
WLAN 框架应用笔记

RT-THREAD 文档中心

上海睿赛德电子科技有限公司版权 ©2019



WWW.RT-THREAD.ORG

Friday 14th December, 2018

目录

目录	i
1 本文的目的和结构	1
1.1 本文的目的和背景	1
1.2 本文的结构	1
2 问题阐述	1
3 问题解决	1
3.1 WLAN 框架介绍	1
3.2 WLAN 框架组成	2
3.3 WLAN 框架功能	2
3.4 WLAN 框架配置及初始化	3
3.4.1. WLAN 框架配置	3
3.4.2. WLAN 设备初始化	5
3.5 WLAN 使用	5
3.5.1. Shell 操作 WiFi	5
3.5.1.1. WiFi 扫描	5
3.5.1.2. WiFi 连接	6
3.5.1.3. WiFi 断开	7
3.5.2. WiFi 扫描	7
3.5.3. WiFi 连接与断开	8
3.5.4. WiFi 开启自动重连	11
4 参考	13
4.1 本文相关 API	13
4.2 API 列表	13
4.3 核心 API 详解	14
4.3.1. rt_wlan_set_mode()	14

4.4	函数功能	14
4.5	函数原型	14
4.6	函数参数	14
4.7	返回值	14
4.7.1.	rt_wlan_prot_attach()	15
4.8	函数功能	15
4.9	函数原型	15
4.10	函数参数	15
4.11	返回值	15
4.11.1.	rt_wlan_scan_sync()	15
4.12	函数功能	15
4.13	函数原型	15
4.14	函数参数	15
4.15	返回值	16
4.15.1.	rt_wlan_connect()	16
4.16	函数功能	16
4.17	函数原型	16
4.18	函数参数	16
4.19	返回值	16
4.19.1.	rt_wlan_disconnect()	17
4.20	函数功能	17
4.21	函数原型	17
4.22	函数参数	17
4.23	返回值	17
4.23.1.	rt_wlan_config_autoreconnect()	17
4.24	函数功能	17
4.25	函数原型	17
4.26	函数参数	17
4.27	返回值	18

1 本文的目的和结构

1.1 本文的目的和背景

随着物联网快速发展，越来越多的嵌入式设备上搭载了 WIFI 无线网络设备。为了能够管理 WIFI 网络设备，RT-Thread 引入了 WLAN 设备管理框架。这套框架具备控制和管理 WIFI 的众多功能，为开发者使用 WIFI 设备提供许多便利。

本文将帮助开发者学会使用这套框架控制管理 WIFI。将从概念，示例等多方面介绍 WLAN 框架的相关知识，通过 Shell 及代码两种方式展示 WIFI 相关的功能，如控制 WIFI 扫描、连接、断开等。

1.2 本文的结构

- WLAN 框架介绍
- WLAN 框架配置
- WLAN 框架使用

2 问题阐述

本应用笔记主要围绕下面几个问题来介绍 WLAN 框架

- WLAN 框架是什么，具有什么功能？
- 如何配置 WLAN 框架？
- 如何使用 WLAN 框架？

想要解决上述问题，就要了解 WLAN 框架的组成原理，学会使用 WLAN 框架中的各种功能，下面将逐步开始介绍 WLAN 框架的组成及相关功能的使用。

3 问题解决

3.1 WLAN 框架介绍

WLAN 框架是 RT-Thread 开发的一套用于管理 WIFI 的中间件。对下连接具体的 WIFI 驱动，控制 WIFI 的连接断开，扫描等操作。对上承载不同的应用，为应用提供 WIFI 控制，事件，数据导流等操作，为上层应用提供统一的 WIFI 控制接口。WLAN 框架主要由三个部分组成。DEV 驱动接口层，为 WLAN 框架提供统一的调用接口。Manage 管理层为用户提供 WIFI 扫描，连接，断线重连等具体功能。Protocol 协议负责处理 WIFI 上产生的数据流，可根据不同的使用场景挂载不同通讯协议，如 LWIP 等。具有使用简单，功能齐全，对接方便，兼容性强等特点。

3.2 WLAN 框架组成

下图是 WIFI 框架的结构框架

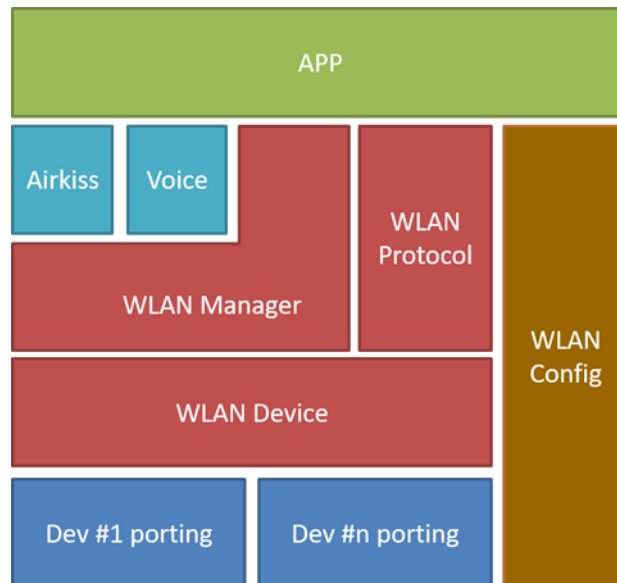


图 1: WIFI 框架

第一部分 app 为应用层。是基于 WLAN 框架的具体应用，如 WiFi 相关的 Shell 命令。

第二部分 airkiss、voice 为配网层。提供无线配网和声波配网等功能。

第三部分 WLAN manager 为 WLAN 管理层。能够对 WLAN 设备进行控制和管理。具备设置模式、连接热点、断开热点、启动热点、扫描热点等 WLAN 控制相关的功能。还提供断线重连，自动切换热点等管理功能。

第四部分 WLAN protocol 为协议层。将数据流递交给具体协议进行解析，用户可以指定使用不同的协议进行通信。

第五部分 WLAN config 为参数管理层。管理连接成功的热点信息及密码，并写入非易失的存储介质中。

第六部分 WLAN dev 为驱动接口层。对接具体 WLAN 硬件，为管理层提供统一的调用接口。

3.3 WLAN 框架功能

- 自动连接打开自动连接功能后，只要 WIFI 处在断线状态，就会自动读取之前连接成功的热点信息，连接热点。如果一个热点连接失败，则切换下一个热点信息进行连接，直到连接成功为止。自动连接使用的热点信息，按连接成功的时间顺序，依次尝试，优先使用最近连接成功的热点信息。连接成功后，将热点信息缓存在最前面，下次断线优先使用。
- 参数存储存储连接成功的 WIFI 参数，WIFI 参数会在内存中缓存一份，如果配置外部非易失存储接口，则会在外部存储介质中存储一份。用户可根据自己的实际情况，实现 `struct rt_wlan_cfg_ops` 这个结构体，将参数保存任何地方。缓存的参数主要给自动连接提供热点信息，wifi 处在未连接状态时，会读取缓存的参数，尝试连接。

- WIFI 控制提供完备的 WIFI 控制接口，扫描，连接，热点等。提供 WIFI 相关状态回调事件，断开，连接，连接失败等。为用户提供简单易用的 WIFI 管理接口。
- Shell 命令可在 Msh 中输入命令控制 WIFI 执行扫描，连接，断开等动作。打印 WIFI 状态等调试信息。

3.4 WLAN 框架配置及初始化

本文将基于正点原子 STM32L4 IOT board 开发板，该开发板板载一颗 AP6181 WiFi 芯片，且 WiFi 驱动已经实现。适合用来学习 WLAN 管理框架。

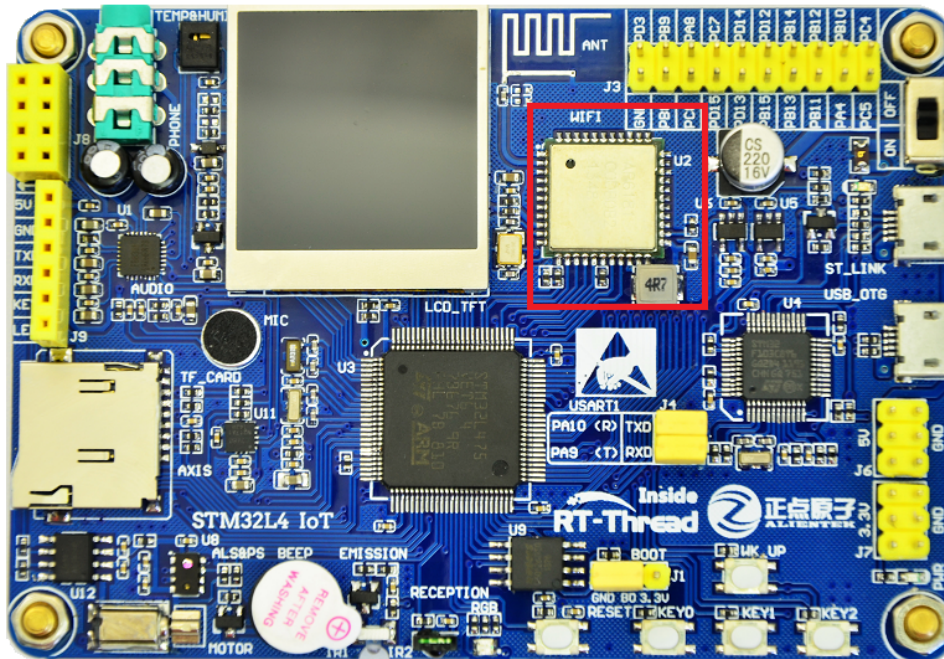


图 2: WIFI 框架

WLAN 框架配置主要包括以下几个方面

- 开启 WLAN 框架，并配置
- 初始化 WLAN 设备，并指定使用的协议

3.4.1. WLAN 框架配置

这里将介绍 WIFI 框架相关的配置，使用 env 工具进入 IOT board 目录，在 env 的命令行中输入 menuconfig 命令，打开图形配置界面

- 在 menuconfig 配置界面依次选择，RT-Thread Components -> Device Drivers -> Using WiFi -> 如下图所示

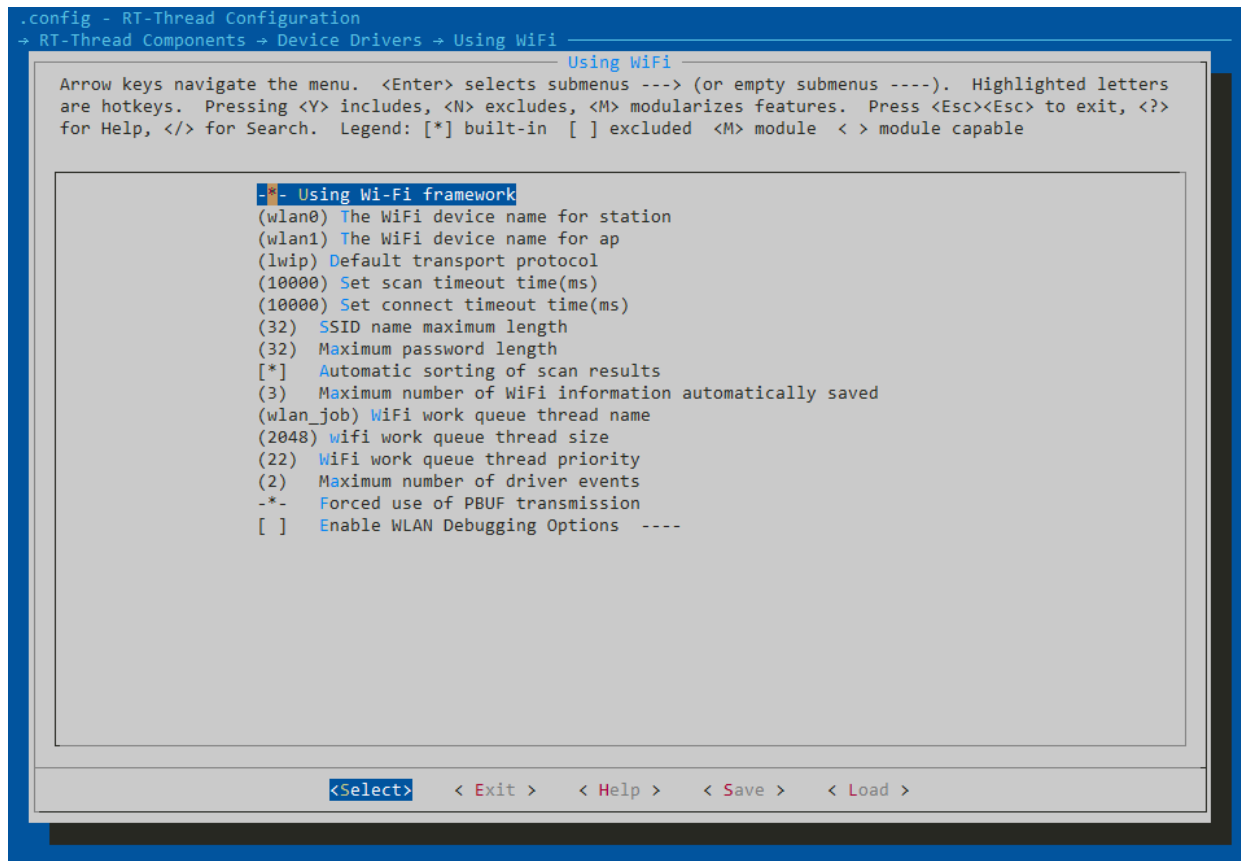


图 3: WLAN 配置

下面将介绍这些配置项

- Using Wi-Fi framework : 使用 WLAN 管理框架
- the WiFi device name for station : Station 设备默认名字
- the WiFi device name for ap : ap 设备默认名字
- Default transport protocol : 默认协议
- Scan timeout time : 扫描结果超时时间
- connect timeout time : 连接超时时间
- SSID name maximum length : SSID 最大长度
- Maximum password length : 密码最大长度
- Automatic sorting of scan results : 扫描结果自动排序
- Maximum number of WiFi information automatically saved : 自动保存最多条目数
- WiFi work queue thread name : WIFI 后台线程名字
- wifi work queue thread size : WIFI 后台线程栈大小
- WiFi work queue thread priority : WIFI 后台线程优先级

- Maximum number of driver events : dev 层一种事件最大注册数
- Forced use of PBUF transmission : 强行使用 PBUF 交换数据
- Enable WLAN Debugging Options : 打开调试 log 日志

按照上图配置好后，保存然后退出

3.4.2. WLAN 设备初始化

WLAN 框架需要指定一个初始化 WLAN 设备的工作模式，这些需要写代码实现，所以需要以下代码进行初始化。代码如下：

```
int wifi_init(void)
{
    rt_wlan_set_mode(RT_WLAN_DEVICE_STA_NAME, RT_WLAN_STATION);    // 配置WLAN设备
    工作模式
    return 0;
}
```

编写好上面的代码，一定要调用执行一次，即可使用使用 WLAN 框架管理设备了。

3.5 WLAN 使用

注：进行下面操作之前，一定要先执行 WLAN 设备初始化，详查看 [WLAN设备初始化](#) 部分

3.5.1. Shell 操作 WiFi

使用 shell 命令，可以帮助我们快速调试 WiFi 相关功能。只需要在 msh 中输入相应的命令，WiFi 就执行相应的动作。下面将通过三条命令，展现使用 shell 命令操作 WiFi。

wifi 相关的 shell 命令如下：

wifi	: 打印帮助
wifi help	: 查看帮助
wifi join SSID [PASSWORD]	: 连接wifi, SSID为空, 使用配置自动连接
wifi ap SSID [PASSWORD]	: 建立热点
wifi scan	: 扫描全部热点
wifi disc	: 断开连接
wifi ap_stop	: 停止热点
wifi status	: 打印wifi状态 sta + ap
wifi smartconfig	: 启动配网功能

3.5.1.1. WiFi 扫描 执行 wifi 扫描命令后，会将周围的热点信息打印在终端上。通过打印的热点信息，可以看到 SSID，MAC 地址等多项属性。

- wifi 扫描命令格式如下


```
wifi scan
```

命令说明

字段	描述
wifi	有关 wifi 命令都以 wifi 开头
scan	wifi 执行扫描动作

在 msh 中输入该命令，即可进行 wifi 命令扫描，扫描结果如下图所示

```
wifi scan
```

SSID	MAC	security	rssi	chn	Mbps
-----	-----	-----	-----	-----	-----
rtt_test_ssid_1	c0:3d:46:00:3e:aa	OPEN	-14	8	300
test_ssid	3c:f5:91:8e:4c:79	WPA2_AES_PSK	-18	6	72
rtt_test_ssid_2	ec:88:8f:88:aa:9a	WPA2_MIXED_PSK	-47	6	144
rtt_test_ssid_3	c0:3d:46:00:41:ca	WPA2_MIXED_PSK	-48	3	300

3.5.1.2. WiFi 连接 执行 WiFi 连接命令后，如果热点存在，且密码正确，开发板会连接上热点，并获得 IP 地址。网络连接成功后，可使用 socket 套接字进行网络通讯。

- wifi 连接命令格式如下

```
wifi join rtt-SSID0 12345678
```

- 命令解析

字段	描述
wifi	有关 wifi 命令都以 wifi 开头
join	wifi 执行连接动作
ssid	热点的名字
123456789	热点的密码，没有密码可不输入这一项

- 连接成功后，将在终端上打印获得的 IP 地址，如下图所示

```
wifi join ssid_test 12345678
[I/WLAN.mgmt] wifi connect success ssid:ssid_test
[I/WLAN.lwip] Got IP address : 192.168.1.110
```

3.5.1.3. WiFi 断开 执行 WiFi 断开命令后，开发板将断开与热点的连接。

- wifi 断开命令格式如下

```
wifi disc
```

- 命令解析

字段	描述
wifi	有关 wifi 命令都以 wifi 开头
disc	wifi 执行断开动作

- 断开成功后，将在终端上打印如下信息，如下图所示

```
wifi disc
[I/WLAN.mgmt] disconnect success!
```

3.5.2. WiFi 扫描

下面这段代码将展示 WiFi 同步扫描，然后将结果打印在终端上。先需要执行 WIFI 初始化，然后执行 WIFI 扫描函数 `rt_wlan_scan_sync`，这个函数是同步的，函数返回的扫描的数量和结果。在这个示例中，会将扫描的热点名字打印出来。

```
#include <rthw.h>
#include <rtthread.h>

#include <wlan_mgmt.h>
#include <wlan_prot.h>
#include <wlan_cfg.h>

void wifi_scan(void)
{
    struct rt_wlan_scan_result *result;
    int i = 0;

    /* Configuring WLAN device working mode */
    rt_wlan_set_mode(RT_WLAN_DEVICE_STA_NAME, RT_WLAN_STATION);
    /* WiFi scan */
    result = rt_wlan_scan_sync();
    /* Print scan results */
    rt_kprintf("scan num:%d\n", result->num);
    for (i = 0; i < result->num; i++)
    {
        rt_kprintf("ssid:%s\n", result->info[i].ssid.val);
    }
}
```

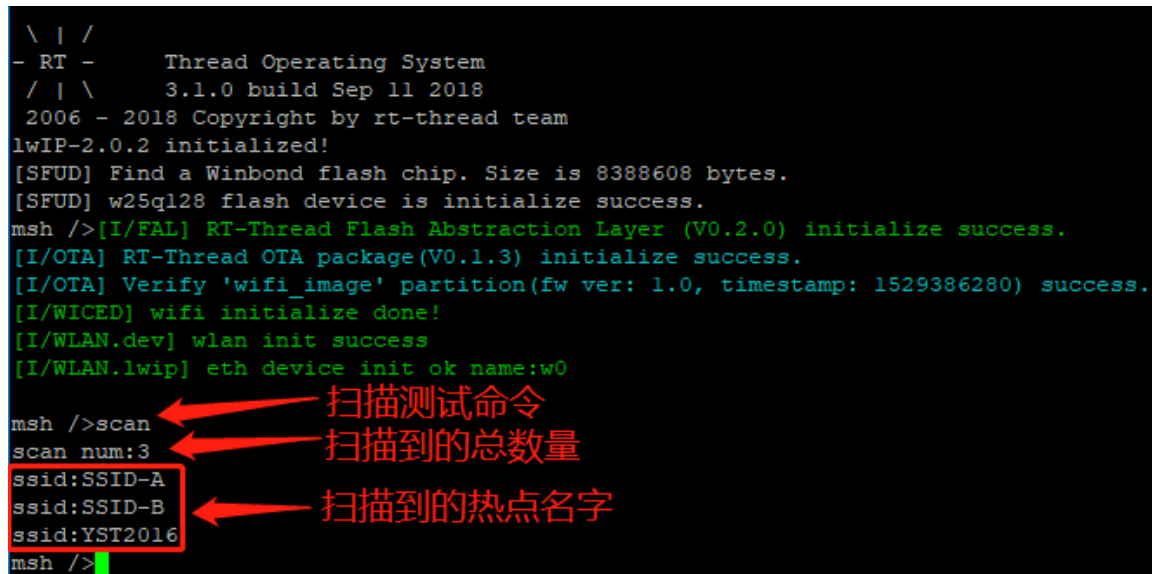
```

    }
}

int scan(int argc, char *argv[])
{
    wifi_scan();
    return 0;
}
MSH_CMD_EXPORT(scan, scan test.);

```

运行结果如下:



```

\ | /
- RT -      Thread Operating System
/ | \      3.1.0 build Sep 11 2018
2006 - 2018 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[SFUD] Find a Winbond flash chip. Size is 8388608 bytes.
[SFUD] w25ql28 flash device is initialize success.
msh />[I/FAL] RT-Thread Flash Abstraction Layer (V0.2.0) initialize success.
[I/OTA] RT-Thread OTA package (V0.1.3) initialize success.
[I/OTA] Verify 'wifi_image' partition (fw ver: 1.0, timestamp: 1529386280) success.
[I/WICED] wifi initialize done!
[I/WLAN.dev] wlan init success
[I/WLAN.lwip] eth device init ok name:w0
msh />scan
scan num:3
ssid:SSID-A
ssid:SSID-B
ssid:YST2016
msh />

```

扫描测试命令

扫描到的总数量

扫描到的热点名字

图 4: 扫描

3.5.3. WiFi 连接与断开

下面这段代码将展示 WiFi 同步连接。

首先需要执行 WIFI 初始化, 然后常见一个用于等待 `RT_WLAN_EVT_READY` 事件的信号量。注册需要关注的事件的回调函数, 执行 `rt_wlan_connect` wifi 连接函数, 函数返回表示是否已经连接成功。但是连接成功还不能进行通信, 还需要等待网络获取 IP。使用事先创建的信号量等待网络准备好, 网络准备好后, 就能正常通信了。

连接上 WIFI 后, 等待一段时间后, 执行 `rt_wlan_disconnect` 函数断开连接。断开操作是阻塞的, 返回值表示是否断开成功。

```

#include <rthw.h>
#include <rtthread.h>

#include <wlan_mngnt.h>
#include <wlan_prot.h>
#include <wlan_cfg.h>

#define WLAN_SSID "SSID-A"

```

```

#define WLAN_PASSWORD            "12345678"
#define NET_READY_TIME_OUT      (rt_tick_from_millisecond(15 * 1000))

static rt_sem_t net_ready = RT_NULL;

static void
wifi_ready_callback(int event, struct rt_wlan_buff *buff, void *parameter)
{
    rt_kprintf("%s\n", __FUNCTION__);
    rt_sem_release(net_ready);
}

static void
wifi_connect_callback(int event, struct rt_wlan_buff *buff, void *parameter)
{
    rt_kprintf("%s\n", __FUNCTION__);
    if ((buff != RT_NULL) && (buff->len == sizeof(struct rt_wlan_info)))
    {
        rt_kprintf("ssid : %s \n", ((struct rt_wlan_info *)buff->data)->ssid.val);
    }
}

static void
wifi_disconnect_callback(int event, struct rt_wlan_buff *buff, void *parameter)
{
    rt_kprintf("%s\n", __FUNCTION__);
    if ((buff != RT_NULL) && (buff->len == sizeof(struct rt_wlan_info)))
    {
        rt_kprintf("ssid : %s \n", ((struct rt_wlan_info *)buff->data)->ssid.val);
    }
}

static void
wifi_connect_fail_callback(int event, struct rt_wlan_buff *buff, void *parameter)
{
    rt_kprintf("%s\n", __FUNCTION__);
    if ((buff != RT_NULL) && (buff->len == sizeof(struct rt_wlan_info)))
    {
        rt_kprintf("ssid : %s \n", ((struct rt_wlan_info *)buff->data)->ssid.val);
    }
}

rt_err_t wifi_connect(void)
{
    rt_err_t result = RT_EOK;

    /* Configuring WLAN device working mode */
    rt_wlan_set_mode(RT_WLAN_DEVICE_STA_NAME, RT_WLAN_STATION);
    /* station connect */

```

```
rt_kprintf("start to connect ap ...\n");
net_ready = rt_sem_create("net_ready", 0, RT_IPC_FLAG_FIFO);
rt_wlan_register_event_handler(RT_WLAN_EVT_READY,
    wifi_ready_callback, RT_NULL);
rt_wlan_register_event_handler(RT_WLAN_EVT_STA_CONNECTED,
    wifi_connect_callback, RT_NULL);
rt_wlan_register_event_handler(RT_WLAN_EVT_STA_DISCONNECTED,
    wifi_disconnect_callback, RT_NULL);
rt_wlan_register_event_handler(RT_WLAN_EVT_STA_CONNECTED_FAIL,
    wifi_connect_fail_callback, RT_NULL);

/* connect wifi */
result = rt_wlan_connect(WLAN_SSID, WLAN_PASSWORD);

if (result == RT_EOK)
{
    /* waiting for IP to be got successfully */
    result = rt_sem_take(net_ready, NET_READY_TIME_OUT);
    if (result == RT_EOK)
    {
        rt_kprintf("networking ready!\n");
    }
    else
    {
        rt_kprintf("wait ip got timeout!\n");
    }
    rt_wlan_unregister_event_handler(RT_WLAN_EVT_READY);
    rt_sem_delete(net_ready);

    rt_thread_delay(rt_tick_from_millisecond(5 * 1000));
    rt_kprintf("wifi disconnect test!\n");
    /* disconnect */
    result = rt_wlan_disconnect();
    if (result != RT_EOK)
    {
        rt_kprintf("disconnect failed\n");
        return result;
    }
    rt_kprintf("disconnect success\n");
}
else
{
    rt_kprintf("connect failed!\n");
}
return result;
}

int connect(int argc, char *argv[])
{

```

```

    wifi_connect();
    return 0;
}
MSH_CMD_EXPORT(connect, connect test.);

```

运行结果如下

```

\ | /
- RT -      Thread Operating System
/ | \      3.1.0 build Sep 11 2018
2006 - 2018 Copyright by rt-thread team
lwIP-2.0.2 initialized!
[SFUD] Find a Winbond flash chip. Size is 8388608 bytes.
[SFUD] w25ql28 flash device is initialize success.
msh />[I/FAL] RT-Thread Flash Abstraction Layer (V0.2.0) initialize success.
[I/OTA] RT-Thread OTA package(V0.1.3) initialize success.
[I/OTA] Verify 'wifi_image' partition(fw ver: 1.0, timestamp: 1529386280) success.
[I/WICED] wifi initialize done!
[I/WLAN.dev] wlan init success
[I/WLAN.lwip] eth device init ok name:w0
msh />connect
start to connect ap ...
join ssid:SSID-A
[I/WLAN.mgmt] wifi connect success ssid:SSID-A
wifi_connect_callback
wifi_ready_callback
networking ready!
[I/WLAN.lwip] Got IP address : 192.168.43.10
wifi disconnect test!
wifi_disconnect_callback
disconnect success
msh />

```

连接测试命令

连接连接

获得IP地址

断开连接

断开成功

图 5: 连接断开

3.5.4. WiFi 开启自动重连

先开启自动重连功能，使用命令行连接上一个热点 A 后，在连接上另一个热点 B。等待几秒后，将热点 B 断电，系统会自动重试连接 B 热点，此时 B 热点连接不上，系统自动切换热点 A 进行连接。连接成功 A 后，系统停止连接。

```

#include <rthw.h>
#include <rtthread.h>

#include <wlan_mgmt.h>
#include <wlan_prot.h>
#include <wlan_cfg.h>

static void
wifi_ready_callback(int event, struct rt_wlan_buff *buff, void *parameter)
{
    rt_kprintf("%s\n", __FUNCTION__);
}

```

```

static void
wifi_connect_callback(int event, struct rt_wlan_buff *buff, void *parameter)
{
    rt_kprintf("%s\n", __FUNCTION__);
    if ((buff != RT_NULL) && (buff->len == sizeof(struct rt_wlan_info)))
    {
        rt_kprintf("ssid : %s \n", ((struct rt_wlan_info *)buff->data)->ssid.val);
    }
}

static void
wifi_disconnect_callback(int event, struct rt_wlan_buff *buff, void *parameter)
{
    rt_kprintf("%s\n", __FUNCTION__);
    if ((buff != RT_NULL) && (buff->len == sizeof(struct rt_wlan_info)))
    {
        rt_kprintf("ssid : %s \n", ((struct rt_wlan_info *)buff->data)->ssid.val);
    }
}

static void
wifi_connect_fail_callback(int event, struct rt_wlan_buff *buff, void *parameter)
{
    rt_kprintf("%s\n", __FUNCTION__);
    if ((buff != RT_NULL) && (buff->len == sizeof(struct rt_wlan_info)))
    {
        rt_kprintf("ssid : %s \n", ((struct rt_wlan_info *)buff->data)->ssid.val);
    }
}

int wifi_autoconnect(void)
{
    /* Configuring WLAN device working mode */
    rt_wlan_set_mode(RT_WLAN_DEVICE_STA_NAME, RT_WLAN_STATION);
    /* Start automatic connection */
    rt_wlan_config_autoreconnect(RT_TRUE);
    /* register event */
    rt_wlan_register_event_handler(RT_WLAN_EVT_READY,
        wifi_ready_callback, RT_NULL);
    rt_wlan_register_event_handler(RT_WLAN_EVT_STA_CONNECTED,
        wifi_connect_callback, RT_NULL);
    rt_wlan_register_event_handler(RT_WLAN_EVT_STA_DISCONNECTED,
        wifi_disconnect_callback, RT_NULL);
    rt_wlan_register_event_handler(RT_WLAN_EVT_STA_CONNECTED_FAIL,
        wifi_connect_fail_callback, RT_NULL);
    return 0;
}

int auto_connect(int argc, char *argv[])

```

```
{
    wifi_autoconnect();
    return 0;
}
MSH_CMD_EXPORT(auto_connect, auto connect test.);
```

运行结果如下:

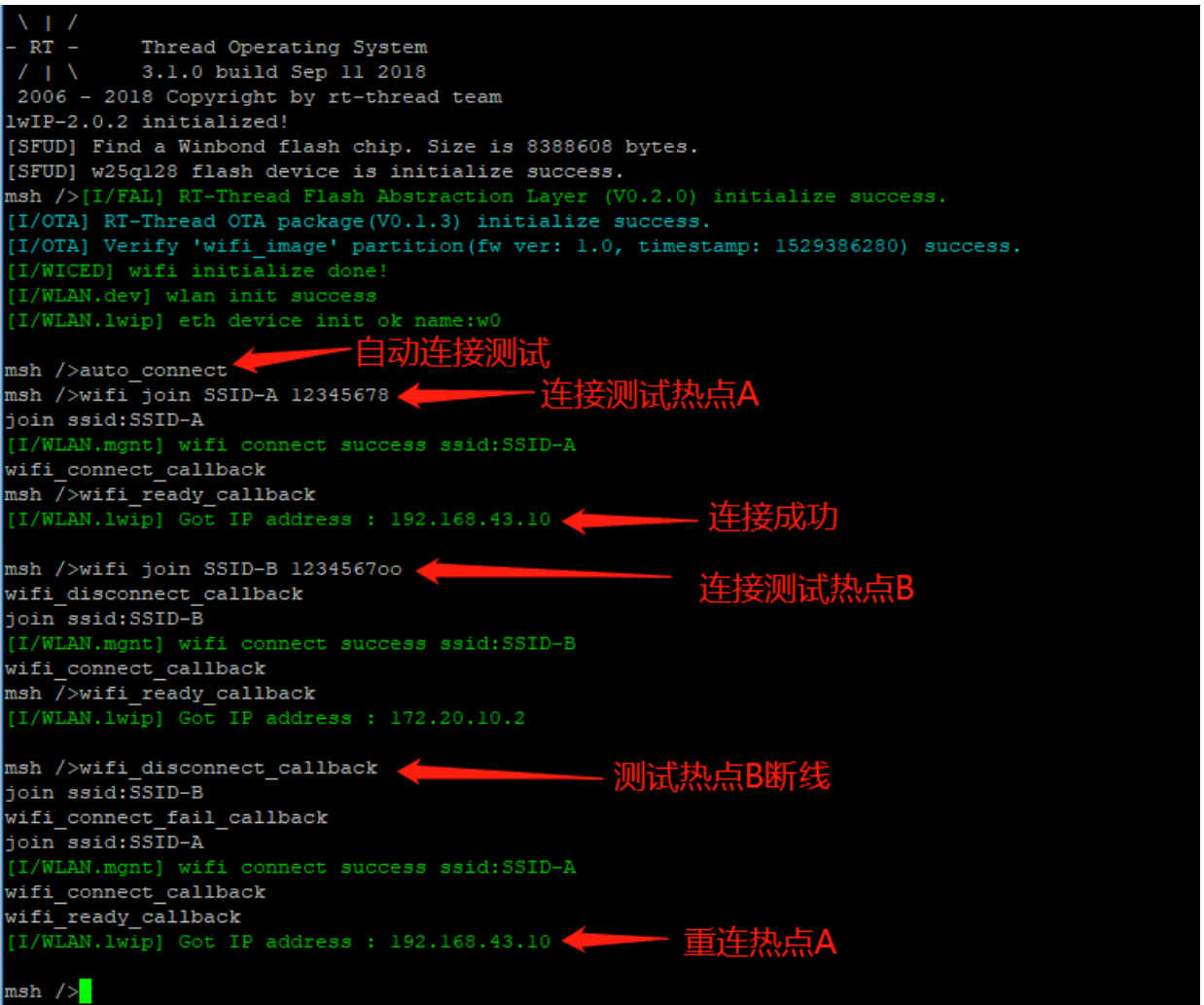


图 6: 自动连接

4 参考

4.1 本文相关 API

4.2 API 列表

API	位置
rt_wlan_set_mode	wlan_mgnt.c

API	位置
rt_wlan_prot_attach	wlan_prot.c
rt_wlan_scan_sync	wlan_mgnt.c
rt_wlan_connect	wlan_mgnt.c
rt_wlan_disconnect	wlan_mgnt.c
rt_wlan_config_autoreconnect	wlan_mgnt.c

4.3 核心 API 详解

4.3.1. rt_wlan_set_mode()

4.4 函数功能

注册 WLAN 设备到 WLAN 设备框架

4.5 函数原型

```
rt_err_t rt_wlan_set_mode(const char *dev_name, rt_wlan_mode_t mode);
```

4.6 函数参数

参数	描述
dev_name	WLAN 设备名
mode	WLAN 设备工作模式

4.7 返回值

返回值	描述
-RT_EINVAL	参数错误
-RT_EIO	设备未找到
-RT_ERROR	执行失败
RT_EOK	执行成功

wlan mode 可取以下值之一 RT_WLAN_NONE 清空工作模式 RT_WLAN_STATION 工作在 STATION 模式 RT_WLAN_AP 工作在 AP 模式

4.7.1. rt_wlan_prot_attach()

4.8 函数功能

指定 WLAN 设备使用的协议

4.9 函数原型

```
rt_err_t rt_wlan_prot_attach(const char *dev_name, const char *prot_name);
```

4.10 函数参数

参数	描述
name	WLAN 设备名
prot_name	协议名

4.11 返回值

返回值	描述
-RT_ERROR	执行失败
RT_EOK	执行成功

type 可取以下值之一 RT_WLAN_PROT_LWIP 协议类型为 LWIP

4.11.1. rt_wlan_scan_sync()

4.12 函数功能

同步扫描热点

4.13 函数原型

```
struct rt_wlan_scan_result *rt_wlan_scan_sync(void);
```

4.14 函数参数

无

4.15 返回值

返回值	描述
rt_wlan_scan_result	扫描结果

扫描结果是一个结构体，成员如下

num : info 数量 info : info 指针

```
struct rt_wlan_scan_result
{
    rt_int32_t num;
    struct rt_wlan_info *info;
};
```

4.15.1. rt_wlan_connect()

4.16 函数功能

同步连接热点

4.17 函数原型

```
rt_err_t rt_wlan_connect(const char *ssid, const char *password);
```

4.18 函数参数

参数	描述
ssid	WIFI 名字
password	WIFI 密码

4.19 返回值

返回值	描述
-RT_EINVAL	参数错误
-RT_EIO	未注册设备
-RT_ERROR	连接失败
RT_EOK	连接成功

返回值	描述
-----	----

4.19.1. rt_wlan_disconnect()

4.20 函数功能

同步断开热点

4.21 函数原型

```
rt_err_t rt_wlan_disconnect(void);
```

4.22 函数参数

无

4.23 返回值

返回值	描述
-RT_EIO	未注册设备
-RT_ENOMEM	内存不足
-RT_ERROR	断开失败
RT_EOK	断开成功

4.23.1. rt_wlan_config_autoreconnect()

4.24 函数功能

配置自动重连模式

4.25 函数原型

```
void rt_wlan_config_autoreconnect(rt_bool_t enable);
```

4.26 函数参数

参数	描述
enable	enanle/disable 自动重连

4.27 返回值

无