

WM_W800_蓝牙系统架构及 API 描述

V1.0

北京联盛德微电子有限责任公司 (winner micro)

地址：北京市海淀区阜成路 67 号银都大厦 1802

电话：+86-10-62161900

WinnerMicro

文档修改记录

[illegible]

目录

文档修改记录	3
目录	4
1 引言	9
1.1 编写目的	9
1.2 预期读者	9
1.3 术语定义	9
1.4 参考资料	9
2 W800 蓝牙系统	10
2.1 芯片蓝牙设计框图	10
2.2 W800 蓝牙系统框图	10
2.3 NimBLE 介绍	11
2.3.1 Nimble	11
2.3.2 NimBLE 目录架构	11
2.4 应用层协议描述	12
2.4.1 GAP	13
2.4.2 ATT	13
2.4.3 GATT	15
2.5 示例代码框架描述	17
2.5.1 蓝牙系统软件代码位置	17
3 API 描述	18

3.1	蓝牙系统 API.....	18
3.2	控制器端 API.....	18
3.3	应用层协议 API.....	20
3.3.1	GAP	20
3.3.2	BLE server.....	22
3.3.3	BLE client.....	25
3.4	蓝牙辅助 WiFi 配网 API	26
3.4.1	应用流程示例	28
3.4.2	辅助 WiFi 配网 Service 定义	28
3.5	用户实现自己的配网 service	28
4	API 使用示例	28
4.1	蓝牙系统使能（退出）	29
4.2	开机运行（退出）demo server.....	29
4.3	开机运行（退出）demo client.....	30
4.4	开机运行多连接（退出）demo client	30
4.5	数据互发功能.....	30
4.6	多连接功能.....	31
4.7	UART 透传功能	32
4.8	开机开启广播.....	32
4.8.1	默认广播数据配置.....	34
4.8.2	用户自定义广播数据设置.....	35
4.9	开机开启扫描	35
4.10	连接态下开启广播/扫描	37

4.10.1	处于 Slave 模式的连接态	37
4.10.2	处于 Master 模式下的连接态	38
5	蓝牙 AT 指令	38
5.1	蓝牙 AT 指令简述	38
5.2	蓝牙系统 AT 指令	40
5.3	蓝牙控制器协议栈 AT 指令	41
5.4	蓝牙应用层 AT 指令	46
5.4.1	设备管理 AT 指令	46
5.4.2	BLE 辅助 WiFi 配网 AT 指令	53
5.4.3	状态码定义:	54
6	蓝牙 AT 指令操作示例	57
6.1	蓝牙系统使能与退出	57
6.1.1	使能蓝牙系统	57
6.1.2	退出蓝牙系统	58
6.2	开关蓝牙 demo 广播	58
6.2.1	使能蓝牙系统	58
6.2.2	开启可连接广播示例	58
6.2.3	停止广播示例	58
6.2.4	退出蓝牙系统	58
6.3	开关蓝牙 demo 扫描	59
6.3.1	使能蓝牙系统	59
6.3.2	开启扫描示例	59
6.3.3	停止扫描示例	59

6.3.4	退出蓝牙系统	59
6.4	开关蓝牙 demo server	59
6.4.1	使能蓝牙系统	59
6.4.2	使能 demo server	59
6.4.3	停止 demo server	59
6.4.4	退出蓝牙系统	60
6.5	开关蓝牙 demo client.....	60
6.5.1	使能蓝牙系统	60
6.5.2	使能 demo client.....	60
6.5.3	停止 demo client.....	60
6.5.4	退出蓝牙系统	60
6.6	开关蓝牙多连接 demo client.....	60
6.6.1	使能蓝牙系统	60
6.6.2	使能多连接 demo client.....	60
6.6.3	停止 demo client.....	60
6.6.4	退出蓝牙系统	60
6.7	开关基于 BLE 的 UART 透传.....	61
6.7.1	使能蓝牙系统	61
6.7.2	使能 UART 透传 Server/Client 端	61
6.7.3	停止 UART 透传	61
6.7.4	退出蓝牙系统	61
6.8	使能辅助 WiFi 配网服务	61
6.8.1	开启蓝牙功能，使能配网服务	61

6.8.2	退出辅助 WiFi 配网服务注销蓝牙系统	62
6.9	W800 测试模式	62
6.9.1	W800 进入测试模式	62
6.9.2	W800 退出信令测试	62

2 引言

2.1 编写目的

本文档用于介绍 W800 蓝牙软件系统, 硬件系统及其开发蓝牙应用参考, 指导用户学习及理解 w800 的蓝牙开发。

2.2 预期读者

蓝牙应用开发人员, 蓝牙协议栈维护人员及测试相关人员

2.3 术语定义

序号	术语/缩略语	说明/定义
1	BT	BlueTooth
2	BLE	Bluetooth Low Energy
3	HCI	Host Controller Interface
4	GAP	General Access Profile
5	IFS	Inter Frame Space

2.4 参考资料

《W800 芯片产品规格书》

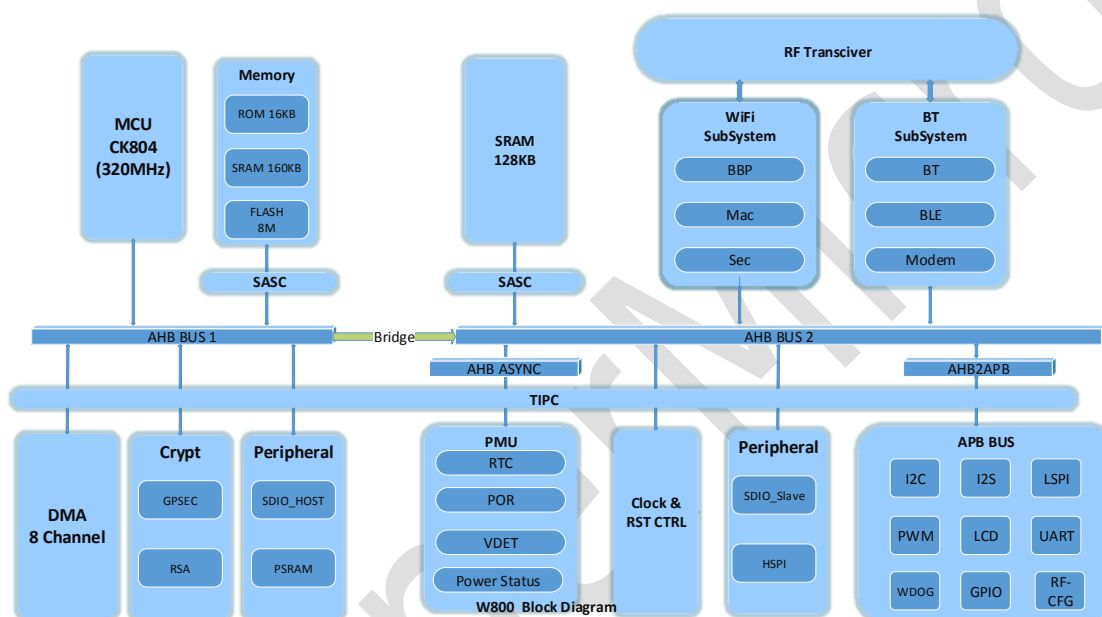
《蓝牙 Core spec4.0 及 4.2》

《WM_W800_蓝牙系统架构以及 API 描述_V1.0》

《蓝牙控制器 spec》

3 W800 蓝牙系统

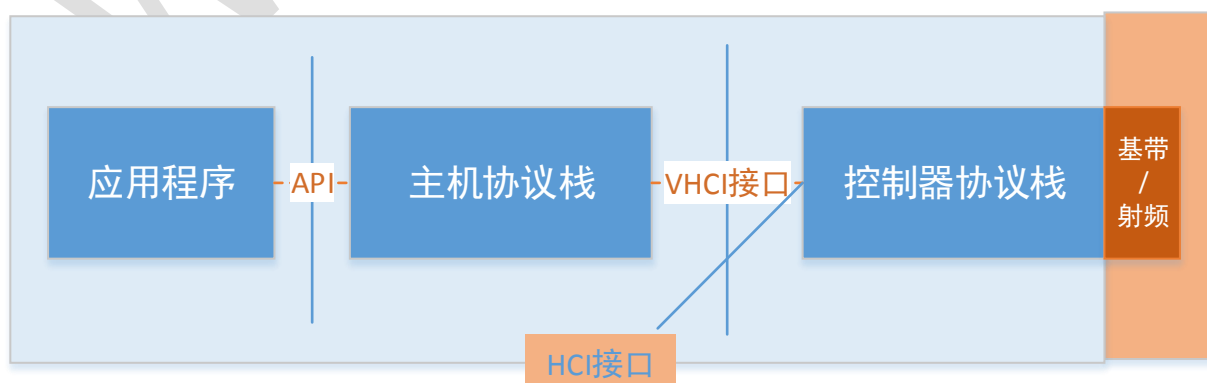
3.1 芯片蓝牙设计框图



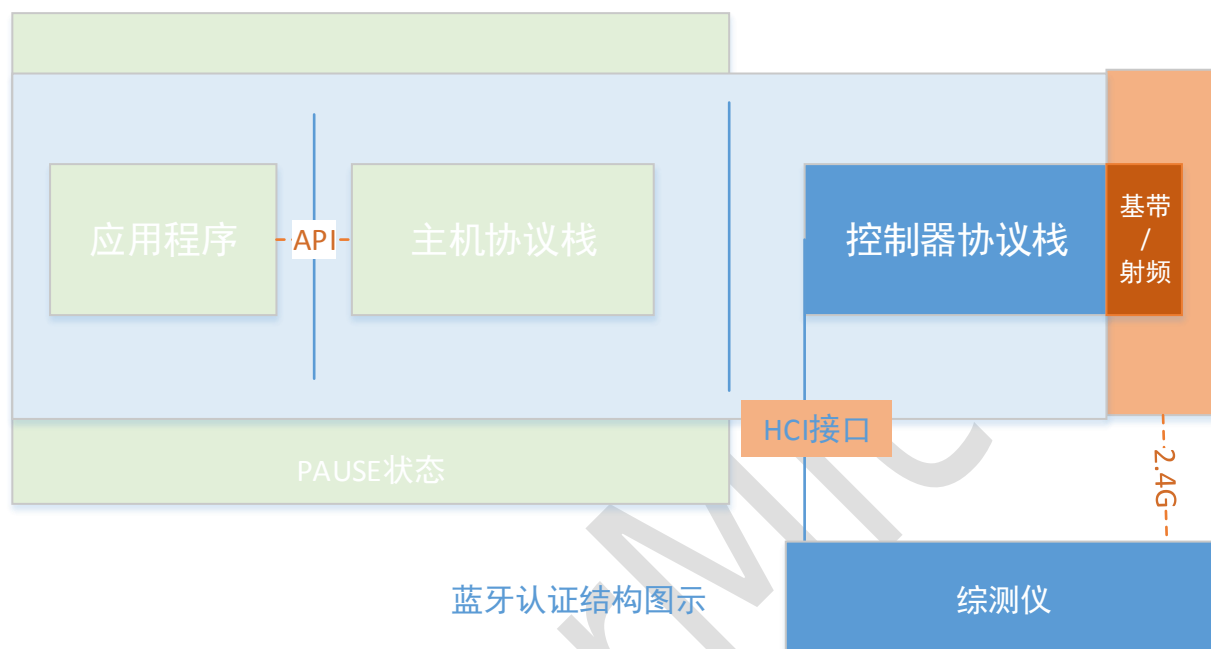
3.2 W800 蓝牙系统框图

W800 蓝牙系统可以分为应用程序部分、主机协议栈、控制器协议栈及蓝牙基带、射频构成。

蓝牙的射频部分和 WiFi 系统共用。



认证的 HCI 串口操作指令参见传统蓝牙非信令测试及 BLE 非信令测试文档。具体测试方法如下图所示：



其中 W800 提供可配置的 UART 口，用于 HCI 指令的响应。综测仪通过 UART 口直接控制控制器。此时主机协议栈处于 freeze 状态。

3.3 NimBLE 介绍

3.3.1 NimBLE

NimBLE 是 Apache 基金会下一个开源的蓝牙 5.0 协议栈,具备完整的 Host 及 Controller 层。资源占用少，支持蓝牙 5.0 特性，也支持 Mesh 等功能。

基于 FreeRTOS 和我们的 Controller，移植了 Host 层。

3.3.2 NimBLE 目录架构

名称	修改日期	类型	大小
docs	2021/3/4 10:13	文件夹	
ext	2021/1/29 16:43	文件夹	
nimble	2021/1/29 16:46	文件夹	
porting	2021/1/29 16:46	文件夹	
Makefile	2021/1/29 17:31	文件	1 KB

整个 nimble 协议栈共包含 4 个目录：

/docs 文件夹包含了 nimble 协议栈的一些说明文档，后缀为.rst

/ext 文件夹包含了 nimble 协议栈使用的加密库

/nimble 文件夹包含了整个 nimble 协议栈代码实现

/porting 文件夹包含了 W800 平台的相关实现

3.4 应用层协议描述

基于我们的 Controller，NimBLE 协议栈支持的功能如下：

- 隐私1.2(LE Privacy 1.2)
- 安全管理(SM),支持传统配对(LE Legacy Pairing),安全连接(LE Secure Connections),特定密钥分发(Transport Specific Key Distribution)
- 链路层PDU数据长度扩展(LE Data Length Extension)
- 多角色并发(主机(central)/从机(peripheral), server/client)
- 同时广播和扫描
- 低速定向广播(Low Duty Cycle Directed Advertising)
- 连接参数请求(Connection parameters request procedure)
- LE Ping
- 完整的GATT客户端，服务端，以及子功能
- 抽象的HCI接口层

3.4.1 GAP

GAP 定义归纳了一系列的角色、模式、流程等概念，用户需要首先理解这些概念，然后根据自己的开发需求，按照 GAP 规范去配置使用 BLE，从而实现 BLE 设备的广播。如，用户如果需要开发一个收发 BLE 广播的应用程序，那么就需要设置 GAP 定义的相关模式，从而实现广播效果。

角色说明如下：

应用角色	应用特性
Broadcaster	用于发送不可连接广播，并对 Observer 发送的扫描请求做出响应，不能与 Observer 建立连接
Observer	接收 Broadcaster 发送的广播，可以选择向 Broadcaster 发送扫描请求，并接收扫描响应
Peripheral	用于发送可连接广播，并根据接收到连接请求与 Central 建立连接
Central	接收可连接广播，并向 Peripheral 发送连接请求，并建立连接

3.4.2 ATT

已连接的 BLE 设备使用 ATT / GATT 规范来进行应用数据交换。

ATT 定义了角色、属性的概念，属性用来保存数据

ATT角色

ATT 角色	应用特性
ATT 服务器	服务器可定义一系列属性，供客户端访问
ATT 客户端	客户端可以使用 ATT 协议来发现、读、写服务器定义的属性

属性

属性逻辑结果如下

属性句柄	属性类型	属性值	属性权限
0x0000- 0xFFFF	UUID	0-N 字节	Read/Write/Indication/Notification

其中：

- 1) 属性句柄由属性服务器分配；
- 2) 属性类型由用户定义或更高层规范指定；
- 3) 属性值由用户定义或更高层规范指定，用来保存应用数据；
- 4) 属性权限由用户定义或更高层规范指定

属性访问方法 - ATT 协议帧

属性访问方法也就是 ATT 协议帧，在蓝牙规范中被称作 ATT PDU(protocol data unit).

ATT PDU 被 ATT 客户端用来发现、读、写属性，或者被 ATT 服务器用来发送属性的 notification、indication。

ATT PDU 的类型有如下 6 种：

ATT PDU 类型	描述
Commands	由客户端发送给服务器的 ATT PDU,服务器不会发送 response
Requests	由客户端发给属性服务器的 ATT PDU，服务器会发送 response 作为响应
Response	服务器发给客户端作为 request 的响应
Notification	由服务器发给客户端，客户端不会发送 confirmation 做为响应

Indication	由服务器发给客户端，客户端需要发送 confirmation 做为响应
Confirmation	由客户端发给服务器，做为 Indication 的响应

3.4.3 GATT

GATT 是为了给应用程序或其他配置文件使用，以便于 ATT 客户端可以跟 ATT 服务器通信。

GATT 定义了使用 ATT 协议 PDU 的框架，这个框架定义了数据交换流程，也定义了应用数据交换格式：服务(service)和特征(characteristics)。

通过 GATT 我们可以发现服务，并读/写或配置对端设备的特征。

GATT 角色

与 ATT 相同，GATT 也存在两种角色：

GATT 角色	角色描述
GATT 服务器	定义服务和特征的 BLE 设备
GATT 客户端	发送数据请求来访问服务与特征的 BLE 设备

GATT 角色是不固定的，只有当启动相应流程时，GATT 角色才被确定，流程结束时 GATT 角色释放。

其中：

GATT 客户端发送 commands 和 requests 给服务器，并且能接收来自服务器的 response, indications 和 notifications;

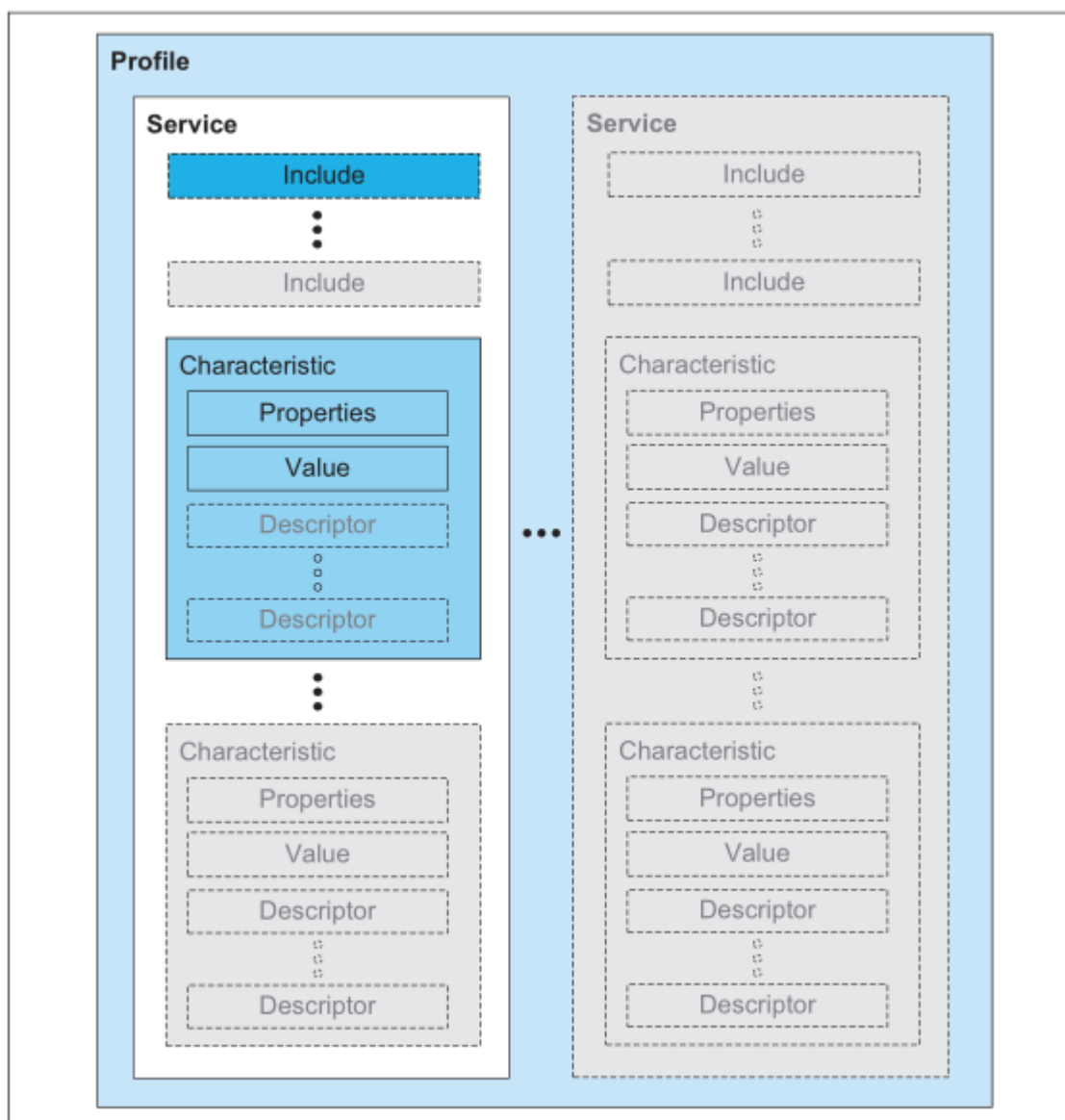
GATT 服务器接收来自客户端的 commands 和 requests 并且发送 response, indication 和 notification 给客户端。

GATT 数据结构

GATT 配置文件指定了数据交换的结构。这个结构定义了基本的元素：**服务(service)**和**特征(characteristics)**。

所有服务和特征都包含在属性中，属性是承载 GATT 数据的容器。

GATT 数据结构如下图所示：



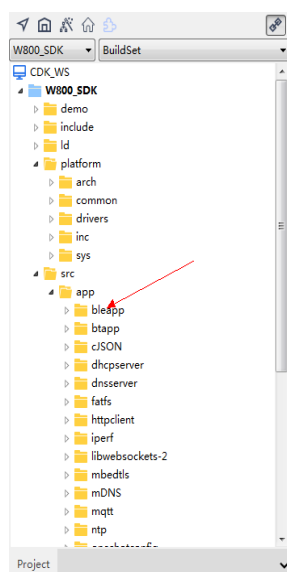
对于 GATT 数据结构的说明：

1. 最顶层是一个 profile，可以理解为一个应用程序，这个应用程序由 1 个或多个服务组成；

2. 每一个服务由特征定义和服务引用组成；
3. 特征包含一个特征值和特征值相关的其他信息；
4. 服务和特征都以属性的形式被 GATT 服务器存储。

3.5 示例代码框架描述

3.5.1 蓝牙系统软件代码位置



bleapp 目录即蓝牙示例代码，用户可以参考或基于此代码进行二次开发。

应用程序文件列表：

No	应用程序模块	说明
1	wm_bt_app.c	主机协议栈主程序入口
2	wm_ble_gap.c	GAP 实现及相关 event 的上报处理
4	wm_ble_server_wifi_prof.c	BLE 辅助配网服务通讯模块，负责传输层的实现
5	wm_ble_server_wifi_app.c	BLE 辅助配网应用协议处理模块，负责应用层协议的实现

6	wm_ble_client_api_demo.c	实现 api 创建 demo server 功能
7	wm_ble_server_api_demo.c	实现 api 创建 demo client 功能
8	wm_ble_client_api_multi_conn_demo.c	实现 api 创建 demo client, 可支持连接 7 个 demo server。
9	wm_ble_uart_if.c	实现基于 BLE 的 UART 透传示例

4 API 描述

4.1 蓝牙系统 API

No	API 名称	描述
1	int tls_bt_init(uint8_t uart_idx)	运行蓝牙系统, 该函数会依次使能主机协议栈和控制器协议栈。
2	int tls_bt_deinit(void)	停止蓝牙系统, 该函数会依次注销主机协议栈和控制器协议栈。

4.2 控制器端 API

No	API 名称	描述
1	tls_bt_status_t tls_bt_ctrl_enable(tls_bt_hci_if_t *p_hci_if, tls_bt_log_level_t log_level)	初始化控制器端协议栈, 分配内存及创建任务等
2	tls_bt_status_t tls_bt_ctrl_disable(void);	注销控制器协议栈

3	<pre>tls_bt_status_t tls_ble_set_tx_power(tls_ble_power_type_t power_type, int8_t power_level);</pre>	设置 BLE 发射功率索引
4	<pre>int8_t tls_ble_get_tx_power(uint8_t power_type);</pre>	读取指定工作类型的发送功率索引
5	<pre>tls_bt_ctrl_status_t tls_bt_controller_get_status(void);</pre>	读取控制器目前的状态,
6	<pre>bool wm_bt_vuart_host_check_send_available(void);</pre>	用于判断主机能否向控制器发送指令
7	<pre>tls_bt_status_t tls_bt_vuart_host_send_packet (uint8_t *data, uint16_t len);</pre>	主机协议栈向控制器发送数据接口
8	<pre>tls_bt_status_t tls_bt_ctrl_if_register (const tls_bt_host_if_t *p_host_if);</pre>	注册控制器数据发送接口, 即主机协议栈接收数据接口
9	<pre>tls_bt_status_t tls_bt_ctrl_sleep (bool enable);</pre>	是否运行控制器在空闲时进入 sleep 模式
10	<pre>bool tls_bt_ctrl_is_sleep (void);</pre>	读取控制器是否处于 sleep 模式
11	<pre>tls_bt_status_t tls_bt_ctrl_wakeup(void)</pre>	退出 sleep 模式
12	<pre>tls_bt_status_t enable_bt_test_mode(tls_bt_hci_if_t *p_hci_if)</pre>	进入蓝牙测试模式

13	tls_bt_status_t exit_bt_test_mode()	退出蓝牙测试模式
----	-------------------------------------	----------

4.3 应用层协议 API

4.3.1 GAP

设备管理层承担控制器的通用设置，如广播，扫描，设备名称修改等功能

4.3.1.1 GAP API 描述

No	API 名称	描述
1.	int tls_ble_gap_init(void);	初始化默认广播、扫描参数；设置设备名称。 注意：该函数在蓝牙系统运行时自动调用。
2.	int tls_ble_gap_deinit(void);	释放资源。 注意：该函数在蓝牙系统注销时自动调用
3.	int tls_ble_gap_set_adv_param(uint8_t adv_type, uint32_t min, uint32_t max, uint8_t chn_map, uint8_t filter_policy, uint8_t *dir_mac, uint8_t dir_mac_type)	设置广播参数
4.	int tls_nimble_gap_adv(wm_ble_adv_type_t type, int duration);	启动、停止广播
5.	int tls_ble_gap_scan(wm_ble_scan_type_t type, bool filter_duplicate);	启动、停止扫描

6.	<pre>int tls_ble_gap_set_scan_param(uint32_t intv, uint32_t window, uint8_t filter_policy, bool limited, bool passive, bool filter_duplicate);</pre>	设置扫描参数
6	<pre>int tls_ble_gap_set_name(const char *dev_name,uint8_t update_flash);</pre>	设置设备名称 注意：如果设备正在进行广播，并且广播参数 struct ble_hs_adv_fields 指定 name_is_complete. 那么设置名称后， 广播需停止并再次使能后，生效。
7	<pre>int tls_ble_gap_get_name(char *dev_name);</pre>	读取设备名称。 注意：该函数首先读取 Flash 中保存的设备名 称，如果不存在，读取 ram 中的设备名称
8	<pre>int tls_ble_gap_set_data(wm_ble_gap_data_t type, uint8_t *data, int data_len);</pre>	用于设置自定义的广播数据或者扫描响应内容
9	<pre>int tls_ble_register_gap_evt(uint32_t evt_type, app_gap_evt_cback_t *evt_cback);</pre>	用于注册 GAP 事件的上报函数
10	<pre>int tls_ble_deregister_gap_evt(uint32_t evt_type, app_gap_evt_cback_t *evt_cback);</pre>	用于注销 GAP 事件的上报函数

4.3.2 BLE server

BLE server 承担 GATT 服务器角色，wm_ble_server_api_demo 模块提供了用户程序开发的示例，示例功能描述为：

- 1, 创建如下 service 列表功能，并启动广播；

```
#define WM_GATT_SVC_UUID      0xFFFF0
#define WM_GATT_INDICATE_UUID 0xFFFF1
#define WM_GATT_WRITE_UUID   0xFFFF2

static const struct ble_gatt_svc_def gatt_demo_svr_svcs[] = {
    {
        /* Service: uart */
        .type = BLE_GATT_SVC_TYPE_PRIMARY,
        .uuid = BLE_UUID16_DECLARE(WM_GATT_SVC_UUID),
        .characteristics = (struct ble_gatt_chr_def[]) { {
            .uuid = BLE_UUID16_DECLARE(WM_GATT_WRITE_UUID),
            .val_handle = &g_ble_demo_attr_write_handle,
            .access_cb = gatt_svr_chr_demo_access_func,
            .flags = BLE_GATT_CHR_F_WRITE,
        }, {
            .uuid = BLE_UUID16_DECLARE(WM_GATT_INDICATE_UUID),
            .val_handle = &g_ble_demo_attr_indicate_handle,
            .access_cb = gatt_svr_chr_demo_access_func,
            .flags = BLE_GATT_CHR_F_INDICATE,
        }, {
            0, /* No more characteristics in this service */
        }
    },
    {
        0, /* No more services */
    }
};
```

- 2, 接收到对方连接后，更新 ATT 层 MTU 功能；
- 3, 接收到对方连接后，如果收到对方的 indication 功能后，持续发送特定的数据到对方。

该模块提供两条对外的 API 分别为初始化及注销，具体代码如下：

```

int tls_ble_server_demo_api_init(tls_ble_output_func_ptr output_func_ptr)
{
    int rc = BLE_HS_EAPP;

    if(bt_adapter_state == WM_BT_STATE_OFF)
    {
        TLS_BT_APPL_TRACE_ERROR("%s failed rc=%s\r\n", __FUNCTION__, tls_bt_rc_2_str(BLE_HS_EDISABLED));
        return BLE_HS_EDISABLED;
    }

    TLS_BT_APPL_TRACE_DEBUG("%s, state=%d\r\n", __FUNCTION__, g_ble_server_state);

    if(g_ble_server_state == BLE_SERVER_MODE_IDLE)
    {
        g_ble_demo_prof_connected = 0;

        //step 0: reset other services. Note
        rc = ble_gatts_reset();
        if(rc != 0)
        {
            TLS_BT_APPL_TRACE_ERROR("tls_ble_server_demo_api_init failed rc=%d\r\n", rc);
            return rc;
        }

        //step 1: config/adding the services
        rc = wm_ble_server_demo_gatt_svr_init();

        if(rc == 0)
        {
            tls_ble_register_gap_evt(WM_BLE_GAP_EVENT_CONNECT|WM_BLE_GAP_EVENT_DISCONNECT|WM_BLE_GAP_EVENT_NOTIFY_T)
            TLS_BT_APPL_TRACE_DEBUG("### wm_ble_server_api_demo_init \r\n");

            g_ble_uart_output_fptr = output_func_ptr;
            /*step 2: start the service*/
            rc = ble_gatts_start();
            assert(rc == 0);

            /*step 3: start advertisement*/
            rc = wm_ble_server_api_demo_adv(true);

            if(rc == 0)
            {
                g_ble_server_state = BLE_SERVER_MODE_ADVERTISING;
            }
            else
            {
                TLS_BT_APPL_TRACE_ERROR("### wm_ble_server_api_demo_init failed(rc=%d)\r\n", rc);
            }
        }
        } ? end if g_ble_server_state==B... ?
    }
    else
    {
        TLS_BT_APPL_TRACE_WARNING("wm_ble_server_api_demo_init registered\r\n");
        rc = BLE_HS_EALREADY;
    }
}

int tls_ble_server_demo_api_deinit()
{
    int rc = BLE_HS_EAPP;

    if(bt_adapter_state == WM_BT_STATE_OFF)
    {
        TLS_BT_APPL_TRACE_ERROR("%s failed rc=%s\r\n", __FUNCTION__, tls_bt_rc_2_str(BLE_HS_EDISABLED));
        return BLE_HS_EDISABLED;
    }

    TLS_BT_APPL_TRACE_DEBUG("%s, state=%d\r\n", __FUNCTION__, g_ble_server_state);

    if(g_ble_server_state == BLE_SERVER_MODE_CONNECTED || g_ble_server_state == BLE_SERVER_MODE_INDICATING)
    {
        g_ble_demo_indicate_enable = 0;

        rc = ble_gap_terminate(g_ble_demo_conn_handle, BLE_ERR_REM_USER_CONN_TERM);
        if(rc == 0)
        {
            g_ble_server_state = BLE_SERVER_MODE_EXITING;
        }
    }
    else if(g_ble_server_state == BLE_SERVER_MODE_ADVERTISING)
    {
        rc = tls_nimble_gap_adv(WM_BLE_ADV_STOP, 0);
        if(rc == 0)
        {
            if(g_ble_uart_output_fptr)
            {
                g_ble_uart_output_fptr = NULL;
            }
            g_send_pending = 0;
            g_ble_server_state = BLE_SERVER_MODE_IDLE;
        }
    }
    else if(g_ble_server_state == BLE_SERVER_MODE_IDLE)
    {
        rc = 0;
    }
    else
    {
        rc = BLE_HS_EALREADY;
    }

    return rc;
} ? end tls_ble_server_demo_api_deinit ?

```

4.3.2.1 BLE server API 描述

NimBLE 协议栈不支持在 GATT service 运行时，动态增加、注销 service 功能。所以，GATT service 必须配置完成后，方可使能 service 功能。

No	API 名称	描述
1	int ble_gatts_reset(void);	复位 GATT service 列表及释放资源。
2	int ble_gatts_count_cfg(const struct ble_gatt_svc_def *defs);	配置 GATT service
3	int ble_gatts_add_svcs(const struct ble_gatt_svc_def *svcs)	添加 GATT service
4	int ble_gatts_start(void)	启动 GATT server
5	int ble_gattc_indicate_custom(uint16_t conn_handle, uint16_t chr_val_handle, struct os_mbuf *txom)	通过指定的 attr_handle 向某个 conn_handle 发送 indication 数据
6	int ble_gattc_notify_custom(uint16_t conn_handle, uint16_t chr_val_handle, struct os_mbuf *txom)	通过指定的 attr_handle 向某个 conn_handle 发送 notification 数据

4.3.3 BLE client

BLE client 承担 GATT 客户端角色，即主动发起扫描，连接，通讯等应用。

wm_ble_client_api_demo 模块提供了如下示例功能：

- 1, 发起扫描；
- 2, 根据广播数据中是否含有 FFF0 的 service 字段，并发起连接；
- 3, 建立连接后读取对方 service 列表；
- 4, 分析 service 列表，判断 characterise 是否含有 FFF1 字段，并使能 indication，收到 indication 数据后打印
- 5, 分析 service 列表，判断 characterise 是否含有 FFF2 字段，并发送 0Xaa,0xbb 字节到对方。

参考该模块实现，用户可以开发自己的应用程序。

4.3.3.1 BLE client API 描述

No	API 名称	描述
1.	int ble_gap_connect(uint8_t own_addr_type, const ble_addr_t *peer_addr, int32_t duration_ms, const struct ble_gap_conn_params *conn_params, ble_gap_event_fn *cb, void *cb_arg)	用于和对方设备建立 BLE 连接
2.	int	建立连接后，读取 server 端

	<pre> ble_gattc_disc_all_svcs(uint16_t conn_handle, ble_gatt_disc_svc_fn *cb, void *cb_arg) </pre>	service 列表
3	<pre> int ble_gattc_exchange_mtu(uint16_t conn_handle, ble_gatt_mtu_fn *cb, void *cb_arg) </pre>	建立连接后，用于和对方交互 ATT 层 MTU 功能
4	<pre> int ble_gattc_write_flat(uint16_t conn_handle, uint16_t attr_handle, const void *data, uint16_t data_len, ble_gatt_attr_fn *cb, void *cb_arg) </pre>	用于向指定的 conn_handle 及 attr_handle 发送数据
5	<pre> int ble_gattc_read(uint16_t conn_handle, uint16_t attr_handle, ble_gatt_attr_fn *cb, void *cb_arg) </pre>	用于向指定的 conn_handle 及 attr_handle 发起读操作

4.4 蓝牙辅助 WiFi 配网 API

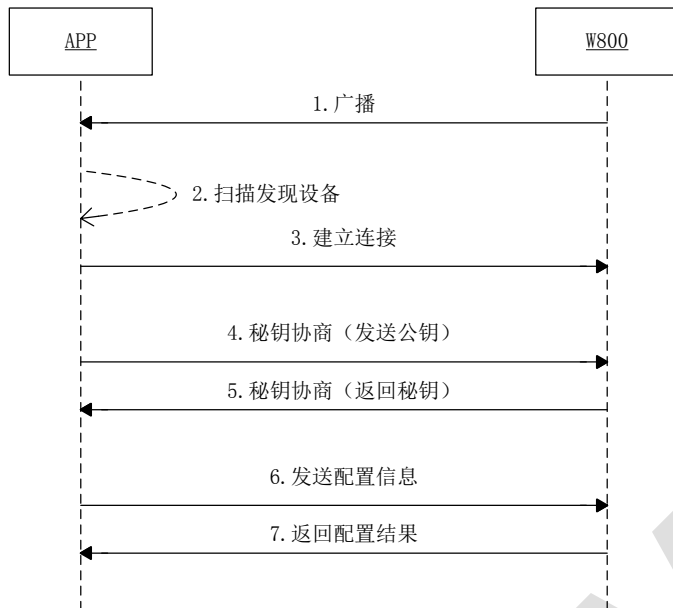
BLE 辅助 WiFi 配网，作为 BLE server 的一个具体应用。wm_ble_server_wifi_prof 实现

BLE profile 的功能，负责数据的传输处理，wm_ble_server_wifi_cfg 处理具体通讯协议处理。这样的层次结构使得应用处理与具体传输层独立开来，逻辑层次调用更加清晰化。

这部分 API 相对简单，如下：

No	API 名称	描述
1	tls_wifi_set_oneshot_flag(flag) flag 0: closed oneshot 1: UDP (broadcast+multicast) 2: AP+socket 3: AP+WEBSERVER 4: BT	当 flag 设置为 4 时，即启动 / 停止 BLE 辅助 WiFi 配网 （使用该模块前需使能蓝牙系统） 注意：1，配网成功后，BLE 的配网 service 会自动退出，广播关闭。如需再次配网请再次调用此 API 即可。 2，如配网失败，用户可以再次配置

4.4.1 应用流程示例



4.4.2 辅助 WiFi 配网 Service 定义

Service 定义:

Service uuid: 0x1824

特征值 uuid: 0x2ABC Write & Indication

特征值描述 uuid: 2902

Write: BleWiFi (手机 APP -> W800) Characteristic UUID: 0x2ABC

Indication: BleWiFi (W800 -> 手机 APP) Characteristic UUID: 0x2ABC

4.5 用户实现自己的配网服务

参考 `wm_ble_server_demo_prof.c` 示例，添加自定义的 service。

5 API 使用示例

W800 蓝牙功能，设备复位后默认是不使能的。如果用户想默认使用蓝牙，可以参考如下

说明。

5.1 蓝牙系统使能（退出）

步骤 1, 在 `tls_bt_entry()` 函数中调用打开蓝牙功能, 关闭蓝牙系统调用 `demo_bt_destroy`;

```
/*This function is called at wm_main.c*/
void tls_bt_entry()
{
    //tls_bt_init(0x01);    //enable it if you want to turn on bluetooth after system booting
}

void tls_bt_exit()
{
    //tls_bt_deinit();      //enable it if you want to turn off bluetooth when system resetting;
}
```

步骤 2, 蓝牙功能打开成功后, 如下回调函数会被调用, 用户在此添加自己的应用程序;

```
static void app_adapter_state_changed_callback(tls_bt_state_t status)
{
    TLS_BT_APPL_TRACE_DEBUG("adapter status = %s\r\n", status==WM_BT_STATE_ON?"bt_state_on":"bt_state_off");
    bt_adapter_state = status;

    #if (TLS_CONFIG_BLE == CFG_ON)
    if(status == WM_BT_STATE_ON)
    {
        TLS_BT_APPL_TRACE_VERBOSE("init base application\r\n");

        //at here, user run their own applications;
        #if 1
        //tls_ble_wifi_cfg_init();
        //tls_ble_server_demo_api_init(NULL);
        //tls_ble_client_demo_api_init(NULL);
        //tls_ble_client_multi_conn_demo_api_init();
        #endif

    }else
    {
        TLS_BT_APPL_TRACE_VERBOSE("deinit base application\r\n");

        //here, user may free their application;
        #if 1
        tls_ble_wifi_cfg_deinit(2);
        tls_ble_server_demo_api_deinit();
        tls_ble_client_demo_api_deinit();
        tls_ble_client_multi_conn_demo_api_deinit();
        #endif

    }

    #endif
}

} ? end app_adapter_state_changed_callback ?
```

5.2 开机运行（退出）示例 server

在 4.1 节中步骤 2 标记的位置处, 调用 `wm_ble_server_demo_api_init()`;

在 4.1 节中步骤 2 标记的位置处, 调用 `wm_ble_server_demo_api_deinit()`; 应用程序的退出功能, 会在蓝牙系统退出时, 自动释放。当然, 蓝牙系统在运行时, 用户也可以随时退出自己的应用程序。

5.3 开机运行（退出）示例 client

在 4.1 节中步骤 2 标记的位置处，调用 `wm_ble_client_demo_api_init()`;

在 4.1 节中步骤 2 标记的位置处，调用 `wm_ble_client_demo_api_deinit()`; 应用程序的退出功能，会在蓝牙系统退出时，自动释放。当然，蓝牙系统在运行时，用户也可以随时退出自己的应用程序。

5.4 开机运行多连接（退出）示例 client

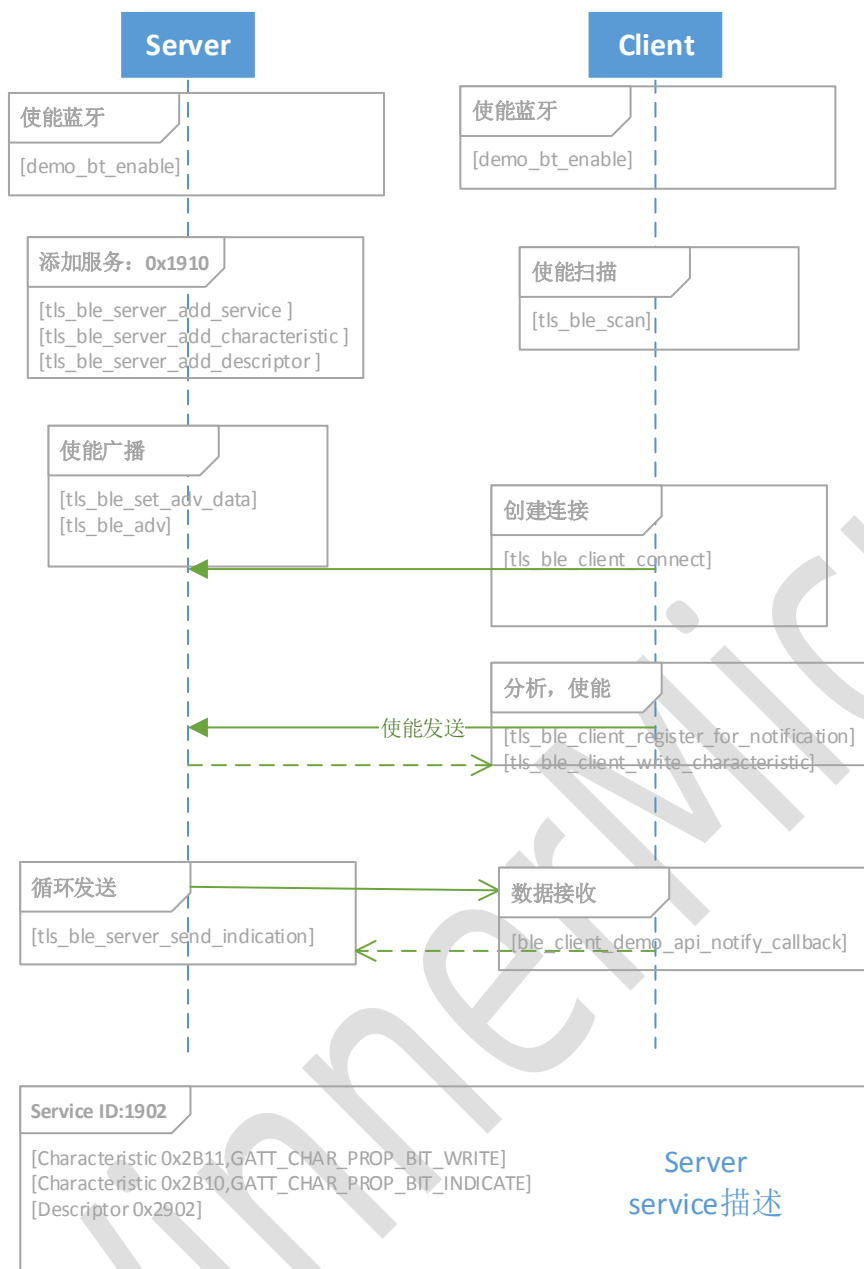
在 4.1 节中步骤 2 标记的位置处，调用 `wm_ble_client_multi_conn_demo_api_init()`;

在 4.1 节中步骤 2 标记的位置处，调用 `wm_ble_client_multi_conn_demo_api_deinit()`; 应用程序的退出功能，会在蓝牙系统退出时，自动释放。当然，蓝牙系统在运行时，用户也可以随时退出自己的应用程序。

5.5 数据互发功能

用两块 demo 板，分别运行 4.2 server demo 和 4.3 client demo，具体 demo 功能参见 3.3.2 和 3.3.3 描述。

连接成功后，server 端会不停的向 client 端以 indication 方式发送数据，时序图如下所示：



5.6 多连接功能

W800 蓝牙系统作为中央设备，最多支持连接 7 个外围设备。该功能的示例配置如下：

- 1, 分别运行 7 个 BLE server 设备，配置模式参见 5.2
- 2, 运行 1 个支持多连接功能的 BLE client，配置模式参见 5.4

此时，client 会依次发起扫描，连接功能，直至连接成功 7 个 BLE server。

注意：限于控制器侧性能，Client 端发起连接时，连接参数必须使用如下间隔：

```
static void wm_ble_update_conn_params(struct ble_gap_conn_params *conn_params)
{
    int i = 0;
    for(i = 0; i<MAX_CONN_DEVICIE_COUNT; i++)
    {
        if(conn_devices[i].conn_state == DEV_DISCONNECTED)
        {
            conn_params->itvl_min = 0x20 + i*16;
            conn_params->itvl_max = 0x22 + i*16;
            return;
        }
    }
}
```

5.7 UART 透传功能

基于 BLE server 和 BLE client 的数据互发，实现了 UART 的透传功能。该功能的示例配置如下：

1,Server 端，采用 UART1,默认属性（115200-8-N-1）透传：在 4.1 章节标记处调用

tls_ble_uart_init(BLE_UART_SERVER_MODE, 0x01, NULL);

2,Client 端，采用 UART1,默认属性（115200-8-N-1）透传：在 4.1 章节标记处调用

tls_ble_uart_init(BLE_UART_CLIENT_MODE, 0x01, NULL);

启动后，server 端开始广播，client 端扫描到广播后，连接 server 端，分析 server 端服务列表，并匹配后，BLE 通道建立。用户可以通过 UART1 进行数据传输。

5.8 开机开启广播

步骤 1，在 tls_bt_entry()函数中调用打开蓝牙功能；

```
/*This function is called at wm_main.c*/
void tls_bt_entry()
{
    //tls_bt_init(0x01); //enable it if you want to turn on bluetooth after system booting
}

void tls_bt_exit()
{
    //tls_bt_deinit(); //enable it if you want to turn off bluetooth when system reseting;
}
```

步骤 2，蓝牙功能打开成功后，如下回调函数会被调用，用户调用开启广播函数

tls_ble_demo_adv(1); //可连接广播


```

void app_adapter_state_changed_callback(tls_bt_state_t status)
{
    tls_bt_host_msg_t msg;
    msg.adapter_state_change.status = status;
    TLS_BT_APPL_TRACE_DEBUG("adapter status = %s\r\n", status==WM_BT_STATE_ON?"bt_state_on":"bt_state_off");

    bt_adapter_state = status;

    #if (TLS_CONFIG_BLE == CFG_ON)

    if(status == WM_BT_STATE_ON)
    {
        TLS_BT_APPL_TRACE_VERBOSE("init base application\r\n");
        /* those funtions should be called basicly*/
        wm_ble_dm_init();
        wm_ble_client_init();
        wm_ble_server_init();

        // at here , user run their own applications;|
        // application_run();
        demo_ble_adv(1);
    }else
    {
        TLS_BT_APPL_TRACE_VERBOSE("deinit base application\r\n");
        wm_ble_dm_deinit();
        wm_ble_client_deinit();
        wm_ble_server_deinit();

        // here, user may free their application;
        // application_stop();
        demo_ble_adv(0);
    }

    #endif
    #if (TLS_CONFIG_BR_EDR == CFG_ON)
    /*class bluetooth application will be enabled by user*/
    #endif

    /*Notify at level application, if registered*/
    if(tls_bt_host_callback_at_ptr)
    {
        tls_bt_host_callback_at_ptr(WM_BT_ADAPTER_STATE_CHG_EVT, &msg);
    }
}

} ? end app_adapter_state_changed_callback ?

```

5.8.1 默认广播数据配置

```

int tls_ble_wifi_adv(bool enable)
{
    int rc;

    if(enable)
    {
        uint8_t own_addr_type;
        struct ble_gap_adv_params adv_params;
        struct ble_hs_adv_fields fields;
        const char *name;
        uint8_t adv_ff_data[] = {0x0C, 0x07, 0x00, 0x10};
        /**
         * Set the advertisement data included in our advertisements:
         *   o Flags (indicates advertisement type and other general info).
         *   o Device name.
         *   o user specific field (winner micro).
         */

        memset(&fields, 0, sizeof fields);

        /* Advertise two flags:
         *   o Discoverability in forthcoming advertisement (general)
         *   o BLE-only (BR/EDR unsupported).
         */
        fields.flags = BLE_HS_ADV_F_DISC_GEN |
                      BLE_HS_ADV_F_BREDR_UNSUP;

        name = ble_svc_gap_device_name();
        fields.name = (uint8_t *)name;
        fields.name_len = strlen(name);
        fields.name_is_complete = 1;

        fields.mfg_data = adv_ff_data;
        fields.mfg_data_len = 4;

        rc = ble_gap_adv_set_fields(&fields);
        if (rc != 0) {
            MODLOG_DFLT(INFO, "error setting advertisement data; rc=%d\r\n", rc);
            return rc;
        }

        MODLOG_DFLT(INFO, "Starting advertising\r\n");

        /* As own address type we use hard-coded value, because we generate
        NRPA and by definition it's random */
        rc = tls_ble_gap_adv(WM_BLE_ADV_IND);
        assert rc == 0;
    } ? end if enable ? else
    {
        MODLOG_DFLT(INFO, "Stop advertising\r\n");
        rc = ble_gap_adv_stop();
    }
    return rc;
} ? end tls_ble_wifi_adv ?

```

5.8.2 用户自定义广播数据设置

```
int tls_ble_demo_adv(uint8_t type)
{
    int rc = 0;
    TLS_BT_APPL_TRACE_DEBUG("### %s type=%d\r\n", __FUNCTION__, type);

    if(bt_adapter_state == WM_BT_STATE_OFF)
    {
        TLS_BT_APPL_TRACE_ERROR("%s failed rc=%s\r\n", __FUNCTION__, tls_bt_rc_2_str(BLE_HS_EDISABLED));
        return BLE_HS_EDISABLED;
    }
    if(type)
    {
        uint8_t bt_mac[6] = {0};
        uint8_t adv_data[] = {
            0x0C, 0x09, 'W', 'M', '-', '0', '0', '0', '0', '0', '0', '0', '0',
            0x02, 0x01, 0x05,
            0x03, 0x19, 0xc1, 0x03};
        extern int tls_get_bt_mac_addr(uint8_t *mac);

        tls_get_bt_mac_addr(bt_mac);
        sprintf(adv_data+8, "%02X:%02X:%02X", bt_mac[3], bt_mac[4], bt_mac[5]);
        adv_data[13] = 0x02; //byte 13 was overwritten to zero by sprintf; recover it;
        rc = tls_ble_gap_set_data(WM_BLE_ADV_DATA, adv_data, 20);
        switch(type)
        {
            case 1:
                rc = tls_ble_gap_adv(WM_BLE_ADV_IND);
                break;
            case 2:
                rc = tls_ble_gap_adv(WM_BLE_ADV_NONCONN_IND);
                break;
            default:
                /*AT/DEMO cmd only support adv_ind and adv_nonconn_ind mode*/
                return BLE_HS_EINVAL;
        }

    } ? end if type ? else
    {
        rc = tls_ble_gap_adv(WM_BLE_ADV_STOP);
    }

    return rc;
} ? end tls_ble_demo_adv ?
```

5.9 开机开启扫描

步骤 1，在 tls_bt_entry()函数中调用打开蓝牙功能;

```
/*This function is called at wm_main.c*/
void tls_bt_entry()
{
    //tls_bt_init(0x01); //enable it if you want to turn on bluetooth after system booting
}

void tls_bt_exit()
{
    //tls_bt_deinit(); //enable it if you want to turn off bluetooth when system reseting;
}
```

步骤 2，蓝牙功能打开成功后，如下回调函数会被调用，用户调用开启扫描函数

```
static void app_adapter_state_changed_callback(tls_bt_state_t status)
{
    TLS_BT_APPL_TRACE_DEBUG("adapter status = %s\r\n", status==WM_BT_STATE_ON?"bt_state_on":"bt_state_off");

    bt_adapter_state = status;

    #if (TLS_CONFIG_BLE == CFG_ON)

    if(status == WM_BT_STATE_ON)
    {
        TLS_BT_APPL_TRACE_VERBOSE("init base application\r\n");

        //at here , user run their own applications;
        #if 1
        //tls_ble_wifi_cfg_init();
        //tls_ble_server_demo_api_init(NULL);
        //tls_ble_client_demo_api_init(NULL);
        //tls_ble_client_multi_conn_demo_api_init();
        tls_ble_demo_scan(1);
        #endif

    }else
    {
        TLS_BT_APPL_TRACE_VERBOSE("deinit base application\r\n");

        //here, user may free their application;
        #if 1
        tls_ble_wifi_cfg_deinit(2);
        tls_ble_server_demo_api_deinit();
        tls_ble_client_demo_api_deinit();
        tls_ble_client_multi_conn_demo_api_deinit();
        #endif

    }

    #endif

} ? end app_adapter_state_changed_callback ?
```

```

static int
ble_gap_evt_cb(struct ble_gap_event *event, void *arg)
{
    struct ble_gap_conn_desc desc;
    struct ble_hs_adv_fields fields;
    int rc = 0;

    switch (event->type) {
    case BLE_GAP_EVENT_DISC:
        rc = ble_hs_adv_parse_fields(&fields, event->disc.data,
                                     event->disc.length_data);
        if (rc != 0) {
            return 0;
        }
        /* An advertisement report was received during GAP discovery. */
        print_adv_fields(&fields);
        return 0;
    case BLE_GAP_EVENT_DISC_COMPLETE:
        break;
    default:
        break;
    }

    return rc;
} ? end ble_gap_evt_cb ?

/**
 * Called          1) AT cmd; 2)demo show;
 *
 * @param type      0: scan stop; 1: scan start, default passive;
 *
 * @return          0 on success; nonzero on failure.
 */
int tls_ble_demo_scan(uint8_t type)
{
    int rc;

    TLS_BT_APPL_TRACE_DEBUG("### %s type=%d\r\n", __FUNCTION__, type);

    if(bt_adapter_state == WM_BT_STATE_OFF)
    {
        TLS_BT_APPL_TRACE_ERROR("%s failed rc=%s\r\n", __FUNCTION__, tls_bt_rc_2_str(BLE_HS_EDISABLED));
        return BLE_HS_EDISABLED;
    }
    if(type)
    {
        tls_ble_register_gap_evt(WM_BLE_GAP_EVENT_DISC|WM_BLE_GAP_EVENT_DISC_COMPLETE, ble_gap_evt_cb);
        rc = tls_ble_gap_scan(WM_BLE_SCAN_PASSIVE, false);
    }else
    {
        rc = tls_ble_gap_scan(WM_BLE_SCAN_STOP, false);
        tls_ble_deregister_gap_evt(WM_BLE_GAP_EVENT_DISC|WM_BLE_GAP_EVENT_DISC_COMPLETE,ble_gap_evt_cb );
    }

    return rc;
} ? end tls_ble_demo_scan ?

```

5.10连接态下开启广播/扫描

步骤 1, 在 tls_bt_entry()函数中调用打开蓝牙功能, 关闭蓝牙系统调用 demo_bt_destroy;

```

void tls_bt_entry()
{
    demo_bt_enable(); //turn on bluetooth system;
}

void tls_bt_exit()
{
    demo_bt_destroy(); //turn off bluetooth system;
}

```

连接态分为 Slave 模式下和 Master 模式下, 下面分两种情况分别予以描述。;

5.10.1 处于 Slave 模式的连接态

步骤 2, 处于 Slave 模式下, 参见 4.2 节。运行 Ble server 的 demo 示例, 运行后, 手机

端发起扫描、连接操作，连接成功后，此时设备侧处于 Slave 模式，手机侧处于 Master 模式。

5.10.1.1 开启广播

步骤 3, [注意]此时设备侧只支持不可连接的广播。

调用 `tls_ble_gap_set_adv_param` 设置广播类型为不可连接广播

调用 `tls_nimble_gap_adv` 开启广播

5.10.1.2 开启扫描

步骤 4 参见 4.4，直接调用开启扫描 API 即可。

```
demo_ble_scan(1);
```

5.10.2 处于 Master 模式下的连接态

参见 4.3 开机运行 demo client 功能，client 同 server 建立连接后：

- 1) 可扫描操作；
- 2) 可发送不可连接的广播操作

6 蓝牙 AT 指令

6.1 蓝牙 AT 指令简述

通过 AT 指令可以控制蓝牙系统，蓝牙 AT 指令共分为 4 类。主机、控制器部分用来配置主机协议栈和控制器协议栈，应用层部分用来配置蓝牙应用程序，测试部分用来配置蓝牙认证功能（该部分部分包含了应用层）。

蓝牙 AT 指令中的缩写含义为：

缩写	含义
CTRL	CONTROLLER
BLESC	BLE SERVICE
BLESV	BLE SERVER
BLEC	BLE CLIENT
POW	POWER
STS	STATUS
DES	DESTORY
PRM	PARAM
FLT	FILTER
CT	CREATE
CH	CHARACTERISTIC
STT	START
STP	STOP
DEL	DELETE
DIS	DISCONNECT
SND	SEND
IND	INDICATION
CONN	CONNECT
NTY	NOTIFICATION
ACC	ACCESS
TEST	TESTMODE

EN	ENABLE
GS	GETSTATUS
TPS	TXPOWERSET
TPG	TXPOWERGET

6.2 蓝牙系统 AT 指令

6.2.1.1 AT+BTEN

功能：

使能蓝牙系统。

格式 (ASCII)：

```
AT+BTEN=<uart_no>,<log_level><CR>
+OK=<status><CR><LF><CR><LF>
```

参数：

uart_no: 串口索引号，定义如下：

值	含 义
1	uart1 目前版本只支持 UART1

Log_level: 日志输出等级，定义如下：

值	含 义
0	关闭 log 输出
1	输出 error 级别的 log
2	输出 warn 级别的 log

3	输出 api 级别的 log
4	输出 event 级别的 log
5	输出 debug 级别的 log
6	输出 verbose 级别的 log

返回：

status：指令响应结果

值	含 义
0	成功
Others>1	失败

6.2.1.2 AT+BTDES

功能：

停止并注销蓝牙系统。

格式（ASCII）：

```
AT+BTDES<CR>
+OK=<status><CR><LF><CR><LF>
```

参数：

参见 BTEN 参数描述

6.3 蓝牙控制器协议栈 AT 指令

6.3.1.1 AT+BTCTRLGS

功能：

获取控制状态。

格式 (ASCII)：

```
AT+BTCTRLGS<CR>
+OK=<status><CR><LF><CR><LF>
```

参数：

status：控制状态，返回格式定义如下：

TLS_BT_CTRL_IDLE	=	(1<<0),
TLS_BT_CTRL_ENABLED	=	(1<<1),
TLS_BT_CTRL_SLEEPING	=	(1<<2),
TLS_BT_CTRL_BLE_ROLE_MASTER	=	(1<<3),
TLS_BT_CTRL_BLE_ROLE_SLAVE	=	(1<<4),
TLS_BT_CTRL_BLE_ROLE_END	=	(1<<5),
TLS_BT_CTRL_BLE_STATE_IDLE	=	(1<<6),
TLS_BT_CTRL_BLE_STATE_ADVERTISING	=	(1<<7),
TLS_BT_CTRL_BLE_STATE_SCANNING	=	(1<<8),
TLS_BT_CTRL_BLE_STATE_INITIATING	=	(1<<9),
TLS_BT_CTRL_BLE_STATE_STOPPING	=	(1<<10),
TLS_BT_CTRL_BLE_STATE_TESTING	=	(1<<11),

6.3.1.2 AT+BTSLEEP

功能：

设置控制器空闲时 sleep 模式。当前版本暂不支持

格式 (ASCII) :

```
AT+BTSLEEP=<cmd><CR>
+OK<CR><LF><CR><LF>
```

参数:

cmd: 控制命令, 定义如下:

值	含义
0	禁止控制器进入sleep
1	允许控制器进入sleep

6.3.1.3 AT+BLETPS

功能:

配置 BLE 特定类型下发送功率。当前版本只支持默认功率设置

格式 (ASCII) :

```
AT+BLETPS=<type>,<level><CR>
+OK<CR><LF><CR><LF>
```

参数:

type: ble类型, 定义如下:

值	含义
0	特定的连接handle
1	特定的连接handle

2	特定的连接handle
3	特定的连接handle
4	特定的连接handle
5	特定的连接handle
6	特定的连接handle
7	特定的连接handle
8	特定的连接handle
9	广播
10	扫描
11	默认功率

level: 功率索引值。

值	含义 dBm
1	1
2	4
3	7
4	10
5	13

6.3.1.4 AT+BLETPG

功能：

获取 BLE 特定类型。当前版本只支持默认功率获取

格式 (ASCII) :

```
AT+BLETPG=?<CR>
+OK=<level><CR><LF><CR><LF>
```

参数：

type: ble类型，定义如下：

值	含义
0	特定的连接handle
1	特定的连接handle
2	特定的连接handle
3	特定的连接handle
4	特定的连接handle
5	特定的连接handle
6	特定的连接handle
7	特定的连接handle
8	特定的连接handle
9	广播
10	扫描
11	默认功率

level: 功率索引值。参见 4.4.1.5

6.3.1.5 AT+BTTEST

功能：

设置蓝牙测试模式。

格式（ASCII）：

```
AT+BTTEST=<mode><CR>
```

```
+OK<CR><LF><CR><LF>
```

参数:

mode: 测试模式, 定义如下:

值	含义
0	退出蓝牙测试模式
1	进入蓝牙测试模式

6.4 蓝牙应用层 AT 指令

蓝牙应用层分为设备管理、BLE server 和 BLE client 三部分。

6.4.1 设备管理 AT 指令

6.4.1.1 AT+BLEADV

功能:

控制 BLE 广播发送和停止。

格式 (ASCII) :

```
AT+BLEADV=<mode><CR>
```

```
+OK<CR><LF><CR><LF>
```

参数:

mode: 控制模式, 定义如下:

值	含义
---	----

0	停止BLE广播
1	启动BLE广播

6.4.1.2 AT+BLEADATA

功能：

配置 BLE 广播内容。

格式（ASCII）：

```
AT+BLEADATA=<data><CR>
+OK<CR><LF><CR><LF>
```

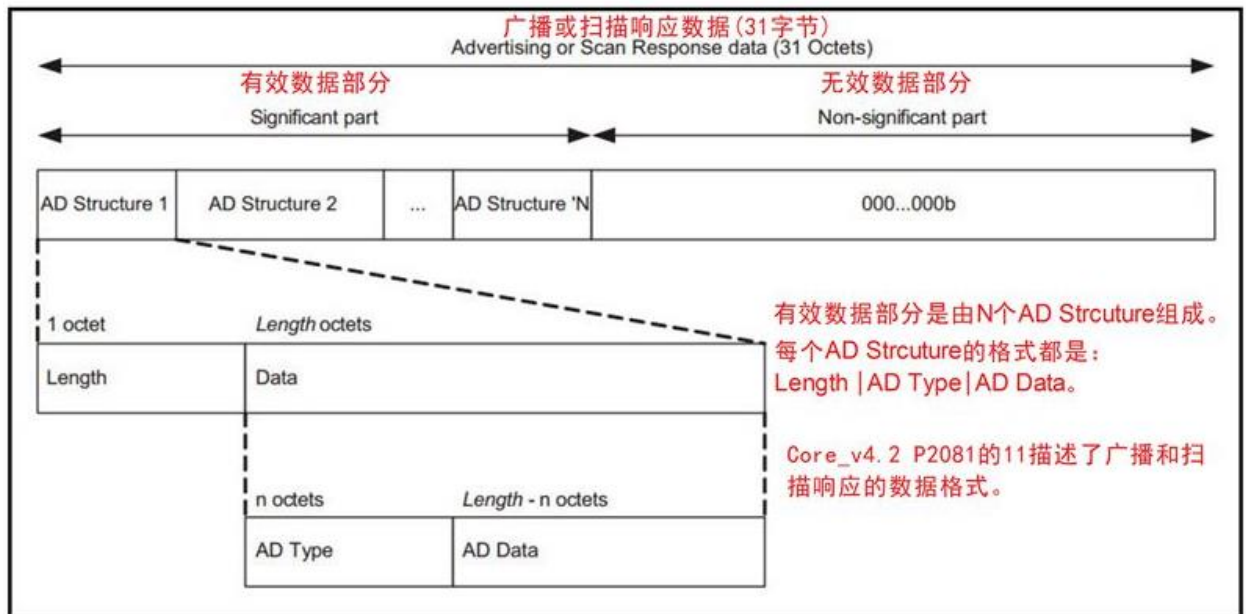
参数：

data：广播内容，为 HEX 格式。最大长度为 62 个字符，相当于 16 进制 31 个字节。

例如，设置广播数据为 0x02 0x01 0x06 0x03 0x09 0x31 0x32，

则设置指令为：AT+BLEADVDATA=02010603093132。

具体广播数据格式定义，参见响应 core specification 描述。



6.4.1.3 AT+BLEAPRM

功能：配置 BLE 广播参数。

格式 (ASCII)：

```
AT+BLEAPRM=<adv_int_min>,<adv_int_max>,<adv_type>,<own_addr_type>,<channel_map>,<adv_filter_policy>,<peer_addr_type>,<peer_addr><CR>
+OK=<adv_int_min>,<adv_int_max>,<adv_type>,<own_addr_type>,<channel_map>,<adv_filter_policy>,<peer_addr_type>,<peer_addr><CR><LF><CR><LF>
```

参数：

adv_int_min: 最小广播间隔，取值范围：0x0020 ~ 0x4000。注意当广播类型值大于等于 3 时，取值范围：0xa0~0x4000

adv_int_max: 最大广播间隔，取值范围：0x0020 ~ 0x4000。注意当广播类型值大于等于 3 时，取值范围：0xa0~0x4000

adv_int_min 和 adv_int_max 填写 16 进制格式，如 10,FF 等

adv_type: 广播类型，定义如下：

值	含义
1	ADV_TYPE_IND可扫描可连接非定向广播
2	ADV_TYPE_DIRECT_IND_HIGH可连接快速定向广播
3	ADV_TYPE_SCAN_IND可扫描不可连接的非定向广播
4	ADV_TYPE_NONCONN_IND不可连接不可扫描非定向广播
5	ADV_TYPE_DIRECT_IND_LOW可连接慢速定向广播

own_addr_type: BLE地址类型，定义如下：（该值由协议栈根据privacy属性值自动填充，AT指令默认填充0即可）

值	含义
0	BLE_ADDR_TYPE_PUBLIC
1	BLE_ADDR_TYPE_RANDOM

channel_map: 广播信道，定义如下：

值	含义
1	ADV_CHNL_37
2	ADV_CHNL_38
4	ADV_CHNL_39
7	ADV_CHNL_ALL

adv_filter_policy: 过滤器，定义如下：

值	含义
0	ADV_FILTER_ALLOW_SCAN_ANY_CON_ANY
1	ADV_FILTER_ALLOW_SCAN_WLST_CON_ANY
2	ADV_FILTER_ALLOW_SCAN_ANY_CON_WLST
3	ADV_FILTER_ALLOW_SCAN_WLST_CON_WLST

peer_addr_type: 对方BLE 地址类型, 定义如下:

值	含义
0	PUBLIC
1	RANDOM

peer_addr: 对方 BLE 地址。

6.4.1.4 AT+BLESCPRM

功能:

配置 BLE 扫描参数。

格式 (ASCII) :

```
AT+BLESCPRM=<window>,<interval>,<scan_mode><CR>
+OK<CR><LF><CR><LF>
```

参数:

windows: 扫描窗口。[0x0004, 0x4000],填写16进制格式, 如10,FF等

interval: 扫描间隔。[0x0004, 0x4000]

scan_mode:扫描方式。[0,1] 被动扫描, 主动扫描

interval的值应大于等于windows，当interval等于windows时，意味控制器始终处于扫描状态，即扫描窗口一直处于打开状态。

6.4.1.5 AT+BLESCAN

功能：

启动或停止扫描。

格式（ASCII）：

```
AT+BLESCAN=<mode><CR>
+OK<CR><LF><CR><LF>
```

参数：

mode：操作模式，定义如下：

值	含义
0	停止扫描
1	启动扫描

扫描结果如下图所示：

```
484661B4A304,-93,HUAWEI,0201020709485541574549
484661B4A304,-93,HUAWEI,0201020709485541574549
484661B4A304,-97,HUAWEI,0201020709485541574549
484661B4A304,-90,HUAWEI,0201020709485541574549
7438B770B0E9,-83,TS300 serie,0201060C085453333030207365726965110622A8FF2F49D8FFFF0100000000000000
6130DE163F82,-103,02011A020A0C0AFF4C001005511C041B92
6130DE163F82,-102,02011A020A0C0AFF4C001005511C041B92
484661B4A304,-91,HUAWEI,0201020709485541574549
7438B770B0E9,-85,TS300 serie,0201060C085453333030207365726965110622A8FF2F49D8FFFF0100000000000000
7438B770B0E9,-88,TS300 serie,0201060C085453333030207365726965110622A8FF2F49D8FFFF0100000000000000
7438B770B0E9,-88,TS300 serie,0201060C085453333030207365726965110622A8FF2F49D8FFFF0100000000000000
```

6.4.1.6 AT+BTNAME

功能：

设置/读取蓝牙名称。

格式 (ASCII) :

```
设置 AT+&BTNAME=[!]<name><CR>

读取 AT+&BTNAME

设置返回: +OK,<CR><LF><CR><LF>

读取返回: +OK=NAME,<CR><LF><CR><LF>
```

参数:

Name 蓝牙名称, ASCII 串。最大长度 16 字节。

6.4.1.7 AT+&BTMAC

功能:

设置/读取蓝牙 MAC 地址。

格式 (ASCII) :

```
设置 AT+&BTMAC=<MAC><CR>

读取 AT+&BTMAC

设置返回: +OK,<CR><LF><CR><LF>

读取返回: +OK=MAC,<CR><LF><CR><LF>
```

参数: MAC 地址

设置示例: AT+&BTMAC=c00d308a0b08

6.4.1.8 AT+ BLESSCM

功能:

指定 BLE 在特定信道扫描。

格式 (ASCII) :

```
AT+ BLESSCM=CH
```

+OK

参数：

CH 定义为：

值	含义
1	指定37信道扫描
2	指定38信道扫描
4	指定39信道扫描
7	跳频，在37、38、39依次扫描（默认）

6.4.2 BLE 辅助 WiFi 配网 AT 指令

6.4.2.1 AT+ONESHOT

功能：

启动或停止配网服务。

格式（ASCII）：

```
AT+ONESHOT=<mode><CR>
+OK=<mode><CR><LF><CR><LF>
```

参数：

mode：操作模式，定义如下：

值	含义
0	停止配网
1	启动UDP配网
2	启动SoftAP+Socket配网

3	启动SoftAP+WebServer配网
4	启动蓝牙配网

注意：

启动蓝牙配网后，用户可以使用手机 APP 进行配置 WiFi 信息。配网成功后，配网服务自动注销，蓝牙关闭广播。如需再次配网请再次启动蓝牙配网。

6.4.3 状态码定义：

6.4.3.1 HCI Reason 码定义：

Success	0x00
Unknown HCI Command	0x01
Unknown Connection Identifier	0x02
Hardware Failure	0x03
Page Timeout	0x04
Authentication Failure	0x05
PIN or Key Missing	0x06
Memory Capacity Exceeded	0x07
Connection Timeout	0x08
Connection Limit Exceeded	0x09
Synchronous Connection Limit To A Device	0x0a

Exceeded	
ACL Connection Already Exists	0x0b
Command Disallowed	0x0c
Connection Rejected due to Limited Resources	0x0d
Connection Rejected Due To Security Reasons	0x0e
Connection Rejected due to Unacceptable BD_ADDR	0x0f
Connection Accept Timeout Exceeded	0x10
Unsupported Feature or Parameter Value	0x11
Invalid HCI Command Parameters	0x12
Remote User Terminated Connection	0x13
Remote Device Terminated Connection due to Low Resources	0x14
Remote Device Terminated Connection due to Power Off	0x15
Connection Terminated By Local Host	0x16
Repeated Attempts	0x17
Pairing Not Allowed	0x18
Unknown LMP PDU	0x19
Unsupported Remote Feature / Unsupported LMP Feature	0x1a
SCO Offset Rejected	0x1b

SCO Interval Rejected	0x1c
SCO Air Mode Rejected	0x1d
Invalid LMP Parameters / Invalid LL Parameters	0x1e
Unspecified Error	0x1f
Unsupported LMP Parameter Value / Unsupported LL Parameter Value	0x20
Role Change Not Allowed	0x21
LMP Response Timeout / LL Response Timeout	0x22
LMP Error Transaction Collision	0x23
LMP PDU Not Allowed	0x24
Encryption Mode Not Acceptable	0x25
Link Key cannot be Changed	0x26
Requested QoS Not Supported	0x27
Instant Passed	0x28
Pairing With Unit Key Not Supported	0x29
Different Transaction Collision	0x2a
Reserved	0x2b
QoS Unacceptable Parameter	0x2c
QoS Rejected	0x2d
Channel Classification Not Supported	0x2e
Insufficient Security	0x2f
Parameter Out Of Mandatory Range	0x30

Reserved	0x31
Role Switch Pending	0x32
Reserved	0x33
Reserved Slot Violation	0x34
Role Switch Failed	0x35
Extended Inquiry Response Too Large	0x36
Secure Simple Pairing Not Supported By Host	0x37
Host Busy - Pairing	0x38
Connection Rejected due to No Suitable Channel Found	0x39
Controller Busy	0x3a
Unacceptable Connection Parameters	0x3b
Directed Advertising Timeout	0x3c
Connection Terminated due to MIC Failure	0x3d
Connection Failed to be Established	0x3e
MAC Connection Failed	0x3f

7 蓝牙 AT 指令操作示例

本章节结合具体示例，给出了蓝牙 AT 指令具体操作规范。黑色截图即 AT 指令的响应。

7.1 蓝牙系统使能与退出

7.1.1 使能蓝牙系统

AT+BTEN=1,0

```
+OK=0,1
```

7.1.2 退出蓝牙系统

AT+BTDES

```
+OK=0,0
```

7.2 开关示例广播

7.2.1 使能蓝牙系统

AT+BTEN=1,0

```
+OK=0,1
```

7.2.2 开启可连接广播示例

AT+BLEDMAADV=1

```
[WM_I] <0:20:53.986> ### tls_ble_demo_adv type=1
starting advertising
GAP procedure initiated: advertise; disc_mode=2 adv_channel_map=0
own_addr_type=0 adv_filter_policy=0 adv_itvl_min=64 adv_itvl_max
=64
+OK
```

7.2.3 停止广播示例

AT+BLEDMAADV=0

```
[WM_I] <0:23:33.818> ### tls_ble_demo_adv type=0
Stop advertising
GAP procedure initiated: stop advertising.
+OK
```

7.2.4 退出蓝牙系统

AT+BTDES

7.3 开关示例扫描

7.3.1 使能蓝牙系统

AT+BTEN=1,0

```
+OK=0,1
```

7.3.2 开启扫描示例

AT+BLEDMSCAN=1

```
[WM_I] <1:41:40.442> ### t1s_ble_demo_scan type=1
GAP procedure initiated: discovery; own_addr_type=0 filter_policy=0 passive=1 limited=0 filter_duplicates=0 duration=forever
+OK

[WM_I] <1:41:40.484> gap_event, [BLE_GAP_EVENT_DISC]
mfg_data=0x06:0x00:0x01:0x09:0x20:0x02:0x6e:0x9d:0xa9:0xc2:0xf7:0xd9:0xbf:0x39:0x6c:0x9c:0x15:0x40:0xf5:0x8d:0x0e:0x16:
0x5d:0x4a:0x0a:0x43:0x9d:0x06:0xce
[WM_I] <1:41:40.506> gap_event, [BLE_GAP_EVENT_DISC]
flags=0x1a
tx_pwr_lvl=7
mfg_data=0x4c:0x00:0x10:0x06:0x6e:0x1d:0x67:0x68:0x89:0x80
[WM_I] <1:41:40.520> gap_event, [BLE_GAP_EVENT_DISC]
flags=0x1a
tx_pwr_lvl=12
mfg_data=0x4c:0x00:0x10:0x07:0x1a:0x1f:0xe6:0x14:0xbf:0x73:0x18
[WM_I] <1:41:40.536> gap_event, [BLE_GAP_EVENT_DISC]
flags=0x06
mfg_data=0x4c:0x00:0x10:0x05:0x59:0x1c:0x60:0x55:0x47
[WM_I] <1:41:40.548> gap_event, [BLE_GAP_EVENT_DISC]
mfg_data=0x06:0x00:0x01:0x09:0x20:0x02:0x90:0xa9:0x5d:0xcf:0x5a:0x59:0x92:0x39:0x3a:0xd9:0x63:0xda:0x9f:0x76:0x10:0x7a:
```

7.3.3 停止扫描示例

AT+BLEDMSCAN=0

7.3.4 退出蓝牙系统

AT+BTDES

7.4 开关示例 server

7.4.1 使能蓝牙系统

AT+BTEN=1,0

```
+OK=0,1
```

7.4.2 使能 demo server

AT+BLEDSD=1

7.4.3 停止 demo server

AT+BLEDSD=0

7.4.4 退出蓝牙系统

AT+BTDES

7.5 开关示例 client

7.5.1 使能蓝牙系统

AT+BTEN=1,0

```
+OK=0,1
```

7.5.2 使能示例 client

AT+BLED=1

7.5.3 停止示例 client

AT+BLED=0

7.5.4 退出蓝牙系统

AT+BTDES

7.6 开关多连接示例 client

7.6.1 使能蓝牙系统

AT+BTEN=1,0

```
+OK=0,1
```

7.6.2 使能多连接 demo client

AT+BLEDPMC=1

7.6.3 停止 demo client

AT+BLEDPMC=0

7.6.4 退出蓝牙系统

AT+BTDES

7.7 开关 UART 透传

7.7.1 使能蓝牙系统

AT+BTEN=1,0

```
+OK=0,1
```

7.7.2 使能 UART 透传 Server/Client 端

AT+BLEUM=1,1 //使能 UART 透传的 server 端，采用 UART1 透传

AT+BLEUM=2,1 //使能 UART 透传的 client 端，采用 UART1 透传

7.7.3 停止 UART 透传

AT+BLEUM=0,1 //关闭 server 端 UART 透传模式

AT+BLEUM=0,2 //关闭 client 端 UART 透传模式

7.7.4 退出蓝牙系统

AT+BTDES

7.8 使能辅助 WiFi 配网服务

7.8.1 开启蓝牙功能，使能配网服务

AT+BTEN=1,0 //使能蓝牙系统

AT+ONESHOT=4 //开启配网服务

此时可以用 APP 进行配网操作；注意配网成功后，系统会自动注销配网服务。

```
+OK=0,1
```

```
+OK
```

7.8.2 退出辅助 WiFi 配网服务注销蓝牙系统

AT+ONESHOT=0 //退出配网服务

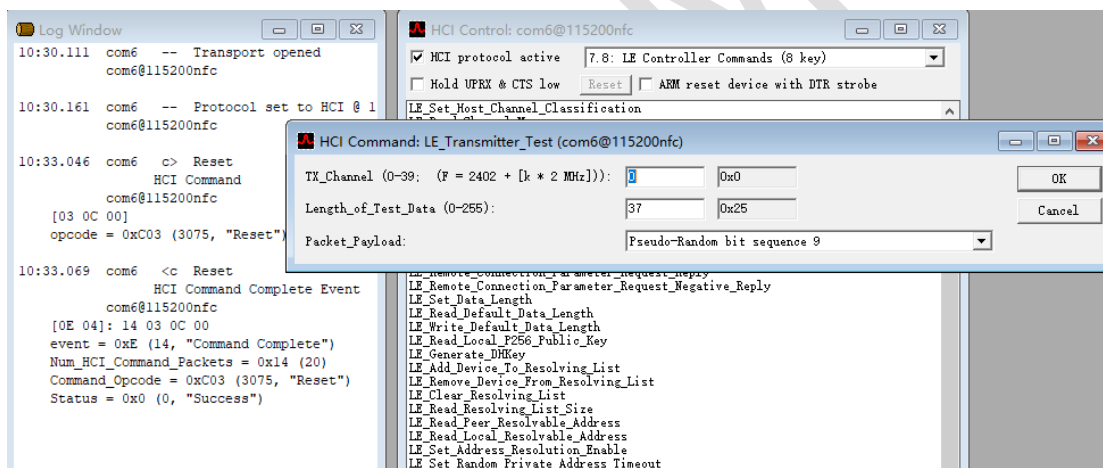
AT+BTDES //退出蓝牙系统

7.9 W800 测试模式

W800 支持实时进入测试模式，客户可以用于测试 RF 性能及控制器功能测试和认证测试。

7.9.1 W800 进入测试模式

AT+BTTEST=1 //进入蓝牙测试，此时可以用测试工具通过配置的 uart 口直接操作 controller。



7.9.2 W800 退出信令测试

AT+BTTEST=0 //退出测试模式，此时主机协议栈控制 controller.