

# 终端应用接口文档

## 库接口

### 1. 获取wiota库版本信息

- 查询版本信息  
wiota库的版本号、git信息以及编译时间
- 语法

```
void uc_wiota_get_version(u8_t *wiota_version, u8_t *git_info, u8_t *time, u32_t *cce_version);
```

- 描述  
wiota库的版本号、git信息以及编译时间
- 返回值  
无
- 参数  
wiota\_version: wiota 库的版本信息  
git\_info: wiota库对应git信息  
time: wiota库编译时间  
cce\_version: CCE版本号

### 2. 初始化

- 目的  
wiota系统初始化
- 语法

```
void uc_wiota_init(void);
```

- 描述  
初始化wiota资源，初始化线程，内存等。
- 返回值  
无
- 参数  
无

### 3. 启动wiota

- 目的  
启动wiota系统
- 语法

```
void uc_wiota_run(void);
```

- 描述  
启动wiota系统，进入NULL状态。
- 返回值  
无
- 参数  
无

## 4. 关闭wiota

- 目的  
关闭wiota系统
- 语法

```
void uc_wiota_exit(void);
```

- 描述  
关闭wiota系统，回收所有wiota系统资源。
- 返回值  
无
- 参数  
无

## 5. 连接同步ap

- 目的  
iote同步ap
- 语法

```
void uc_wiota_connect(void);
```

- 描述  
同步ap，同步帧结构，进入SYNC状态，wiota系统处于待命状态，随时可发起随机接入。
- 返回值  
无
- 参数  
无
- 注意  
在wiota启动之后调用

## 6. 断开与ap的同步

- 目的  
断开同步状态
- 语法

```
void uc_wiota_disconnect(void);
```

- 描述  
断开与AP的同步连接，回到NULL状态
- 返回值  
无
- 参数  
无

## 7. 查询wiotra当前状态

- 目的  
查询wiotra状态，为下一步操作做准备
- 语法

```
UC_WIOTA_STATUS uc_wiota_get_state(void);
```

- 描述  
查询wiotra当前状态
- 返回值  
状态枚举值

```
typedef enum {
    UC_STATUS_NULL = 0,
    UC_STATUS_SYNC,
    UC_STATUS_SYNC_LOST,
    UC_STATUS_SLEEP,
    UC_STATUS_ERROR,
} UC_WIOTA_STATUS;
```

初始化或者退出协议栈，处于NULL状态，  
同步开始但并未同步成功时，处于NULL状态，  
同步成功后，处于SYNC状态，  
同步失败后，或者在SYNC状态时出现异常，失步之后，处于SYNC\_LOST状态，  
协议栈休眠时，处于SLEEP状态，该状态暂未支持，  
其他状态均为ERROR状态。

- 参数  
无

## 8. 设置频点

- 目的  
设置频点，iote和ap需要设置相同频点才能同步
- 语法

```
void uc_wiota_set_freq_info(u8_t freq_idx);
```

- 描述  
设置频点，频点范围470M-510M，每200K一个频点

- 返回值  
无
- 参数  
频点idx, 范围0~200, 代表频点 ( $470+0.2*idx$ )
- 注意  
在初始化系统之后, 在系统启动之前调用, 否则无法生效

## 9. 查询频点

- 目的  
获取频点idx
- 语法

```
unsigned char uc_wiota_get_freq_info();
```

- 描述  
查询频点, 频点范围470M-510M, 每200K一个频点
- 返回值  
频点idx, 范围0~200, 代表频点 ( $470+0.2*idx$ )
- 参数  
无
- 注意  
无

## 10. 设置用户id

- 目的  
设置用户id
- 语法

```
int uc_wiota_set_userid(unsigned int* id, unsigned char id_len);
```

- 描述  
设置用户id, 此id为终端唯一标识
- 返回值  
0: 正常  
1: 参数异常
- 参数  
id: 用户id  
id建议使用2个32bit的16进制数值列表, 方便传输, 例: uid\_list = [0x12345678]  
id\_len: id长度, 取值范围1~4字节
- 注意  
目前只支持4字节长度的user id

## 11. 获取用户id

- 目的  
获取用户id
- 语法

```
void uc_wiota_get_userid(unsigned int* id, unsigned char* id_len);
```

- 描述  
获取用户id, 此id为终端唯一标识
- 返回值  
id: user\_id  
id\_len: id长度, 取值2,4,6,8字节
- 参数  
无
- 注意  
目前只支持4字节长度的user id

## 12. 获取系统配置

- 目的  
获取系统配置
- 语法

```
void uc_wiota_get_system_config(sub_system_config_t *config);
```

- 描述  
获取系统配置
- 返回值  
子系统配置结构表
- 参数  
无
- 结构体

```
typedef struct {
    unsigned char ap_max_pow;    // ap最大发射功率, 默认21db. 范围 0 - 31 db.
    unsigned char id_len;        // id长度, 取值0,1,2,3代表2,4,6,8字节
    unsigned char pn_num;        // 固定为1, 暂时不提供修改
    unsigned char symbol_length; // 帧配置, 取值0,1,2,3代表128,256,512,1024
    unsigned char dlu_l_ratio;   // 帧配置, 下上行比例, 取值0,1代表1:1和1:2
    unsigned char btvalue;       // 和调制信号的滤波器带宽对应, BT越大, 信号带宽越大,
    // 取值0,1代表1.2和0.3, BT=1.2的数据速率比BT=0.3的高
    unsigned char group_number;   // 帧配置, 取值0,1,2,3代表1,2,4,8个上行group数量,
    // 在symbol_length为0/1/2/3时, group_number最高限制为3/2/1/0
    unsigned char spectrum_idx;   // 频谱序列号, 默认为3, 即470-510M(具体见频谱idx表)
    unsigned int  systemid;       // 系统id
    unsigned int  subsystemid;    // 子系统id
    unsigned char na[48];
}sub_sysstrm_config_t;
```

频谱idx	低频MHz	高频MHz	中心频率MHz	带宽MHz	频点stepMHz	频点idx	频点个数
0 (other1)	223	235	229	12	0.2	0~60	61
1 (other2)	430	432	431	2	0.2	0~10	11
2 (EU433)	433.05	434.79	433.92	1.74	0.2	0~8	9
3 (CN470-510)	470	510	490	40	0.2	0~200	201
4 (CN779-787)	779	787	783	8	0.2	0~40	41
5 (other3)	840	845	842.5	5	0.2	0~25	26
6 (EU863-870)	863	870	866.5	7	0.2	0~35	36
7 (US902-928)	902	928	915	26	0.2	0~130	131

- 注意
  - (1) 子系统配置表需要与ap一样才能同步
  - (2) 暂不支持BT=1.2

### 13. 设置系统配置

- 目的  
设置系统配置
- 语法

```
void uc_wiota_set_system_config(sub_system_config_t *config);
```

- 描述  
设置系统配置
- 返回值  
无
- 参数  
子系统配置结构表
- 结构体  
同前一个接口
- 注意  
子系统配置表需要与ap一样才能同步

### 14. 获取无线信道状态

- 目的  
获取信道参数
- 语法

```
void uc_wiota_get_radio_info(radio_info_t *radio);
```

- 描述  
设置系统配置
- 返回值  
无线信道参数表:  
rssi: 信号强度  
ber: 误码率, 暂不支持  
snr: 信噪比, 范围 -25dB ~ 30dB  
cur\_pow: 当前发射功率, 范围 -16~21dBm  
max\_pow: 最大发射功率, 范围 -16~21dBm  
cur\_mcs: 当前数据发送速率级别, 范围 0~7  
max\_mcs: 截止目前最大数据发送速率级别, 范围 0~7
- 参数  
无
- 结构体

```
typedef struct {
    unsigned char    rssi; // absolute value, 0~150 means 0 ~ -150
    unsigned char    ber;
    signed char      snr;
    signed char      cur_pow;
    signed char      max_pow;
    unsigned char    cur_mcs;
    unsigned char    max_mcs;
}radio_info_t;
```

- 注意  
无

## 15. 设置数据传输速率

- 目的  
根据应用需求设置数据传输速率
- 语法

```
void uc_wiota_set_data_rate(unsigned char rate_mode, unsigned int rate_value);
```

- 描述  
设置最大速率模式和级别, 三种模式  
第一种基本模式, 是基本速率设置, 有9档mcs速率级别 (包括自动mcs), 详见 UC\_MCS\_LEVEL, 默认为自动mcs, 设置非自动mcs时同时关闭自动速率匹配功能  
在第一种模式的基础上, 在系统配置中dlul\_ratio为1:2时, 才能打开第二种模式, 打开该模式能够提高该帧结构情况下两倍速率, 默认第二种模式开启状态

在第一种模式的基础上，打开第三种模式，能够提升  $(8 \times (1 \ll \text{group\_number}))$  倍单终端的速率，但是会影响网络中其他终端的上行，建议在大数据量快速传输需求时使用  
备注：group\_number为系统配置中的参数

- 返回值  
无
- 参数  
rate\_mode: UC\_DATA\_RATE\_MODE  
rate\_value: 当rate\_mode为UC\_RATE\_NORMAL时，rate\_value为UC\_MCS\_LEVEL  
当rate\_mode为UC\_RATE\_MID时，rate\_value为0或1，表示关闭或打开  
当rate\_mode为UC\_RATE\_HIGH时，rate\_value为0，表示关闭，rate\_value为其他值，表示当实际发送数据量（byte）大于等于该值时才会真正开启该模式，常用建议设置rate\_value为100

```
typedef enum {
    UC_RATE_NORMAL = 0,
    UC_RATE_MID,
    UC_RATE_HIGH,
}UC_DATA_RATE_MODE;

typedef enum {
    UC_MCS_LEVEL_0 = 0,
    UC_MCS_LEVEL_1,
    UC_MCS_LEVEL_2,
    UC_MCS_LEVEL_3,
    UC_MCS_LEVEL_4,
    UC_MCS_LEVEL_5,
    UC_MCS_LEVEL_6,
    UC_MCS_LEVEL_7,
    UC_MCS_AUTO = 8,
}UC_MCS_LEVEL;
```

BT=0.3时在不同symbol length和不同MCS时，对应每帧传输的应用数据量（byte），表中0表示不支持该MCS

symbol length	mcs0	mcs1	mcs2	mcs3	mcs4	mcs5	mcs6	mcs7
128	5	7	50	64	78	0	0	0
256	5	13	20	50	106	155	190	0
512	5	13	29	40	71	134	253	295
1024	5	13	29	61	106	218	449	617

初始化协议栈时为自动速率匹配功能打开状态，调用该接口入参为0~7时，设置最大速率级别，同时关闭自动速率匹配功能，再次调用该接口入参为8（或者不是0~7）时，会打开自动速率匹配功能。重启协议栈也会恢复初始功能。  
为了保证接入成功率，接入短消息暂只使用mcs0~3，由于其中需要携带user id，正常会再减去4个字节空间，实际给应用的数据量会比正常短消息少。  
接入短消息的MCS还有其他限制（应用层可不关注），symbol length为128/256/512/1024时，接入短消息的MCS最高为1/2/3/3。

每帧时间长度（frameLen）的粗略计算公式：（单位微妙）



```
// dlGroupNum和ulGroupNum取值0,1,2,3, ulGroupNum即系统参数配置中的group_number
groupNum = (1 << dlGroupNum) + (1 << ulGroupNum);
symbolNum = 11 + 2 * (1 << pn_num) + 64 * groupNum; // pn_num目前固定为1
frameLen = symbolNum * 4 * 128 * (1 << symbol_length); // symbol_length取值为
0,1,2,3
```

举例：系统配置中group\_number为0，dlul\_ratio为0，symbol\_length为1，则

```
groupNum = 1 + 1 = 2;
symbolNum = 15 + 128 = 143;
frameLen = 143 * 4 * 128 * 2 = 146432 us
```

在此帧结构配置情况下，如果选择MCS2，则应用数据速率为  $8 \times 20 / 0.146432 = 1093$  bps  
(计算上行数据速率时，一般不考虑第一个包即随机接入包)

- 注意  
一味提高速率，可能导致上行始终无法成功

## 16. 设置DCXO

- 目的  
设置频偏
- 语法

```
void uc_wiota_set_dcxo(unsigned int dcxo);
```

- 描述  
每块芯片的频偏不同，在系统启动之前需要单独配置，测试模式使用，之后量产时会测好后固定写在系统静态变量中，不需要应用管理。
- 返回值  
无
- 参数  
dcxo：频偏
- 注意  
在初始化系统之后，在系统启动之前调用，否则无法生效

## 17. 设置终端连接时间

- 目的  
设置终端接入后保持的时间
- 语法

```
void uc_wiota_set_active_time(unsigned int active_s);
```

- 描述  
终端在接入后，即进入连接态，当无数据发送或者接收时，会保持一段时间的连接态状态，在此期间ap和终端双方如果有数据需要发送则不需要再进行接入操作，一旦传输数据就会重置连接时间，而在时间到期后，终端自动退出连接态，ap同时删除该终端连接态信息。正常流程是终端接入后发完上行数据，ap再开始发送下行数据，显然，这段时间不能太短，否则会底层自动丢掉终

端的信息，导致下行无法发送成功。默认连接时间是3秒，也就是说ap侧应用层在收到终端接入后，需要在3秒内下发下行数据。

- 返回值  
无
- 参数  
active\_s: 生存周期，单位秒
- 注意  
需要跟AP侧同步设置，否则终端状态会不同步。

## 18. 获取终端连接时间

- 目的  
获取终端接入后保持的时间
- 语法

```
unsigned int uc_wiota_get_active_time(void);
```

- 描述  
同上
- 返回值  
active\_s, 单位秒
- 参数  
无
- 注意  
无

## 19. 设置当前功率(自动/手动切换)

- 目的  
设置固定功率或者自动功率
- 语法

```
void uc_wiota_set_cur_power(signed char power);
```

- 描述  
设置功率值，如果功率值为正常范围值，则设置成该功率，并且关闭自动功率模式；如果功率值为127 (0x7F)，则代表恢复自动功率模式
- 返回值  
无
- 参数  
power, 范围-16 ~ 21db
- 注意  
无

## 20. 设置最大功率

- 目的  
设置最大功率
- 语法

```
void uc_wiota_set_max_power(signed char power);
```

- 描述  
设置最大功率值，在自动功率模式情况下会用到最大功率值
- 返回值  
无
- 参数  
输入power，范围-16 ~ 21db
- 注意  
无

## 21. 开关gating省电模式

- 目的  
开关gating省电模式
- 语法

```
void uc_wiota_set_is_gating(unsigned char is_gating);
```

- 描述  
设置gating开关标志
- 返回值  
无
- 参数  
is\_gating: 0, 关闭gating功能; 1, 打开gating功能
- 注意  
该功能在协议栈开启时才有效，需初始化协议栈之后再打开该功能，关闭协议栈则自动关闭gating功能

## 22. 设置gating省电模式下的中断唤醒源

- 目的  
设置唤醒源
- 语法

```
void uc_wiota_set_gating_event(unsigned char action, unsigned char event_id);
```

- 描述  
设置唤醒源
- 返回值  
无

- 参数  
action: 0, 清除该event\_id唤醒源; 1, 设置该event\_id唤醒源  
event\_id: 对应于中断向量表, 将某一个中断作为唤醒源, 参考代码interrupt\_handle.c
- 注意  
(1) 该接口在协议栈开启时才有效, 需初始化协议栈之后再设置  
(2) 不支持修改和配置event\_id为0/1/23/24/29的唤醒源, 分别为  
RTC/CCE/UART0/UART1/SYSTIMER

## 23. 扫频

- 目的  
扫频, 获取可接入频点的RSSI和SNR, 用于判断接入哪个频点
- 语法

```
void uc_wiota_scan_freq(unsigned char* data, unsigned short len, unsigned int
timeout, uc_recv callback, uc_recv_back_p recv_result);
```

- 描述  
发送扫频频点数据, 等待返回结果, 提供两种模式  
如果回调函数不为NULL, 则非阻塞模式, 扫频结束或者超时会调用callback返回结果  
如果回调函数为NULL, 则为阻塞模式, 扫频结束或者超时该函数才会返回结果
- 返回值  
recv\_result  
结构体:

```
typedef struct {
    u8_t    result;
    u8_t    type; // UC_RECV_DATA_TYPE
    u16_t   data_len;
    u8_t*   data;
}uc_recv_back_t,*uc_recv_back_p;

typedef enum {
    UC_RECV_MSG = 0,
    UC_RECV_BC,
    UC_RECV_OTA,
    UC_RECV_SCAN_RESULT,
    UC_RECV_SYNC_LOST,
}UC_RECV_DATA_TYPE;

typedef void (*uc_recv)(uc_recv_back_p recv_data);
```

其中的data, 内容为uc\_freq\_scan\_result\_t的结构体数组, 其频点个数需要根据len计算得到

- 参数  
data: 需要传输的数据的头指针, 在收到返回结果之前不能释放, 数据内容为uc\_freq\_scan\_req\_t的结构体数组  
len: 数据长度, 由于频点idx为8bit, 所以该len也代表频点个数, 如果len为0并且data为空, 则代表需要全频带扫频  
callback: 回调函数, 非阻塞时处理返回结果  
timeout: 超时时间, 单位ms
- 结构体

```
typedef struct {
    unsigned char freq_idx;
}uc_freq_scan_req_t,*uc_freq_scan_req_p;

typedef struct {
    unsigned char freq_idx;
    signed char snr;
    signed char rssi;
    unsigned char is_synced;
}uc_freq_scan_result_t,*uc_freq_scan_result_p;
```

- 注意  
需要先初始化协议栈，并且配置系统参数，特别是其中的频带信息，再启动协议栈后才能扫频操作，每次扫频只能扫一个频带的频点  
上报结果目前固定为4个，如果能同步的频点不满4个，则会从RSSI最大的频点依次上报，填满4个频点为止。

## 24. 发送数据

- 目的  
发送数据给ap
- 语法

```
UC_OP_RESULT uc_wiota_send_data(unsigned char* data, unsigned short len,
    unsigned short timeout, uc_send callback);
```

- 描述  
发送数据给ap，等待返回结果，提供两种模式  
如果回调函数不为NULL，则非阻塞模式，成功发送数据或者超时会调用callback返回结果  
如果回调函数为NULL，则为阻塞模式，成功发送数据或者超时该函数才会返回结果
- 返回值  
阻塞模式时该返回值有效

```
typedef enum {
    UC_OP_SUCC = 0,
    UC_OP_TIMEOUT,
    UC_OP_FAIL,
}UC_OP_RESULT;
```

- 参数  
data: 需要传输的数据的头指针  
len: 数据长度，数据最长为310字节  
callback: 回调函数，非阻塞时处理返回结果  
timeout: 超时时间，单位ms
- 结构体

```
typedef struct {
    unsigned short result;
}uc_send_back_t,*uc_send_back_p;

typedef void (*uc_send)(uc_send_back_p send_result);
```

- 注意
  - (1) 在收到返回结果之前不能释放data内存，并且需要预留2字节的空间给底层CRC使用，比如数据len为101，则申请data\_buffer大小为103，可参考AT+WIOTASEND的代码实现
  - (2) 不能在callback函数里释放内存
  - (3) 数据最长为310字节，数据超过310将被丢掉。如果应用层需要传超过310字节的数据，建议自己先分包。

## 25. 被动接收数据接口注册

- 目的  
被动接收数据
- 语法

```
void uc_wiota_register_rcv_data(uc_rcv callback);
```

- 描述  
注册一个接收数据的被动回调函数，只需要系统启动后注册一次即可，每当iote收到普通数据或者广播数据时，会调用改回调函数上报数据。
- 返回值  
无
- 参数  
回调函数用于接收数据结果
- 结构体

```
typedef struct {
    u8_t    result;
    u8_t    type; // UC_RECV_DATA_TYPE
    u16_t   data_len;
    u8_t*   data;
}uc_rcv_back_t,*uc_rcv_back_p;

typedef enum {
    UC_RECV_MSG = 0,
    UC_RECV_BC,
    UC_RECV_OTA,
    UC_RECV_SCAN_RESULT,
    UC_RECV_SYNC_LOST,
}UC_RECV_DATA_TYPE;

typedef void (*uc_rcv)(uc_rcv_back_p rcv_data);
```

## 26. 主动接收数据

- 目的  
iote主动向ap申请下行数据
- 语法

```
void uc_wiota_rcv_data_callback(uc_rcv_back_p rcv_result, unsigned short
timeout, uc_rcv callback);
```

- 描述  
发送申请给ap，等待返回数据结果，提供两种模式  
如果回调函数不为NULL，则非阻塞模式，成功收到数据或者超时后会调用callback返回数据和结果  
如果回调函数为NULL，则为阻塞模式，成功收到数据或者超时该函数才会返回数据结果
- 返回值  
recv\_result:阻塞模式时，返回的结果
- 参数  
timeout: 超时时间，单位ms  
callback: 回调函数，非阻塞时处理返回结果
- 结构体  
参见上述接口

## 27. 设置wiota log开关

- 目的  
设置协议层的log开关
- 语法

```
void uc_wiota_log_switch(unsigned char log_type, unsigned char is_open);

typedef enum {
    UC_LOG_UART = 0,
    UC_LOG_SPI,
} UC_LOG_TYPE;
```

- 描述  
开关协议层的log，包括uart和spi两种
- 返回值  
无
- 参数  
log\_type: uart和spi两种  
is\_open: 是否开启该log
- 结构体  
参见上述接口

## 28. wiota统计信息获取

- 目的  
设置协议层的log开关
- 语法

```
unsigned int uc_wiota_get_stats(unsigned char type);
void uc_wiota_get_all_stats(uc_stats_info_p stats_info_ptr);
void uc_wiota_reset_stats(unsigned char type);
```

- 描述  
获取/重置/增加某个/所有统计信息的计数
- 返回值  
uc\_wiota\_get\_stats: 返回对应type的统计计数  
stats\_info\_ptr: 本地统计信息表，用来获取所有统计信息

- 参数  
type: UC\_STATS\_TYPE, 与uc\_stats\_info\_t的参数一一对应  
注意, 在uc\_wiota\_get\_stats中type为0, 则返回无效值0
- 结构体

```
typedef struct {
    unsigned int rach_fail;
    unsigned int active_fail;
    unsigned int ul_succ;
    unsigned int dl_fail;
    unsigned int dl_succ;
    unsigned int bc_fail;
    unsigned int bc_succ;
    unsigned int ul_sm_succ;
    unsigned int ul_sm_total;
}uc_stats_info_t,*uc_stats_info_p;

typedef enum {
    UC_STATS_READ = 0,
    UC_STATS_WRITE,
}UC_STATS_MODE;

typedef enum {
    UC_STATS_TYPE_ALL = 0,
    UC_STATS_RACH_FAIL,
    UC_STATS_ACTIVE_FAIL,
    UC_STATS_UL_SUCC,
    UC_STATS_DL_FAIL,
    UC_STATS_DL_SUCC,
    UC_STATS_BC_FAIL,
    UC_STATS_BC_SUCC,
    UC_STATS_UL_SM_SUCC,
    UC_STATS_UL_SM_TOTAL,
    UC_STATS_TYPE_MAX,
}UC_STATS_TYPE;
```

## 29. 设置CRC校验开关

- 目的  
设置CRC校验开关
- 语法

```
void uc_wiota_set_crc(unsigned short crc_limit);
```

- 描述  
开关协议层的CRC, 并设置校验长度的标准  
如果crc\_limit为0, 表示关闭CRC校验功能  
如果crc\_limit大于0, 表示数据长度大于等于crc\_limit时, 才打开CRC校验功能, 所以crc\_limit设置为1, 则可表示任意长度的数据均加CRC
- 返回值  
无
- 参数  
crc\_limit: 校验长度限制



- 注意

终端和AP的crc\_limit设置需要一致!