



D1-H Tina Linux 蓝牙 开发指南

版本号: 1.1

发布日期: 2021.04.06

版本历史

版本号	日期	制/修订人	内容描述
1.0	2021.04.06	AWA1381	1. 建立初始版本。
1.1	2021.04.06	AWA1381	1. 修改 bt_init.sh 的路径。

目 录

1 概述	1
1.1 编写目的	1
1.2 适用范围	1
1.3 相关人员	1
2 Bluetooth 简介	2
2.1 Bluetooth Controller	3
2.1.1 BR/EDR Controller	3
2.1.2 LE Controller	4
2.1.2.1 LE Device address	5
2.1.2.2 Physical channel	8
2.1.2.3 LE 广播通信	9
2.1.2.4 AdvData 和 ScanRspData 格式	10
2.2 Bluetooth HOST	12
3 Tina 蓝牙协议栈介绍	13
3.1 运行 tina 蓝牙协议栈	14
3.1.1 蓝牙上电	15
3.2 bluez 协议栈配置	17
4 经典蓝牙	18
4.1 GAP	18
4.2 A2DP	18
4.3 AVRCP	20
4.4 HFP	21
4.5 经典蓝牙 API 使用说明	23
4.5.1 btmanager 数据结构说明	23
4.5.1.1 log 控制等级	23
4.5.1.2 BT 状态	23
4.5.1.3 BT 扫描模式	24
4.5.1.4 BT 绑定状态	24
4.5.1.5 BT A2dp_sink 连接状态	24
4.5.1.6 BT A2dp_sink stream 状态	24
4.5.1.7 BT AVRCP 状态	25
4.5.1.8 BT AVRCP 命令	25
4.5.1.9 BT 音乐信息	25
4.5.1.10 回调函数	26
4.5.2 初始化 API	27
4.5.2.1 设置打印级别	27
4.5.2.2 获取打印级别	27
4.5.2.3 预初始化	27
4.5.2.4 初始化	28

4.5.2.5 反初始化	28
4.5.3 GAP 协议 API	28
4.5.3.1 设置模式	28
4.5.3.2 profile 默认使能	28
4.5.3.3 蓝牙使能	29
4.5.3.4 配对回复确认	29
4.5.3.5 配对自动回复	29
4.5.3.6 启动扫描	30
4.5.3.7 停止扫描	30
4.5.3.8 判断是否在扫描状态	30
4.5.3.9 蓝牙配对	30
4.5.3.10 取消配对	30
4.5.3.11 获取状态	31
4.5.3.12 获取本地蓝牙名称	31
4.5.3.13 设置本地蓝牙名称	31
4.5.3.14 获取 mac 地址	31
4.5.3.15 指定 profile 连接	32
4.5.3.16 指定 profile 断开连接	32
4.5.3.17 蓝牙通用连接	32
4.5.3.18 蓝牙通用断开	32
4.5.3.19 移除设备	33
4.5.4 A2dp sink 协议相关 API	33
4.5.5 A2dp Source API	33
4.5.5.1 初始化	33
4.5.5.2 反初始化	33
4.5.5.3 开始启动播放	33
4.5.5.4 发送音频数据	34
4.5.5.5 停止播放	34
4.5.6 AVRCP API	34
4.5.6.1 音频控制	34
4.5.6.2 音量控制	35
4.5.7 HFP API	35
4.5.7.1 接听电话	35
4.5.7.2 拒接或挂断电话	35
4.5.7.3 指定号码拨号	35
4.5.7.4 拨打上一次电话	36
4.5.7.5 获取本机号码	36
4.6 API 调用指南	36
5 蓝牙低功耗	38
5.1 Attribute	38
5.1.1 Attribute Type	38
5.1.2 Attribute Handle	39

5.1.3	Attribute Value	39
5.1.4	Attribute Permissions	39
5.2	GATT	40
5.3	GATT Server	41
5.4	GATT Server API 介绍	45
5.4.1	GATT Server 常见的数据结构	45
5.4.1.1	characteristic properties	45
5.4.1.2	Characteristic descriptor properties	45
5.4.1.3	Attribute Permissions	45
5.4.1.4	回调函数与参数相关结构体	46
5.4.1.5	服务注册相关结构体	47
5.4.1.6	广播类结构体	49
5.4.2	初始化 API	49
5.4.2.1	gatt server 初始化	49
5.4.2.2	gatt server 反初始化	50
5.4.3	服务注册类函数	50
5.4.3.1	创建一个服务	50
5.4.3.2	添加一个 characteristic	50
5.4.3.3	添加一个 descriptor	50
5.4.3.4	启动一个服务	51
5.4.3.5	停止一个服务	51
5.4.3.6	删除一个服务	51
5.4.4	服务操作类函数	51
5.4.4.1	回复 client 读请求	51
5.4.4.2	回复 client 写请求	52
5.4.4.3	通知 client	52
5.4.4.4	指示 client	52
5.4.5	ble gap API	52
5.4.5.1	设置随机地址	52
5.4.5.2	使能广播	53
5.4.5.3	设置广播数据	53
5.4.6	总结 API 的使用说明	53
6	demo 使用指南	54
6.1	a2dp sink 测试步骤	55
6.2	a2dp Souce 测试步骤	55
6.3	avrcp 测试步骤	56
6.4	gatt server 测试步骤	56
6.4.0.1	hfp client 测试步骤	56
6.4.1	配置文件	56
7	蓝牙常见问题排查指南	58
7.1	排查指南顺序	58

插图

2-1 协议结构图	2
2-2 BR/EDR 链路状态	3
2-3 LE 链路状态	5
2-4 Format of static address	6
2-5 Format of non-resolvable private address	7
2-6 Format of resolvable private address	7
2-7 Mapping of PHY channel to physical channel index and channel type . .	8
2-8 Advertising physical channel PDU	9
2-9 Advertising type	9
2-10 Advertising and Scan Response data format	10
2-11 Permitted usages for data types	11
3-1 tina 蓝牙协议栈结构图	13
3-2 主控与 bt 硬件连接简图	15
4-1 A2DP 传输结构	19
4-2 A2DP 传输例子	19
4-3 AVRCP 框架	20
4-4 AVRCP 示例	21
4-5 HFP 框架	22
5-1 Attribute	38
5-2 GATT Profile attribute types	40
5-3 gatt 通信模型	41
5-4 gatt server 模型	42
5-5 weight service	44

1 概述

1.1 编写目的

介绍 Allwinner 平台上 Bluetooth 开发。

1.2 适用范围

Allwinner 软件平台 Tina。

Allwinner 硬件平台 D1-H。

1.3 相关人员

适用 Tina 平台的广大客户和对蓝牙感兴趣的人员。

2 Bluetooth 简介

蓝牙技术发展至今已经迭代多个版本，截至目前（2020 年）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：

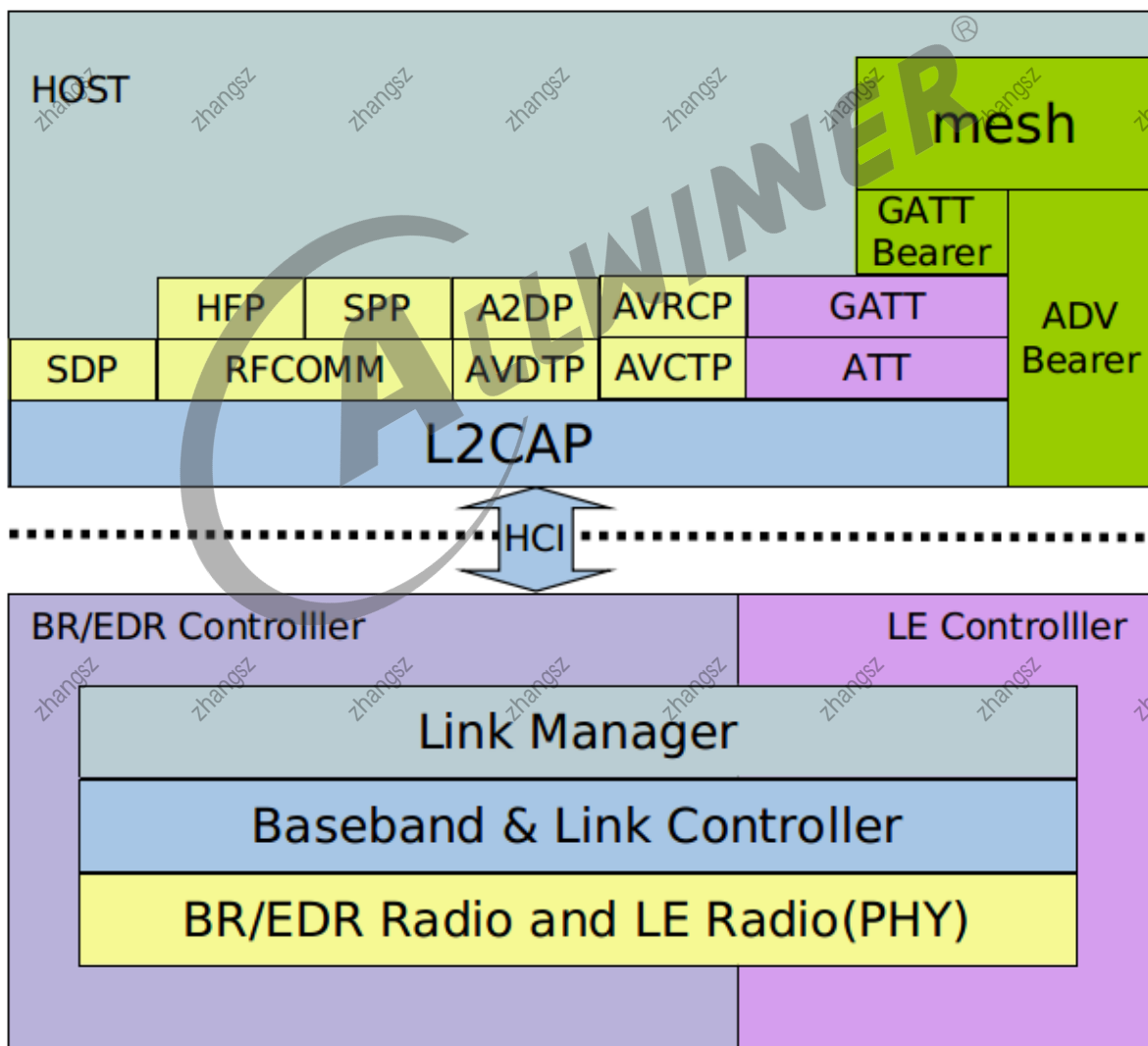


图 2-1: 协议结构图

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 D1-H，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 Bluetooth Controller

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：

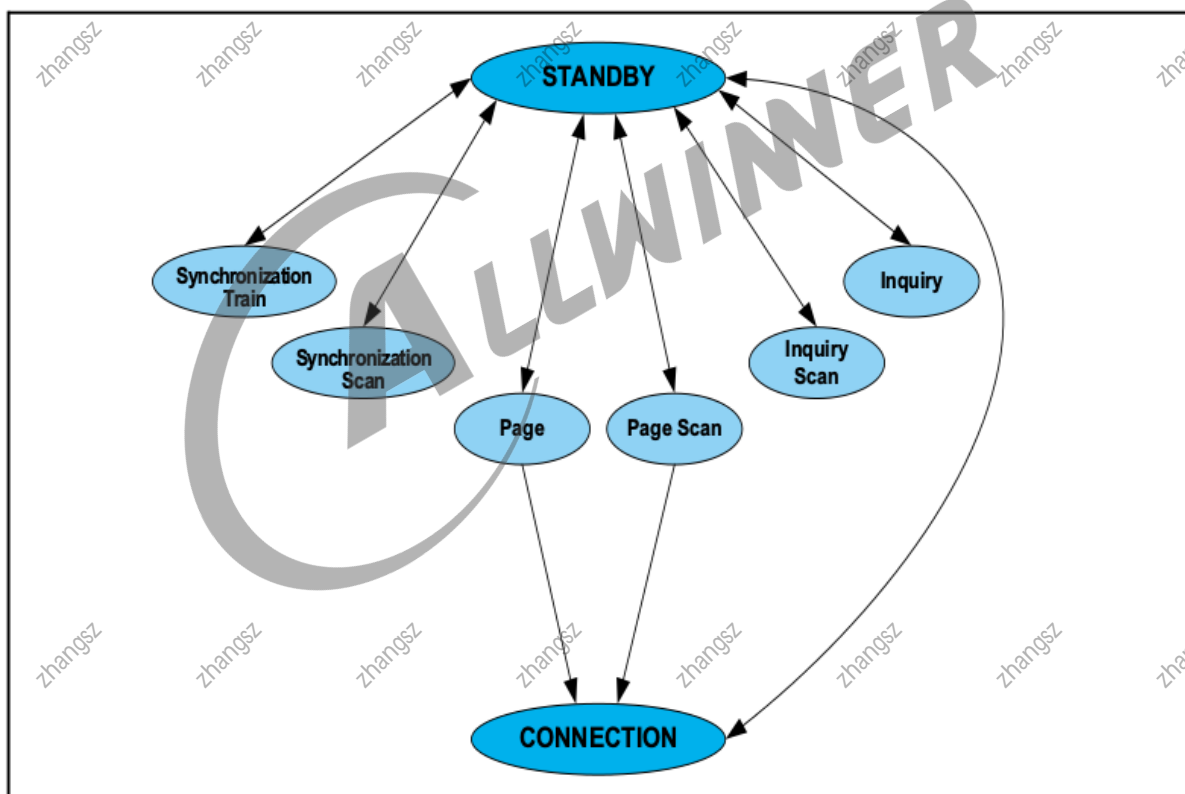


图 2-2: BR/EDR 链路状态

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。

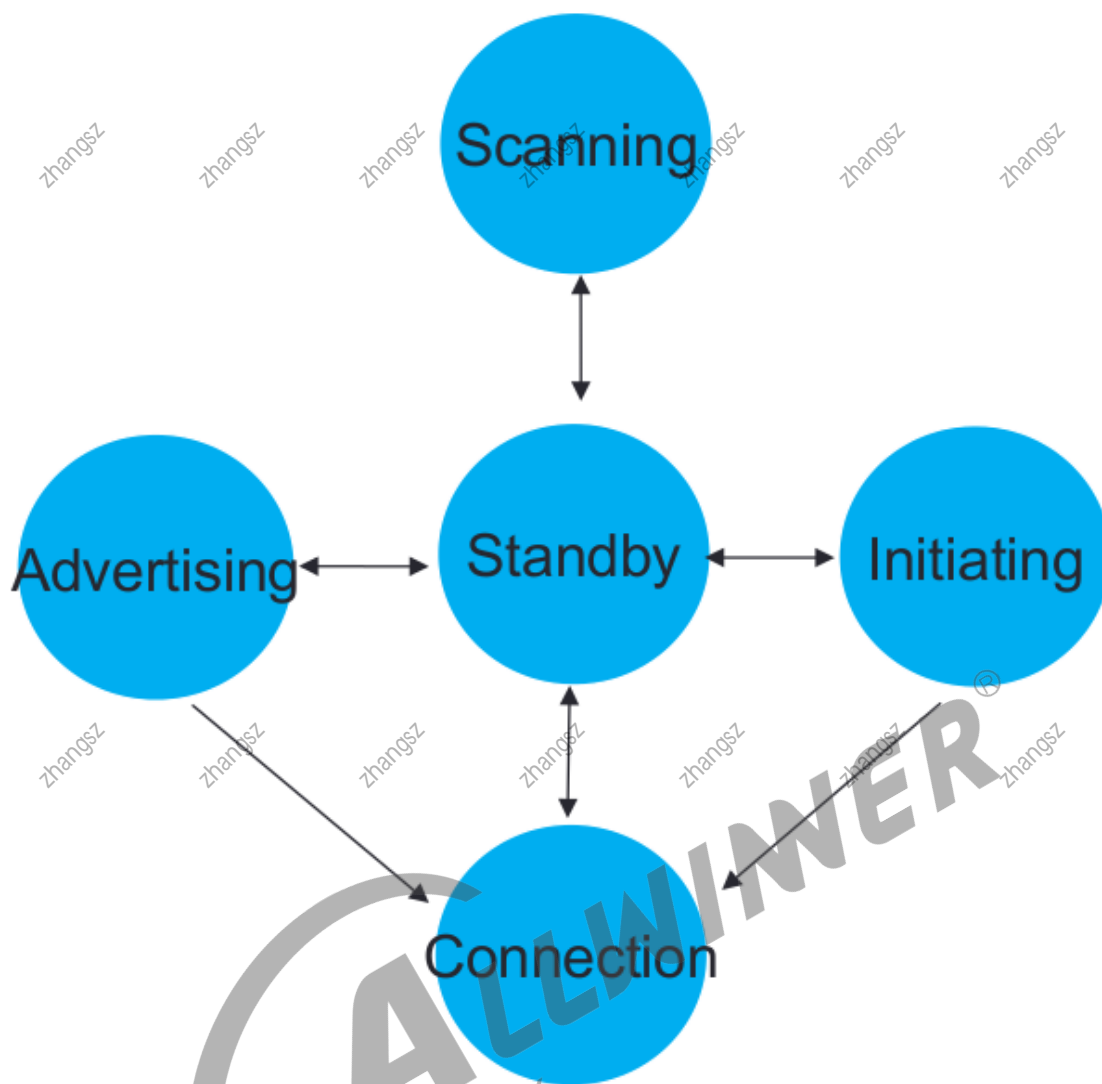


图 2-3: LE 链路状态

- 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

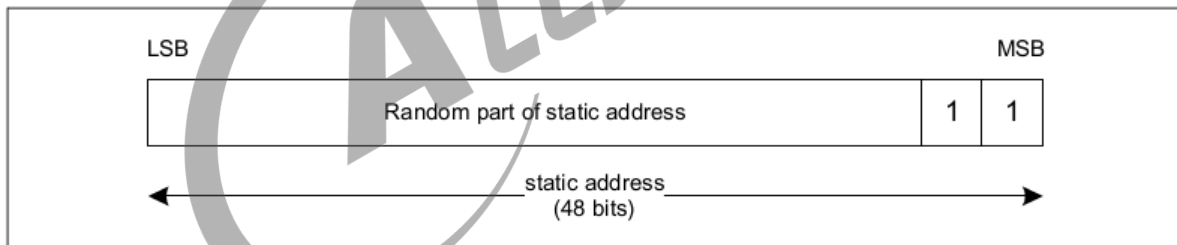


图 2-4: Format of static address

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

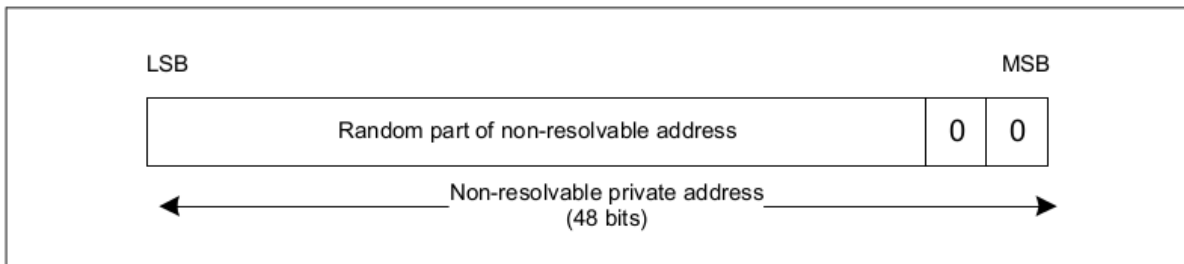


图 2-5: Format of non-resolvable private address

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

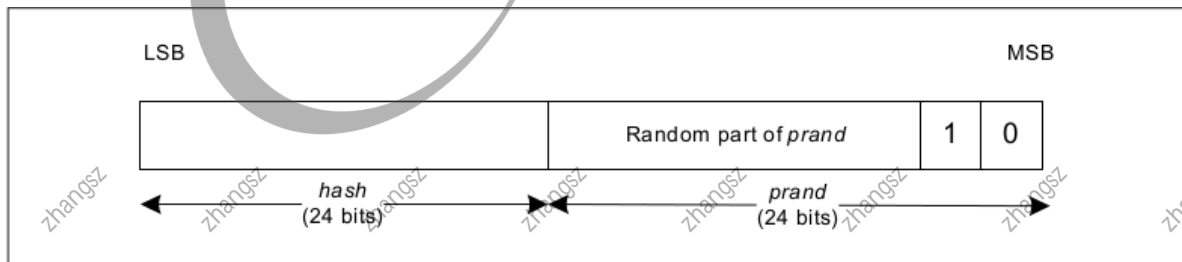


图 2-6: Format of resolvable private address

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：

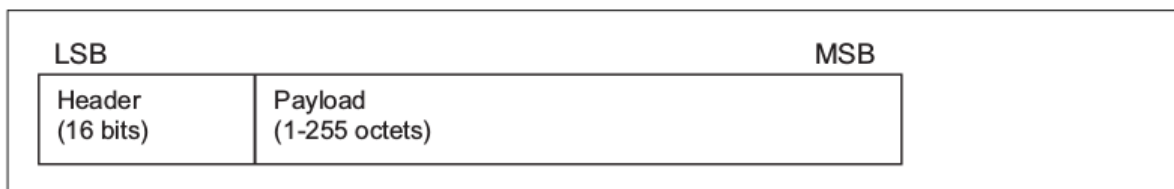


Figure 2.4: Advertising physical channel PDU

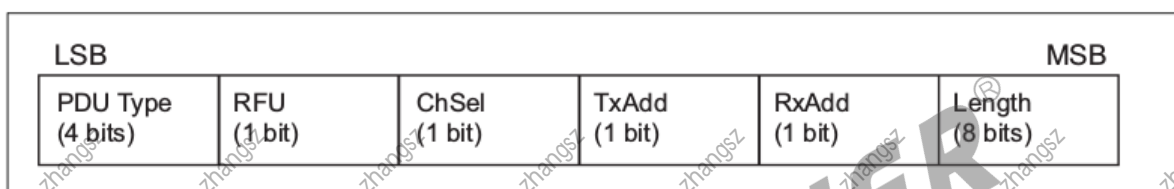


Figure 2.5: Advertising physical channel PDU header

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_DIRECT_IND	AdvA(6 octets) TargetA(6 octets)	可被指定的设备连接，不可被扫描
	ADV_NONCONN_IND	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

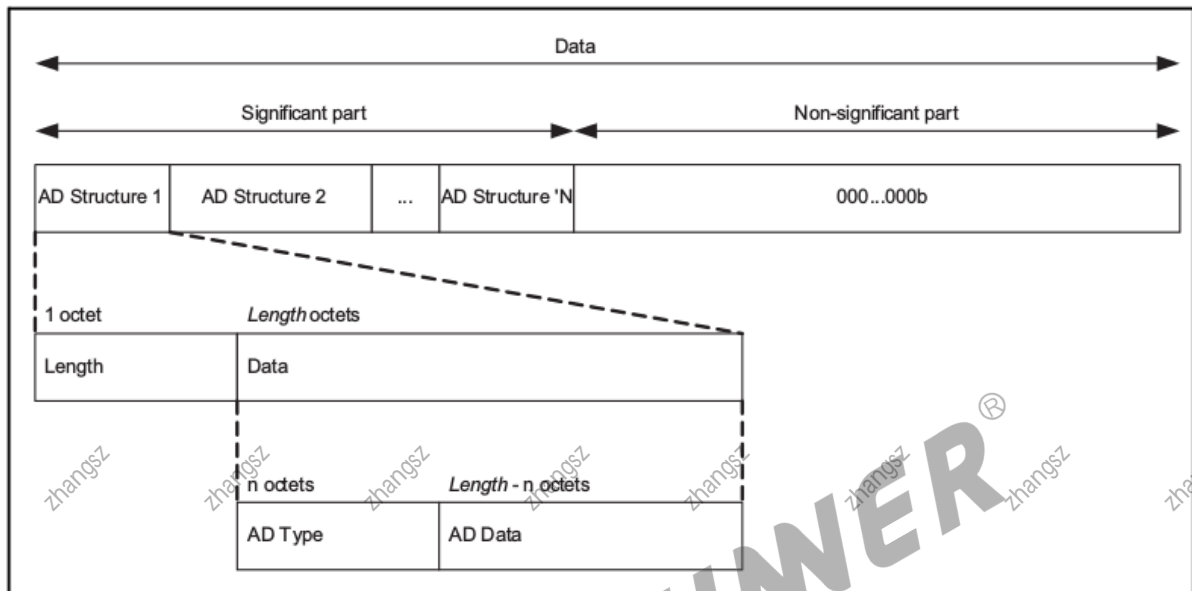


图 2-10: Advertising and Scan Response data format

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。数据内容又分为 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 Bluetooth HOST

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 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）。

具体的 profile 我们在后续章节再进行详细介绍。

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 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。当前的软件结构图如下：

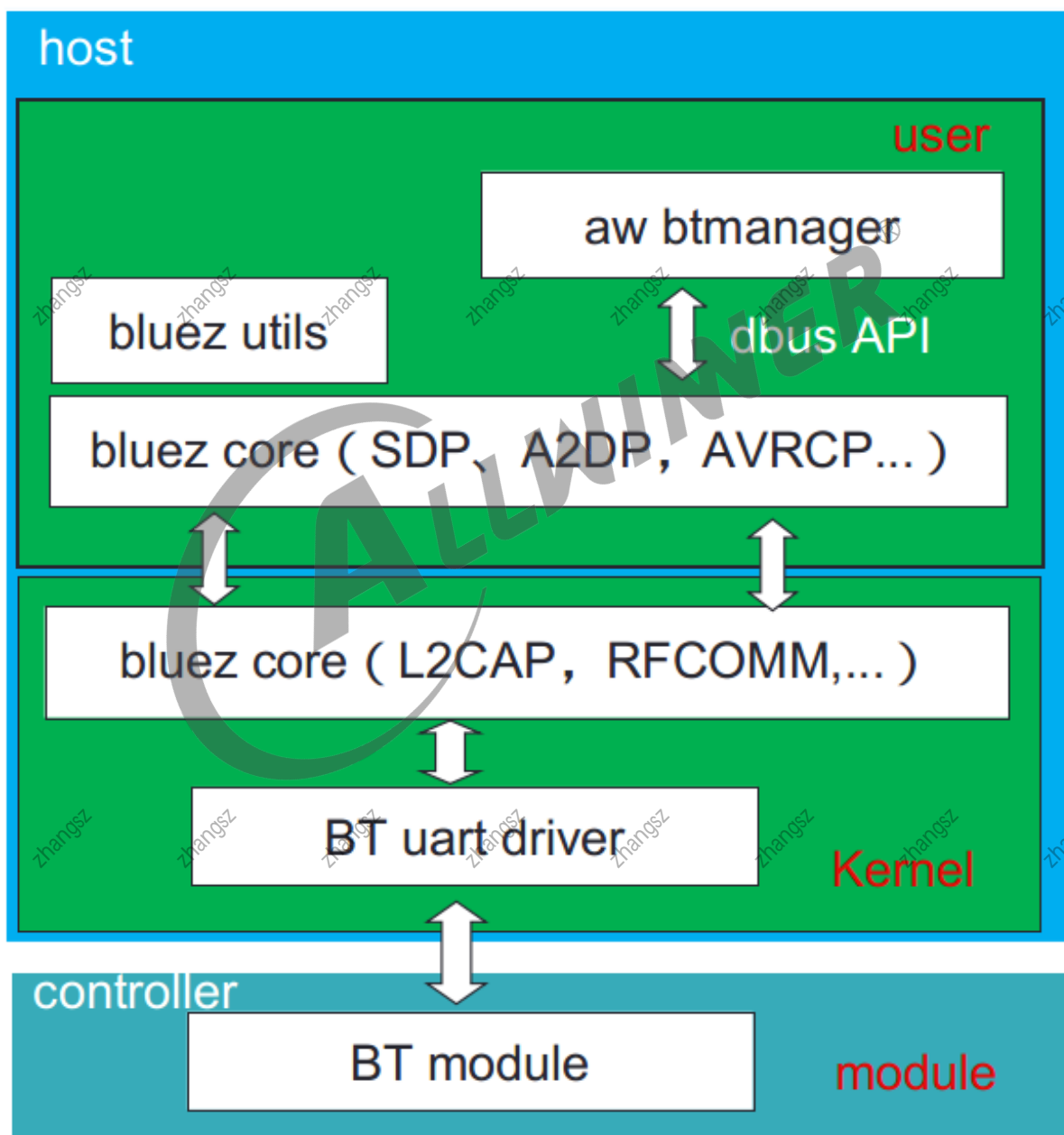


图 3-1: tina 蓝牙协议栈结构图

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 D1-H）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：

```
tina/package/allwinner/btmanager/config/xradio_bt_init.sh
```

bt_init.sh 的内容主要为以下：

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

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

    //启动bluez协议栈
    /etc/Bluetooth/Bluetoothd start
}
```

3.1.1 蓝牙上电

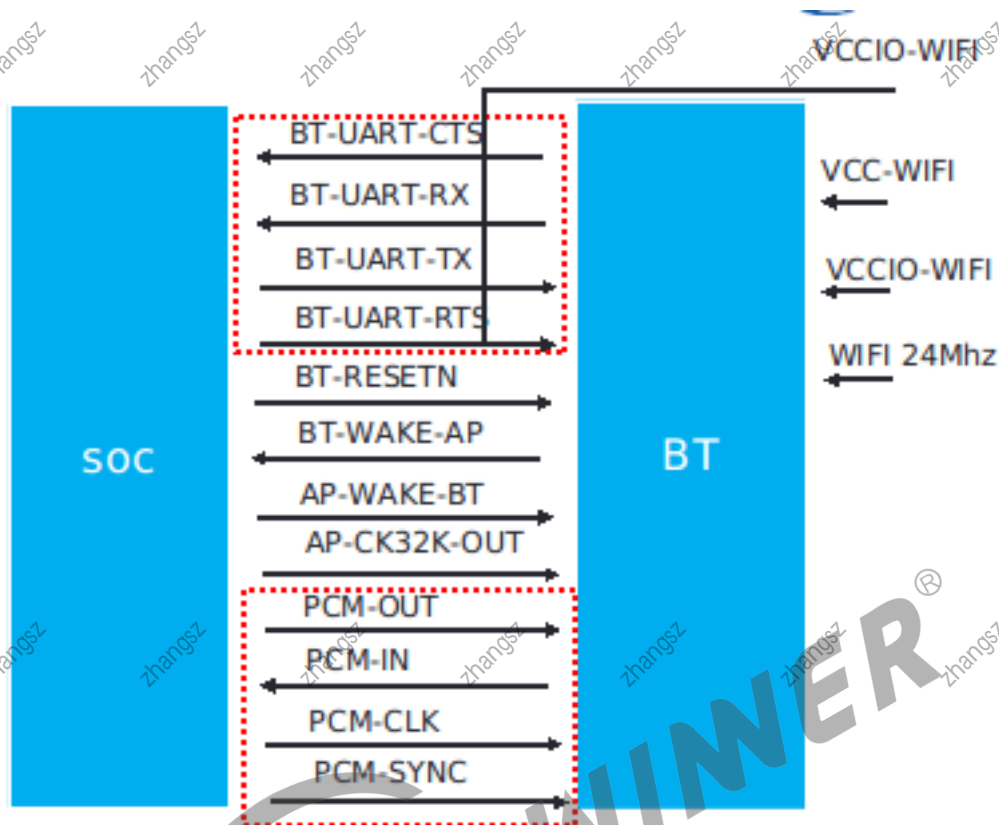


图 3-2: 主控与 bt 硬件连接简图

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 Wi-Fi，BT 一体，所以大部分操作同 Wi-Fi 一致，详情可参考《D1-H_Tina_Linux_Wi-Fi_开发指南》。

以下是 linux 5.4 board.dts Bluetooth 相关的配置

```
157     uart1_pins_a: uart1_pins@0 { /* For EVB1 board */
158         pins = "PG6", "PG7", "PG8", "PG9";
159         function = "uart1";
```

```
160         drive-strength = <10>;
161         bias-pull-up;
162     };
163
164     uart1_pins_b: uart1_pins { /* For EVB1 board */
165         pins = "PG6", "PG7", "PG8", "PG9";
166         function = "gpio_in";
167     };
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477 &uart1 {
478     pinctrl-names = "default", "sleep";
479     pinctrl-0 = <&uart1_pins_a>;
480     pinctrl-1 = <&uart1_pins_b>;
481     status = "okay";
482 };
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535     rfkill: rfkill@0 {
536         compatible = "allwinner,sunxi-rfkill";
537         chip_en;
538         power_en;
539         status = "okay";
540     };
541
542     ...
543
544
545
546
547
548     bt: bt@0 {
549         compatible = "allwinner,sunxi-bt";
550         pinctrl-0 = <&wlan_pins_a>;
551         pinctrl-names = "default";
552         clock-names = "32k-fanout1";
553         clocks = <&ccu CLK_FANOUT1_OUT>;
554         /*bt_power_num = <0x01>;*/
555         /*bt_power = "axp803-dldo1";*/
556         /*bt_io_regulator = "axp803-dldo1";*/
557         /*bt_io_vol = <3300000>;*/
558         /*bt_power_vol = <330000>;*/
559         bt_rst_n = <&pio PG 18 GPIO_ACTIVE_LOW>;
560         status = "okay";
561     };
562
563
564
565
566
567
568
569
570
571
572
573
574     btlpm: btlpm@0 {
575         compatible = "allwinner,sunxi-btlpm";
576         uart_index = <0x1>;
577         bt_wake = <&pio PG 16 GPIO_ACTIVE_HIGH>;
578         bt_hostwake = <&pio PG 17 GPIO_ACTIVE_HIGH>;
579         status = "okay";
580     };
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
```

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

D1-H 目前已经适配了 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 --->
btmanager --->
  *- btmanager-core..... Bluetooth manager core
  <*> btmanager-demo..... Tina btmanager app demo
```

(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 firmware 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。
```

4 经典蓝牙

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 应运而生。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 client(针对 HFP over pcm)

4.1 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，它定义了蓝牙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。

4.2 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。

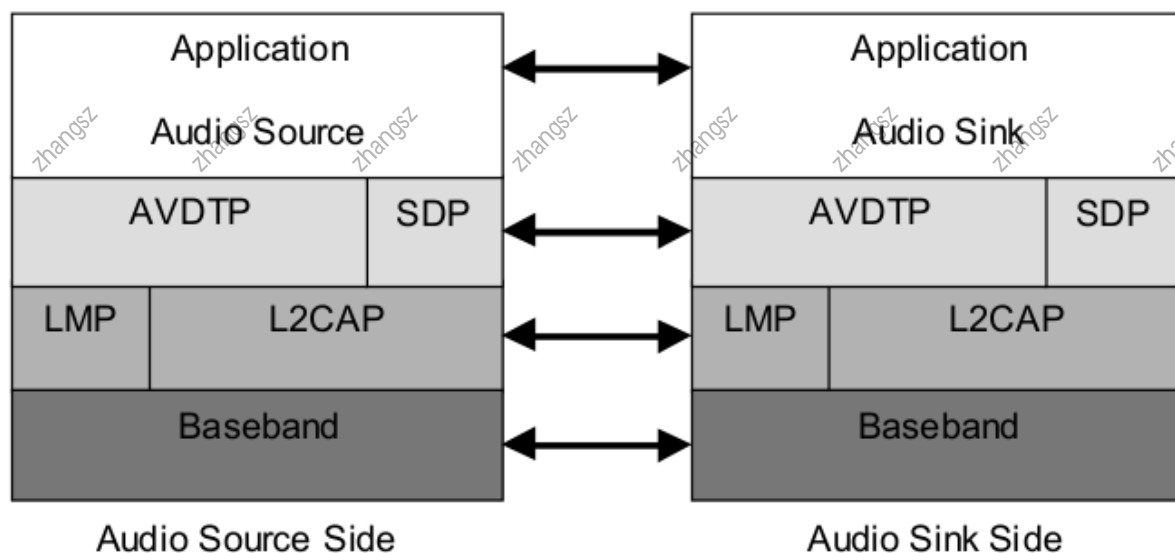


图 4-1: A2DP 传输结构

发送音频数据那一端我们称为 Source 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，信令提供数据流的发现、配置、建立和传输控制等功能，可以理解 AVDTP 是 A2DP 更基础的协议。

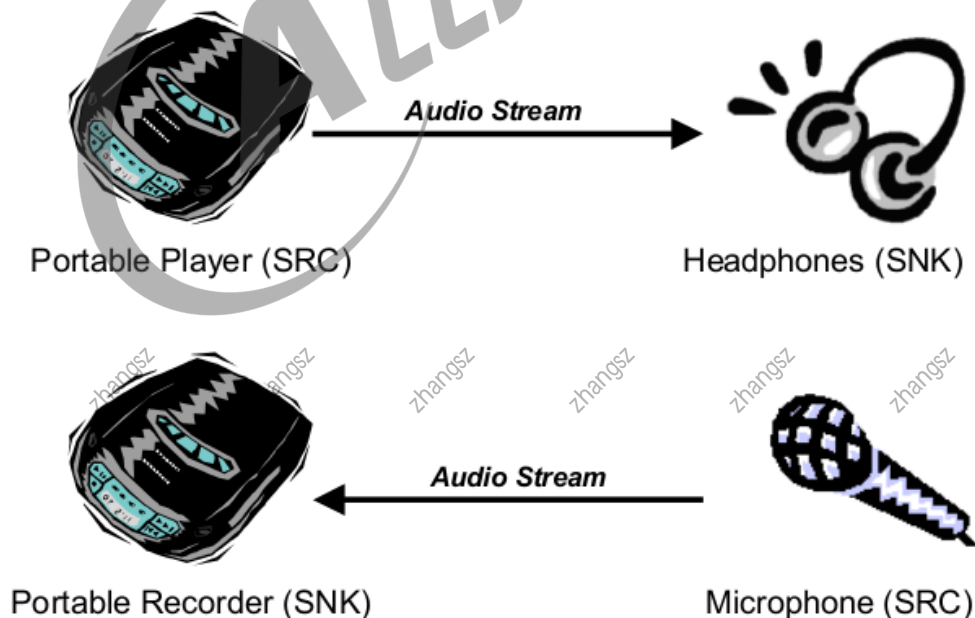


图 4-2: A2DP 传输例子

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

为 A2DP Sink。

4.3 AVRCP

AVRCP 是蓝牙音频实现蓝牙无线遥控功能的规范。

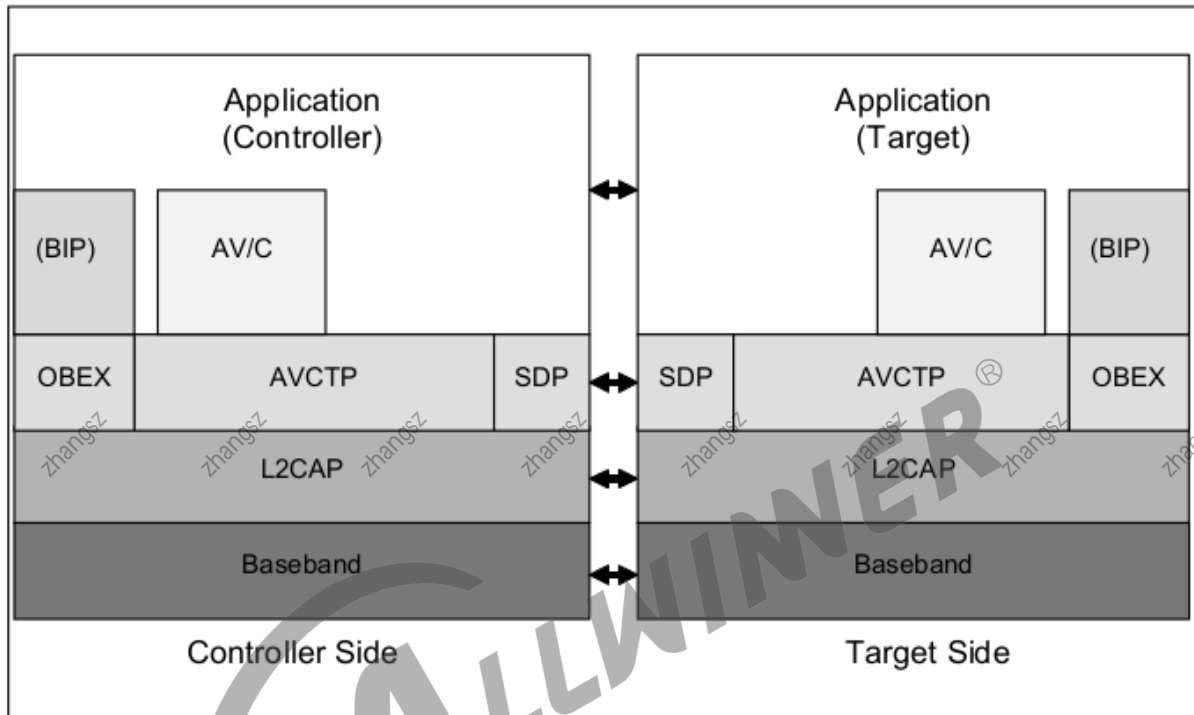


图 4-3: AVRCP 框架

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。

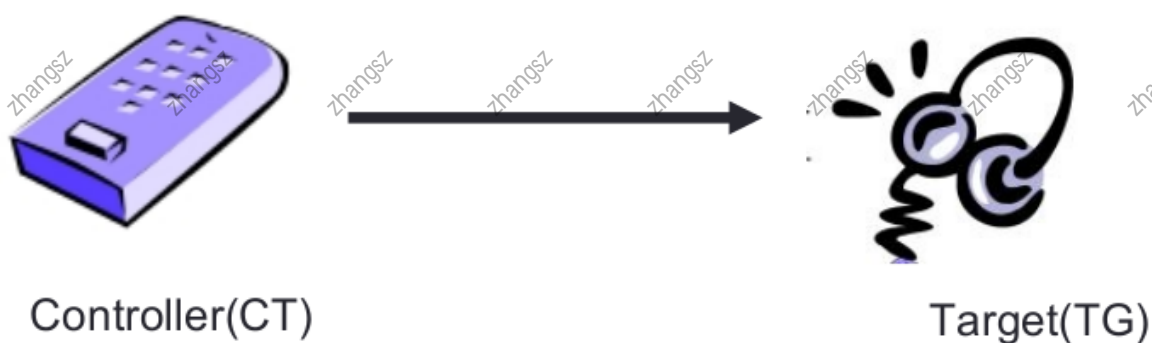


图 4-4: AVRCP 示例

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。

4.4 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）。另外一个为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）。

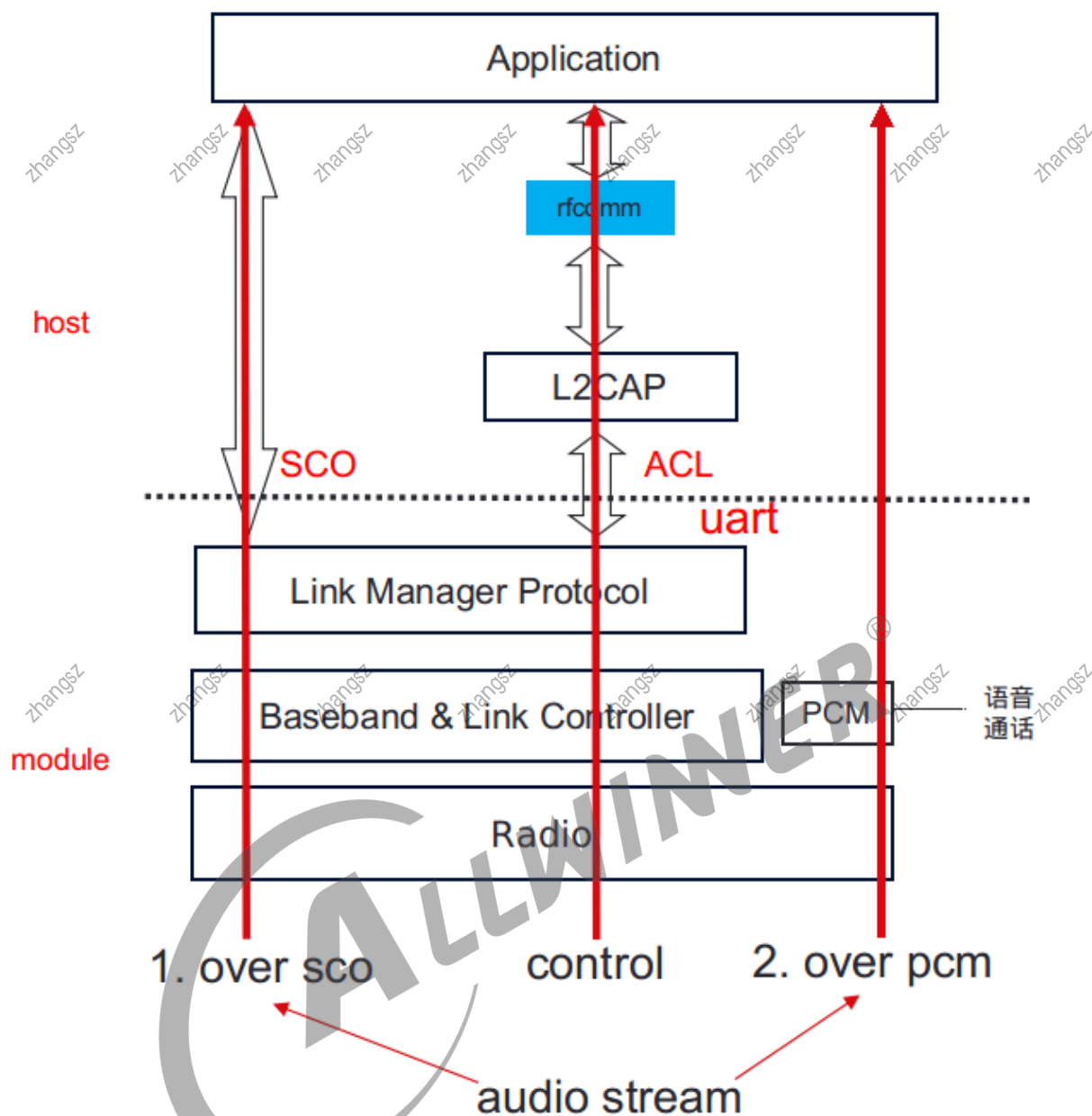


图 4-5: HFP 框架

(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。

4.5 经典蓝牙 API 使用说明

代码位置：

```
tina/package/allwinner/btmanager
```

使用示例：

```
tina/package/allwinner/btmanager/demo
```

4.5.1 btmanager 数据结构说明

4.5.1.1 log 控制等级

```
typedef enum btmg_log_level_t {  
    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_WARNNG,        //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;
```

4.5.1.2 BT 状态

```
typedef enum {  
    BTMG_STATE_OFF,  
    BTMG_STATE_ON,  
    BTMG_STATE_TURNING_ON,  
    BTMG_STATE_TURNING_OFF,  
} btmg_state_t;
```

数据结构 btmg_state_t 规定了 BT 可能处于的状态。bt_manager_get_state() 可主动获取当前 BT 的状态。如果注册了 gap_status_cb() 回调函数，BT 状态改变时，会立即回调返回当前状态。

4.5.1.3 BT 扫描模式

```
typedef enum {  
    BTMG_SCAN_MODE_NONE, //设备不可被发现和连接  
    BTMG_SCAN_MODE_CONNECTABLE, //可被连接不可被发现  
    BTMG_SCAN_MODE_CONNECTABLE_DISCOVERABLE, //可被发现可被连接  
} btmg_discovery_mode_t;
```

可被连接不可被发现此种模式一般为已经配对的设备进行直连。

4.5.1.4 BT 绑定状态

```
typedef enum {  
    BTMG_BOND_STATE_NONE,  
    BTMG_BOND_STATE_BONDING,  
    BTMG_BOND_STATE_BONDED,  
} 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() 即时返回配对状态。

4.5.1.5 BT A2dp_sink 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_connection_state_t;
```

btmg_a2dp_sink_connection_state_t 规定了 a2dp_sink 协议的连接状态，通过注册的回调函数 a2dp_sink_connection_state_cb() 即时返回连接状态。

4.5.1.6 BT A2dp_sink stream 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;
```

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。

4.5.1.7 BT AVRCP 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_ERROR,  
} 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。

4.5.1.8 BT 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() 可以发送了 AVRCP 命令。

4.5.1.9 BT 音乐信息

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

```
} btmg_track_info_t;
```

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

4.5.1.10 回调函数

btmg_callback_t 是总回调函数结构体，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，然后通过调用 bt_manager_preinit() 即可对该指针进行初始化并分配相应的空间。

```
typedef struct btmg_callback_t {  
    btmg_manager_callback_t btmg_manager_cb;  
    btmg_gap_callback_t btmg_gap_cb;  
    btmg_a2dp_sink_callback_t btmg_a2dp_sink_cb;  
    btmg_avrcp_callback_t btmg_avrcp_cb;  
    btmg_hfp_callback_t btmg_hfp_cb;  
}btmg_callback_t;
```

其中：btmg_manager_cb 用于返回 bt_manager 本身的事件回调，目前仅用于内部测试使用。btmg_gap_cb 用于返回所有的 GAP 协议相关的事件回调，对应的 btmg_gap_callback_t 定义如下：

```
typedef struct btmg_gap_callback_t {  
    bt_gap_status_cb gap_status_cb; /*used for return results of bt_manager_enable and  
    status of BT*/  
    bt_gap_discovery_status_cb gap_disc_status_cb; /*used for return discovery status of BT  
    */  
    bt_gap_dev_found_cb gap_dev_found_cb; /*used for device found event*/  
    bt_gap_update_rssi_cb gap_update_rssi_cb; /*update rssi for discovered and bonded  
    devices*/  
    bt_gap_bond_state_cb gap_bond_state_cb; /*used for bond state event*/  
    bt_gap_ssp_request_cb gap_ssp_request_cb; /*used for ssp request*/  
    bt_gap_pin_request_cb gap_pin_request_cb; /*used for pin request*/  
} btmg_gap_callback_t;
```

btmg_a2dp_sink_cb 用于返回所有的 a2dp_sink 协议相关的事件回调，对应的定义如下：

```
typedef struct btmg_a2dp_sink_callback_t {  
    /*used to report the a2dp_sink connection state*/  
    bt_a2dp_sink_connection_state_cb a2dp_sink_connection_state_cb;  
    /*used to report the a2dp_sink audio state, not recommended as mentioned before*/  
    bt_a2dp_sink_audio_state_cb a2dp_sink_audio_state_cb;  
    /*used to report the a2dp_sink volume, range: 0~16*/  
    bt_a2dp_sink_audio_volume_cb a2dp_sink_audio_volume_cb;  
} btmg_a2dp_sink_callback_t;
```

btmg_hfp_callback_t 用于返回 hfp 协议相关事件的回调，对应定义如下：


```
typedef struct btmg_hfp_callback_t {  
    bt_hfp_hs_event_cb hfp_hf_event_cb;  
} btmg_hfp_callback_t;
```

btmg_avrcp_callback_t 用于返回所有的 AVRCP 协议相关的事件回调，对应的定义如下：

```
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;  
} btmg_avrcp_callback_t;
```

其中，avrcp_play_state_cb 用于返回当前的播放状态；avrcp_track_changed_cb 在切换歌曲回即时返回当前播放音乐的信息（设备地址、歌曲名称、歌手名、歌曲专辑名、当前音乐位于音乐列表的序号、总播放列表音乐数、音乐类型、播放总时长）；avrcp_play_position_cb 用于实时返回当前音乐播放的进度（总时长、当前播放时刻）。

4.5.2 初始化 API

4.5.2.1 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 4.2.3.1
返回说明	int 0: 成功；-1: 失败。
功能描述	设置 bt_manager 内部打印等级。

4.5.2.2 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。

4.5.2.3 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。

函数原型	int bt_manager_preinit(btmg_callback_t **btmg_cb)
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。

4.5.2.4 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；-1: 失败。
功能描述	进行蓝牙的初始化设置（如读取解析配置文件、加载蓝牙协议栈、为内部变量分配空间、启动内部线程等）。

4.5.2.5 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 指针类型的指针。
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，用以退出内部线程、保存配置文件、恢复内部状态等。

4.5.3 GAP 协议 API

4.5.3.1 设置模式

函数原型	int bt_manager_set_discovery_mode(btmg_discovery_mode_t mode)
参数说明	btmg_discovery_mode_t mode：BT 设备扫描模式（详见 4.2.3.3）。
返回说明	int 0: 成功；-1: 失败。
功能描述	设置本地 BT 设备扫描模式。

4.5.3.2 profile 默认使能

函数原型	int bt_manager_set_enable_default(bool is_default)
参数说明	bool is_default: 在调用 bt_manager_init() 时是否直接默认使能 BT。 true: 默认使能 BT; false: 关闭默认使能。
返回说明	int 0: 成功; 非 0: 失败。
功能描述	在调用 bt_manager_init() 时是否直接默认使能 BT。第一次启动 BT, 只有在 bt_manager_init() 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。

4.5.3.3 蓝牙使能

函数原型	int bt_manager_enable(bool enable)
参数说明	bool enable: true: 使能 BT; false: 关闭 BT。
返回说明	int 0: 成功; 非 0: 失败。
功能描述	使能/关闭 BT。

4.5.3.4 配对回复确认

函数原型	int bt_manager_set_auto_ssp_reply(bool auto_reply)
参数说明	bool auto_reply: 是否自动回复 ssp 配对请求。true: 使能自动回复 ssp 请求; false: 关闭使能自动回复 ssp 请求。
返回说明	int 0: 成功; -1: 失败。
功能描述	在用户调用 bt_manager 的进程未退出的情况下该设置一直有效。目前初始化版本未将该设置写入配置文件, 后续版本更新会将该设置保存到配置文件。

4.5.3.5 配对自动回复

函数原型	int bt_manager_set_auto_pin_reply(bool auto_reply)
参数说明	bool auto_reply: 是否自动回复 pin 码配对请求。true: 使能自动回复 pin 码配对请求; false: 关闭使能自动回复 pin 码配对请求。
返回说明	int 0: 成功; -1: 失败。
功能描述	在用户调用 bt_manager 的进程未退出的情况下该设置一直有效。目前初始化版本未将该设置写入配置文件, 后续版本更新会将该设置保存到配置文件, (用户调用 bt_manager 是否退出的情况下都) 将永久有效。

4.5.3.6 启动扫描

函数原型	int bt_manager_start_discovery(void)
参数说明	无。
返回说明	int 0: 成功; -1: 失败。
功能描述	发起 BT 扫描。扫描状态通过 gap 回调函数 gap_disc_status_cb() 即时返回。

4.5.3.7 停止扫描

函数原型	int bt_manager_cancel_discovery(void)
参数说明	无。
返回说明	int 0: 成功; -1: 失败。
功能描述	取消 BT 扫描。扫描状态通过 gap 回调函数 gap_disc_status_cb() 即时返回。

4.5.3.8 判断是否在扫描状态

函数原型	bool bt_manager_is_discovering()
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	设置 bt_manager 内部打印等级。

4.5.3.9 蓝牙配对

函数原型	int bt_manager_pair(char *addr)
参数说明	无。
返回说明	btmg_log_level_t 打印等级类型, 详见 4.2.3.1。
功能描述	设置 bt_manager 内部打印等级。

4.5.3.10 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 设备地址。
返回说明	int 0: 成功；非 0: 失败。
功能描述	取消 BT 配对。配对状态通过 gap 回调函数 gap_bond_state_cb() 即时返回。

4.5.3.11 获取状态

函数原型	btmg_state_t bt_manager_get_state()
参数说明	无。
返回说明	btmg_state_t 类型蓝牙状态。
功能描述	获取 BT 状态。

4.5.3.12 获取本地蓝牙名称

函数原型	int bt_manager_get_name(char *name, int size)
参数说明	char *name: 用于保存 bt_name 的指针。int size: 用于保存 bt_name 空间的大小。空间大小推荐设置 MAX_BT_NAME_LEN+1。
返回说明	int 0: 成功；非 0: 失败。
功能描述	获取本地 BT 设备名称。

4.5.3.13 设置本地蓝牙名称

函数原型	int bt_manager_set_name(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 MAX_BT_NAME_LEN，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 BT 设备名称。

4.5.3.14 获取 mac 地址

函数原型	int bt_manager_get_address(char *addr, int size)
参数说明	char *addr: 用于保存本地 BT 设备地址的指针;int size: 用于保存 BT 地址空间的大小。空间大小推荐设置 MAX_BT_ADDR_LEN+1。
返回说明	int 0: 成功; 非 0: 失败。
功能描述	获取本地 BT 设备地址。

4.5.3.15 指定 profile 连接

函数原型	int bt_manager_profile_connect(char *addr,btmg_profile_t profile)
参数说明	addr: 需要连接的蓝牙设备地址; profile: 需要连接的 profile。
返回说明	int 0: 成功; 非 0: 失败。
功能描述	指定 profile 进行连接。

4.5.3.16 指定 profile 断开连接

函数原型	int bt_manager_profile_disconnect(char *addr,btmg_profile_t profile);
参数说明	addr: 需要断开的蓝牙设备地址; profile: 需要断开的 profile。
返回说明	int 0: 成功; 非 0: 失败。
功能描述	指定 profile 进行断开连接。

4.5.3.17 蓝牙通用连接

函数原型	int bt_manager_connect(const char *addr);
参数说明	addr: 需要连接的蓝牙设备地址。
返回说明	int 0: 成功; 非 0: 失败。
功能描述	蓝牙通用连接, 包括对端所有 profile。

4.5.3.18 蓝牙通用断开

函数原型	int bt_manager_disconnect(const char *addr);
参数说明	addr: 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, 包括对端所有 profile。

4.5.3.19 移除设备

函数原型	<code>int bt_manager_remove_device(const char *addr);</code>
参数说明	addr: 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	除掉指定蓝牙设备，如果是连接的设备，会将其断开，然后再将删除其配对信息，下次对端设备连接需要重新配对。

4.5.4 A2dp sink 协议相关 API

A2DP Sink 没有相关 API，已经在 btmanager 内部实现，用户不需要关心，用户要使用 A2DP sink，只需要在使能 profile 的时候使能 A2DP Sink 即可。

4.5.5 A2dp Source API

4.5.5.1 初始化

函数原型	<code>int bt_manager_a2dp_src_init(uint16_t channels,uint16_t sampling);</code>
参数说明	channels: 音频通道；sampling: 音频采样率
返回说明	int 0: 成功；非 0: 失败。
功能描述	初始化

4.5.5.2 反初始化

函数原型	<code>int bt_manager_a2dp_src_deinit(void);</code>
参数说明	无
返回说明	int 0: 成功；非 0: 失败。
功能描述	不使用的時候，进行反初始化

4.5.5.3 开始启动播放

函数原型	int bt_manager_a2dp_src_stream_start(uint32_t len);
参数说明	len：内部每次写入蓝牙协议栈的数据长度
返回说明	int 0: 成功；非 0: 失败。
功能描述	开始启动播放

4.5.5.4 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	data: 数据, len: 发送的数据长度
返回说明	int 0: 成功；非 0: 失败。
功能描述	发送音频数据

4.5.5.5 停止播放

函数原型	bt_manager_a2dp_src_stream_stop(void);
参数说明	无
返回说明	int 0: 成功；非 0: 失败。
功能描述	停止播放

4.5.6 AVRCP API

4.5.6.1 音频控制

函数原型	int bt_manager_avrcp_command(char *addr, btmg_avrcp_command_t command)
参数说明	需要控制的设备的地址; 控制命令。
返回说明	int 0: 成功；非 0: 失败。
功能描述	AVRCP 控制。

btmg_avrcp_command_t 类型如下:

BTMG_AVRCP_PAUSE: 暂停播放;

BTMG_AVRCP_STOP: 停止播放;

BTMG_AVRCP_FASTFORWARD: 快进;

BTMG_AVRCP_REWIND：快退；

BTMG_AVRCP_FORWARD：下一首；

BTMG_AVRCP_BACKWARD：前一首；

BTMG_AVRCP_VOL_UP：调高音量；

BTMG_AVRCP_VOL_DOWN：调低音量；

4.5.6.2 音量控制

函数原型	int bt_manager_vol_changed_noti(char *vol_level)
参数说明	char *vol_level: 需要设置的音量等级。设置范围为“0”~“16”。
返回说明	int 0: 成功；非 0: 失败。
功能描述	设置绝对音量等级。

4.5.7 HFP API

4.5.7.1 接听电话

函数原型	int bt_manager_hfp_client_send_at_ata(void)
参数说明	char *vol_level: 需要设置的音量等级。设置范围为“0”~“16”。
返回说明	int 0: 成功；非 0: 失败。
功能描述	设置绝对音量等级。

4.5.7.2 拒接或挂断电话

函数原型	int bt_manager_hfp_client_send_at_chup(void)
参数说明	char *vol_level: 需要设置的音量等级。设置范围为“0”~“16”。
返回说明	int 0: 成功；非 0: 失败。
功能描述	设置绝对音量等级。

4.5.7.3 指定号码拨号

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

4.5.7.4 拨打上一次电话

函数原型	int bt_manager_hfp_client_send_at_bldn(void)
参数说明	无
返回说明	int 0: 成功; 非 0: 失败。
功能描述	播打上一次播打过的电话

4.5.7.5 获取本机号码

函数原型	int bt_manager_hfp_client_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

4.6 API 调用指南

根据 4.5 章节 API, 编写了使用示例, 供用户参考, 主要的代码路径如下:

```
package/allwinner/btmanager/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 API 的使用可以总结为以下几步, (可参考 bt_test.c::_bt_init):

- 设置打印级别: bt_manager_set_loglevel。
- 预初始化: bt_manager_preinit, 运行期间只需要调用一次。
- 初始化回调函数: 主要是填充 btmng_callback_t 结构体
- 使能需要的 profile: bt_manager_enable_profile, 如果没有定制化, 将默认从 Bluetooth.json 文件 profile 条目中读取使能默认的 profile。
- 经典蓝牙初始化: bt_manager_init。

- 分配绑定和扫描的存储结构。
- 经典蓝牙使能：bt_manager_enable。



5 蓝牙低功耗

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。

5.1 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)。

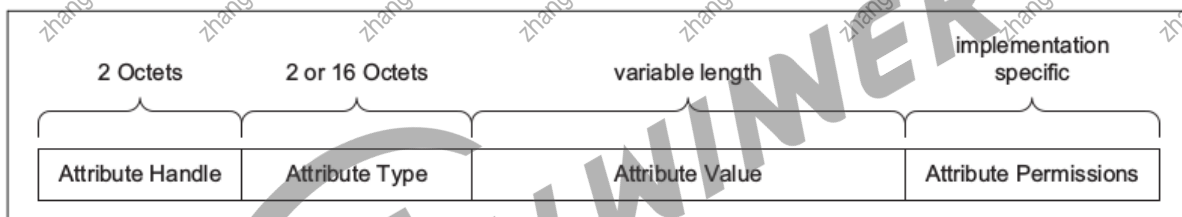


图 5-1: Attribute

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

5.1.1 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。

如蓝牙基数如下：

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

5.1.2 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，这样就无法通过 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。

5.1.3 Attribute Value

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

5.1.4 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 5-2: GATT Profile attribute types

5.2 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

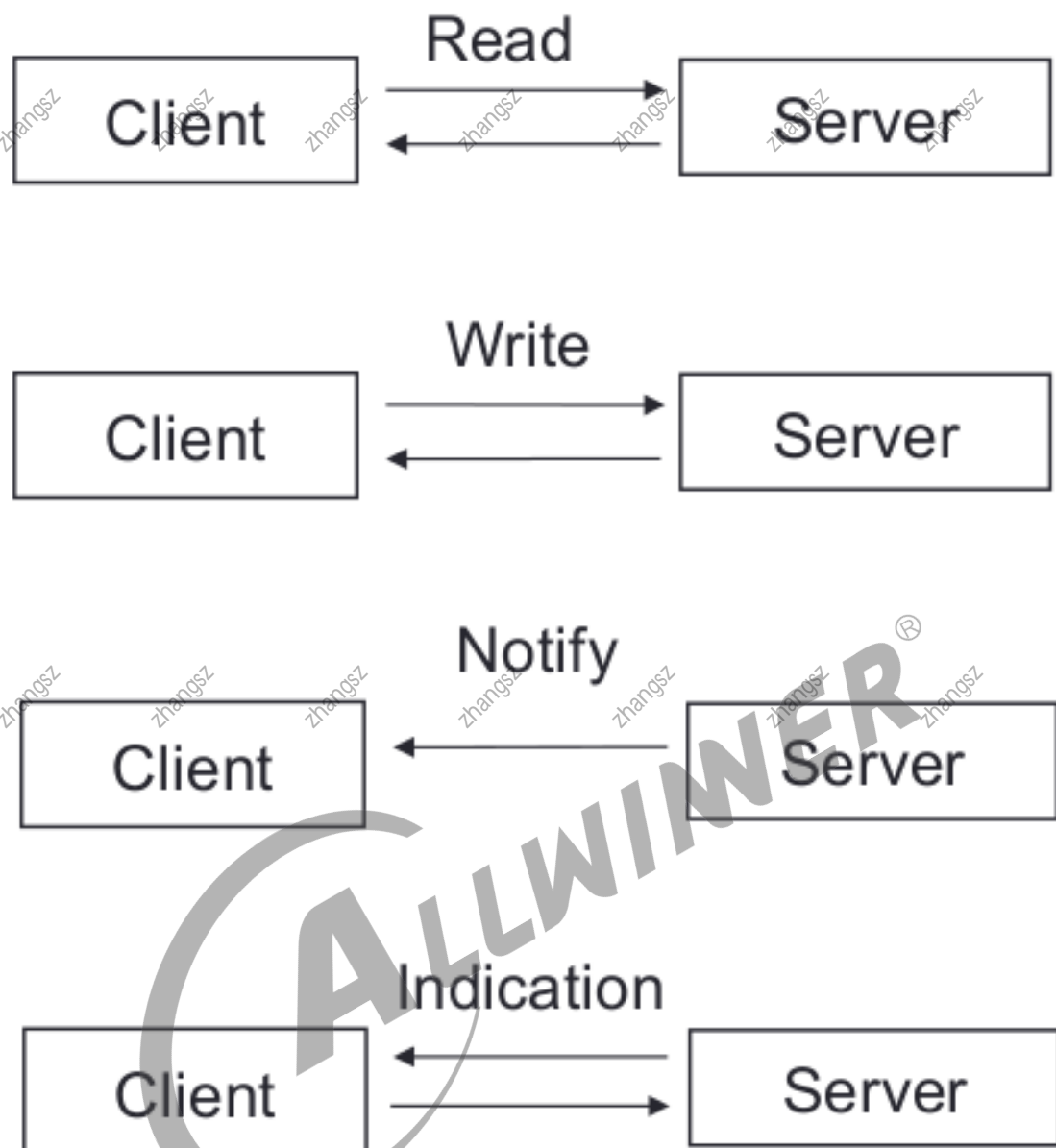


图 5-3: gatt 通信模型

其中 Server Notify 和 Server Indication 的区别是前者 server 发送数据给 client 端不需要 client 回复，后者是需要 client 端回复。

5.3 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：

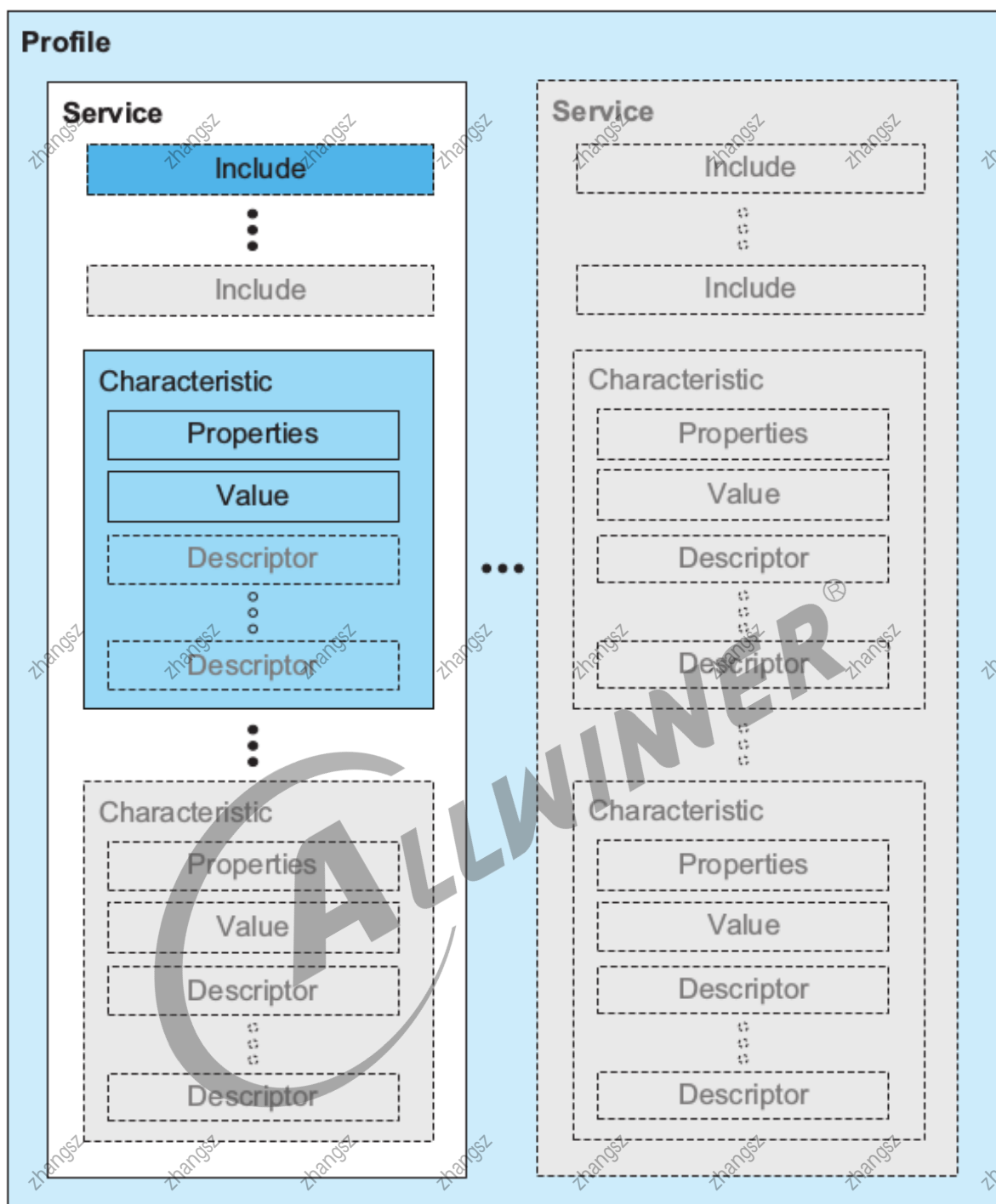


图 5-4: gatt server 模型

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 5-5: weight service

5.4 GATT Server API 介绍

当前 btmanager 仅仅支持 gatt.server 部分，后续会再逐渐支持 gatt.client。

5.4.1 GATT Server 常见的数据结构

5.4.1.1 characteristic properties

```
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;
```

5.4.1.2 Characteristic descriptor properties

```
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;
```

5.4.1.3 Attribute Permissions

```
typedef enum {  
    BT_GATT_PERM_READ = 0x01,  
    BT_GATT_PERM_WRITE = 0x02,  
    BT_GATT_PERM_READ_ENCRYPT = 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;
```

5.4.1.4 回调函数与参数相关结构体

(1) 回调函数

```
typedef struct {  
    bt_gatt_add_service_cb gatt_add_svc_cb; //增加一个service的回调函数  
    bt_gatt_add_char_cb gatt_add_char_cb;    //增加一个characteristic的回调函数  
    bt_gatt_add_desc_cb gatt_add_desc_cb;    //增加一个descriptor的回调函数  
  
    bt_gatt_connection_event_cb gatt_connection_event_cb; //gatt连接和断开事件回调函数  
    bt_gatt_service_ready_cb gatt_service_ready_cb; //gatt启动service成功后回调该函数  
  
    bt_gatt_char_read_req_cb gatt_char_read_req_cb; //  
    bt_gatt_char_write_req_cb gatt_char_write_req_cb;  
    bt_gatt_char_notify_req_cb gatt_char_notify_req_cb;  
  
    bt_gatt_desc_read_req_cb gatt_desc_read_req_cb; //client读descriptor回调函数  
    bt_gatt_desc_write_req_cb gatt_desc_write_req_cb; //client写descriptor回调函数  
  
    bt_gatt_send_indication_cb gatt_send_indication_cb; //service通知或指示回调函数  
} gatt_server_cb_t;
```

(2) 回调的参数

gatt server 连接事件

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

增加一个 Characteristic 回调函数参数

```
typedef struct {  
    int num_handle; //service中一共有多少个handle  
    int svc_handle; //表征service的handle  
} gatt_add_svc_msg_t;
```

增加一个 server 回调函数参数

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

增加一个 descriptor 回调函数参数

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

client 读请求回调函数参数

```
typedef struct {
    unsigned int trans_id;
    int attr_handle;
    int offset;    //大量数据读取的偏移
    bool is_blob_req; //是否大量数据读取，client端对一次大量数据读取可以分多次完成
} gatt_char_read_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[AG_GATT_MAX_ATTR_LEN];
    int value_len;
    bool need_rsp;
    /*是否需要回复，client如果是write req是需要回复的，如果是write cmd不需要回复。*/
} gatt_char_write_req_t;
```

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;
} gatt_desc_read_req_t;
```

descriptor 写请求回调函数参数

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

5.4.1.5 服务注册相关结构体

增加一个服务函数的参数类型

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

增加一个 characteristic 函数的参数类型

```
typedef struct {  
    char *uuid;          /*128-bit characteristic UUID*/  
    int properties;       /*The GATT characteristic properties*/  
    int permissions;     /*The GATT characteristic permissions*/  
    int svc_handle;      /*the service attr handle*/  
} gatt_add_char_t;
```

增加一个 descriptor 函数的参数类型

```
typedef struct {  
    char *uuid;          /*128-bit descriptor UUID*/  
    int properties;      /*The GATT descriptor properties*/  
    int permissions;     /*The GATT descriptor permissions*/  
    int svc_handle;  
} gatt_add_desc_t;
```

启动一个 service 函数的参数类型

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

停止一个 service 函数的参数类型

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

删除一个 service 函数的参数类型

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

service 回复 client 读操作函数的参数类型

```
typedef struct {  
    unsigned int trans_id;  
    int status;  
    int svc_handle;  
    char *value;  
    int value_len;  
    int auth_req;  
} gatt_send_read_rsp_t;
```

service 回复 client 写操作函数的参数类型

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

service 通知 client 的参数类型

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

service 指示 client 的参数类型

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

5.4.1.6 广播类结构体

gatt 广播数据结构体

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

gatt 回复 scan 广播数据结构体

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

gatt 广播属性参数

```
typedef struct {  
    uint16_t min_interval; //广播最小时间间隔  
    uint16_t max_interval; //广播最大时间间隔  
    gatt_le_advertising_type_t adv_type; //广播类型  
    gatt_le_addr_type_t own_addr_type; //广播地址  
    gatt_le_peer_addr_type_t peer_addr_type; //广播对端地址  
    char peer_addr[18];  
    uint8_t chan_map;  
    gatt_le_advertising_filter_policy_t filter; //广播过滤类型  
} gatt_le_advertising_parameters_t;
```

5.4.2 初始化 API

5.4.2.1 gatt server 初始化

函数原型	int bt_manager_gatt_server_init(gatt_server_cb_t *cb)
参数说明	见 5.4.1.5
返回说明	int 0: 成功; -1: 失败。
功能描述	gatt server 初始化函数

5.4.2.2 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; -1: 失败。
功能描述	gatt server 反初始化

5.4.3 服务注册类函数

5.4.3.1 创建一个服务

函数原型	int bt_manager_gatt_create_service(gatt_add_svc_t *svc)
参数说明	见 5.4.1.5
返回说明	int 0: 成功; -1: 失败。
功能描述	创建一个服务

5.4.3.2 添加一个 characteristic

函数原型	int bt_manager_gatt_add_characteristic(gatt_add_char_t *chr)
参数说明	见 5.4.1.5
返回说明	int 0: 成功; -1: 失败。
功能描述	指定服务中添加 characteristic

5.4.3.3 添加一个 descriptor

函数原型	int bt_manager_gatt_add_descriptor(gatt_add_desc_t *desc)
参数说明	见 5.4.1.5
返回说明	int 0: 成功; -1: 失败。

函数原型	int bt_manager_gatt_add_descriptor(gatt_add_desc_t *desc)
功能描述	指定服务中添加 descriptor

5.4.3.4 启动一个服务

函数原型	int bt_manager_gatt_start_service(gatt_star_svc_t *start_svc)
参数说明	见 5.4.1.5
返回说明	int 0: 成功; -1: 失败。
功能描述	启动 gatt service

5.4.3.5 停止一个服务

函数原型	int bt_manager_gatt_stop_service(gatt_stop_svc_t *stop_svc)
参数说明	见 5.4.1.5
返回说明	int 0: 成功; -1: 失败。
功能描述	停止一个 gatt service

5.4.3.6 删除一个服务

函数原型	int bt_manager_gatt_delete_service(gatt_del_svc_t *del_svc)
参数说明	见 5.4.1.5
返回说明	int 0: 成功; -1: 失败。
功能描述	删除一个 gatt service, 删除后如果还要使用, 需要重新注册

5.4.4 服务操作类函数

5.4.4.1 回复 client 读请求

函数原型	int bt_manager_gatt_send_read_response(gatt_send_read_rsp_t *pData)
参数说明	见 5.4.1.5
返回说明	int 0: 成功; -1: 失败。

函数原型	int bt_manager_gatt_send_read_response(gatt_send_read_rsp_t *pData)
功能描述	client 端读取 server 属性的时候，会激活对应的回调函数，server 通过该函数回复读请求的内容

5.4.4.2 回复 client 写请求

函数原型	int bt_manager_gatt_send_write_response(gatt_write_rsp_t *pData)
参数说明	见 5.4.1.5
返回说明	int 0: 成功; -1: 失败。
功能描述	client 端写 server 属性的时候，会激活对应的回调函数，server 通过该函数回复写请求，以通知 client 写是否成功

5.4.4.3 通知 client

函数原型	int bt_manager_gatt_send_notification(gatt_notify_rsp_t *pData)
参数说明	见 5.4.1.5
返回说明	int 0: 成功; -1: 失败。
功能描述	gatt server 通过该函数通知 client 消息，client 不需要回复

5.4.4.4 指示 client

函数原型	int bt_manager_gatt_send_indication(gatt_indication_rsp_t *pData)
参数说明	见 5.4.1.5
返回说明	int 0: 成功; -1: 失败。
功能描述	gatt server 通过该函数指示 client 消息，client 需要回复

5.4.5 ble gap API

5.4.5.1 设置随机地址

函数原型	int bt_manager_gatt_set_random_address(void)
参数说明	无

函数原型	int bt_manager_gatt_set_random_address(void)
返回说明	int 0: 成功; -1: 失败。
功能描述	ble 设备使用随机地址

5.4.5.2 使能广播

函数原型	int bt_manager_gatt_enable_adv(bool enable, gatt_le_advertising_parameters_t *adv_param)
参数说明	enable: 启动或关闭广播, adv_param: 广播属性参数, 见 5.4.1.6
返回说明	int 0: 成功; -1: 失败。
功能描述	使能广播

5.4.5.3 设置广播数据

函数原型	int bt_manager_gatt_set_adv_data(gatt_adv_data_t *adv_data)
参数说明	见 5.4.1.6
返回说明	int 0: 成功; -1: 失败。
功能描述	设置广播数据, 数据的格式需要按照 2.1.2.4 中的格式要求

5.4.6 总结 API 的使用说明

gatt server API 的使用步骤主要是以下几点

- 初始化 gatt server, 调用 bt_manager_gatt_server_init 函数, 其功能主要是将蓝牙协议 run 起来。
- 注册相关的回调函数, 包括构建 server, 读、写、通知、指示等。
- 构建一个 server, 主要包括创建一个 service (bt_manager_gatt_create_service), 填充 service 中的内容特性内容和描述信息 (bt_manager_gatt_add_characteristic, bt_manager_gatt_add_descriptor)。
- 构建完成一个 server 后, 就可以启动 server 了 (bt_manager_gatt_start_service)
- 使能广播, 设置广播参数 (bt_manager_gatt_enable_adv)。
- 如果需要, 还可以设置广播的数据 (bt_manager_gatt_set_adv_data)。

详情的使用例子可参考:

```
package/allwinner/btmanager/demo/gatt_server_test.c
```

小机端执行: gatt_server_test, 接着再执行test, 就可以用手机ble app进行连接读写server。

6 demo 使用指南

测试蓝牙的命令是 `bt_test`, 该 app 可以后台运行, 也可以交互运行。

启动运行帮助命令:

```
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
  -p, --profile=NAME    enable BT profile
  -i, --interaction     interaction
  -a2dp-source          Advanced Audio Source
  -a2dp-sink            Advanced Audio Sink
  -hfp-hf              Hands-Free
  -hfp-ag              Hands-Free Audio Gateway
  -hsp-hs              Headset
  -hsp-ag              Headset Audio Gateway
```

后台模式:

```
bt_test
```

交互模式:

```
bt_test -i

交互式命令列表:

[bt]#help
Available commands:
  enable [0/1]: open bt or not
  scan [0/1]: scan for devices
  scan_list: list available devices
  pair [mac]: pair with devices
  unpair [mac]: unpair with devices
  paired_list: list paired devices
  get_state: get bt state
  get_name: get bt name
  set_name [name]: set bt name
  get_addr: get bt address
  set_dis [0~2]: 0-NONE, 1-page scan, 2-inquiry scan&
  page scan
  avrcp [play/pause/stop/fastforward/rewind/forward/backward]: avrcp control
  profile_cn [mac]: a2dp sink connect
  profile_dis [mac]: a2dp sink disconnect
```

connect	connect [mac]:generic method to connect
disconnect	disconnect [mac]:generic method to disconnect
remove	remove [mac]:removes the remote device
a2dp_src_start	a2dp_src_start:start a2dp source playing
a2dp_src_stop	a2dp_src_stop:stop a2dp source playing
remove	remove [mac]:removes the remote device
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_cnum: Subscriber Number Information
hfp_last_num	hfp_last_num: calling the last phone number dialed
hfp_vol	hfp_vol [0~15]: update phone volume.
get_version	get_version: get btmanager version
debug	debug [0~5]: set debug level
ex_dbg	ex_dbg [mask]: set ex debug mask

6.1 a2dp sink 测试步骤

1. 终端执行: `bt_test -p a2dp-sink` 或者 `bt_test -p a2dp-sink -i` (将进入交互模式)
2. 使用手机打开蓝牙, 搜索"aw-bt-test-xxxx"的设备, 并进行链接
3. 手机打开播放器app, 进行播放音乐, 设备端将同步输出声音

6.2 a2dp Souce 测试步骤

a2dp source 模式必须要交互模式运行: `bt_test -i`

1. 用adb先将音频文件push 到/tmp目录下, 并命名为44100-stereo-s16_le-10s.wav。音频文件可以从tina sdk中以下路径获取:
`tina/package/testtools/testdata/audio_wav/common/44100-stereo-s16_le-10s.wav`
2. 执行: `bt_test -i -p a2dp-source`
3. 扫描指定设备获取到mac地址: `scan 1`, 扫描到后停止扫描: `scan 0`, 获取已经扫描到的设备: `scan_list`
4. 连接指定蓝牙音响: `connect mac_address(connect 40:EF:4C:7B:77:ED)`
5. 连接成功提示"connect.."字样。
6. 开始播放: `a2dp_src_start`。
7. 停止播放: `a2dp_src_stop`。

6.3 avrcp 测试步骤

1. 在 a2dp sink 测试步骤前提下（执行：bt_test -p a2dp-sink -i 进入交互模式）。
2. 分别执行：avrcp play/pause/stop/fastforward/rewind/forward/backward 可进行音乐播放，暂停，快进，快退，上下曲等操作。

6.4 gatt server 测试步骤

1. 执行：gatt_server_test。
2. 执行：test。
3. 手机app (ble scanner或nrf connect) 。
4. 连接到"aw-bt-testxx"字样的蓝牙服务。
5. 对uuid为3334分别进行read和write操作。
6. read操作时手机app会收到数字累计增加。
7. write操作时手机app发送的字符会显示在样机的串口终端上。

6.4.0.1 hfp client 测试步骤

1. 执行：bt_test -i。
2. 手机连接上蓝牙设备 。
3. 来电接听：hfp_answer 。
4. 来电拒绝：hfp_hangup 。
5. 样机拨号：hfp_dial 10001 。
6. 样机拨打上一个电话：hfp_last_num 。
7. 样机获取手机：hfp_cnum。

6.4.1 配置文件

经典蓝牙涉及到的配置文件为 bt_init.sh 和 Bluetooth.json 两个文件，前者主要功能是将蓝牙协议栈带起来，请参考 3.1 章节，而 Bluetooth.json，请参考如下：

```
{
  "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_size":30080,
    "period_size":3760
  }
}
```

```
    },
    "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 条目: 表示默认需要使能的 profile，当用户没有主动调用 bt_manager_enable_profile 使能那些 profile 时，将默认从这个条目进行读取配置，使能那些 profile。
- a2dp_sink 条目: 该条目主要是用 a2dp sink 播放音频相关的配置，device 表示使用的硬件声卡，buffer_size 为对应 alsa 参数的 buffer size，period_size 对应 alsa 参数 period size。
- a2dp_source: 用于 a2sp_source 的配置参数，暂时未用到。
- hfp_pcm: 用于 hfp over pcm 的参数配置，rate 表示蓝牙 pcm 用的采样率，跟蓝牙模组有关；phone_to_dev_cap 表示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 表示我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 蓝牙常见问题排查指南

7.1 排查指南顺序

- 1 根据模组型号确认 3.2 小节的配置正确。
- 2 根据原理图检查 bt 的上电 gpio（包括 reset pin, wake ap, hostwake）、uart 号等。
- 3 检查 bt_init.sh 脚本是否正确。

编译路径：

tina/package/allwinner/btmanager/config/xradio_bt_init.sh

样机路径：

/etc/Bluetooth/bt_init.sh

注意：

如果模组是H4协议："\$bt_hciattach" -n -s 115200 /dev/ttyS1 rtk_h4

如果模组是H5协议："\$bt_hciattach" -n -s 115200 /dev/ttyS1 rtk_h5

- 4 检查 Bluetooth.json 文件，尤其注意声卡选择是否正确，参考 4.6.1

编译路径：

tina/target/allwinner/dl-h-nezha/base-files/etc/Bluetooth/Bluetooth.json

样机路径：

/etc/Bluetooth/Bluetooth.json




著作权声明

版权所有 © 2022 珠海全志科技股份有限公司。保留一切权利。

本文档及内容受著作权法保护，其著作权由珠海全志科技股份有限公司（“全志”）拥有并保留一切权利。

本文档是全志的原创作品和版权财产，未经全志书面许可，任何单位和个人不得擅自摘抄、复制、修改、发表或传播本文档内容的部分或全部，且不得以任何形式传播。

商标声明

、 **全志科技** （不完全列举）均为珠海全志科技股份有限公司的商标或者注册商标。在本文档描述的产品中出现的其它商标，产品名称，和服务名称，均由其各自所有人拥有。

免责声明

您购买的产品、服务或特性应受您与珠海全志科技股份有限公司（“全志”）之间签署的商业合同和条款的约束。本文档中描述的全部或部分产品、服务或特性可能不在您所购买或使用的范围内。使用前请认真阅读合同条款和相关说明，并严格遵循本文档的使用说明。您将自行承担任何不当使用行为（包括但不限于如超压，超频，超温使用）造成的不利后果，全志概不负责。

本文档作为使用指导仅供参考。由于产品版本升级或其他原因，本文档内容有可能修改，如有变更，恕不另行通知。全志尽全力在本文档中提供准确的信息，但并不确保内容完全没有错误，因使用本文档而发生损害（包括但不限于间接的、偶然的、特殊的损失）或发生侵犯第三方权利事件，全志概不负责。本文档中的所有陈述、信息和建议并不构成任何明示或暗示的保证或承诺。

本文档未以明示或暗示或其他方式授予全志的任何专利或知识产权。在您实施方案或使用产品的过程中，可能需要获得第三方的权利许可。请您自行向第三方权利人获取相关的许可。全志不承担也不代为支付任何关于获取第三方许可的许可费或版税（专利税）。全志不对您所使用的第三方许可技术做出任何保证、赔偿或承担其他义务。