝牙设备；

4.2.8 btmg_scan_filter_t

BT 扫描过滤信息

```
typedef struct {
    const char **uuid_list;
    uint32_t uuid_num;
    int16_t rssi;
    btmg_scan_type_t type;
} btmg_scan_filter_t;
```

btmg_scan_filter_t 规定了扫描过滤的信息：

- uuid_list：指定过滤的 uuid 列表；
- uuid_num：指定 uuid 列表里面的 uuid 个数；
- rssi：设置支持最小的信号强度，例如：-90，表示支持扫描到最小的信号强度为-90db；
- type：指定扫描设备的类型；

4.3 通用 API

4.3.1 设置打印级别

函数原型	int bt_manager_set_loglevel(btmg_log_level_t log_level)
参数说明	btmg_log_level_t 打印等级类型，详见通用数据结构章节；
返回说明	int 0: 成功；非 0: 失败；
功能描述	设置 bt_manager 内部打印等级，默认打印等级是 BTMG_LOG_LEVEL_WARNG；

4.3.2 获取打印级别

函数原型	btmg_log_level_t bt_manager_get_loglevel(void)
参数说明	无；
返回说明	返回当前使用的 btmg_log_level_t 类型打印等级值；
功能描述	获取 bt_manager 内部当前使用的打印等级；

4.3.3 设置拓展调试标志位

函数原型	int bt_manager_set_ex_debug_mask(int mask)
参数说明	详细说明见下文描述;
返回说明	int 0: 成功；非 0: 失败；
功能描述	设置拓展调试标志位，用于打开特殊打印输出；

ex_debug_mask 定义在 bt_log.h 中：

```
enum ex_debug_mask {
    EX_DBG_RING_BUFF_WRITE = 1 << 0,
    EX_DBG_RING_BUFF_READ = 1 << 1,
    EX_DBG_A2DP_SINK_RATE = 1 << 2,
    EX_DBG_A2DP_SOURCE_LOW_RATE = 1 << 3,
    EX_DBG_A2DP_SOURCE_UP_RATE = 1 << 4,
    EX_DBG_A2DP_SINK_DUMP_RB = 1 << 5,
    EX_DBG_A2DP_SINK_DUMP_HW = 1 << 6,
    EX_DBG_A2DP_SOURCE_DUMP_UP = 1 << 7,
    EX_DBG_A2DP_SOURCE_DUMP_LOW = 1 << 8,
    EX_DBG_HFP_HF_DUMP_Capture = 1 << 9,
    EX_DBG_MASK_MAX = 1 << 10,
};
```

- EX_DBG_RING_BUFF_WRITE: ring buff 写入的调试日志；
- EX_DBG_RING_BUFF_READ: ring buff 读出的调试日志；
- EX_DBG_A2DP_SINK_RATE：A2DP Sink 速率日志；
- EX_DBG_A2DP_SOURCE_LOW_RATE：A2DP Source 写入 bluealsa 虚拟声卡的速率日志；
- EX_DBG_A2DP_SOURCE_UP_RATE：应用调用 bt_manager_a2dp_src_stream_send API 的速率统计；
- EX_DBG_A2DP_SINK_DUMP_RB：Dump 出从 bluealsa 读到的裸数据，路径:/tmp/a2dp_bluealsa.raw
- EX_DBG_A2DP_SINK_DUMP_HW：Dump 出写到本地声卡的裸数据, 路径:/tmp/a2dp_sink_hw.raw；
- EX_DBG_A2DP_SOURCE_DUMP_UP：Dump 出调用 bt_manager_a2dp_src_stream_send 发送的裸数据，路径:/tmp/a2dp_src_stream.raw；

- EX_DBG_A2DP_SOURCE_DUMP_LOW：Dump 出写入 bluealsa 虚拟声卡的裸数据，路径:/tmp/a2dp_bluealsa.raw；
- EX_DBG_HFP_HF_DUMP_Capture：Dump 出 HFP 的 Capture 数据, 路径:/tmp/hfp_cap.raw‘’

调用例子：bt_manager_set_ex_debug_mask（EX_DBG_A2DP_SOURCE_LOW_RATE | EX_DBG_A2DP_SOURCE_UP_RATE）；

4.3.4 获取拓展调试标志位

函数原型	int bt_manager_get_ex_debug_mask(void)
参数说明	无；
返回说明	返回当前设置拓展调试标志位的值；
功能描述	获取当前设置拓展调试标志位的值；

4.3.5 获取错误信息

函数原型	const char *bt_manager_get_error_info(int error)
参数说明	int error：btmanager 内部定义的错误码；
返回说明	返回错误提示信息描述字符串；
功能描述	通过错误码获取对应的错误提示信息；

4.3.6 预初始化

函数原型	int bt_manager_preinit(btmg_callback_t **btmg_cb)
参数说明	指向 btmg_callback_t 指针类型的指针；
返回说明	int 0: 成功；非 0: 失败；
功能描述	用于对用户定义的回调函数结构体指针 btmg_callback_t * 进行分配内存初始化，用户也可自行显示地对指针进行初始化。初始化的指针在用户程序 exit 之前必须调用 bt_manager_deinit() 进行回收；

4.3.7 初始化

函数原型	int bt_manager_init(btmg_callback_t *btmg_cb)
参数说明	已经初始化了的回调函数结构体指针 btmg_callback_t *;
返回说明	int 0: 成功；非 0: 失败；
功能描述	bt_manager 的初始化，为内部变量分配空间

4.3.8 反初始化

函数原型	int bt_manager_deinit(btmg_callback_t *btmg_cb)
参数说明	指向 btmg_callback_t 类型的指针；
返回说明	int 0: 成功；非 0: 失败；
功能描述	bt_manager 反初始化。在 bt_manager 蓝牙应用程序退出前必要进行的调用;

4.3.9 使能 profile

函数原型	int bt_manager_enable_profile(int profile)
参数说明	int profile：传入的 profile 值，详细见下文描述；
返回说明	int 0: 成功；非 0: 失败。
功能描述	设置蓝牙的 profile 功能，在 bt_manager_enable() 之前设置有效，后续在用户调用 bt_manager 的进程未退出的情况下该设置一直有效。

profile 定义如下：

```
#define BTMG_A2DP_SINK_ENABLE (1 << 0)
#define BTMG_A2DP_SOUCE_ENABLE (1 << 1)
#define BTMG_AVRCP_ENABLE (1 << 2)
#define BTMG_HFP_HF_ENABLE (1 << 3)
#define BTMG_GATT_SERVER_ENABLE (1 << 4)
#define BTMG_GATT_CLIENT_ENABLE (1 << 5)
#define BTMG_SPP_ENABLE (1 << 6)
```

调用例子：bt_manager_enable_profile(BTMG_A2DP_SOUCE_ENABLE | BTMG_SPP_ENABLE)

4.3.10 设置默认 profile

函数原型	int bt_manager_set_enable_default(bool is_default)
参数说明	bool is_default： - true: 使用 bluetooth.json 配置文件中默认的配置； - false: 不使用 bluetooth.json 配置文件中默认的配置；
返回说明	int 0: 成功；非 0: 失败。
功能描述	是否使用 bluetooth.json 配置文件的默认配置。在 bt_manager_enable() 之前设置有效，后续在用户调用 bt_manager 的进程未退出的情况下该设置一直有效。

4.3.11 蓝牙开关

函数原型	int bt_manager_enable(bool enable)
参数说明	bool enable： - true: 打开 BT； - false: 关闭 BT；
返回说明	int 0: 成功；非 0: 失败；
功能描述	开关蓝牙的作用, 涉及 profile 配置文件、btmanager 内部线程、hci 设备、蓝牙协议栈等操作;

4.3.12 设置本地设备 scan 模式

函数原型	int bt_manager_set_scan_mode(btmg_scan_mode_t mode)
参数说明	btmg_discovery_mode_t mode：BT 设备扫描模式，详见通用数据结构章节；
返回说明	int 0: 成功；非 0: 失败；
功能描述	设置本地 BT 设备扫描模式（PSCAN,ISCAN），决定设备是否可连接，可发现；

4.3.13 设置扫描过滤

函数原型	int bt_manager_discovery_filter(btmg_scan_filter_t *filter)
参数说明	结构体指针 btmg_scan_filter_t *filter，详见通用数据结构章节；
返回说明	int 0: 成功；非 0: 失败；
功能描述	扫描过滤设置，如指定经典蓝牙扫描，指定信号强度扫描；

4.3.14 开始扫描

函数原型	int bt_manager_start_discovery(void)
参数说明	无；
返回说明	int 0: 成功；非 0: 失败；
功能描述	开始扫描周围的蓝牙设备；注意： - 如果设备是第一次连接，务必先要扫描到才能发起连接，否则无法连接； - 每次开始扫描之前，btmanager 内部会清除上次的扫描列表才开始扫描； - 不会扫到已配对的设备； - 如果没有停止扫描，会一直保持扫描状态； - 扫描状态请通过 gap 回调函数 gap_disc_status_cb() 判断； - 该接口用于经典蓝牙扫描连接，建议在扫描前指定过滤经典蓝牙设备，详细参考 demo 代码；

4.3.15 停止扫描

函数原型	int bt_manager_stop_discovery(void)
参数说明	无；
返回说明	int 0: 成功；非 0: 失败；
功能描述	停止 BT 扫描设备的动作： - 扫描状态请通过 gap 回调函数 gap_disc_status_cb() 判断； - Bluez 协议栈会在停止扫描 3 分钟后自动清空扫描列表缓存；

4.3.16 判断扫描状态

函数原型	bool bt_manager_is_discovering(void)
参数说明	无；
返回说明	bool true:BT 扫描中；bool false:BT 未扫描；
功能描述	获得当前 BT 扫描状态；

4.3.17 蓝牙配对

函数原型	int bt_manager_pair(char *addr)
参数说明	char *addr: 需要配对的 BT 设备 mac 地址；
返回说明	int 0: 成功；非 0: 失败；

函数原型	int bt_manager_pair(char *addr)
功能描述	配对蓝牙设备，配对状态请根据 gap 回调函数 gap_bond_state_cb() 判断；

4.3.18 取消配对

函数原型	int bt_manager_unpair(char *addr)
参数说明	char *addr: 需要取消配对的 BT 设备 mac 地址；
返回说明	int 0: 成功；非 0: 失败。
功能描述	取消配对，如果设备处于连接状态，会先断开后再取消配对；配对状态请根据 gap 回调函数 gap_bond_state_cb() 判断；

4.3.19 获取已配对设备列表

函数原型	int bt_manager_get_paired_devices(btmg_bt_device_t **dev_list, int *count)
参数说明	- btmg_bt_device_t **dev_list：指向 btmg_callback_t 指针类型的指针； - int *count：用于获取列表个数;
返回说明	int 0: 成功；非 0: 失败。
功能描述	获取已配对列表，获取完后，务必使用 bt_manager_free_paired_devices(btmg_bt_device_t *dev_list, int count) 释放 dev_list，避免内存泄漏；

4.3.20 释放已配对列表的内存

函数原型	int bt_manager_free_paired_devices(btmg_bt_device_t *dev_list, int count)
参数说明	- btmg_bt_device_t *dev_list：指向 btmg_callback_t 结构体类型的指针； - int count：已配对设备个数;
返回说明	int 0: 成功；非 0: 失败；
功能描述	用于释放 bt_manager_get_paired_devices() API 获取已配对设备时已申请到的内存，避免内存泄漏；

4.3.21 获取本地蓝牙状态

函数原型	btmg_adapter_state_t bt_manager_get_adapter_state(void)
参数说明	无；
返回说明	btmg_adapter_state_t 类型蓝牙状态；
功能描述	获取蓝牙开关状态；

4.3.22 获取本地蓝牙名称

函数原型	int bt_manager_get_adapter_name(char *name)
参数说明	char *name: 用于保存蓝牙名称的指针；
返回说明	int 0: 成功；非 0: 失败；
功能描述	获取本地蓝牙设备名称；

4.3.23 设置本地蓝牙名称

函数原型	int bt_manager_set_adapter_name(const char *name)
参数说明	char *name: 用于设置的 BT 名称。字符串长度不能超过 248，否则会被截断；
返回说明	int 0: 成功；非 0: 失败；
功能描述	设置本地蓝牙设备名称;

4.3.24 获取本地蓝牙 mac 地址

函数原型	int bt_manager_get_adapter_address(char *addr)
参数说明	char *addr: 用于保存本地蓝牙设备地址的指针; 空间大小推荐设置为 18；
返回说明	int 0: 成功；非 0: 失败;
功能描述	获取本地蓝牙设备地址;

4.3.25 蓝牙通用连接

函数原型	int bt_manager_connect(const char *addr)
参数说明	const char *addr: 指定需要连接的蓝牙设备地址;
返回说明	int 0: 调用成功；非 0: 调用失败;
功能描述	蓝牙通用连接； - 目前 SPP Profile 和 GATT Profile 不使用该接口连接； - 设备连接状态请通过回调函数判断，比如： bt_a2dp_source_connection_state_cb、 bt_a2dp_sink_connection_state_cb;

4.3.26 蓝牙通用断开

函数原型	int bt_manager_disconnect(const char *addr)
参数说明	const char *addr: 指定需要断开的蓝牙设备地址;
返回说明	int 0: 成功；非 0: 失败;
功能描述	蓝牙通用断开连接； - 目前 SPP Profile 和 GATT Profile 不使用该接口连接； - 设备连接状态请通过回调函数判断，比如： bt_a2dp_source_connection_state_cb、 bt_a2dp_sink_connection_state_cb;

4.3.27 判断对端设备是否连接

函数原型	bool bt_manager_device_is_connected(const char *addr)
参数说明	const char *addr: 需要判断的蓝牙设备地址;
返回说明	bool true: 已连接；bool false: 未连接；
功能描述	判断对端设备是否跟本地设备存在连接关系；

4.3.28 移除缓存设备

函数原型	int bt_manager_remove_device(const char *addr)
参数说明	const char *addr: 需要移除的蓝牙设备地址；
返回说明	int 0: 成功；非 0: 失败；
功能描述	移除指定缓存的蓝牙设备，移除扫描列表缓存；

4.3.29 设置 Page Timeout

函数原型	int bt_manager_set_page_timeout(int slots)
参数说明	int slots: 一个 slots 为 0.625ms，因此设置的 timeout = slots * 0.625；
返回说明	int 0: 成功；非 0: 失败；
功能描述	设置 page timeout，即蓝牙链路建立的最大超时时间；

4.3.30 设置 Link Supervision Timeout

函数原型	int bt_manager_set_link_supervision_timeout(const char *addr, int slots)
参数说明	- const char *addr：需要设置并且已连接的设备 mac 地址； - int slots: 一个 slots 为 0.625ms，因此设置的 timeout = slots * 0.625；
返回说明	int 0: 成功；非 0: 失败；
功能描述	- 设置 Supervision 超时时间； - Supervision Timeout 作用：判断设备断开的超时时间，ACL 连接双方约定多长时间没有联系上算断开； - 如果要设置，一般在设备刚连接上的时候设置，断开后会失效，因此每次连接上都需要设置一次； - 例如：设备已经连上一个蓝牙音箱，此时主动将蓝牙音箱的电关闭，应用层可能会过一段时间才能收到断开的事件。如果设置了 Supervision Timeout 为 1000ms，那么蓝牙音箱断电 1000ms 后，应用层就可以收到断开的事件了。

4.4 Bluez Agent

bluez 协议栈使用 agent(代理) 来实现配对，配对过程中的交互都由 agent 来完成。代理根据不同的配对方式有不同的行为；

4.4.1 回调函数

```
typedef struct {
    bt_agent_request_pincode agent_request_pincode;
    bt_agent_display_pincode agent_display_pincode;
    bt_agent_request_passkey agent_request_passkey;
    bt_agent_display_passkey agent_display_passkey;
    bt_agent_request_confirm_passkey agent_requestconfirm_passkey;
    bt_agent_request_authorize agent_request_authorize;
    bt_agent_authorize_service agent_authorize_service;
} btmg_agent_callback_t;
```

- agent_request_pincode: 收到对端设备 pincode 的请求，需要在回调中回复 pincode，回复内容为 1~16 个字符的字符串，可以是字母数字；
- agent_display_pincode: 显示 pincode 以进行身份验证；
- agent_request_passkey: 对端需要获取验证密钥，需要回复一个 1~999999 之间的数值;
- agent_display_passkey: 显示 passkey 以进行身份验证；
- agent_request_confirm_passkey: 需要确认密码才能进行身份验证，需要回复一个空消息；
- agent_request_authorize: 需要去认证对端的配对请求，否则是触发 just-works 配对；
- agent_authorize_service: 需要认证一个连接或者服务请求；

4.4.2 Agent API

4.4.2.1 设置 io_capability

函数原型	int bt_manager_agent_set_io_capability(btmg_io_capability_t io_cap);
参数说明	btmg_io_capability_t io_cap：输入、输出能力，详见通用数据结构章节；
返回说明	int 0: 成功；非 0: 失败。
功能描述	设置本地蓝牙设备的输入、输出能力；

4.4.2.2 发送 Pincode

函数原型	int bt_manager_agent_send_pincode(void *handle, char *pincode);
参数说明	- handle：回调函数返回的 handle； - pincode：pincode 内容为 1~16 个字符的字符串，可以是字母数字；
返回说明	int 0: 成功；非 0: 失败。
功能描述	回复 Pincode 给对端设备；

4.4.2.3 发送 passkey

函数原型	int bt_manager_agent_send_passkey(void *handle, unsigned int passkey);
参数说明	- handle：回调函数返回的 handle； - passkey：passkey 内容一个 1~999999 之间的数值；
返回说明	int 0: 成功；非 0: 失败。
功能描述	回复 passkey 给对端设备；

4.4.2.4 发送空回复

函数原型	int bt_manager_agent_pair_send_empty_response(void *handle);
参数说明	handle：回调函数返回的 handle；
返回说明	int 0: 成功；非 0: 失败。
功能描述	发送一个空消息给对端设备；

4.4.2.5 发送配对错误

函数原型	int bt_manager_agent_send_pair_error(void *handle, btmg_pair_request_error_t e, const char *err_msg);
参数说明	- handle：回调函数返回的 handle； - e：错误类型； - err_msg：错误信息，通常置 NULL；
返回说明	int 0: 成功；非 0: 失败。
功能描述	回复配对错误给对端设备；

4.5 A2DP Sink

作为 A2DP Sink 的设备一般是被 A2DP Source 设备连接，A2DP Source 设备发送音频流给 A2DP Sink 设备播放出来；用户使用 A2DP Sink，只需要使能 A2DP Sink 功能，关注几个回调即可。

A2DP Sink 的连接、断开使用通用连接和通用断开函数。

4.5.1 回调函数

```
typedef struct btmg_a2dp_sink_callback_t {
    bt_a2dp_sink_connection_state_cb a2dp_sink_connection_state_cb;
    bt_a2dp_sink_audio_state_cb a2dp_sink_audio_state_cb;
    bt_a2dp_sink_stream_cb a2dp_sink_stream_cb;
} btmg_a2dp_sink_callback_t;
```

- a2dp_sink_connection_state_cb：A2DP Sink 连接状态的回调；
- a2dp_sink_audio_state_cb：A2DP Sink 音频状态回调，建议使用 AVRCP 的音频状态；
- a2dp_sink_stream_cb：音频数据流的回调，部分用户希望直接得到 audio stream；

4.5.2 回调函数的参数

4.5.2.1 btmg_a2dp_sink_connection_state_t

btmg_a2dp_sink_connection_state_t 规定了 A2DPSink 的连接状态，通过注册回调函数 a2dp_sink_connection_state_cb() 即时返回连接状态。

```
typedef enum {
    BTMG_A2DP_SINK_DISCONNECTED,
    BTMG_A2DP_SINK_CONNECTING,
    BTMG_A2DP_SINK_CONNECTED,
    BTMG_A2DP_SINK_DISCONNECTING,
    BTMG_A2DP_SINK_CONNECT_FAILED,
    BTMG_A2DP_SINK_DISCONNEC_FAILED,
} btmg_a2dp_sink_connection_state_t;
```

4.5.2.2 btmg_a2dp_sink_audio_state_t

btmg_a2dp_sink_audio_state_t 规定了 A2DP Sink 音频流播放状态，通过注册的回调函数 a2dp_sink_audio_state_cb() 返回音频播放状态。由于 A2DP Sink 音频状态底层走的是 AVDTP 协议，部分手机蓝牙协议栈在暂停以后发送暂停状态存在数秒的延迟，因此通过 a2dp_sink_audio_state_cb() 返回音频播放状态会因手机而异存在暂停状态回调延迟于实际音频暂停状态数秒的情况。故不推荐使用 a2dp_sink_audio_state_cb() 返回音频播放状态，请使用基于 AVRCP 协议的 avrcp_play_state_cb() 回调函数，获取即时的音频播放状态。

```
typedef enum {
    BTMG_A2DP_SINK_AUDIO_SUSPENDED,
    BTMG_A2DP_SINK_AUDIO_STOPPED,
    BTMG_A2DP_SINK_AUDIO_STARTED,
} btmg_a2dp_sink_audio_state_t;
```

4.6 AVRCP CT

AVRCP CT 一般与 A2DP Sink 一起配合使用；

4.6.1 回调函数

btmg_avrcp_callback_t 结构体规定了 AVRCP 的回调；

```
typedef struct btmg_avrcp_callback_t {
    bt_avrcp_play_state_cb avrcp_play_state_cb;
    bt_avrcp_track_changed_cb avrcp_track_changed_cb;
    bt_avrcp_play_position_cb avrcp_play_position_cb;
    bt_avrcp_audio_volume_cb avrcp_audio_volume_cb;
} btmg_avrcp_callback_t;
```

- avrcp_play_state_cb：播放状态的回调；
- avrcp_track_changed_cb：播放信息的回调；
- avrcp_play_position_cb：播放进度的回调，实时返回当前音乐播放的进度；
- avrcp_audio_volume_cb：播放音量的回调；

4.6.2 回调函数的参数

4.6.2.1 btmg_avrcp_play_state_t

btmg_avrcp_play_state_t 规定了基于 AVRCP 协议返回的音频播放状态，通过注册的回调函数 avrcp_play_state_cb() 即时返回音频播放状态。相比使用 a2dp_sink_audio_state_cb() 返回音频播放状态，会具有更好的实时性，并且能返回更多的音频状态。

```
typedef enum {
    BTMG_AVRCP_PLAYSTATE_STOPPED,
    BTMG_AVRCP_PLAYSTATE_PLAYING,
    BTMG_AVRCP_PLAYSTATE_PAUSED,
    BTMG_AVRCP_PLAYSTATE_FWD_SEEK,
    BTMG_AVRCP_PLAYSTATE_REV_SEEK,
    BTMG_AVRCP_PLAYSTATE_FORWARD,
    BTMG_AVRCP_PLAYSTATE_BACKWARD,
    BTMG_AVRCP_PLAYSTATE_ERROR,
} btmg_avrcp_play_state_t;
```

- BTMG_AVRCP_PLAYSTATE_STOPPED：停止播放；
- BTMG_AVRCP_PLAYSTATE_PLAYING：开始播放；
- BTMG_AVRCP_PLAYSTATE_PAUSED：暂停播放；
- BTMG_AVRCP_PLAYSTATE_FWD_SEEK：快进；
- BTMG_AVRCP_PLAYSTATE_REV_SEEK：后退；

- BTMG_AVRCP_PLAYSTATE_FORWARD：下一曲；
- BTMG_AVRCP_PLAYSTATE_BACKWARD：上一曲；
- BTMG_AVRCP_PLAYSTATE_ERROR：错误状态；

4.6.2.2 btmg_track_info_t

btmg_track_info_t 规定了蓝牙音乐播放切换歌曲时 avrcp_track_changed_cb 回调接口返回的歌曲信息。

```
typedef struct btmg_track_info_t {
    char title[512];
    char artist[256];
    char album[256];
    char track_num[64];
    char num_tracks[64];
    char genre[256];
    char duration[256];
} btmg_track_info_t;
```

title：歌曲名称或者歌词，一般为歌曲名称，如果有歌词信息，则 title 为歌词内容；

artist：歌曲演唱者；

album：歌曲的专辑名称；

track_num：当前音乐位于音乐列表的顺序号；

num_tracks：总播放列表音乐数；

genre：音乐类型；

duration：歌曲播放总时长；

4.6.3 AVRCP 命令

```
typedef enum {
    BTMG_AVRCP_PLAY,
    BTMG_AVRCP_PAUSE,
    BTMG_AVRCP_STOP,
    BTMG_AVRCP_FASTFORWARD,
    BTMG_AVRCP_REWIND,
    BTMG_AVRCP_FORWARD,
    BTMG_AVRCP_BACKWARD,
    BTMG_AVRCP_VOL_UP,
    BTMG_AVRCP_VOL_DOWN,
} btmg_avrcp_command_t;
```

btmg_avrcp_command_t 规定了 API bt_manager_avrcp_command() 可以发送的 AVRCP 命令；

- BTMG_AVRCP_PAUSE：暂停播放；
- BTMG_AVRCP_STOP：停止播放；
- BTMG_AVRCP_FASTFORWARD：快进；
- BTMG_AVRCP_REWIND：后退；
- BTMG_AVRCP_FORWARD：下一曲；
- BTMG_AVRCP_BACKWARD：上一曲；
- BTMG_AVRCP_VOL_UP：调高音量；
- BTMG_AVRCP_VOL_DOWN：调低音量；

4.6.4 AVRCP CT API

4.6.4.1 发送 AVRCP 命令

函数原型	int bt_manager_avrcp_command(char *addr, btmg_avrcp_command_t command);
参数说明	- addr：对端设备的 mac 地址； - btmg_avrcp_command_t command：需要发送的命令；
返回说明	int 0: 发送成功；非 0: 发送失败；
功能描述	发送命令控制 A2DP Source 设备的播放；

4.7 A2DP Source

4.7.1 回调函数

目前 A2DP Source 只有连接状态的回调；

```
typedef struct btmg_a2dp_source_callback_t {
    bt_a2dp_source_connection_state_cb a2dp_source_connection_state_cb;
} btmg_a2dp_source_callback_t;
```

4.7.2 回调函数的参数

4.7.2.1 btmg_a2dp_source_connection_state_t

```
typedef enum {
    BTMG_A2DP_SOURCE_DISCONNECTED,
    BTMG_A2DP_SOURCE_CONNECTING,
    BTMG_A2DP_SOURCE_CONNECTED,
    BTMG_A2DP_SOURCE_DISCONNECTING,
    BTMG_A2DP_SOURCE_CONNECT_FAILED,
    BTMG_A2DP_SOURCE_DISCONNEC_FAILED,
} btmg_a2dp_source_connection_state_t;
```

4.7.3 A2DP Source API

A2DP Source API 调用顺序如下：

1. bt_manager_a2dp_src_init();
2. bt_manager_a2dp_src_stream_start();
3. bt_manager_a2dp_src_stream_send();
4. bt_manager_a2dp_src_stream_stop();
5. bt_manager_a2dp_src_deinit();

4.7.3.1 初始化

函数原型	int bt_manager_a2dp_src_init(uint16_t channels, uint16_t sampling)
参数说明	
	- uint16_t channels：音频通道，单声道 or 双声道； - uint16_t sampling：音频采样率;
返回说明	int 0: 成功；非 0: 失败；
功能描述	
	- A2DP Source 初始化，btmanager 内部根据传入的通道和采样率信息进行初始化；- 初始化务必在“启动播放线程”和“发送音频数据”之前完成；- 后面发送的音频参数务必和初始化的参数保持一致，否则需要反初始化后再重新初始化相应的音频参数；

4.7.3.2 启动播放

函数原型	int bt_manager_a2dp_src_stream_start(uint32_t len);
参数说明	uint32_t len：btmanager 内部每次写入蓝牙协议栈的数据长度; 建议设置为 4096；
返回说明	int 0: 成功；非 0: 失败；
功能描述	启动内部播放线程；

4.7.3.3 发送音频数据

函数原型	int bt_manager_a2dp_src_stream_send(char *data, int len);
参数说明	- char *data：要发送的数据; - len：每次发送给 btmanager 的数据长度，建议与 bt_manager_a2dp_src_stream_start() 中的 len 保持一致；
返回说明	int 0: 成功；非 0: 失败;
功能描述	应用发送音频数据给蓝牙协议栈；

4.7.3.4 判断是否在发送数据

函数原型	bool bt_manager_a2dp_src_is_stream_start(void)
参数说明	无；
返回说明	bool true: 处于发送状态；bool false: 不处于发送状态；
功能描述	判读当前是否处于发送状态；

4.7.3.5 停止播放

函数原型	int bt_manager_a2dp_src_stream_stop(bool drop)
参数说明	bool drop：true: 丢弃缓存区数据，立刻停止；false：等缓存区数据消耗完才停止；
返回说明	int 0: 成功；非 0: 失败;
功能描述	停止内部播放线程，停止播放；

4.7.3.6 反初始化

函数原型	int bt_manager_a2dp_src_deinit(void);
参数说明	无；
返回说明	int 0: 成功；非 0: 失败；
功能描述	反初始化，与 bt_manager_a2dp_src_init 相反；

4.8 AVRCP TG

4.8.1 回调函数

回调函数结构体 btmg_avrcp_callback_t，详见 AVRCP CT 章节的回调函数：

- bt_avrcp_play_state_cb avrcp_play_state_cb;

收到 A2DP Sink 端 AVRCP CT 的控制命令的回调；

- bt_avrcp_audio_volume_cb avrcp_audio_volume_cb;

音量改变的回调，一般用于绝对音量；

4.8.2 回调函数的参数

详见 AVRCP CT 章节的回调函数的参数；

4.8.3 AVRCP TG API

4.8.3.1 设置 A2DP 设备音量

函数原型	int bt_manager_a2dp_set_vol(int vol_value)
参数说明	int vol_value：设置的音量大小，范围区间 0~100；
返回说明	int 0: 成功；非 0: 失败;
功能描述	设置 A2DP 蓝牙音乐的音量大小： - 设备作为 A2dp Source：调用此接口可以调整播放音量大小； - 设备作为 A2dp Sink：如果对端设备支持并打开了绝对音量，可以调整绝对音量的大小；

4.8.3.2 获取 A2DP 设备音量

函数原型	int bt_manager_a2dp_get_vol(void)
参数说明	无;
返回说明	int：获取的音量大小;
功能描述	获取当前 A2DP 蓝牙音乐的音量大小;

4.8.3.3 发送播放状态

函数原型	void bt_manager_set_avrcp_status(int status);
参数说明	status：当前的播放状态
返回说明	int：获取的音量大小;
功能描述	获取当前 A2DP 蓝牙音乐的音量大小;

4.9 HFP HF

4.9.1 回调函数

```
typedef struct btmg_hfp_callback_t {
    bt_hfp_hf_connection_state_cb hfp_hf_connection_state_cb;
    bt_hfp_hf_event_cb hfp_hf_event_cb;
} btmg_hfp_callback_t;
```

4.9.1.1 bt_hfp_hf_connection_state_cb

连接状态回调

函数原型	typedef void (*bt_hfp_hf_connection_state_cb)(const char *bd_addr, btmg_hfp_hf_connection_state_t state);
参数说明	- char *bd_addr：对端设备地址; - btmg_hfp_hf_connection_state_t state：返回连接状态;
功能描述	即时返回与对端设备 hfp 的连接状态;

4.9.1.2 bt_hfp_hf_event_cb

返回的事件类型

函数原型	typedef void (*bt_hfp_hf_event_cb)(btmg_hfp_even_t event, char *data);
参数说明	- btmg_hfp_even_t event：对端返回的事件类型； - char *data：返回携带的数据；
功能描述	即时返回与对端设备交互事件；

4.9.2 回调函数的参数

4.9.2.1 btmg_hfp_hf_connection_state_t

连接状态

```
typedef enum {
    BTMG_HFP_HF_DISCONNECTED,
    BTMG_HFP_HF_CONNECTING,
    BTMG_HFP_HF_CONNECTED,
    BTMG_HFP_HF_DISCONNECTING,
    BTMG_HFP_HF_CONNECT_FAILED,
    BTMG_HFP_HF_DISCONNEC_FAILED,
} btmg_hfp_hf_connection_state_t;
```

4.9.2.2 btmg_hfp_even_t

事件类型

```
typedef enum {
    BTMG_HFP_CONNECT,
    BTMG_HFP_CONNECT_LOST,
    BTMG_HFP_CIND = 11, /* Indicator string from AG */
    BTMG_HFP_CIEV, /* Indicator status from AG */
    BTMG_HFP_RING, /* RING alert from AG */
    BTMG_HFP_CLIP, /* Calling subscriber information from AG */
    BTMG_HFP_BSIR, /* In band ring tone setting */
    BTMG_HFP_BVRA, /* Voice recognition activation/deactivation */
    BTMG_HFP_CCWA, /* Call waiting notification */
    BTMG_HFP_CHLD, /* Call hold and multi party service in AG */
    BTMG_HFP_VGM, /* MIC volume setting */
    BTMG_HFP_VGS, /* Speaker volume setting */
    BTMG_HFP_BINP, /* Input data response from AG */
    BTMG_HFP_BTRH, /* CCAP incoming call hold */
    BTMG_HFP_CNUM, /* Subscriber number */
    BTMG_HFP_COPS, /* Operator selection info from AG */
```

```
    BTMG_HFP_CMEE, /* Enhanced error result from AG */
    BTMG_HFP_CLCC, /* Current active call list info */
    BTMG_HFP_UNAT, /* AT command response fro AG which is not specified in HFP or HSP */
    BTMG_HFP_OK, /* OK response */
    BTMG_HFP_ERROR, /* ERROR response */
    BTMG_HFP_BCS /* Codec selection from AG */
} btmg_hfp_even_t;
```

4.9.3 HFP HF API

4.9.3.1 接听电话

函数原型	int bt_manager_hfp_hf_send_at_ata(void);
参数说明	无；
返回说明	int 0: 成功；非 0: 失败；
功能描述	主动接听电话；

4.9.3.2 拒接或挂断电话

函数原型	int bt_manager_hfp_hf_send_at_chup(void)
参数说明	无；
返回说明	int 0: 成功；非 0: 失败；
功能描述	主动拒接或挂断电话；

4.9.3.3 指定号码拨号

函数原型	int bt_manager_hfp_hf_send_at_atd(char *number)
参数说明	char *number：指定拨打的电话号码
返回说明	int 0: 成功；非 0: 失败；
功能描述	指定电话号码拨号

4.9.3.4 回拨

函数原型	int bt_manager_hfp_hf_send_at_bldn(void)
参数说明	无；
返回说明	int 0: 成功；非 0: 失败；

函数原型	int bt_manager_hfp_hf_send_at_bldn(void)
功能描述	回拨最后一个通话的号码；

4.9.3.5 查询与报告状态

函数原型	int bt_manager_hfp_hf_send_at_btrh(bool query, uint32_t val)
参数说明	- bool query：true：查询状态，等价于发送 “AT+BTRH¿‘；false：报告状态，等价于发送 “AT+BTRH=x“； - uint32_t val：该参数在报告状态的时候才有效； - val 值说明： 0：Put Incoming call on hold，保持来电； 1：Accept a held incoming call，接听来电； 2：Reject a held incoming call，拒绝来电；
返回说明	int 0: 成功；非 0: 失败;
功能描述	查询与报告通话的状态；

4.9.3.6 拨打分机

函数原型	int bt_hfp_hf_send_at_vts(char code)
参数说明	char code：分机号码；
返回说明	int 0: 成功；非 0: 失败;
功能描述	拨打分机，发送 AT+VTS=“分机号码”；

4.9.3.7 HF 侧发起音频连接

函数原型	int bt_manager_hfp_hf_send_at_bcc(void)
参数说明	无；
返回说明	int 0: 成功；非 0: 失败;
功能描述	发送 AT 命令 AT+BCC 给 AG，发起音频编解码连接；

4.9.3.8 获取本机号码

函数原型	int bt_manager_hfp_hf_send_at_cnum(void)
参数说明	无
返回说明	int 0: 成功；非 0: 失败；
功能描述	获取本机号码，本机号码将通过回调函数返回

4.9.3.9 扬声器音量调节

函数原型	int bt_manager_hfp_hf_send_at_vgs(uint32_t volume)
参数说明	uint32_t volume：音量大小，取值 0~15；
返回说明	int 0: 成功；非 0: 失败；
功能描述	调节手机端扬声器音量大小；

4.9.3.10 麦克风音量调节

函数原型	int bt_manager_hfp_hf_send_at_vgm(uint32_t volume)
参数说明	uint32_t volume：音量大小，取值 0~15；
返回说明	int 0: 成功；非 0: 失败；
功能描述	调节手机端麦克风音量大小；

4.9.3.11 发送自构建 AT 命令

函数原型	int bt_manager_hfp_hf_send_at_cmd(char *cmd)
参数说明	char *cmd：命令字符串；
返回说明	int 0: 成功；非 0: 失败；
功能描述	用户可以根据实际情况构建自己需要的 AT 命令，命令格式必须是“AT+XXX” 的格式；

4.10 SPP Client

4.10.1 回调函数

```
typedef struct btmg_spp_client_callback_t {
    bt_spp_client_connection_state_cb spp_client_connection_state_cb;
    bt_spp_client_recvdata_cb spp_client_recvdata_cb;
} btmg_spp_client_callback_t;
```

4.10.1.1 spp_client_connection_state_cb

连接状态回调

函数原型	typedef void (*bt_spp_client_connection_state_cb)(const char *bd_addr, btmg_spp_client_connection_state_t state);
参数说明	- const char *bd_addr：对端设备的 mac 地址； - btmg_spp_client_connection_state_t state：连接的状态；
功能描述	即时返回与对端设备连接状态；

4.10.1.2 spp_client_recvdata_cb

接收数据回调

函数原型	typedef void (*bt_spp_client_recvdata_cb)(const char *bd_addr, char *data, int data_len);
参数说明	- const char *bd_addr：对端设备的 mac 地址； - char *data：数据指针； - int data_len ： 数据长度；
功能描述	即时返回对端设备发送过来的数据；

4.10.2 回调函数参数

4.10.2.1 btmg_spp_client_connection_state_t

连接状态

```
typedef enum {
    BTMG_SPP_CLIENT_DISCONNECTED,
    BTMG_SPP_CLIENT_CONNECTING,
    BTMG_SPP_CLIENT_CONNECTED,
    BTMG_SPP_CLIENT_DISCONNECTING,
    BTMG_SPP_CLIENT_CONNECT_FAILED,
    BTMG_SPP_CLIENT_DISCONNEC_FAILED,
} btmg_spp_client_connection_state_t;
```

4.10.3 SPP Client API

4.10.3.1 连接设备

函数原型	int bt_manager_spp_client_connect(const char *dst)
参数说明	const char *dst：指定连接设备的 mac 地址；
返回说明	int 0: 成功；非 0: 失败；
功能描述	连接 SPP Server 设备，注意，如果目标设备是第一次连接，需要先通过 bt_manager_pair() 配对后再调用此接口连接；

4.10.3.2 发送数据

函数原型	int bt_manager_spp_client_send(char *data, uint32_t len)
参数说明	- char *data：数据指针； - uint32_t len：发送的数据长度；
返回说明	int 0: 成功；非 0: 失败；
功能描述	发送数据给对端设备；

4.10.3.3 断开设备

函数原型	int bt_manager_spp_client_disconnect(const char *dst)
参数说明	const char *dst：指定断开连接设备的 mac 地址；
返回说明	int 0: 成功；非 0: 失败；
功能描述	断开与对端设备连接;

4.11 SPP Server

4.11.1 回调函数

```
typedef struct btmg_spp_server_callback_t {
    bt_spp_server_connection_state_cb spp_server_connection_state_cb;
    bt_spp_server_accept_cb spp_server_accept_cb;
} btmg_spp_server_callback_t;
```

4.11.1.1 spp_server_connection_state_cb

连接状态回调

函数原型	typedef void (*bt_spp_server_connection_state_cb)(const char *bd_addr, btmg_spp_server_connection_state_t state);
参数说明	
	- const char *bd_addr：对端设备的 mac 地址； - btmg_spp_server_connection_state_t state：连接的状态；
功能描述即	时返回与对端设备连接状态；

4.11.1.2 spp_server_accept_cb

接收数据回调

函数原型	typedef void (*bt_spp_server_accept_cb)(const char *bd_addr, char *data, int data_len);
参数说明	
	- const char *bd_addr：对端设备的 mac 地址； - char *data：数据指针； - int data_len ： 数据长度；
功能描述即	时返回对端设备发送过来的数据；

4.11.2 回调函数的参数

4.11.2.1 btmg_spp_server_connection_state_t

连接状态

```
typedef enum {
    BTMG_SPP_SERVER_DISCONNECTED,
    BTMG_SPP_SERVER_CONNECTING,
    BTMG_SPP_SERVER_CONNECTED,
    BTMG_SPP_SERVER_DISCONNECTING,
    BTMG_SPP_SERVER_CONNECT_FAILED,
    BTMG_SPP_SERVER_DISCONNEC_FAILED,
} btmg_spp_server_connection_state_t;
```

4.11.3 SPP Server API

4.11.3.1 开始监听

函数原型	int bt_manager_spp_service_accept(void)
参数说明	无;
返回说明	int 0: 成功；非 0: 失败；
功能描述	初始化 SPP Server 并开设监听 Client 设备；

4.11.3.2 发送数据

函数原型	int bt_manager_spp_service_send(char *data, uint32_t len)
参数说明	- char *data：数据指针； - uint32_t len：数据长度；
返回说明	int 0: 成功；非 0: 失败；
功能描述	发送数据到对端设备；

4.11.3.3 断开连接

函数原型	int bt_manager_spp_service_disconnect(const char *dst);
参数说明	const char *dst：指定断开连接设备的 mac 地址；
返回说明	int 0: 成功；非 0: 失败；
功能描述	断开与对端设备连接;

4.12 总结经典蓝牙调用流程

关于经典蓝牙 API 的使用，已编写了使用示例，供用户参考，代码路径如下：

```
package/allwinner/wireless/btmanager4.0/demo

├── bt_cmd.c  API调用示例
├── bt_cmd.h
├── bt_test.c  main入口
└── Makefile   编译Makefile
```

经典蓝牙开发的调用流程：(可参考 bt_test.c::_bt_init)

1. 调用 bt_manager_preinit，预初始化，运行期间只需要调用一次。
2. 调用 bt_manager_enable_profile，设置本次要使用的 profile。
3. 填充 btmg_callback_t 结构体，即回调函数。
4. 调用 bt_manager_init，蓝牙初始化。
5. 调用 btmg_dev_list_new，创建已扫描设备列表。
6. 调用 bt_manager_enable，打开蓝牙。
7. 调用 bt_manager_set_adapter_name，设置蓝牙名称。
8. 调用 bt_manager_agent_set_io_capability，设置 io_capability 能力。
9. 调用 bt_manager_set_scan_mode，设置发现模式。

5 低功耗蓝牙开发介绍

5.1 通用数据结构

5.1.1 btmg_le_addr_type_t

设备地址类型

```
typedef enum {
    BTMG_LE_PUBLIC_ADDRESS = 0x00,
    BTMG_LE_RANDOM_ADDRESS = 0x01,
    BTMG_LE_PUBLIC_ADDRESS_ID = 0x02,
    BTMG_LE_RANDOM_ADDRESS_ID = 0x03,
} btmg_le_addr_type_t;
```

BLE 设备可以使用两种类型地址：Public Device Address 和 Random Device Address。

- BTMG_LE_PUBLIC_ADDRESS

Public Device Address：BR/EDR 同样使用该地址，一般需要向 IEEE 购买；

- BTMG_LE_RANDOM_ADDRESS

Random Device Address 根据安全性又分为 Static Device Address 和 Private Device Address；

1.Static Device Addres

蓝牙设备上电时随机生成的，一般保持一个上电周期；

2.Private Device Address

Private Device Address 又分为 Non-resolvable Private Address 和 Resolvable Private Address，主要区别在于是否加密；

ii. Non-resolvable Private Address

顾名思义，不可解析的 Private Address，通过定时更新方式让地址一直随机变来变去；

更新的周期称作 T_GAP(private_addr_int) ，建议值是 15 分钟；

ii. Resolvable Private Address

通过定时更新 + 加密的方式：一个随机数和一个称作 identity resolving key (IRK) 的密钥生成，因此只能被拥有相同 IRK 的设备扫描到，可以防止被未知设备扫描和追踪；更新周期建议是 15 分钟；

- BTMG_LE_PUBLIC_ADDRESS_ID

这类地址是抽象，用于识别设备的地址。如果设备仅支持 public address，则 public address 可以作为这个设备的 indentify address，在配对过程中使用；

- BTMG_LE_RANDOM_ADDRESS_ID

如果设备使用 Static Device Addres，可以直接作为 indentify address 使用；

如果设备使用 Resolvable Private Address，通过 IRK 解析之后的地址，才是 indentify address；

5.2 GATT Server

5.2.1 通用数据结构

5.2.1.1 adv_channel

广播信道

```
#define BTMG_LE_ADV_CHANNEL_NONE 0x00
#define BTMG_LE_ADV_CHANNEL_37 0x01
#define BTMG_LE_ADV_CHANNEL_38 (0x01 << 1)
#define BTMG_LE_ADV_CHANNEL_39 (0x01 << 2)
#define BTMG_LE_ADV_CHANNEL_ALL (BTMG_LE_ADV_CHANNEL_NONE | BTMG_LE_ADV_CHANNEL_37 |
                                 BTMG_LE_ADV_CHANNEL_38 | BTMG_LE_ADV_CHANNEL_39)
```

5.2.1.2 btmg_adv_data_t

广播数据

```
typedef struct {
    uint8_t data[31];
    uint8_t data_len;
} btmg_adv_data_t;
```

5.2.1.3 btmg_scan_rsp_data_t

扫描响应数据

```
typedef struct {
    uint8_t data[31];
    uint8_t data_len;
} btmg_scan_rsp_data_t;
```

5.2.1.4 btmg_le_advertising_type_t

广播类型

```
typedef enum {
    BTMG_LE_ADV_IND = 0x00,
    BTMG_LE_ADV_DIRECT_HIGH_IND = 0x01,
    BTMG_LE_ADV_SCAN_IND = 0x02,
    BTMG_LE_ADV_NONCONN_IND = 0x03,
    BTMG_LE_ADV_DIRECT_LOW_IND = 0x04,
    BTMG_LE_ADV_TYPE_MAX = 0x05,
} btmg_le_advertising_type_t;
```

- BTMG_LE_ADV_IND：可连接可扫描的非定向广播，通用广播指示；
- BTMG_LE_ADV_DIRECT_HIGH_IND：可连接的高占空比定向广播
- BTMG_LE_ADV_SCAN_IND：不可连接可扫描的非定向广播
- BTMG_LE_ADV_NONCONN_IND：可连接的低占空比定向广播

5.2.1.5 btmg_le_advertising_filter_policy_t

广播的过滤规则与白名单机制有关；

```
typedef enum {
    BTMG_LE_PROCESS_ALL_REQ = 0x00,
    BTMG_LE_PROCESS_CONN_REQ = 0x01,
    BTMG_LE_PROCESS_SCAN_REQ = 0x02,
    BTMG_LE_PROCESS_WHITE_LIST_REQ = 0x03,
    BTMG_LE_FILTER_POLICY_MAX = 0x04,
} btmg_le_advertising_filter_policy_t;
```

- BTMG_LE_PROCESS_ALL_REQ：禁用白名单机制，允许任何设备连接和扫描；
- BTMG_LE_PROCESS_CONN_REQ：允许任何设备连接，但只允许白名单中的设备扫描；
- BTMG_LE_PROCESS_SCAN_REQ：允许任何设备扫描，但只允许白名单中的设备连接；
- BTMG_LE_PROCESS_WHITE_LIST_REQ：只允许白名单中的设备扫描和连接；

5.2.1.6 btmg_le_peer_addr_type_t

对端 BLE 设备的地址类型

```
typedef enum {
    /*public device address(default) or public indentiy address*/
    BTMG_LE_PEER_PUBLIC_ADDRESS = 0x00,
    /*random device address(default) or random indentiy address*/
    BTMG_LE_PEER_RANDOM_ADDRESS = 0x01,
} btmg_le_peer_addr_type_t;
```

5.2.1.7 btmg_le_advertising_parameters_t

广播参数

```
typedef struct {
    uint16_t min_interval;
    uint16_t max_interval;
    btmg_le_advertising_type_t adv_type;
    btmg_le_addr_type_t own_addr_type;
    btmg_le_peer_addr_type_t peer_addr_type;
    char peer_addr[18];
    uint8_t chan_map;
    btmg_le_advertising_filter_policy_t filter;
} btmg_le_advertising_parameters_t;
```

- min_interval：最小广播间隔，广播间隔范围 20ms-10.24s，此处实际时间 =min_interval * 0.625 ms；
- max_interval：最大广播间隔，此处实际时间 =min_interval * 0.625 ms；
- adv_type：广播类型，详见通用数据结构章节；
- own_addr_type：本机设备类型，详见通用数据结构章节 btmg_le_advertising_type_t；
- peer_addr_type：对端设备类型，详见通用数据结构章节 btmg_le_addr_type_t；
- peer_addr：对端的设备类型；
- chan_map：广播信道；
- filter：广播过滤规则，详见通用数据结构章节 btmg_le_advertising_filter_policy_t；

5.2.1.8 gatt_char_properties_t

chracteristic property

```
typedef enum {
    BT_GATT_CHAR_PROPERTY_BROADCAST = 0x01,
    BT_GATT_CHAR_PROPERTY_READ = 0x02,
    BT_GATT_CHAR_PROPERTY_WRITE_NO_RESPONSE = 0x04,
    BT_GATT_CHAR_PROPERTY_WRITE = 0x08,
    BT_GATT_CHAR_PROPERTY_NOTIFY = 0x10,
    BT_GATT_CHAR_PROPERTY_INDICATE = 0x20,
    BT_GATT_CHAR_PROPERTY_AUTH_SIGNED_WRITE = 0x40
```

```
} gatt_char_properties_t;
```

- BT_GATT_CHAR_PROPERTY_BROADCAST: 可广播；
- BT_GATT_CHAR_PROPERTY_READ：可读；
- BT_GATT_CHAR_PROPERTY_WRITE_NO_RESPONSE：可写且无回复；
- BT_GATT_CHAR_PROPERTY_WRITE：可写；
- BT_GATT_CHAR_PROPERTY_NOTIFY：支持通知；
- BT_GATT_CHAR_PROPERTY_INDICATE；支持指示；
- BT_GATT_CHAR_PROPERTY_AUTH_SIGNED_WRITE：支持写签名；

5.2.1.9 gatt_desc_properties_t

Characteristic descriptor property

```
typedef enum {
    BT_GATT_DESC_PROPERTY_BROADCAST = 0x01,
    BT_GATT_DESC_PROPERTY_READ = 0x02,
    BT_GATT_DESC_PROPERTY_WRITE_NO_RESPONSE = 0x04,
    BT_GATT_DESC_PROPERTY_WRITE = 0x08,
    BT_GATT_DESC_PROPERTY_NOTIFY = 0x10,
    BT_GATT_DESC_PROPERTY_INDICATE = 0x20,
    BT_GATT_DESC_PROPERTY_AUTH_SIGNED_WRITE = 0x40
} gatt_desc_properties_t;
```

Characteristic descriptor 的 property 描述，请参考上文 chracteristic property；

5.2.1.10 gatt_permissions_t

GATT Attribute Permissions，与前面的 chracteristic property 和 Characteristic descriptor property 是两个不同概念；

Attribute Permissions 描述的是整个属性的权限，主要有四种类型：

- 访问权限（Access Permission）：包含只读、只写、读写；服务器端使用访问权限来确定客户端是否可以读取和/或写入属性值；
- 加密权限（Encryption Permission）：包含加密、不加密；
- 认证权限（Authentication Permission）：是否需要认证、无需认证；服务器使用认证权限来确定当客户端试图访问某个属性时是否需要经过身份验证的物理链接。在向客户端发送通知或指示之前，服务器还使用身份验证权限来确定是否需要经过身份验证的物理链接；
- 授权权限（Authorization Permission）：是否需要授权、无需授权；授权权限决定在访问属性值之前是否需要对客户端进行授权。

一般属性的权限可以是访问权限、加密权限、认证权限和授权权限的组合。

```
typedef enum {
    BT_GATT_PERM_READ = 0x01,
    BT_GATT_PERM_WRITE = 0x02,
    BT_GATT_PERM_READ_ENCYPT = 0x04,
    BT_GATT_PERM_WRITE_ENCRYPT = 0x08,
    BT_GATT_PERM_ENCRYPT = 0x04 | 0x08,
    BT_GATT_PERM_READ_AUTHEN = 0x10,
    BT_GATT_PERM_WRITE_AUTHEN = 0x20,
    BT_GATT_PERM_AUTHEN = 0x10 | 0x20,
    BT_GATT_PERM_AUTHOR = 0x40,
    BT_GATT_PERM_NONE = 0x80
} gatt_permissions_t;
```

- BT_GATT_PERM_READ：可读；
- BT_GATT_PERM_WRITE：可写；
- BT_GATT_PERM_READ_ENCYPT：写需要加密；
- BT_GATT_PERM_WRITE_ENCRYPT：写需要加密；
- BT_GATT_PERM_ENCRYPT：读写都需要加密；
- BT_GATT_PERM_READ_AUTHEN：读需要认证；
- BT_GATT_PERM_WRITE_AUTHEN：写需要认证；
- BT_GATT_PERM_AUTHEN：读写需要认证；
- BT_GATT_PERM_AUTHOR：访问需要授权；
- BT_GATT_PERM_NONE：没有设置权限；

5.2.1.11 gatt_attr_res_code_t

Response 错误码

```
typedef enum {
    BT_GATT_SUCCESS = 0x00,
    BT_GATT_ERROR_INVALID_HANDLE = 0x01, /*Invalid Handle*/
    BT_GATT_ERROR_READ_NOT_PERMITTED = 0x02, /*Read not Permitted*/
    BT_GATT_ERROR_WRITE_NOT_PERMITTED = 0x03, /*Writed Not Permitted*/
    BT_GATT_ERROR_INVALID_PDU = 0x04, /*Invalid PDU*/
    BT_GATT_ERROR_AUTHENTICATION = 0x05, /*Insufficient Authentication*/
    BT_GATT_ERROR_REQUEST_NOT_SUPPORTED = 0x06, /*Request Not Supported*/
    BT_GATT_ERROR_INVALID_OFFSET = 0x07, /*Invalid Offset*/
    BT_GATT_ERROR_AUTHORIZATION = 0x08, /*Insufficient Authorization*/
    BT_GATT_ERROR_PREPARE_QUEUE_FULL = 0x09, /*Prepare Queue Full*/
    BT_GATT_ERROR_ATTRIBUTE_NOT_FOUND = 0x0A, /*Attribute Not Found*/
    BT_GATT_ERROR_ATTRIBUTE_NOT_LONG = 0x0B, /*Attribute Not Long*/
    BT_GATT_ERROR_INSUFFICIENT_ENCRYPTION_KEY_SIZE = 0x0C, /*Insufficient Encryption Key
    Size*/
    BT_GATT_ERROR_INVALID_ATTRIBUTE_VALUE_LEN = 0x0D, /*Invalid Attribute Value Length*/
    BT_GATT_ERROR_UNLIKELY = 0x0E, /*Unlikely Error*/
    BT_GATT_ERROR_INSUFFICIENT_ENCRYPTION = 0x0F, /*Insufficient Encryption*/
    BT_GATT_ERROR_UNSUPPORTED_GROUP_TYPE = 0x10, /*Unsupported Group Type*/
    BT_GATT_ERROR_INSUFFICIENT_RESOURCES = 0x11, /*Insufficient Resources*/
    BT_GATT_ERROR_DB_OUT_OF_SYNC = 0x12, /*Database Out of Sync*/
    BT_GATT_ERROR_VALUE_NOT_ALLOWED = 0x13 /*Value Not Allowed*/
    /*0x80-0x9F Application Error*/
    /*0xE0-0xFF Common Profile and Service Error Codes*/
} gatt_attr_res_code_t;
```

5.2.2 回调函数

```
typedef struct {
    /*gatt add ... callback*/
    bt_gatts_add_service_cb gatts_add_svc_cb;
    bt_gatts_add_char_cb gatts_add_char_cb;
    bt_gatts_add_desc_cb gatts_add_desc_cb;
    /*gatt event callback*/
    bt_gatts_connection_event_cb gatts_connection_event_cb;
    /*gatt characteristic request callback*/
    bt_gatts_char_read_req_cb gatts_char_read_req_cb;
    bt_gatts_char_write_req_cb gatts_char_write_req_cb;
    bt_gatts_char_notify_req_cb gatts_char_notify_req_cb;
    /*gatt descriptor request callback*/
    bt_gatts_desc_read_req_cb gatts_desc_read_req_cb;
    bt_gatts_desc_write_req_cb gatts_desc_write_req_cb;
    bt_gatts_send_indication_cb gatts_send_indication_cb;
} btmg_gatt_server_cb_t;
```

- gatts_add_svc_cb：增加一个 Service 的回调函数；
- gatts_add_char_cb：增加一个 Characteristic 的回调函数；
- gatts_add_desc_cb: 增加一个 Characteristic descriptor 的回调函数；
- gatts_connection_event_cb：连接状态的回调函数；
- gatts_char_read_req_cb：Characteristic 读请求的回调函数；
- gatts_char_write_req_cb：Characteristi 写请求的回调函数；
- gatts_char_notify_req_cb：Characteristi 通知请求的回调函数；
- gatts_desc_read_req_cb：Characteristic descriptor 读请求的回调函数；
- gatts_desc_write_req_cb：Characteristic descriptor 写请求的回调函数；
- gatts_send_indication_cb：发送指示后收到确认的回调函数；

5.2.3 回调函数的参数

5.2.3.1 gatts_add_svc_msg_t

增加一个 service 回调函数参数

```
typedef struct {
    int handle_num;
    int svc_handle;
} gatts_add_svc_msg_t;
```

- handle_num：表示该 service 能包含多少个属性条目数量，一个 handle 是一个属性条目；
- svc_handle：表示添加完成的 Service 的起始 handle 值；

一般是在回调函数中使用到该结构体；

5.2.3.2 gatts_add_char_msg_t

增加一个 Characteristic 回调函数参数

```
typedef struct {
    char *uuid;
    int char_handle;
} gatts_add_char_msg_t;
```

- uuid：要添加的 Characteristic 的 uuid；
- char_handle: 表示添加完成的 Characteristic 的 handle 值；

一般是在回调函数中使用到该结构体；

5.2.3.3 gatts_add_desc_msg_t

增加一个 Characteristic descriptor 回调函数参数

```
typedef struct {
    int desc_handle;
} gatts_add_desc_msg_t;
```

- desc_handle: 表示添加完成的 Characteristic descriptor 的 handle 值；

一般是在回调函数中使用到该结构体；

5.2.3.4 gatts_connection_event_t

gatt server 连接事件

```
typedef enum {
    BT_GATT_CONNECTION,
    BT_GATT_DISCONNECT,
} gatts_connection_event_t;
```

一般是在回调函数中使用到该结构体；

5.2.3.5 gatts_char_read_req_t

client 端读请求回调函数参数

```
typedef struct {
    unsigned int trans_id;
    int attr_handle;
```

```
    int offset;
    bool is_blob_req;
} gatts_char_read_req_t;
```

- trans_id：协议栈内部传上来，用于记录 requset 的 id，用于正确 response；
- attr_handle：属性 handle 值；
- offset：大数据读取的偏移
- is_blob_req：是否大数据读取，client 端对一次大数据读取可以分多次完成

5.2.3.6 gatt_char_write_req_t

client 写请求回调函数参数

```
typedef struct {
    unsigned int trans_id;
    int attr_handle;
    int offset;
    char value[GATT_MAX_ATTR_LEN];
    int value_len;
    bool need_rsp;
} gatt_char_write_req_t;
```

- trans_id：协议栈内部传上来，用于记录 requset 的 id，用于正确 response；
- attr_handle：属性 handle 值；
- offset：数据偏移；
- value[GATT_MAX_ATTR_LEN]：client 端写入的数据内容，其中 GATT_MAX_ATTR_LEN 定义为 600；
- value_len：client 端写入的数据长度；
- need_rsp：是否需要回复，如果 client 端是 write req 需要回复，如果 write cmd 不需要回复；

5.2.3.7 gatts_desc_read_req_t

Characteristic descriptor 读请求回调函数参数

```
typedef struct {
    unsigned int trans_id;
    int attr_handle;
    int offset;
    bool is_blob_req;
} gatts_desc_read_req_t;
```

- trans_id：协议栈内部传上来，用于记录 requset 的 id，用于正确 response；
- attr_handle：属性 handle 值；
- offset：数据偏移；
- is_blob_req：是否大数据读取，client 端对一次大数据读取可以分多次完成；

5.2.3.8 gatts_desc_write_req_t

Characteristic descriptor 写请求回调函数参数

```
typedef struct {
    unsigned int trans_id;
    int attr_handle;
    int offset;
    char value[GATT_MAX_ATTR_LEN];
    int value_len;
    bool need_rsp;
} gatts_desc_write_req_t;
```

- trans_id：协议栈内部传上来，用于记录 requset 的 id，用于正确 response；
- attr_handle：属性 handle 值；
- offset：数据偏移；
- value[GATT_MAX_ATTR_LEN]：client 端写入的数据内容，其中 GATT_MAX_ATTR_LEN 定义为 600；
- value_len：client 端写入的数据长度；
- need_rsp：是否需要回复，如果 client 端是 write req 需要回复，如果 write cmd 不需要回复；

5.2.3.9 gatts_send_indication_t

发送指示后收到确认的回调函数参数

```
typedef struct {
} gatts_send_indication_t;
```

目前该参数为空；

5.2.4 服务注册相关结构体

5.2.4.1 gatts_add_svc_t

增加一个 service 函数（bt_manager_gatt_server_create_service）的参数类型

```
typedef struct {
    char *uuid;      /*128-bit service UUID*/
    bool primary;    /* If true, this GATT service is a primary service */
    int number;
} gatts_add_svc_t;
```

- uuid: service 的 UUID；
- primary：是否是 primary service；
- number：表示该 service 能包含多少个属性条目数量，一个 handle 是一个属性条目；

5.2.4.2 gatts_add_char_t

增加一个 characteristic 函数（bt_manager_gatt_server_add_characteristic）的参数类型

```
typedef struct {
    char *uuid;
    int properties;
    int permissions;
    int svc_handle;
} gatts_add_char_t;
```

- uuid: Characteristic 的 UUID；
- properties：Characteristic 的属性；
- permissions：Characteristic 的权限；
- svc_handle：Characteristic 添加目标 Service 的 handle；

5.2.4.3 gatts_add_desc_t

增加一个 Characteristic descriptor 函数（bt_manager_gatt_server_add_descriptor）的参数类型

```
typedef struct {
    char *uuid;
    int properties;
    int permissions;
    int svc_handle;
} gatts_add_desc_t;
```

- uuid: Characteristic descriptor 的 UUID；
- properties：Characteristic descriptor 的属性；
- permissions：Characteristic descriptor 的权限；
- svc_handle：Characteristic descriptor 添加目标 Service 的 handle，注意不是 Characteristic 的 handle；

一般 descriptor 添加在哪个 Characteristic 后面就是对哪个 Characteristic 描述；

5.2.4.4 gatts_star_svc_t

启动一个 service 函数（bt_manager_gatt_server_start_service）的参数类型

```
typedef struct {
    int svc_handle;
} gatts_star_svc_t;
```

- svc_handle：需要启动 service 的 handle；

5.2.4.5 gatts_stop_svc_t

停止一个 service 函数（bt_manager_gatt_server_stop_service）的参数类型

```
typedef struct {
    int svc_handle;
} gatts_stop_svc_t;
```

- svc_handle：需要停止 service 的 handle；

5.2.4.6 gatts_remove_svc_t

删除一个 service 函数（bt_manager_gatt_server_remove_service）的参数类型

```
typedef struct {
    int svc_handle;
} gatts_remove_svc_t;
```

- svc_handle：需要删除 service 的 handle；

5.2.4.7 gatts_send_read_rsp_t

service 回复 client 读操作函数（bt_manager_gatt_server_send_read_response）的参数类型

```
typedef struct {
    unsigned int trans_id;
    int status;
    int attr_handle;
    char *value;
    int value_len;
    int auth_req;
} gatts_send_read_rsp_t;
```

- trans_id：协议栈内部传上来，用于记录 requset 的 id，用于正确 response；
- status：回复的状态，暂时没有用到；
- attr_handle：client 端要读的目标属性 handle；
- value：回复数据的指针；
- value_len：回复的数据长度；
- auth_req：认证请求；

5.2.4.8 gatts_write_rsp_t

service 回复 client 写操作函数（bt_manager_gatt_server_send_write_response）的参数类型

```
typedef struct {
    unsigned int trans_id;
    int attr_handle;
    gatt_attr_res_code_t state;
} gatts_write_rsp_t;
```

- trans_id：协议栈内部传上来，用于记录 requset 的 id，用于正确 response；
- attr_handle：client 端要读的目标属性 handle；
- state：回复错误码；

5.2.4.9 gatts_notify_data_t

service 通知 client 函数（bt_manager_gatt_server_send_notify）的参数类型

```
typedef struct {
    int attr_handle;
    char *value;
    int value_len;
} gatts_notify_data_t;
```

- attr_handle：client 端要读的目标属性 handle；
- value：回复数据的指针；
- value_len：回复的数据长度；

5.2.4.10 gatts_indication_data_t

service 指示 client 函数（bt_manager_gatt_server_send_indication）的参数类型

```
typedef struct {
    int attr_handle;
    char *value;
    int value_len;
} gatts_indication_data_t;
```

- attr_handle：client 端要读的目标属性 handle；
- value：回复数据的指针；
- value_len：回复的数据长度；

5.2.5 GATT Server API

5.2.5.1 GATT Server 初始化

函数原型	int bt_manager_gatt_server_init(void)
参数说明	无；
返回说明	int 0: 成功；非 0: 失败；
功能描述	gatt server 初始化；

5.2.5.2 GATT Server 反初始化

函数原型	int bt_manager_gatt_server_deinit(void)
参数说明	无；
返回说明	int 0: 成功；非 0: 失败；
功能描述	gatt server 反初始化；

5.2.5.3 设置随机地址

函数原型	int bt_manager_le_set_random_address(void)
参数说明	无；
返回说明	int 0: 成功；非 0: 失败；
功能描述	BLE 设备设置使用随机地址；

5.2.5.4 设置广播的参数

函数原型	int bt_manager_le_set_adv_param(btmg_le_advertising_parameters_t *adv_param)
参数说明	见通用数据结构章节 btmg_le_advertising_parameters_t；
返回说明	int 0: 成功；非 0: 失败；
功能描述	设置广播的参数，如广播的间隔、广播的信道，广播的类型等，详细查看参数说明；

5.2.5.5 设置广播的数据

函数原型	int bt_manager_le_set_adv_data(btmg_adv_data_t *adv_data)
参数说明	见通用数据结构章节 btmg_adv_data_t ；
返回说明	int 0: 成功；非 0: 失败；
功能描述	设置广播携带的数据内容，如 BLE 设备名称、UUID 等；

5.2.5.6 使能广播

函数原型	int bt_manager_le_enable_adv(bool enable)
参数说明	enable：true 打开广播；false: 关闭广播；
返回说明	int 0: 成功；非 0: 失败；
功能描述	打开或者关闭广播；打开广播之前先设置广播的参数和内容；

5.2.5.7 设置扫描响应数据

广播包含扫描请求 SCAN_REQ 和扫描响应 SCAN_RSP：

- 扫描请求：由链路层处于扫描态的设备发送，链路层处于广播态的设备接收；
- 扫描响应：由链路层处于广播态的设备发送，链路层处于扫描态的设备接收；

处于扫描态的设备可以接收广播信道的报文，通过扫描可以侦听哪些设备正在广播;

扫描分为主动扫描和被动扫描：

- 主动扫描在接收到广播报文后会发送扫描请求来获取额外的数据;
- 被动扫描仅仅接收广播报文，不会发送扫描请求;

函数原型	int bt_manager_gatt_set_adv_data(gatt_adv_data_t *adv_data)
参数说明	见通用数据结构章节 btmg_adv_data_t ；
返回说明	int 0: 成功；非 0: 失败；
功能描述	设置扫描响应数据；

5.2.5.8 断开所有 BLE 连接

函数原型	int bt_manager_le_disconnect(void)
参数说明	无；
返回说明	int 0: 成功；非 0: 失败；
功能描述	断开当前所有的 BLE 连接；

5.2.5.9 创建服务

函数原型	int bt_manager_gatt_server_create_service(gatt_add_svc_t *svc)
参数说明	见服务注册相关结构体章节 gatt_add_svc_t ；
返回说明	int 0: 成功；非 0: 失败；
功能描述	创建一个 service；

5.2.5.10 添加 characteristic

函数原型	int bt_manager_gatt_server_add_characteristic(gatt_add_char_t *chr)
参数说明	见服务注册相关结构体章节 gatt_add_char_t ；
返回说明	int 0: 成功；非 0: 失败；
功能描述	指定服务中添加 characteristic;

5.2.5.11 添加 descriptor

函数原型	int bt_manager_gatt_server_add_descriptor(gatt_add_desc_t *desc)
参数说明	见服务注册相关结构体章节 gatt_add_desc_t ；
返回说明	int 0: 成功；非 0: 失败；
功能描述	指定服务中添加 descriptor；

5.2.5.12 启动服务

函数原型	int bt_manager_gatt_server_start_service(gatt_star_svc_t *start_svc)
参数说明	见服务注册相关结构体章节 gatt_star_svc_t ；
返回说明	int 0: 成功；非 0: 失败；

函数原型	int bt_manager_gatt_server_start_service(gatt_star_svc_t *start_svc)
功能描述	启动指定的 service；创建 service 后不需要调用该 API，一般用于启动已经停止的 service；

5.2.5.13 停止服务

函数原型	int bt_manager_gatt_server_stop_service(gatt_stop_svc_t *stop_svc)
参数说明	见服务注册相关结构体章节 gatt_stop_svc_t；
返回说明	int 0: 成功；非 0: 失败；
功能描述	停止指定的 service；

5.2.5.14 删除服务

函数原型	int bt_manager_gatt_server_remove_service(gatts_remove_svc_t *remove_svc);
参数说明	见服务注册相关结构体章节 gatts_remove_svc_t ；
返回说明	int 0: 成功；非 0: 失败；
功能描述	删除一个 gatt service，删除后如果还要使用，需要重新创建；

5.2.5.15 回复 client 读请求

函数原型	int bt_manager_gatt_server_send_read_response(gatt_send_read_rsp_t *pData)
参数说明	见服务注册相关结构体章节 gatt_send_read_rsp_t ；
返回说明	int 0: 成功；非 0: 失败；
功能描述	Client 端读取 server 属性的时候，会激活 server 端对应的回调函数，server 通过该函数回复读请
求，以通知 cl	ient 读是否成功；

5.2.5.16 回复 client 写请求

函数原型	int bt_manager_gatt_server_send_write_response(gatt_write_rsp_t *pData)
参数说明	见服务注册相关结构体章节 gatt_write_rsp_t ；
返回说明	int 0: 成功；非 0: 失败；
功能描述	Client 端写 server 属性的时候，会激活 server 端对应的回调函数，server 通过该函数回复写请
求，以通知	ient 写是否成功；
cl	

5.2.5.17 通知 client

函数原型	int bt_manager_gatt_server_send_notify(gatts_notify_data_t *pData)
参数说明	见服务注册相关结构体章节 gatts_notify_data_t；
返回说明	int 0: 成功；非 0: 失败；
功能描述	Server 通过该函数通知 client，client 不需要回复；

5.2.5.18 指示 client

函数原型	int bt_manager_gatt_server_send_indication(gatts_indication_data_t *pData)
参数说明	见服务注册相关结构体章节 gatts_indication_data_t ；
返回说明	int 0: 成功；非 0: 失败；
功能描述	Server 通过该函数指示 client，client 需要回复；

5.2.5.19 获取当前 mtu

函数原型	int bt_manager_gatt_server_get_mtu(void)
参数说明	无；
返回说明	int 0: 成功；非 0: 失败；
功能描述	获取当前 mtu；

5.2.6 总结 GATT Server 的调用流程

gatt server API 的使用流程：(可参考 bt_test.c::_bt_init)

1. 调用 bt_manager_preinit，预初始化，运行期间只需要调用一次。
2. 调用 bt_manager_enable_profile(BTMG_GATT_SERVER_ENABLE)，设置本次要使用的 profile。
3. 填充 btmg_callback_t 结构体，即回调函数。
4. 调用 bt_manager_init，蓝牙初始化。
5. 调用 bt_manager_enable，打开蓝牙协议栈。
6. 调用 add_cmd_table，如果使用 cmd 命令需添加 gatt server 的命令列表。
7. 调用 bt_manager_gatt_server_init，初始化 gatt server，其功能主要是建立 L2CAP socket。
8. 构建一个 server，主要包括创建一个 service（bt_manager_gatt_server_create_service），填充

service 中的内容特性内容和描述信息。可参考 bt_gatt_server_register_bt_test_svc。

9. 调用 bt_manager_le_set_adv_param，设置广播参数。可参考 set_adv_param。
10. 调用 bt_manager_le_set_adv_data，设置广播数据。可参考 le_set_adv_data。
11. 调用 bt_manager_le_enable_adv，使能广播。

5.3 GATT Client

5.3.1 通用数据结构

5.3.1.1 btmg_le_scan_param_t

BLE 扫描参数

```
typedef struct {
    btmg_le_scan_type_t scan_type;
    uint16_t scan_interval;
    uint16_t scan_window;
    btmg_le_scan_filter_duplicate_t filter_duplicate;
    btmg_le_addr_type_t own_addr_type;
    btmg_le_scan_filter_policy_t filter_policy;
    uint16_t timeout; // Scan timeout between 0x0001 and 0xFFFF in seconds, 0x0000 disables
     timeout.
} btmg_le_scan_param_t;
```

- scan_interval：扫描间隔，范围是 2.5ms~10.24s，此处实际时间 =scan_interval * 0.625；
- scan_window：扫描窗口，范围是 2.5ms~10.24s，此处实际时间 =scan_interval * 0.625；
- filter_duplicate：重复过滤配置，详见 btmg_le_scan_filter_duplicate_t；
- own_addr_type：本地设备地址类型；
- filter_policy：扫描过滤策略，详见 btmg_le_addr_type_t 说明；
- timeout：扫描超时时间，保留，当前无效；

扫描窗口必须要小于或者等于扫描间隔，如果扫描窗口等于扫描间隔，说明一直进行扫描；

5.3.1.2 btmg_le_scan_type_t

BLE 扫描类型

```
typedef enum {
    LE_SCAN_TYPE_PASSIVE = 0x00, // passive scanning
    LE_SCAN_TYPE_ACTIVE, // active scanning
} btmg_le_scan_type_t;
```

- LE_SCAN_TYPE_PASSIVE：被动扫描；
- LE_SCAN_TYPE_ACTIVE：主动扫描；

5.3.1.3 btmg_le_scan_filter_policy_t

BLE 扫描过滤策略

```
typedef enum {
    LE_SCAN_FILTER_ALLOW_ALL = 0x00,
    LE_SCAN_FILTER_ALLOW_ONLY_WLST,
    LE_SCAN_FILTER_ALLOW_UND_RPA_DIR,
    LE_SCAN_FILTER_ALLOW_WLIST_RPA_DIR,
} btmg_le_scan_filter_policy_t;
```

- LE_SCAN_FILTER_ALLOW_ALL:
- 禁用白名单机制，处理所有的广播包 (除了那些不是发给扫描者的可连接定向广播数据包) ；
- LE_SCAN_FILTER_ALLOW_ONLY_WLST:
- 处理在白名单中的那些设备发送的广播包；
- 处理那些发给扫描者的可连接定向广播数据包；
- LE_SCAN_FILTER_ALLOW_UND_RPA_DI
- 处理所有非定向广播数据包；

- 处理指向当前扫描者的可连接定向广播数据包；
- 处理广播者的地址为可解析私有地址的可连接定向广播数据包；
- LE_SCAN_FILTER_ALLOW_WLIST_RPA_DIR: 接受如下的广播包：

（1）在白名单中的设备发送的广播包；

（2）广播者地址为 resolvable private address 的 directed advertising packets；

（3）给本设备发送的 directed advertising packets；

5.3.1.4 btmg_le_scan_filter_duplicate_t

是否开启重复过滤

```
typedef enum {
    LE_SCAN_DUPLICATE_DISABLE = 0x0, // will not filter
    LE_SCAN_DUPLICATE_ENABLE = 0x1, // filter duplicate packets
} btmg_le_scan_filter_duplicate_t;
```

- LE_SCAN_DUPLICATE_DISABLE：禁用重复过滤；
- LE_SCAN_DUPLICATE_ENABLE：打开重复过滤；

5.3.1.5 btmg_le_conn_param_t

连接参数

```
typedef struct {
    uint16_t min_conn_interval;
    uint16_t max_conn_interval;
    uint16_t slave_latency;
    uint16_t conn_sup_timeout;
} btmg_le_conn_param_t;
```

- min_conn_interval：最小连接间隔，取值范围 0x0006 to 0x0C80,Time = N * 1.25 msec，因此时间范围为 7.5ms~4s;
- max_conn_interval：最大连接间隔，取值范围 0x0006 to 0x0C80,Time = N * 1.25 msec，因此时间范围为 7.5ms~4s;
- slave_latency：从设备延迟，取值范围：0x0000 ~ 0x01F3；
- conn_sup_timeout：连接监控超时，单位是 ms；取值范围：0x000A ~0x0C80, Time = N * 10 ms, Time Range: 100 ms ~ 32 s；

5.3.1.6 btmg_adv_data_type_t

对端设备广播数据类型

```
typedef enum {
    LE_ADV_DATA,
    LE_SCAN_RSP_DATA,
} btmg_adv_data_type_t;
```

- LE_ADV_DATA：广播数据；
- LE_SCAN_RSP_DATA：扫描响应数据；

5.3.1.7 btmg_scan_rsp_data_t

扫描响应的数据内容

```
typedef struct {
    uint8_t data[31];
    uint8_t data_len;
} btmg_scan_rsp_data_t;
```

5.3.1.8 btmg_le_scan_report_t

ble 扫描结果

```
typedef struct {
    uint8_t peer_addr[6];
    btmg_le_addr_type_t addr_type;
    btmg_adv_data_type_t adv_type;
    int8_t rssi;
    btmg_scan_rsp_data_t report;
} btmg_le_scan_report_t;
```

- peer_addr：扫描到的设备 mac 地址；
- addr_type：地址类型；
- adv_type：广播类型，详见 btmg_adv_data_type_t 说明；
- rssi：信号强度大小；
- report：扫描响应数据，详见 btmg_scan_rsp_data_t 说明；

5.3.1.9 btmg_security_level_t

BLE 的 GAP 绑定安全级别

```
typedef enum {
    BTMG_SECURITY_LOW = 1,
```

```
    BTMG_SECURITY_MEDIUM = 2,
    BTMG_SECURITY_HIGH = 3,
    BTMG_SECURITY_FIPS = 4,
} btmg_security_level_t;
```

- BTMG_SECURITY_LOW：AuthReq 关闭 MITM 和 Bonding_flags；
- BTMG_SECURITY_MEDIUM：AuthReq 只打开 Bonding_flags，关闭 MITM，但若 io_capability 不等于 HCI_IO_NO_INPUT_OUTPUT，内核会继续打开 MITM；
- BTMG_SECURITY_HIGH: AuthReq 打开 MITM 和 Bonding_flags；
- BTMG_SECURITY_FIPS: AuthReq 打开 MITM 和 Bonding_flags；

5.3.2 回调函数

```
typedef struct {
    bt_gattc_connect_cb gattc_conn_cb;
    bt_gattc_disconnect_cb gattc_disconn_cb;
    bt_gattc_read_cb gattc_read_cb;
    bt_gattc_write_cb gattc_write_cb;
    bt_gattc_write_long_cb gattc_write_long_cb;
    bt_gattc_notify_cb gattc_notify_cb;
    bt_gattc_service_changed_cb gattc_service_changed_cb;
    bt_gattc_dis_service_cb gattc_dis_service_cb;
    bt_gattc_dis_char_cb gattc_dis_char_cb;
    bt_gattc_dis_desc_cb gattc_dis_desc_cb;
    bt_gattc_connected_list_cb gattc_connected_list_cb;
} btmg_gatt_client_cb_t;
```

- gattc_conn_cb: 连接回调；
- gattc_disconn_cb：断开回调；
- gattc_read_cb：读请求的回调；
- gattc_write_cb：写请求的回调；
- gattc_write_long_cb：写长数据请求的回调；
- gattc_notify_cb：收到 Server 端的 notify 回调函数；
- gattc_service_changed_cb：Server 端的 Service 发生了改变，Client 端收到通知的回调函数；
- gattc_dis_service_cb：Client 端查询 Server 端 Service 的回调函数；
- gattc_dis_char_cb：Client 端查询 Server 端 Characteristic 的回调函数；
- gattc_dis_desc_cb：Client 端查询 Server 端 Descriptor 的回调函数；
- gattc_connected_list_cb：获取当前连接设备列表的回调函数；

5.3.3 回调函数的参数

5.3.3.1 gattc_notify_indicate_cb_para_t

收到 Server 端 notify 或 indicate 的回调函数的参数

```
typedef struct {
    uint16_t value_handle;
    const uint8_t *value;
    uint16_t length;
    void *user_data;
} gattc_notify_indicate_cb_para_t;
```

- value_handle：Sever 端 Characteristic Value 的 handle；
- value：Sever 端发送过来的数据内容；
- length：数据长度；
- user_data：用户数据指针；

5.3.3.2 gattc_write_cb_para_t

写请求回调函数的参数

```
typedef struct {
    bool success;
    uint8_t att_ecode;
    void *user_data;
} gattc_write_cb_para_t;
```

- success：是否写成功；
- att_ecode：如果写失败了，返回的错误码
- user_data：用户数据指针；

5.3.3.3 gattc_write_long_cb_para_t

写长数据请求回调函数的参数

```
typedef struct {
    bool success;
    bool reliable_error;
    uint8_t att_ecode;
    void *user_data;
} gattc_write_long_cb_para_t;
```

- success：是否写成功；
- reliable_error：可靠写入是否验证；

- att_ecode：如果写失败了，返回的错误码
- user_data：用户数据指针；

5.3.3.4 gattc_read_cb_para_t

读请求回调函数的参数

```
typedef struct {
    bool success;
    uint8_t att_ecode;
    const uint8_t *value;
    uint16_t length;
    void *user_data;
} gattc_read_cb_para_t;
```

- success：是否写成功；
- att_ecode：如果写失败了，返回的错误码
- const uint8_t *value：读到的数据；
- length: 数据长度；
- user_data：用户数据指针；

5.3.3.5 gattc_conn_cb_para_t

连接回调函数的参数

```
typedef struct {
    int conn_id;
    bool success;
    uint8_t att_ecode;
    void *user_data;
} gattc_conn_cb_para_t;
```

- conn_id：返回的连接标识符，每个连接都有专门的标识符；
- success：是否连接成功；
- att_ecode：如果连接失败了，返回的错误码；
- user_data：用户数据指针；

5.3.3.6 gattc_connected_list_cb_para_t

获取连接列表回调函数的参数

```
typedef struct {
    int conn_id;
    uint8_t *addr;
    int addr_type;
```

```
} gattc_connected_list_cb_para_t;
```

- conn_id：连接标识符；
- addr：已连接设备的 mac 地址；
- addr_type：mac 地址类型；

5.3.3.7 gattc_disconnect_reason_t

断开连接原因

```
typedef enum {
    UNKNOWN_OTHER_ERROR = 0,
    CONNECTION_TIMEOUT,
    REMOTE_USER_TERMINATED,
    LOCAL_HOST_TERMINATED,
} gattc_disconnect_reason_t;
```

- UNKNOWN_OTHER_ERROR：未知错误；
- CONNECTION_TIMEOUT：连接超时；
- REMOTE_USER_TERMINATED：对端设备断开；
- LOCAL_HOST_TERMINATED：本地主动断开；

5.3.3.8 gattc_disconn_cb_para_t

断开连接回调函数的参数

```
typedef struct {
    uint16_t conn_handle;
    gattc_disconnect_reason_t reason;
} gattc_disconn_cb_para_t;
```

- conn_handle：暂时没有用到；
- reason：断开原因；

5.3.3.9 gattc_service_changed_cb_para_t

Server 端服务改变回调函数的参数

```
typedef struct {
    uint16_t start_handle;
    uint16_t end_handle;
    void *user_data;
} gattc_service_changed_cb_para_t;
```

- start_handle：更新后的起始 handle；
- end_handle：更新后的结束 handle；
- user_data：用户数据指针；

5.3.3.10 gattc_dis_service_cb_para_t

发现 Service 回调函数的参数

```
typedef struct {
    int conn_id;
    uint16_t start_handle;
    uint16_t end_handle;
    bool primary;
    btmg_uuid_t uuid;
    void *attr;
} gattc_dis_service_cb_para_t;
```

- conn_id：连接标识符；
- start_handle：Service 起始 handle；
- end_handle：Service 结束 handle；
- bool primary: 是否为 primary 服务；
- uuid：Service 的 UUID；
- void *attr：指向 Service 内容的指针，使用 bt_manager_gatt_client_discover_service_all_char() 通过该指针可以发现 Characteristic；

5.3.3.11 gattc_dis_char_cb_para_t

发现 Characteristic 回调函数的参数

```
typedef struct {
    int conn_id;
    uint16_t start_handle;
    uint16_t value_handle;
    uint8_t properties;
    uint16_t ext_prop;
    btmg_uuid_t uuid;
    void *attr;
} gattc_dis_char_cb_para_t;
```

- conn_id：连接标识符；
- start_handle：Characteristic 起始 handle；
- value_handle：Characteristic Value 的 handle；
- properties: Characteristic Value 的属性；
- ext_prop：Characteristic Value 的拓展属性；
- uuid: Characteristic 的 UUID；

- void *attr：指向 Service 内容的指针，使用 bt_manager_gatt_client_discover_char_all_descriptor() 通过该指针可以发现 Characteristic Descriptor；

5.3.3.12 gattc_dis_desc_cb_para_t

发现 Characteristic Descriptor 回调函数的参数

```
typedef struct {
    int conn_id;
    uint16_t handle;
    btmg_uuid_t uuid;
} gattc_dis_desc_cb_para_t;
```

- conn_id：连接标识符；
- handle：handle 值；
- uuid：Characteristic Descriptor 的 UUID；

5.3.4 GATT Client API

5.3.4.1 GATT Client 初始化

函数原型	int bt_manager_gatt_client_init(void)
参数说明	无；
返回说明	int 0: 初始化成功；非 0: 初始化失败；
功能描述	初始化 GATT Client 功能；

5.3.4.2 GATT Client 反初始化

函数原型	int bt_manager_gatt_client_deinit(void)
参数说明	无；
返回说明	int 0: 反初始化成功；非 0: 反初始化失败；
功能描述	反初始化 GATT Client 功能；

5.3.4.3 转换错误信息

函数原型	const char *bt_manager_gatt_client_ecode_to_string(uint8_t ecode)
参数说明	无；
返回说明	错误信息字符串；
功能描述	通过错误码获取相应的错误提示信息；

5.3.4.4 UUID 转换为 128 位

函数原型	void bt_manager_uuid_to_uuid128(const btmg_uuid_t *src, btmg_uuid_t *dst)
参数说明	- btmg_uuid_t *src：需要转换的 16 位 UUID; - btmg_uuid_t *dst：转换完成的 128 位 UUID；
返回说明	无；
功能描述	把 16 位的 UUID 转换为 128 位的 UUID；

5.3.4.5 UUID 转换为字符串

函数原型	int bt_manager_uuid_to_string(const btmg_uuid_t *uuid, char *str, size_t n)
参数说明	- btmg_uuid_t *src：需要转换的 UUID; - char *str：存放目标字符串的地址； - n: 存放目标字符串的空间大小；
返回说明	无；
功能描述	把 UUID 转换为字符串；

5.3.4.6 启动 BLE 扫描

函数原型	int bt_manager_le_scan_start(btmg_le_scan_type_t scan_type, btmg_le_scan_param_t *scan_param)
参数说明	- btmg_le_scan_type_t scan_type：扫描的类型，被动扫描或者主动扫描； - btmg_le_scan_param_t *scan_param：扫描参数，详见通用数据结构 btmg_le_scan_param_t ;
返回说明	int 0: 启动扫描成功；非 0: 启动扫描失败；

函数原型	int bt_manager_le_scan_start(btmg_le_scan_type_t scan_type, btmg_le_scan_param_t *scan_param)
功能描述	开始扫描 BLE 设备，扫描结果通过 gap_le_scan_report_cb 回调函数返回；

5.3.4.7 停止 BLE 扫描

函数原型	int bt_manager_le_scan_stop(void)
参数说明	无；
返回说明	int 0：停止扫描成功；非 0：停止扫描失败；
功能描述	停止扫描 BLE 设备；

5.3.4.8 设置扫描参数

函数原型	int bt_manager_le_set_scan_parameters(btmg_le_scan_type_t scan_type, btmg_le_scan_param_t *scan_param, btmg_le_addr_type_t own_type, uint8_t filter_policy)
参数说明	- btmg_le_scan_type_t scan_type：扫描的类型，被动扫描或者主动扫描； - btmg_le_scan_param_t *scan_param：扫描参数，详见通用数据结构 btmg_le_scan_param_t ； - own_type：本地 BLE 设备地址类型； - filter_policy：扫描过滤策略；
返回说明	int 0：停止扫描成功；非 0：停止扫描失败；
功能描述	停止扫描 BLE 设备；

5.3.4.9 更新连接参数

函数原型	int bt_manager_le_update_conn_params(btmg_le_conn_param_t *conn_params)
参数说明	btmg_le_conn_param_t *conn_params：连接参数，详见通用数据结构 btmg_le_conn_param_t ；
返回说明	int 0：更新成功；非 0：更新失败；
功能描述	更新 BLE 连接参数；

5.3.4.10 连接 Server 设备

函数原型	int bt_manager_gatt_client_connect(uint8_t *addr, btmg_le_addr_type_t dst_type, uint16_t mtu, btmg_security_level_t sec)
参数说明	- addr：GATT Server 设备 mac 地址； - btmg_le_addr_type_t dst_type：地址类型，public 或者 random； - mtu：期望设置的 mtu 大小，但是该指并不一定是最终 mtu 的值，因为连接后，client 和 Server 会协商 mtu，选取小的作为最终 mut； - btmg_security_level_t sec：GAP 绑定安全级别，详见通用数据结构 btmg_security_level_t ；
返回说明	int 0: 成功；非 0: 失败；
功能描述	连接 GATT Server 设备，设备最终连接状态请通过回调函数 gattc_conn_cb 判断，包含连接标识符；

5.3.4.11 断开 Server 设备

函数原型	int bt_manager_gatt_client_disconnect(uint8_t *addr)
参数说明	addr：GATT Server 设备 mac 地址；
返回说明	int 0: 成功；非 0: 失败；
功能描述	断开与 GATT Server 设备的连接，设备最终连接状态请通过回调函数 gattc_conn_cb 判断；

5.3.4.12 获取已连接设备列表

函数原型	int bt_manager_gatt_client_get_conn_list(void)
参数说明	无；
返回说明	int 0: 成功；非 0: 失败；
功能描述	获取已连接设备的列表，结果通过 gattc_connected_list_cb 返回，目前是每个设备会触发一次回调；

5.3.4.13 获取 MTU

函数原型	int bt_manager_gatt_client_get_mtu(int conn_id)
参数说明	conn_id：连接标识符，一连多时，不同的连接有不同的 ID 值；
返回说明	获取的 MTU 的值;
功能描述	获取当前连接协商后使用的 MTU 大小；

5.3.4.14 设置安全级别

函数原型	int bt_manager_gatt_client_set_security(int conn_id, btmg_security_level_t sec_level)
参数说明	- conn_id：连接标识符，一连多时，不同的连接有不同的 ID 值； - btmg_security_level_t：详见通用数据结构 btmg_security_level_t ;
返回说明	int 0: 成功；非 0: 失败；
功能描述	设置 GATT 安全级别；

5.3.4.15 获取安全级别

函数原型	btmg_security_level_t bt_manager_gatt_client_get_security(int conn_id)
参数说明	conn_id：连接标识符，一连多时，不同的连接有不同的 ID 值；
返回说明	返回安全等级；
功能描述	获取 GATT 安全级别；

5.3.4.16 注册 notify 或 indicate

函数原型	int bt_manager_gatt_client_register_notify(int conn_id, int char_handle)
参数说明	- conn_id：连接标识符，一连多时，不同的连接有不同的 ID 值； - int char_handle：Server 端包含 “notify“或者”indicate” 属性的 Characteristic 的 handle；
返回说明	注册的 id 号，用于注销使用；
功能描述	- 注册 notify 或 indicate - 协议栈底层已经包含写 CCC 的操作，因此无需再写 “Client Characteristic Configuration”；

函数原型	int bt_manager_gatt_client_register_notify(int conn_id, int char_handle)
	- 注册完成后，Server 端才可以对 Client 通知或者指示；

5.3.4.17 注销 notify 或 indicate

函数原型	int bt_manager_gatt_client_unregister_notify_indicate(int conn_id, int id)
参数说明	
	- conn_id：连接标识符，一连多时，不同的连接有不同的 ID 值； - id：注册时返回的 id；
返回说明	int 0：成功；非 0：失败；
功能描述	注销 notify 或 indicate；

5.3.4.18 写数据请求

函数原型	int bt_manager_gatt_client_write_request(int conn_id, int char_handle, uint8_t *value, uint16_t len)
参数说明	
	- conn_id：连接标识符，一连多时，不同的连接有不同的 ID 值； - char_handle：Server 端包含可写属性的 Characteristic value 的 handle； - value：数据指针； - len：数据长度；
返回说明	int 0：调用成功；非 0：调用失败；
功能描述	
	- 往 GATT Server 设备写数据，Server 需要 response； - 最终的写入结果需要通过 gattc_write_cb 回调函数获取；

5.3.4.19 写长数据请求

函数原型	int bt_manager_gatt_client_write_long_request(int conn_id, bool reliable_writes, int char_handle, int offset, uint8_t *value, uint16_t len)
参数说明	
	- conn_id：连接标识符，一连多时，不同的连接有不同的 ID 值；

函数原型	int bt_manager_gatt_client_write_long_request(int conn_id, bool reliable_writes, int char_handle, int offset, uint8_t *value, uint16_t len)
	- bool reliable_writes：是否为可靠写； - char_handle：Server 端包含可写属性的 Characteristic value 的 handle； - offset：数据偏移； - value：数据指针； - len: 数据长度；
返回说明	int 0: 调用成功；非 0: 调用失败；
功能描述	- 往 GATT Server 设备写长数据，Server 需要 response； - 最终的写入结果需要通过 gattc_write_long_cb 回调函数获取；

5.3.4.20 写命令请求

函数原型	int bt_manager_gatt_client_write_command(int conn_id, int char_handle, bool signed_write, uint8_t *value, uint16_t len)
参数说明	- conn_id：连接标识符，一连多时，不同的连接有不同的 ID 值； - char_handle：Server 端包含可写属性的 Characteristic value 的 handle； - signed_write：是否需要添加认证签名； - value：数据指针； - len: 数据长度；
返回说明	int 0: 调用成功；非 0: 调用失败；
功能描述	往 GATT Server 设备写命令，Server 不需要 response；

5.3.4.21 读请求

函数原型	int bt_manager_gatt_client_read_request(int conn_id, int char_handle)
参数说明	- conn_id：连接标识符，一连多时，不同的连接有不同的 ID 值； - char_handle：Server 端包含可读属性的 Characteristic value 的 handle；
返回说明	int 0: 调用成功；非 0: 调用失败；
功能描述	

函数原型	int bt_manager_gatt_client_read_request(int conn_id, int char_handle)
	- 读取 GATT Server 设备的数据； - 最终的读取结果需要通过 gattc_read_cb 回调函数获取；

5.3.4.22 读长数据请求

函数原型	int bt_manager_gatt_client_read_long_request(int conn_id, int char_handle, int offset)
参数说明	
	- conn_id：连接标识符，一连多时，不同的连接有不同的 ID 值； - char_handle：Server 端包含可读属性的 Characteristic value 的 handle； - 数据的偏移；
返回说明	int 0：调用成功；非 0：调用失败；
功能描述	
	- 读取 GATT Server 设备的数据； - 最终的读取结果需要通过 gattc_read_cb 回调函数获取；

5.3.4.23 发现所有 Service

函数原型	int bt_manager_gatt_client_discover_all_services(int conn_id, uint16_t start_handle, uint16_t end_handle)
参数说明	
	- conn_id：连接标识符，一连多时，不同的连接有不同的 ID 值； - start_handle：起始的 handle; - end_handle：结束的 handle;
返回说明	int 0：调用成功；非 0：调用失败；
功能描述	
	- 从指定 handle 范围中发现所有的服务； - 发现的结果通过 gattc_dis_service_cb 回调函数返回；

5.3.4.24 指定 UUID 发现 Service

函数原型	int bt_manager_gatt_client_discover_services_by_uuid(int conn_id, const char *uuid, uint16_t start_handle, uint16_t end_handle)
参数说明	- conn_id：连接标识符，一连多时，不同的连接有不同的 ID 值； - uuid：指定的 uuid； - start_handle：起始的 handle; - end_handle：结束的 handle;
返回说明	int 0: 调用成功；非 0: 调用失败；
功能描述	- 指定 uuid 从指定 handle 范围中发现所有符合条件的服务； - 发现的结果通过 gattc_dis_service_cb 回调函数返回；

5.3.4.25 发现指定 Service 的 Characteristic

函数原型	int bt_manager_gatt_client_discover_service_all_char(int conn_id, void *svc_handle)
参数说明	- conn_id：连接标识符，一连多时，不同的连接有不同的 ID 值； - svc_handle：指定 Service 的 handle;
返回说明	int 0: 调用成功；非 0: 调用失败；
功能描述	- 发现指定 Service 里面的所有 Characteristic； - 发现的结果通过 gattc_dis_char_cb 回调函数返回；

5.3.4.26 发现指定 Characteristic 的 descriptor

函数原型	int bt_manager_gatt_client_discover_char_all_descriptor(int conn_id, void *char_handle)
参数说明	- conn_id：连接标识符，一连多时，不同的连接有不同的 ID 值； - char_handle：指定 Characteristic 的 handle;
返回说明	int 0: 调用成功；非 0: 调用失败；
功能描述	- 发现指定 Characteristic 里面的所有 descriptor； - 发现的结果通过 gattc_dis_desc_cb 回调函数返回；

5.3.5 总结 GATT Client 的调用流程

gatt client API 的使用流程：(可参考 bt_test.c::_bt_init)

1. 调用 bt_manager_preinit，预初始化，运行期间只需要调用一次。
2. 调用 bt_manager_enable_profile(BTMG_GATT_CLIENT_ENABLE)，设置本次要使用的 profile。
3. 填充 btmg_callback_t 结构体，即回调函数。
4. 调用 bt_manager_init，蓝牙初始化。
5. 调用 bt_manager_enable，打开蓝牙协议栈。
6. 调用 add_cmd_table，如果使用 cmd 命令需添加 gatt client 的命令列表。
7. 调用 bt_manager_gatt_client_init，初始化 gatt client ，其功能主要是建立 L2CAP socket，注册

EVENT 事件回调。

8. 调用 bt_manager_le_scan_start，开启扫描。可参考 cmd_ble_scan。
9. 调用 bt_manager_gatt_client_connect，发起连接。可参考 cmd_gatt_client_connect。
10. 连接成功后即可发现服务，读写属性内容等。可参考 bt_gattc_cmd_table 命令。

6 配置文件介绍

btmanager 依赖的配置文件有 bt_init.sh 和 bluetooth.json 两个文件。

小机端路径：

```
/etc/bluetooth/
```

SDK 源码路径：

```
package/allwinner/wireless/btmanager4.0/config/
```

6.1 bt_init.sh

内容如下：

```
#!/bin/sh
bt_hciattach="hciattach"

//bt reset pin复位，需提前配置bt的pin脚，请参考蓝牙上电章节
reset_bluetooth_power()
{
    echo 0 > /sys/class/rfkill/rfkill0/state;
    sleep 1
    echo 1 > /sys/class/rfkill/rfkill0/state;
    sleep 1
}

start_hci_attach()
{
    h=`ps | grep "$bt_hciattach" | grep -v grep`
    [ -n "$h" ] && {
        killall "$bt_hciattach"
    }

    # reset_bluetooth_power

    //下载firmware，模组初始化
    "$bt_hciattach" -n ttyS1 xradio >/dev/null 2>&1 &

    wait_hci0_count=0
    while true
    do
        [ -d /sys/class/bluetooth/hci0 ] && break
        usleep 100000
        let wait_hci0_count++
        [ $wait_hci0_count -eq 70 ] && {
            echo "bring up hci0 failed"
            exit 1
```

```
        }
    done
}

start() {

    if [ -d "/sys/class/bluetooth/hci0" ];then
        echo "Bluetooth init has been completed!!"
    else
        start_hci_attach
    fi

    d=`ps | grep bluetoothd | grep -v grep`
    [ -z "$d" ] && {
        # bluetoothd -n &

        //启动bluez协议栈
        /etc/bluetooth/bluetoothd start
        sleep 1
    }
}

ble_start() {
    if [ -d "/sys/class/bluetooth/hci0" ];then
        echo "Bluetooth init has been completed!!"
    else
        start_hci_attach
    fi

    hci_is_up=`hciconfig hci0 | grep UP`
    [ -z "$hci_is_up" ] && {
        hciconfig hci0 up
    }

    MAC_STR=`hciconfig | grep "BD Address" | awk '{print $3}'`
    LE_MAC=${MAC_STR/2/C}
    OLD_LE_MAC_T=`cat /sys/kernel/debug/bluetooth/hci0/random_address`
    OLD_LE_MAC=$(echo $OLD_LE_MAC_T | tr [a-z] [A-Z])
    if [ -n "$LE_MAC" ];then
        if [ "$LE_MAC" != "$OLD_LE_MAC" ];then
            hciconfig hci0 lerandaddr $LE_MAC
        else
            echo "the ble random_address has been set."
        fi
    fi
}

stop() {
    d=`ps | grep bluetoothd | grep -v grep`
    [ -n "$d" ] && {
        killall bluetoothd
        sleep 1
    }

    h=`ps | grep "$bt_hciattach" | grep -v grep`
    [ -n "$h" ] && {
        killall "$bt_hciattach"
        sleep 1
    }
    echo 0 > /sys/class/rfkill/rfkill0/state;
```

```
    sleep 1
    echo "stop bluetoothd and hciattach"
}

case "$1" in
  start|"")
        start
        ;;
  stop)
        stop
        ;;
  ble_start)
        ble_start
        ;;
  *)
        echo "Usage: $0 {start|stop}"
        exit 1
esac
```

- reset_bluetooth_power: 对蓝牙模组的 reset 引脚上下电，达到复位目的；
- start_hci_attach: 调用 hciattach，实现和蓝牙模组 uart 波特率同步，下载 firmware，生成 hci0 节点；
- start: 启动蓝牙，包括执行 hciattach 和 bluetoothd，其被 bt_manager_enable 函数调用；
- ble_start: 启动 BLE，暂时用不到，统一用 start 函数；
- stop: 关闭蓝牙，kill 掉蓝牙相关进程，其被 bt_manager_enable 函数调用；

注意事项：SDK 源码路径下的 bt_init.sh 不一定和小机端使用的内容相同，btmanager4.0/Makefile 会根据平台和模组使用情况，通过 sed 修改 bt_init.sh 内容。

6.2 bluetooth.json

内容如下：

```
{
        "profile":{
                "a2dp_sink":1,
                "a2dp_source":0,
                "avrcp":1,
                "hfp_hf":1,
                "hfp_ag":0,
                "gatt_client":0,
                "gatt_server":0
        },
        "a2dp_sink":{
                "device":"default",
                "buffer_time":400000,
                "period_time":100000
        },
        "a2dp_source":{
```

```
                "hci_index":0,
                "DEV":"00:00:00:00:00:00",
                "DELAY":20000
        },
        "hfp_pcm":{
                "rate":16000,
                "phone_to_dev_cap":"hw:snddaudio1",
                "phone_to_dev_play":"default",
                "dev_to_phone_cap":"CaptureMic",
                "dev_to_phone_play":"hw:snddaudio1"
        }
}
```

- profile 指默认使能的 profile

  如果用户没有主动调用 bt_manager_set_default_profile() 使能指定的 profile 时，将默认从这个条目读取配置作为默认使能的 profile。

- a2dp_sink 指 a2dp sink 播放音频相关的配置

  device：表示使用的硬件声卡；

  buffer_size：alsa 的 buffer size 参数；

  period_size：alsa 的 period size 参数；

- a2dp_source 指 a2dp_source 的配置参数，当前尚未使用；

- hfp_pcm 指 hfp over pcm 的配置参数；

  rate：蓝牙 pcm 使用的采样率，跟具体蓝牙模组有关；

  phone_to_dev_cap：主控端从蓝牙模组获取蓝牙通话音频的声卡（手机先传给蓝牙模组，蓝牙模组再通过 i2s 传给主控端，也就是对端手机讲话的声音);

  phone_to_dev_play：对端手机讲话的声音在主控端进行播放的声卡；

  dev_to_phone_cap：表示录制我方讲话声音的声卡；

  dev_to_phone_play：表示我方声音写入蓝牙模组的声卡（传输到对端手机中);

7 Btmanager-demo 使用指南

7.1 bt_test 简介

demo 程序名字为 bt_test，是一个可执行程序，该程序可以在前台或者后台运行；

查看运行帮助：

```
root@TinaLinux:/# bt_test -h
Usage:
  [OPTION]...

Options:
  -h, --help            print this help and exit
  -d, --debug    open debug :-d [0~5]
  -s, --stop     stop bt_test
  -i, --interaction      interaction
  -p, --profile=NAME    enable BT profile
  [supported profile name]:
  - a2dp-source Advanced Audio Source
  - a2dp-sink   Advanced Audio Sink
  - hfp-hf      Hands-Free
  - spp Serial Port Profile
  - gatt-server BLE server
  - gatt-client BLE client
```

参数说明：

- -h：查看帮助；
- -d：打印等级，如果不指定，打印等级默认为 3(WARNG)，使用时加上打印等级，例如-d4；
- -s：关闭 bt_test，一般在后台模式使用；
- -i：表示是否运行在交互模式；
- -p：指定 profile，当前支持 a2dp-source/a2dp-sink/hfp-hf/spp/gatt-server/gatt-client。如果需要指

定两个 profile，通过-p profile1 -p profile2 这种使用方法；

例子：运行 bt _test 在交互模式，并且指定 a2dp-sink 和 gatt-server 两个 profile，打印等级设置为 4(DEBUG):

```
bt_test -i -p a2dp-sink -p gatt-server -d4
```

默认 profile 说明

如果运行 bt_test 不加-p 指定 profile，默认选择 a2dp-sink 和 hfp-hf 这两个 profile。

可以通过修改 bluetooth.json 文件配置默认 profile;

7.1.1 后台模式

后台模式指的是运行 bt_test 程序时不加上-i 参数，例如：

```
bt_test -p a2dp-sink
```

后台模式运行之后没有程序控制台，为了方便调试与测试，我们支持在后台模式下通过写节点的方式发送控制命令，控制节点为/tmp/bt_io，例如：

```
echo get_adapter_name >/tmp/bt_io
```

7.1.2 交互模式

交互模式指指的是运行 bt_test 程序时加上了-i 参数，运行后会有控制终端，可以输入命令，例如：

```
bt_test -p a2dp-sink -i
```

在交互模式下同样可以通过控制节点/tmp/bt_io 进行控制。

```
[BT]:help
Available commands:
        enable                          enable [0/1]: open bt or not
        scan                            scan [0/1]: scan for devices
        scan_list                       scan_list: list available devices
        pair                            pair [mac]: pair with devices
        unpair                          unpair [mac]: unpair with devices
        paired_list                     paired_list: list paired devices
        get_adapter_state               get_adapter_state: get bt adapter state
        get_adapter_name                get_adapter_name: get bt adapter name
        set_adapter_name                set_adapter_name [name]: set bt adapter name
        get_adapter_addr                get_adapter_addr: get bt adapter address
        get_device_name                 get_device_name[mac]: get remote device name
        set_scan_mode                   set_scan_mode [0~2]:0-NONE,1-page scan,2-inquiry
    scan&page scan
        set_page_to                     real timeout = slots * 0.625ms
        set_io_cap                      set_io_cap [0~4]:0-keyboarddisplay,1-displayonly,2-
    displayyesno,3-keyboardonly,4-noinputnooutput
        avrcp                           avrcp [play/pause/stop/fastforward/rewind/forward/
    backward]: avrcp control
        connect                         connect [mac]:generic method to connect
        disconnect                      disconnect [mac]:generic method to disconnect
        remove                          remove [mac]:removes the remote device
        a2dp_src_run                    a2dp_src_run -p [folderpath]  or  a2dp_src_run -f [
    firepath]
        a2dp_src_status                 a2dp_src_status [status]:play pause forward
    backward
        a2dp_src_stop                   a2dp_src_stop:stop a2dp source playing
```

```
a2dp_set_vol                    a2dp_set_vol: set a2dp device volme
a2dp_get_vol                    a2dp_src_get_vol: get a2dp device volme
hfp_answer                      hfp_answer: answer the phone
hfp_hangup                      hfp_hangup: hangup the phone
hfp_dial                        hfp_dial [num]: call to a phone number
hfp_cnum                        hfp_cum: Subscriber Number Information
hfp_last_num                   hfp_last_num: calling the last phone number dialed
hfp_vol                         hfp_vol [0~15]: update phone's volume.
sppc_connect                    sppc_connect[mac]:connect to spp server
sppc_send                       sppc_send xxx: send data
sppc_disconnect                 sppc_disconnect[mac]:disconnect spp server
spps_accept                     spps_accept the client
spps_send                       spps_send data
spps_disconnect                 spps_disconnect dst
get_version                     get_version: get btmanager version
debug                           debug [0~5]: set debug level
ex_dbg                          ex_dbg [mask]: set ex debug mask
```

7.1.2.1 通用命令说明

命令的定义实现位于 bt_cmd.c, 路径：

```
package/allwinner/wireless/btmanager4.0/demo/bt_cmd.c
```

7.1.2.1.1 help

- 用法：直接输入命令；
- 说明：打印出目前支持的命令和使用方法。

7.1.2.1.2 quit

- 用法：直接输入命令；
- 说明：退出 bt_test 程序；

7.1.2.1.3 enable

- 用法：enable 1 或者 enable 0；
- 说明：打开或关闭蓝牙；

7.1.2.1.4 get_version

- 用法：直接输入命令；
- 说明：获取 btmanager 版本信息；

7.1.2.1.5 debug

- 用法：debug [0~5]，例如：debug 4，打印等级请参考 btmg_log_level_t；
- 说明：设置打印等级；

7.1.2.1.6 ex_dbg

- 用法：ex_dbg [mask]，例如：ex_dbg [16]，mask 的含义请参考 ex_debug_mask；
- 说明：设置拓展调试标志位，用于打开特殊打印信息；

7.1.2.2 经典蓝牙命令说明

7.1.2.2.1 scan

- 用法：scan 1 或者 scan 0；
- 说明：打开或关闭 BT 扫描;

7.1.2.2.2 scan_list

- 用法：直接输入命令；
- 说明：获取扫描设备缓存列表；

7.1.2.2.3 pair

- 用法：pair [mac]，例如：pair 10:48:B1:97:9D:5F；
- 说明：配对指定设备；

7.1.2.2.4 unpair

- 用法：unpair [mac]，例如：unpair 10:48:B1:97:9D:5F；
- 说明：取消与指定设备的配对；

7.1.2.2.5 paired_list

- 用法：直接输入命令；
- 说明：获取已配对设备缓存列表；

7.1.2.2.6 get_adapter_state

- 用法：直接输入命令;
- 说明：获取 BT 状态;

7.1.2.2.7 get_adapter_name

- 用法：直接输入命令;
- 说明：获取本地 BT 名称;

7.1.2.2.8 set_adapter_name

- 用法：set_adapter_name [name]，例如：set_adapter_name allwinner-bt
- 说明：设置本地 BT 名称;

7.1.2.2.9 get_adapter_addr

- 用法：直接输入命令;
- 说明：获取本地 BT Mac 地址;

7.1.2.2.10 get_device_name

- 用法：get_device_name[mac]，例如：get_device_name 10:48:B1:97:9D:5F;
- 说明：根据 Mac 获取对端 BT 设备名称

7.1.2.2.11 set_scan_mode

- 用法：set_scan_mode [0~2]

0-NONE ：不可以被发现，不可以被连接;

1-page scan：不可以被发现，可以被连接;

2-inquiry scan&page scan：可以被发现，可以被连接;

- 说明：设置可发现、可连接状态;

7.1.2.2.12 set_page_to

- 用法：set_page_to [slots]，page timeout = slots * 0.625ms；

例如 set_page_to 1000，设置的时间为 625ms；

- 说明：设置 page timeout；

7.1.2.2.13 set_io_cap

- 用法：set_io_cap [0~4]；

0-keyboarddisplay

1-displayonly

2-displayyesno

3-keyboardonly

4-noinputnooutput

详细请参考 btmg_io_capability_t 的说明；

- 说明：设置设备输入输出能力，

7.1.2.2.14 avrcp

- 用法：avrcp [play/pause/stop/fastforward/rewind/forward/backward]，例如：avrcp play；

详细请参考 btmg_avrcp_command_t 的说明；

- 说明：Avrcp CT 发送音频控制命令；

7.1.2.2.15 connect

- 用法：connect [mac]，例如：connect 10:48:B1:97:9D:5F；
- 说明：连接指定设备，连接的设备务必是扫描列表或者已配对列表里面的；

7.1.2.2.16 disconnect

- 用法：disconnect [mac]，例如：disconnect 10:48:B1:97:9D:5F；
- 说明：断开与指定设备的连接；

7.1.2.2.17 remove

- 用法：remove[mac]，例如：remove 10:48:B1:97:9D:5F；
- 说明：移除扫描缓存中的指定设备；

7.1.2.2.18 a2dp_src_run

- 用法：a2dp_src_run -p [folder path] 或 a2dp_src_run -f [file path]；

a2dp_src_run -p [folder path] ： 指定文件夹播放，例如：a2dp_src_run -p /mnt;

a2dp_src_run -f [file path]: 指定文件播放，例如：a2dp_src_run -f /mnt/bt_test.wav;

- 说明：初始化 a2dp_src 并开始播放音频，当前仅支持播放 wav 格式；

7.1.2.2.19 a2dp_src_set_status

- 用法：a2dp_src_set_status [status]；状态有 play、pause、forward、backward；

例如：a2dp_src_set_status forward

- 说明：用于控制播放、暂停以及文件夹播放时上一曲和下一曲操作；

7.1.2.2.20 a2dp_src_stop

- 用法：直接输入命令;
- 说明：停止 a2dp_src 播放；

7.1.2.2.21 a2dp_set_vol

- 用法：set_vol [0~100]，例如：set_vol 80；
- 说明：设置 A2DP 设备播放的音量，设置作为 A2DP Sink 时，手机如果支持绝对音量，可以通过该命令调节手机的音量。设备作为 A2DP Source 时，通过该接口调节播放音量大小；

7.1.2.2.22 a2dp_get_vol

- 用法：直接输入命令；
- 说明：与 a2dp_set_vol 相反，获取 A2DP 设备的音量；

7.1.2.2.23 hfp_answer

- 用法：直接输入命令；
- 说明：接听电话；

7.1.2.2.24 hfp_hangup

- 用法：直接输入命令；
- 说明：拒接或挂断电话；

7.1.2.2.25 hfp_dial

- 用法：hfp_dial [number], 例如：hfp_dial 10086；
- 说明：指定号码拨号；

7.1.2.2.26 hfp_cnum

- 用法：直接输入命令；
- 说明：获取本机号码；

7.1.2.2.27 hfp_last_num

- 用法：直接输入命令；
- 说明：回拨最后一个通话；

7.1.2.2.28 hfp_vol

- 用法：hfp_vol [0~15]，例如：hfp_vol 8；
- 说明：设置通话音量；

7.1.2.2.29 sppc_connect

- 用法：sppc_connect [mac]，例如：sppc_connect 10:48:B1:97:9D:5F；
- 说明：连接 spp server 设备，连接之前请确认设备已经使用 pair 命令配对过了；

7.1.2.2.30 sppc_send

- 用法：sppc_send [data]，例如：sppc_send 112233AABB；
- 说明：发送数据给 spp server；

7.1.2.2.31 sppc_disconnect

- 用法：sppc_disconnect[mac]，例如：sppc_disconnect 10:48:B1:97:9D:5F；
- 说明：断开与 spp server 设备连接；

7.1.2.2.32 spps_accept

- 用法：直接输入命令；
- 说明：启动 spp server 并开始监听；

7.1.2.2.33 spps_send

- 用法：spps_send[data]，例如：spps_send112233AABB；
- 说明：发送数据给 spp client；

7.1.2.2.34 spps_disconnect

- 用法：spps_disconnect[mac]，例如：spps_disconnect10:48:B1:97:9D:5F；
- 说明：断开与 spp client 设备连接；

7.1.2.3 BLE-GATT_Server 命令说明

```
root@TinaLinux:# bt_test -i -p gatt-server
....
[BT]:help
Available commands:
        gatts_init                      gatts_init: gatt server init
```

```
        gatts_deinit                    gatts_deinit:gatt server deinit
        ble_name                        ble name [name/none] to set name or get name
        ble_advertise                   ble_advertise [0/1]/[off/on]: Disable/Enable ble
    advertising
        ble_disconnect_all              ble_disconnect: disconnect all le connections
        gatt_indicate                   gatt_indicate <char_handle=0> [value]
        gatt_notify                     gatt_notify <char_handle=0> [value]
        gatt_testcase_server            gatt_testcase_server [testid]
```

7.1.2.3.1 gatts_init

- 用法：直接输入命令;
- 说明：初始化 GATT server，注意运行 bt_test -i -p gatt-server 之后不需要单独执行 gatts_init 了，除非执行了 gatts_deinit 又需要重新初始化;

7.1.2.3.2 gatts_deinit

- 用法：直接输入命令;
- 说明：反初始化 GATT server;

7.1.2.3.3 ble_name

- 用法：

ble name [name]：设置 BLE 名字;

ble name：不带参数，获取 BLE 名字;

- 说明：设置和获取 BLE 名字;

7.1.2.3.4 ble_advertise

- 用法：

ble_advertise on：打开广播;

ble_advertise off：关闭广播;

- 说明：打开或关闭 BLE 广播;

7.1.2.3.5 ble_disconnect_all

- 用法：直接输入命令;
- 说明：断开所有连接;

7.1.2.3.6 gatt_indicate

- 用法：gatt_indicate [char_handle] [value]，例如：gatt_indicate 0x0038 123456789
- 说明：指定 handle 发出指示;

7.1.2.3.7 gatt_notify

- 用法：gatt_notify [char_handle] [value]，例如：gatt_notify 0x0038 123456789
- 说明：指定 handle 发出通知;

7.1.2.3.8 gatt_testcase_server

- 用法：gatt_testcase_server [testid]，例如：gatt_testcase_server 123;
- 说明：用来 gatt 两板对测，相同的 testid，即对端执行 gatt_testcase_client 123;

7.1.2.4 BLE-GATT_Client 命令说明

```
root@TinaLinux:# bt_test -i -p gatt-client
....
[BT]:help
Available commands:
        gattc_init                      gattc_init [none]: gatt client init
        gattc_deinit                    gattc_deinit [none]: gatt client deinit
        ble_scan                        ble_scan [on, passive, off] [int=0x<hex>] [win=0x<
    hex>] [wl] [dups/nodups]
        ble_connect                     ble_connect [mac] [public random]: gatt client
    method to connect
        ble_disconnect                  ble_disconnect [mac]: gatt client method to
    disconnect
        ble_conn_update                 ble_conn_update [min][max][latency][timeout]
        ble_connections                 ble_connections
        ble_select                      ble_select [mac]
        gatt_write                      gatt_write <handle> <offset=0> <data> [string]:
    gatt write request
        gatt_write_cmd                  gatt_write_cmd <handle> <sign=0> <data> [string] [
    repeat]: gatt write command without response
        gatt_read                       gatt_read <handle>:gatt read request
        gatt_write_long                 gatt_read <handle> [reliable]:gatt write long
    request
        gatt_discover                   gatt_discover: discovery all services
```

```
    gatt_discover_uuid              gatt_discover_uuid [uuid]: discovery services by
uuid
    get_mtu                         get_mtu: get MTU
    register_notify_indicate                register_notify [value handle]:gatt
register notify callback
    unregister_notify_indicate              unregister_notify_indicate: gatt unregister
 all notify callback
    gatt_testcase_client            gatt_testcase_client [testid]
```

7.1.2.4.1 gattc_init

- 用法：直接输入命令;
- 说明：初始化 GATT Client，注意运行 bt_test -i -p gatt-client 之后不需要单独执行 gattc_init 了，除非执行了 gattc_deinit 又需要重新初始化;

7.1.2.4.2 gattc_deinit

- 用法：直接输入命令;
- 说明：反初始化 GATT Client;

7.1.2.4.3 ble_scan

- 用法：ble_scan [on, passive, off] [int=0x] [win=0x] [wl] [dups/nodups]

on 和 off 是必选项，例如 ble on；其他参数可选，参数含义如下：

passive：被动扫描；

int：扫描间隔；

win：扫描窗口；

wl：扫描过滤策略，对应 filter_policy 参数；

dups/nodups：是否打开重复过滤；

例如

```
ble_scan on
ble_scan off
ble_scan on int=0x0010 win=0x0010
ble_scan on wl
ble_scan on int=0x0010 win=0x0010 nodups
```

- 说明：打开或关闭 BLE 扫描；

7.1.2.4.4 ble_connect

- 用法：ble_connect [mac] [public random]

例如 ble_connect 48:45:20:FE:F0:B9 random；public 与 random 表示对端设备的 mac 地址类型；

- 说明：GATT Client 设备连接 GATT Server 设备；

7.1.2.4.5 ble_disconnect

- 用法：ble_disconnect [mac]

例如 ble_connect 48:45:20:FE:F0:B9；

- 说明：GATT Client 设备断开与 GATT Server 设备的连接；

7.1.2.4.6 ble_conn_update

- 用法：ble_conn_update [min] [max] [latency] [timeout]

min：最小连接间隔，取值范围：0x0006 ~ 0x0C80;

max：最大连接间隔，取值范围：0x0006 ~ 0x0C80;

latency：从设备延迟，取值范围：0x0000 ~0x01F3；

timeout：连接监控超时，取值范围：0x000A ~0x0C80, Time = N * 10 ms, Time Range: 100 ms ~ 32 s；

- 说明：更新连接参数；

7.1.2.4.7 ble_connections

- 用法：直接输入命令;
- 说明：获取当前连接列表，结果通过回调函数打印出来；

7.1.2.4.8 ble_select

- 用法：ble_select [mac];
- 说明：在连接列表中选择要操作的设备；

7.1.2.4.9 gatt_write

- 用法：

handle：特征值 handle;

offser：固定位为 0;

data：数据内容，例如:123456aabb，如果后一个参数是 string，可以任意内容;

[strings]：可选项，如果有 string，会把解析成字符串，否则是 16 进制数据;

- 说明：写 Gatt Server 端特征值，Server 需要 response;

7.1.2.4.10 gatt_write_cmd

- 用法：gatt_write_cmd [string] [repeat]

handle 特征值 handle;

sign：是否需要添加认证签名;

data：数据内容，例如:123456aabb，如果后一个参数是 string，可以任意内容;

[strings]：可选项，如果有 string，会把解析成字符串，否则是 16 进制数据;

repeat：重复写的次数;

- 说明：写 Gatt Server 端特征值，Server 不需要 response;

7.1.2.4.11 gatt_read

- 用法：gatt_read

handle 特征值 handle;

- 说明：读 Gatt Server 端特征值内容;

7.1.2.4.12 gatt_read_long

- 用法：gatt_read [reliable]

handle：特征值 handle;

reliable：是否可靠读;

- 说明：以数据方式读 Gatt Server 端特征值;

7.1.2.4.13 gatt_discover

- 用法：直接输入命令;
- 说明：发现当前连接的服务列表，通过回调函数打印出来;

7.1.2.4.14 gatt_discover_uuid

- 用法：gatt_discover_uuid [uuid]

uuid：指定 service 的 uuid;

- 说明：指定 uuid 发现服务;

7.1.2.4.15 get_mtu

- 用法：直接输入命令;
- 说明：获取当前 BLE 连接的 MTU 大小;

7.1.2.4.16 register_notify_indicate

- 用法：register_notify_indicate [value handle]

value handle：对端 CCC 特征值的 handle;

- 说明：注册 notify 或者 indicate，同时打开对方的 notify 或 indicate 开关，执行该命令之后，对端才能 notify 或者 indicate 给本机;

7.1.2.4.17 unregister_notify_indicate

- 用法：直接输入命令;
- 说明：注销 notify 或者 indicate;

7.1.2.4.18 gatt_testcase_client

- 用法：gatt_testcase_client[testid]，例如：gatt_testcase_client123;
- 说明：用来 gatt 两板对测，相同的 testid，即对端执行 gatt_testcase_server 123;

7.2 功能验证

7.2.1 A2DP Sink 测试

1. 指定 a2dp-sink 运行 bt_test:

```
bt_test -p a2dp-sink
```

或者进入交互模式（建议）

```
bt_test -p a2dp-sink -i
```

2. 手机打开蓝牙，搜索 “aw-bt-test-xxxx” 的蓝牙设备，并发起连接;

3. 连上之后，手机打开音乐播放器 APP，播放音乐，设备端将同步输出声音;

7.2.2 AVRCP CT 测试

通过 avrcp play/pause/stop/fastforward/rewind/forward/backward 命令可进行音乐播放，暂停，快进，快退，上下曲等操作，例如下一曲:

```
avrcp forward
```

7.2.3 A2DP Souce 测试

1. 确认音频文件

A2DP Souce 测试需要播放音频，需要保证设备中有 wav 音频文件，否则请通过 adb push 进去，路径可以根据实际情况选择，一般建议/mnt 或者/tmp 路径，音频文件可以在 SDK 中获取，例如:

```
tina/package/testtools/testdata/audio_wav/common/44100-stereo-s16_le-10s.wav
```

2. 指定 a2dp-source 运行 bt_test

建议运行为交互模式

```
bt_test -i -p a2dp-source
```

3. 扫描蓝牙音箱/耳机设备

如果需要连接的设备尚未配对（第一次连接或者已经取消配对），务必先扫描设备：

```
scan 1
```

通过观察打印或者通过命令查看是否扫描到：

```
scan_list
```

如果扫描到了，请停止扫描：

```
scan 0
```

4. 连接蓝牙音箱/耳机设备

连接的蓝牙设备务必是扫描缓存列表或者已配对列表中的，否则连接失败；

```
connect [mac]，例如: connect 40:EF:4C:7B:77:ED
```

连接成功会有如下打印：

BTMG[bt_test_a2dp_source_connection_state_cb:228]: A2DP source connected with device:....;

连接的过程中包含了配对的过程；

5. 开始播放

- a2dp_src_run -p [folder path]： 指定文件夹播放，例如：

```
a2dp_src_run -p  /mnt
```

- a2dp_src_run -f [file path]: 指定文件播放，例如：

```
a2dp_src_run -p  /mnt/bt_test.wav
```

可以通过 a2dp_src_set_status [status] 进行播放控制，播放状态有 play、pause、forward、backward；

forward、backward 在指定文件夹播放时才有用。例如播放暂停：

```
a2dp_src_set_status pause
```

6. 音量大小的设置和获取可以通过如下命令：

```
a2dp_set_vol [0~100]和 a2dp_get_vol
```

7. 结束播放

停止播放的线程，结束播放；

```
a2dp_src_stop
```

7.2.4 AVRCP TG 测试

连接上蓝牙音箱/耳机，点击蓝牙音箱/耳机的播放暂停、上下曲测试，设备端会收到对应的按键事件，并通过回调函数上报，同时会有如下打印：“BT palying music playing with device:xxx”....

7.2.5 SPP Client 测试

1. 指定 spp 运行 bt_test

```
bt_test -i -p spp
```

2. 扫描设备

如果需要连接的设备尚未配对（第一次连接或者已经取消配对），务必先扫描设备：

```
scan 1
```

扫描设备之后停止扫描

```
scan 0
```

3. 配对设备

如果设备尚未配对过或者已经取消配对，在连接之前务必先配对：

```
pair [mac]
```

4. 连接设备

连接已经配对的设备：

```
sppc_connect [mac]
```

5. 发送数据

```
sppc_send xxxxx
```

6. 断开连接

```
sppc_disconnect [mac]
```

7.2.6 SPP Server 测试

1. 指定 spp 运行 bt_test

```
bt_test -i -p spp
```

2. 初始化 pp server 并开启线程监听 client 设备

```
spps_accept
```

3. 发送数据

```
spps_send xxxx
```

4. 断开连接

主动断开对端设备

```
spps_disconnect [mac]
```

7.2.7 HFP HF 测试

1. 运行 bt_test

不带-p 指定 profile，默认为 a2dp-sink + hfp-hf

```
bt_test -i
```

2. 手机 A 搜索连接上样机，名字为 “aw-bt-test-xxxx”

3. 手机 B 播打手机 A，手机 A 接听电话

```
hfp_answer
```

4. 手机 B 播打手机 A，手机 A 拒接电话

```
hfp_hangup
```

5. 样机拨打电话

```
hfp_dial 10086
```

6. 回拨最后一个通话

```
hfp_last_num
```

7. 获取手机号码

```
hfp_cnum
```

8. 设置通话的音量

```
hfp_vol [0~15]
```

7.2.8 GATT Server 测试

1. 指定 gatt-server 运行 bt_test

```
bt_test -i -p gatt-server
```

2. 手机搜索连接

手机需要安装 “nrf connect” APP，打开 APP 后扫描搜索到 “aw-ble-test-007” 名称的 BLE 设备，并连接上；

手机 APP 可以发现样机上的服务信息，目前 demo 注册了两个 Service：

- test_svc：UUID 为 “1112”
- uart_svc: UUID 为 “6e400001-b5a3-f393-e0a9-e50e24dcca9e”

3. 读写数据

对 uuid 为 “3344” 的 characteristic 分别进行 read 和 write 操作：

- read 操作时手机 APP 会收到数值的累计增加；
- write 操作时手机 APP 发送的字符会打印在样机的终端上；

4.notify 通知测试

在 nrf connect 里面设置 CCC 为通知，点击选中【三个下】箭头，然后样机端输入命令通知手机端：

```
gatt_notify [char_handle] [value]
```

5.indicate 指示测试

在 nrf connect 里面设置 CCC 为指示，点击选中【上下】箭头，然后样机端输入命令指示手机端：

```
gatt_indicate [char_handle] [value]
```

7.2.9 GATT Client 测试

1. 指定 gatt-client 运行 bt_test

```
bt_test -i -p gatt-client
```

2. 手机创建 server 端

手机需要安装 “nrf connect” APP，打开 APP 后创建一个 device, 添加 server，并打开广播。

3. 板子搜索连接

```
ble_scan 1
```

4. 扫描到设备后停止扫描

```
ble_scan 0
```

5. 连接设备

```
ble_connect [mac]
```

6. 发现所有服务

```
gatt_discover
```

7.read 测试

参数 handle 根据发现服务列表中获取

```
gatt_read <handle>
```

读取到的值会显示在样机终端上。暂时不支持需配对才能读取的 handle。

8 常见问题排查指南

由于内容过多，已单独整理到《蓝牙常见问题排查指南.pdf》

著作权声明

商标声明

免责声明