XR806 BLE Host 应用开发指南

版本号：1.0

发布时间：2021-02-07

版本历史

版本	日期	责任人	版本描述
1.0	2021-02-07	AWA1426	创建文档

目录

表格目录

1 前言

1.1 文档简介

本文档介绍了 XR806 SDK 的 BLE 接口函数的功能、使用说明及使用示例，以指导读者使用 XR806 BLE 模块提供的接口进行应用的开发。

1.2 目标读者

XR806 BLE 开发相关人员。

1.3 适用范围

此文档适用于 XR806 SDK，支持 XR806 系列芯片产品。

1.4 文档约定

1.4.1 标志说明

本文档采用各种醒目的标志来表示在操作过程中应该特别注意的地方，这些标志的含义如下：

标识	说明
警告	该标志后的说明应给予格外关注，如果不遵守，可能会导致人员受伤或死亡。
注意	提醒操作中应注意的事项。不当的操作可能会损坏器件，影响可靠性、降低性能等。
说明	为准确理解文中指令、正确实施操作而提供的补充或强调信息。
窍门	一些容易忽视的小功能、技巧。了解这些功能或技巧能帮助解决特定问题或者节省操作时间。

2 概述

2.1 背景说明

XR806 通过 Zephyr 协议栈提供 BLE 功能的支持，提供 API 供用户进行调用。API 所提供的功能主要有 BLE 初始化和反初始化、广播打开和关闭、扫描打开和关闭、连接和断开连接等。

2.2 规格特性

XR806 BLE Host 协议栈基于 Zephyr 提供的开源 Bluetooth Host 协议栈进行二次开发，本文撰写期间 XR806 BLE Host 采用 Zephyr V2.3.0 版本。

2.3 文件位置

XR806 BLE 接口属于 BLE Host 提供的 API，其代码位置如表 2-1 所示。

表 2-1 XR806 BLE 接口代码位置

模块	文件类型	代码位置
BLE 接口	header	./include/ble/bluetooth/bluetooth.h
		./include/ble/bluetooth/conn.h
		./include/ble/bluetooth/gatt.h

说明

XR806 SDK 可在以下 Github 仓库获取：https://github.com/XradioTech/xr806_sdk.git

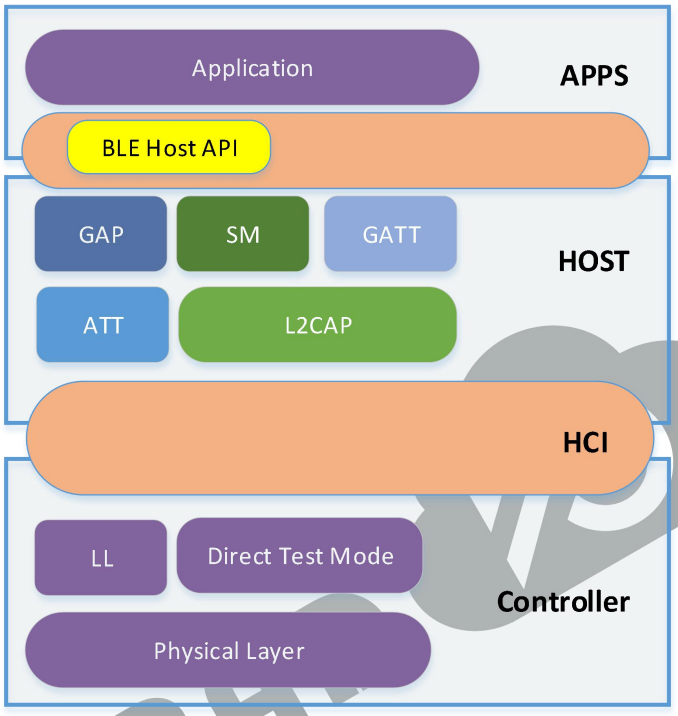
3 技术说明

3.1 模块框架

XR806 BLE 模块的系统层次如下图 3-1 所示。

XR806 BLE 接口位于 BLE 系统的最上层，是 BLE Host 提供给应用的接口。应用通过调用 BLE API，实现完整的 BLE 功能。

图 3-1 BLE 系统层次



3.2 BLE 接口使用流程

使用 XR806 BLE 接口开发 BLE 功能前，需要先进行 BLE 的初始化。接口使用步骤如下：

(1) 使用接口 bt_enable()进行 BLE 初始化，对 BLE 所用到的资源进行配置。

(2) 此后可以通过调用 BLE 功能接口实现各个不同的功能，如通过 bt_le_adv_start()接口启动 BLE 广播。

(3) 需要关闭 BLE 功能时，可以使用 bt_disable()接口。

4 应用说明

4.1 配置说明

启用 BLE 需要使能 BLE 的相关配置，通过工程配置即可启用，如表 4-1 所示。

表 4-1 XR806 BLE 配置列表

配置项	配置说明
BLE 功能使能	设置说明： 此项配置用于在 SDK 中启用 BLE 功能。 设置方法： 编译之前执行命令“make menuconfig”，进入 BLE 功能项下，将以下功能项打开： **[*] BLE Controller** **[*] Ble Host**

4.2 接口说明

XR806 BLE 提供了各个功能的接口，主要功能涵括了初始化和反初始化、广播、扫描、连接等。但需要注意的是，应用可使用的接口还取决于 Kconfig 选项，因为大多数 BLE 功能都是在编译时已经确定好的。例如，任何与连接相关的 API 都需要配置 CONFIG_BT_CONN 宏，其在 menuconfig 中对应的选项为 BT_CONN。

各接口按照功能可分为 4 类，其简要说明如表 4-2 所示。

表 4-2 BLE 接口功能分类说明

接口名	简要说明
基础接口	本接口共有 26 个接口函数，主要用于 BLE 基础功能，如广播、扫描等。
CONN 接口	本接口共有 16 个接口函数，主要用于 BLE 的连接与链路管理等。
SMP 接口	本接口共有 10 个接口函数，主要用于 BLE 的配对绑定。
GATT 接口	本接口共有 16 个接口函数，主要用于 GATT 的读写、订阅通知等。

4.2.1 基础接口

4.2.1.1 bt_enable

表 4-3 bt_enable 接口函数说明

信息项	说明
原型	int bt_enable(bt_ready_cb_t cb);
功能	BLE 的初始化，软硬件的初始化和资源配置等

信息项	说明
参数	bt_ready_cb_t cb 含义解释：BLE 初始化完成的回调函数，由用户实现或者设置为 NULL(若设置为 NULL 则阻塞到初始化完成) 使用说明：请查看表 4-4
返回值	int 类型 0：初始化成功 非 0：初始化失败

表 4-4 bt_ready_cb_t 接口函数说明

信息项	说明
原型	typedef void (*bt_ready_cb_t)(int err);
功能	BLE 初始化完成后的回调函数，由应用具体实现，进行一些应用实现的其他回调函数的注册，如扫描回调函数的注册等
参数	int err 含义解释：BLE 初始化的结果 使用说明： 0：BLE 初始化成功 非 0：初始化失败
返回值	无

4.2.1.2 bt_disable

表 4-5 bt_disable 接口函数说明

信息项	说明
原型	int bt_disable(void);
功能	BLE 的反初始化接口，软硬件的复位和资源释放等，需在 menuconfig 中使能 BT_Deinit 功能
参数	无
返回值	int 类型 0：反初始化成功 非 0：反初始化失败

4.2.1.3 bt_set_name

表 4-6 bt_set_name 接口函数说明

信息项	说明
原型	int bt_set_name(const char *name);

信息项	说明
功能	设置 BLE 设备名称
参数	const char *name 含义解释：将要设置的新的设备名称 使用说明：char 型指针，指向设备名称的首字母，所能支持的设备名称最长长度为 CONFIG_BT_DEVICE_NAME_MAX+1 字节(CONFIG_BT_DEVICE_NAME_MAX 可配置修改)
返回值	int 类型 0：名称设置成功 非 0：名称设置失败

4.2.1.4 bt_get_name

表 4-7 bt_get_name 接口函数说明

信息项	说明
原型	const char *bt_get_name(void);
功能	获取 BLE 当前设备名称
参数	无
返回值	const char * 类型 指向 BLE 设备名称字符串首字母的指针

4.2.1.5 bt_set_id_addr

表 4-8 bt_set_id_addr 接口函数说明

信息项	说明
原型	int bt_set_id_addr(const bt_addr_le_t *addr);
功能	由应用根据指定的 BLE 设备地址产生一个新的本地 ID 地址，注意：此 API 必须在 bt_enable 接口调用之前使用，其他任意时间调用都会失败
参数	const bt_addr_le_t *addr 含义解释：指定的 BLE 设备地址 使用说明：bt_addr_le_t 结构体的说明，请查看表 4-9。注意：当前通过此接口设置的地址必须为 static random 地址，其他地址无效
返回值	int 类型 0：ID 设置成功 非 0：ID 设置失败

表 4-9 bt_addr_le_t 结构体成员说明

成员项	说明
uint8_t type	含义解释：BLE 设备地址类型

成员项	说明
	使用说明：type 为枚举值，具体含义如下 BT_ADDR_LE_PUBLIC：public 地址 BT_ADDR_LE_RANDOM：random 地址 BT_ADDR_LE_PUBLIC_ID：public identity 地址 BT_ADDR_LE_RANDOM_ID：random identity 地址
bt_addr_t a	含义解释：BLE 设备实际地址 使用说明：bt_addr_t 结构体的说明，请查看表 4-10

表 4-10 bt_addr_t 结构体成员说明

成员项	说明
uint8_t val[6]	含义解释：存储 BLE 设备地址的数组 使用说明：BLE 地址形式为 XX:XX:XX:XX:XX:XX，从左到右分别存储到 val 数组的低到高位

4.2.1.6 bt_id_get

表 4-11 bt_id_get 接口函数说明

信息项	说明
原型	void bt_id_get(bt_addr_le_t *addrs, size_t *count);
功能	获取当前 BLE 设备存在的地址
参数	bt_addr_le_t *addrs 含义解释：存储获取到的 BLE 的设备地址 使用说明：当前 BLE 的设备地址可能存在多个，该接口会获取到所有的设备地址，所以一般需要以 bt_addr_le_t 结构体(结构体说明见表 4-9)数组传入，大小为 CONFIG_BT_ID_MAX 配置的大小 size_t *count 含义解释：当前 BLE 存在的设备地址个数 使用说明：需要应用初始化为结构体数组元素个数，接口函数返回后该参数为 BLE 实际存在的设备地址个数
返回值	无

4.2.1.7 bt_id_create

表 4-12 bt_id_create 接口函数说明

信息项	说明
原型	int bt_id_create(bt_addr_le_t *addr, uint8_t *irk);
功能	创建一个新的 BLE 设备地址 ID。注意：在 bt_enable 之前调用此接口函数，则可以重写 Controller 中的地址(如果存在)。如果希望产生随机地址，并将其存储到 flash 中，则需要先执行 bt_enable，然后调用 settings_load，然后使用 bt_id_get 获取当前存在地址的个数，若还有空间创建新的地址，则可调用 bt_id_create 创建新的地址
参数	bt_addr_le_t *addrs 含义解释：需要产生的新的 BLE 设备地址 使用说明：结构体说明见表 4-9，可为设置值或是 NULL，如果为 NULL 或是初始化为 BT_ADDR_LE_ANY，则将产生一个新的 static random 地址，并在函数返回时存放在 addr 中(如果 addrs 为非 NULL) uint8_t *irk 含义解释：16 字节身份解析密钥指针 使用说明：与创建的 ID 相绑定的 IRK，如果设置为全零或 NULL，则 BLE 会产生一个随机 IRK，并在函数返回时将 irk 指针指向该密钥(如果 irk 为非 NULL)。如果未开启 Privacy 功能(CONFIG_BT_PRIVACY)，则该参数必须为 NULL
返回值	int 类型 >=0：ID 创建成功 <0：ID 创建失败

4.2.1.8 bt_id_reset

表 4-13 bt_id_reset 接口函数说明

信息项	说明
原型	int bt_id_reset(uint8_t id, bt_addr_le_t *addr, uint8_t *irk);
功能	Reset ID 供重用，注意：使用该函数后将断开使用该 ID 所创建的任何连接链路，删除对应的绑定的配对密钥或其他数据，然后根据 addr 和 irk 参数在此位置中创建一个新的 ID
参数	uint8_t id 含义解释：现有 ID 标识 使用说明：该参数必须是现有的 ID 号，如果不存在则函数将返回错误 bt_addr_le_t *addrs 含义解释：需要重新设置的 BLE 的设备地址 使用说明：结构体说明见表 4-9，可为设置值或是 NULL，如果为 NULL 或是初始化为 BT_ADDR_LE_ANY，则将产生一个新的 static random 地址，并在函数返回时存放在 addr 中(如果 addrs 为非 NULL)

信息项	说明
	uint8_t *irk 含义解释：16 字节身份解析密钥指针 使用说明：与创建的 id 相绑定的 IRK，如果设置为全零或 NULL，则 BLE 会产生一个随机 IRK，并在函数返回时将 irk 指针指向该密钥(如果 irk 为非 NULL)。如果未开启 Privacy 功能(CONFIG_BT_PRIVACY)，则该参数必须为 NULL
返回值	int 类型 >=0：重置并创建成功 <0：重置并创建失败

4.2.1.9 bt_id_delete

表 4-14 bt_id_delete 接口函数说明

信息项	说明
原型	int bt_id_delete(uint8_t id);
功能	删除指定的 ID，注意：使用该函数后将断开使用该 ID 所创建的任何连接链路，删除对应的绑定的配对密钥或其他数据
参数	uint8_t id 含义解释：已存在的 ID 使用说明：参数必须为当前已存在或添加过的 ID
返回值	int 类型 0：删除成功 非 0：删除失败

4.2.1.10 bt_le_adv_start

表 4-15 bt_le_adv_start 接口函数说明

信息项	说明
原型	int bt_le_adv_start(const struct bt_le_adv_param *param, const struct bt_data *ad, size_t ad_len, const struct bt_data *sd, size_t sd_len);
功能	使能 BLE 广播，主要设置广播数据，scan rsp 数据和扫描参数
参数	const struct bt_le_adv_param *param 含义解释：存储广播所使用的各个参数，如广播周期等 使用说明：bt_le_adv_param 结构体具体见表 4-16 const struct bt_data *ad 含义解释：广播包中的广播数据 使用说明：bt_data 结构体具体见表 4-17 size_t ad_len 含义解释：广播数据中的元素数量

信息项	说明
	使用说明：ad_len 不是广播数据整体长度大小，而是其中存在的元素个数 const struct bt_data *sd 含义解释：scan rsp 包中的数据 使用说明：bt_data 结构体具体见表 4-17 size_t sd_len 含义解释：scan rsp 数据中的元素数量 使用说明：同 ad_len 一致
返回值	int 类型 0：启动成功 非 0：启动失败

表 4-16 bt_le_adv_param 结构体成员说明

成员项	说明
uint8_t id	含义解释：广播对应的 ID 使用说明：不支持一个 BLE 设备同时存在多条广播链路
uint8_t sid	含义解释：扩展广播对应的 ID 使用说明：需使能 BT_LE_ADV_OPT_EXT_ADV 才可使用
uint8_t secondary_max_skip	含义解释：两次在辅助广播信道上发送广播数据之间允许的最大间隔 使用说明：需使能 BT_LE_ADV_OPT_EXT_ADV 才可使用
uint32_t options	含义解释：广播特性的位域 使用说明：每个 bit 对应广播的一个特性，置起则表明当前广播正在使用该 bit 对应特性
uint32_t interval_min	含义解释：最小广播事件间隔 使用说明：实际的最小广播事件间隔时间(单位:ms)=interval_min*0.625
uint32_t interval_max	含义解释：最大广播事件间隔 使用说明：计算方法同 interval_min
const bt_addr_le_t *peer	含义解释：记录对端地址 使用说明：当该参数被设置时，则广播数据会定向发送到指定地址，bt_addr_le_t 结构体说明见表 4-9

表 4-17 bt_data 结构体成员说明

成员项	说明
uint8_t type	含义解释：数据元素类型 使用说明：取值范围从 0x00~0xff，每一个值对应一种数据元素，具体见 Spec
uint8_t data_len	含义解释：当前数据元素长度 使用说明：通过 sizeof 计算出的当前数据元素的长度
const uint8_t *data	含义解释：实际数据指针 使用说明：指向实际数据存储位置首地址的指针

4.2.1.11 bt_le_adv_update_data

表 4-18 bt_le_adv_update_data 接口函数说明

信息项	说明
原型	int bt_le_adv_update_data(const struct bt_data *ad, size_t ad_len, const struct bt_data *sd, size_t sd_len);
功能	广播和 scan rsp 数据更新
参数	const struct bt_data *ad 含义解释：广播包中的广播数据 使用说明：bt_data 结构体具体见表 4-17 size_t ad_len 含义解释：广播数据中的元素数量 使用说明：ad_len 不是广播数据整体长度大小，而是其中存在的元素个数 const struct bt_data *sd 含义解释：scan rsp 包中的数据 使用说明：bt_data 结构体具体见表 4-17 size_t sd_len 含义解释：scan rsp 数据中的元素数量 使用说明：同 ad_len 一致
返回值	int 类型 0：更新成功 非 0：更新失败

4.2.1.12 bt_le_adv_stop

表 4-19 bt_le_adv_stop 接口函数说明

信息项	说明
原型	int bt_le_adv_stop(void);
功能	关闭 BLE 广播
参数	无
返回值	int 类型 0：更新成功 非 0：更新失败

4.2.1.13 bt_le_scan_start

表 4-20 bt_le_scan_start 接口函数说明

信息项	说明
原型	int bt_le_scan_start(const struct bt_le_scan_param *param, bt_le_scan_cb_t cb);
功能	使用给定的参数开始 BLE 扫描，并通过指定的回调函数上传扫描结果。在禁用 CONFIG_BT_PRIVACY 情况下，为了防止主动扫描设备在向广播设备请求其他信息时暴露自身信息，默认情况下 BLE 扫描设备不适用本设备的 ID 地址。只有当启动了 CONFIG_BT_SCAN_WITH_IDENTITY 选项时才会上报定向广播
参数	const struct bt_le_scan_param *param 含义解释：存储 BLE 扫描所使用的各个参数，如扫描窗口和扫描间隔等 使用说明：用该参数来配置 BLE 设备的扫描行为，具体说明见表 4-21 bt_le_scan_cb_t cb 含义解释：扫描结果上报回调函数 使用说明：相关描述见表 4-22，如果回调函数是通过 bt_le_scan_cb_register 进行注册的，则可能为 NULL
返回值	int 类型 0：启动成功 非 0：启动失败

表 4-21 bt_le_scan_param 结构体成员说明

成员项	说明
uint8_t type	含义解释：扫描类型 使用说明：type 为枚举类型变量，具体含义如下 BT_LE_SCAN_TYPE_PASSIVE(被动扫描) (主动扫描)BT_LE_SCAN_TYPE_ACTIVE=0x1
uint32_t options	含义解释：扫描特性的位域

成员项	说明
	使用说明：每个 bit 对应扫描的一个特性，置起则表明当前扫描正在使用该特性
uint16_t interval	含义解释：两次扫描间隔 使用说明：实际扫描间隔(单位:ms)=interval * 0.625
uint16_t window	含义解释：扫描窗口 使用说明：计算方式同 interval
uint16_t timeout	含义解释：扫描超时时间 使用说明：实际扫描超时时间(单位:ms)=timeout * 10。扫描超时将通过回调函数通知应用，设置为 0 时禁用超时
uint16_t interval_coded	含义解释：BLE Code PHY 编码方式下两次扫描间隔 使用说明：计算方式同 interval，设置为 0 以使用与 BLE 1M PHY 相同的扫描间隔
uint16_t window_coded	含义解释：BLE Code PHY 编码方式下扫描窗口 使用说明：计算方式同 interval，设置为 0 以使用与 BLE 1M PHY 相同的扫描窗口

表 4-22 bt_le_scan_cb_t 接口函数说明

信息项	说明
原型	typedef void bt_le_scan_cb_t(const bt_addr_le_t *addr, int8_t rssi, uint8_t adv_type, struct net_buf_simple *buf);
功能	BLE 扫描回调函数，用于上报 BLE 扫描结果，由应用具体实现，例如根据扫描结果选取某个特定设备进行连接。传入到 bt_le_scan_start 函数中，扫描到任意 BLE 设备时调用
参数	const bt_addr_le_t *addr 含义解释：扫描到的 BLE 设备地址 使用说明：从该参数中可以获取到当前扫描到的 BLE 设备地址 int8_t rssi 含义解释：扫描到的 BLE 设备的 rssi 值 使用说明：可通过该值来判断接收到的 BLE 广播信号的强弱 uint8_t adv_type 含义解释：接收到的广播包的类型 使用说明：adv_type 为枚举类型变量，具体含义如下 BT_GAP_ADV_TYPE_ADV_IND：普通广播 BT_GAP_ADV_TYPE_ADV_DIRECT_IND：定向广播

信息项	说明
	BT_GAP_ADV_TYPE_ADV_SCAN_IND：可扫描不可连接广播 BT_GAP_ADV_TYPE_ADV_NONCONN_IND：不可扫描不可连接广播 BT_GAP_ADV_TYPE_SCAN_RSP：Scan Response 包 BT_GAP_ADV_TYPE_EXT_ADV：扩展广播包 struct net_buf_simple *buf 含义解释：存储广播数据的 buffer 使用说明：指针指向的是广播数据的首地址，从该指针可获取到实际的广播数据
返回值	无

表 4-23 net_buf_simple 结构体成员说明

成员项	说明
u8_t *data	含义解释：指向 buffer 中数据开头的指针 使用说明：通过该指针访问实际的广播数据，而不是直接操作广播数据
u16_t len	含义解释：数据指针后面的数据长度 使用说明：buffer 中实际存在数据的大小，取出或放出数据后会动态变化
u16_t size	含义解释：该 buffer 可存储的数据量 使用说明：该值在初始化时已经确认，不需进行修改
u8_t *__buf	含义解释：数据存储的开头 使用说明：不能直接访问，应该使用数据指针进行访问

4.2.1.14 bt_le_scan_stop

表 4-24 bt_le_scan_stop 接口函数说明

信息项	说明
原型	int bt_le_scan_stop(void);
功能	关闭正在进行的扫描操作
参数	无
返回值	int 类型 0：关闭成功 非 0：关闭失败

4.2.1.15 bt_le_scan_cb_register

表 4-25 bt_le_scan_cb_register 接口函数说明

信息项	说明
原型	void bt_le_scan_cb_register(struct bt_le_scan_cb *cb);
功能	注册扫描回调函数到 BLE 协议栈中
参数	struct bt_le_scan_cb *cb 含义解释：BLE 扫描回调函数 使用说明：由应用进行实现，具体如表 4-22 所示
返回值	无

4.2.1.16 bt_le_scan_cb_unregister

表 4-26 bt_le_scan_cb_unregister 接口函数说明

信息项	说明
原型	void bt_le_scan_cb_unregister(struct bt_le_scan_cb *cb);
功能	将注册的扫描回调函数从 BLE 协议栈中去除
参数	struct bt_le_scan_cb *cb 含义解释：BLE 扫描回调函数 使用说明：在 BLE 关闭时使用，由用户进行调用，具体如表 4-22 所示
返回值	无

4.2.1.17 bt_le_whitelist_add

表 4-27 bt_le_whitelist_add 接口函数说明

信息项	说明
原型	int bt_le_whitelist_add(const bt_addr_le_t *addr);
功能	添加对端 BLE 设备地址到白名单中，在 BLE 设备使用白名单的过程中不允许修改白名单
参数	const bt_addr_le_t *addr 含义解释：对端 BLE 设备地址 使用说明：需要添加到白名单中的 BLE 设备地址，结构体说明见表 4-9
返回值	int 类型 0：关闭成功 非 0：关闭失败

4.2.1.18 bt_le_whitelist_rem

表 4-28 bt_le_whitelist_rem 接口函数说明

信息项	说明
原型	int bt_le_whitelist_rem(const bt_addr_le_t *addr);
功能	从白名单中移除对端 BLE 设备地址，在 BLE 设备使用白名单的过程中不允许修改白名单
参数	const bt_addr_le_t *addr 含义解释：对端 BLE 设备地址 使用说明：需要从白名单中移除的 BLE 设备地址，结构体说明见表 4-9
返回值	int 类型 0：移除成功 非 0：移除失败

4.2.1.19 bt_le_whitelist_clear

表 4-29 bt_le_whitelist_clear 接口函数说明

信息项	说明
原型	int bt_le_whitelist_clear(void);
功能	从白名单中移除所有 BLE 设备地址，在 BLE 设备使用白名单的过程中不允许修改白名单
参数	无
返回值	int 类型 0：清除成功 非 0：清除失败

4.2.1.20 bt_le_set_chan_map

表 4-30 bt_le_set_chan_map 接口函数说明

信息项	说明
原型	int bt_le_set_chan_map(uint8_t chan_map[5]);
功能	设置 BLE channel map
参数	uint8_t chan_map[5] 含义解释：BLE 的 Channle map 使用说明：每个 bit 代表一个 BLE 的 channel，从低位到高位分别对应 0~39 信道，bit 为 0 表示该信道不可用，bit 为 1 表示该信道可用
返回值	int 类型 0：设置成功 非 0：设置失败

4.2.1.21 bt_data_parse

表 4-31 bt_data_parse 接口函数说明

信息项	说明
原型	void bt_data_parse(struct net_buf_simple *ad, bool (*func)(struct bt_data *data, void *user_data), void *user_data);
功能	广播/EIR/OOB 数据的解析函数，一般在 bt_le_scan_cb_t 回调函数中对收到的广播数据进行解析
参数	struct net_buf_simple *ad 含义解释：需要解析的数据 使用说明：一般为收到的广播数据，结构体说明见表 4-21 bool (*func)(struct bt_data *data, void *user_data) 含义解释：回调函数 使用说明：解析函数找到广播数据中的各个元素，然后调用该回调函数对解析出的元素进行处理，回调返回 true 则继续解析，返回 false 则停止解析 void *user_data 含义解释：回调传入参数 使用说明：可能需要传入到回调 func 中的参数
返回值	无

4.2.1.22 bt_le_oob_get_local

表 4-32 bt_le_oob_get_local 接口函数说明

信息项	说明
原型	int bt_le_oob_get_local(uint8_t id, struct bt_le_oob *oob);
功能	获取本地 BLE 的 OOB 信息，用于后续带外配对或带外连接创建
参数	uint8_t id 含义解释：身份标识 使用说明：一般使用 BT_ID_DEFAULT struct bt_le_oob *oob 含义解释：BLE 的 OOB 信息 使用说明：bt_le_oob 结构体说明见表 4-33，应用直接进行使用
返回值	int 类型 0：获取成功 非 0：获取失败

表 4-33 bt_le_oob 结构体成员说明

成员项	说明
bt_addr_le_t addr	含义解释：本地 BLE 设备地址 使用说明：结构体说明见表 4-9，如果开启了 privacy 功能，则存储的是可解析的私有地址
struct bt_le_oob_sc_data le_sc_data	含义解释：BLE 安全连接配对带外数据 使用说明：bt_le_oob_sc_data 结构体说明见表 4-34

表 4-34 bt_le_oob_sc_data 结构体成员说明

成员项	说明
uint8_t r[16]	含义解释：随机参数 使用说明：共 128bit，由 BLE 协议栈计算产生，应用可直接获取使用
uint8_t c[16]	含义解释：确认参数 使用说明：共 128bit，由 BLE 协议栈计算产生，应用可直接获取使用

4.2.1.23 bt_addr_le_to_str

表 4-35 bt_addr_le_to_str 接口函数说明

信息项	说明
原型	static inline int bt_addr_le_to_str(const bt_addr_t *addr, char *str, size_t len);
功能	将二进制 BLE 设备地址转换成字符串输出
参数	const bt_addr_le_t *addr 含义解释：包含 BLE 二进制地址的 buffer 使用说明：一般存储形式即为二进制，只有在应用打印时才会将二进制转换为字符串，一般仅用于打印，结构体说明见表 4-9 char *str 含义解释：字符串 buffer 使用说明：用于存储地址转后的字符串，需要有足够的空间来存储格式化后的字符串 size_t len 含义解释：字符串长度 使用说明：该参数为复制到字符串 buffer 中数据长度，一般值为 BT_ADDR_STR_LEN
返回值	int 类型 从二进制地址成功转换到字符串的字节数

4.2.1.24 bt_addr_le_from_str

表 4-36 bt_addr_le_from_str 接口函数说明

信息项	说明
原型	int bt_addr_le_from_str(const char *str, const char *type, bt_addr_le_t *addr);
功能	将字符串 BLE 设备地址转换成二进制地址
参数	const char *str 含义解释：字符串 buffer 使用说明：用于存储地址转换前的字符串，一般在命令行输入时使用 const char *type 含义解释：BLE 设备地址类型 使用说明：为字符串形式，一般在命令行输入时使用 const bt_addr_le_t *addr 含义解释：包含 BLE 二进制地址的 buffer 使用说明：存储从字符串转换为二进制格式的设备地址，结构体说明见表 4-9
返回值	int 类型 0：转换成功 非 0：转换失败

4.2.1.25 bt_unpair

表 4-37 bt_unpair 接口函数说明

信息项	说明
原型	int bt_unpair(uint8_t id, const bt_addr_le_t *addr);
功能	清除已配对信息
参数	uint8_t id 含义解释：身份标识 使用说明：一般使用 BT_ID_DEFAULT const bt_addr_le_t *addr 含义解释：对端 BLE 设备地址 使用说明：一般为绑定后的对端设备地址，若为 NULL 或 BT_ADDR_LE_ANY 则清除所有本机存储的对端设备的绑定信息
返回值	int 类型 0：清除成功 非 0：清除失败

4.2.1.26 bt_foreach_bond

表 4-38 bt_foreach_bond 接口函数说明

信息项	说明
原型	void bt_foreach_bond(uint8_t id, void (*func)(const struct bt_bond_info *info, void *user_data), void *user_data);
功能	遍历所有设备当前存储的绑定信息
参数	uint8_t id 含义解释：身份标识 使用说明：一般使用 BT_ID_DEFAULT void (*func)(const struct bt_bond_info *info, void *user_data) 含义解释：绑定处理回调函数 使用说明：遍历中每查询到一个设备绑定信息，都会调用此回调函数进行处理，由应用进行实现 void *user_data 含义解释：回调传入参数 使用说明：可能需要传入到回调 func 中的参数
返回值	无

4.2.2 CONN 接口

4.2.2.1 bt_conn_ref

表 4-39 bt_conn_ref 接口函数说明

信息项	说明
原型	struct bt_conn *bt_conn_ref(struct bt_conn *conn);
功能	增加连接链路的引用计数
参数	struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中
返回值	struct bt_conn * 含义解释：连接链路指针 使用说明：引用计数增加后的连接链路

4.2.2.2 bt_conn_unref

表 4-40 bt_conn_unref 接口函数说明

信息项	说明
原型	void bt_conn_unref(struct bt_conn *conn);
功能	减少连接链路的引用计数

信息项	说明
参数	struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中
返回值	无

4.2.2.3 bt_conn_foreach

表 4-41 bt_conn_foreach 接口函数说明

信息项	说明
原型	void bt_conn_foreach(int type, void (*func)(struct bt_conn *conn, void *data), void *data);
功能	遍历当前设备存在的连接链路
参数	int type 含义解释：连接类型 使用说明：type 为枚举型变量，其取值为 BT_CONN_TYPE_LE：BLE 类型 BT_CONN_TYPE_BR：BR/EDR 类型 BT_CONN_TYPE_SCO：SCO 类型 BT_CONN_TYPE_ALL：上述三种类型均包含 void (*func)(struct bt_conn *conn, void *data) 含义解释：连接链路处理回调函数 使用说明：遍历时每查询到一条连接链路，都会调用此回调函数进行处理，由应用进行实现 void *user_data 含义解释：回调传入参数 使用说明：可能需要传入到回调 func 中的参数
返回值	无

4.2.2.4 bt_conn_lookup_addr_le

表 4-42 bt_conn_lookup_addr_le 接口函数说明

信息项	说明
原型	struct bt_conn *bt_conn_lookup_addr_le(uint8_t id, const bt_addr_le_t *peer);
功能	根据对端地址遍历现有连接链路进行查找
参数	uint8_t id 含义解释：身份标识 使用说明：一般使用 BT_ID_DEFAULT(该值可配置) const bt_addr_le_t *peer

信息项	说明
	含义解释：对端 BLE 设备地址 使用说明：需要查找连接链路的地址，结构体说明见表 4-9
返回值	struct bt_conn * NULL：未找到对应连接链路 非 NULL：查找到的连接链路

4.2.2.5 bt_conn_get_dst

表 4-43 bt_conn_get_dst 接口函数说明

信息项	说明
原型	const bt_addr_le_t *bt_conn_get_dst(const struct bt_conn *conn);
功能	根据特定连接链路获取对端 BLE 设备地址
参数	struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中
返回值	const bt_addr_le_t * 根据连接链路查到到的对端 BLE 设备地址

4.2.2.6 bt_conn_get_info

表 4-44 bt_conn_get_info 接口函数说明

信息项	说明
原型	int bt_conn_get_info(const struct bt_conn *conn, struct bt_conn_info *info);
功能	获取连接链路的信息
参数	const struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中 struct bt_conn_info *info 含义解释：连接链路信息 使用说明：存储获取到的连接链路信息，结构体说明见表 4-45
返回值	int 类型 0：获取成功 非 0：获取失败

表 4-45 bt_conn_info 结构体成员说明

成员项	说明
uint8_t type	含义解释：连接类型

成员项	说明
	使用说明：type 为枚举型变量，其取值为 BT_CONN_TYPE_LE：BLE 类型 BT_CONN_TYPE_BR：BR/EDR 类型 BT_CONN_TYPE_SCO：SCO 类型 BT_CONN_TYPE_ALL：上述三种类型均包含
uint8_t role	含义解释：本机设备的连接链路角色 使用说明：存在两种角色，分别为： (Master 角色)BT_HCI_ROLE_MASTER=0x00 (Slave 角色)BT_HCI_ROLE_SLAVE=0x01
uint8_t id	含义解释：本地 BLE 的 ID 使用说明：通过该 ID 确认使用哪个设备地址创建的连接链路
struct bt_conn_le_info le	含义解释：BLE 连接特殊信息 使用说明：BLE 连接相关的特性信息，结构体说明见表 4-46

表 4-46 bt_conn_le_info 结构体成员说明

成员项	说明
const bt_addr_le_t *src	含义解释：本地 BLE 设备地址 使用说明：存储的本地设备地址，通过该地址和对端设备创建连接，在 RPA 时会被修改
const bt_addr_le_t *dst	含义解释：对端 BLE 设备地址 使用说明：连接链路对端 BLE 设备的地址，若对端为私有地址则存储的是已解析后的地址，在 RPA 时会被修改
const bt_addr_le_t *local	含义解释：创建连接时所使用的地址 使用说明：存储创建连接时所使用的本地 BLE 设备地址
const bt_addr_le_t *remote	含义解释：创建连接时所使用的对端地址 使用说明：存储创建连接时所使用的对端 BLE 设备地址
uint16_t interval	含义解释：连接间隔 使用说明：连接链路的连接事件间隔
uint16_t latency	含义解释：允许跳过的事件个数 使用说明：允许 slave 跳过接收的事件个数
uint16_t timeout	含义解释：超时时间

成员项	说明
	使用说明：连接链路的超时断开连接时间

4.2.2.7 bt_conn_le_param_update

表 4-47 bt_conn_le_param_update 接口函数说明

信息项	说明
原型	int bt_conn_le_param_update(struct bt_conn *conn, const struct bt_le_conn_param *param);
功能	更新连接链路参数
参数	const struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中 const struct bt_le_conn_param *param 含义解释：连接链路参数 使用说明：需要更新的连接链路参数存储在参数中，结构体说明见表 4-45
返回值	int 类型 0：获取成功 非 0：获取失败

表 4-48 bt_le_conn_param 结构体成员说明

成员项	说明
uint16_t interval_min	含义解释：连接事件最小间隔 使用说明：实际最小连接事件间隔时间(单位:ms)=interval_min * 1.25
uint16_t interval_max	含义解释：连接事件最大间隔 使用说明：实际最大连接事件间隔时间(单位:ms)=interval_max * 1.25
uint16_t latency	含义解释：允许跳过的事件个数 使用说明：允许 slave 跳过接收的事件个数
uint16_t timeout	含义解释：超时时间 使用说明：连接链路的超时断开连接时间(单位:ms)=timeout * 10

4.2.2.8 bt_conn_le_data_len_update

表 4-49 bt_conn_le_data_len_update 接口函数说明

信息项	说明
原型	int bt_conn_le_data_len_update(struct bt_conn *conn, const struct bt_conn_le_data_len_param *param);
功能	更新连接链路上发送数据包的最大大小
参数	const struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中 const struct bt_conn_le_data_len_param *param 含义解释：连接链路数据包长度参数 使用说明：需要更新的连接链路数据包长度存储在参数中，结构体说明见表 4-50
返回值	int 类型 0：更新成功 非 0：更新失败

表 4-50 bt_conn_le_data_len_param 结构体成员说明

成员项	说明
uint16_t tx_max_len	含义解释：最大 tx 传输 payload 长度 使用说明：单位为 bytes
uint16_t tx_max_time	含义解释：最大 tx 传输 payload 时间 使用说明：单位为 us

4.2.2.9 bt_conn_le_phy_update

表 4-51 bt_conn_le_phy_update 接口函数说明

信息项	说明
原型	int bt_conn_le_phy_update(struct bt_conn *conn, const struct bt_conn_le_phy_param *param);
功能	更新连接链路上 PHY 参数
参数	const struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中 const struct bt_conn_le_phy_param *param 含义解释：连接链路 PHY 参数 使用说明：需要更新的连接链路 PHY 存储在参数中，结构体说明见表 4-52
返回值	int 类型

信息项	说明
	0：更新成功 非 0：更新失败

表 4-52 bt_conn_le_phy_param 结构体成员说明

成员项	说明
uint16_t options	含义解释：连接链路 PHY 的参数 使用说明：为枚举变量，其取值为： BT_CONN_LE_PHY_OPT_NONE：未指定 PHY 编码类型 BT_CONN_LE_PHY_OPT_CODED_S2：用 S=2 方式编码数据包 BT_CONN_LE_PHY_OPT_CODED_S8：用 S=8 方式编码数据包
uint8_t pref_tx_phy	含义解释：首选发送使用的 PHY 类型的掩码 使用说明：为枚举变量，其取值为： BT_GAP_LE_PHY_NONE：不设置 PHY 时使用 BT_GAP_LE_PHY_1M：PHY 使用 1M 速率 BT_GAP_LE_PHY_2M：PHY 使用 2M 速率 BT_GAP_LE_PHY_CODED：PHY 使用 CODED 方式
uint8_t pref_rx_phy	含义解释：首选接收使用的 PHY 类型的掩码 使用说明：为枚举变量，其取值为： BT_GAP_LE_PHY_NONE：不设置 PHY 时使用 BT_GAP_LE_PHY_1M：PHY 使用 1M 速率 BT_GAP_LE_PHY_2M：PHY 使用 2M 速率 BT_GAP_LE_PHY_CODED：PHY 使用 CODED 方式

4.2.2.10 bt_conn_disconnect

表 4-53 bt_conn_disconnect 接口函数说明

信息项	说明
原型	int bt_conn_disconnect(struct bt_conn *conn, uint8_t reason);
功能	断开指定的连接链路
参数	const struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中 uint8_t reason 含义解释：断开连接原因 使用说明：连接断开可能因为不同原因，如主动断开连接一般设置为 BT_HCI_ERR_REMOTE_USER_TERM_CONN

信息项	说明
返回值	int 类型 0：断开连接成功 非 0：断开连接失败

4.2.2.11 bt_conn_le_create

表 4-54 bt_conn_le_create 接口函数说明

信息项	说明
原型	int bt_conn_le_create(const bt_addr_le_t *peer, const struct bt_conn_le_create_param *create_param, const struct bt_le_conn_param *conn_param, struct bt_conn **conn);
功能	与指定对端 BLE 设备创建连接链路
参数	const bt_addr_le_t *peer 含义解释：对端 BLE 设备地址 使用说明：需要进行连接的对端 BLE 设备 const struct bt_conn_le_create_param *create_param 含义解释：连接链路参数 使用说明：使用该参数进行连接链路的创建协商，结构体说明见表 4-55 const struct bt_le_conn_param *conn_param 含义解释：连接链路参数 使用说明：初始化的连接链路参数，结构体说明见表 4-48 struct bt_conn **conn 含义解释：连接链路 使用说明：创建完成后的连接链路信息存储在该参数中
返回值	int 类型 0：创建连接成功 非 0：创建连接失败

表 4-55 bt_conn_le_create_param 结构体成员说明

成员项	说明
uint32_t options	含义解释：连接链路创建位域 使用说明：为枚举变量，其取值为： BT_CONN_LE_OPT_NONE：无 options 指定时使用 BT_CONN_LE_OPT_CODED：以 Coded PHY 方式进行扫描 BT_CONN_LE_OPT_NO_1M：关闭 1M PHY 方式进行扫描
uint16_t interval	含义解释：扫描事件间隔

成员项	说明
	使用说明：实际扫描事件间隔时间计算(单位:ms)=interval * 0.625
uint16_t window	含义解释：扫描事件窗口 使用说明：实际扫描事件窗口时间计算(单位:ms)=window* 0.625
uint16_t interval_coded	含义解释：BLE Code PHY 编码方式下两次扫描间隔 使用说明：计算方式同 interval，设置为 0 以使用与 BLE 1M PHY 相同的扫描间隔
uint16_t window_coded	含义解释：BLE Code PHY 编码方式下扫描窗口 使用说明：计算方式同 interval，设置为 0 以使用与 BLE 1M PHY 相同的扫描窗口
uint16_t timeout	含义解释：超时时间 使用说明：连接链路的超时断开连接时间(单位:ms)=timeout * 10

4.2.2.12 bt_conn_le_create_auto

表 4-56 bt_conn_le_create_auto 接口函数说明

信息项	说明
原型	int bt_conn_le_create_auto(const struct bt_conn_le_create_param *create_param, const struct bt_le_conn_param *conn_param);
功能	自动连接白名单中的对端 BLE 设备
参数	const struct bt_conn_le_create_param *create_param 含义解释：连接链路参数 使用说明：使用该参数进行连接链路的创建协商，结构体说明见表 4-55 const struct bt_le_conn_param *conn_param 含义解释：连接链路参数 使用说明：初始化的连接链路参数，结构体说明见表 4-48
返回值	int 类型 0：创建连接成功 非 0：创建连接失败

4.2.2.13 bt_conn_create_auto_stop

表 4-57 bt_conn_create_auto_stop 接口函数说明

信息项	说明
原型	int bt_conn_create_auto_stop(void);
功能	停止自动连接

信息项	说明
参数	无
返回值	int 类型 0：停止自动连接成功 非 0：停止自动连接失败

4.2.2.14 bt_le_set_auto_conn

表 4-58 bt_le_set_auto_conn 接口函数说明

信息项	说明
原型	int bt_le_set_auto_conn(const bt_addr_le_t *addr, const struct bt_le_conn_param *param);
功能	自动连接到指定的对端 BLE 设备，启用此功能后，每次 BLE 设备断开与对端设备的连接时，若收到对端 BLE 设备的广播包，则将重新建立连接。在主动扫描期间需要禁止此功能。
参数	const bt_addr_le_t *addr 含义解释：对端 BLE 设备地址 使用说明：需要进行连接的对端 BLE 设备 const struct bt_le_conn_param *conn_param 含义解释：连接链路参数 使用说明：初始化的连接链路参数，结构体说明见表 4-48
返回值	int 类型 0：设置自动连接成功 非 0：设置自动连接失败

4.2.2.15 bt_conn_cb_register

表 4-59 bt_conn_cb_register 接口函数说明

信息项	说明
原型	void bt_conn_cb_register(struct bt_conn_cb *cb);
功能	注册连接回调函数
参数	struct bt_conn_cb *cb 含义解释：连接回调函数集 使用说明：由应用进行实现，结构体说明见表 4-60
返回值	无

表 4-60 bt_conn_cb 结构体成员说明

成员项	说明
void (*connected)(struct bt_conn *conn, uint8_t err)	含义解释：连接完成回调函数

成员项	说明
	使用说明：由用户实现，用于在连接完成后回调提示应用进行下一步操作
void (*disconnected)(struct bt_conn *conn, uint8_t reason);	含义解释：断开连接回调函数 使用说明：由用户实现，用于在断开连接完成后回调提示应用进行下一步操作
bool (*le_param_req)(struct bt_conn *conn, struct bt_le_conn_param *param);	含义解释：BLE 连接参数更新请求回调函数 使用说明：由应用实现，在收到对端 BLE 设备的连接参数跟新请求时进行回调调用。应用通过返回 true 来接受参数，通过返回 false 来拒绝参数
void (*le_param_updated)(struct bt_conn *conn, uint16_t interval, uint16_t latency, uint16_t timeout);	含义解释：BLE 连接参数已更新回调函数 使用说明：此回调通知应用 BLE 的连接参数已完成更新
void (*identity_resolved)(struct bt_conn *conn, const bt_addr_le_t *rpa, const bt_addr_le_t *identity);	含义解释：对端 BLE 的 ID 地址已解析回调函数 使用说明：此回调函数通知应用已完成对端 BLE 设备的 ID 地址解析
void (*security_changed)(struct bt_conn *conn, bt_security_t level, enum bt_security_err err);	含义解释：连接安全等级已更改回调函数 使用说明：此回调函数通知应用连接的安全等级已完成更改
void (*remote_info_available)(struct bt_conn *conn, struct bt_conn_remote_info *remote_info);	含义解释：已获取对端信息回调函数 使用说明：此回调函数通知应用已获取到对端 BLE 设备的信息
void (*le_phy_updated)(struct bt_conn *conn, struct bt_conn_le_phy_info *param);	含义解释：PHY 参数更新回调函数 使用说明：此回调函数通知应用已完成 PHY 参数的更新
void (*le_data_len_updated)(struct bt_conn *conn, struct bt_conn_le_data_len_info *info)	含义解释：数据长度参数更新回调函数 使用说明：此回调函数通知应用已完成数据长度参数的更新
struct bt_conn_cb *_next	含义解释：下一个连接回调函数集 使用说明：为了简单起见，最好只有一个回调函数集。如果存在多个回调函数集，则需要对每个回调函数集潜在要求进行处理

4.2.2.16 bt_conn_cb_unregister

表 4-61 bt_conn_cb_unregister 接口函数说明

信息项	说明
原型	void bt_conn_cb_unregister(struct bt_conn_cb *cb);
功能	注销连接回调函数
参数	struct bt_conn_cb *cb 含义解释：连接回调函数集 使用说明：由应用进行实现，结构体说明见表 4-60
返回值	无

4.2.3 SMP 接口

4.2.3.1 bt_conn_set_security

表 4-62 bt_conn_set_security 接口函数说明

信息项	说明
原型	int bt_conn_set_security(struct bt_conn *conn, bt_security_t sec);
功能	设置连接链路的安全等级，并开始进行配对绑定，若配对过程已有本设备或对端设备启动，则可能返回错误，完成后通过 bt_conn_auth_cb_register 注册的 pairing_complete 回调告知应用
参数	const struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中 bt_security_t sec 含义解释：链路安全等级 使用说明：为枚举变量，其值为： BT_SECURITY_L0：安全等级为 0，BR/EDR 使用 BT_SECURITY_L1：安全等级为 1，不加密不认证 BT_SECURITY_L2：安全等级为 2，加密不认证(no MITM) BT_SECURITY_L3：安全等级为 3，加密认证(MITM) BT_SECURITY_L4：安全等级为 4，经过身份验证的安全连接，使用 128bit 密钥
返回值	int 类型 0：配对绑定成功 非 0：配对绑定失败

4.2.3.2 bt_conn_get_security

表 4-63 bt_conn_get_security 接口函数说明

信息项	说明
原型	bt_security_t bt_conn_get_security(struct bt_conn *conn);
功能	获取当前连接链路的安全等级
参数	const struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中
返回值	bt_security_t 类型 BT_SECURITY_L0：安全等级为 0，BR/EDR 使用 BT_SECURITY_L1：安全等级为 1，不加密不认证 BT_SECURITY_L2：安全等级为 2，加密不认证(no MITM) BT_SECURITY_L3：安全等级为 3，加密认证(MITM) BT_SECURITY_L4：安全等级为 4，经过身份验证的安全连接，使用 128bit 密钥

4.2.3.3 bt_conn_enc_key_size

表 4-64 bt_conn_enc_key_size 接口函数说明

信息项	说明
原型	uint8_t bt_conn_enc_key_size(struct bt_conn *conn);
功能	获取加密密钥的大小
参数	const struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中
返回值	uint8_t 类型 0：没有启用加密 其他：加密 key 的大小

4.2.3.4 bt_set_bondable

表 4-65 bt_set_bondable 接口函数说明

信息项	说明
原型	void bt_set_bondable(bool enable);
功能	设置连接链路书否允许进行绑定，该标志的初始化取决于 BT_BONDABLE 标志
参数	bool enable 含义解释：是否允许绑定标志 使用说明：true 为允许绑定，false 为不允许绑定
返回值	无

4.2.3.5 bt_passkey_set

表 4-66 bt_passkey_set 接口函数说明

信息项	说明
原型	int bt_passkey_set(unsigned int passkey);
功能	设置用于配对的固定密码，仅当启用 CONFIG_BT_FIXED_PASSKEY 配置选项时，此 API 才可使用
参数	unsigned int passkey 含义解释：需要设置的固定密码 使用说明：有效范围在 0~999999 之间，或者设置为 BT_PASSKEY_INVALID
返回值	int 类型 0：设置成功 非 0：设置失败

4.2.3.6 bt_conn_auth_cb_register

表 4-67 bt_conn_auth_cb_register 接口函数说明

信息项	说明
原型	int bt_conn_auth_cb_register(const struct bt_conn_auth_cb *cb);
功能	注册配对认证回调函数
参数	const struct bt_conn_auth_cb *cb 含义解释：配对认证回调函数集 使用说明：由应用进行实现
返回值	int 类型 0：注册成功 非 0：注册失败

表 4-68 bt_conn_auth_cb 结构体成员说明

成员项	说明
void (*passkey_display)(struct bt_conn *conn, unsigned int passkey)	含义解释：显示 passkey 给用户的回调函数 使用说明：由用户实现，用于在配对操作时显示 passkey
void (*passkey_entry)(struct bt_conn *conn)	含义解释：请求应用输入 passkey 的回调函数 使用说明：由用户实现，用于在配对绑定时要求用户输入 passkey
void (*passkey_confirm)(struct bt_conn *conn, unsigned int passkey)	含义解释：请求用户对 passkey 进行确认的回调函数 使用说明：由应用实现，用于在配对绑定过程中要求用户对收到的 passkey 进行确认

成员项	说明
void (*cancel)(struct bt_conn *conn)	含义解释：取消正在进行的配对绑定的回调函数 使用说明：此回调取消正在进行的配对绑定流程
void (*pairing_confirm)(struct bt_conn *conn)	含义解释：请求用户对配对绑定流程进行确认的回调函数 使用说明：在无密钥情况下使用该接口对配对绑定流程进行确认
void (*pairing_complete)(struct bt_conn *conn, bool bonded)	含义解释：配对绑定流程完成回调函数 使用说明：此回调函数通知应用配对绑定流程已完成
void (*pairing_failed)(struct bt_conn *conn, enum bt_security_err reason)	含义解释：配对绑定流程失败回调函数 使用说明：此回调函数通知应用配对绑定流程失败
void (*bond_deleted)(uint8_t id, const bt_addr_le_t *peer)	含义解释：配对信息删除完毕回调函数 使用说明：此回调函数通知应用已完成配对绑定信息删除
enum bt_security_err (*pairing_accept)(struct bt_conn *conn, const struct bt_conn_pairing_feat *const feat)	含义解释：接收配对绑定回调函数 使用说明：此回调函数通知应用收到了配对绑定流程，由应用确认是否接收进行配对绑定后续流程

4.2.3.7 bt_conn_auth_passkey_entry

表 4-69 bt_conn_auth_passkey_entry 接口函数说明

信息项	说明
原型	int bt_conn_auth_passkey_entry(struct bt_conn *conn, unsigned int passkey);
功能	使用输入的 passkey 对配对绑定进行回复，在回调函数集 bt_conn_auth_cb 中的 passkey_entry 回调函数中调用此函数
参数	struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中 unsigned int passkey 含义解释：要输入的 passkey 使用说明：取值在 0~999999 之间
返回值	int 类型 0：回复成功 非 0：回复失败

4.2.3.8 bt_conn_auth_cancel

表 4-70 bt_conn_auth_cancel 接口函数说明

信息项	说明
原型	int bt_conn_auth_cancel(struct bt_conn *conn);
功能	取消正在进行的配对绑定流程
参数	struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中
返回值	int 类型 0：取消成功 非 0：取消失败

4.2.3.9 bt_conn_auth_passkey_confirm

表 4-71 bt_conn_auth_passkey_confirm 接口函数说明

信息项	说明
原型	int bt_conn_auth_passkey_confirm(struct bt_conn *conn);
功能	如果确认了 passkey 与用户匹配，则使用此函数进行回复。在回调函数集 bt_conn_auth_cb 中的 passkey_confirm 回调函数中调用此函数
参数	struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中
返回值	int 类型 0：回复成功 非 0：回复失败

4.2.3.10 bt_conn_auth_pairing_confirm

表 4-72 bt_conn_auth_pairing_confirm 接口函数说明

信息项	说明
原型	int bt_conn_auth_pairing_confirm(struct bt_conn *conn);
功能	如果用户对配对绑定进行确认，则用此函数进行回复。在回调函数集 bt_conn_auth_cb 中的 pairing_confirm 回调函数中调用此函数
参数	struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中
返回值	int 类型 0：回复成功 非 0：回复失败

4.2.4 GATT 接口

4.2.4.1 bt_gatt_service_register

表 4-73 bt_gatt_service_register 接口函数说明

信息项	说明
原型	int bt_gatt_service_register(struct bt_gatt_service *svc);
功能	注册 GATT 服务，应用可使用如 BT_GATT_PRIMARY_SERVICE， BT_GATT_CHARACTERISTIC，BT_GATT_DESCRIPTOR 等宏构建自己的 GATT 服务
参数	struct bt_gatt_service *svc 含义解释：包含可用属性的服务 使用说明：结构体说明见表 4-74
返回值	int 类型 0：注册成功 非 0：注册失败

表 4-74 bt_gatt_service 结构体成员说明

成员项	说明
struct bt_gatt_attr*attrs	含义解释：服务属性 使用说明：结构体说明见表 4-75
size_t attr_count	含义解释：服务属性数量 使用说明：计算 attrs 数组的大小
sys_snode_t node	含义解释：服务节点 使用说明：各个服务都是通过链表连接起来

表 4-75 bt_gatt_attr 结构体成员说明

成员项	说明
const struct bt_uuid *uuid	含义解释：属性 UUID 使用说明：每个属性都有自己专属的 UUID，通用 UUID 定义见 Spec
ssize_t (*read)(struct bt_conn *conn, const struct bt_gatt_attr *attr, void *buf, uint16_t len, uint16_t offset);	含义解释：属性读回调函数 使用说明：应用实现，对特定属性进行读操作
ssize_t (*write)(struct bt_conn *conn, const struct bt_gatt_attr *attr, const void *buf, uint16_t len, uint16_t offset, uint8_t flags);	含义解释：属性写回调函数 使用说明：应用实现，对特定属性进行写操作
void *user_data	含义解释：属性中的用户数据

成员项	说明
	使用说明：属性中所使用的用户数据从此参数传入
uint16_t handle	含义解释：属性特性对应 handle 使用说明：handle 值与各个属性分别进行绑定，从 0 开始增加，可通过 handle 查找访问特定属性
uint8_t perm	含义解释：属性权限 使用说明：对各个属性进行限定，如有的属性只允许进行读，不允许进行写

4.2.4.2 bt_gatt_service_unregister

表 4-76 bt_gatt_service_unregister 接口函数说明

信息项	说明
原型	int bt_gatt_service_unregister(struct bt_gatt_service *svc);
功能	注销 GATT 服务
参数	struct bt_gatt_service *svc 含义解释：包含可用属性的服务 使用说明：结构体说明见表 4-74
返回值	int 类型 0：注销成功 非 0：注销失败

4.2.4.3 bt_gatt_foreach_attr

表 4-77 bt_gatt_foreach_attr 接口函数说明

信息项	说明
原型	static inline void bt_gatt_foreach_attr(uint16_t start_handle, uint16_t end_handle, bt_gatt_attr_func_t func, void *user_data)
功能	在给定的范围内遍历各个属性
参数	uint16_t start_handle 含义解释：遍历起始 handle 使用说明：从起始 handle 开始遍历存在的各个属性，递增进行遍历 uint16_t end_handle 含义解释：遍历结束 handle 使用说明：递增到结束 handle 则不继续进行遍历 bt_gatt_attr_func_t func 含义解释：属性处理回调函数

信息项	说明
	使用说明：对遍历到的属性使用回调函数进行处理 void *user_data 含义属性：自定义参数 使用说明：传递到属性处理回调函数中的参数
返回值	无

4.2.4.4 bt_gatt_attr_read

表 4-78 bt_gatt_attr_read 接口函数说明

信息项	说明
原型	ssize_t bt_gatt_attr_read(struct bt_conn *conn, const struct bt_gatt_attr *attr, void *buf, uint16_t buf_len, uint16_t offset, const void *value, uint16_t value_len);
功能	读取 attr 属性的值
参数	struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中 const struct bt_gatt_attr *attr 含义解释：指定读取的 attr 使用说明：需要读取的 attr，结构体说明见表 4-75 void *buf 含义解释：缓存 buffer 使用说明：读取出的 attr 的值存储在该 buffer 中 uint16_t buf_len 含义属性：buffer 长度 使用说明：存储 buffer 的大小 uint16_t offset 含义属性：偏移量 使用说明：从 attr 的 offset 偏移处开始读 const void *value 含义属性：属性值 使用说明：实际存储属性值的地方 uint16_t value_len 含义属性：属性值长度 使用说明：实际属性值大小
返回值	ssize_t 类型 >=0 成功读取到的 bytes

信息项	说明
	<0 读取出错

4.2.4.5 bt_gatt_notify_cb

表 4-79 bt_gatt_notify_cb 接口函数说明

信息项	说明
原型	int bt_gatt_notify_cb(struct bt_conn *conn, struct bt_gatt_notify_params *params);
功能	attr 属性值订阅通知
参数	struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中 struct bt_gatt_notify_params *params 含义解释：通知参数 使用说明：通过该参数确定订阅通知相关属性，结构体说明见表 4-80
返回值	ssize_t 类型 0 订阅通知成功 非 0 订阅通知失败

表 4-80 bt_gatt_notify_params 结构体成员说明

成员项	说明
const struct bt_uuid *uuid	含义解释：通知属性 UUID 使用说明：对特定 UUID 进行通知
const struct bt_gatt_attr *attr	含义解释：通知属性目标 使用说明：对应需要通知的属性
const void *data	含义解释：通知数据 使用说明：需要通知的数据
uint16_t len	含义解释：通知数据长度 使用说明：需要通知的数据长度
bt_gatt_complete_func_t func	含义解释：通知数据回调函数 使用说明：通知成功后的后处理回调函数
void *user_data	含义解释：应用数据 使用说明：回调函数中所要使用的数据

4.2.4.6 bt_gatt_notify

表 4-81 bt_gatt_notify 接口函数说明

信息项	说明
原型	static inline int bt_gatt_notify(struct bt_conn *conn, const struct bt_gatt_attr *attr, const void *data, uint16_t len)
功能	attr 属性值改变时进行订阅通知
参数	struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中 const struct bt_gatt_attr *attr 含义解释：指定读取的 attr 使用说明：需要读取的 attr，结构体说明见表 4-75 const void *data 含义解释：通知数据 使用说明：需要通知的被修改了的数据 uint16_t len 含义解释：通知数据长度 使用说明：需要通知的被修改了的数据的长度
返回值	ssize_t 类型 0 通知成功 非 0 通知失败

4.2.4.7 bt_gatt_indicate

表 4-82 bt_gatt_indicate 接口函数说明

信息项	说明
原型	int bt_gatt_indicate(struct bt_conn *conn, struct bt_gatt_indicate_params *params);
功能	attr 属性值改变时进行提示，如果 conn 为 NULL，则 Indicate 给所有通过 CCC 订阅的 BLE 设备，否则仅直接 Indicate 给指定的 conn。Indicate 和 Notify 区别在于 Notify 需要进行包的回复确认收到，Indicate 不需回复确认收到
参数	struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中 struct bt_gatt_indicate_params *params 含义解释：提示参数 使用说明：通过该参数进行提示相关属性，结构体说明见表 4-83
返回值	int 类型

信息项	说明
	0 提示成功 非 0 提示失败

表 4-83 bt_gatt_indicate_params 结构体成员说明

成员项	说明
const struct bt_uuid *uuid	含义解释：提示属性 UUID 使用说明：对特定 UUID 进行提示
const struct bt_gatt_attr *attr	含义解释：提示属性目标 使用说明：对应需要提示的属性
const void *data	含义解释：提示数据 使用说明：需要提示的数据
uint16_t len	含义解释：提示数据长度 使用说明：需要提示的数据长度
bt_gatt_indicate_func_t func	含义解释：提示数据回调函数 使用说明：提示成功后的后处理回调函数

4.2.4.8 bt_gatt_get_mtu

表 4-84 bt_gatt_indicate 接口函数说明

信息项	说明
原型	uint16_t bt_gatt_get_mtu(struct bt_conn *conn);
功能	获取连接链路进行 ATT 交互的 MTU 包的最大属性数据量
参数	struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中 struct bt_gatt_indicate_params *params 含义解释：提示参数
返回值	int 类型 0 获取成功 非 0 获取失败

4.2.4.9 bt_gatt_exchange_mtu

表 4-85 bt_gatt_exchange_mtu 接口函数说明

信息项	说明
原型	int bt_gatt_exchange_mtu(struct bt_conn *conn, struct bt_gatt_exchange_params *params);
功能	交换 MTU，修改 buffer 可容纳的最大 size，每个连接只能使用一次
参数	struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中 struct bt_gatt_exchange_params *params 含义解释：交换 MTU 参数 使用说明：通过该参数进行 MTU 的交互，结构体说明见表 4-86
返回值	int 类型 0 交互成功 非 0 交互失败

表 4-86 bt_gatt_exchange_params 结构体成员说明

成员项	说明
void (*func)(struct bt_conn *conn, uint8_t err, struct bt_gatt_exchange_params *params);	含义解释：交换 MTU 回调函数 使用说明：交换 MTU 成功的回调函数，由应用进行实现

4.2.4.10 bt_gatt_discover

表 4-87 bt_gatt_discover 接口函数说明

信息项	说明
原型	int bt_gatt_discover(struct bt_conn *conn, struct bt_gatt_discover_params *params);
功能	GATT 服务特性发现，客户端使用此函数来发现服务器上的属性。对于找到的每个属性，都会调用回调，然后可以决定是继续发现还是停止。该过程是异步的，所以参数在使用时需保持有效
参数	struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中 struct bt_gatt_discover_params *params 含义解释：发现参数 使用说明：通过该参数进行发现的交互，结构体说明见表 4-88
返回值	int 类型 0 交互成功 非 0 交互失败

表 4-88 bt_gatt_discover_params 结构体成员说明

成员项	说明
const struct bt_uuid *uuid	含义解释：需要查找的 UUID 使用说明：每个 UUID 需填充对应 UUID 值和类型，类型可以为 16、32 和 128bit
bt_gatt_discover_func_t func	含义解释：查找属性回调函数 使用说明：用于查找到属性后通知应用进行下一步处理，由应用进行实现
uint16_t start_handle	含义解释：遍历起始 handle 使用说明：从起始 handle 开始查找属性，递增进行遍历
uint16_t end_handle	含义解释：查找结束 handle 使用说明：查找到结束 handle 则不继续向下进行查找
uint8_t type	含义解释：查找类型 使用说明：属性有多种类别，可根据此参数只进行一种类别的属性查找。type 为枚举类型，可取值为： BT_GATT_DISCOVER_PRIMARY：查找主要服务 BT_GATT_DISCOVER_SECONDARY：查找次要服务 BT_GATT_DISCOVER_INCLUDE：查找包含服务 BT_GATT_DISCOVER_CHARACTERISTIC：查找特征值 BT_GATT_DISCOVER_DESCRIPTOR：查找描述符 BT_GATT_DISCOVER_ATTRIBUTE：查找属性

4.2.4.11 bt_gatt_read

表 4-89 bt_gatt_read 接口函数说明

信息项	说明
原型	int bt_gatt_read(struct bt_conn *conn, struct bt_gatt_read_params *params);
功能	读取指定 handle/UUID 对应的属性的值。当通过 UUID 读取属性时，若存在多个指定的 UUID，则会调用多次上报和回调。如果 UUID 对应属性值比一个 long 型值要长，则需要使用 handle 和 offset 分别读取剩余的其他数据
参数	struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中 struct bt_gatt_read_params *params 含义解释：属性读取参数

信息项	说明
	使用说明：通过该参数进行属性读取的交互，结构体说明见表 4-90
返回值	int 类型 0 读取成功 非 0 读取失败

表 4-90 bt_gatt_read_params 结构体成员说明

成员项	说明
bt_gatt_read_func_t func	含义解释：属性读取回调函数 使用说明：用于读取完属性后通知应用进行下一步处理，由应用进行实现
size_t handle_count	含义解释：需要读取的 handle 上限 使用说明：若服务中存在多个相同 UUID 对应的属性，则只读取前 handle_count 个该 UUID 对应的属性值
struct { uint16_t handle; uint16_t offset; } single;	含义解释：从指定 handle 属性值的 offset 处开始读取 使用说明：从指定 handle 属性值的 offset 处读取 long 长度数据，用于属性值过长无法一次读取完的情况
uint16_t *handles	含义解释：需要读取的 handle 使用说明：指定需要读取的 handle 对应的属性值
struct { uint16_t start_handle; uint16_t end_handle; const struct bt_uuid *uuid; } by_uuid;	含义解释：读取指定 UUID 对应的属性值 使用说明：从 start_handle 开始进行查找，查找到 end_handle，若中间存在指定 UUID 属性，则读取该属性值

4.2.4.12 bt_gatt_write

表 4-91 bt_gatt_write 接口函数说明

信息项	说明
原型	int bt_gatt_write(struct bt_conn *conn, struct bt_gatt_write_params *params);
功能	向指定 handle 写入属性值
参数	struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中 struct bt_gatt_write_params *params 含义解释：属性值写入参数

信息项	说明
	使用说明：通过该参数进行属性写入的交互，结构体说明见表 4-92
返回值	int 类型 0 写入成功 非 0 写入失败

表 4-92 bt_gatt_write_params 结构体成员说明

成员项	说明
bt_gatt_write_func_t func	含义解释：属性写入回调函数 使用说明：用于写入完属性值后通知应用进行下一步处理，由应用进行实现
uint16_t *handles	含义解释：需要写入的 handle 使用说明：指定需要写入的 handle，向该 handle 对应的属性写入值
uint16_t offset	含义解释：写入属性值偏移 使用说明：从属性值的 offset 处开始进行传入数据的写入
const void *data	含义解释：写入的数据 使用说明：需要写入的数据，与 offset 搭配进行数据的写入
uint16_t length	含义解释：数据长度 使用说明：需要写入的数据的长度

4.2.4.13 bt_gatt_write_without_response

表 4-93 bt_gatt_write_without_response 接口函数说明

信息项	说明
原型	static inline int bt_gatt_write_without_response(struct bt_conn *conn, uint16_t handle, const void *data, uint16_t length, bool sign)
功能	向指定 handle 写入属性值，无需确认写入是否成功
参数	struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中 uint16_t handle 含义解释：需要写入的 handle 使用说明：需要写入属性对应的 handle const void *data 含义解释：写入的数据

信息项	说明
	使用说明：需要写入属性值的数据 uint16_t length 含义解释：数据长度 使用说明：需要写入属性值的数据的长度 bool sign 含义解释：签名写入 使用说明：通过该标志确认是否需要签名写入，如果为签名写入，则需要先进行配对绑定
返回值	int 类型 0 写入成功 非 0 写入失败

4.2.4.14 bt_gatt_subscribe

表 4-94 bt_gatt_subscribe 接口函数说明

信息项	说明
原型	int bt_gatt_subscribe(struct bt_conn *conn, struct bt_gatt_subscribe_params *params);
功能	订阅属性值通知，通过 CCC 进行属性值通知订阅。如果收到通知，则调用回调函数通知应用收到的订阅的属性值，由应用决定是否直接取消此订阅
参数	struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中 struct bt_gatt_subscribe_params *params 含义解释：订阅参数 使用说明：通过该参数进行订阅交互，结构体说明见表 4-95
返回值	int 类型 0 订阅成功 非 0 订阅失败

表 4-95 bt_gatt_subscribe_params 结构体成员说明

成员项	说明
bt_gatt_notify_func_t notify	含义解释：通知回调函数 使用说明：收到属性通知通过回调函数通知应用，由应用进行实现
uint16_t value_handle	含义解释：需要订阅的属性值的 handle 使用说明：指定需要订阅的属性值对应的 handle

成员项	说明
uint16_t ccc_handle	含义解释：需要订阅属性值对应的 ccc handle 使用说明：每个能够订阅的属性值都需要有一个对应的 CCC 属性，在进行订阅时需要将对应 CCC handle 传入
uint16_t value	含义解释：订阅形式 使用说明：存在两种订阅形式，分别为： BT_GATT_CCC_NOTIFY：Notify 形式订阅 BT_GATT_CCC_INDICATE：Indicate 形式订阅 两种订阅形式区别在于 Notify 需要回复包进行确认收到，Indicate 不需进行确认

4.2.4.15 bt_gatt_unsubscribe

表 4-96 bt_gatt_unsubscribe 接口函数说明

信息项	说明
原型	int bt_gatt_unsubscribe(struct bt_conn *conn, struct bt_gatt_subscribe_params *params);
功能	取消属性值订阅，通过使用 CCC 取消订阅
参数	struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中 struct bt_gatt_subscribe_params *params 含义解释：取消订阅参数 使用说明：通过该参数进行取消订阅交互，结构体说明见表 4-95
返回值	int 类型 0 取消订阅成功 非 0 取消订阅失败

4.2.4.16 bt_gatt_cancel

表 4-97 bt_gatt_cancel 接口函数说明

信息项	说明
原型	void bt_gatt_cancel(struct bt_conn *conn, void *params);
功能	取消 GATT 还处于 pending 的请求
参数	struct bt_conn *conn 含义解释：连接链路 使用说明：当前连接的链路相关信息存储在当中

信息项	说明
	void *params 含义解释：请求参数 使用说明：对应各个不同请求参数的结构体
返回值	无

5 示例说明

本章节提供了两个示例，分别对配对绑定和 GATT 读写进行演示说明。

5.1 配对绑定示例

5.1.1 示例简介

本示例通过演示配对绑定场景，简要介绍配对绑定过程的基本使用方法。

5.1.2 获取方法

通过 ble 工程可以完成连接过程演示，ble 工程位于 XR806 SDK 的/project/example/ble 目录。

说明

XR806 SDK 可在以下 Github 仓库获取：https://github.com/XradioTech/xr806_sdk.git

5.1.3 准备工作

示例运行的硬件准备有如下。

1. 评估板：运行示例工程代码。
2. 串口线：连接评估板的 Uart0 插针，用于镜像的烧录和串口的输入和输出。
3. PC 机：用于镜像烧录和串口的输入和输出。

5.1.4 操作步骤

1. 将 main.c 文件中 PERIPHERAL_BOND_EXAMPLE 宏置为 1，其他宏置为 0，编译 ble 工程，获取到 peripheral 对应 image。
2. 将 main.c 文件中 CENTRAL_BOND_EXAMPLE 宏置为 1，其他宏置为 0，编译 ble 工程，获取到 central 对应的 image。
3. 将 image 分别烧录到两个评估板中，完成烧写后，复位即可，示例代码会自动运行。

5.1.5 代码解析

1)peripheral 端代码解析

1. 初始化：

```
static void bt_ready(int err)
{
	/*应用需要的初始化操作*/
}

int main(void)
{
	...
#if (EXAMPLE_USE_PERIPHERAL)
	/*设置 BLE 地址*/
	err = bt_set_id_addr(&pripheral_static_id_addr);
	if (err) {
		printf("Unable to set peripheral identity address (err: %d)\n", err);
		return err;
	}
	/*BLE 协议栈初始化*/
	err = bt_enable(bt_ready);
	if (err) {
		printf("Bluetooth init failed (err %d)\n", err);
		return -1;
	}
	return 0;
}
```

2. 启动广播：

```
/*设置广播数据*/
static const struct bt_data ad[] = {
	BT_DATA_BYTES(BT_DATA_FLAGS, (BT_LE_AD_GENERAL | BT_LE_AD_NO_BREDR)),
	BT_DATA_BYTES(BT_DATA_UUID16_ALL,
				0x0d, 0x18, 0x0f, 0x18, 0x05, 0x18),
	BT_DATA_BYTES(BT_DATA_UUID128_ALL,
				0xf4, 0xde, 0xbc, 0x9a, 0x78, 0x56, 0x34, 0x12,
				0x78, 0x56, 0x34, 0x12, 0x78, 0x56, 0x34, 0x12),
};

/*设置扫描回复包数据*/
static const struct bt_data sd[] = {
	BT_DATA(BT_DATA_NAME_COMPLETE, DEVICE_NAME, DEVICE_NAME_LEN),
};
```

```
/*启动广播*/
err = bt_le_adv_start(BT_LE_ADV_CONN, ad, ARRAY_SIZE(ad), sd, ARRAY_SIZE(sd));
	if (err) {
		printf("Advertising failed to start (err 0x%02x)\n", err);
		return;
	}
```

3. 配对绑定回调函数注册

```
/*配对绑定显示密钥回调函数*/
static void auth_passkey_display(struct bt_conn *conn, unsigned int passkey)
{
	char addr[BT_ADDR_LE_STR_LEN];
	bt_addr_le_to_str(bt_conn_get_dst(conn), addr, sizeof(addr));
	printf("    [Passkey for %s: %06u]\n", addr, passkey);
}

/*配对绑定确认回调函数*/
static void auth_pairing_confirm(struct bt_conn * conn)
{
	char addr[BT_ADDR_LE_STR_LEN];
	bt_addr_le_to_str(bt_conn_get_dst(conn), addr, sizeof(addr));
	printf("Confirm pairing for %s\n", addr);
	bt_conn_auth_pairing_confirm(conn);
	printf("Pairing Confirm Success\n");
}

/*配对绑定失败回调函数*/
static void auth_pairing_failed(struct bt_conn * conn, enum bt_security_err reason)
{
	char addr[BT_ADDR_LE_STR_LEN];
	bt_addr_le_to_str(bt_conn_get_dst(conn), addr, sizeof(addr));
	printf("Pairing failed with %s reason 0x%02x\n", addr, reason);
	disconnected(conn, reason);
}

/*配对绑定成功回调函数*/
static void auth_pairing_complete(struct bt_conn * conn, bool bonded)
{
	char addr[BT_ADDR_LE_STR_LEN];
	bt_addr_le_to_str(bt_conn_get_dst(conn), addr, sizeof(addr));
	printf("%s with %s\n", bonded ? "Bonded" : "Paired", addr);
```

```
}

/*配对绑定回调函数集*/
static struct bt_conn_auth_cb auth_cb_display = {
	.passkey_display = auth_passkey_display,
	.passkey_entry = NULL,
	.passkey_confirm = NULL,
	.cancel = auth_cancel,
	.pairing_confirm = auth_pairing_confirm,
	.pairing_failed = auth_pairing_failed,
	.pairing_complete = auth_pairing_complete,
};

/*配对绑定回调函数集注册*/
bt_conn_auth_cb_register(&auth_cb_display);
```

4. 连接回调函数注册

```
/*断连回调函数*/
static void disconnected(struct bt_conn *conn, uint8_t reason)
{
	/*应用进行实现*/
}

/*连接回调函数*/
static void connected(struct bt_conn *conn, uint8_t err)
{
	/*应用进行实现*/
}

/*连接回调函数集*/
static struct bt_conn_cb conn_callbacks = {
	.connected = connected,
	.disconnected = disconnected,
};

/*连接回调函数集注册*/
bt_conn_cb_register(&conn_callbacks);
```

2）central 端代码解析

1. 初始化

同 5.1.5 节 peripheral 端初始化

2. 配对绑定回调函数注册

同 5.1.5 节 peripheral 端配对绑定回调函数注册

3. 连接回调函数注册

同 5.1.5 节 peripheral 端连接回调函数注册

4. 启动扫描

```
/*广播包解析后的处理函数*/
static bool eir_found(struct bt_data *data, void *user_data)
{
	bt_addr_le_t *addr = user_data;
	int i;
	struct bt_conn *conn_p;
	switch (data->type) {
	case BT_DATA_UUID128_SOME:
	case BT_DATA_UUID128_ALL:
		if (data->data_len % (sizeof(uint8_t) * 16) != 0) {
			printf("AD malformed\n");
			return true;
		}
		printf("[AD]: %u data_len %u\n", data->type, data->data_len);
		for (i = 0; i < data->data_len; i += (sizeof(uint8_t) * 16)) {
			struct bt_uuid_128 uuid;
			int err;
			uuid.uuid.type = BT_UUID_TYPE_128;
			memcpy(&uuid.val, &data->data[i], 16);
			if (bt_uuid_cmp(&uuid.uuid, &vnd_uuid.uuid)) {
				continue;
			}
			err = bt_le_scan_stop();
			if (err) {
				printf("Stop LE scan failed (err %d)\n", err);
				continue;
			}
			conn_p = &default_conn;
			err = bt_conn_le_create(addr, BT_CONN_LE_CREATE_CONN,
BT_LE_CONN_PARAM_DEFAULT, &conn_p);
			if (err) {
```

```
                        printf("Create connection failed (err %d)\n", err);
                        start_scan();
                        return false;
                }
                return false;
        }
    default:
        break;
    }
    return true;
}

/*接收到广播包的处理函数*/
static void device_found(const bt_addr_le_t *addr, int8_t rssi, uint8_t type,
                                        struct net_buf_simple *ad)
{
    char dev[BT_ADDR_LE_STR_LEN];
    bt_addr_le_to_str(addr, dev, sizeof(dev));
    printf("[DEVICE]: %s, AD evt type %u, AD data len %u, RSSI %i\n",
                dev, type, ad->len, rssi);
    /* We're only interested in connectable events */
    if (type == BT_HCI_ADV_IND || type == BT_HCI_ADV_DIRECT_IND) {
        bt_data_parse(ad, eir_found, (void *)addr);
    }
}

/*启动扫描*/
static void start_scan(void)
{
    int err;
    err = bt_le_scan_start(BT_LE_SCAN_PASSIVE, device_found);
    if (err) {
        printf("Scanning failed to start (err %d)\n", err);
        return ;
    }
    printf("Scanning successfully started\n");
}
```

5.1.6 效果展示

1. Peripheral 端运行效果：

```
ble controller open
```

```
version        : 9.1.16
build sha1 : v9.1.16-3-g29cdb2c
build date : Feb   6 2021
build time : 17:28:00
platform      : xr806


Device A not active,waking up!
ble rf_init done!
BLE INIT ALL DONE!
BT Coex. Init. OK.
Start BLE Example!
== XRadio BLE HOST V2.3.0.0 ==
[bt] [WRN] set_flow_control: Controller to host flow control not supported
[bt] [INF] bt_dev_show_info: Identity: C0:90:78:56:34:12 (random)
[bt] [INF] bt_dev_show_info: HCI: version 5.0 (0x09) revision 0x0110, manufacturer 0x063d
[bt] [INF] bt_dev_show_info: LMP: version 5.0 (0x09) subver 0x0110
Bluetooth initialized
************************************************
[RandomAddress C0:90:78:56:34:12 ]
************************************************
Advertising successfully started
[H] Connected!!
========== Connection Parameter ==========
= Remote Address 20:90:F0:00:20:60
= Internval          32
= Latency             0
= Timeout             400
==========================================
Confirm pairing for C0:21:43:65:87:09 (random)
Pairing Confirm Success


@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
    [Passkey for C0:21:43:65:87:09 (random): 123456]
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

Bonded with C0:21:43:65:87:09 (random)

$
```

2. Central 端运行效果：

```
ble controller open
version        : 9.1.16
build sha1 : v9.1.16-3-g29cdb2c
build date : Feb    6 2021
build time : 17:28:00
platform       : xr806


Device A not active,waking up!
ble rf_init done!
BLE INIT ALL DONE!
BT Coex. Init. OK.
Start BLE Example!
== XRadio BLE HOST V2.3.0.0 ==
[bt] [WRN] set_flow_control: Controller to host flow control not supported
[bt] [INF] bt_dev_show_info: Identity: C0:21:43:65:87:09 (random)
[bt] [INF] bt_dev_show_info: HCI: version 5.0 (0x09) revision 0x0110, manufacturer 0x063d
[bt] [INF] bt_dev_show_info: LMP: version 5.0 (0x09) subver 0x0110
Bluetooth initialized
**********************************************
[RandomAddress 25:88:63:75:1D:D5 ]
**********************************************
Scanning successfully started
[DEVICE]: 22:22:75:69:76:98 (public), AD evt type 0, AD data len 31, RSSI -82
[DEVICE]: 7C:25:DA:03:8D:F4 (public), AD evt type 3, AD data len 19, RSSI -88
[DEVICE]: 10:62:53:90:7F:51 (random), AD evt type 0, AD data len 28, RSSI -67
[DEVICE]: DA:1F:33:CA:3A:25 (random), AD evt type 0, AD data len 3, RSSI -63
[DEVICE]: 4E:3A:B2:CD:ED:F7 (random), AD evt type 0, AD data len 17, RSSI -82
[DEVICE]: 7A:F0:F4:DB:51:87 (random), AD evt type 0, AD data len 3, RSSI -91
[DEVICE]: 63:2A:E7:65:A8:6D (random), AD evt type 0, AD data len 17, RSSI -79
[DEVICE]: 15:D6:7A:DC:39:78 (random), AD evt type 3, AD data len 31, RSSI -70
[DEVICE]: 64:90:C1:67:09:29 (public), AD evt type 0, AD data len 29, RSSI -97
[DEVICE]: 7C:25:DA:02:A4:D1 (public), AD evt type 3, AD data len 19, RSSI -88
[DEVICE]: 88:11:96:5B:F5:2E (public), AD evt type 0, AD data len 22, RSSI -84
[DEVICE]: 38:18:4C:F9:27:18 (public), AD evt type 0, AD data len 18, RSSI -56
[DEVICE]: 5D:31:CE:20:8A:DE (random), AD evt type 0, AD data len 20, RSSI -64
[DEVICE]: 5D:63:50:62:48:A4 (random), AD evt type 3, AD data len 31, RSSI -81
[DEVICE]: C0:90:78:56:34:12 (random), AD evt type 0, AD data len 29, RSSI -55
[AD]: 7 data_len 16
**********************************************
[RandomAddress C0:21:43:65:87:09 ]
**********************************************
```

```
[H] Connected!!
========== Connection Parameter ==========
= Remote Address 20:92:60:00:20:62
= Internval          32
= Latency             0
= Timeout             400
==========================================
Enter passkey for C0:90:78:56:34:12 (random)
Enter Passkey Success
Bonded with C0:90:78:56:34:12 (random)

$
```

5.2 GATT 读写示例

5.2.1 示例简介

本示例通过演示 GATT 读写场景，简要介绍 GATT 读写过程的基本使用方法。

5.2.2 获取方法

通过 ble 工程可以完成连接过程演示，ble 工程位于 XR806 SDK 的/project/example/ble 目录。

说明

XR806 SDK 可在以下 Github 仓库获取：https://github.com/XradioTech/xr806_sdk.git

5.2.3 准备工作

示例运行的硬件准备有如下。

1. 评估板：运行示例工程代码。
2. 串口线：连接评估板的 Uart0 插针，用于镜像的烧录和串口的输入和输出。
3. PC 机：用于镜像烧录和串口的输入和输出。

5.2.4 操作步骤

1. 将 main.c 文件中 PERIPHERAL_HT_EXAMPLE 宏置为 1，其他宏置为 0，编译 ble 工程，获取到 peripheral 对应 image。
2. 将 main.c 文件中 CENTRAL_HT_EXAMPLE 宏置为 1，其他宏置为 0，编译 ble 工程，获取到 central 对应的 image。
3. 将 image 分别烧录到两个评估板中，完成烧写后，复位即可，示例代码会自动运行。

5.2.5 代码解析

1) peripheral 端代码解析

1. 初始化：

同 5.1.5 节 peripheral 端初始化

2. 启动广播：

同 5.1.5 节 peripheral 端启动广播

3. 连接回调函数注册

同 5.1.5 节 peripheral 端连接回调函数注册

4. GATT 服务注册

```
/*Vendor 基础服务声明*/
static struct bt_gatt_attr vnd_attrs[] = {
    /* Vendor Primary Service Declaration */
    BT_GATT_PRIMARY_SERVICE(&vnd_uuid),
    BT_GATT_CHARACTERISTIC(&vnd_ind_uuid.uuid, BT_GATT_CHRC_READ | BT_GATT_CHRC_WRITE |
BT_GATT_CHRC_INDICATE,
                           BT_GATT_PERM_READ | BT_GATT_PERM_WRITE,
                           read_vnd, write_ind_vnd, &vnd_ind_val),
    BT_GATT_CCC(vnd_ccc_ind_changed, BT_GATT_PERM_READ | BT_GATT_PERM_WRITE),
    BT_GATT_CHARACTERISTIC(&vnd_write_uuid.uuid, BT_GATT_CHRC_READ | BT_GATT_CHRC_WRITE,
                BT_GATT_PERM_READ |BT_GATT_PERM_WRITE,
                read_vnd, write_vnd, &vnd_value),
    BT_GATT_CHARACTERISTIC(&vnd_notify_uuid.uuid, BT_GATT_CHRC_READ |
BT_GATT_CHRC_NOTIFY,
                BT_GATT_PERM_READ, read_vnd, NULL, &vnd_notify_value),
    BT_GATT_CCC(vnd_ccc_notify_changed, BT_GATT_PERM_READ | BT_GATT_PERM_WRITE),
};

/*vnd 服务*/
static struct bt_gatt_service vnd_svc = BT_GATT_SERVICE(vnd_attrs);

/*注册 GATT 服务*/
bt_gatt_service_register(&vnd_svc);
```

5. GATT 通知

```
static uint8_t vnd_notify_enabled;
```

```
static uint32_t vnd_notify_value = 0; /*需要通知的属性值*/
static struct bt_gatt_notify_params notify_params;
/*通知成功回调函数*/
static void notify_cb(struct bt_conn *conn, void *user_data)
{
	printf("Nofication sent to conn %p\n", conn);
}

/*接收到通知订阅的处理函数*/
static void vnd_ccc_notify_changed(const struct bt_gatt_attr *attr, u16_t value)
{
	vnd_notify_enabled = (value == BT_GATT_CCC_NOTIFY);
	printf("VND notifications %s\n", vnd_notify_enabled ? "enabled" : "disabled");
}

/*服务端构建通知包进行发送*/
if (vnd_notify_enabled) {
	notify_params.uuid = &vnd_notify_uuid.uuid;
	notify_params.attr = &vnd_svc.attrs[7];
	notify_params.data = &vnd_notify_value;
	notify_params.len = sizeof(vnd_notify_value);
	notify_params.func = notify_cb;
	if (bt_gatt_notify_cb(NULL, &notify_params) == 0) {
	}
}
```

6. GATT 提示

```
static uint32_t vnd_ind_val = 0; /*需要提示的属性值*/
static uint8_t simulate_vnd;
static struct bt_gatt_indicate_params ind_params;
/*需要提示的属性值修改函数*/
static ssize_t write_ind_vnd(struct bt_conn *conn, const struct bt_gatt_attr *attr,
                             const void *buf, uint16_t len, uint16_t offset,
                             uint8_t flags)
{
	uint8_t *value = attr->user_data;
	if (offset + len > sizeof(vnd_ind_val)) {
		return BT_GATT_ERR(BT_ATT_ERR_INVALID_OFFSET);
	}
	memcpy(value + offset, buf, len);
	return len;
```

```
}

/*接收到提示订阅的处理函数*/
static void vnd_ccc_ind_changed(const struct bt_gatt_attr *attr, uint16_t value)
{
	simulate_vnd = (value == BT_GATT_CCC_INDICATE) ? 1 : 0;
	printf("VND indication %s\n", simulate_vnd ? "enabled" : "disabled");
}

/*服务端构建提示包进行发送*/
if (simulate_vnd) {
	ind_params.attr = &vnd_svc.attrs[2];
	ind_params.func = indicate_cb;
	ind_params.data = &vnd_ind_val;
	ind_params.len = sizeof(vnd_ind_val);
	if (bt_gatt_indicate(NULL, &ind_params) == 0) {
	}
}
```

2) central 端代码解析

1. 初始化：

同 5.1.5 节 peripheral 端初始化

2. 连接回调函数注册

同 5.1.5 节 peripheral 端连接回调函数注册

3. 启动扫描

同 5.1.5 节 central 端启动扫描

4. GATT 写操作

```
/*在发现指定 UUID 后构建写参数*/
static u8_t discover_write_func(struct bt_conn *conn,
                                const struct bt_gatt_attr *attr,
                                struct bt_gatt_discover_params *params)
{
	char addr[BT_ADDR_LE_STR_LEN];
	bt_addr_le_to_str(bt_conn_get_dst(conn), addr, sizeof(addr));
	if (!attr) {
		printk("Discover write %s complete\n", addr);
```

```
            memset(params, 0, sizeof(*params));
            return BT_GATT_ITER_STOP;
        }
        printk("[ATTRIBUTE] %s write handle %u\n", addr, attr->handle);
        write_params.handle = attr->handle;
        write_params.func = write_func;
        write_params.data = &write_data;
        write_params.length = 1;
        return BT_GATT_ITER_CONTINUE;
}

/*对指定 UUID 进行属性值写*/
err = bt_gatt_write(default_conn, &write_params);
if( err ){
        printf("Write to %s FAIL! %d\n", addr, err);
} else {
        printf("Write Req 0x%x to %s Success!\n", *(u8_t*)write_params.data , addr);
}
```

5. GATT 属性发现

```
/*在发现指定 UUID 后构建写参数*/
static u8_t discover_write_func(struct bt_conn *conn,
                                const struct bt_gatt_attr *attr,
                                struct bt_gatt_discover_params *params)
{
        char addr[BT_ADDR_LE_STR_LEN];
        bt_addr_le_to_str(bt_conn_get_dst(conn), addr, sizeof(addr));
        if (!attr) {
            printk("Discover write %s complete\n", addr);
            memset(params, 0, sizeof(*params));
            return BT_GATT_ITER_STOP;
        }
        printk("[ATTRIBUTE] %s write handle %u\n", addr, attr->handle);
        write_params.handle = attr->handle;
        write_params.func = write_func;
        write_params.data = &write_data;
        write_params.length = 1;
        return BT_GATT_ITER_CONTINUE;
}

/*构建属性请求后进行发送，回调函数是在发现目标 UUID 属性后进行写操作*/
discover_write_params.uuid = &vnd_write_uuid.uuid;
```

```
discover_write_params.func = discover_write_func;
discover_write_params.start_handle = 0x0001;
discover_write_params.end_handle = 0xffff;
discover_write_params.type = BT_GATT_DISCOVER_DESCRIPTOR;
discover_service = 1;
err = bt_gatt_discover(default_conn, &discover_write_params);
if (err) {
	printk("Discover write failed(err %d)\n", err);
	return;
}
```

6. GATT 通知订阅

```
/*接收到通知后的回调处理函数*/
static uint8_t notify_func(struct bt_conn *conn, struct bt_gatt_subscribe_params *params,
                           const void *data, u16_t length)
{
	uint32_t vnd;
	char addr[BT_ADDR_LE_STR_LEN];
	bt_addr_le_to_str(bt_conn_get_dst(conn), addr, sizeof(addr));
	if (!data) {
		params->value_handle = 0;
		return BT_GATT_ITER_STOP;
	}
	memcpy(&vnd, data, length);
	printk("[NOTIFICATION](%s, VND: %d)\n", addr, vnd);
	return BT_GATT_ITER_CONTINUE;
}

/*发现订阅目标后的回调函数，在其中构建订阅参数进行订阅*/
static uint8_t notify_discover_func(struct bt_conn *conn, const struct bt_gatt_attr *attr,
                                    struct bt_gatt_discover_params *params)
{
	int err;
	if (!attr) {
		printf("Discover complete\n");
		(void)memset(params, 0, sizeof(*params));
		return BT_GATT_ITER_STOP;
	}
	printf("[ATTRIBUTE] handle %u\n", attr->handle);
	if (!bt_uuid_cmp(discover_params.uuid, &vnd_uuid.uuid)) {
		discover_params.uuid = &vnd_notify_uuid.uuid;
		discover_params.start_handle = attr->handle + 1;
```

```
            discover_params.type = BT_GATT_DISCOVER_CHARACTERISTIC;

            err = bt_gatt_discover(conn, &discover_params);
            if (err) {
                printf("Discover failed (err %d)\n", err);
            }
        } else if (!bt_uuid_cmp(discover_params.uuid, &vnd_notify_uuid.uuid)) {
            memcpy(&uuid, BT_UUID_GATT_CCC, sizeof(uuid));
            discover_params.uuid = &uuid.uuid;
            discover_params.start_handle = attr->handle + 2;
            discover_params.type = BT_GATT_DISCOVER_DESCRIPTOR;
            subscribe_params[1].value_handle = bt_gatt_attr_value_handle(attr);
            err = bt_gatt_discover(conn, &discover_params);
            if (err) {
                printk("Discover failed (err %d)\n", err);
            }
        } else {
            subscribe_params[1].notify = notify_func;
            subscribe_params[1].value = BT_GATT_CCC_NOTIFY;
            subscribe_params[1].ccc_handle = attr->handle;
            err = bt_gatt_subscribe(conn, &subscribe_params[1]);
            if (err && err != -EALREADY) {
                printk("Subscribe failed (err %d)\n", err);
            } else {
                printk("[SUBSCRIBED]\n");
                notify_flag = 1;
                discover_service = 0;
            }
            return BT_GATT_ITER_STOP;
        }
        return BT_GATT_ITER_STOP;
}

/*构建发现参数并对指定的 UUD 进行发现*/
discover_params.uuid = &vnd_uuid.uuid;
discover_params.func = notify_discover_func;
discover_params.start_handle = 0x0001;
discover_params.end_handle = 0xffff;
discover_params.type = BT_GATT_DISCOVER_PRIMARY;
discover_service = 1;
err = bt_gatt_discover(default_conn, &discover_params);
if (err) {
        printf("Discover failed(err %d)\n", err);
```

```
    return ;
}
```

7. GATT 提示订阅

```
/*接收到提示后的回调处理函数*/
static uint8_t indicate_func(struct bt_conn *conn, struct bt_gatt_subscribe_params *params,
                             const void *data, uint16_t length)
{
    uint32_t vnd;
    char addr[BT_ADDR_LE_STR_LEN];
    bt_addr_le_to_str(bt_conn_get_dst(conn), addr, sizeof(addr));
    if (!data) {
        params->value_handle = 0;
        return BT_GATT_ITER_STOP;
    }
    memcpy(&vnd, data, length);

    printk("[INDICATION](%s, VND: %d)\n", addr, vnd);
    return BT_GATT_ITER_CONTINUE;
}

/*发现订阅目标后的回调函数，在其中构建订阅参数进行订阅*/
static uint8_t indicate_discover_func(struct bt_conn *conn, const struct bt_gatt_attr *attr,
                                      struct bt_gatt_discover_params *params)
{
    int err;
    if (!attr) {
        printf("Discover complete\n");
        (void)memset(params, 0, sizeof(*params));
        return BT_GATT_ITER_STOP;
    }
    printf("[ATTRIBUTE] handle %u\n", attr->handle);
    if (!bt_uuid_cmp(discover_params.uuid, &vnd_uuid.uuid)) {
        discover_params.uuid = &vnd_ind_uuid.uuid;
        discover_params.start_handle = attr->handle + 1;
        discover_params.type = BT_GATT_DISCOVER_CHARACTERISTIC;
        err = bt_gatt_discover(conn, &discover_params);
        if (err) {
            printf("Discover failed (err %d)\n", err);
        }
    } else if (!bt_uuid_cmp(discover_params.uuid, &vnd_ind_uuid.uuid)) {
        memcpy(&uuid, BT_UUID_GATT_CCC, sizeof(uuid));
```

```
            discover_params.uuid = &uuid.uuid;
            discover_params.start_handle = attr->handle + 2;
            discover_params.type = BT_GATT_DISCOVER_DESCRIPTOR;
            subscribe_params[0].value_handle = bt_gatt_attr_value_handle(attr);
            err = bt_gatt_discover(conn, &discover_params);
            if (err) {
                printk("Discover failed (err %d)\n", err);
            }
        } else {
            subscribe_params[0].notify = indicate_func;
            subscribe_params[0].value = BT_GATT_CCC_INDICATE;
            subscribe_params[0].ccc_handle = attr->handle;
            err = bt_gatt_subscribe(conn, &subscribe_params[0]);
            if (err && err != -EALREADY) {
                printk("Subscribe failed (err %d)\n", err);
            } else {
                printk("[SUBSCRIBED]\n");
                indicate_flag = 1;
                discover_service = 0;
            }
            return BT_GATT_ITER_STOP;
        }
        return BT_GATT_ITER_STOP;
}

/*构建发现参数并对指定的 UUD 进行发现*/
discover_params.uuid = &vnd_uuid.uuid;
discover_params.func = indicate_discover_func;
discover_params.start_handle = 0x0001;
discover_params.end_handle = 0xffff;
discover_params.type = BT_GATT_DISCOVER_PRIMARY;
discover_service = 1;
err = bt_gatt_discover(default_conn, &discover_params);
if (err) {
        printf("Discover failed(err %d)\n", err);
        return ;
}
```

5.2.6 效果展示

1. Peripheral 端运行效果：

```
ble controller open
```

```
version       : 9.1.16
build sha1 : v9.1.16-3-g29cdb2c
build date : Feb    6 2021
build time : 17:28:00
platform      : xr806


Device A not active,waking up!
ble rf_init done!
BLE INIT ALL DONE!
BT Coex. Init. OK.
Start BLE Example!
== XRadio BLE HOST V2.3.0.0 ==
[bt] [WRN] set_flow_control: Controller to host flow control not supported
[bt] [INF] bt_dev_show_info: Identity: C0:90:78:56:34:12 (random)
[bt] [INF] bt_dev_show_info: HCI: version 5.0 (0x09) revision 0x0110, manufacturer 0x063d
[bt] [INF] bt_dev_show_info: LMP: version 5.0 (0x09) subver 0x0110
Bluetooth initialized
**********************************************
[RandomAddress C0:90:78:56:34:12 ]
**********************************************
Advertising successfully started
[H] Connected!!
========== Connection Parameter ==========
= Remote Address 20:92:B0:00:20:62
= Internval           32
= Latency              0
= Timeout            400
==========================================
VND indication enabled
Indication success
VND notifications enabled
Nofication sent to conn 0x209248
Indication success
Nofication sent to conn 0x209248
Indication success
write vnd:0
Nofication sent to conn 0x209248
Indication success
write vnd:1
Nofication sent to conn 0x209248
Indication success
write vnd:2
Nofication sent to conn 0x209248
```

```
Indication success
write vnd:3
Nofication sent to conn 0x209248
Indication success
write vnd:4
Nofication sent to conn 0x209248
Indication success
write vnd:5
Nofication sent to conn 0x209248
Indication success
```

2. Central 端运行效果：

```
ble controller open
version       : 9.1.16
build sha1 : v9.1.16-3-g29cdb2c
build date : Feb  6 2021
build time : 17:28:00
platform      : xr806


Device A not active,waking up!
ble rf_init done!
BLE INIT ALL DONE!
BT Coex. Init. OK.
Start BLE Example!
== XRadio BLE HOST V2.3.0.0 ==
[bt] [WRN] set_flow_control: Controller to host flow control not supported
[bt] [INF] bt_dev_show_info: Identity: C0:21:43:65:87:09 (random)
[bt] [INF] bt_dev_show_info: HCI: version 5.0 (0x09) revision 0x0110, manufacturer 0x063d
[bt] [INF] bt_dev_show_info: LMP: version 5.0 (0x09) subver 0x0110
Bluetooth initialized
**********************************************
[RandomAddress 1E:26:6C:55:58:9E ]
**********************************************
Scanning successfully started
[DEVICE]: DA:1F:33:CA:3A:25 (random), AD evt type 0, AD data len 3, RSSI -76
[DEVICE]: 10:62:53:90:7F:51 (random), AD evt type 0, AD data len 28, RSSI -67
[DEVICE]: D0:23:4D:59:61:06 (random), AD evt type 0, AD data len 31, RSSI -85
[DEVICE]: C0:90:78:56:34:12 (random), AD evt type 0, AD data len 29, RSSI -16
[AD]: 7 data_len 16
addr:C0:90:78:56:34:12
**********************************************
[RandomAddress C0:21:43:65:87:09 ]
```

```
**********************************************
[H] Connected!!
========== Connection Parameter ==========
= Remote Address 20:97:A8:00:20:66
= Internval          32
= Latency             0
= Timeout             400
==========================================
indicate flag
write vnd
[ATTRIBUTE] handle 16
[ATTRIBUTE] handle 17
[ATTRIBUTE] handle 19
[SUBSCRIBED]
[INDICATION](C0:90:78:56:34:12 (random), VND: 0)
notify flag
write vnd
[ATTRIBUTE] handle 16
[ATTRIBUTE] handle 22
[ATTRIBUTE] handle 24
[SUBSCRIBED]
[INDICATION](C0:90:78:56:34:12 (random), VND: 1)
[NOTIFICATION](C0:90:78:56:34:12 (random), VND: 0)
write_flag
[ATTRIBUTE] C0:90:78:56:34:12 (random) write handle 21
[INDICATION](C0:90:78:56:34:12 (random), VND: 2)
[NOTIFICATION](C0:90:78:56:34:12 (random), VND: 1)
write vnd
Write Req 0x0 to C0:90:78:56:34:12 (random) Success!
Write 0x0 to C0:90:78:56:34:12 (random) SUCCESS
Discover write C0:90:78:56:34:12 (random) complete
[INDICATION](C0:90:78:56:34:12 (random), VND: 3)
[NOTIFICATION](C0:90:78:56:34:12 (random), VND: 2)
write vnd
Write Req 0x1 to C0:90:78:56:34:12 (random) Success!
Write 0x1 to C0:90:78:56:34:12 (random) SUCCESS
[INDICATION](C0:90:78:56:34:12 (random), VND: 4)
[NOTIFICATION](C0:90:78:56:34:12 (random), VND: 3)
write vnd
Write Req 0x2 to C0:90:78:56:34:12 (random) Success!
Write 0x2 to C0:90:78:56:34:12 (random) SUCCESS
[INDICATION](C0:90:78:56:34:12 (random), VND: 5)
[NOTIFICATION](C0:90:78:56:34:12 (random), VND: 4)
write vnd
```

```
Write Req 0x3 to C0:90:78:56:34:12 (random) Success!
Write 0x3 to C0:90:78:56:34:12 (random) SUCCESS
[INDICATION](C0:90:78:56:34:12 (random), VND: 6)
[NOTIFICATION](C0:90:78:56:34:12 (random), VND: 5)
write vnd
Write Req 0x4 to C0:90:78:56:34:12 (random) Success!
Write 0x4 to C0:90:78:56:34:12 (random) SUCCESS
[INDICATION](C0:90:78:56:34:12 (random), VND: 7)
[NOTIFICATION](C0:90:78:56:34:12 (random), VND: 6)
write vnd
Write Req 0x5 to C0:90:78:56:34:12 (random) Success!
Write 0x5 to C0:90:78:56:34:12 (random) SUCCESS
[INDICATION](C0:90:78:56:34:12 (random), VND: 8)
[NOTIFICATION](C0:90:78:56:34:
```

著作权声明

商标声明

免责声明