Espressif IoT SDK 编程手册

版本信息

日期	版本	撰写人	审核人	修改说明
2013.12.25	0.1	巫建刚		初稿
2013.12.25	0.1.1	刘晗		修订 json 处理 API
2014.1.15	0.2	巫建刚		增加、修改部分接口配置函数
2014.1.29	0.3	刘晗		增加 client/server 接口函数
2014.3.20	0.4	巫建刚/刘晗		1.增双 uart 接口； 2.增加 i2c master 接口； 3.修改 client/server 接口函数； 4.增加加密接口； 5.增加 upgrade 接口
2014.4.17	0.5	巫建刚/刘晗		1.修改 espconn 接口； 2.增加 gpio 接口 api 说明； 3.增加其他说明；
2014.5.14	0.6	巫建刚		增加若干 API

目录

1. 前言

基于 ESP8266 物联网平台的 SDK 为用户提供了一个简单、快速、高效开发物联网产品的软件平台。

本文旨在介绍该 SDK 的基本框架，以及相关的 API 接口。主要的阅读对象为需要在 ESP8266 物联网平台进行软件开发的嵌入式软件开发人员。

2. 软件框架

为了让用户不用关心底层网络，如 WIFI、TCP/IP 等的具体实现，仅专注于物联网应用的开发，SDK 为用户提供了一套数据接收、发送函数接口，用户只需利用相应接口即可完成网络数据的收发。

ESP8266 物联网平台的所有网络功能均在库中实现，对用户不透明，用户初始化功能在 user_main.c 文件中实现。

函数 void usre_init(void)的作用是给用户提供一个初始化接口，用户可在该函数内增加硬件初始化、网络参数设置、定时器初始化等功能。

SDK 中提供了对 json 包的处理 API，用户也可以采用自定义数据包格式，自行对数据进行处理。

3. SDK 提供的 API 接口

3.1. 定时器接口

3.1.1. os_timer_arm

功能：初始化定时器

函数定义：

Void os_timer_arm(ETSTimer *ptimer,uint32_t milliseconds, bool repeat_flag)

输入参数：

ETSTimer*ptimer——定时器结构

uint32_t milliseconds——定时时间，单位毫秒

bool repeat_flag——该定时是否重复

返回：

无

3.1.2. os_timer_disarm

功能：取消定时器定时

函数定义：

Void os_timer_arm(ETSTimer *ptimer)

输入参数：

ETSTimer *ptimer——定时器结构

返回：

无

3.1.3. os_timer_setfn

功能：设置定时器回调函数

函数定义：

Void os_timer_setfn(ETSTimer *ptimer, ETSTimerFunc *pfunction, void *parg)

输入参数：

ETSTimer *ptimer——定时器结构

TESTimerFunc *pfunction——定时器回调函数

void*parg——回调函数参数

返回：

无

3.1.4. os_timer_done

功能：关闭定时器

函数定义：

Void os_timer_done(ETSTimer *ptimer)

输入参数：

ETSTimer *ptimer——定时器结构

返回：

无

3.2. 底层用户接口

3.2.1. system_restore

功能：恢复出厂设置

函数定义：

void system_restore(void)

输入参数：

无

返回：

无

3.2.2. system_restart

功能：重启

函数定义：

void system_restart(void)

输入参数：

无

返回：

无

3.2.3. system_upgrade_init

功能：upgrade 前初始化

函数定义：

Void system_upgrade_init (uint8 bin)

输入参数：

Uint8 bin——bin 所对应的序号；

UPGRADE_FW_BIN1 代表 eagle.app.v6.flash.bin

UPGRADE_FW_BIN2 代表 eagle.app.v6.irom0text.bin

返回：

无

3.2.4. system_upgrade_deinit

功能：upgrade 完成后，释放部分参数

函数定义：

Void system_upgrade_deinit (uint8 bin)

输入参数：

Uint8 bin——bin 所对应的序号；

UPGRADE_FW_BIN1 代表 eagle.app.v6.flash.bin

UPGRADE_FW_BIN2 代表 eagle.app.v6.irom0text.bin

返回：

无

3.2.5. system_upgrade_flag_check

功能：获取对应 bin 下载状态

函数定义：

Uint8 system_upgrade_flag_check(uint8 bin)

输入参数：

Uint8 bin——bin 所对应的序号；

UPGRADE_FW_BIN1 代表 eagle.app.v6.flash.bin

UPGRADE_FW_BIN2 代表 eagle.app.v6.irom0text.bin

返回：

Uint8——升级状态

UPGRADE_FLAG_IDLE	0x00
UPGRADE_FLAG_START	0x01
UPGRADE_FLAG_FINISH	0x02

3.2.6. system_upgrade_flag_set

功能：设置对应 bin 下载状态

函数定义：

Void system_upgrade_flag_set(uint8 bin, uint8 flag)

输入参数：

Uint8 bin——bin 所对应的序号；

UPGRADE_FW_BIN1 代表 eagle.app.v6.flash.bin

UPGRADE_FW_BIN2 代表 eagle.app.v6.irom0text.bin

Uint8 flag——状态

UPGRADE_FLAG_IDLE	0x00
UPGRADE_FLAG_START	0x01
UPGRADE_FLAG_FINISH	0x02

返回：

无

3.2.7. system_upgrade

功能：将对应 bin 的升级数据写入 SPI flash

函数定义：

Void system_upgrade(uint8 bin, uint8 *data, uint16 len)

输入参数：

Uint8 bin——bin 所对应的序号；

UPGRADE_FW_BIN1 代表 eagle.app.v6.flash.bin

UPGRADE_FW_BIN2 代表 eagle.app.v6.irom0text.bin

Uint8 *data——待写入数据

Uint8 len——待写入数据长度

返回：

无

3.2.8. system_timer_reinit

功能：当需要使用 us 级 timer 时，需要重新初始化 timer，注意需要同时定义 USE_US_TIMER

函数定义：

Void system_timer_reinit (void)

输入参数：

无

返回：

无

3.2.9. system_get_time

功能：获取系统上电开始运行的时间，单位为 us

函数定义：

Uint32 system_get_time (void)

输入参数：

无

返回：

Uint32 当前时间

3.2.10. load_user_param

功能：从 Flash 中载入用户参数

函数定义：

Bool load_user_param(uint16 sec, void* param, uint16 size)

输入参数：

uint16 sec——参数所在扇区，当前用户可用扇区为 0~1，每扇区 4KB

void*param——参数数据指针

uint16 size——参数长度

返回：

True or false

3.2.11. save_user_param

功能：将用户参数存到 Flash

函数定义：

Bool save_user_param(uint16 sec, void* param, uint16 size)

输入参数：

uint16 sec——参数所在扇区，当前用户可用扇区为 0~1，每扇区 4KB

void*param——参数数据指针

uint16 size——参数长度

返回：

True or false

3.2.12. restore_user_param

功能：恢复用户参数，主要是擦除

函数定义：

Bool restore_user_param(uint16 sec)

输入参数：

uint16 sec——参数所在扇区，当前用户可用扇区为 0~1，每扇区 4KB

返回：

True or false

3.2.13. wifi_get_opmode

功能：获取 wifi 工作模式

函数定义：

uint8 wifi_get_opmode (void)

输入参数：

无

返回：

wifi 工作模式，其中 0x01 时为 STATION_MODE，0x02 时为 SOFTAP_MODE，0x03 时为 STATIONAP_MODE。

3.2.14. wifi_set_opmode

功能：设置 wifi 工作模式为 STATION、SOFTAP、STATION+SOFTAP

函数定义：

Void wifi_set_opmode (uint8 opmode)

输入参数：

uint8opmode——wifi 工作模式，其中 STATION_MODE 为 0x01，SOFTAP_MODE 为 0x02，STATIONAP_MODE 为 0x03。

返回：

无

3.2.15. wifi_station_get_config

功能：获取 wifi 的 station 接口参数

函数定义：

Void wifi_station_get_config (struct station_config *config)

输入参数：

struct station_config *config——wifi 的 station 接口参数指针

返回：

无

3.2.16. wifi_station_set_config

功能：设置 wifi 的 station 接口参数

函数定义：

Void wifi_station_set_config (struct station_config *config)

输入参数：

struct station_config *config——wifi 的 station 接口参数指针

返回：

无

3.2.17. wifi_station_connect

功能：wifi 的 station 接口连接所配置的路由

函数定义：

Void wifi_station_connect(void)

输入参数：

无

返回：

无

3.2.18. wifi_station_disconnect

功能：wifi 的 station 接口断开所连接的路由

函数定义：

Void wifi_station_disconnect(void)

输入参数：

无

返回：

无

3.2.19. wifi_station_scan

功能：获取 AP 热点信息

函数定义：

void wifi_station_scan (scan_done_cb_t cb);

输入参数：

scan_done_cb_t cb——获取 AP 热点信息回调 function

返回：

无

3.2.20. scan_done_cb_t

功能：scan 回调 function

函数定义：

void scan_done_cb_t (void *arg, STATUS status);

输入参数：

void *arg——获取的 AP 热点信息入口参数

STATUS status——获取结果

返回：

无

3.2.21. wifi_softap_get_config

功能：设置 wifi 的 softap 接口参数

函数定义：

void wifi_softap_get_config(struct softap_config *config)

输入参数：

struct softap_config *config——wifi 的 softap 接口参数指针

返回：

无

3.2.22. wifi_softap_set_config

功能：设置 wifi 的 softap 接口参数

函数定义：

void wifi_softap_set_config (struct softap_config *config)

输入参数：

struct softap_config *config——wifi 的 softap 接口参数指针

返回：

无

3.2.23. wifi_softap_set_ssid_hidden

功能：设置 wifi 的 softap 接口是否隐藏 ssid

函数定义：

Void wifi_softap_set_ssid_hidden (bool hidden)

输入参数：

Bool hidden——是否隐藏，true or false

返回：

无

3.2.24. wifi_get_ip_info

功能：获取 wifi 的 station 或 softap 接口 ip 信息

函数定义：

Void wifi_get_ip_info(uint8 if_index, struct ip_info *info)

输入参数：

uint8 if_index——获取 ip 信息的接口，其中 STATION_IF 为 0x00，SOFTAP_IF 为 0x01。

struct ip_info *info——获取的指定接口的 ip 信息指针

返回：

无

3.2.25. wifi_get_macaddr

功能：获取 wifi 的 station 或 softap 接口 ip 信息

函数定义：

Void wifi_get_macaddr(uint8 if_index , uint8 *macaddr)

输入参数：

uint8 if_index——获取 mac 信息的接口，其中 STATION_IF 为 0x00，SOFTAP_IF 为 0x01。

uint8 *macaddr——获取的指定接口的 mac 信息指针

返回：

无

3.3. espconn 接口

3.3.1. 回调接口

3.3.1.1. espconn_regist_sentcb

功能：注册数据发送函数，数据发送成功后回调

函数定义：

void espconn_regist_sentcb(struct espconn *espconn, espconn_sent_callback sent_cb)

输入参数：

struct espconn *espconn——相应连接的控制块结构

espconn_sent_callback sent_cb——注册的回调函数

返回：

无

3.3.1.2. espconn_regist_connectcb

功能：注册连接函数，成功连接时回调

函数定义：

Void espconn_regist_connectcb(struct espconn *espconn, espconn_connect_callback connect_cb)

输入参数：

struct espconn *espconn——相应连接的控制块结构

espconn_connect_callback connect_cb——注册的回调函数

返回：

无

3.3.1.3. espconn_regist_recvcb

功能：注册数据接收函数，收到数据时回调

函数定义：

void espconn_regist_recvcb(struct espconn *espconn, espconn_recv_callback recv_cb)

输入参数：

struct espconn *espconn——相应连接的控制块结构

espconn_connect_callback connect_cb——注册的回调函数

返回：

无

3.3.1.4. espconn_regist_reconcb

功能：注册重连函数，出错重连时回调

函数定义：

void espconn_regist_reconcb(struct espconn *espconn, espconn_connect_callback recon_cb)

输入参数：

struct espconn *espconn——相应连接的控制块结构

espconn_connect_callback connect_cb——注册的回调函数

返回：

无

3.3.1.5. espconn_regist_disconcb

功能：注册断开连接函数，断开连接成功时回调

函数定义：

void espconn_regist_disconcb(struct espconn *espconn, espconn_connect_callback discon_cb)

输入参数：

struct espconn *espconn——相应连接的控制块结构

espconn_connect_callback connect_cb——注册的回调函数

返回：

无

3.3.1.6. espconn_sent_callback

功能：数据发送结束回调

函数定义：

void espconn_sent_callback (void *arg)

输入参数：

void *arg——回调函数参数

返回：

无

3.3.1.7. espconn_recv_callback

功能：接收数据回调函数

函数定义：

void espconn_recv_callback (void *arg, char *pdata, unsigned short len)

输入参数：

void *arg——回调函数参数

char *pdata——接收数据入口参数

unsigned short len——接收数据长度

返回：

无

3.3.1.8. espconn_connect_callback

功能：侦听或连接成功回调

函数定义：

Void espconn_connect_callback (void *arg)

输入参数：

void *arg——回调函数参数

返回：

无

3.3.1.9. espconn_sent

功能：发送数据

函数定义：

void espconn_sent(struct espconn *espconn, uint8 *psent, uint16 length)

输入参数：

struct espconn *espconn——相应连接的控制块结构

uint8 *psent——sent 数据指针

uint16 length——sent 数据长度

返回：

无

3.3.2. 服务器接口

3.3.2.1. espconn_accept

功能：侦听连接

函数定义：

void espconn_accept(struct espconn *espconn)

输入参数：

struct espconn *espconn——相应连接的控制块结构

返回：

无

3.3.3. 客户端接口

3.3.3.1. espconn_port

功能：获取未使用的端口

函数定义：

uint32 espconn_port(void);

输入参数：

无

返回：

uint32——获取的端口号

3.3.3.2. espconn_connect

功能：建立连接

函数定义：

Void espconn_connect(struct espconn *espconn)

输入参数：

struct espconn *espconn——相应连接的控制块结构

返回：

无

3.3.3.3. espconn_disconnect

功能：断开连接

函数定义：

Void espconn_disconnect(struct espconn *espconn);

输入参数：

struct espconn *espconn——相应连接的控制块结构

返回：

无

3.3.3.4. espconn_encry_connect

功能：加密连接

函数定义：

espconn_handle espconn_encry_connect (struct espconn *espconn, espconn_connect_callback connect_callback, espconn_recv_callback recv_callback)

输入参数：

struct espconn *espconn——相应连接的控制块结构

espconn_connect_callback connect_callback——连接回调函数

espconn_recv_callback recv_callback——接收数据回调函数

返回：

espconn_handle——client 句柄

3.3.3.5. espconn_encry_sent

功能：加密发送数据

函数定义：

Void espconn_encry_sent (struct espconn *espconn, uint8 *psent, uint16 length)

输入参数：

struct espconn *espconn——相应连接的控制块结构

uint8 *psent——sent 数据指针

uint16 length——sent 数据长度

返回：

无

3.3.3.6. espconn_encry_disconnect

功能：加密断开连接

函数定义：

Void espconn_encry_disconnect(struct espconn *espconn)

输入参数：

struct espconn *espconn——相应连接的控制块结构

返回：

无

3.4. json API 接口

3.4.1. jsonparse_setup

功能：json 解析初始化

函数定义：

void jsonparse_setup(struct jsonparse_state *state, const char *json,int len)

输入参数：

struct jsonparse_state *state——json 解析指针

const char *json——json 解析字符串

int len——字符串长度

返回：

无

3.4.2. jsonparse_next

功能：解析 json 格式下一个元素

函数定义：

Int jsonparse_next(struct jsonparse_state *state)

输入参数：

struct jsonparse_state *state——json 解析指针

返回：

int——解析结果

3.4.3. jsonparse_copy_value

功能：复制当前解析字符串到指定缓存

函数定义：

Int jsonparse_copy_value(struct jsonparse_state *state, char *str, int size)

输入参数：

struct jsonparse_state *state——json 解析指针

char *str——缓存指针

int size——缓存大小

返回：

int——复制结果

3.4.4. jsonparse_get_value_as_int

功能：解析 json 格式为整形数据

函数定义：

Int jsonparse_get_value_as_int(struct jsonparse_state *state)

输入参数：

struct jsonparse_state *state——json 解析指针

返回：

int——解析数据

3.4.5. jsonparse_get_value_as_long

功能：解析 json 格式为长整形数据

函数定义：

Long jsonparse_get_value_as_long(struct jsonparse_state *state)

输入参数：

struct jsonparse_state *state——json 解析指针

返回：

long——解析数据

3.4.6. jsonparse_get_len

功能：解析 json 格式数据长度

函数定义：

Int jsonparse_get_value_len(struct jsonparse_state *state)

输入参数：

struct jsonparse_state *state——json 解析指针

返回：

int——解析的 json 格式数据长度

3.4.7. jsonparse_get_value_as_type

功能：解析 json 格式数据类型

函数定义：

Int jsonparse_get_value_as_type(struct jsonparse_state *state)

输入参数：

struct jsonparse_state *state——json 解析指针

返回：

int——json 格式数据类型

3.4.8. jsonparse_strcmp_value

功能：比较解析的 json 数据与特定字符串

函数定义：

Int jsonparse_strcmp_value(struct jsonparse_state *state, const char *str)

输入参数：

struct jsonparse_state *state——json 解析指针

const char *str——字符缓存

返回：

int——比较结果

3.4.9. jsontree_set_up

功能：生成 json 格式数据树

函数定义：

void jsontree_setup(struct jsontree_context *js_ctx,

struct jsontree_value *root, int (* putchar)(int))

输入参数：

struct jsontree_context *js_ctx——json 格式树元素指针

struct jsontree_value *root——根树元素指针

int (* putchar)(int)——输入函数

返回：

无

3.4.10. jsontree_reset

功能：设置 json 树

函数定义：

void jsontree_reset(struct jsontree_context *js_ctx)

输入参数：

struct jsontree_context *js_ctx——json 格式树指针

返回：

无

3.4.11. jsontree_path_name

功能：json 树参数获取

函数定义：

const char *jsontree_path_name(const struct jsontree_cotext *js_ctx,int depth)

输入参数：

struct jsontree_context *js_ctx——json 格式树指针

int depth——json 格式树深度

返回：

char*——参数指针

3.4.12. jsontree_write_int

功能：整形数写入 json 树

函数定义：

void jsontree_write_int(const struct jsontree_context *js_ctx, int value)

输入参数：

struct jsontree_context *js_ctx——json 格式树指针

int value——整形值

返回：

无

3.4.13. jsontree_write_int_array

功能：整形数数组写入 json 树

函数定义：

void jsontree_write_int_array(const struct jsontree_context *js_ctx, const int *text, uint32 length)

输入参数：

struct jsontree_context *js_ctx——json 格式树指针

int *text——数组入口地址

uint32 length——数组长度

返回：

无

3.4.14. jsontree_write_string

功能：字符串写入 json 树

函数定义：

void jsontree_write_string(const struct jsontree_context *js_ctx, const char *text)

输入参数：

struct jsontree_context *js_ctx——json 格式树指针

const char* text——字符串指针

返回：

无

3.4.15. jsontree_print_next

功能：json 树深度

函数定义：

int jsontree_print_next(struct jsontree_context *js_ctx)

输入参数：

struct jsontree_context *js_ctx——json 格式树指针

返回：

int——json 树深度

3.4.16. jsontree_find_next

功能：查找 json 树元素

函数定义：

struct jsontree_value *jsontree_find_next(struct jsontree_context *js_ctx, int type)

输入参数：

struct jsontree_context *js_ctx——json 格式树指针

int——类型

返回：

struct jsontree_value *——json 格式树元素指针

4. 数据结构定义

4.1. 定时器结构

```
typedef void ETSTimerFunc(void *timer_arg);


typedef struct _ETSTIMER_ {
struct _ETSTIMER_       *timer_next;
      uint32_t                    timer_expire;
      uint32_t                    timer_period;
     ETSTimerFunc              *timer_func;
     void                         *timer_arg;
} ETSTimer;
```

4.2. wifi 参数

4.2.1. station 配置参数

```
struct station_config {
uint8 ssid[32];
uint8 password[64];
};
```

4.2.2. softap 配置参数

```
typedef enum _auth_mode {
      AUTH_OPEN                 = 0,
      AUTH_WEP,
      AUTH_WPA_PSK,
      AUTH_WPA2_PSK,
```

```
    AUTH_WPA_WPA2_PSK
} AUTH_MODE;
struct softap_config {
uint8 ssid[32];
uint8 password[64];
    uint8 channel;
    uint8 authmode;
};
```

4.2.3. scan 参数

```
struct bss_info {
    STAILQ_ENTRY(bss_info)      next;
    u8 bssid[6];
    u8 ssid[32];
    u8 channel;
    s8 rssi;
    u8 authmode;
};
typedef void (* scan_done_cb_t)(void *arg, STATUS status);
```

4.3. json 相关结构

4.3.1. json 结构

```
struct jsontree_value {
uint8_t type;
};


struct jsontree_pair {
```

```
const char *name;
struct jsontree_value *value;
};

struct jsontree_context {
struct jsontree_value *values[JSONTREE_MAX_DEPTH];
uint16_t index[JSONTREE_MAX_DEPTH];
int (* putchar)(int);
uint8_t depth;
uint8_t path;
int callback_state;
};

struct jsontree_callback {
uint8_t type;
int (* output)(struct jsontree_context *js_ctx);
int (* set)(struct jsontree_context *js_ctx, struct jsonparse_state *parser);
};

struct jsontree_object {
uint8_t type;
uint8_t count;
struct jsontree_pair *pairs;
};

struct jsontree_array {
uint8_t type;
uint8_t count;
struct jsontree_value **values;
```

```
};

struct jsonparse_state {
const char *json;
int pos;
int len;
int depth;
int vstart;
int vlen;
char vtype;
char error;
char stack[JSONPARSE_MAX_DEPTH];
};
```

4.3.2. json 宏定义

```
#define JSONTREE_OBJECT(name, ...)                                          \
static struct jsontree_pair jsontree_pair_##name[] = {__VA_ARGS__};     \
static struct jsontree_object name = {                                      \
    JSON_TYPE_OBJECT,                                   \
sizeof(jsontree_pair_##name)/sizeof(struct jsontree_pair),              \
    jsontree_pair_##name }

#define JSONTREE_PAIR_ARRAY(value) (struct jsontree_value *)(value)
#define JSONTREE_ARRAY(name, ...)                                           \
static struct jsontree_value* jsontree_value_##name[] = {__VA_ARGS__};     \
static struct jsontree_array name = {                                       \
    JSON_TYPE_ARRAY,                                    \
sizeof(jsontree_value_##name)/sizeof(struct jsontree_value*),           \
```

jsontree_value_##name }

4.4. espconn 参数

4.4.1 回调 function

/** callback prototype to inform about events for a espconn */

typedef void (* espconn_recv_callback)(void *arg, char *pdata, unsigned short len);

typedef void (* espconn_callback)(void *arg, char *pdata, unsigned short len);

typedef void (* espconn_connect_callback)(void *arg);

4.4.2 espconn

```
typedef void* espconn_handle;
typedef struct _esp_tcp {
    int client_port;
    int server_port;
    char ipaddr[4];
    espconn_connect_callback connect_callback;
    espconn_connect_callback reconnect_callback;
    espconn_connect_callback disconnect_callback;
} esp_tcp;

typedef struct _esp_udp {
    int _port;
    char ipaddr[4];
} esp_udp;

/** Protocol family and type of the espconn */
enum espconn_type {
    ESPCONN_INVALID     = 0,
```

```
        /* ESPCONN_TCP Group */
        ESPCONN_TCP             = 0x10,
        /* ESPCONN_UDP Group */
        ESPCONN_UDP             = 0x20,
    };

    /** Current state of the espconn. Non-TCP espconn are always in state ESPCONN_NONE! */
    enum espconn_state {
        ESPCONN_NONE,
        ESPCONN_WAIT,
        ESPCONN_LISTEN,
        ESPCONN_CONNECT,
        ESPCONN_WRITE,
        ESPCONN_READ,
        ESPCONN_CLOSE
    };

    /** A espconn descriptor */
    struct espconn {
        /** type of the espconn (TCP, UDP) */
        enum espconn_type type;
        /** current state of the espconn */
        enum espconn_state state;
        union {
            esp_tcp *tcp;
            esp_udp *udp;
        } proto;
        /** A callback function that is informed about events for this espconn */
```

```
    espconn_recv_callback recv_callback;
    espconn_sent_callback sent_callback;
    espconn_handle esp_pcb;
    uint8 *ptrbuf;
    uint16 cntr;
};
```

5. 驱动接口

5.1. GPIO 接口 API

关于 gpio 接口 API 操作的具体应用，可参看 examples\0.switch(WAN+LAN sta+softAP)\user\ user_switch.c。

5.1.1. PIN 脚功能设置宏

- ✓ PIN_PULLUP_DIS(PIN_NAME)

  管脚上拉屏蔽

- ✓ PIN_PULLUP_EN(PIN_NAME)

  管脚上拉使能

- ✓ PIN_PULLDWN_DIS(PIN_NAME)

  管脚下拉屏蔽

- ✓ PIN_PULLDWN_EN(PIN_NAME)

  管脚下拉使能

- ✓ PIN_FUNC_SELECT(PIN_NAME, FUNC)

  管脚功能选择

  例如：PIN_FUNC_SELECT(PERIPHS_IO_MUX_MTDI_U, FUNC_GPIO12);

  选择 MTDI 管脚为复用 GPIO12。

5.1.2. gpio_output_set

功能：设置 gpio 口属性

函数定义：

void gpio_output_set(uint32 set_mask, uint32 clear_mask, uint32 enable_mask, uint32 disable_mask)

输入参数：

uint32 set_mask——设置输出为高的位，对应位为 1，输出高，对应位为 0，不改变状态

uint32 clear_mask——设置输出为低的位，对应位为 1，输出低，对应位为 0，不改变状态

uint32 enable_mask——设置使能输出的位

uint32 disable_mask——设置使能输入的位

返回：

无

例子：

- ✓ 设置 GPIO12 输出高电平，则：gpio_output_set(BIT12, 0, BIT12, 0);
- ✓ 设置 GPIO12 输出低电平，则：gpio_output_set(0, BIT12, BIT12, 0);
- ✓ 设置 GPIO12 输出高电平，GPIO13 输出低电平，则：gpio_output_set(BIT12, BIT13, BIT12|BIT13, 0);
- ✓ 设置 GPIO12 为输入，则 gpio_output_set(0, 0, 0, BIT12);

5.1.3. GPIO 输入输出相关宏

- ✓ GPIO_OUTPUT_SET(gpio_no, bit_value)

  设置 gpio_no 管脚输出 bit_value，同 5.1.2 例子中输出高低电平的功能。
- ✓ GPIO_DIS_OUTPUT(gpio_no)

  设置 gpio_no 管脚为输入，同 5.1.2 例子中输入。
- ✓ GPIO_INPUT_GET(gpio_no)

  获取 gpio_no 管脚的电平状态。

5.1.4. GPIO 中断控制相关宏

- ✓ ETS_GPIO_INTR_ATTACH(func, arg)

  注册 GPIO 中断处理函数。
- ✓ ETS_GPIO_INTR_DISABLE()

  关 GPIO 中断。
- ✓ ETS_GPIO_INTR_ENABLE()

  开 GPIO 中断。

5.1.5. gpio_pin_intr_state_set

功能：设置 gpio 脚中断触发状态

函数定义：

```
void gpio_pin_intr_state_set(uint32 i, GPIO_INT_TYPE intr_state)
```

输入参数：

uint32 i——GPIO 管脚 ID，如需设置 GPIO14，则为 GPIO_ID_PIN(14)；

GPIO_INT_TYPE intr_state——中断触发状态

其中：

```
typedef enum{
    GPIO_PIN_INTR_DISABLE = 0,
    GPIO_PIN_INTR_POSEDGE= 1,
    GPIO_PIN_INTR_NEGEDGE= 2,
    GPIO_PIN_INTR_ANYEGDE=3,
    GPIO_PIN_INTR_LOLEVEL= 4,
    GPIO_PIN_INTR_HILEVEL = 5
}GPIO_INT_TYPE;
```

返回：

无

5.1.6. GPIO 中断处理函数

在 GPIO 中断处理函数内，需要做如下操作来清除响应位的中断状态：

```
uint32 gpio_status;
gpio_status = GPIO_REG_READ(GPIO_STATUS_ADDRESS);
//clear interrupt status
GPIO_REG_WRITE(GPIO_STATUS_W1TC_ADDRESS, gpio_status);
```

5.2. 双 UART 接口 API

默认情况下，UART0 作为系统的 debug 输出接口，当配置为双 UART 时，UART0 作为数据收发接口，UART1 作为 debug 输出接口。

使用时，请确保硬件连接正确。

5.1.1. uart_init

功能：双 uart 模式，两个 uart 波特率初始化

函数定义：

void uart_init(UartBautRate uart0_br, UartBautRate uart1_br)

输入参数：

UartBautRate uart0_br——uart0 波特率

UartBautRate uart1_br——uart1 波特率

其中：

```
typedef enum {
    BIT_RATE_9600      = 9600,
    BIT_RATE_19200     = 19200,
    BIT_RATE_38400     = 38400,
    BIT_RATE_57600     = 57600,
    BIT_RATE_74880     = 74880,
    BIT_RATE_115200 = 115200,
    BIT_RATE_230400 = 230400,
    BIT_RATE_460800 = 460800,
    BIT_RATE_921600 = 921600
} UartBautRate;
```

返回：

无

5.1.2. uart0_tx_buffer

功能：通过 UART0 发送用户自定义数据

函数定义：

Void uart0_tx_buffer(uint8 *buf, uint16 len)

输入参数：

Uint8 *buf——待发送数据

Uint16 len——待发送数据长度

返回：

无

5.1.3. uart0_rx_intr_handler

功能：UART0 中断处理函数，用户可在该函数内添加对接收到数据包的处理。（接收缓冲区大小为 0x100，如果接受数据大于 0x100，请自行处理）

函数定义：

Void uart0_rx_intr_handler(void *para)

输入参数：

Void*para——指向 RcvMsgBuff 结构的指针

返回：

无

5.3. i2c master 接口

5.2.1. i2c_master_gpio_init

功能：i2c master 模式时，对相应 GPIO 口进行设置

函数定义：

Void i2c_master_gpio_init (void)

输入参数：

无

返回：

无

5.2.2. i2c_master_init

功能：初始化 i2c 操作

函数定义：

Void i2c_master_init(void)

输入参数：

无

返回：

无

5.2.3. i2c_master_start

功能：设置 i2c 进入发送状态

函数定义：

Void i2c_master_start(void)

输入参数：

无

返回：

无

5.2.4. i2c_master_stop

功能：设置 i2c 进入停止发送状态

函数定义：

Void i2c_master_stop(void)

输入参数：

无

返回：

无

5.2.5. i2c_master_setAck

功能：设置 i2c ACK

函数定义：

Void i2c_master_setAck (uint8 level)

输入参数：

Uint8 level——ack 0 or 1

返回：

无

5.2.6. i2c_master_getAck

功能：获取 slave 的 ACK

函数定义：

Uint8 i2c_master_getAck (void)

输入参数：

无

返回：

uint8——0 or 1

5.2.7. i2c_master_readByte

功能：从 slave 读取一字节

函数定义：

Uint8 i2c_master_readByte (void)

输入参数：

无

返回：

uint8——读取到的值

5.2.8. i2c_master_writeByte

功能：向 slave 写一字节

函数定义：

Void i2c_master_writeByte (uint8 wrdata)

输入参数：

uint8 wrdata——待写数据

返回：

无

5.4. pwm

当前支持 3 路 PWM，可在 pwm.h 中对采用的 GPIO 口进行配置选择。

5.3.1. pwm_init

功能：pwm 功能初始化，包括 gpio，频率以及占空比

函数定义：

Void pwm_init(uint16 freq, uint8 *duty)

输入参数：

Uint16 freq——pwm 的频率；

uint8 *duty——各路的占空比

返回：

无

5.3.2. pwm_set_duty

功能：对某一路设置占空比

函数定义：

Void pwm_set_duty(uint8 duty, uint8 channel)

输入参数：

uint8 duty——占空比

uint8 channel——某路

返回：

无

5.3.3. pwm_set_freq

功能：设置 pwm 频率

函数定义：

Void pwm_set_freq(uint16 freq)

输入参数：

Uint16 freq——pwm 频率

返回：

无

5.3.4. pwm_get_duty

功能：获取某路的占空比

函数定义：

uint8 pwm_get_duty(uint8 channel)

输入参数：

uint8 channel——待获取占空比的 channel

返回：

uint8——占空比

5.3.5. pwm_get_freq

功能：获取 pwm 频率

函数定义：

Uint16 pwm_get_freq(void)

输入参数：

无

返回：

Uint16——频率

5.3.6. i2c_master_getAck

功能：获取 slave 的 ACK

函数定义：

Uint8 i2c_master_getAck (void)

输入参数：

无

返回：

uint8——0 or 1

5.3.7. i2c_master_readByte

功能：从 slave 读取一字节

函数定义：

Uint8 i2c_master_readByte (void)

输入参数：

无

返回：

uint8——读取到的值

5.3.8. i2c_master_writeByte

功能：向 slave 写一字节

函数定义：

Void i2c_master_writeByte (uint8 wrdata)

输入参数：

uint8 wrdata——待写数据

返回：

6. 无附录

A. ESPCONN 编程

A.1. client 模式

A.1.1. 说明

ESP8266 工作在 Station 模式下，确认已经分配到 IP 地址时，启用 client 连接。

ESP8266 工作在 softap 模式下，确认连接 ESP8266 的设备分配到 ip 地址，启用 client 连接。

A.1.2. 步骤

1) 依据工作协议初始化 espconn 参数。
2) 注册 connect 回调函数。udp 协议可以省略该步骤，注册 recv 回调函数。
3) 调用 espconn_connect 函数建立与 host 的连接。
4) 连接成功后将调用注册的 connect 函数，该函数中根据应用注册相应的回调函数，建议 disconnect 回调函数必须注册。udp 协议省略该步骤。
5) 在 recv 回调函数或 sent 回调函数执行 disconnect 操作时，建议适当延时一定时间，确保底层函数执行结束。

A.2. server 模式

A.2.1. 说明

ESP8266 工作在 Station 模式下，确认已经分配到 IP 地址，启用 server 侦听。
ESP8266 工作在 softap 模式下，启用 server 侦听。

A.2.2. 步骤

1) 依据工作协议初始化 espconn 参数。
2) 注册 connect 回调函数。udp 协议可以省略该步骤，注册 recv 回调函数。
3) 调用 espconn_accept 函数侦听 host 的连接。

连接成功后将调用注册的 connect 函数，该函数中根据应用注册相应的回调函数。udp 协议省略该步骤。