WMSDK 用户手册

V1.0.9

北京联盛德微电子有限责任公司（winner micro）

地址：北京市海淀区上园村 3 号交大知行大厦七层

电话：+86-10-62161900

公司网址：www.winnermicro.com

文档历史

版本	完成日期	修订记录	作者	审核	批准
V1.0.0	2014-4-4	创建			
V1.0.1	2014-4-23	1、 增加内存管理说明 2、 增加一键配置说明 3、 修改程序入口说明			
V1.0.2	2014-5-7	增加标准 socket 接口			
V1.0.3	2014-5-14	1、 增加软 AP 创建接口 2、 增加设置 SPI 传输类型接口			
V1.0.4	2014-5-27	1、增加 SPI 从接口			
V1.0.5	2014-6-5	1、增加 httpfwup 的 API 接口			
V1.0.6	2014-6-13	1.增加 HTTP Client 的 API 接口			
V1.0.7	2014-6-30	1、 增加扫描网络接口 2、 增加切换信道接口 3、设置/获取 Wi-Fi 自动重连标志接口 4、 获取当前 Wi-Fi 网络信息接口			
V1.0.8	2015-01-24	1、整理 EM-WIFI 工具及一键配置的说明			
V1.0.9	2015-08-20	1、增加获取无线报文更多信息的接口 2、增加 AP 和 APSTA 获取已经关联上的所有 STA 的接口 3、增加获取 Wi-Fi 层状态的接口 4、增加 Wi-Fi 错误码相关的接口 5、增加 APSTA 加网相关的接口			

目录

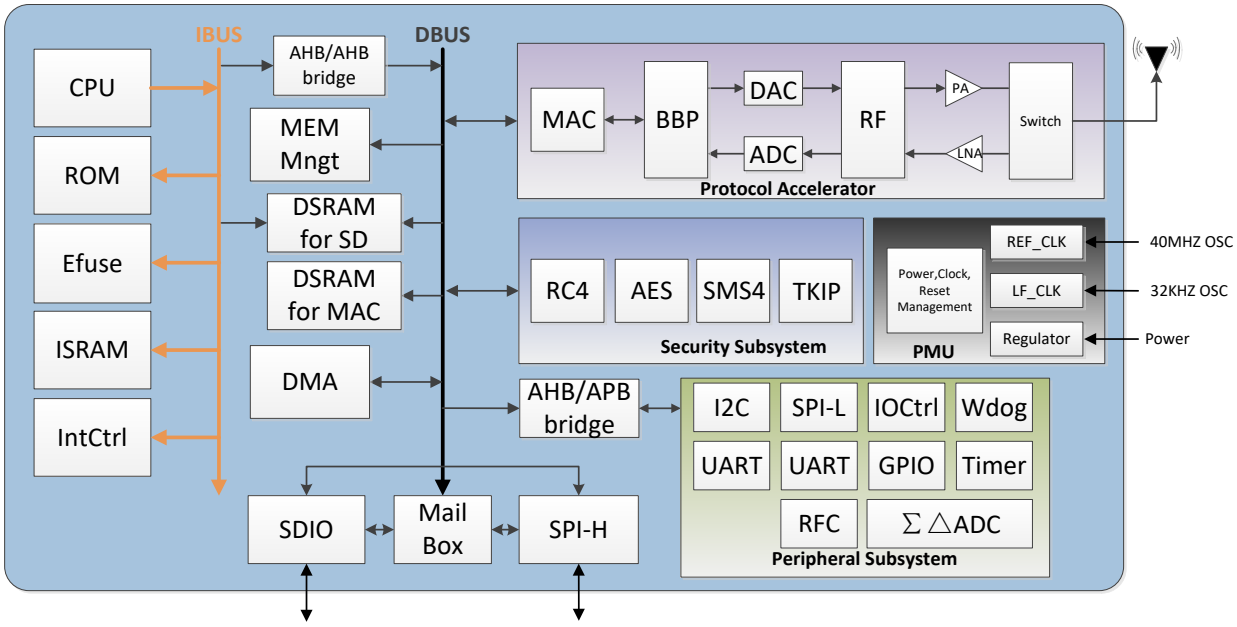
1 引言

1.1 概述

本文主要描述 W500（HED10W07SN）软件开发包（SDK）的功能和使用方法，该 SDK 集成了 W500（HED10W07SN）硬件驱动（BSP）、实时操作系统、TCP/IP 协议栈、WiFi 协议栈以及其它公共模块，能够满足大部分应用软件的需求。

1.2 芯片架构简介

图 1.1 芯片架构图



其中 MCU 最高主频 160M，目前默认主频 80M。

1.3 SDK 基本特征

接口

- ✓ 8 个 GPIO@SPI；6 个 GPIO@UART
- ✓ I2C
- ✓ 4 路 ADC
- ✓ H-SPI

- ✓ L-SPI
- ✓ UART

无线

- ✓ 支持 IEEE802.11b/g/i 无线标准
- ✓ 支持频率范围：2.412~2.484 GHz
- ✓ 支持基础网（Infra）
  - 支持多种加密和认证机制：

    OPEN/WEP64/WEP128/ TKIP/CCMP/WPA-PSK/WPA2-PSK
  - 支持快速联网模式（指定信道与 BSSID）
  - 支持无线漫游
  - 支持 Sleep 节能模式（PS-POLL 方式）
  - 支持 WPS 功能
- ✓ 支持自组网（Adhoc）：
  - 支持 OPEN、WEP 加密认证
  - 支持网络不存在时自动创建
- ✓ 支持软 AP
  - 支持 OPEN、WEP 加密认证
  - 最多支持 4 个 station 连接
  - 支持 STA 节能（PS-POLL 方式）

其它

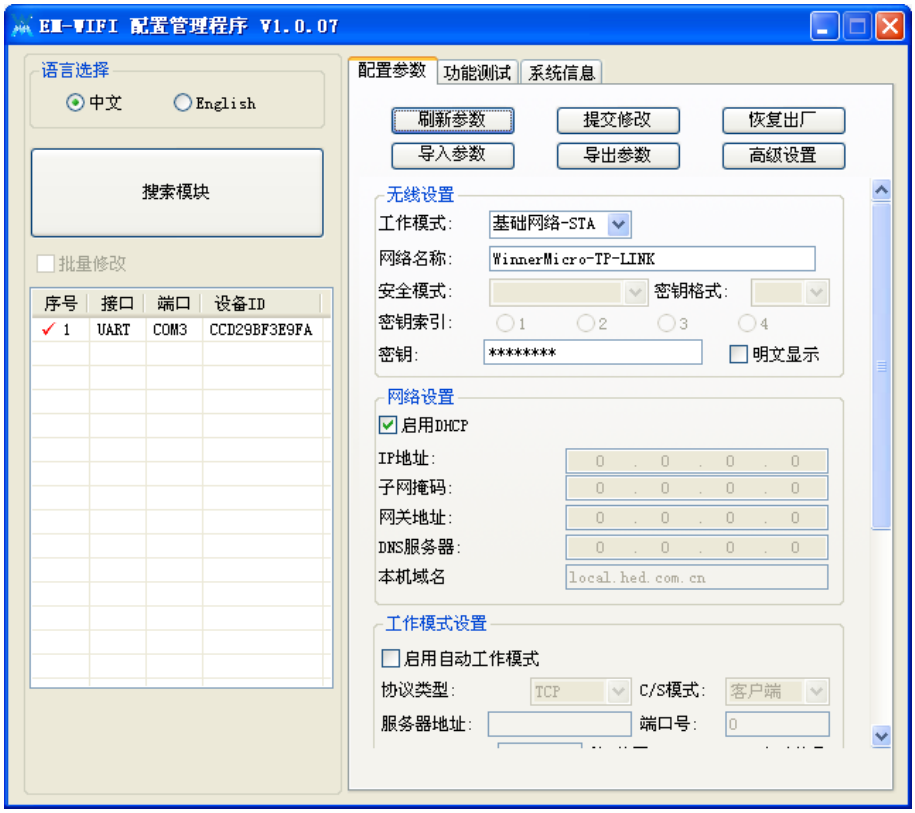
- ✓ 支持 SPI、UART 接口通信
  - 支持高速 SPI 数据接口，接口速率 20Mbps
  - 支持 UART 数据接口，接口速率 2Mbps
- ✓ 支持用户可编程的 GPIO 控制
- ✓ 支持自动工作模式
  - 基于参数配置方式
- ✓ 支持多种网络协议：TCP/UDP/ICMP/DHCP/DNS
- ✓ 支持 DHCP Server、DNS Server
- ✓ 支持 HTTP Client， HTTP Server 功能

✓ 支持基于串口方式的模块生产测试

2 开发工具

2.1 EM-Wifi 工具

2.1.1.1 参数配置界面



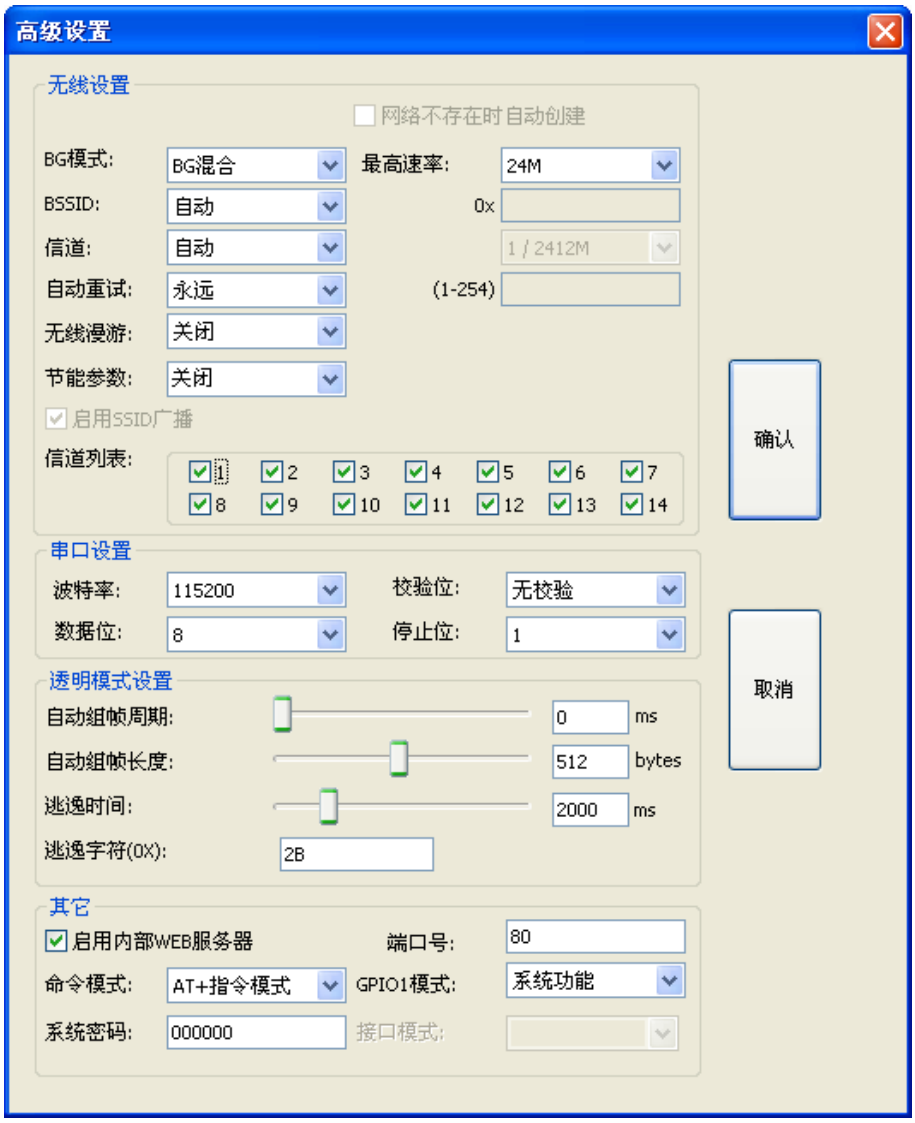
1、EM-WIFI 工具主界面如上图所示，点击搜索模块按钮会自动列出当前连接到 PC 的模块，由于开发板有 UART0 和 UART1 两个串口，上图界面可能会搜索到两个串口模块，请选择 UART1 对应的串口模块进行通讯。

2、EM-WIFI 工具设置完参数后，必须点击提交修改按钮，将当前设置的参数更新到模块参数区后，模块复位后才会生效。

3、对于客户配置的低波特率的模块，工具搜索耗时会比较长，建议修改 EM-WIFI 工具根目录下的 user.ini 文件默认搜索的串口波特率。

例如：BaudRate=9600

4、对于加网失败的情况，请检查高级设置界面中 BSSID 连接设置是否为自动，如果是指定那么只能连接上固定 BSSID 的路由器。



2.1.1.2　功能测试界面

如下图所示，功能测试界面提供加网和 SOCKET 通讯的基本功能测试；

加网：连接当前配置 SSID 的网络；

断网：断开当前连接的网络；

扫描：扫面 1-14 信道的网络；

状态：显示当前网络的连接状态；

复位：复位模块；

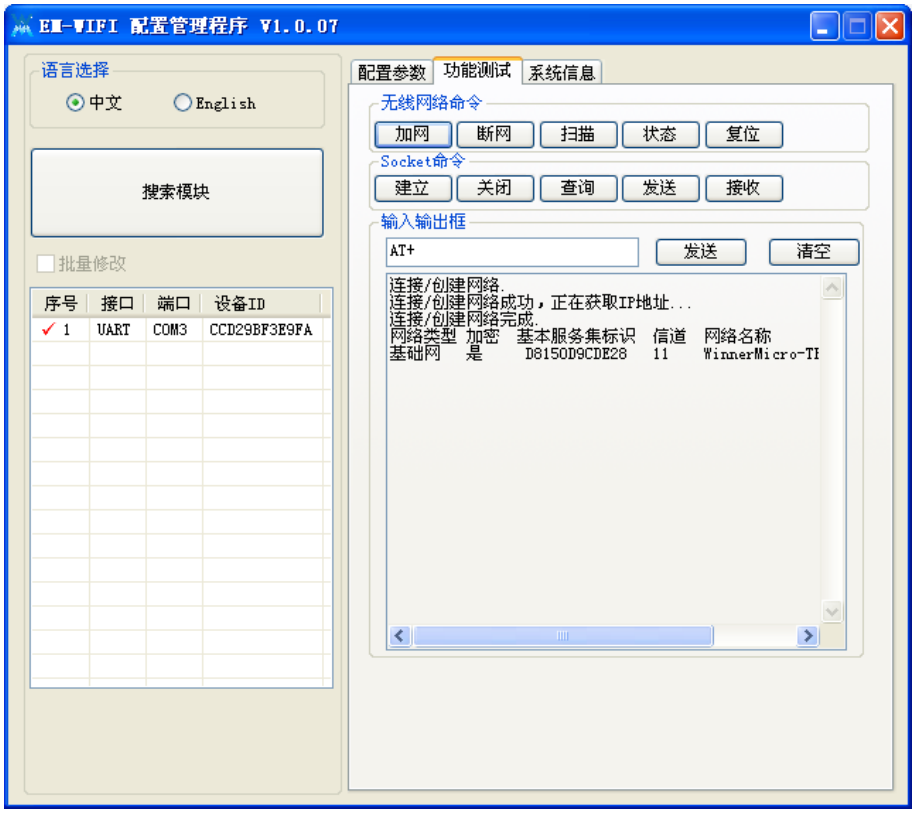
建立：根据配置创建 socket；

关闭：根据 socket 号关闭 socket；

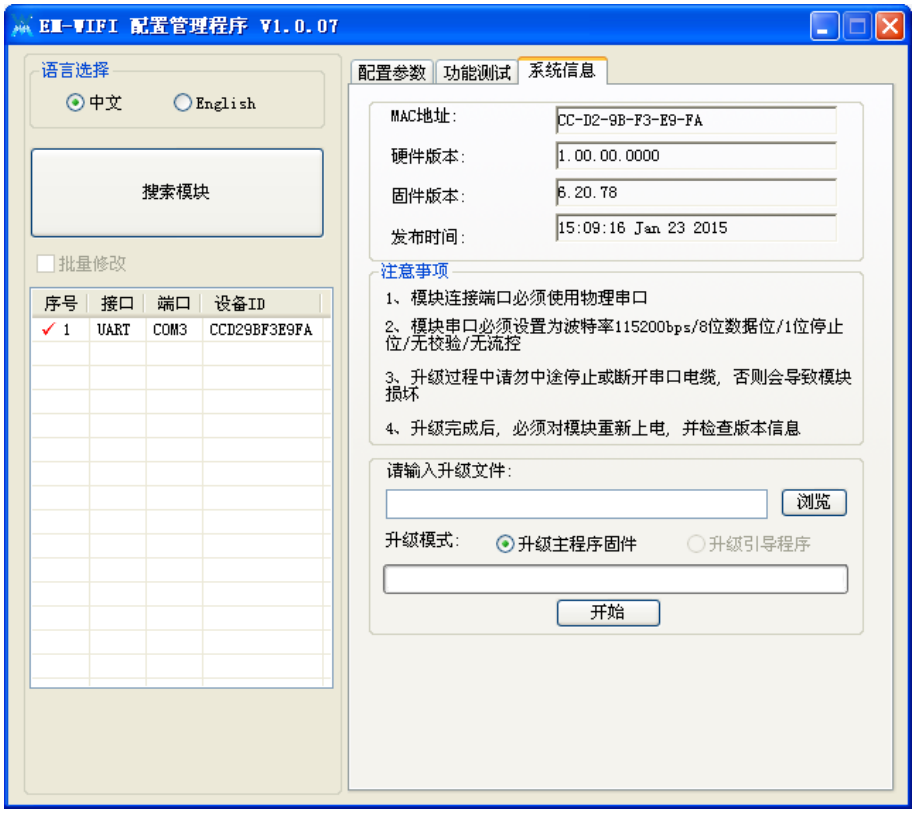
查询：根据 socket 号查询 socket 状态；

发送：向指定 socket 号的连接发送数据；

接收：接收指定 socket 号的数据；



2.1.1.3 固件升级界面



选择正常的固件 img 文件，点击开始则开始更新固件，建议升级时候串口波特率选择 115200，可以减少升级的时间。

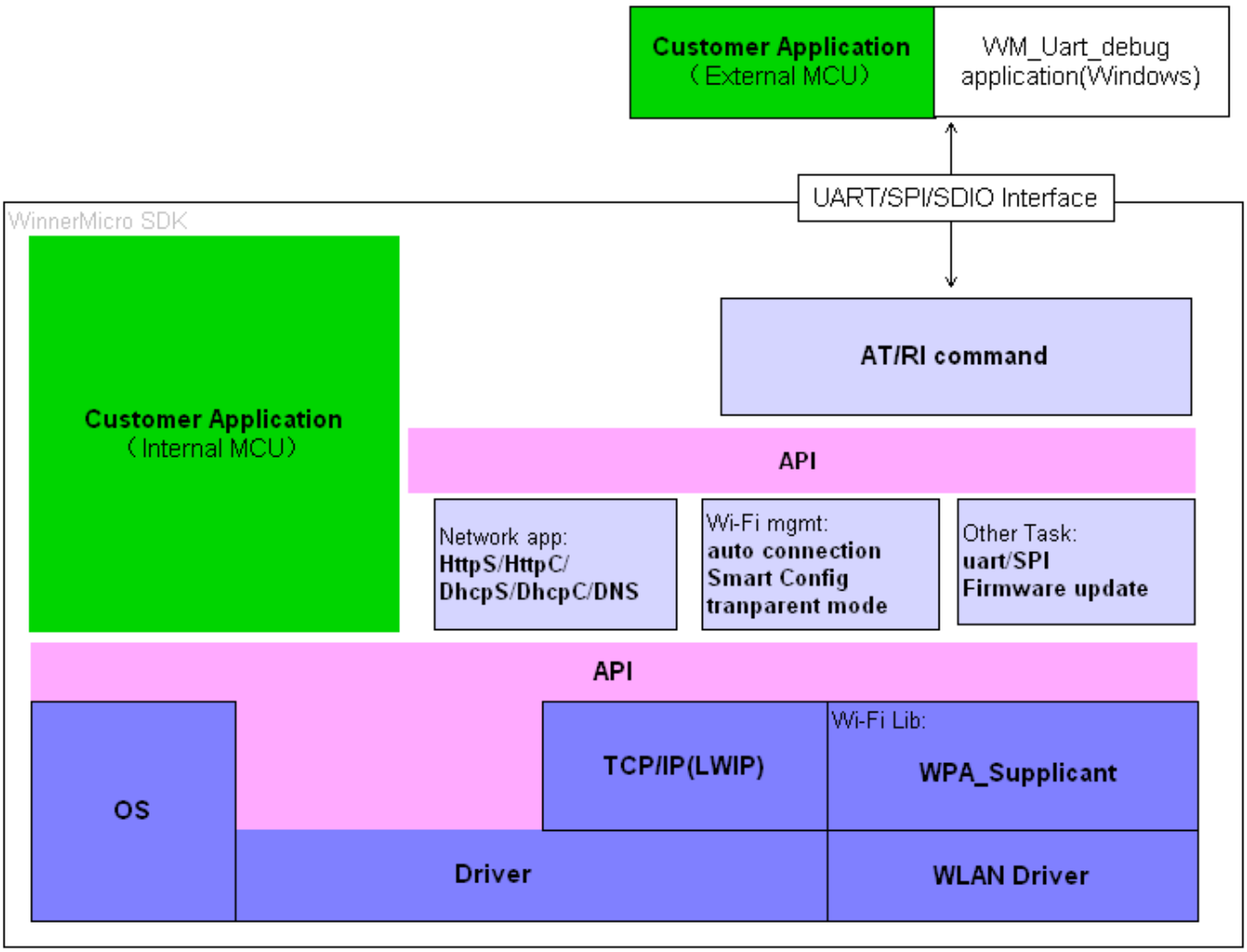
2.2 一键配置

打开 OneshotConfig 工具，会自动显示手机当前连接的无线网络的 SSID，输入当前连接网络的密码，点击配置，当前网络的 SSID 和密码会以广播的方式发送到已经进入一键配置模式的无线模块，模块收到当前手机连接路由器的账号和密码后会自动加入当前路由器的网络。

3 SDK 使用说明

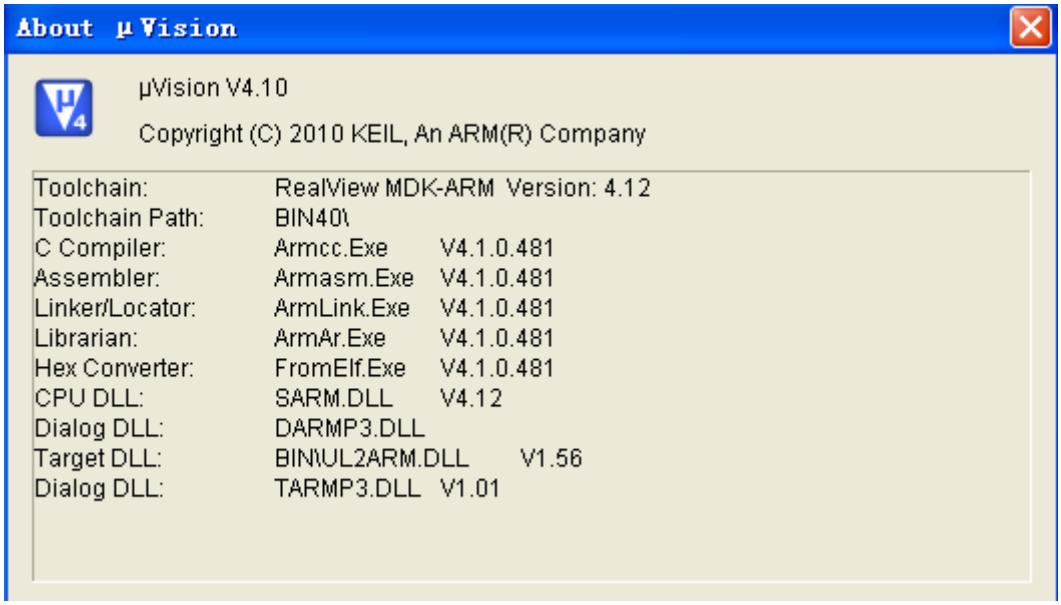
3.1 软件架构

图 3.1 WM SDK 软件架构图



3.2 开发环境

SDK 集成开发环境：Arm Keil uVersion V4.10



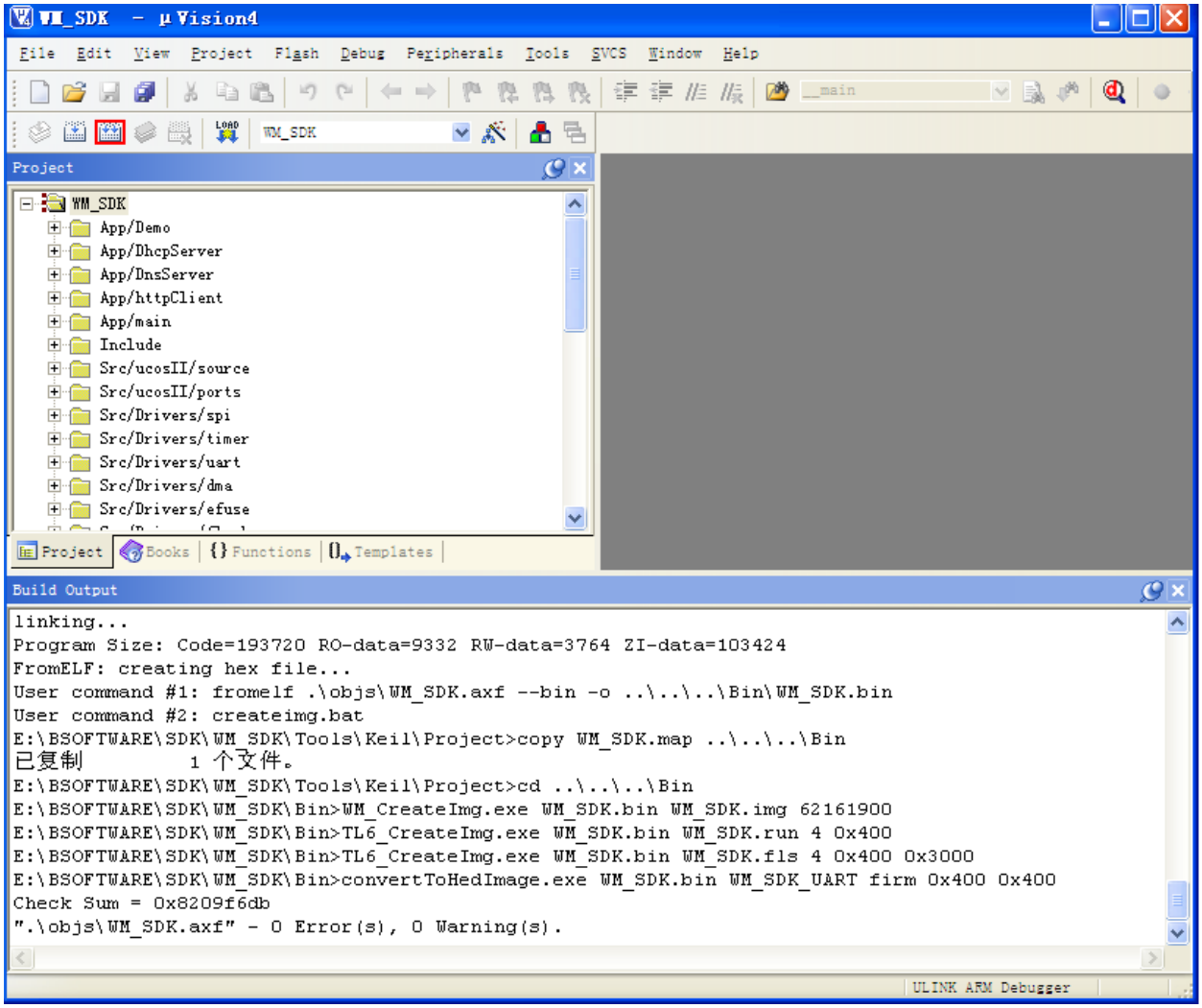
3.3 目录结构

WM_SDK

├──App	应用程序
├──Bin	可执行文件及打包工具
├──Doc	API 文档
├──Include	API 头文件
├──Lib	Lib 库
├──Src	源代码
└──Tools	编译工具及 Keil 工程

3.4 编译连接

在 Tools/Keil/Project 目录下点击 WM_SDK.uvproj，打开工程，点击下图红色按钮进行编译链接，生成文件放置在 Bin 目录下。



输出文件说明：

WM_SDK.img	EM-WIFI 工具升级镜像
WM_SDK.fls	SPI Flash 烧录镜像
WM_SDK_UART.img	Uart Xmodem 下载镜像

3.5 程序调试

SDK 支持标准 C 的 printf 打印调试的功能，输出到物理串口 UART0。

开发阶段 UART0 支持串口 Xmodem 下载固件到内存的功能。基于 W500（HED10W07SN）开发模组在上电时按住 ESC 键可以进入串口 Xmodem 下载模式。

4 开发指南

4.1 FLASH 空间

采用 1M Byte SPI Flash 空间，具体分配如下：

地址空间	大小（Kbyte）	功能描述
0x00000 - 0x02FFF	12	系统参数保留区
0x03000 - 0x4DFFF	300	固件 1 区
0x4E000 - 0x98FFF	300	固件 2 区
0x99000 - 0xFFFFF	约 410	用户可用空间

注：对于二次开发用户，可以使用“固件 1 区/2 区” 存储固件。

4.2 启动方式

WM_SDK 开发支持两种 CODE 加载执行方式：

- 从 SPI Flash 固件区搬移 CODE 到 SRAM 中执行
- 使用 UART0 通过 X modem 方式下载镜像到 SRAM 中执行

4.3 程序入口

汇编入口：start.s 不建议修改此文件，修改不当可能导致固件无法运行

C 程序入口：main.c 用户可以在 CreateUserTask 函数中创建自己的 Task，完成定制化功能。目前 DEMO CODE 提供了一个 CreateDemoTask 用于测试 SDK 提供的 API，客户可以根据实际需要修改。

```
void CreateUserTask(void)
{
    printf("\n user task\n");
#if WM_DEMO
    CreateDemoTask();
#endif
}
```

4.4 内存管理

W500（HED10W07SN）共有 384K Byte SRAM，地址空间为【0x00000 – 0x60000】用于加载固件代码和系统运行时堆栈的开销。

修改 start.s 中如下代码：

Heap_Size EQU 0x00010000

可以配置系统堆的大小，目前默认配置为 64K Bytes，用户可用堆空间约 38K Bytes。

5 编程接口

5.1 操作系统接口

系统目前使用的操作系统是 ucOS，后续的版本会增加 freeRTOS，为了兼容两套 OS，所以对 OS 的接口进行封装。推荐使用封装接口。

Ucos 接口 api 详细参数说明参见 wm_os.h。

封装接口 api 详细参数说明参见 wm_osal_ucos.h。

5.1.1 任务相关接口

5.1.1.1 ucos 接口

```
INT8U OSTaskCreate(void (*task)(void *p_arg), void *p_arg, OS_STK *ptos,INT8U prio);
INT8U OSTaskSuspend(INT8U prio);
INT8U OSTaskResume(INT8U prio);
void OSStart(void);
```

5.1.1.2 封装 api 接口

```
static __inline tls_os_status_t tls_os_task_create(tls_os_task_t *task,
        const char* name,
        void (*entry)(void* param),
        void* param,
        u8 *stk_start,
        u32 stk_size,
        u32 prio,
        u32 flag);
static __inline tls_os_status_t tls_os_task_suspend(tls_os_task_t *task);
static __inline tls_os_status_t tls_os_task_resume(tls_os_task_t *task);
```

5.1.2 信号量相关接口

5.1.2.1 ucos 相关接口

```
OS_EVENT *OSSemCreate(INT16U cnt);
OS_EVENT *OSSemDel(OS_EVENT *pevent, INT8U opt, INT8U *perr);
void OSSemPend(OS_EVENT *pevent, INT32U timeout, INT8U *perr);
INT8U OSSemPost(OS_EVENT *pevent);
void OSSemSet (OS_EVENT *pevent, INT16U cnt, INT8U *perr);
```

5.1.2.2 封装 api 接口

```
static __inline tls_os_status_t tls_os_sem_create(tls_os_sem_t **sem, u32 cnt);
static __inline tls_os_status_t tls_os_sem_delete(tls_os_sem_t *sem);
static __inline tls_os_status_t tls_os_sem_acquire(tls_os_sem_t *sem, u32 wait_time);
static __inline tls_os_status_t tls_os_sem_release(tls_os_sem_t *sem);
static __inline tls_os_status_t tls_os_sem_set(tls_os_sem_t *sem, INT16U cnt);
```

5.1.3 消息队列相关接口

5.1.3.1 ucos 相关接口

```
OS_EVENT *OSQCreate(void **start, INT16U size);
OS_EVENT *OSQDel (OS_EVENT *pevent, INT8U opt, INT8U *perr);
INT8U OSQPost(OS_EVENT *pevent, void *pmsg);
void *OSQPend(OS_EVENT *pevent, INT32U timeout, INT8U *perr);
```

5.1.3.2 封装 api 接口

```
static __inline tls_os_status_t tls_os_queue_create(tls_os_queue_t **queue,
        void *queue_start,
        u32 queue_size,
        u32 msg_size);
static __inline tls_os_status_t tls_os_queue_delete(tls_os_queue_t *queue);
static __inline tls_os_status_t tls_os_queue_send(tls_os_queue_t *queue,
        void *msg,
```

```
        u32 msg_size);
static __inline tls_os_status_t tls_os_queue_receive(tls_os_queue_t *queue,
        void **msg,
        u32 msg_size,
        u32 wait_time);
```

5.1.4 邮箱相关接口

5.1.4.1 ucos 相关接口

```
OS_EVENT *OSMboxCreate(void *pmsg);
INT8U OSMboxPost(OS_EVENT *pevent, void *pmsg);
void *OSMboxPend(OS_EVENT *pevent, INT32U timeout, INT8U *perr);
```

5.1.4.2 封装 api 接口

```
static __inline tls_os_status_t tls_os_mailbox_create(tls_os_mailbox_t **mailbox,
        void *mailbox_start,
        u32 mailbox_size,
        u32 msg_size);
static __inline tls_os_status_t tls_os_mailbox_delete(tls_os_mailbox_t *mailbox);
static __inline tls_os_status_t tls_os_mailbox_send(tls_os_mailbox_t *mailbox, void *msg);
static __inline tls_os_status_t tls_os_mailbox_receive(tls_os_mailbox_t *mailbox,
        void **msg,
        u32 wait_time);
```

5.1.5 定时器相关接口

5.1.5.1 ucos 相关接口

```
OS_TMR *OSTmrCreate(INT32U dly, INT32U period, INT8U opt, OS_TMR_CALLBACK
callback, void *callback_arg, INT8U *pname, INT8U *perr);
BOOLEAN OSTmrStart(OS_TMR *ptmr, INT8U *perr);
BOOLEAN OSTmrStop(OS_TMR *ptmr, INT8U opt, void *callback_arg, INT8U *perr);
```

5.1.5.2 封装 api 接口

```
static __inline tls_os_status_t tls_os_timer_create(tls_os_timer_t **timer,
                TLS_OS_TIMER_CALLBACK callback,
                void *callback_arg,
                u32 period,
                bool repeat,
                u8 *name);
static __inline void tls_os_timer_start(tls_os_timer_t *timer);
static __inline void tls_os_timer_change(tls_os_timer_t *timer, u32 ticks);
static __inline void tls_os_timer_stop(tls_os_timer_t *timer);
```

5.1.6 时间相关接口

5.1.6.1 ucos 相关接口

```
INT32U OSTimeGet (void);
void OSTimeDly(INT32U ticks);
```

5.1.6.2 封装 api 接口

```
static __inline u32 tls_os_get_time(void);
static __inline void tls_os_time_delay(u32 ticks);
```

5.1.7 中断相关接口

5.1.7.1 ucos 相关接口

```
OS_CPU_SR OSCPUSaveSR(void);
void OSCPURestoreSR(OS_CPU_SR cpu_sr);
```

5.1.7.2 封装 api 接口

```
static __inline u32 tls_os_set_critical(void);
static __inline void tls_os_release_critical(u32 cpu_sr);
```

5.2 驱动接口

5.2.1 Gpio 相关接口

void tls_gpio_cfg(u16 gpio_pin, u16 dir, u16 attr);

u16 tls_gpio_read(u16 gpio_pin);

void tls_gpio_write(u16 gpio_pin, u16 value);

void tls_gpio_int_enable(u16 gpio_pin, u16 mode);

void tls_gpio_int_disable(u16 gpio_pin);

int tls_get_gpio_int_flag(u16 gpio_pin);

void tls_clr_gpio_int_flag(u16 gpio_pin);

void tls_gpio_isr_register(void (*callback)(void *context), void *context);

说明：对 gpio 读写操作之前，需要先用 tls_gpio_cfg 接口配置。

使用 gpio 的中断功能，需要先用 tls_gpio_cfg 配置，然后用 tls_gpio_isr_register 注册回调，再用 tls_gpio_int_enable 使能即可。

5.2.2 Uart 相关接口

Uart 有两组接口，uart0 和 uart1，串口打印使用 uart0，uart1 用于传输数据，用户模式的接口默认使用 uart1。

int tls_uart_port_init(int uart_no, tls_uart_options_t *opts);

初始化函数，uart0 和 uart1 需要单独初始化，

void tls_uart_cfg_user_mode(void);

void tls_uart_disable_user_mode(void);

说明：uart1 接口在固件启动时已经初始化，用于固件指令传输。如果用户需要使用 uart1 作为私有数据传输，需要先用 tls_uart_cfg_user_mode 切换成用户模式。

Uart rx 数据：

void tls_uart_rx_register(u16 uart_no, s16(*rx_callback)(char *buf, u16 len));

void tls_user_uart_rx_register(s16 (*rx_callback)(char *buf, u16 len));

说明：注册 rx 回调函数。

两个接口功能一致，tls_user_uart_rx_register 专用于 uart1 rx 回调函数的注册。

从注册的回调函数中可以收到 uart rx FIFO 中一次收到的数据，不超过 32 字节。用户需要把数据拷贝出去，否则下次中断会覆盖。不建议在回调函数中做数据解析等长时间处理，以免影响中断接收新的数据。

Uart tx 数据：Tx 数据有两种方式。

方法一：使用下面的几个接口。

int tls_uart_write(struct tls_uart_port *port, char *buf, u32 count);

说明：把数据写入 uart tx 缓存。

void tls_uart_tx_chars_start(struct tls_uart_port *port);

说明：把缓存数据放到 uart tx FIFO，启动传输。

void tls_uart_tx_register(u16 uart_no, s16 (*tx_callback)(struct tls_uart_port *port));

void tls_user_uart_tx_register(s16 (*tx_callback)(struct tls_uart_port *port));

说明：注册 tx 回调函数。两个接口功能一致，tls_user_uart_tx_register 专用于 uart1 tx 回调函数的注册。

工作流程：用 tls_uart_write 把数据写入 tx 缓存中，然后调用 tls_uart_tx_chars_start 启动传输，tx 缓存剩余数据不超过 256 字节时，会调用注册的 tx 回调函数，在回调函数中，如果还有数据没有放入缓存，则把数据放入缓存，继续传输。

方法二：是对方法一中接口的封装。推荐使用。

int tls_uart_tx(char *buf, u16 len);

说明：异步方式，函数直接返回，中断中传输。

int tls_uart_tx_sync(char *buf, u16 len);

说明：同步方式，传输完毕之后返回。

int tls_uart_tx_length(void);

说明：获取已经传输的数据长度。

方法二接口是为了方便使用，对方法一中几个接口的封装，tx 缓存 TLS_UART_TX_BUF_SIZE(4096)字节，如果用户数据长度超过缓存长度，会先传输一部分，待缓存空余一部分时再次拷贝传输， 但是建议用户一次传输数据长度不要超过缓存长度。

int tls_user_uart_set_parity(TLS_UART_PMODE_T paritytype);

int tls_user_uart_set_baud_rate(u32 baudrate);

int tls_user_uart_set_stop_bits(TLS_UART_STOPBITS_T stopbits);

5.2.3 Host spi 相关接口

int tls_spi_init(void);

void tls_spi_slave_sel(u16 slave);

int tls_spi_setup(u8 mode, u8 cs_active, u32 fclk);

int tls_spi_write(const u8 *buf, u32 len);

int tls_spi_read(u8 *buf, u32 len);

int tls_spi_write_then_read(const u8 *txbuf, u32 n_tx, u8 *rxbuf, u32 n_rx);

int tls_spi_write32_then_writen(const u32 *buf32, const u8 *txbuf, u32 n_tx);

void tls_spi_trans_type(u8 type); //not used，

说明：host spi 固件启动时已经初始化，用于 flash 的读写。

Host spi 可以同时操作多个从设备，切换设备使用 tls_spi_slave_sel。

tls_spi_trans_type 接口是旧有接口，现在已经去除，统一使用 DMA 传输。

5.2.4 Flash 相关接口

int tls_fls_read(u32 addr, u8 *buf, u32 len);

int tls_fls_write(u32 addr, u8 *buf, u32 lenght);

5.2.5 SPI 从接口

详见 wm_hspi.h

int tls_slave_spi_init(int portmode);

void tls_set_hspi_user_mode(u8 ifenable);

void tls_set_hspi_interface_type(int type);

void tls_hspi_rx_cmd_register(s16 (*rx_cmd_callback)(char *buf));

收到 rx cmd 中断时，调用注册的回调函数，把 cmd buffer 返回。

void tls_hspi_rx_data_register(s16 (*rx_data_callback)(char *buf));

收到 rx data 中断时，调用注册的回调函数，把 data buffer 返回。

void tls_hspi_tx_data_register(s16 (*tx_data_callback)(char *buf));

收到 tx data 完成中断时，调用注册的回调函数。可不注册。

void tls_hspi_tx_data(char *txbuf, int len);

说明：spi 从接口固件启动时已经初始化，并在固件的任务中使用。

用户使用之前先用 tls_set_hspi_user_mode 配置成用户模式。

该套接口由 slave spi 和 slave sdio 共用，通过 tls_set_hspi_interface_type 设置不同接口类型。

5.2.6 Irq 相关接口

void tls_irq_init(void);

void tls_irq_register_handler (u8 vec_no, intr_handler_func handler, void *data);

void tls_irq_enable(u8 vec_no);

void tls_irq_disable(u8 vec_no);

5.2.7 Timer 定时器相关接口

void tls_timer_irq_register(void (*callback)(void));

void tls_timer_start(int timeout);

void tls_timer_stop(void);

说明：使用 timer 定时，需要先用 tls_timer_irq_register 注册回调函数，然后使用 tls_timer_start 启动计时，单位是 us，计时时间到，会调用回调函数。在回调函数中需要调用 tls_timer_stop 清除 timer 中断。

5.2.8 系统复位相关接口

void tls_watchdog_clr(void);

void tls_watchdog_isr(void *data);

void tls_watchdog_init(void);

void tls_sys_reset(void);
void tls_sys_clk_set(u32 clk);
切换系统时钟。系统支持 40M，80M，160M 三种时钟，接口的详细使用参见 wm_cpu.h。

5.2.9　读写 mac 地址及 txgain 接口
int tls_get_mac_addr(u8 *mac);
int tls_set_mac_addr(u8 *mac);
int tls_get_tx_gain(u8 *txgain);
int tls_set_tx_gain(u8 *txgain);

5.3　Wi-Fi 功能接口

详见 wm_wifi.h
5.3.1　Wi-Fi 监听模式接口
void tls_wifi_set_listen_mode(u8 enable);

typedef void (*tls_wifi_data_recv_callback)(u8* data, u32 data_len);
void tls_wifi_data_recv_cb_register(tls_wifi_data_recv_callback callback);

void tls_wifi_change_chanel(u32 chanid);

用户可以通过该接口监听空中任意设备发出的 802.11 帧；典型的应用是在没有任何连接的情况下自动配置设备与路由器相连。
Enable 参数：
1：表示启动监听模式；
0：表示禁止监听模式；
用户使能监听模式前，需要注册接收数据的回调函数；
硬件收到任何的数据包都会通过该回调函数处理；

对于 802.11 的数据帧字段定义以及字段处理的接口定义在 wm_ieee80211.h 中；

用户可以使用 tls_wifi_change_chanel 函数选择要监听的信道；系统初始化时会默认工作在 1 信道。

5.3.2 Wi-Fi 加网接口

u8 tls_wifi_connect(u8 *ssid, u8 ssid_len, u8 *pwd, u8 pwd_len);

u8 tls_wifi_connect_by_bssid(u8 *bssid, u8 *pwd, u8 pwd_len);

说明：该接口设置 Wi-FI 网络名称和密码并触发 Wi-FI 加网动作

用户调用此接口启动联网之前，需要注册 Wi-FI 状态回调接口；

如果使用 bssid 作为参数来联网，默认 bssid 为 MAC 地址，长度 6 个字节；

#define WIFI_JOIN_SUCESS 0x1

#define WIFI_JOIN_FAILED 0x2

#define WIFI_DISCONNECTED 0x3

void tls_wifi_status_change_cb_register(void (*callback)(u8 status));

Wi-Fi 网络状态变化后，此回调会被调用。但此回调只是通知 Wi-Fi MAC 层的状态变化，表明 Wi-Fi 成功连接到 AP（WIFI_JOIN_SUCESS）； 或者没有连接成功（WIFI_JOIN_FAILED）；或者连接之后断开（WIFI_DISCONNECTED）；通常 Wi-Fi 连接成功之后，应用程序会启动 DHCP Client 获取 IP 地址；

用户也可以调用：err_t tls_netif_add_status_event(tls_netif_status_event_fn event_fn);

注册网络层的回调函数，该回调会返回上述 Wi-Fi 状态，以及获取到 IP 地址之后的状态；详见 wm_netif.h 接口文件；

当用户调用加网接口后，20s 没有联网成功会返回 WIFI_JOIN_FAILED；

5.3.3 创建 AP 接口

int tls_wifi_softap_create (struct tls_softap_info_t *apinfo, struct tls_ip_info_t *ipinfo);

用户创建 AP 时，需要提供 AP 所需的 SSID, 密码，加密方式，即默认 IP 地址；

其中加密相关的信息通过以下数据结构。

用户需要注册网络状态回调函数来获取创建 AP 成功与否的状态；

表 5-1

数据结构	字段	含义

struct tls_key_info_t{
u8 format;
u8 index;
u8 key_len;
u8 key[64];
};
	u8 format	密钥格式，0 表示 HEX，1 表示 ASCII

	u8 index	密钥索引号，1～4 用于 WEP 加密密钥(默认为 1)，其它加密方式固定为 0
	u8 key_len	密钥长度
	u8 key[64]	密钥。例如：12345，

表 5-2 密钥

安全模式	密钥格式	
	HEX	ASCII
WEP64	10 个 16 进制字符（注 1）	5 个 ASCII 字符（注 2）
WEP128	26 个 16 进制字符	13 个 ASCII 字符
WPA-PSK(TKIP)	64 个 16 进制字符	8~63 个 ASCII 字符
WPA-PSK(CCMP/AES)	64 个 16 进制字符	8~63 个 ASCII 字符
WPA2-PSK(TKIP)	64 个 16 进制字符	8~63 个 ASCII 字符
WPA2-PSK(CCMP/AES)	64 个 16 进制字符	8~63 个 ASCII 字符

注 1：16 进制字符指 0~9、a~f（不区分大小写），如"11223344dd"

注 2：ASCII 字符指国际标准化组织（ISO）规定的标准 ASCII 字符集中的数字 0~9 与字母 a~z（区分大小写），如"14u6E"

表 5-3

数据结构	字段	含义

struct tls_softap_info_t{
u8 ssid[32];
u8 encrypt;
u8 channel;
struct tls_key_info_t keyinfo;
};
	u8 ssid[32]	密钥格式，0 表示 HEX，1 表示 ASCII

	u8 encrypt	加密类型（见表 5-4）
	u8 channel	AP 信道号，有效范围 1～14；0 或者其他，系统默认为 1
	struct tls_key_info_t keyinfo	见表 5-2

表 5-4 Encrypt 定义：

值	含 义
0	OPEN
1	WEP64
2	WEP128

5.3.4 Wi-Fi 断开网络接口

void tls_wifi_disconnect(void);

5.3.5 一键配置联网接口

一键配置：使用手机端（android/iOS）应用程序对模块进行配置，使之自动连接到用户选择的 AP（与用户手机相连接的 AP）。

void tls_wifi_set_oneshot_flag(u8 flag);

int tls_wifi_get_oneshot_flag(void);

说明：设置一键加网标志，获取一键加网标志

WinnerMicro 实现了一套基于监听模式的自动配置路由器参数的功能——一键加网；如果用户需要使用 WinnerMicro 提供的一键加网功能，使用此接口。此接口调用后，模块进入一键配置模式，此时用户需要启动手机一键配置应用程序进行配置，配置成功后，模块利用配置获得的 BSSID 和密码进行联网，并将联网结果反馈给用户；

Flag:

1: 表示使能一键配置功能；

0: 停止一键配置功能；

用户需要先注册网络状态回调函数，再调用此接口；具体描述见 Wi-Fi 加网接口；

5.3.6 WPS 接口：启动 PIN 模式联网

int tls_wps_start_pin(void)

int tls_wps_get_pin(u8* pin)

int tls_wps_set_pin(u8* pin, u8 pin_len)

使用 PIN 模式 WPS 功能，用户需要为模块提供 PIN CODE（8 字节）; PIN CODE 获得方式：

1. 生产模块时，或者系统初始化时将一组 PIN CODE 设置到模块，使用 tls_wps_set_pin 接口；通常没有显示的设备使用此种方式；
   对于这种情况，用户可以直接调用 tls_wps_start_pin 启动 PIN 模式，模块将使用存储在 Flash 中的 PIN;
2. 调用 tls_wps_get_pin 接口从系统获取随机生产的 PIN CODE. 并显示在界面或者显示屏上，通常用户需要调用 tls_wps_set_pin 将随机获取的 PIN 写会模块 flash 保存，在调用 tls_wps_start_pin 启动 PIN 模式

启动 WPS 模式联网后，模块会自动连接网络，并返回联网结果；用户程序需要注册获取联网结果的回调函数，具体参考“Wi-Fi 加网接口”

180s 为超时时间，会返回 WIFI_JOIN_FAILED 消息；
超时后，需要再次调用启动联网；

5.3.7 WPS 接口：启动按键(PBC)模式联网
int tls_wps_start_pbc(void)
调用此接口后，程序将启动 PBC 模式 WPS 功能；

启动 WPS 模式联网后，模块会自动连接网络，并返回联网结果；用户程序需要注册获取联网结果的回调函数，具体参考“Wi-Fi 加网接口”

180s 为超时时间，会返回 WIFI_JOIN_FAILED 消息；
超时后，需要再次调用启动联网；

5.3.8 Wi-Fi 数据接口
typedef int (*net_rx_data_cb)(u8 *buf, u32 buf_len);
void tls_ethernet_data_rx_callback(net_rx_data_cb callback);
用户注册此回调函数，可以接收到网卡所连接网络内正常的数据帧；
通常情况下，该接口被上层 TCP/IP 协议模块调用，例如 LWIP。

5.3.9 Wi-Fi 扫描接口
int tls_wifi_scan(void);
void tls_wifi_scan_result_cb_register(void (*callback)(void));
int tls_wifi_get_scan_rslt(u8* buf, u32 buffer_size);

用户可以使用上述接口实现主动扫描，并获取扫描结果的功能；在调用 tls_wifi_scan 启动扫描前，需要使用 tls_wifi_scan_result_cb_register 函数注册自己的回调函数；Wi-Fi 扫描完成后，会回调此函数。在回调函数中，用户可以调用 tls_wifi_get_scan_rslt 函数来读取扫描的 BSS（附近 AP）列表；用户需要提供存储 BSS 列表的缓冲区。缓冲区大小决定返回的 BSS 数目；每条 BSS 占用 43Bytes。

5.3.10 设置/获取 Wi-Fi 自动重连标志接口

int tls_wifi_auto_connect_flag(u8 opt, u8* mode);

用户如果希望启动自动联网功能，只需要调用此接口设置自动重连标志。
注：该功能需要用户使用 SDK 中 tls_sys 任务；该任务专注处理网络状态变化；
如果不使用该任务，用户可以自行实现重连功能。

5.3.11 获取当前 Wi-Fi 网络信息接口

void tls_wifi_get_current_bss(struct tls_curr_bss_t* bss)

在联网成功后，用户可以调用该接口获取当前 BSS（AP）的信息，例如信道，加密类型，BSSID，信号强度等；

5.3.12 获取 Wi-Fi 层状态的接口

enum tls_wifi_states tls_wifi_get_state(void);

5.3.13 获取无线报文更多信息的接口

void tls_wifi_data_ext_recv_cb_register(tls_wifi_data_ext_recv_callback callback)

使用该接口不但可以接收到报文，还能取到该报文携带的 rssi 信息。

5.3.14 获取已经关联上的所有 STA 的接口

void tls_wifi_get_authed_sta_info(u32 *sta_num, u8 *buf, u32 buf_size)

调用此函数，可以获取 AP 和 APSTA 已经关联上的所有 STA 信息。

5.3.15 获取 Wi-Fi 错误码相关的接口

int tls_wifi_get_errno(void);

void tls_wifi_perror(const char *info);

```
const char *tls_wifi_get_errinfo(int errno);
```

5.3.16 APSTA 加网相关的接口

```
int tls_wifi_apsta_start(u8 *ssid, u8 ssid_len, u8 *pwd, u8 pwd_len, u8 *apsta_ssid, u8 apsta_ssid_len);
int tls_wifi_apsta_start_by_bssid(u8 *bssid, u8 *pwd, u8 pwd_len, u8 *apsta_ssid, u8 apsta_ssid_len);
```

5.4 网络接口

5.4.1 网络应用接口

5.4.1.1 网络初始化接口

```
int tls_ethernet_init(void);
```

5.4.1.2 获取网络状态以及 IP 接口

```
struct tls_ethif {
    struct ip_addr ip_addr;
    struct ip_addr netmask;
    struct ip_addr gw;
    struct ip_addr dns1;
    struct ip_addr dns2;
    u8 status;//0:net down; 1:net up
};
struct tls_ethif * tls_netif_get_ethif(void);
```

5.4.1.3 DHCP Client 相关以及设定 IP、加网、断网接口

```
err_t tls_dhcp_start(void);
err_t tls_dhcp_stop(void);
err_t tls_netif_set_addr(ip_addr_t *ipaddr,
                         ip_addr_t *netmask,
                         ip_addr_t *gw);
err_t tls_netif_set_up(void);
err_t tls_netif_set_down(void);
```

tls_netif_set_up: 调用此接口成功后，模块可以进行 IP 数据通信

tls_netif_set_down: 调用此接口成功后，模块不可以进行 IP 数据通信注：通常只有收到 Wi-Fi 断网消息时，调用此接口；

对于 DHCP 获取 IP 地址，在 wifi 加网成功后，只需调用 tls_dhcp_start 接口，启动 DHCP 自动获取 IP 地址；
对于静态 IP 地址，先调用 tls_netif_set_addr 接口设置 IP 地址等，再调用 tls_netif_set_up 启用网络。

用户启动 DHCP 或者设置静态 IP 地址之前，需要注册网络状态回调函数，以获取返回状态；具体见“网络状态回调注册、删除接口”

5.4.1.4 网络状态回调注册、删除接口

```
#define  NETIF_WIFI_JOIN_SUCESS          0x1
#define NETIF_WIFI_JOIN_FAILED           0x2
#define  NETIF_WIFI_DISCONNECTED        0x3
#define  NETIF_IP_NET_UP                        0x4

typedef void (*tls_netif_status_event_fn)(u8 staus);
err_t tls_netif_add_status_event(tls_netif_status_event_fn event_fn);
err_t tls_netif_remove_status_event(tls_netif_status_event_fn event_fn);
```

如果用户没有通过 tls_wifi_status_change_cb_register 接口注册 Wi-Fi 状态回调，网络层会在初始化时，默认注册此回调。用户通过 tls_netif_add_status_event 接口注册网络状态回调后，回调函数中会返回上述定义的四种状态。

如果用户在网络初始化后，调用 tls_wifi_status_change_cb_register 接口注册 Wi-Fi 状态回调，那么用户再通过 tls_netif_add_status_event 接口注册网络状态回调，则回调函数只会返回 NETIF_IP_NET_UP 一种状态。其他三种状态会由 Wi-Fi 层返回；

tls_netif_add_status_event 接口可以被重复调用，对于相同的回调函数指针，如果已经注册了，则不再进行重复注册。不同的任务调用此接口，会形成一个回调函数链表，网络状态发生变化时，会逐一执行回调函数；

因此：推荐的设计是，回调函数内只发送消息给相关任务，然后立即返回。

5.4.1.5 DHCP 服务器接口

INT8S tls_dhcps_start(void);

void tls_dhcps_stop(void);

5.4.1.6 DNS 服务器接口

INT8S tls_dnss_start(INT8U * DnsName);

void tls_dnss_stop(void);

void tls_netif_dns_setserver(u8 numdns, ip_addr_t *dnsserver);

5.4.2 RAW Socket 接口

RAW Socket 接口采用回调的方式，返回 socket 接收的数据、socket 错误信息、socket 连接成功、socket 作为 server 监听到有客户端连接进来等信息。回调函数在 TCP/IP 的任务中被执行，要求及时处理数据并返回，推荐只在这些回调函数中发送消息给相关任务，在其它任务里处理用户逻辑。

typedef void (*socket_err_fn)(u8 skt_num, err_t err);

typedef err_t (*socket_recv_fn)(u8 skt_num, struct pbuf *p, err_t err);

typedef err_t (*socket_connected_fn)(u8 skt_num, err_t err);

typedef err_t (*socket_poll_fn)(u8 skt_num);

typedef err_t (*socket_accept_fn)(u8 skt_num, err_t err);

表 5-1 tls_socket_desc 数据结构定义

数据结构	字段	含义
struct tls_socket_desc { enum tls_socket_cs_mode cs_mode; enum tls_socket_protocol protocol; u8 ip_addr[4]; u16 port; u16 localport; char host_name[32];	cs_mode	Server or Client 模式
	protocol	TCP or UDP 协议
	ip_addr	对 TCP Client 或 UDP，表示远程 IP 地址；对 TCP Server 该值没有意义
	port	对 TCP Client 或 UDP，表示远程服务器的端口；对 TCP Server，表示本地监听的端口
	localport	对 TCP Client 或 UDP，表示本地监听的端口；对 TCP Server 该值没有意义

u8 host_len; u32 timeout; socket_err_fn errf; socket_recv_fn recvf; socket_connected_fn connf; socket_poll_fn pollf; socket_accept_fn acceptf; };	host_name	远程服务器名，保留，目前没有实现
	host_len	远程服务器名长度，保留，目前没有实现
	timeout	空闲超时，保留，目前没有实现
	errf	TCP 连接出错回调
	recvf	UDP、TCP 接收数据回调
	connf	TCP Client 连接成功回调
	pollf	TCP 空闲回调
	acceptf	TCP Server 监听到连接回调

int tls_socket_create(struct tls_socket_desc * skd);

int tls_socket_send(u8 skt_num, void *pdata, u16 len);

int tls_socket_close(u8 skt_num);

用户通过设置 tls_socket_desc 数据结构，设置需要创建的 Socket 的类型（UDP 或 TCP）、TCP，是作为 Server 或 Client、IP 地址、端口以及回调函数指针，调用 tls_socket_create 接口创建所需的 Socket，通过 tls_socket_send 接口发送数据，通过 tls_socket_close 接口关闭 Socket。

5.4.2.1 pbuf 数据结构相关接口

struct pbuf *pbuf_alloc(pbuf_layer l, u16_t length, pbuf_type type);

u8 pbuf_free(struct pbuf *p);

u16_t pbuf_copy_partial(struct pbuf *p, void *dataptr, u16_t len, u16_t offset);

pbuf 数据结构，是 Raw Socket 接收数据回调函数的参数类型，用户通过以上接口操作此数据结构。包括从 pbuf 结构中 copy 数据到用户自定义 buffer 中，分配和释放此数据结构。需要注意的是，如果在 Socket 接收数据回调函数中返回正确（ERR_OK）时，需要用户自己通过 pbuf_free 接口释放 pbuf 结构，否则，不需要用户释放此结构，系统会自行释放它。

5.4.3 标准 Socket 接口

int accept(int s, struct sockaddr *addr, socklen_t *addrlen);

int bind(int s, const struct sockaddr *name, socklen_t namelen);

int shutdown(int s, int how);

int closesocket(int s);

```
int connect(int s, const struct sockaddr *name, socklen_t namelen);
int getsockname(int s, struct sockaddr *name, socklen_t *namelen);
int getpeername(int s, struct sockaddr *name, socklen_t *namelen);
int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);
int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen);
int listen(int s, int backlog);
int recv(int s, void *mem, size_t len, int flags);
int recvfrom(int s, void *mem, size_t len, int flags,
                    struct sockaddr *from, socklen_t *fromlen);
int send(int s, const void *data, size_t size, int flags);
int sendto(int s, const void *data, size_t size, int flags,
            const struct sockaddr *to, socklen_t tolen);
int socket(int domain, int type, int protocol);
int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
                     struct timeval *timeout);
int ioctlsocket(int s, long cmd, void *argp);
struct hostent* gethostbyname(const char *name);
```

5.5 升级接口

固件支持两种升级方式，uart 升级和 web 升级。

5.5.1 基础升级相关接口

```
int tls_fwup_init(void);
u32 tls_fwup_enter(enum tls_fwup_image_src image_src);
int tls_fwup_exit(u32 session_id);
int tls_fwup_request_sync(u32 session_id, u8 *data, u32 data_len);
u16 tls_fwup_current_state(u32 session_id);
int tls_fwup_reset(u32 session_id);
int tls_fwup_clear_error(u32 session_id);
int tls_fwup_set_crc_error(u32 session_id);
int tls_fwup_get_status(void);
int tls_fwup_set_update_numer(int number);
```

int tls_fwup_get_current_session_id(void);

说明：固件启动时已经初始化升级接口。
升级时需要用 tls_fwup_enter 选择升级模式，两种类型：TLS_FWUP_IMAGE_SRC_LUART，TLS_FWUP_IMAGE_SRC_WEB
用 tls_fwup_request_sync 即可把升级数据写入 flash 中。
详见 wm_fwup.h 接口文件

5.5.2 Http 升级接口

Http 升级以上述接口为基础，通过 tls_fwup_enter 选择升级源。

```
#define   HTTPAuthSchemaNone        0
#define   HTTPAuthSchemaBasic       1
#define   HTTPAuthSchemaDigest      2
#define   HTTPAuthSchemaKerberos    3
#define   HTTPAuthNotSupported      4

typedef struct _HTTPParameters
{
    CHAR*                   Uri;
    CHAR*                   ProxyHost;
    UINT32                  UseProxy ;
    UINT32                  ProxyPort;
    UINT32                  Verbose;
    CHAR*                   UserName;
    CHAR*                   Password;
    UINT32                  AuthType;

} HTTPParameters;
UINT32    http_fwup(HTTPParameters ClientParams);
```

用户可以调用此接口，从 Uri 指定的 HTTP 服务器地址 Download 并升级系统固件。

5.6 参数区相关接口

参数区详细结构以及接口定义参见 wm_param.h。

```
int tls_param_set(int id, void *argv, bool to_flash);
```

int tls_param_get(int id, void *argv, bool from_flash);

5.7 Memory 管理接口

tls_mem_alloc 对应 malloc
tls_mem_free 对应 free

5.8 HTTP Client 接口

5.8.1 打开和关闭请求

```
HTTP_SESSION_HANDLE HTTPClientOpenRequest(HTTP_CLIENT_SESSION_FLAGS Flags);
UINT32        HTTPClientCloseRequest(HTTP_SESSION_HANDLE *pSession);
```

打开和关闭一个 Http 请求，打开时分配一块内存空间保存 Http 状态，关闭时释放该空间。

5.8.2 设置身份验证方式及证书

```
UINT32    HTTPClientSetAuth(HTTP_SESSION_HANDLE pSession,
                            HTTP_AUTH_SCHEMA AuthSchema,
                            void *pReserved);
UINT32       HTTPClientSetCredentials (HTTP_SESSION_HANDLE pSession,
                                       CHAR *pUserName,
                                       CHAR *pPassword);
```

设置身份验证方式和用户名、密码，目前仅支持 Basic 和 Digest 身份验证方式。

5.8.3 设置代理服务器

```
UINT32 HTTPClientSetProxy(HTTP_SESSION_HANDLE pSession,
                          CHAR *pProxyName,
                          UINT16 nPort,
                          CHAR *pUserName,
                          CHAR *pPassword);
```

设置 Http 代理服务器地址、端口、用户名和密码。

5.8.4 设置 Http Method

```
UINT32 HTTPClientSetVerb(HTTP_SESSION_HANDLE pSession, HTTP_VERB HttpVerb);
```

5.8.5 添加请求消息报头及发送请求正文

```
UINT32 HTTPClientAddRequestHeaders (HTTP_SESSION_HANDLE pSession,
                                    CHAR *pHeaderName,
                                    CHAR *pHeaderData,
                                    BOOL nInsert);
```

```
UINT32 HTTPClientSendRequest (HTTP_SESSION_HANDLE pSession,
                              CHAR *pUrl, VOID *pData,
                              UINT32 nDataLength,
                              BOOL TotalLength,
                              UINT32 nTimeout,
                              UINT32 nClientPort);
```

5.8.6 接收服务器响应

```
UINT32  HTTPClientRecvResponse(HTTP_SESSION_HANDLE pSession,
                               UINT32 nTimeout);
```

5.8.7 读写请求、响应消息正文

```
UINT32 HTTPClientReadData(HTTP_SESSION_HANDLE pSession,
                          VOID *pBuffer,
                          UINT32 nBytesToRead,
                          UINT32 nTimeout,
                          UINT32 *nBytesRecived);
UINT32      HTTPClientWriteData(HTTP_SESSION_HANDLE pSession,
                          VOID *pBuffer,
                          UINT32 nBufferLength,
                          UINT32 nTimeout);
```

5.8.8 获取请求、响应状态等信息

```
UINT32  HTTPClientGetInfo (HTTP_SESSION_HANDLE pSession,
                           HTTP_CLIENT *HTTPClient);
```

5.8.9 检索响应消息报文头信息

```
UINT32 HTTPClientFindFirstHeader (HTTP_SESSION_HANDLE pSession,
                                  CHAR *pSearchClue,
                                  CHAR *pHeaderBuffer,
                                  UINT32 *nLength);
UINT32     HTTPClientGetNextHeader (HTTP_SESSION_HANDLE pSession,
                                  CHAR *pHeaderBuffer,
                                  UINT32 *nLength);
UINT32         HTTPClientFindCloseHeader (HTTP_SESSION_HANDLE pSession);
```

HTTPClientFindFirstHeader 接口初始化并返回第一个符合检索条件的响应消息头，例如：Content-length：215。结束检索需要调用 HTTPClientFindCloseHeader 接口释放资源。

5.9 加密接口

见 wm_crypto.h 文件；

本接口所有算法采用软件实现。

5.9.1 AES 接口

int aes_128_cbc_encrypt(const u8 *key, const u8 *iv, u8 *data, size_t data_len);

int aes_128_cbc_decrypt(const u8 *key, const u8 *iv, u8 *data, size_t data_len);

128 位 AES 加解密接口；

注：明文和密文都存储在 data 指向的地址，加密和解密之后的数据会替换原数据

5.9.2 MD5 接口

int md5(const u8 *addr, int len, u8 *mac);

计算 addr 指向的地址内 len 长度的数据的 16 字节 MD5 值，存储在 mac 指向的地址

5.9.3 SHA1 接口

int sha1(const u8 *addr, int len, u8 *mac);

注：mac 为 20 字节的内存区；

5.9.4 HMAC-MD5 接口

int hmac_md5(const u8 *key, size_t key_len, const u8 *data, size_t data_len, u8 *mac);

参考 RFC2104，输出 128bit MAC 值. mac 指向 16 字节内存区；

5.9.5 RC4 接口

int rc4(const u8 *key, size_t keylen, u8 *data, size_t data_len);

加解密使用同一个接口；data 指向明文和密文，执行加解密操作后，被对应的密文和明文代替；