

WS63V100 软件

# 开发指南

文档版本 01

发布日期 2024-04-10

# 前言

## 概述

本文档详细介绍了 WS63V100 Wi-Fi、BLE&SLE 接口功能以及开发流程。

## 产品版本

与本文档对应的产品版本如下。

产品名称	产品版本
WS63	V100

## 读者对象





本文档主要适用以下工程师：

- 技术支持工程
- 软件开发工程师

## 符号约定

在本文中可能出现下列标志，它们所代表的含义如下。

符号	说明
 危险	表示如不避免则将会导致死亡或严重伤害的具有高等级风险的危

符号	说明
	害。
 警告	表示如不可避免则可能导致死亡或严重伤害的具有中等级风险的危害。
 注意	表示如不可避免则可能导致轻微或中度伤害的具有低等级风险的危害。
 须知	用于传递设备或环境安全警示信息。如不可避免则可能会导致设备损坏、数据丢失、设备性能降低或其它不可预知的结果。 “须知”不涉及人身伤害。
 说明	对正文中重点信息的补充说明。 “说明”不是安全警示信息，不涉及人身、设备及环境伤害信息。

修改记录

文档版本	发布日期	修改说明
01	2024-04-10	第一次正式版本发布。
00B06	2024-03-29	<ul style="list-style-type: none"><li>更新 “2.6.2 使用指南” 小节内容。</li><li>更新 “2.7.2 使用指南” 小节内容。</li></ul>
00B05	2024-03-14	更新 “3.3 BLE&SLE 功率档位定制” 小节内容。
00B04	2024-02-22	<ul style="list-style-type: none"><li>更新 “2.4 STA&amp;SoftAp 共存” 小节内容。</li><li>更新 “2.5.2 开发流程” 小节内容。</li><li>更新 “2.7.2 使用指南” 小节内容。</li><li>更新 “3.2.5.2 开发流程” 小节内容。</li><li>更新 “4.2 开发流程” 小节内容。</li><li>更新 “4.3 注意事项” 小节内容。</li></ul>

文档版本	发布日期	修改说明
00B03	2024-01-15	<ul style="list-style-type: none"><li>新增 "4 雷达 软件开发" 章节内容。</li></ul>
00B02	2023-12-18	<ul style="list-style-type: none"><li>更新 "2.2.4 Sample 用例" 小节内容。</li><li>更新 "2.3.4 Sample 用例" 小节内容。</li><li>更新 "5 注意事项" 小节内容。</li></ul>
00B01	2023-11-27	第一次临时版本发布。

目 录

前言 .....i

1 概述 .....1

2 Wi-Fi 软件开发.....3

2.1 驱动加载与卸载.....3

2.1.1 概述 .....3

2.1.2 开发流程 .....3

2.1.3 编程实例 .....4

2.2 STA 功能.....5

2.2.1 概述 .....5

2.2.2 开发流程 .....6

2.2.3 注意事项 .....7

2.2.4 Sample 用例 .....8

2.3 SoftAp 功能.....8

2.3.1 概述 .....8

2.3.2 开发流程 .....9

2.3.3 注意事项 .....10

2.3.4 Sample 用例 .....11

2.4 STA&SoftAp 共存.....11

2.4.1 概述 .....11

2.4.2 开发流程 .....11

2.4.3 编程实例 .....12

2.5 Wi-Fi&蓝牙共存 .....12

2.5.1 概述 .....12

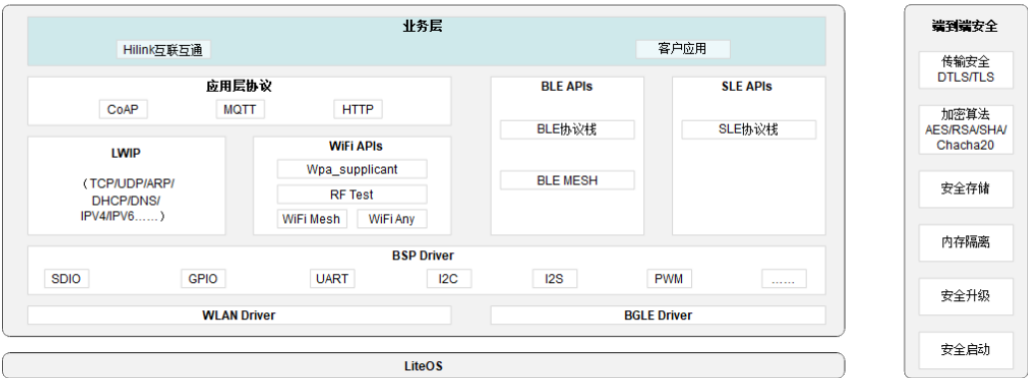
2.5.2 开发流程 .....	13
2.5.3 注意事项 .....	13
2.6 国家码功能配置 .....	13
2.6.1 使用背景 .....	13
2.6.2 使用指南 .....	14
2.7 频偏温度补偿功能 .....	17
2.7.1 使用背景 .....	17
2.7.2 使用指南 .....	17
2.8 FAQ .....	18
2.8.1 频偏修正 .....	18
<b>3 BLE&amp;SLE 软件开发 .....</b>	<b>19</b>
3.1 BLE 开发流程 .....	19
3.1.1 概述 .....	19
3.1.2 GAP 接口 .....	19
3.1.2.1 概述 .....	19
3.1.2.2 开发流程 .....	20
3.1.2.3 注意事项 .....	24
3.1.3 GATT server 接口 .....	24
3.1.3.1 概述 .....	24
3.1.3.2 开发流程 .....	24
3.1.4 GATT client 接口 .....	28
3.1.4.1 概述 .....	28
3.1.4.2 开发流程 .....	28
3.1.5 错误码 .....	30
3.2 星闪开发流程 .....	31
3.2.1 概述 .....	31
3.2.2 Device Discovery 接口 .....	32
3.2.2.1 概述 .....	32
3.2.2.2 开发流程 .....	32
3.2.2.3 注意事项 .....	35
3.2.3 Connection Manager 接口 .....	35

3.2.3.1 概述 .....	35
3.2.3.2 开发流程 .....	35
3.2.4 SSAP server 接口 .....	37
3.2.4.1 概述 .....	37
3.2.4.2 开发流程 .....	38
3.2.5 SSAP client 接口.....	41
3.2.5.1 概述 .....	41
3.2.5.2 开发流程 .....	41
3.2.6 错误码.....	44
3.3 BLE&SLE 功率档位定制 .....	45
<b>4 雷达 软件开发 .....</b>	<b>47</b>
4.1 概述 .....	47
4.2 开发流程 .....	47
4.2.1 数据结构 .....	47
4.2.2 APIs .....	48
4.2.3 错误码.....	49
4.3 注意事项 .....	49
4.4 编程实例 .....	49
<b>5 注意事项.....</b>	<b>52</b>
5.1 看门狗.....	52

# 1 概述

WS63V100 通过 API（Application Programming Interface）面向开发者提供 Wi-Fi 功能的开发和应用接口，包括芯片初始化、资源配置、Station 创建和配置、扫描、关联以及去关联、状态查询等一系列功能，框架结构如图 1-1 所示。

图1-1 WS63V100 解决方案框图



各功能模块说明如下：

- 业务层：用户基于 API 接口的二次开发。
- 应用层协议：应用层网络协议。
- LWIP 协议栈：TCP/IP 协议栈。
- WiFi APIs：提供基于 SDK 的 WiFi 通用接口。
- BLE APIs：提供基于 SDK 的 BLE 通用接口。
- SLE APIs：提供基于 SDK 的 SLE 通用接口。
- 雷达 APIs：提供基于 SDK 的雷达通用接口。
- BSP Driver：芯片和外围设备驱动。



- WLAN Driver: 802.11 协议实现模块。
- BLE&SLE Driver: BLE 及星闪协议实现模块。

#### 说明

该文档描述各个模块功能的开发流程。

# 2 Wi-Fi 软件开发

- 2.1 驱动加载与卸载
- 2.2 STA 功能
- 2.3 SoftAp 功能
- 2.4 STA&SoftAp 共存
- 2.5 Wi-Fi&蓝牙共存
- 2.6 国家码功能配置
- 2.7 频偏温度补偿功能
- 2.8 FAQ

## 2.1 驱动加载与卸载

### 2.1.1 概述

在完成芯片上电后，驱动加载实现对芯片寄存器的初始配置、校准参数读取与写入、软件资源的申请和配置；驱动卸载实现软件资源的释放。

### 2.1.2 开发流程

#### 使用场景

Wi-Fi 驱动初始化为 Wi-Fi 功能提供基本资源配置和芯片初始化，是 Wi-Fi 功能实现的第一步。当需要配置 Wi-Fi 功能时，必须先完成驱动的初始化，Wi-Fi 功能使用完成后，可以使用去初始化完成资源释放也可以使用软复位来完成资源释放。

功能

Wi-Fi 驱动加载与卸载提供的接口如表 2-1 所示。

表2-1 Wi-Fi 驱动加载与卸载接口描述

接口名称	描述
wifi_init	Wi-Fi 驱动初始化。
wifi_deinit	Wi-Fi 驱动去初始化。

开发流程

使用驱动加载与卸载的典型流程：

- 步骤 1 调用 wifi\_init，完成 Wi-Fi 驱动初始化。
- 步骤 2 参考 “2.2 STA 功能” 或 “2.3 SoftAp 功能” 配置 Wi-Fi 功能。
- 步骤 3 调用 wifi\_deinit，完成 Wi-Fi 驱动去初始化。

----结束

返回值

Wi-Fi 驱动加载与卸载的返回值如表 2-2 所示。

表2-2 Wi-Fi 驱动加载与卸载返回值说明

序号	定义	实际数值	描述
1	ERRCODE_SUCC	0x0	执行成功。
2	ERRCODE_FAIL	0xFFFFFFFF	执行失败。

2.1.3 编程实例

示例 1：基于 LiteOS 的 app\_main 函数，在系统初始化时已自动完成 Wi-Fi 驱动的加载，此加载方式无须执行，系统 reboot 时自动完成驱动卸载和加载。

### 代码示例

```
td_void app_main(td_void)
{
    td_u32 ret;

    ret = wifi_init();
    if (ret != 0) {
        printf("fail to init wifi\n");
    } else {
        printf("wifi init success\n");
    }

    //此处添加测试代码

    ret = wifi_deinit();
    if (ret != 0) {
        printf("fail to deinit wifi\n");
    } else {
        printf("wifi deinit success\n");
    }

    return;
}
```

### 结果验证

```
wifi init success
wifi deinit success
```

## 2.2 STA 功能

### 2.2.1 概述

STA 功能即 Station 功能，实现驱动 STA 设备的创建、扫描、关联以及 DHCP，完成通信链路的建立。开发 STA 功能前，须完成驱动加载。

2.2.2 开发流程

使用场景

当需要接入某个网络并与该网络通信时，需要启动 STA 功能。

功能

驱动 STA 功能提供的接口如表 2-3 所示。

表2-3 驱动 STA 功能接口描述

接口名称	描述
wifi_sta_enable	启动 STA 接口。
wifi_sta_set_reconnect_policy	设置 STA 接口自动重连配置。
wifi_register_event_cb	注册事件的回调函数。
wifi_sta_scan	触发 STA 扫描。
wifi_sta_scan_advance	执行带特定参数的扫描。
wifi_sta_get_scan_info	获取 STA 扫描结果。
wifi_sta_connect	触发 STA 连接 Wi-Fi 网络。
wifi_sta_get_ap_info	获取 STA 连接的网络状态。
netifapi_dhcp_start	启动 DHCP 客户端，获取 IP 地址。
netifapi_dhcp_stop	停止 DHCP 客户端。
wifi_sta_disconnect	触发 STA 离开当前网络。
wifi_sta_disable	关闭 STA 接口。
wifi_sta_fast_connect	STA 快速连接接口，连接加密路由器时，不支持 WPA3 的加密方式。
wifi_raw_scan	WiFi 驱动直接发起的扫描。
wifi_set_intrf_mode	是否使能抗干扰模式。

开发流程

STA 功能开发的典型流程：

- 步骤 1 调用 `wifi_sta_enable`，启动 STA。
- 步骤 2（可选，根据需要配置）调用 `wifi_sta_set_reconnect_policy`，设置自动重连。
- 步骤 3 调用 `wifi_sta_scan`（或调用 `aich_wifi_sta_advance_scan`，执行带参数扫描），触发 STA 扫描。
- 步骤 4 调用 `wifi_sta_get_scan_info`，获取扫描结果。
- 步骤 5 根据接入网络需求，自定义筛选扫描结果，调用 `wifi_sta_connect`，进行连接。
- 步骤 6 调用 `wifi_sta_get_ap_info`，查询 Wi-Fi 连接状态。
- 步骤 7 连接成功后，调用 `netifapi_dhcp_start`，启动 DHCP 客户端，获取 IP 地址。
- 步骤 8 调用 `wifi_sta_disconnect`，离开当前连接的网络。
- 步骤 9（可选）调用 `netifapi_dhcps_stop`，停止 DHCP 客户端。
- 步骤 10 调用 `wifi_sta_disable`，关闭 STA（会自动关闭 DHCP 客户端）。

----结束

返回值

STA 功能的返回值如表 2-4 所示。

表2-4 STA 功能返回值说明

序号	定义	实际数值	描述
1	ERRCODE_SUCC	0x0	执行成功。
2	ERRCODE_FAIL	0xFFFFFFFF	执行失败。

2.2.3 注意事项

- 扫描为非阻塞式接口，扫描命令下发成功后需要延迟一段时间后再获取扫描结果，全信道扫描延迟时间建议设置为 1s。

- 可通过指定 SSID、BSSID、信道等带指定参数的扫描，实现更精准地扫描，缩短扫描时间。
- 已知待连接网络的参数时，可省去扫描过程，直接发起连接。
- 连接为非阻塞式接口，连接命令下发成功后，需要通过命令获取连接状态。
- 注册事件回调函数后，Wi-Fi 相关的事件会通过该回调上报用户，用户可根据事件执行后续动作。
- 不支持重复启动 STA，再次启动 STA 时须先执行关闭 STA。
- 关闭 STA 步骤为可选，设备所处的网络地位不变，不需要执行关闭 STA。
- STA 默认支持发送和接收 AMPDU 聚合帧。

## 2.2.4 Sample 用例

### 说明

1. STA Sample 文件位于 application\samples\wifi\sta\_sample 目录；
2. 实现当 STA Sample 软件版本成功烧录后，单板启动时会自启动 STA，并不停的扫描，直到发现存在 SSID 为 “my\_softAP” 的目标 AP，然后会通过密码 “my\_password” 进行连接，若连接失败，则重复扫描动作，若连接成功，则进一步获取动态 IP 地址，获取 IP 成功后，串口会打印 “STA connect success.”

步骤 1 在 SDK 包的根目录下，执行命令 “python3 build.py ws63-liteos-app menuconfig” 进入 menuconfig。

步骤 2 依次选择 Application -> Enable Sample -> Enable the Sample of WIFI -> Sample -> Support WIFI STA Sample，并按 S 键保存，然后通过 Esc 键退出 menuconfig

步骤 3 在 SDK 包的根目录下，执行命令 “python3 build.py ws63-liteos-app”，即可编译 STA Sample 软件版本

----结束

## 2.3 SoftAp 功能

### 2.3.1 概述

SoftAp 功能提供网络接入点供其他 STA 接入，并对接入的 STA 提供 DHCP Server 服务。

2.3.2 开发流程

使用场景

当需要创建一个网络接入点，供其他设备接入并共享网络内的数据时，需要使用 SoftAp 功能。

功能

驱动 SoftAp 功能提供的接口如表 2-5 所示。

表2-5 驱动 SoftAp 功能接口描述

接口名称	描述
wifi_softap_enable	启动 SoftAp 接口。
wifi_set_softap_config_advance	设置 SoftAp 协议模式、beacon 周期、dtim 周期、秘钥更新时间、是否隐藏 SSID、GI 参数。
wifi_get_softap_config_advance	获取 SoftAp 协议模式、beacon 周期、dtim 周期、秘钥更新时间、是否隐藏 SSID、GI 参数
netifapi_netif_set_addr	设置 SoftAp 的 DHCP 服务器的 IP 地址、子网掩码和网关参数。
netifapi_dhcps_start	启动 SoftAp 的 DHCP 服务器。
netifapi_dhcps_stop	停止 SoftAp 的 DHCP 服务器。
wifi_softap_get_sta_list	获取当前接入的 STA 信息。
wifi_softap_deauth_sta	断开指定 STA 的连接。
wifi_softap_disable	关闭 SoftAp 接口。

开发流程

SoftAp 功能开发的典型流程：

- 步骤 1（可选）调用 wifi\_set\_softap\_config\_advance，设置 SoftAp 协议模式、beacon 周期、dtim 周期、秘钥更新时间、是否隐藏 SSID、GI 参数。



- 步骤 2 调用 `wifi_softap_start`，启动 SoftAp。
- 步骤 3 调用 `netifapi_netif_set_addr`，配置 DHCP 服务器。
- 步骤 4 调用 `netifapi_dhcps_start`，启动 DHCP 服务器。
- 步骤 5（可选）调用 `netifapi_dhcps_stop`，停止 DHCP 服务器。
- 步骤 6 调用 `aich_wifi_softap_stop`，关闭 SoftAp（会自动关闭 DHCP 服务器）。

----结束

返回值

SoftAp 功能的返回值如表 2-6 所示。

表2-6 SoftAp 功能返回值说明

序号	定义	实际数值	描述
1	ERRCODE_SUC C	0x0	执行成功。
2	ERRCODE_FAIL	0xFFFFF FFF	执行失败。

2.3.3 注意事项

- SoftAp 的网络参数为可选配置，无特殊要求均可使用初始默认值。
- SoftAp 默认启动 20M 带宽 SoftAp。
- SoftAp 的网络参数在关闭 SoftAp 时不会重置，会继续沿用上一次配置，重启单板可恢复至初始默认值。
- SoftAp 模式下最大关联用户数限制：
  - 最大关联用户不超过 6 个。

## 2.3.4 Sample 用例

### 📖 说明

1. SoftAp Sample 文件位于 application\samples\wifi\softap\_sample 目录
2. 实现当 SoftAP Sample 软件版本成功烧录后，单板启动时会自启动 SoftAp，其加密方式为 wpa/wpa2，SSID 为 “my\_softAP”，密码为 “my\_password”，IP 地址默认设置为 192.168.43.1，网关地址默认设置为 192.168.43.2

步骤 1 在 SDK 包的根目录下，执行命令 “python3 build.py ws63-liteos-app menuconfig” 进入 menuconfig。

步骤 2 依次选择 Application -> Enable Sample -> Enable the Sample of WIFI -> Sample -> Support WIFI SoftAP Sample，并按 S 键保存，然后通过 Esc 键退出 menuconfig

步骤 3 在 SDK 包的根目录下，下发命令 “python3 build.py ws63-liteos-app”，即可编译 SoftAP Sample 软件版本

----结束

## 2.4 STA&SoftAp 共存

### 2.4.1 概述

STA&SoftAp 共存即 STA 功能和 SoftAp 功能同时工作，仅支持同信道共存。

### 2.4.2 开发流程

#### 使用场景

配网时，产品先启动 SoftAp，手机关联 SoftAp 后发送家居网络的 SSID 和密码给产品，产品获取到家居网络的连接参数后启动 STA 关联家居网络，完成产品联网，产品联网成功后，关闭 SoftAp，只保留 STA 作为端侧长期保持连接。共存场景可视产品形态和需求自行使用。

#### 功能

共存功能分别使用 STA 功能和 SoftAp 功能的 API 接口，无额外新增 API 接口。

## 开发流程

配网模式下共存功能开发的典型流程：

- 步骤 1 创建 SoftAp 网络接口（详细内容请参见“2.3 SoftAp 功能”）。
- 步骤 2 手机关联 SoftAp，并通过手机 APP 发送家居网络 SSID 和密码。
- 步骤 3 创建 STA 网络接口，并根据 SSID 和密码完成关联（详细内容请参见“2.2 STA 功能”）。
- 步骤 4 关闭 SoftAp（详细内容请参见“2.3 SoftAp 功能”）。

----结束

## 返回值

返回值请参见对应模块功能的返回值说明。

### 2.4.3 编程实例

请参考 STA 和 SoftAp 功能的编程实例（详细内容请参见“2.2 STA 功能”或“2.3 SoftAp 功能”）。

## 2.5 Wi-Fi&蓝牙共存

### 2.5.1 概述

蓝牙（BT，Bluetooth）和 Wi-Fi 均可能工作在 2.4G ISM，因此可能互相干扰。分时是利用蓝牙和 Wi-Fi 间的握手信号，使蓝牙和 Wi-Fi 分时在 2.4G 工作，这样可以避免噪音干扰和阻塞干扰。

802.15.2 规定仲裁方式和信号（PTA，Packet Traffic Arbitration）的框架，在蓝牙或 Wi-Fi 有收发业务时，提交申请给 PTA controller（集成在 Wi-Fi 中），由 PTA controller 进行许可。

蓝牙和 Wi-Fi 间的握手信号定义如下：

- Wi-Fi 给 PTA 信号 wl\_tx\_status：Wi-Fi 有发包业务。
- Wi-Fi 给 PTA 信号 wl\_rx\_status：Wi-Fi 有收包业务。
- Wi-Fi 给 PTA 信号 wl\_priority：Wi-Fi 业务状态，Wi-Fi 高优先级。

- Wi-Fi 给 PTA 信号 `wl_occupied`: Wi-Fi 业务状态, Wi-Fi 最高优先级。
- 蓝牙给 PTA 信号 `bt_status`: 蓝牙有收发业务。
- 蓝牙给 PTA 信号 `bt_priority`: 蓝牙业务状态, 指示蓝牙优先级。

## 2.5.2 开发流程

### 使用场景

需要同时使用 Wi-Fi 和蓝牙时, 开启 Wi-Fi&BT 共存功能。

### 开发流程

Wi-Fi&BT 共存功能开发的典型流程:

步骤 1 加载蓝牙后, 执行蓝牙共存初始化函数

步骤 2 创建 STA 网络接口 (详细内容请参见 “2.2 STA 功能” )。

步骤 3 创建蓝牙业务(详情内容请参见 “3 BLE&SLE 软件开发” )

----结束

## 2.5.3 注意事项

- Wi-Fi 与 BT 共存支持 STA 模式、支持 SoftAp 模式。
- 模组或产品内同时集成了 Wi-Fi 芯片和蓝牙芯片, 常开 Wi-Fi&BT 共存功能。
- 不支持外部共存

## 2.6 国家码功能配置

### 2.6.1 使用背景

对于发货给不同国家的用户, 希望使用同一套固件, 可以通过修改 NV 中的管制域信息 (包括国家码, 信道, 发射功率等) 实现。

2.6.2 使用指南

步骤 1 在 NV 配置文件 `middleware/chips/ws63/nv/nv_config/cfg/acore/app.json` 中, NV ID `"=0x2003"` 的表项用于确定国家码, value 的含义是国家码对应 ASIC 码, 例如: 十进制 67, 78 分别对应 ASIC 码中的 'C', 'N'

```
"country":{
  "key_id": "0x2003",
  "key_status": "alive",
  "structure_type": "country_type_t",
  "attributions": 2,
  "value": [[67,78]]
},
```

国家码与四个区域管制域的对应关系如下。

key_id	区域	国家码
0x2053	北美	US,CA,KH
0x2054	欧洲	RU,AU,MY,ID,TR,PL,FR,PT,IT,DE,ES,AR,ZA,MA,PH,TH,GB,CO,MX,EC,PE,CL,SA,EG,AE
0x2055	日本	JP
0x2056	亚太	CN

步骤 2 基于步骤 1 确定的对应关系, 刷新对应的发射功率表项, 支持调整不同速率的目标功率, 以及调整在不同工作信道下的最大功率。

刷新 0x2053, 0x2054, 0x2055, 0x2056 表项:

```
"fe_tx_power_fcc":{
  "key_id": "0x2053",
  "key_status": "alive",
  "structure_type": "fe_tx_power_type_t",
  "attributions": 2,
  "value": [
    [230],
    [46, 46, 46, 43, 42, 42, 42, 42, 42, 42, 40, 38, 40, 40, 40, 37, 37, 37, 37, 36, 33, 30, 40, 40,
    40, 37, 37, 37, 37, 36, 33, 30, 22],
    [60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60,
    60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60,
    60, 60, 60, 60, 60, 60],
  ]
},
```

```
        [60, 60, 60]
    ],
},
"fe_tx_power_etsi":{
    "key_id": "0x2054",
    "key_status": "alive",
    "structure_type": "fe_tx_power_type_t",
    "attributions": 2,
    "value": [
        [230],
        [46, 46, 46, 43, 42, 42, 42, 42, 42, 42, 40, 38, 40, 40, 40, 37, 37, 37, 37, 36, 33, 30, 40, 40,
40, 37, 37, 37, 37, 36, 33, 30, 22],
        [60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60,
60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60,
60, 60, 60, 60, 60, 60],
        [60, 60, 60]
    ]
},
"fe_tx_power_japan":{
    "key_id": "0x2055",
    "key_status": "alive",
    "structure_type": "fe_tx_power_type_t",
    "attributions": 2,
    "value": [
        [230],
        [46, 46, 46, 43, 42, 42, 42, 42, 42, 42, 40, 38, 40, 40, 40, 37, 37, 37, 37, 36, 33, 30, 40, 40,
40, 37, 37, 37, 37, 36, 33, 30, 22],
        [60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60,
60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60,
60, 60, 60, 60, 60, 60],
        [60, 60, 60]
    ]
},
"fe_tx_power_common":{
    "key_id": "0x2056",
    "key_status": "alive",
    "structure_type": "fe_tx_power_type_t",
    "attributions": 2,
    "value": [
        [230],
        [46, 46, 46, 43, 42, 42, 42, 42, 42, 42, 40, 38, 40, 40, 40, 37, 37, 37, 37, 36, 33, 30, 40, 40,
40, 37, 37, 37, 37, 36, 33, 30, 22],
        [60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60,
60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60,
60, 60, 60, 60, 60, 60]
```

```
60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60,
60, 60, 60, 60, 60, 60],
    [60, 60, 60]
]
},
```

对于上述发射功率的每个表项中的参数值结构参见 fe\_tx\_power\_type\_t:

```
#define WLAN_RF_FE_MAX_POWER_NUM 1
#define WLAN_RF_FE_TARGET_POWER_NUM 33
#define WLAN_RF_FE_LIMIT_POWER_NUM 56
#define WLAN_RF_FE_SAR_POWER_NUM 3
typedef struct {
    uint8_t chip_max_power[WLAN_RF_FE_MAX_POWER_NUM];
    uint8_t target_power[WLAN_RF_FE_TARGET_POWER_NUM];
    uint8_t limit_power[WLAN_RF_FE_LIMIT_POWER_NUM];
    uint8_t sar_power[WLAN_RF_FE_SAR_POWER_NUM];
} fe_tx_power_type_t;
```

其中:

chip\_max\_power 是最大发射功率, 当前已更新为芯片实际能力。单位 0.1dB。

target\_power 是不同协议速率下的目标功率, 依次分别是 11b 协议(1M, 2M, 5.5M, 11M), 11g 协议(6M, 9M, 12M, 18M, 24M, 36M, 48M, 54M), 11n/11ax 协议 20M(mcs0~mcs9), 11n/11ax 协议 40M(mcs0~mcs9 以及 mcs32)。单位 0.5dB。

limit\_power 是按信道划分的限制功率, 总共 14 个信道, 每个信道 4 个限制功率值, 分别对应 11b 协议、11g 协议、11n/11ax 协议 20M、11n/11ax 协议 40M。单位 0.5dB。

sar\_power 是比吸收率功率限制值, 共三个值用于选择。单位 0.5dB。

步骤 3 刷新 NV 配置文件后重新编译 NV 固件并加载。NV 支持配置四个区域的发射功率。用户可以调用 AT 命令 AT+CC=\${COUNTRY} 实现国家码变更, 从而按照国家码所处的区域更新功率表。

其中\${COUNTRY} 从步骤 1 中的 region\_country\_map 中取。

----结束

#### 说明

- 最大发射功率不支持修改, 已按照芯片能力配置, 修改可能导致发射功率异常。
- 用户自定义发射功率表项时, 不得超过最大的射频发射功率。

- 11n 支持 20M 和 40M，最大支持速率 mcs7，不支持 mcs32；11ax 不支持 40M
- AT+CC 命令需要在上电之后，并且关联上用户之前进行调用。

## 2.7 频偏温度补偿功能

### 2.7.1 使用背景

由于射频频偏在不同工作温度下会发生变化，为了满足发射信号的频偏在规格范围内，可开启频偏温度补偿功能来针对特定温度出现频偏过大的情况进行补偿，使得频偏能够继续保持在规格范围内。

### 2.7.2 使用指南

- 步骤 1 参见《WS63V100 产线工装 用户指南》完成软件版本加载，并参考章节“STA 模式 冒烟测试”中测试步骤 4 完成 WiFi 初始化，再参考步骤 5 第 1 节完成常发。
- 步骤 2 获取常温下频偏调节的能力  $F_{avg}$ 。参考 WS63V100 产线工装 用户指南的章节 2.2.2 的步骤 5 第 4 节配置细调频偏值，分别配置 value 为 0 和 127，记录下 value 为 0 时的频偏值  $F_0$ ，value 为 127 时的频偏值  $F_{127}$ 。计算得到频偏补偿能力  $F_{avg}=(F_{127}-F_0)/127$ 。
- 步骤 3 记录工作温度下的频偏值。重新启动单板，并按步骤 1 完成常发。再调节温箱温度，从低温遍历到高温，可按实际使用场景确认温度范围，在不同温箱温度下，观察信号频偏随温度变化，记录芯片温度  $T_i$  (参考 WS63V100 产线工装 用户指南的章节 2.2.2 的步骤 5 第 5 节)以及频偏值  $F_i$ 。 $T_i$  值推荐  $T_0=-30$ ， $T_1=-10$ ， $T_2=10$ ， $T_3=30$ ， $T_4=50$ ， $T_5=70$ ， $T_6=90$ ， $T_7=110$ 。
- 步骤 4 计算频偏温度补偿值并使用补偿。根据步骤 3 中得到的频偏值  $F_i$ ，确认需要补偿的温度  $T_i$ ，计算补偿值  $FC_i=F_i/F_{avg}$  (对于不需要补偿的温度点， $FC_i=0$ )，结果四舍五入取整，并保证在取值范围内 $[-127,127]$ ，超过范围的按范围边界值配置。将计算得到的  $FC_i$  填写至 NV 表项 key\_id 为 0x7 对应结构中，并使能 key\_id 为 0x6 的补偿开关。

```
"xo_trim_temp_param":{
  "key_id": "0x7",
  "key_status": "alive",
  "structure_type": "xo_trim_temp_type_t",
  "attributions": 1,
  "value": [[FC0,FC1,FC2,FC3,FC4,FC5,FC6,FC7]]
```



```
},  
"xo_trim_temp_sw":{  
"key_id": "0x6",  
"key_status": "alive",  
"structure_type": "uint8_t",  
"attributions": 1,  
"value": 1  
},
```

----结束

#### 说明

- 频偏温度补偿功能是基于产线频偏校准的，是作为优化功能，请保证已完成产线频偏校准，具体流程可参考 WS63V100 产线工装 用户指南的章节 2.2.2。
- 获取频偏温度补偿值时请使用温箱并保持温度稳定。
- 频偏补偿值可基于同一批次单板的平均数据获取。

## 2.8 FAQ

### 2.8.1 频偏修正

若遇到信号频偏较大，在没有频偏产测校准的情况下，可以考虑通过修改默认的频偏补偿值来修正频偏。请参考以下流程：

步骤 1 在频偏补偿配置所在文件 application/ws63/ws63\_liteos\_application/clock\_init.c 中找到频偏补偿宏

RG\_CMU\_XO\_TRIM\_COARSE

步骤 2 根据当前频偏的情况，如果频偏为正，则将频偏补偿宏的低 8 位调大；反之则调小

步骤 3 修改后确认信号频偏，满足要求则可以按新的配置值更新频偏补偿宏

----结束

# 3 BLE&SLE 软件开发

- 3.1 BLE 开发流程
- 3.2 星闪开发流程
- 3.3 BLE&SLE 功率档位定制

## 3.1 BLE 开发流程

### 3.1.1 概述

WS63V100 通过 API (Application Programming Interface) 面向开发者提供 BLE 功能的开发和应用接口, 包括 GAP、GATT server 和 GATT client 接口。

各组件功能说明如下:

- GAP: 通用访问协议 (Generic Access Profile), 包含蓝牙本地设置和低功耗蓝牙的发现和连接接口。
- GATT: 通用属性协议 (Generic Attribute Profile), 包含服务注册、服务发现等功能相关接口。

#### 说明

该文档描述各个模块功能的基本流程和 API 接口描述。

### 3.1.2 GAP 接口

#### 3.1.2.1 概述

GAP 实现蓝牙设备开关控制、设备信息管理、广播管理、主动连接和断开连接等功能。

3.1.2.2 开发流程

使用场景

打开蓝牙设备开关是使用蓝牙功能的首要条件，蓝牙启动后可进行设备信息管理，包括获取与设置本地设备名称、获取本地设备地址、获取配对信息、获取远端设备名称/设备类型/接收信号强度等。

当蓝牙设备需要被动与对端设备建立连接时，可设置广播参数并启动广播等待对端连接；当蓝牙设备需要主动与对端设备建立连接时，可向对端发起主动连接；当对端地址已知时，用户可直接向对端发起主动连接；当对端地址未知时，可打开蓝牙设备的扫描功能，获取正在广播的设备信息，并向对端发起主动连接；当蓝牙设备处于连接状态时，可获取设备连接信息；当蓝牙设备不需要与对端设备保持连接时，可主动断开连接。

功能

GAP 提供的接口如下表所示。

接口名称	描述	入参说明	返回信息说明
enable_ble	使能 BLE。	-	接口返回值：错误码。
disable_ble	去使能 BLE。	-	接口返回值：错误码。
gap_ble_set_local_addr	设置本地设备地址。	mac：本地设备地址指针； len：本地设备地址长度。	接口返回值：错误码。
gap_ble_get_local_addr	获取本地设备地址。	mac：本地设备地址指针； len：本地设备地址长度	本地设备地址存储在入参 mac 中； 接口返回值：错误码。
gap_ble_set_local_name	设置本地设备名称。	local_name：本地设备名称指针； length：本地	接口返回值：错误码。

接口名称	描述	入参说明	返回信息说明
		设备名称长度。	
gap_ble_set_local_appearance	设置本地设备 appearance。	appearance: 本地设备外貌。	接口返回值：错误码。
gap_ble_get_local_name	获取本地设备名称。	local_name: 本地设备名称指针； length: 本地设备名称长度。	本地设备名称存储在入参 local_name 中； 接口返回值：错误码。
gap_ble_get_paired_devices_number	获取 BLE 配对设备数量。	number: 配对设备数量指针。	配对设备数量存储在入参 number 中； 接口返回值：错误码。
gap_ble_get_paired_devices	获取 BLE 配对设备。	number: 配对设备数量指针； addr: 配对设备地址指针。	配对设备数量存储在入参 number 中； 配对设备地址存储在入参 addr 中； 接口返回值：错误码。
gap_ble_get_pair_state	获取 BLE 设备的配对状态。	addr: 对端设备地址。	接口返回值：配对状态 (GAP_PAIR_NONE、GAP_PAIR_PAIRING、GAP_PAIR_PAIRING) 。
gap_ble_remove_all_pairs	删除所有配对设备。	-	接口返回值：错误码。
gap_ble_remove_pair	删除配对设备	addr: 配对设备地址。	接口返回值：错误码。
gap_ble_disconnect_remote_device	断开 BLE 设备连接。	addr: 对端设备地址。	接口返回值：错误码。

接口名称	描述	入参说明	返回信息说明
gap_ble_connect_remote_device	与设备建立 ACL 连接。	addr: 对端设备地址。	接口返回值: 错误码。
gap_ble_pair_remote_device	与已连接设备进行配对。	addr: 对端设备地址。	接口返回值: 错误码。
gap_ble_connect_param_update	连接参数更新。	params: 待更新连接参数。	接口返回值: 错误码。
gap_ble_set_adv_data	设置 BLE 广播数据。	adv_id: 广播 id;  data: 设置的广播数据。	接口返回值: 错误码。
gap_ble_set_adv_param	设置广播参数。	adv_id: 广播 id;  param: 设置的广播数据。  <b>注: 使用板端地址发送广播时, own_addr 应设置成全 0。</b>	接口返回值: 错误码。
gap_ble_start_adv	启动 BLE 广播。	adv_id: 广播 id;	接口返回值: 错误码。
gap_ble_stop_adv	停止 BLE 广播。	adv_id: 广播 id。	接口返回值: 错误码。
gap_ble_set_scan_parameters	设置扫描参数。	param: 设置的扫描参数。	接口返回值: 错误码。
gap_ble_start_scan	启动扫描。	-	接口返回值: 错误码。
gap_ble_stop_scan	停止扫描。	-	接口返回值: 错误码。

接口名称	描述	入参说明	返回信息说明
gap_ble_register_callbacks	注册 BLE GAP 回调。	func: 用户回调函数。	接口返回值: 错误码。
bth_ota_init	初始化 bth ota 通道。  <b>注: 在收到 ble 使能成功的回调之后调用。</b>	-	接口返回值: 错误码。

具体操作流程:

GAP 开发的具体编程实例可参考 application/samples/bt。

GAP 开发的典型流程(指令中的数据可根据按照 AT 命令使用指南自行修改):

Slave:

- 步骤 1 调用 gap\_ble\_register\_callbacks 注册用户回调函数。
- 步骤 2 调用 enable\_ble, 打开蓝牙开关。
- 步骤 3 调用 gap\_ble\_set\_local\_addr, 设置本地蓝牙地址。
- 步骤 4 调用 gap\_ble\_set\_local\_name, 设置本地设备名称。
- 步骤 5 调用 gap\_ble\_set\_adv\_param, 设置广播参数
- 步骤 6 调用 gap\_ble\_set\_adv\_data, 设置广播数据
- 步骤 7 调用 gap\_ble\_start\_adv, 启动广播。

----结束

Master:

- 步骤 1 调用 gap\_ble\_register\_callbacks 注册用户回调函数。
- 步骤 2 调用 enable\_ble, 打开蓝牙开关。
- 步骤 3 调用 gap\_ble\_set\_local\_addr, 设置本地蓝牙地址。
- 步骤 4 调用 gap\_ble\_set\_local\_name, 设置本地设备名称。

步骤 5 调用 gap\_ble\_set\_scan\_parameters，设置扫描参数

步骤 6 调用 gap\_ble\_start\_scan，启动扫描

步骤 7 调用 gap\_connect\_remote\_device，连接到目标设备。

步骤 8 调用 gap\_ble\_pair\_remote\_device，与目标设备配对

---结束

返回值

获取配对状态返回值如下表所示。

序号	定义	实际数值	描述
1	GAP_PAIR_NONE	1	未配对。
2	GAP_PAIR_PAIRING	2	配对中。
3	GAP_PAIR_PAIED	3	已配对。

3.1.2.3 注意事项

- WS63V100 产品可支持 8 路蓝牙连接。
- 若扫描不到设备，请先检查设备是否已在配对设备列表中，或者设备是否已与其他设备配对（此情况下需要先清除设备端配对信息）。

3.1.3 GATT server 接口

3.1.3.1 概述

GATT 是一个基于蓝牙 GAP 连接的发送和接收数据的通用规范，支持在两个蓝牙设备间进行数据传输。

3.1.3.2 开发流程

使用场景

GATT server 主要接收对端设备的命令和请求，给对端设备发送响应、指示或者通知。

功能

GATT server 提供的接口如下表所示。

接口名称	描述	入参说明	返回信息说明
gatts_register_server	Gatt server 注册。根据传入的 UUID 注册 server，回调函数返回 server 接口 ID。 <b>注：目前只支持注册一个 GATT server。</b>	app_uuid：应用 UUID 指针； server_id：服务端 ID 指针。	服务端 ID 存储在 server_id 中； 接口返回值：错误码。
gatts_unregister_server	注销 gatt 服务端。	server_id：服务器 ID。	接口返回值：错误码。
gatts_add_service	添加 service。	server_id：服务器 ID； service_uuid：服务 UUID； is_primary：是否是首要服务。	接口返回值：错误码。
gatts_add_characteristic	添加 characteristic 到指定的 service。	server_id：服务器 ID； service_handle：服务句柄； character：特征信息。	接口返回值：错误码。
gatts_add_descriptor	添加 descriptor 到对应的 characteristic。 包含当前 Characteristic 的描述信息、配置信息、表示格式信息等。	server_id：服务器 ID； service_handle：服务句柄； descriptor：描述符信息。	接口返回值：错误码。
gatts_add_service_sync	添加一个 gatt 服务同步接口，服务句柄同步返回。	server_id：服务器 ID；	服务句柄存储在 handle 中；



接口名称	描述	入参说明	返回信息说明
		service_uuid: 服务 UUID;  is_primary: 是否是首要服务;  handle: 服务句柄指针。	接口返回值: 错误码。
gatts_add_characteristic_sync	添加一个 gatt 特征同步接口, 特征句柄同步返回。	server_id: 服务器 ID;  service_handle: 服务 UUID;  character: GATT 特征;  handle: 特征句柄指针。	特征句柄存储在 handle 中;  接口返回值: 错误码。
gatts_add_descriptor_sync	添加一个 gatt 特征描述符同步接口, 特征描述符句柄同步返回。	server_id: 服务器 ID;  service_handle: 服务 UUID;  character: 特征描述符;  handle: 特征描述符句柄指针。	特征描述符句柄存储在 handle 中;  接口返回值: 错误码。
gatts_start_service	启动 service。	server_id: 服务器 ID;  service_handle: 服务句柄。	接口返回值: 错误码。
gatts_delete_all_services	删除所有 GATT 服务。	server_id: 服务器 ID;	接口返回值: 错误码。
gatts_send_response	用户发送响应。	server_id: 服务器 ID;	接口返回值: 错误码。

接口名称	描述	入参说明	返回信息说明
		conn_id: 连接 ID; param: 响应参数。	
gatts_notify_indicate	给远端 client 发送 indication/notification。	server_id: 服务器 ID; conn_id: 连接 ID; param: 通知或指示参数。	接口返回值: 错误码。
gatts_notify_indicate_by_uuid	按照 UUID 给远端 client 发送 indication/notification。	server_id: 服务器 ID; conn_id: 连接 ID; param: 通知或指示参数。	接口返回值: 错误码。
gatts_set_mtu_size	在连接之前设置服务端接收 mtu。	server_id: 服务器 ID; mtu_size: 服务端接收 mtu 值。	接口返回值: 错误码。
gatts_register_callbacks	注册 GATT server 回调函数。	func: 用户回调函数。	接口返回值: 错误码。

开发流程

GATT server 开发具体编程实例可参考 application/samples/bt。

GATT server 开发的典型流程：添加服务和特征及描述信息并启动服务。

- 步骤 1 调用 gatts\_register\_callbacks，注册 GATT server 用户回调函数。
- 步骤 2 调用 enable\_ble，打开蓝牙开关。
- 步骤 3 调用 gatts\_register\_server，创建一个 server。
- 步骤 4 调用 gatts\_add\_service，根据 UUID 创建 service。
- 步骤 5 调用 gatts\_add\_characteristic，对创建的服务添加特征值。

- 步骤 6 调用 gatts\_add\_descriptor，对服务中的特征添加描述信息。
- 步骤 7 调用 gatts\_start\_service，启动 service。
- 步骤 8 启动广播，等待对端连接。
- 步骤 9 被对端使能为“可通知”后，调用 gatts\_notify\_indicate 或 gatts\_notify\_indicate\_by\_uuid 向对端发起特征值通知。
- 结束

3.1.4 GATT client 接口

3.1.4.1 概述

GATT 是一个基于蓝牙 GAP 连接的发送和接收数据的通用规范，支持在两个蓝牙设备间进行数据传输。

3.1.4.2 开发流程

使用场景

GATT client 主要给对端发送命令和请求，接收对端回复的响应、指示和通知。

功能

GATT client 提供的接口如下表所示。

接口名称	描述	入参说明	返回信息说明
gattc_register_client	注册 GATT client。	app_uuid：应用 UUID； client_id：客户端 ID 指针。	客户端 id 存储在 client_id 中 接口返回值：错误码。
gattc_unregister_client	取消注册 GATT client。	client_id：客户端 ID。	接口返回值：错误码。
gattc_discover_service	发现服务。	client_id：客户端 ID； conn_id：连接 ID； uuid：服务 UUID。	接口返回值：错误码。

接口名称	描述	入参说明	返回信息说明
gattc_discover_character	发现特征。	client_id: 客户端 ID; conn_id: 连接 ID; param: 发现的特征参数。	接口返回值: 错误码。
gattc_discover_descriptor	发现特征描述符。	client_id: 客户端 ID; conn_id: 连接 ID; character_handle: 特征声明句柄。	接口返回值: 错误码。
gattc_read_req_by_handle	发起按照句柄读取请求。	client_id: 客户端 ID; conn_id: 连接 ID; handle: 句柄。	接口返回值: 错误码。
gattc_read_req_by_uuid	发起按照 UUID 读取请求。	client_id: 客户端 ID; conn_id: 连接 ID; param: 按照 UUID 读取请求参数。	接口返回值: 错误码。
gattc_write_req	发起写请求。	client_id: 客户端 ID; conn_id: 连接 ID; param: 写请求参数。	接口返回值: 错误码。
gattc_write_cmd	发起写命令。	client_id: 客户端 ID; conn_id: 连接 ID; param: 写命令参数。	接口返回值: 错误码。
gattc_exchange_mtu_req	发送交换 mtu 请求。	client_id: 客户端 ID; conn_id: 连接 ID; mtu_size: 客户端接收 mtu。	接口返回值: 错误码。
gattc_register_callbacks	注册 GATT client 回调函数。	func: 用户回调函数。	接口返回值: 错误码。

开发流程

GATT client 开发的具体编程实例可参考 application/samples/bt。

GATT client 开发的典型流程：连接对端设备，发现对端设备的服务，读写对端特征值，订阅对端的通知或者指示。

- 步骤 1 调用 gattc\_register\_callbacks，注册 GATT client 用户回调函数。
- 步骤 2 调用 enable\_ble，打开蓝牙开关。
- 步骤 3 调用 gattc\_register\_client，创建一个 client。
- 步骤 4 递归调用 gattc\_discovery\_service，gattc\_discovery\_character 和 gattc\_discovery\_descriptor，获取对端的属性数据库。
- 步骤 5 调用 gattc\_write\_req 或 gattc\_write\_cmd 将关注的对端特征的客户端特征配置写为 0x0001 或 0x0002，设置为前者时可收到关注特征的特征通知，设置为后者时可收到关注特征的特征指示。
- 步骤 6 调用相应读写接口操作 GATT server 的特征和描述符。

----结束

3.1.5 错误码

BLE 错误码返回值如下表所示。

序号	定义	实际数值	描述
1	ERRCODE_BT_SUCCESS	0x0	执行成功错误码。
2	ERRCODE_BT_FAIL	0x80006000	执行失败错误码。
3	ERRCODE_BT_NOT_READY	0x80006001	执行状态未就绪错误码。
4	ERRCODE_BT_MALLOC_FAIL	0x80006002	内存不足错误码。
5	ERRCODE_BT_MEMCPY_FAIL	0x80006003	内存拷贝错误错误码。
6	ERRCODE_BT_BUSY	0x80006004	繁忙无法响应错误码。

序号	定义	实际数值	描述
7	ERRCODE_BT_DONE	0x80006005	执行完成错误码。
8	ERRCODE_BT_UN SUPPORTED	0x80006006	不支持错误码。
9	ERRCODE_BT_PARAM_ERR	0x80006007	无效参数错误码。
10	ERRCODE_BT_STATE_ERR	0x80006008	状态错误。
11	ERRCODE_BT_UNHANDLED	0x80006009	未处理错误码。
12	ERRCODE_BT_AUTH_FAIL	0x8000600A	鉴权失败错误码。
13	ERRCODE_BT_RMT_DEV_DO WN	0x8000600B	远端设备关闭错误码。
14	ERRCODE_BT_AUTH_REJEC TED	0x8000600C	鉴权被拒错误码。

## 3.2 星闪开发流程

### 3.2.1 概述

WS63V100 通过 API（Application Programming Interface）面向开发者提供 SLE 功能的开发和应用接口，包括 Device Discovery, Connection Manager, SSAP 等。

各组件功能说明如下：

- Device Discovery：星闪设备发现协议，包括设备管理、设备公开和设备发现接口。
- Connection Manager：星闪连接管理协议，包括设备连接、配对相关接口。
- SSAP：星闪服务交互协议（SparkLink Service Access Protocol），包含服务注册、服务发现、属性数据读写等功能相关接口。
- Low Latency：低时延初始化和低时延数据收发接口。

#### 说明

该文档描述各个模块功能的基本流程和 API 接口描述。

### 3.2.2 Device Discovery 接口

#### 3.2.2.1 概述

Device Discovery 接口是星闪设备发现协议的软件实现，主要功能有 SLE 设备开关、设备管理、设备公开和设备发现。

#### 3.2.2.2 开发流程

##### 使用场景

打开 SLE 设备开关是使用 SLE 功能的首要条件，SLE 启动后可进行设备信息管理，包括获取与设置本地设备名称、获取与设置本地设备地址和设置本地设备外观。

- 当 SLE 设备需要进行设备公开时，可先设置设备公开参数、设备公开数据，然后使能设备公开。
- 当 SLE 设备需要进行设备发现时，可先设置设备发现参数，然后使能设备发现，并通过回调函数观察发现到的设备公开数据包。

##### 功能

Device Discovery 提供的接口如下表所示。

接口名称	描述	参数说明	返回信息说明
enable_sle	使能 SLE。	-	接口返回值：错误码。
disable_sle	去使能 SLE。	-	接口返回值：错误码。
sle_set_local_addr	设置本地设备地址。	addr: 本地设备地址。	接口返回值：错误码。
sle_get_local_addr	获取本地设备地址。	addr: [out]本地设备地址。	接口返回值：错误码。
sle_set_local_name	设置本地设备名称。	name: 本地设备名称； len: 本地设备名称长度。	接口返回值：错误码。

接口名称	描述	参数说明	返回信息说明
sle_get_local_name	获取本地设备名称。	name: [out] 本地设备名称;  len: [inout]入参时为用户预留内存大小, 出参时为本地设备名称长度。	接口返回值: 错误码。
sle_set_announcement_data	设置设备公开数据。	announce_id : 设备公开ID;  data: 设备公开数据。	接口返回值: 错误码。
sle_set_announcement_param	设置设备公开参数。	announce_id : 设备公开ID;  data: 设备公开参数。	接口返回值: 错误码。
sle_start_announcement	开始设备公开。	announce_id : 设备公开ID。	接口返回值: 错误码。
sle_stop_announcement	停止设备公开。	announce_id : 设备公开ID。	接口返回值: 错误码。
sle_set_seek_param	设置设备发现参数。	param: 设备发现参数。	接口返回值: 错误码。
sle_start_seek	开始设备发现。	-	接口返回值: 错误码。
sle_stop_seek	停止设备发现。	-	接口返回值: 错误码。
sle_announce_s	注册设备公	func: 用户回	接口返回值: 错误码。



接口名称	描述	参数说明	返回信息说明
seek_register_callbacks	开和设备发现回调函数。	调函数。	

开发流程

Device Discovery 开发的典型流程如下，具体编程实例可参考 application/samples/bt。

Terminal Node:

- 步骤 1 调用 enable\_sle，打开 SLE 开关。
- 步骤 2 调用 sle\_announce\_seek\_register\_callbacks，注册设备公开和设备发现回调函数。
- 步骤 3 调用 sle\_set\_local\_addr，设置本地设备地址。
- 步骤 4 调用 sle\_set\_local\_name，设置本地设备名称。
- 步骤 5 调用 sle\_set\_announce\_param，设置设备公开参数
- 步骤 6 调用 sle\_set\_announce\_data，设置设备公开数据
- 步骤 7 调用 sle\_start\_announce，启动设备公开。

----结束

Grant Node:

- 步骤 1 调用 enable\_sle，打开 SLE 开关。
- 步骤 2 调用 sle\_announce\_seek\_register\_callbacks，注册设备公开和设备发现回调函数。
- 步骤 3 调用 sle\_set\_local\_addr，设置本地设备地址。
- 步骤 4 调用 sle\_set\_local\_name，设置本地设备名称。
- 步骤 5 调用 sle\_set\_seek\_param，设置设备发现参数。
- 步骤 6 调用 sle\_start\_seek，启动设备发现，并在回调函数中获得正在进行设备公开的设备信息。

----结束

3.2.2.3 注意事项

- WS63V100 支持 8 路星闪连接，可同时作为 BLE 和星闪设备工作。
- 若扫描不到设备，请先检查设备是否已在配对设备列表中，或者设备是否已与其他设备配对（此情况下需要先清除设备端配对信息）。

3.2.3 Connection Manager 接口

3.2.3.1 概述

Connection Manager 接口是星闪连接管理协议的软件实现，主要功能有连接、配对和读远端设备 RSSI 值。

3.2.3.2 开发流程

使用场景

当设备需要与对端设备建立连接时，可向对端设备发起连接请求。在连接过程中，设备可读取远端设备 RSSI 值，当设备需要更新连接参数时，可向对端设备发起连接参数更新请求，当设备需要与对端设备配对时，可向对端设备发起配对请求。在配对过程中，可获取当前本端设备与指定对端设备的配对状态。设备可获取当前配对设备数量以及当前配对设备信息链表。

功能

Connection Manager 提供的接口如下表所示。

接口名称	描述	参数说明	返回信息说明
sle_connect_remote_device	向对端设备发起连接请求。	addr: 对端设备地址。	接口返回值：错误码。
sle_disconnect_remote_device	向对端设备发起断连请求。	addr: 对端设备地址。	接口返回值：错误码。
sle_update_connect_param	连接参数更新。	params: 连接参数	接口返回值：错误码。
sle_pair_remote_device	向对端设备发起配对请求。（目前星闪鉴权流程仅支持免输入模式）	addr: 对端设备地址。	接口返回值：错误码。

接口名称	描述	参数说明	返回信息说明
sle_remove_paired_remote_device	与对端设备取消配对。	addr: 对端设备地址。	接口返回值: 错误码。
sle_remove_all_pairs	取消与所有对端设备的配对。	-	接口返回值: 错误码。
sle_get_paired_devices_num	获取配对设备数量。	number: [out]配对设备数量。	接口返回值: 错误码。
sle_get_paired_devices	获取配对设备信息。	addr: [out]设备地址链表; number: [inout]入参时为用户预留内存大小, 出参时为设备数量。	接口返回值: 错误码。
sle_get_pair_state	获取配对状态。	addr: 设备地址; state: [out]配对状态。	接口返回值: 错误码。
sle_read_remote_device_rssi	读对端设备 RSSI 值。	conn_id: 连接 id	接口返回值: 错误码。
sle_connection_register_callbacks	注册连接管理回调函数。	func: 用户回调函数。	接口返回值: 错误码。

开发流程

Connection Manager 开发的典型流程如下，具体编程实例可参考 application/samples/bt。

Terminal Node:

- 步骤 1 调用 enable\_sle，打开 SLE 开关。
- 步骤 2 调用 sle\_announce\_seek\_register\_callbacks，注册设备公开和设备发现回调函数。
- 步骤 3 调用 sle\_connection\_register\_callbacks，注册连接管理回调函数。

步骤 4 调用 sle\_set\_local\_addr, 设置本地设备地址。

步骤 5 调用 sle\_set\_local\_name, 设置本地设备名称。

步骤 6 调用 sle\_set\_announce\_param, 设置设备公开参数

步骤 7 调用 sle\_set\_announce\_data, 设置设备公开数据

步骤 8 调用 sle\_start\_announce, 启动设备公开。

----结束

#### **Grant Node:**

步骤 1 调用 enable\_sle, 打开 SLE 开关。

步骤 2 调用 sle\_announce\_seek\_register\_callbacks, 注册设备公开和设备发现回调函数。

步骤 3 调用 sle\_connection\_register\_callbacks, 注册连接管理回调函数。

步骤 4 调用 sle\_set\_local\_addr, 设置本地设备地址。

步骤 5 调用 sle\_set\_local\_name, 设置本地设备名称。

步骤 6 调用 sle\_set\_seek\_param, 设置设备发现参数。

步骤 7 调用 sle\_start\_seek, 启动设备发现, 并在回调函数中获得正在进行设备公开的设备信息。

步骤 8 调用 sle\_connect\_remote\_device, 向对端设备发起连接请求。

步骤 9 调用 sle\_pair\_remote\_device, 向对端设备发起配对请求。

步骤 10 调用 sle\_get\_paired\_devices\_num, 获取当前配对设备数量。

步骤 11 调用 sle\_get\_paired\_devices, 获取当前配对设备信息。

步骤 12 调用 sle\_get\_pair\_state, 获取配对状态。

----结束

## 3.2.4 SSAP server 接口

### 3.2.4.1 概述

SSAP 是 SLE 发送和接收数据的通用规范, 支持在两个 SLE 设备间进行数据传输。

3.2.4.2 开发流程

使用场景

SSAP Server 主要接收对端的请求和命令，向对端发送响应、通知和指示。

功能

SSAP Server 提供的接口如下表所示。

接口名称	描述	参数说明	返回信息说明
ssaps_register_server	注册 SSAP server。  <b>注：目前只支持注册一个 SSAP server。</b>	app_uuid：应用 UUID 指针；  server_id：[out] server id 指针。	接口返回值：错误码。
ssaps_unregister_server	注销 SSAP server。	server_id：server id。	接口返回值：错误码。
ssaps_add_service	添加服务。	server_id：server id  service_uuid：服务 UUID；  is_primary：是否是首要服务。	接口返回值：错误码。
ssaps_add_property	添加特征。	server_id：server id；  service_handle：服务句柄；  property：特征信息。	接口返回值：错误码。
ssaps_add_descriptor	添加特征描述符。	server_id：server id；  service_handle：服务句柄；  property_handle：特	接口返回值：错误码。

接口名称	描述	参数说明	返回信息说明
		征句柄;  descriptor: 描述符信息。	
ssaps_add_service_sync	添加服务同步接口, 服务句柄同步返回。	server_id: server id。  service_uuid: 服务UUID;  is_primary: 是否是首要服务;  handle: [out]服务句柄指针。	接口返回值: 错误码。
ssaps_add_property_sync	添加特征同步接口, 特征句柄同步返回。	server_id: server id。  service_handle: 服务句柄;  property: 特征;  handle: [out]特征句柄指针。	接口返回值: 错误码。
ssaps_add_descriptor_sync	添加特征描述符同步接口, 特征描述符句柄同步返回。	server_id: server id。  service_handle: 服务句柄;  property_handle: 特征句柄  descriptor: 特征描述符;  handle: [out]特征描述符句柄指针。	接口返回值: 错误码。
ssaps_start_service	启动服务。	server_id: server id;  service_handle: 服	接口返回值: 错误码。

接口名称	描述	参数说明	返回信息说明
		务句柄。	
ssaps_delete_all_services	删除所有服务。	server_id: server id。	接口返回值：错误码。
ssaps_send_response	发送响应。	server_id: server id; conn_id: 连接 ID; param: 响应参数。	接口返回值：错误码。
ssaps_notify_indicate	给对端发送通知或指示。明确下发数据的速率要大于连接间隔 30%。	server_id: server id; conn_id: 连接 ID; param: 通知或指示参数。	接口返回值：错误码。
ssaps_notify_indicate_by_uuid	按照 uuid 给对端发送通知或指示。明确下发数据的速率要大于连接间隔 30%。	server_id: server id; conn_id: 连接 ID; param: 通知或指示参数。	接口返回值：错误码。
ssaps_set_info	在连接之前设置 server 信息。	server_id: server id; info: server 信息。	接口返回值：错误码。
ssaps_register_callbacks	注册 SSAP server 回调函数。	func: 用户回调函数。	接口返回值：错误码。

开发流程

SSAP server 开发的典型流程：注册 SSAP server，注册本端属性数据库，接收对端的请求和命令，向对端发送通知和指示，具体编程实例可参考 application/samples/bt。

步骤 1 调用 enable\_sle，打开 SLE 开关。

- 步骤 2 调用 `ssaps_register_callbacks`，注册 SSAP server 回调。
- 步骤 3 调用 `sle_announce_seek_register_callbacks`，注册设备公开和设备发现回调函数。
- 步骤 4 调用 `ssaps_register_server`，创建一个 server 实体。
- 步骤 5 调用 `ssaps_add_service_sync`、`ssaps_add_property_sync`、`ssaps_add_descriptor_sync` 和 `ssaps_start_service` 注册本端属性数据库，每一个服务及其内容添加完成后调用 `ssaps_start_service` 启动服务。
- 步骤 6 调用 `sle_set_local_addr`，设置本地设备地址。
- 步骤 7 调用 `sle_set_local_name`，设置本地设备名称。
- 步骤 8 调用 `sle_set_announce_param`，设置设备公开参数。
- 步骤 9 调用 `sle_set_announce_data`，设置设备公开数据。
- 步骤 10 调用 `sle_start_announce`，启动设备公开。
- 步骤 11 连接建立。
- 步骤 12 接收对端设备的读写请求，当对端设备读写需要授权的特征或描述符时，调用 `ssaps_send_response` 向对端发送响应并修改本端特征值。
- 步骤 13 当某个特征的客户端特征配置描述符为 0x0001 时，在特征值变化时向对端设备发送通知，当某个特征的客户端特征配置描述符为 0x0002 时，在特征值变化时向对端设备发送指示。

----结束

## 3.2.5 SSAP client 接口

### 3.2.5.1 概述

SSAP 是 SLE 发送和接收数据的通用规范，支持在两个 SLE 设备间进行数据传输。

### 3.2.5.2 开发流程

#### 使用场景

SSAP Client 主要向对端发送请求和命令，接收对端的响应、通知和指示。



功能

SSAP Client 提供的接口如下表所示。

接口名称	描述	参数说明	返回信息说明
ssapc_register_client	注册 SSAP client。 <b>注：目前只支持注册一个 SSAP client。</b>	app_uuid：应用 UUID 指针； client_id：[out] client id 指针。	接口返回值：错误码。
ssapc_unregister_client	注销 SSAP client。	client_id：client id。	接口返回值：错误码。
ssapc_find_structure	查找对端服务、特征和描述符。	client_id：client id； conn_id：连接 ID； param：查找参数。	接口返回值：错误码。
ssapc_read_req_by_uuid	向对端发送按照 uuid 读取请求。	client_id：client id； conn_id：连接 ID； param：读取参数。	接口返回值：错误码。
ssapc_read_req	向对端发送读取请求。	client_id：client id； conn_id：连接 ID； handle：句柄； type：类型。	接口返回值：错误码。
ssapc_write_req	向对端发送写请求。明确下发数据的速率要大于连接间隔 30%。	client_id：client id； conn_id：连接 ID； param：写参数。	接口返回值：错误码。
ssapc_write_cmd	向对端发送写命令。明确下发数据的速率要大于连接间隔 30%。	client_id：client id； conn_id：连接 ID； param：写参数。	接口返回值：错误码。
ssapc_exchange_info_req	向对端发送交换信息请求。	client_id：client id； conn_id：连接 ID； param：交换信息参	接口返回值：错误码。

接口名称	描述	参数说明	返回信息说明
		数。	
ssapc_register_callbacks	注册 SSAP client 回调函数。	func：用户回调函数。	接口返回值：错误码。

开发流程

SSAP Client 开发的典型流程：注册 SSAP Client，查找对端属性数据库，向对端发送请求和命令，接收对端的通知和指示，具体编程实例可参考 application/samples/bt。

SSAP Server:

- 步骤 1 调用 enable\_sle，打开 SLE 开关。
- 步骤 2 调用 ssaps\_register\_callbacks，注册 SSAP server 回调。
- 步骤 3 调用 sle\_announce\_seek\_register\_callbacks，注册设备公开和设备发现回调函数。
- 步骤 4 调用 ssaps\_register\_server，创建一个 server 实体。
- 步骤 5 调用 ssaps\_add\_service\_sync、ssaps\_add\_property\_sync、ssaps\_add\_descriptor\_sync 和 ssaps\_start\_service 注册本端属性数据库，每一个服务及其内容添加完成后调用 ssaps\_start\_service 启动服务。
- 步骤 6 调用 sle\_set\_local\_addr，设置本地设备地址。
- 步骤 7 调用 sle\_set\_local\_name，设置本地设备名称。
- 步骤 8 调用 sle\_set\_announce\_param，设置设备公开参数。
- 步骤 9 调用 sle\_set\_announce\_data，设置设备公开数据。
- 步骤 10 调用 sle\_start\_announce，启动设备公开。
- 步骤 11 连接建立。
- 步骤 12 接收对端设备的读写请求，当对端设备读写需要授权的特征或描述符时，调用 ssaps\_send\_response 向对端发送响应并修改本端特征值。
- 步骤 13 当某个特征的客户端特征配置描述符为 0x0001 时，在特征值变化时向对端设备发送通知，当某个特征的客户端特征配置描述符为 0x0002 时，在特征值变化时向对端设备发送指示。

----结束

**SSAP Client:**

- 步骤 1 调用 enable\_sle，打开 SLE 开关。
- 步骤 2 调用 ssapc\_register\_callbacks，注册 SSAP client 回调。
- 步骤 3 调用 sle\_announce\_seek\_register\_callbacks，注册设备公开和设备发现回调函数。
- 步骤 4 调用 ssapc\_register\_client，创建一个 client 实体。
- 步骤 5 递归调用 ssapc\_find\_structure 查找对端属性数据库。
- 步骤 6 如果关注对端某个特征，可调用 ssapc\_write\_req 或 ssapc\_write\_cmd 将该特征的客户端特征配置描述符写为 0x0001 或 0x0002，前者可使得能对端特征通知，后者可使得能对端特征指示。
- 步骤 7 调用读写接口操作对端属性数据库。

----结束

3.2.6 错误码

SLE sdk 使用错误码指示用户当前任务执行结果，如下表所示。

序号	定义	实际数值	描述
1	ERRCODE_SLE_SUCCESS	0	执行成功错误码。
2	ERRCODE_SLE_CONTINUE	0x80006000	继续执行错误码。
3	ERRCODE_SLE_DIRECT_RETURN	0x80006001	直接返回错误码。
5	ERRCODE_SLE_PARAM_ERR	0x80006002	参数错误错误码。
6	ERRCODE_SLE_FAIL	0x80006003	执行失败错误码。
7	ERRCODE_SLE_TIMEOUT	0x80006004	执行超时错误码。
8	ERRCODE_SLE_UNSUPPORTED	0x80006005	参数不支持错误码。
9	ERRCODE_SLE_GETRECORD_FAIL	0x80006006	获取当前记录失败错误码。

序号	定义	实际数值	描述
10	ERRCODE_SLE_POINTER_NULL	0x80006007	指针为空错误码。
11	ERRCODE_SLE_NO_RECORD	0x80006008	无记录返回错误码。
12	ERRCODE_SLE_STATUS_ERROR	0x80006009	状态错误错误码。
13	ERRCODE_SLE_NOMEM	0x8000600a	内存不足错误码。
14	ERRCODE_SLE_AUTH_FAIL	0x8000600b	认证失败错误码。
15	ERRCODE_SLE_AUTH_PKEY_MISS	0x8000600c	PIN 码或密钥丢失致认证失败错误码。
16	ERRCODE_SLE_RMT_DEV_DOWN	0x8000600d	对端设备关闭错误码。
17	ERRCODE_SLE_PAIRING_REJECT	0x8000600e	配对拒绝错误码。
18	ERRCODE_SLE_BUSY	0x8000600f	系统繁忙错误码。
19	ERRCODE_SLE_NOT_READY	0x80006010	系统未准备好错误码。
20	ERRCODE_SLE_CONN_FAIL	0x80006011	连接失败错误码。
21	ERRCODE_SLE_OUT_OF_RANGE	0x80006012	越界错误码。
22	ERRCODE_SLE_MEMCPY_FAIL	0x80006013	拷贝失败错误码。
23	ERRCODE_SLE_MALLOC_FAIL	0x80006014	内存申请失败错误码。

### 3.3 BLE&SLE 功率档位定制

步骤 1 在 NV 配置文件 `middleware/chips/ws63/nv/nv_config/cfg/acore/app.json` 中, NV ID = "0x20A0" 的表项用于设置 BLE&SLE 的最大功率档位, value 的含义是 BLE&SLE 的最大功率档位。当前各个档位的发射功率为-6, -2, 2, 6, 10, 14, 16, 20。

如果设置为 7, 那么可以使用的档位是 0~7, 每档对应-6, -2, 2, 6, 10, 14, 16, 20;

如果设置为 5，那么可以使用的档位是 0~5，每档对应-6, -2, 2, 6, 10, 14。

```
"bt_txpower":{  
    "key_id": "0x20A0",  
    "key_status": "alive",  
    "structure_type": "btc_power_type_t",  
    "attributions": 1,  
    "value": [7]  
},
```

----结束

# 4 雷达 软件开发

- 4.1 概述
- 4.2 开发流程
- 4.3 注意事项
- 4.4 编程实例

## 4.1 概述

雷达特性为周期性地收发雷达信号以检测运动目标的功能特性，用户可以调用雷达 APIs 接口使用该特性。

## 4.2 开发流程

### 4.2.1 数据结构

雷达状态设置枚举定义：

```
typedef enum {  
    RADAR_STATUS_STOP = 0, /* 雷达状态配置停止 */  
    RADAR_STATUS_START,    /* 雷达状态配置启动 */  
    RADAR_STATUS_RESET,    /* 雷达状态配置复位 */  
    RADAR_STATUS_RESUME,   /* 雷达状态配置状态恢复 */  
} radar_set_sts_t;
```

雷达状态查询枚举定义：

```
typedef enum {  
    RADAR_STATUS_IDLE = 0, /* 雷达状态未工作 */  
    RADAR_STATUS_RUNNING, /* 雷达状态工作 */  
} radar_get_sts_t;
```

雷达结果上报结构体定义：

```
typedef struct {  
    uint32_t lower_boundary; /* 雷达结果靠近检测下边界 */  
    uint32_t upper_boundary; /* 雷达结果靠近检测上边界 */  
    uint8_t is_human_presence; /* 雷达结果有无人体存在 */  
    uint8_t reserved_0;  
    uint8_t reserved_1;  
    uint8_t reserved_2;  
} radar_result_t;
```

结果回调函数数据结构定义：

```
typedef void (*radar_result_cb_t)(radar_result_t *result);
```

4.2.2 APIs

雷达 APIs 接口如下表所示。

接口名称	描述	参数说明	返回信息说明
uapi_radar_set_status	设置雷达状态	sts:: 雷达状态	接口返回值： 错误码。
uapi_radar_get_status	获取雷达状态	*sts: 雷达状态	接口返回值： 错误码。
uapi_radar_register_result_cb	雷达结果回调注册函数	cb: 回调函数	接口返回值： 错误码。
uapi_radar_set_delay_time	设置退出延迟时间	time: 退出延迟时间	接口返回值： 错误码。
uapi_radar_get_delay_time	获取退出延迟时间	*time: 退出延迟时间	接口返回值： 错误码。
uapi_radar_get_isolatio	获取天线隔离度信	*iso: 天线隔离度	接口返回值：

接口名称	描述	参数说明	返回信息说明
n	息	信息	错误码。

4.2.3 错误码

序号	定义	实际数值	描述
1	ERRCODE_SUCC	0	执行成功错误码。
2	ERRCODE_FAIL	0xFFFF FFF	执行失败错误码。

4.3 注意事项

雷达特性需要在 WiFi 信道上进行工作，所以需注意打开雷达前，需要确保 WiFi 信道有配置，WiFi 进入 softAP 或 STA 模式即可。

4.4 编程实例

```
typedef void (*radar_result_cb_t)(radar_result_t *result);

#define WIFI_IFNAME_MAX_SIZE      16
#define WIFI_MAX_SSID_LEN        33
#define WIFI_SCAN_AP_LIMIT       64
#define WIFI_MAC_LEN              6
#define WIFI_INIT_WAIT_TIME      500 // 5s
#define WIFI_START_STA_DELAY     100 // 1s

#define RADAR_STATUS_SET_START    1
#define RADAR_STATUS_QUERY_DELAY 1000 // 10s

// WiFi启动STA模式实现样例
td_s32 radar_start_sta(td_void)
{
    (void)osDelay(WIFI_INIT_WAIT_TIME); /* 500: 延时0.5s, 等待wifi初始化完毕 */
    PRINT("STA try enable.\r\n");
```



```
/* 创建STA接口 */
if (wifi_sta_enable() != 0) {
    PRINT("sta enable fail !\r\n");
    return -1;
}

/* 连接成功 */
PRINT("STA connect success.\r\n");
return 0;
}

// 雷达结果回调函数实现样例
static void radar_print_res(radar_result_t *res)
{
    PRINT("[RADAR_SAMPLE] lb:%u, hb:%u, hm:%u\r\n", res->lower_boundary, res->upper_boundary, res->is_human_presence);
}

int radar_demo_init(void *param)
{
    PRINT("[RADAR_SAMPLE] radar_demo_init sta!\r\n");

    param = param;
    // WiFi启动STA模式
    radar_start_sta();

    // 注册雷达结果回调函数
    uapi_radar_register_result_cb(radar_print_res);

    // 启动雷达
    (void)osDelay(WIFI_START_STA_DELAY);
    uapi_radar_set_status(RADAR_STATUS_SET_START);

    // 雷达查询接口示例
    while(1) {
        (void)osDelay(RADAR_STATUS_QUERY_DELAY);
        uint8_t sts;
        uapi_radar_get_status(&sts);
        uint16_t time;
        uapi_radar_get_delay_time(&time);
    }
}
```

```
        uint16_t iso;  
        uapi_radar_get_isolation(&iso);  
    }  
  
    return 0;  
}
```

# 5 注意事项

## 5.1 看门狗

### 5.1 看门狗

#### 概述

WS63V100 默认提供看门狗功能，看门狗功能是一个指定时间（可编程）的定时器，用于系统异常恢复，如果未得到更新则当定时器到期时会产生一个系统复位信号，当看门狗在到期之前关闭或进行踢狗动作（即刷新定时器），则不会产生复位信号。

#### 功能描述

看门狗提供接口如下表所示：

接口名称	描述
uapi_watchdog_enable(mode)	使能看门狗； 参数 mode 为工作类型：（当前默认使用 mode0） 0-当看门狗触发时，将重启系统； 1-当看门狗触发时，将进入中断，如果在中断中没有喂狗，系统将重启；
uapi_watchdog_disable()	去使能看门狗；
uapi_watchdog_set_time(timeout)	设置看门狗定时时间，单位为秒；
uapi_watchdog_kick()	踢狗，重置看门狗定时器时间。

## 开发指引

看门狗功能的作用是为了暴露业务中出现卡死或无意义死循环的问题，正常业务流程中应当周期性进行踢狗操作，而当业务异常卡死或进入死循环时，踢狗操作得不到调度，便会在看门狗时间到期后触发复位信号。

修改看门狗定时时间步骤如下：

步骤 1 调用 `uapi_watchdog_disable`，去使能看门狗。（初始化时已默认使能）。

步骤 2 调用 `uapi_watchdog_set_time`，设置定时时间。

步骤 3 调用 `uapi_watchdog_enable`，重新使能看门狗。

----结束

踢狗步骤如下：

步骤 1 调用 `uapi_watchdog_kick`，刷新看门狗定时器时间为设置的定时时间。

----结束

## 注意事项

当前看门狗功能默认开启，定时时间为 7s，仅在 liteOS 的 IDLE 线程中保留踢狗动作，因此在进行 WiFi 打流等较占用 CPU 和系统资源的上层应用场景下，IDLE 线程可能无法得到调度，需要应用主动调用踢狗接口进行踢狗操作。

IPERF 打流业务功能

(`kernel/liteos/liteos_v207.0.0/Huawei_LiteOS/net/los_iperf/src/los_iperf.c`) 进行踢狗动作示例代码：

```
static void lperfFeedWdg(void)
{
    UINT32 ret = LOS_HistoryTaskCpuUsage(OsGetIdleTaskId(), CPUP_LAST_ONE_SECONDS);
    if (ret < IPERF_MIN_IDEL_RATE) {
        uapi_watchdog_kick();
    }
}

void lperfServerPhase2(lperfContext *context)
{
    int32_t recvLen;
    struct sockaddr peer;
```

```
    struct sockaddr *from = &peer;
    socklen_t slen = sizeof(struct sockaddr);
    socklen_t *fromLen = &slen;
    BOOL firstData = FALSE;
    lperfUdpHdr *udpHdr = (lperfUdpHdr *)context->param.buffer;
#ifdef LOSCFG_NET_IPERF_JITTER
    if (IPERF_IS_UDP(context->param.mask)) {
        context->udpStat->lastID = -1;
    }
#endif
    while ((context->isFinish == FALSE) && (context->isKilled == FALSE)) {
        recvLen = recvfrom(context->trafficSock, context->param.buffer,
                           context->param.bufLen, 0, from, fromLen);

        if (recvLen < 0) {
            if ((firstData == FALSE) && (errno == EAGAIN)) {
                continue;
            }
            IPERF_PRINT("recv failed %d\r\n", errno);
            break;
        } else if (recvLen == 0) { /* tcp connection closed by peer side */
            context->isFinish = TRUE;
            break;
        } else {
            if (firstData == FALSE) {
                firstData = TRUE;
                lperfServerFirstDataProcess(context, from, *fromLen);
                from = NULL;
                fromLen = NULL;
            }
            context->param.total += (uint32_t)recvLen;
            if (lperfServerDataProcess(context, (uint32_t)recvLen) && ((int32_t)ntohl(udpHdr->id)
< 0)) {
                context->isFinish = TRUE;
                break;
            }
            UNUSED(udpHdr);
#ifdef IPERF_FEED_WDG
            lperfFeedWdg();
#endif /* IPERF_FEED_WDG */
        }
    }
    gettimeofday(&context->end, NULL);
}
```