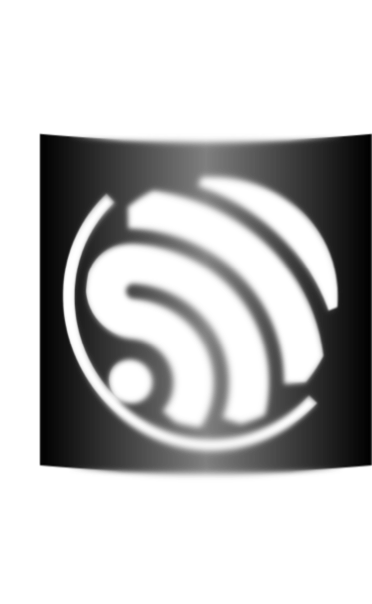


Espressif Systems IOT Team

Copyright (c) 2015

ESP8266 SDK API Guide



**Version 1.0.1**

**Disclaimer and Copyright Notice**

Information in this document, including URL references, is subject to change without notice.

THIS DOCUMENT IS PROVIDED AS IS WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR

SAMPLE. All liability, including liability for infringement of any proprietary rights, relating to use of information in this document is disclaimed. No licenses express or implied, by estoppel or otherwise, to any intellectual property rights are granted herein.

The Wi-Fi Alliance Member Logo is a trademark of the Wi-Fi Alliance.

All trade names, trademarks and registered trademarks mentioned in this document are property of their respective owners, and are hereby acknowledged.

Copyright © 2015 Espressif Systems Inc. All rights reserved.



Table of Content

1. [Preambles 11](#_TOC_250245)
2. [Overview 11](#_TOC_250244)
3. [Application Programming Interface (APIs) 13](#_TOC_250243)
   1. [Timer 13](#_TOC_250242)
      1. [os\_timer\_arm 13](#_TOC_250241)
      2. [os\_timer\_disarm 13](#_TOC_250240)
      3. [os\_timer\_setfn 14](#_TOC_250239)
   2. [System APIs 14](#_TOC_250238)
      1. [system\_restore 14](#_TOC_250237)
      2. [system\_restart 15](#_TOC_250236)
      3. [system\_timer\_reinit 15](#_TOC_250235)
      4. [system\_init\_done\_cb 15](#_TOC_250234)
      5. [system\_get\_chip\_id 16](#_TOC_250233)
      6. [system\_deep\_sleep 16](#_TOC_250232)
      7. [system\_deep\_sleep\_set\_option 17](#_TOC_250231)
      8. [system\_set\_os\_print 17](#_TOC_250230)
      9. [system\_print\_meminfo 18](#_TOC_250229)
      10. [system\_get\_free\_heap\_size 18](#_TOC_250228)
      11. [system\_os\_task 18](#_TOC_250227)
      12. [system\_os\_post 19](#_TOC_250226)
      13. [system\_get\_time 20](#_TOC_250225)
      14. [system\_get\_rtc\_time 20](#_TOC_250224)
      15. [system\_rtc\_clock\_cali\_proc 20](#_TOC_250223)
      16. [system\_rtc\_mem\_write 21](#_TOC_250222)
      17. [system\_rtc\_mem\_read 22](#_TOC_250221)
      18. [system\_uart\_swap 22](#_TOC_250220)
      19. [system\_get\_boot\_version 23](#_TOC_250219)
      20. [system\_get\_userbin\_addr 23](#_TOC_250218)
      21. [system\_get\_boot\_mode 23](#_TOC_250217)
      22. [system\_restart\_enhance 24](#_TOC_250216)
      23. [system\_update\_cpu\_freq 24](#_TOC_250215)
      24. [system\_get\_cpu\_freq 24](#_TOC_250214)
   3. [SPI Flash Related APIs 25](#_TOC_250213)
      1. [spi\_flash\_get\_id 25](#_TOC_250212)
      2. [spi\_flash\_erase\_sector 25](#_TOC_250211)
      3. [spi\_flash\_write 26](#_TOC_250210)
      4. [spi\_flash\_read 26](#_TOC_250209)
   4. [WIFI Related APIs 28](#_TOC_250208)
      1. [wifi\_get\_opmode 28](#_TOC_250207)
      2. [wifi\_get\_opmode\_default 28](#_TOC_250206)
      3. [wifi\_set\_opmode 28](#_TOC_250205)
      4. [wifi\_set\_opmode\_current 29](#_TOC_250204)
      5. [wifi\_station\_get\_config 29](#_TOC_250203)
      6. [wifi\_station\_get\_config\_default 30](#_TOC_250202)
      7. [wifi\_station\_set\_config 30](#_TOC_250201)
      8. [wifi\_station\_set\_config\_current 30](#_TOC_250200)
      9. [wifi\_station\_connect 31](#_TOC_250199)
      10. [wifi\_station\_disconnect 31](#_TOC_250198)
      11. [wifi\_station\_get\_connect\_status 32](#_TOC_250197)
      12. [wifi\_station\_scan 32](#_TOC_250196)
      13. [scan\_done\_cb\_t 33](#_TOC_250195)
      14. [wifi\_station\_ap\_number\_set 33](#_TOC_250194)
      15. [wifi\_station\_get\_ap\_info 34](#_TOC_250193)
      16. [wifi\_station\_ap\_change 34](#_TOC_250192)
      17. [wifi\_station\_get\_current\_ap\_id 34](#_TOC_250191)
      18. [wifi\_station\_get\_auto\_connect 35](#_TOC_250190)
      19. [wifi\_station\_set\_auto\_connect 35](#_TOC_250189)
      20. [wifi\_station\_dhcpc\_start 36](#_TOC_250188)
      21. [wifi\_station\_dhcpc\_stop 36](#_TOC_250187)
      22. [wifi\_station\_dhcpc\_status 36](#_TOC_250186)
      23. [wifi\_station\_set\_reconnect\_policy 37](#_TOC_250185)
      24. [wifi\_softap\_get\_config 37](#_TOC_250184)
      25. [wifi\_softap\_get\_config\_default 38](#_TOC_250183)
      26. [wifi\_softap\_set\_config 38](#_TOC_250182)
      27. [wifi\_softap\_set\_config\_current 38](#_TOC_250181)
      28. [wifi\_softap\_get\_station\_info 39](#_TOC_250180)
      29. [wifi\_softap\_free\_station\_info 39](#_TOC_250179)
      30. [wifi\_softap\_dhcps\_start 40](#_TOC_250178)
      31. [wifi\_softap\_dhcps\_stop 40](#_TOC_250177)
      32. [wifi\_softap\_set\_dhcps\_lease 41](#_TOC_250176)
      33. [wifi\_softap\_dhcps\_status 42](#_TOC_250175)
      34. [wifi\_softap\_dhcps\_set\_offer\_option 42](#_TOC_250174)
      35. [wifi\_set\_phy\_mode 43](#_TOC_250173)
      36. [wifi\_get\_phy\_mode 44](#_TOC_250172)
      37. [wifi\_get\_ip\_info 44](#_TOC_250171)
      38. [wifi\_set\_ip\_info 44](#_TOC_250170)
      39. [wifi\_set\_macaddr 45](#_TOC_250169)
      40. [wifi\_get\_macaddr 46](#_TOC_250168)
      41. [wifi\_set\_sleep\_type 46](#_TOC_250167)
      42. [wifi\_get\_sleep\_type 47](#_TOC_250166)
      43. [wifi\_status\_led\_install 47](#_TOC_250165)
      44. [wifi\_status\_led\_uninstall 48](#_TOC_250164)
      45. [wifi\_set\_broadcast\_if 48](#_TOC_250163)
      46. [wifi\_get\_broadcast \_if 49](#_TOC_250162)
      47. [wifi\_set\_event\_handler\_cb 49](#_TOC_250161)
   5. [Upgrade (FOTA) APIs 52](#_TOC_250160)
      1. [system\_upgrade\_userbin\_check 52](#_TOC_250159)
      2. [system\_upgrade\_flag\_set 52](#_TOC_250158)
      3. [system\_upgrade\_flag\_check 52](#_TOC_250157)
      4. [system\_upgrade\_start 53](#_TOC_250156)
      5. [system\_upgrade\_reboot 53](#_TOC_250155)
   6. Sniﬀer Related APIs 54
      1. [wifi\_promiscuous\_enable 54](#_TOC_250154)
      2. [wifi\_promiscuous\_set\_mac 54](#_TOC_250153)
      3. [wifi\_set\_promiscuous\_rx\_cb 55](#_TOC_250152)
      4. [wifi\_get\_channel 55](#_TOC_250151)
      5. [wifi\_set\_channel 55](#_TOC_250150)
   7. [smart config APIs 56](#_TOC_250149)
      1. [smartconfig\_start 56](#_TOC_250148)
      2. [smartconfig\_stop 57](#_TOC_250147)
      3. [get\_smartconfig\_status 57](#_TOC_250146)
   8. [SNTP APIs 58](#_TOC_250145)
      1. [sntp\_setserver 58](#_TOC_250144)
      2. [sntp\_getserver 58](#_TOC_250143)
      3. [sntp\_setservername 59](#_TOC_250142)
      4. [sntp\_getservername 59](#_TOC_250141)
      5. [sntp\_init 59](#_TOC_250140)
      6. [sntp\_stop 60](#_TOC_250139)
      7. [sntp\_get\_current\_timestamp 60](#_TOC_250138)
      8. [sntp\_get\_real\_time 60](#_TOC_250137)
      9. [SNTP Example 60](#_TOC_250136)
4. [TCP/UDP APIs 62](#_TOC_250135)
   1. [Generic TCP/UDP APIs 62](#_TOC_250134)
      1. [espconn\_delete 62](#_TOC_250133)
      2. [espconn\_gethostbyname 62](#_TOC_250132)
      3. [espconn\_port 63](#_TOC_250131)
      4. [espconn\_regist\_sentcb 64](#_TOC_250130)
      5. [espconn\_regist\_recvcb 64](#_TOC_250129)
      6. [espconn\_sent\_callback 65](#_TOC_250128)
      7. [espconn\_recv\_callback 65](#_TOC_250127)
      8. [espconn\_sent 65](#_TOC_250126)
   2. [TCP APIs 66](#_TOC_250125)
      1. [espconn\_accept 66](#_TOC_250124)
      2. [espconn\_secure\_accept 66](#_TOC_250123)
      3. [espconn\_regist\_time 67](#_TOC_250122)
      4. [espconn\_get\_connection\_info 68](#_TOC_250121)
      5. [espconn\_connect 68](#_TOC_250120)
      6. [espconn\_connect\_callback 69](#_TOC_250119)
      7. [espconn\_set\_opt 69](#_TOC_250118)
      8. [espconn\_clear\_opt 70](#_TOC_250117)
      9. [espconn\_set\_keepalive 70](#_TOC_250116)
      10. [espconn\_get\_keepalive 71](#_TOC_250115)
      11. [espconn\_disconnect 72](#_TOC_250114)
      12. [espconn\_regist\_connectcb 73](#_TOC_250113)
      13. [espconn\_regist\_reconcb 73](#_TOC_250112)
      14. [espconn\_regist\_disconcb 74](#_TOC_250111)
      15. [espconn\_regist\_write\_finish 74](#_TOC_250110)
      16. [espconn\_secure\_set\_size 74](#_TOC_250109)
      17. [espconn\_secure\_get\_size 75](#_TOC_250108)
      18. [espconn\_secure\_connect 75](#_TOC_250107)
      19. [espconn\_secure\_sent 76](#_TOC_250106)
      20. [espconn\_secure\_disconnect 76](#_TOC_250105)
      21. [espconn\_tcp\_get\_max\_con 77](#_TOC_250104)
      22. [espconn\_tcp\_set\_max\_con 77](#_TOC_250103)
      23. [espconn\_tcp\_get\_max\_con\_allow 77](#_TOC_250102)
      24. [espconn\_tcp\_set\_max\_con\_allow 78](#_TOC_250101)
      25. [espconn\_recv\_hold 78](#_TOC_250100)
      26. [espconn\_recv\_unhold 79](#_TOC_250099)
   3. [UDP APIs 79](#_TOC_250098)
      1. [espconn\_create 79](#_TOC_250097)
      2. [espconn\_igmp\_join 79](#_TOC_250096)
      3. [espconn\_igmp\_leave 80](#_TOC_250095)
   4. [mDNS APIs 80](#_TOC_250094)
      1. [espconn\_mdns\_init 80](#_TOC_250093)
      2. [espconn\_mdns\_close 81](#_TOC_250092)
      3. [espconn\_mdns\_server\_register 81](#_TOC_250091)
      4. [espconn\_mdns\_server\_unregister 82](#_TOC_250090)
      5. [espconn\_mdns\_get\_servername 82](#_TOC_250089)
      6. [espconn\_mdns\_set\_servername 82](#_TOC_250088)
      7. [espconn\_mdns\_set\_hostname 82](#_TOC_250087)
      8. [espconn\_mdns\_get\_hostname 83](#_TOC_250086)
      9. [espconn\_mdns\_disable 83](#_TOC_250085)
      10. [espconn\_mdns\_enable 83](#_TOC_250084)
5. [Application Related 85](#_TOC_250083)
   1. [AT APIs 85](#_TOC_250082)
      1. [at\_response\_ok 85](#_TOC_250081)
      2. [at\_response\_error 85](#_TOC_250080)
      3. [at\_cmd\_array\_regist 85](#_TOC_250079)
      4. [at\_get\_next\_int\_dec 86](#_TOC_250078)
      5. [at\_data\_str\_copy 86](#_TOC_250077)
      6. [at\_init 87](#_TOC_250076)
      7. [at\_port\_print 87](#_TOC_250075)
      8. [at\_set\_custom\_info 87](#_TOC_250074)
      9. [at\_enter\_special\_state 88](#_TOC_250073)
      10. [at\_leave\_special\_state 88](#_TOC_250072)
      11. [at\_get\_version 88](#_TOC_250071)
      12. [at\_register\_uart\_rx\_intr 89](#_TOC_250070)
   2. [Related JSON APIs 90](#_TOC_250069)
      1. [jsonparse\_setup 90](#_TOC_250068)
      2. [jsonparse\_next 90](#_TOC_250067)
      3. [jsonparse\_copy\_value 90](#_TOC_250066)
      4. [jsonparse\_get\_value\_as\_int 91](#_TOC_250065)
      5. [jsonparse\_get\_value\_as\_long 91](#_TOC_250064)
      6. [jsonparse\_get\_len 91](#_TOC_250063)
      7. [jsonparse\_get\_value\_as\_type 92](#_TOC_250062)
      8. [jsonparse\_strcmp\_value 92](#_TOC_250061)
      9. [jsontree\_set\_up 92](#_TOC_250060)
      10. [jsontree\_reset 93](#_TOC_250059)
      11. [jsontree\_path\_name 93](#_TOC_250058)
      12. [jsontree\_write\_int 94](#_TOC_250057)
      13. [jsontree\_write\_int\_array 94](#_TOC_250056)
      14. [jsontree\_write\_string 94](#_TOC_250055)
      15. [jsontree\_print\_next 95](#_TOC_250054)
      16. [jsontree\_find\_next 95](#_TOC_250053)
6. [Definition of Structures 96](#_TOC_250052)
   1. [Timer 96](#_TOC_250051)
   2. [WiFi Related Structures 96](#_TOC_250050)
      1. [Station Related 96](#_TOC_250049)
      2. [soft-AP related 96](#_TOC_250048)
      3. [scan related 97](#_TOC_250047)
      4. [WiFi event related structure 97](#_TOC_250046)
      5. [smart config structure 100](#_TOC_250045)
   3. [JSON Related Structure 100](#_TOC_250044)
      1. [json structure 100](#_TOC_250043)
      2. [json macro definition 101](#_TOC_250042)
   4. [espconn parameters 102](#_TOC_250041)
      1. [callback function 102](#_TOC_250040)
      2. [espconn 102](#_TOC_250039)
7. [Peripheral Related Drivers 105](#_TOC_250038)
   1. [GPIO Related APIs 105](#_TOC_250037)
      1. [PIN Related Macros 105](#_TOC_250036)
      2. [gpio\_output\_set 105](#_TOC_250035)
      3. [GPIO input and output macro 106](#_TOC_250034)
      4. [GPIO interrupt 106](#_TOC_250033)
      5. [gpio\_pin\_intr\_state\_set 106](#_TOC_250032)
      6. [GPIO Interrupt Handler 107](#_TOC_250031)
   2. [UART Related APIs 107](#_TOC_250030)
      1. [uart\_init 107](#_TOC_250029)
      2. [uart0\_tx\_buffer 108](#_TOC_250028)
      3. [uart0\_rx\_intr\_handler 108](#_TOC_250027)
   3. [I2C Master Related APIs 109](#_TOC_250026)
      1. [i2c\_master\_gpio\_init 109](#_TOC_250025)
      2. [i2c\_master\_init 109](#_TOC_250024)
      3. [i2c\_master\_start 109](#_TOC_250023)
      4. [i2c\_master\_stop 109](#_TOC_250022)
      5. [i2c\_master\_send\_ack 110](#_TOC_250021)
      6. [i2c\_master\_send\_nack 110](#_TOC_250020)
      7. [i2c\_master\_checkAck 110](#_TOC_250019)
      8. [i2c\_master\_readByte 111](#_TOC_250018)
      9. [i2c\_master\_writeByte 111](#_TOC_250017)
   4. [PWM Related 111](#_TOC_250016)
      1. [pwm\_init 111](#_TOC_250015)
      2. [pwm\_start 112](#_TOC_250014)
      3. [pwm\_set\_duty 112](#_TOC_250013)
      4. [pwm\_set\_freq 112](#_TOC_250012)
      5. [pwm\_get\_duty 113](#_TOC_250011)
      6. [pwm\_get\_freq 113](#_TOC_250010)
8. [Appendix 114](#_TOC_250009)
   1. [ESPCONN Programming 114](#_TOC_250008)
      1. [TCP Client Mode 114](#_TOC_250007)
      2. [TCP Server Mode 114](#_TOC_250006)
   2. [RTC APIs Example 115](#_TOC_250005)
   3. Sniﬀer Structure Introduction 116
   4. [ESP8266 ADC & VDD33 119](#_TOC_250004)

[Applications Scenario 1：Measure power voltage of VDD3P3 pin 3 and 4 119](#_TOC_250003)

[Applications Scenario 2：Input Voltage of TOUT pin 6 119](#_TOC_250002)

[Notice 120](#_TOC_250001)

* 1. [ESP8266 soft-AP and station channel configuration 120](#_TOC_250000)

# Preambles

ESP8266 WiFi SoC предлагает полное и автономное сетевое решение Wi-Fi; его можно использовать для размещения приложения или для разгрузки сетевых функций Wi-Fi от другого процессора приложений. Когда ESP8266 размещает приложение, оно загружается непосредственно с внешней флэш-памяти. В него встроен кэш для повышения производительности системы в таких приложениях. В качестве альтернативы, беспроводной доступ в Интернет может быть добавлен в качестве адаптера Wi-Fi к любой конструкции на базе микроконтроллера с простым подключением через интерфейс UART или мостовой интерфейс CPU AHB.

ESP8266EX является одним из самых интегрированных чипов WiFi в отрасли; он объединяет антенные переключатели, радиочастотный балун, усилитель мощности, малошумящий приемный усилитель, фильтры, модули управления питанием, требует минимальной внешней схемы, а все решение, включая входной модуль, спроектировано так, чтобы занимать минимальную площадь печатной платы.

ESP8266EX также интегрирует улучшенную версию 32-разрядного процессора Tensilica серии L106 Diamond со встроенной памятью SRAM, а также функциями Wi-Fi. ESP8266EX часто интегрируется с внешними датчиками и другими устройствами для конкретных приложений через GPIO; Коды для таких приложений приведены в примерах в пакете SDK.

Сложные функции системного уровня включают быстрое переключение контекста спящего режима/бодрствования для энергоэффективной VoIP, адаптивное смещение радиосигнала для работы с низким энергопотреблением, расширенную обработку сигналов, а также функции подавления сигналов и сосуществования радио для подавления помех обычной сотовой связи, Bluetooth, DDR, LVDS, ЖК-дисплея.

SDK на базе платформы ESP8266 IoT предлагает пользователям простой, быстрый и эффективный способ разработки IoT-устройств. В этом руководстве по программированию представлен обзор пакета SDK, а также подробные сведения об API. Он написан для разработчиков встроенного программного обеспечения, чтобы помочь им программировать на ESP8266 платформе IoT.

# Overview

SDK предоставляет набор интерфейсов для функций приема и передачи данных на уровне Wi-Fi и TCP/ IP, чтобы программисты могли сосредоточиться на разработке приложений на высоком уровне. Пользователи могут легко использовать соответствующие интерфейсы для реализации приема и передачи данных.

Все сетевые функции на IoT-платформе ESP8266 реализованы в библиотеке, и не прозрачны для пользователей. Вместо этого пользователи могут инициализировать интерфейс в user\_main.c.

void user\_init(void) является методом по умолчанию. Пользователи могут добавлять в интерфейс такие функции, как инициализация прошивки, настройка сетевых параметров и инициализация таймера.

Пакет SDK предоставляет API для обработки JSON, и пользователи также могут использовать самостоятельно определенные типы данных для их обработки .

Примечание:

* + - * При использовании однопоточного пакета SDK, не относящегося к ОС, любая задача не должна занимать процессор слишком долго;
        + Если задача занимала процессор в течение длительного времени, ESP8266 не может накормить собаку, вызовет сброс сторожевого таймера;
        + Задача не должна занимать процессор более 10 мс, в противном случае может произойти разрыв Wi-Fi соединения.
      * Мы рекомендуем использовать таймер для периодической проверки.
      * При использовании SDK, не относящегося к ОС, не вызывайте функцию, определенную с помощью обработчика прерываний ICACHE\_FLASH\_ATTR in.
      * Мы предлагаем использовать RTOS SDK, RTOS может планировать различные задачи.

# Application Programming Interface (APIs)

## Timer

Timer can be found: /esp\_iot\_sdk/include/osapi.h.

Please be noted that os\_timer\_arm cannot be invoked during interruption.

For the same timer, os\_timer\_arm cannot be invoked repeatedly. os\_timer\_disarm should be invoked first.

os\_timer\_setfn can only be invoked when the timer is not enabled, i.e., before os\_timer\_arm or after os\_timer\_disarm.

Timer callback may not be precisely executed at the right time, it depends on priority.

### os\_timer\_arm

**Function**:

Initialising the timer.

**Prototype**:

void os\_timer\_arm ( ETSTimer \*ptimer, uint32\_t milliseconds, bool repeat\_flag

)

**Parameters**:

ETSTimer \*ptimer : Timer structure

uint32\_t milliseconds : Timing, Unit: millisecond, the maximum value is 6871947ms

bool repeat\_flag : Whether the timer will be invoked repeatedly or not

**Return:**

null

### os\_timer\_disarm

**Function**:

Disarm timer

**Prototype**:

void os\_timer\_disarm (ETSTimer \*ptimer)

**Parameters**:

ETSTimer \*ptimer : Timer structure

**Return**:

null

### os\_timer\_setfn

**Function**:

Set timer callback function

**Prototype**:

void os\_timer\_setfn( ETSTimer \*ptimer, ETSTimerFunc \*pfunction, void \*parg

)

**Parameters**:

ETSTimer \*ptimer : Timer structure

TESTimerFunc \*pfunction : timer callback function void \*parg : callback function parameter

**Return**:

null

## System APIs

### system\_restore

**Function**:

Reset to default settings of following APIs ：wifi\_station\_set\_auto\_connect, wifi\_set\_phy\_mode, wifi\_softap\_set\_config related, wifi\_station\_set\_config related, wifi\_set\_opmode, and APs information recorded by #define AP\_CACHE

**Prototype**:

void system\_restore(void)

**Parameters**: null

**Return**:

null

### system\_restart

**Function**:

Restart

**Prototype**:

void system\_restart(void)

**Parameters**: null

**Return**:

null

### system\_timer\_reinit

**Function**:

Reinitiate the timer when you need to use microsecond timer

**Notes**:

1. Define USE\_US\_TIMER;
2. Put system\_timer\_reinit at the beginning of user\_init , in the first sentence.

**Prototype**:

void system\_timer\_reinit (void)

**Parameters**: null

**Return:**

null

### system\_init\_done\_cb

**Function**:

Call this API in user\_init to register a system-init-done callback.

**Note**:

wifi\_station\_scan has to be called after system init done and station enable.

**Prototype**:

void system\_init\_done\_cb(init\_done\_cb\_t cb)

**Parameter**:

init\_done\_cb\_t cb : system-init-done callback

**Return**:

null

**Example**:

void to\_scan(void) { wifi\_station\_scan(null,scan\_done); } void user\_init(void) {

wifi\_set\_opmode(STATION\_MODE); system\_init\_done\_cb(to\_scan);

}

### system\_get\_chip\_id

**Function**:

Get chip ID

**Prototype**:

uint32 system\_get\_chip\_id (void)

**Parameters**: null

**Return**:

Chip ID

### system\_deep\_sleep

**Function**:

Configures chip for deep-sleep mode. When the device is in deep-sleep, it automatically wakes up periodically; the period is configurable. Upon waking up, the device boots up from user\_init.

**Prototype**:

void system\_deep\_sleep(uint32 time\_in\_us)

P**arameters**:

uint32 time\_in\_us : during the time (us) device is in deep-sleep

**Return**:

null

**Note**:

Hardware has to support deep-sleep wake up (XPD\_DCDC connects to EXT\_RSTB with 0R).

system\_deep\_sleep(0): there is no wake up timer; in order to wakeup, connect a GPIO to pin RST, the chip will wake up by a falling-edge on pin RST

### system\_deep\_sleep\_set\_option

**Function**:

Call this API before system\_deep\_sleep to set what the chip will do when deep-sleep wake up.

**Prototype**:

bool system\_deep\_sleep\_set\_option(uint8 option)

**Parameter**:

uint8 option :

deep\_sleep\_set\_option(0): Radio calibration after deep-sleep wake up depends on init data byte 108.

deep\_sleep\_set\_option(1): Radio calibration is done after deep-sleep wake up; this increases the current consumption.

deep\_sleep\_set\_option(2): No radio calibration after deep-sleep wake up; this reduces the current consumption.

deep\_sleep\_set\_option(4): Disable RF after deep-sleep wake up, just like modem sleep; this has the least current consumption; the device is not able to transmit or receive data after wake up.

**Note**:

Init data refers esp\_init\_data\_default.bin.

**Return**:

true : succeed false : fail

### system\_set\_os\_print

**Function**:

Turn on/off print logFunction

**Prototype:**

void system\_set\_os\_print (uint8 onoff)

**Parameters**:

uint8 onoff

**Note**:

onoff==0: print function off onoff==1: print function on

**Default**:

print function on

**Return**:

null

### system\_print\_meminfo

**Function**:

Print memory information, including data/rodata/bss/heap

**Prototype:**

void system\_print\_meminfo (void)

**Parameters**: null

**Return**:

null

### system\_get\_free\_heap\_size

**Function**:

Get free heap size

**Prototype:**

uint32 system\_get\_free\_heap\_size(void)

**Parameters**: null

**Return**:

uint32 : available heap size

### system\_os\_task

**Function**:

Set up tasks

**Prototype:**

bool system\_os\_task( os\_task\_t task,

uint8 prio, os\_event\_t \*queue, uint8 qlen

)

**Parameters**:

os\_task\_t task : task function

uint8 prio : task priority. 3 priorities are supported: 0/1/2; 0 is the lowest priority.

os\_event\_t \*queue : message queue pointer uint8 qlen : message queue depth

**Return**:

true: succeed false: fail

**Example**:

#define SIG\_RX

0

#define TEST\_QUEUE\_LEN 4 os\_event\_t \*testQueue;

void test\_task (os\_event\_t \*e) { switch (e->sig) {

case SIG\_RX:

os\_printf(sig\_rx %c/n, (char)e->par); break;

default:

break;

}

}

void task\_init(void) {

testQueue=(os\_event\_t \*)os\_malloc(sizeof(os\_event\_t)\*TEST\_QUEUE\_LEN); system\_os\_task(test\_task,USER\_TASK\_PRIO\_0,testQueue,TEST\_QUEUE\_LEN);

}

### system\_os\_post

**Function**: send message to task

**Prototype:**

bool system\_os\_post ( uint8 prio, os\_signal\_t sig, os\_param\_t par

)

**Parameters**:

uint8 prio

: task priority, corresponding to that you set up

os\_signal\_t sig : message type os\_param\_t par : message parameters

**Return**:

true: succeed false: fail

**Referring to the above example**: void task\_post(void) {

system\_os\_post(USER\_TASK\_PRIO\_0, SIG\_RX, ‘a’);

}

**Printout**:

sig\_rx a

### system\_get\_time

**Function**:

Get system time (us).

**Prototype**:

uint32 system\_get\_time(void)

**Parameter**: null

**Return**:

System time in microsecond.

### system\_get\_rtc\_time

**Function**: Get RTC time, as denoted by the number of RTC clock periods.

**Example**:

If system\_get\_rtc\_time returns 10 (it means 10 RTC cycles), and system\_rtc\_clock\_cali\_proc returns 5 (means 5us per RTC cycle), then the real time is 10 x 5 = 50 us.

**Note**:

System time will return to zero because of system\_restart, but RTC still goes on.

**Prototype**:

uint32 system\_get\_rtc\_time(void)

**Parameter**: null

**Return**:

RTC time

### system\_rtc\_clock\_cali\_proc

**Function**:

Get RTC clock period.

**Prototype**:

uint32 system\_rtc\_clock\_cali\_proc(void)

**Parameter**: null

**Return**:

RTC clock period (in us), bit11~ bit0 are decimal.

**Note**:

see RTC demo in Appendix.

### system\_rtc\_mem\_write

**Function**:

During deep sleep, only RTC still working, so maybe we need to save some user data in RTC memory. Only user data area can be used by user.

|<--------system data--------->|<-----------------user data >|

| 256 bytes | 512 bytes |

**Note**:

RTC memory is 4 bytes aligned for read and write operations. Parameter des\_addr means block number(4 bytes per block). So, if we want to save some data at the beginning of user data area, des\_addr will be 256/4 = 64, save\_size will be data length.

**Prototype**:

bool system\_rtc\_mem\_write ( uint32 des\_addr,

void \* src\_addr, uint32 save\_size

)

**Parameter**:

uint32 des\_addr : des\_addr >=64 void \* src\_addr : uint32 save\_size :

destination address (block number) in RTC memory,

data pointer.

data length ( byte)

**Return**:

true: succeed false: fail

### system\_rtc\_mem\_read

**Function**:

Read user data from RTC memory. Only user data area should be accessed by the user.

|<--------system data--------->|<-----------------user data >|

| 256 bytes | 512 bytes |

**Note**:

RTC memory is 4 bytes aligned for read and write operations. Parameter src\_addr means block number(4 bytes per block). So, to read data from the beginning of user data area, src\_addr will be 256/4=64, save\_size will be data length.

**Prototype**:

bool system\_rtc\_mem\_read ( uint32 src\_addr,

void \* des\_addr, uint32 save\_size

)

**Parameter**:

uint32 src\_addr : 64

void \* des\_addr :

uint32 save\_size :

source address (block number) in rtc memory, src\_addr >=

data pointer

data length, byte

**Return**:

true: succeed false: fail

### system\_uart\_swap

**Function**:

UART0 swap. Use MTCK as UART0 RX, MTDO as UART0 TX, so ROM log will not output from this new UART0. We also need to use MTDO (U0CTS) and MTCK (U0RTS) as UART0 in hardware.

**Prototype**:

void system\_uart\_swap (void)

**Parameter**: null

**Return**:

null

### system\_get\_boot\_version

**Function**:

Get version info of boot

**Prototype**:

uint8 system\_get\_boot\_version (void)

**Parameter**: null

**Return**:

Version info of boot.

**Note**:

If boot version >= 3 , you could enable boot enhance mode (refer to system\_restart\_enhance)

### system\_get\_userbin\_addr

**Function**: Get address of the current running user bin (user1.bin or user2.bin).

**Prototype**:

uint32 system\_get\_userbin\_addr (void)

**Parameter**: null

**Return**:

Start address info of the current running user bin.

### system\_get\_boot\_mode

**Function**: Get boot mode.

**Prototype**:

uint8 system\_get\_boot\_mode (void)

**Parameter**: null

**Return**:

#define SYS\_BOOT\_ENHANCE\_MODE 0

#define SYS\_BOOT\_NORMAL\_MODE 1

**Note**:

Enhance boot mode: can load and run FW at any address;

Normal boot mode: can only load and run normal user1.bin (or user2.bin).

### system\_restart\_enhance

**Function**:

Restarts system, and enters enhance boot mode.

**Prototype**:

bool system\_restart\_enhance( uint8 bin\_type,

uint32 bin\_addr

)

**Parameter**:

uint8 bin\_type : type of bin

#define SYS\_BOOT\_NORMAL\_BIN 0 // user1.bin or user2.bin

#define SYS\_BOOT\_TEST\_BIN 1 // can only be Espressif test bin uint32 bin\_addr : start address of bin file

**Return**:

true: succeed false: Fail

**Note**:

SYS\_BOOT\_TEST\_BIN is for factory test during production; you can apply for the test bin from Espressif Systems.

### system\_update\_cpu\_freq

**Function**:

Set CPU frequency. Default is 80MHz.

**Prototype**:

bool system\_update\_cpu\_freq(uint8 freq)

**Parameter**:

uint8 freq : CPU frequency #define SYS\_CPU\_80MHz 80

#define SYS\_CPU\_160MHz 160

**Return**:

true: succeed false: fail

### system\_get\_cpu\_freq

**Function**:

Get CPU frequency.

**Prototype**:

uint8 system\_get\_cpu\_freq(void)

**Parameter**: null

**Return**:

CPU frequency, unit : MHz.

## SPI Flash Related APIs

### spi\_flash\_get\_id

**Function**:

Get ID info of spi flash

**Prototype**:

uint32 spi\_flash\_get\_id (void)

**Parameters**: null

**Return**:

SPI flash ID

### spi\_flash\_erase\_sector

**Function**:

Erase sector in flash

**Note**:

More details in document Espressif IOT Flash RW Operation

**Prototype**:

SpiFlashOpResult spi\_flash\_erase\_sector (uint16 sec)

**Parameters**:

uint16 sec : Sector number, the count starts at sector 0, 4KB per sector.

**Return**:

typedef enum{

SPI\_FLASH\_RESULT\_OK, SPI\_FLASH\_RESULT\_ERR, SPI\_FLASH\_RESULT\_TIMEOUT

} SpiFlashOpResult;

### spi\_flash\_write

**Function**:

Write data to flash. Flash read/write has to be 4-bytes aligned.

**Note**:

More details in document Espressif IOT Flash RW Operation

**Prototype**:

SpiFlashOpResult spi\_flash\_write ( uint32 des\_addr,

uint32 \*src\_addr, uint32 size

)

**Parameters**:

uint32 des\_addr : destination address in flash. uint32 \*src\_addr : source address of the data. uint32 size :length of data

**Return**:

typedef enum{

SPI\_FLASH\_RESULT\_OK, SPI\_FLASH\_RESULT\_ERR, SPI\_FLASH\_RESULT\_TIMEOUT

} SpiFlashOpResult;

### spi\_flash\_read

**Function**:

Read data from flash. Flash read/write has to be 4-bytes aligned.

**Prototype**:

SpiFlashOpResult spi\_flash\_read( uint32 src\_addr,

uint32 \* des\_addr, uint32 size

)

**Parameters**:

uint32 src\_addr: source address in flash

uint32 \*des\_addr: destination address to keep data. uint32 size: length of data

**Return**:

typedef enum { SPI\_FLASH\_RESULT\_OK, SPI\_FLASH\_RESULT\_ERR, SPI\_FLASH\_RESULT\_TIMEOUT

} SpiFlashOpResult;

**Example：**

uint32 value;

uint8 \*addr = (uint8 \*)&value;

spi\_flash\_read(0x3E \* SPI\_FLASH\_SEC\_SIZE, (uint32 \*)addr, 4);

os\_printf("0x3E sec:%02x%02x%02x%02x\r\n", addr[0], addr[1], addr[2], addr[3]);

## WIFI Related APIs

Flash system parameter area is the last 16KB of flash.

### wifi\_get\_opmode

**Function**:

get WiFi current operating mode

**Prototype:**

uint8 wifi\_get\_opmode (void)

**Parameters**: null

**Return**:

WiFi working modes: 0x01: station mode 0x02: soft-AP mode

0x03: station+soft-AP

### wifi\_get\_opmode\_default

**Function**:

get WiFi operating mode that saved in flash

**Prototype:**

uint8 wifi\_get\_opmode\_default (void)

**Parameters**: null

**Return**:

WiFi working modes: 0x01: station mode 0x02: soft-AP mode

0x03: station+soft-AP

### wifi\_set\_opmode

**Function**:

Sets WiFi working mode as station, soft-AP or station+soft-AP, and save it to flash. Default is soft-AP mode.

**Note**:

Versions before esp\_iot\_sdk\_v0.9.2, need to call system\_restart() after this api; after esp\_iot\_sdk\_v0.9.2, need not to restart.

This configuration will be saved in flash system parameter area if changed.

**Prototype:**

bool wifi\_set\_opmode (uint8 opmode)

**Parameters**:

uint8 opmode: WiFi operating modes: 0x01: station mode

0x02: soft-AP mode 0x03: station+soft-AP

**Return**:

true: succeed false: fail

### wifi\_set\_opmode\_current

**Function**:

Sets WiFi working mode as station, soft-AP or station+soft-AP, and won’t save to flash

**Prototype:**

bool wifi\_set\_opmode\_current (uint8 opmode)

**Parameters**:

uint8 opmode: WiFi operating modes: 0x01: station mode

0x02: soft-AP mode 0x03: station+soft-AP

**Return**:

true: succeed false: fail

### wifi\_station\_get\_config

**Function**:

Get WiFi station current configuration

**Prototype:**

bool wifi\_station\_get\_config (struct station\_config \*config)

**Parameters**:

struct station\_config \*config : WiFi station configuration pointer

**Return**:

true: succeed false: fail

### wifi\_station\_get\_config\_default

**Function**:

Get WiFi station configuration that saved in flash

**Prototype:**

bool wifi\_station\_get\_config\_default (struct station\_config \*config)

**Parameters**:

struct station\_config \*config : WiFi station configuration pointer

**Return**:

true: succeed false: fail

### wifi\_station\_set\_config

**Function**:

Set WiFi station configuration, and save it to flash

**Note**:

If wifi\_station\_set\_config is called in user\_init , there is no need to call wifi\_station\_connect after that, ESP8266 will connect to router automatically; otherwise, need wifi\_station\_connect to connect.

In general, station\_config.bssid\_set need to be 0, otherwise it will check bssid which is the MAC address of AP.

This configuration will be saved in flash system parameter area if changed.

**Prototype:**

bool wifi\_station\_set\_config (struct station\_config \*config)

**Parameters**:

struct station\_config \*config: WiFi station configuration pointer

**Return**:

true: succeed false: fail

### wifi\_station\_set\_config\_current

**Function**:

Set WiFi station configuration, won’t save to flash

**Note**:

If wifi\_station\_set\_config\_current is called in user\_init , there is no need to call wifi\_station\_connect after that, ESP8266 will connect to router automatically; otherwise, need wifi\_station\_connect to connect.

In general, station\_config.bssid\_set need to be 0, otherwise it will check bssid which is the MAC address of AP.

**Prototype:**

bool wifi\_station\_set\_config\_current (struct station\_config \*config)

**Parameters**:

struct station\_config \*config: WiFi station configuration pointer

**Return**:

true: succeed false: fail

### wifi\_station\_connect

**Function**:

To connect WiFi station to AP

**Note**:

If ESP8266 has already connected to a router, then we need to call wifi\_station\_disconnect first, before calling wifi\_station\_connect.

**Prototype:**

bool wifi\_station\_connect (void)

**Parameters**: null

**Return**:

true: succeed false: fail

### wifi\_station\_disconnect

**Function**:

Disconnects WiFi station from AP

**Prototype:**

bool wifi\_station\_disconnect (void)

**Parameters**: null

**Return**:

true: succeed false: fail

### wifi\_station\_get\_connect\_status

**Function**:

Get connection status of WiFi station to AP

**Prototype:**

uint8 wifi\_station\_get\_connect\_status (void)

**Parameters**: null

**Return**:

enum{

STATION\_IDLE = 0, STATION\_CONNECTING, STATION\_WRONG\_PASSWORD, STATION\_NO\_AP\_FOUND, STATION\_CONNECT\_FAIL, STATION\_GOT\_IP

};

### wifi\_station\_scan

**Function**:

Scan all available APs

**Note**:

Do not call this API in user\_init. This API need to be called after system initialize done and station enable.

**Prototype:**

bool wifi\_station\_scan (struct scan\_config \*config, scan\_done\_cb\_t cb);

**Structure**:

struct scan\_config {

uint8 \*ssid; // AP’s ssid uint8 \*bssid; // AP’s bssid

uint8 channel; //scan a specific channel

uint8 show\_hidden; //scan APs of which ssid is hidden.

};

**Parameters**:

struct scan\_config \*config: AP config for scan if config==null: scan all APs

if config.ssid==null && config.bssid==null && config.channel!=null: ESP8266 will scan the specific channel.

scan\_done\_cb\_t cb: callback function after scan

**Return**:

true: succeed false: fail

### scan\_done\_cb\_t

**Function**:

Callback function for wifi\_station\_scan

**Prototype:**

void scan\_done\_cb\_t (void \*arg, STATUS status)

**Parameters**:

void \*arg: information of APs that be found, refer to struct bss\_info STATUS status: get status

**Return**:

null

**Example**:

wifi\_station\_scan(&config, scan\_done);

static void ICACHE\_FLASH\_ATTR scan\_done(void \*arg, STATUS status) { if (status == OK) {

struct bss\_info \*bss\_link = (struct bss\_info \*)arg; bss\_link = bss\_link->next.stqe\_next; //ignore first

...

}

}

### wifi\_station\_ap\_number\_set

**Function**:

Sets the number of APs that will be cached for ESP8266 station mode. Whenever ESP8266 station connects to an AP, it keeps caches a record of this AP's SSID and password. The cached ID index starts from 0.

This configuration will be saved in flash system parameter area if changed.

**Prototype**:

bool wifi\_station\_ap\_number\_set (uint8 ap\_number)

**Parameters**:

uint8 ap\_number: the number of APs can be recorded (MAX: 5)

**Return**:

true: succeed false: fail

### wifi\_station\_get\_ap\_info

**Function**:

Get information of APs recorded by ESP8266 station.

**Prototype**:

uint8 wifi\_station\_get\_ap\_info(struct station\_config config[])

**Parameters**:

struct station\_config config[]: information of APs, array size has to be 5.

**Return**:

The number of APs recorded.

**Example**:

struct station\_config config[5];

int i = wifi\_station\_get\_ap\_info(config);

### wifi\_station\_ap\_change

**Function**:

Switch ESP8266 station connection to AP as specified

**Prototype**:

bool wifi\_station\_ap\_change (uint8 new\_ap\_id)

**Parameters**:

uint8 new\_ap\_id : AP’s record id, start counting from 0.

**Return**:

true: succeed false: fail

### wifi\_station\_get\_current\_ap\_id

**Function**:

Get the current record id of AP.

**Prototype**:

uint8 wifi\_station\_get\_current\_ap\_id ();

**Parameter**: null

**Return**:

The index of the AP, which ESP8266 is currently connected to, in the cached AP list.

### wifi\_station\_get\_auto\_connect

**Function**:

Checks if ESP8266 station mode will connect to AP (which is cached) automatically or not when it is powered on.

**Prototype**:

uint8 wifi\_station\_get\_auto\_connect(void)

**Parameter**: null

**Return**:

0: wil not connect to AP automatically; Non-0: will connect to AP automatically.

### wifi\_station\_set\_auto\_connect

**Function**:

Set whether ESP8266 station will connect to AP (which is recorded) automatically or not when power on. Default to enable auto-connect.

**Note**:

Call this API in user\_init, it is effective in this current power on; call it in other place, it will be effective in next power on.

This configuration will be saved in flash system parameter area if changed.

**Prototype**:

bool wifi\_station\_set\_auto\_connect(uint8 set)

**Parameter**:

uint8 set: Automatically connect or not: 0: will not connect automatically 1: to connect automatically

**Return**:

true: succeed false: fail

### wifi\_station\_dhcpc\_start

**Function**:

Enable ESP8266 station DHCP client.

**Note**:

DHCP default enable.

This configuration interacts with static IP API (wifi\_set\_ip\_info): If enable DHCP, static IP will be disabled;

If enable static IP, DHCP will be disabled; This will depend on the last configuration.

**Prototype**:

bool wifi\_station\_dhcpc\_start(void)

**Parameter**: null

**Return**:

true: succeed false: fail

### wifi\_station\_dhcpc\_stop

**Function**:

Disable ESP8266 station DHCP client.

**Note**:

DHCP default enable.

**Prototype**:

bool wifi\_station\_dhcpc\_stop(void)

**Parameter**: null

**Return**:

true: succeed false: fail

### wifi\_station\_dhcpc\_status

**Function**: Get ESP8266 station DHCP client status.

**Prototype**:

enum dhcp\_status wifi\_station\_dhcpc\_status(void)

**Parameter**: null

**Return**:

enum dhcp\_status { DHCP\_STOPPED, DHCP\_STARTED

};

### wifi\_station\_set\_reconnect\_policy

**Function**:

Set whether reconnect or not when ESP8266 station disconnected from AP

**Note：**

We suggest to call this API in user\_init

This API can only be called when ESP8266 station enable.

**Prototype**:

bool wifi\_station\_set\_reconnect\_policy(bool set)

**Parameter**:

bool set - true, enable reconnect；false，disable reconnect

**Return**:

true：succeed false：fail

### wifi\_softap\_get\_config

**Function**:

Get WiFi soft-AP current configuration

**Prototype:**

bool wifi\_softap\_get\_config(struct softap\_config \*config)

**Parameter**:

struct softap\_config \*config : ESP8266 soft-AP config

**Return**:

true: succeed false: fail

### wifi\_softap\_get\_config\_default

**Function**:

Get WiFi soft-AP configuration that saved in flash

**Prototype:**

bool wifi\_softap\_get\_config\_default(struct softap\_config \*config)

**Parameter**:

struct softap\_config \*config : ESP8266 soft-AP config

**Return**:

true: succeed false: fail

### wifi\_softap\_set\_config

**Function**:

Set WiFi soft-AP configuration and save it to flash

**Note：**

1. This configuration will be saved in flash system parameter area if changed.
2. In soft-AP + station mode, ESP8266 soft-AP will adjust its channel configuration to be as same as ESP8266. More details in appendix or BBS <http://bbs.espressif.com/viewtopic.php?f=10&t=324>

**Prototype:**

bool wifi\_softap\_set\_config (struct softap\_config \*config)

**Parameter**:

struct softap\_config \*config : WiFi soft-AP configuration pointer

**Return**:

true: succeed false: fail

### wifi\_softap\_set\_config\_current

**Function**:

Set WiFi soft-AP configuration, won’t save it to flash

**Note:**

In soft-AP + station mode, ESP8266 soft-AP will adjust its channel configuration to be as same as ESP8266. More details in appendix or BBS <http://bbs.espressif.com/viewtopic.php?f=10&t=324>

**Prototype:**

bool wifi\_softap\_set\_config\_current (struct softap\_config \*config)

**Parameter**:

struct softap\_config \*config : WiFi soft-AP configuration pointer

**Return**:

true: succeed false: fail

### wifi\_softap\_get\_station\_info

**Function**:

Get connected station devices under soft-AP mode, including MAC and IP

**Note:**

This API can only be used when soft-AP DHCP enabled.

**Prototype:**

struct station\_info \* wifi\_softap\_get\_station\_info(void)

**Input Parameters**: null

**Return**:

struct station\_info\* : station information structure

### wifi\_softap\_free\_station\_info

**Function**:

Frees the struct station\_info by calling the wifi\_softap\_get\_station\_info function

**Prototype:**

void wifi\_softap\_free\_station\_info(void)

**Input Parameters**: null

**Return**:

null

**Examples 1 (Getting MAC and IP information):**

struct station\_info \* station = wifi\_softap\_get\_station\_info(); struct station\_info \* next\_station;

while(station) {

os\_printf(bssid : MACSTR, ip : IPSTR/n,

MAC2STR(station->bssid), IP2STR(&station->ip)); next\_station = STAILQ\_NEXT(station, next); os\_free(station); // Free it directly

station = next\_station;

}

**Examples 2 (Getting MAC and IP information):**

struct station\_info \* station = wifi\_softap\_get\_station\_info(); while(station){

os\_printf(bssid : MACSTR, ip : IPSTR/n,

MAC2STR(station->bssid), IP2STR(&station->ip)); station = STAILQ\_NEXT(station, next);

}

wifi\_softap\_free\_station\_info(); // Free it by calling functions

### wifi\_softap\_dhcps\_start

**Function**: Enable ESP8266 soft-AP DHCP server.

**Note**:

DHCP default enable.

This configuration interacts with static IP API (wifi\_set\_ip\_info): If enable DHCP, static IP will be disabled;

If enable static IP, DHCP will be disabled; This will depend on the last configuration.

**Prototype**:

bool wifi\_softap\_dhcps\_start(void)

**Parameter**: null

**Return**:

true: succeed false: fail

### wifi\_softap\_dhcps\_stop

**Function**: Disable ESP8266 soft-AP DHCP server.

**Note**: DHCP default enable.

**Prototype**:

bool wifi\_softap\_dhcps\_stop(void)

**Parameter**: null

**Return**:

true: succeed false: fail

### wifi\_softap\_set\_dhcps\_lease

**Function**:

Set the IP range that can be got from ESP8266 soft-AP DHCP server.

**Note**:

This API has to be called during DHCP server disable(wifi\_softap\_dhcps\_stop)

This configuration only take effect on next wifi\_softap\_dhcps\_start, if then wifi\_softap\_dhcps\_stop is called； user needs to call this API to set IP range again if needed，then call wifi\_softap\_dhcps\_start to take effect.

**Prototype**:

bool wifi\_softap\_set\_dhcps\_lease(struct dhcps\_lease \*please)

**Parameter**:

struct dhcps\_lease {

struct ip\_addr start\_ip; struct ip\_addr end\_ip;

};

**Return**:

true: succeed false: fail

**Example:**

void dhcps\_lease\_test(void)

{

struct dhcps\_lease dhcp\_lease;

const char\* start\_ip = “192.168.5.100”; const char\* end\_ip = “192.168.5.105”;

dhcp\_lease.start\_ip.addr = ipaddr\_addr(start\_ip); dhcp\_lease.end\_ip.addr = ipaddr\_addr(end\_ip); wifi\_softap\_set\_dhcps\_lease(&dhcp\_lease);

}

or

void dhcps\_lease\_test(void)

{

struct dhcps\_lease dhcp\_lease; IP4\_ADDR(&dhcp\_lease.start\_ip, 192, 168, 5, 100);

IP4\_ADDR(&dhcp\_lease.end\_ip, 192, 168, 5, 105); wifi\_softap\_set\_dhcps\_lease(&dhcp\_lease);

}

### wifi\_softap\_dhcps\_status

**Function**: Get ESP8266 soft-AP DHCP server status.

**Prototype**:

enum dhcp\_status wifi\_softap\_dhcps\_status(void)

**Parameter**: null

**Return**:

enum dhcp\_status { DHCP\_STOPPED, DHCP\_STARTED

};

### wifi\_softap\_dhcps\_set\_offer\_option

**Function**:

Set ESP8266 soft-AP DHCP server option.

**Structure：**

enum dhcps\_offer\_option{ OFFER\_START = 0x00, OFFER\_ROUTER = 0x01,

OFFER\_END

};

**Prototype**:

bool wifi\_softap\_dhcps\_set\_offer\_option(uint8 level, void\* optarg)

**Parameter**:

uint8 level - OFFER\_ROUTER set router option void\* optarg - default to be enable

bit0, 0 disable router information from ESP8266 softAP DHCP server；

bit0, 1 enable router information from ESP8266 softAP DHCP server；

**Return**:

true : succeed false : fail

**Example:**

uint8 mode = 0； wifi\_softap\_dhcps\_set\_offer\_option(OFFER\_ROUTER, &mode);

### wifi\_set\_phy\_mode

**Fuction**: Set ESP8266 physical mode (802.11b/g/n).

**Note**: ESP8266 soft-AP only support bg.

**Prototype**:

bool wifi\_set\_phy\_mode(enum phy\_mode mode)

**Parameter**:

enum phy\_mode mode : physical mode enum phy\_mode {

PHY\_MODE\_11B = 1,

PHY\_MODE\_11G = 2,

PHY\_MODE\_11N = 3

};

**Return**:

true : succeed false : fail

### wifi\_get\_phy\_mode

**Function**:

Get ESP8266 physical mode (802.11b/g/n)

**Prototype**:

enum phy\_mode wifi\_get\_phy\_mode(void)

**Parameter**: null

**Return**:

enum phy\_mode{ PHY\_MODE\_11B = 1,

PHY\_MODE\_11G = 2,

PHY\_MODE\_11N = 3

};

### wifi\_get\_ip\_info

**Function**:

Get IP info of WiFi station or soft-AP interface

**Prototype:**

bool wifi\_get\_ip\_info( uint8 if\_index, struct ip\_info \*info

)

**Parameters**:

uint8 if\_index : the interface to get IP info: 0x00 for STATION\_IF, 0x01 for SOFTAP\_IF.

struct ip\_info \*info : pointer to get IP info of a certain interface

**Return**:

true: succeed false: fail

### wifi\_set\_ip\_info

**Function**:

Set IP address of ESP8266 station or soft-AP

**Note**:

To set static IP, please disable DHCP first (wifi\_station\_dhcpc\_stop or wifi\_softap\_dhcps\_stop):

If enable static IP, DHCP will be disabled;

If enable DHCP, static IP will be disabled;

**Prototype:**

bool wifi\_set\_ip\_info( uint8 if\_index, struct ip\_info \*info

)

**Prototype**:

uint8 if\_index : set station IP or soft-AP IP #define STATION\_IF 0x00

#define SOFTAP\_IF 0x01 struct ip\_info \*info : IP information

**Example**:

struct ip\_info info; wifi\_station\_dhcpc\_stop(); wifi\_softap\_dhcps\_stop();

IP4\_ADDR(&info.ip, 192, 168, 3, 200);

IP4\_ADDR(&info.gw, 192, 168, 3, 1);

IP4\_ADDR(&info.netmask, 255, 255, 255, 0); wifi\_set\_ip\_info(STATION\_IF, &info);

IP4\_ADDR(&info.ip, 10, 10, 10, 1);

IP4\_ADDR(&info.gw, 10, 10, 10, 1);

IP4\_ADDR(&info.netmask, 255, 255, 255, 0); wifi\_set\_ip\_info(SOFTAP\_IF, &info);

**Return**:

true: succeed false: fail

### wifi\_set\_macaddr

**Function**:

Sets MAC address

**Note**:

Can only be used in user\_init.

**Prototype:**

bool wifi\_set\_macaddr( uint8 if\_index, uint8 \*macaddr

)

**Parameter**:

uint8 if\_index : set station MAC or soft-AP mac #define STATION\_IF 0x00

#define SOFTAP\_IF 0x01 uint8 \*macaddr : MAC address

**Example**:

char sofap\_mac[6] = {0x16, 0x34, 0x56, 0x78, 0x90, 0xab}; char sta\_mac[6] = {0x12, 0x34, 0x56, 0x78, 0x90, 0xab}; wifi\_set\_macaddr(SOFTAP\_IF, sofap\_mac); wifi\_set\_macaddr(STATION\_IF, sta\_mac);

**Return**:

true: succeed false: fail

### wifi\_get\_macaddr

**Function**: get MAC address

**Prototype:**

bool wifi\_get\_macaddr( uint8 if\_index, uint8 \*macaddr

)

**Parameter**:

uint8 if\_index : set station MAC or soft-AP mac #define STATION\_IF 0x00

#define SOFTAP\_IF 0x01 uint8 \*macaddr : MAC address

**Return**:

true: succeed false: fail

### wifi\_set\_sleep\_type

**Function**:

Sets sleep type for power saving. Set NONE\_SLEEP\_T to disable power saving.

**Note**: Default to be Modem sleep.

**Prototype**:

bool wifi\_set\_sleep\_type(enum sleep\_type type)

**Parameters**:

enum sleep\_type type : sleep type

**Return**:

true: succeed false: fail

### wifi\_get\_sleep\_type

**Function**:

Gets sleep type.

**Prototype**:

enum sleep\_type wifi\_get\_sleep\_type(void)

**Parameters**: null

**Return**:

enum sleep\_type { NONE\_SLEEP\_T = 0; LIGHT\_SLEEP\_T, MODEM\_SLEEP\_T

};

### wifi\_status\_led\_install

**Function**:

Installs WiFi status LED

**Prototype:**

void wifi\_status\_led\_install ( uint8 gpio\_id,

uint32 gpio\_name, uint8 gpio\_func

)

**Parameter**:

uint8 gpio\_id : gpio number uint8 gpio\_name : gpio mux name uint8 gpio\_func : gpio function

**Return**:

null

**Example**:

Use GPIO0 as WiFi status LED #define HUMITURE\_WIFI\_LED\_IO\_MUX #define HUMITURE\_WIFI\_LED\_IO\_NUM #define HUMITURE\_WIFI\_LED\_IO\_FUNC

PERIPHS\_IO\_MUX\_GPIO0\_U 0

FUNC\_GPIO0

wifi\_status\_led\_install(HUMITURE\_WIFI\_LED\_IO\_NUM,

HUMITURE\_WIFI\_LED\_IO\_MUX, HUMITURE\_WIFI\_LED\_IO\_FUNC)

### wifi\_status\_led\_uninstall

**Function**: Uninstall WiFi status LED

**Prototype:**

void wifi\_status\_led\_uninstall ()

**Parameter**: null

**Return**:

null

### wifi\_set\_broadcast\_if

**Function**:

Set ESP8266 send UDP broadcast from station interface or soft-AP interface, or both station and soft-AP interfaces. Default to be soft-AP.

**Note：**

If set broadcast interface to be station only，ESP8266 softAP DHCP server will be disable.

**Prototype**:

bool wifi\_set\_broadcast\_if (uint8 interface)

**Parameter**:

uint8 interface : 1:station; 2:soft-AP, 3:station+soft-AP

**Return**:

true: succeed false: fail

### wifi\_get\_broadcast \_if

**Function**:

Get interface which ESP8266 sent UDP broadcast from. This is usually used when you have STA+soft-AP mode to avoid ambiguity.

**Prototype**:

uint8 wifi\_get\_broadcast\_if (void)

**Parameter**: null

**Return**:

1: station

2: soft-AP

3: both station and soft-AP

### wifi\_set\_event\_handler\_cb

**Function**:

Register WiFi event handler

**Prototype**:

void wifi\_set\_event\_handler\_cb(wifi\_event\_handler\_cb\_t cb)

**Parameter**:

wifi\_event\_handler\_cb\_t cb - callback

**返回**:

无

**示例：**

void wifi\_handle\_event\_cb(System\_Event\_t \*evt)

{

os\_printf("event %x\n", evt->event); switch (evt->event) {

case EVENT\_STAMODE\_CONNECTED:

os\_printf("connect to ssid %s, channel %d\n", evt->event\_info.connected.ssid,

evt->event\_info.connected.channel);

break;

case EVENT\_STAMODE\_DISCONNECTED:

os\_printf("disconnect from ssid %s, reason %d\n",

evt->event\_info.disconnected.ssid, evt->event\_info.disconnected.reason);

break;

case EVENT\_STAMODE\_AUTHMODE\_CHANGE:

os\_printf("mode: %d -> %d\n",

evt->event\_info.auth\_change.old\_mode, evt->event\_info.auth\_change.new\_mode);

break;

case EVENT\_STAMODE\_GOT\_IP:

os\_printf("ip:" IPSTR ",mask:" IPSTR ",gw:" IPSTR,

IP2STR(&evt->event\_info.got\_ip.ip), IP2STR(&evt->event\_info.got\_ip.mask), IP2STR(&evt->event\_info.got\_ip.gw));

os\_printf("\n"); break;

case EVENT\_SOFTAPMODE\_STACONNECTED:

os\_printf("station: " MACSTR "join, AID = %d\n", MAC2STR(evt->event\_info.sta\_connected.mac), evt->event\_info.sta\_connected.aid);

break;

case EVENT\_SOFTAPMODE\_STADISCONNECTED:

os\_printf("station: " MACSTR "leave, AID = %d\n", MAC2STR(evt->event\_info.sta\_disconnected.mac), evt->event\_info.sta\_disconnected.aid);

break; default:

break;

}

}

void user\_init(void)

{

// TODO: add your own code here.... wifi\_set\_event\_hander\_cb(wifi\_handle\_event\_cb);

}

## Upgrade (FOTA) APIs

### system\_upgrade\_userbin\_check

**Function**:

Checks user bin

**Prototype:**

uint8 system\_upgrade\_userbin\_check()

**Input Parameters**: none

**Return**:

0x00 : UPGRADE\_FW\_BIN1, i.e. user1.bin

0x01 : UPGRADE\_FW\_BIN2, i.e. user2.bin

### system\_upgrade\_flag\_set

**Function**:

Sets upgrade status flag.

**Note**:

If you using system\_upgrade\_start to upgrade, this API need not be called. If you using spi\_flash\_write to upgrade firmware yourself, this flag need to be set to UPGRADE\_FLAG\_FINISH, then call system\_upgrade\_reboot to reboot to run new firmware.

**Prototype**:

void system\_upgrade\_flag\_set(uint8 flag)

**Parameter**:

uint8 flag:

#define UPGRADE\_FLAG\_IDLE #define UPGRADE\_FLAG\_START #define UPGRADE\_FLAG\_FINISH

0x00

0x01

0x02

**Return**:

null

### system\_upgrade\_flag\_check

**Function**:

Gets upgrade status flag.

**Prototype**:

uint8 system\_upgrade\_flag\_check()

**Parameter:**

null

**Return:**

#define UPGRADE\_FLAG\_IDLE #define UPGRADE\_FLAG\_START #define UPGRADE\_FLAG\_FINISH

0x00

0x01

0x02

### system\_upgrade\_start

**Function**:

Configures parameters and start upgrade

**Prototype:**

bool system\_upgrade\_start (struct upgrade\_server\_info \*server)

**Parameters**:

struct upgrade\_server\_info \*server : server related parameters

**Return:**

true: start upgrade

false: upgrade can’t be started.

### system\_upgrade\_reboot

**Function**: reboot system and use new version

**Prototype:**

void system\_upgrade\_reboot (void)

**Parameters**: none

**Return**:

none

## Sniffer Related APIs

### wifi\_promiscuous\_enable

**Function:**

Enable promiscuous mode for sniffer

**Note：**

1. promiscuous mode can only be enabled in station mode.
2. During promiscuous mode（sniffer），ESP8266 station and soft-AP are disabled.
3. Before enable promiscuous mode，please disable auto-connection function first wifi\_station\_set\_auto\_connect(0)
4. Don’t call any other APIs during sniffer，please call wifi\_promiscuous\_enable(0) first.

**Prototype:**

void wifi\_promiscuous\_enable(uint8 promiscuous)

**Parameter**:

uint8 promiscuous :

0: disable promiscuous;

1: enable promiscuous

**Return**:

null

### wifi\_promiscuous\_set\_mac

**Function**:

Set MAC address filter for sniffer.

**Note**:

This filter only available in the current sniffer phase, if you disable sniffer and then enable sniffer, you need to set filter again if you need it.

**Prototype**:

void wifi\_promiscuous\_set\_mac(const uint8\_t \*address)

**Parameter**:

const uint8\_t \*address : MAC address

**Return**:

null

### wifi\_set\_promiscuous\_rx\_cb

**Function**:

Registers an RX callback function in promiscuous mode, which will be called when data packet is received.

**Prototype:**

void wifi\_set\_promiscuous\_rx\_cb(wifi\_promiscuous\_cb\_t cb)

**Parameter**:

wifi\_promiscuous\_cb\_t cb : callback

**Return:**

null

### wifi\_get\_channel

**Function**:

Get channel number for sniffer functions

**Prototype:**

uint8 wifi\_get\_channel(void)

**Parameters**: null

**Return**:

Channel number

### wifi\_set\_channel

**Function**:

Set channel number for sniffer functions

**Prototype:**

bool wifi\_set\_channel (uint8 channel)

**Parameters**:

uint8 channel : channel number

**Return**:

true: succeed false: fail

## smart config APIs

### smartconfig\_start

#### Function:

Start smart configuration mode, to connect ESP8266 station to AP, by sniffing for special packets from the air, containing SSID and password of desired AP. You need to broadcast the SSID and password (e.g. from mobile device or computer) with the SSID and password encoded.

#### Note:

* + - 1. This api can only be called in station mode.
      2. During smart config，ESP8266 station and soft-AP are disabled.
      3. Can not call smartconfig\_start twice before it finish，please call smartconfig\_stop first.
      4. Don’t call any other APIs during smart config, please call smartconfig\_stop first.

#### Prototype:

bool smartconfig\_start( sc\_type type, sc\_callback\_t cb, uint8 log

)

#### Parameter:

sc\_type type : smart config protocol type: AirKiss or ESP-TOUCH. sc\_callback\_t cb : pointer to struct station\_config; smart config callback; executed when ESP8266 successfully obtains SSID and password of target AP; for parameter of this callback, more information in example.

uint8 log : 1: UART output logs; otherwise: UART only outputs the result.

#### Return:

true: succeed false: fail

**Example**:

void ICACHE\_FLASH\_ATTR

smartconfig\_done(void \*data) {

struct station\_config \*sta\_conf = data; wifi\_station\_set\_config(sta\_conf); wifi\_station\_disconnect(); wifi\_station\_connect(); user\_devicefind\_init(); user\_esp\_platform\_init();

}

smartconfig\_start(SC\_TYPE\_ESPTOUCH,smartconfig\_done);

### smartconfig\_stop

**Function**:

stop smart config, free the buffer taken by smartconfig\_start.

**Note**:

Whether connect to AP succeed or not, this API should be called to free memory taken by smartconfig\_start.

**Prototype**:

bool smartconfig\_stop(void)

**Parameter**: null

**Return**:

true: succeed false: fail

### get\_smartconfig\_status

**Function**:

Get smart config status

**Note**:

Can not call this API after smartconfig\_stop, because smartconfig\_stop will try to free (again) the memory which contains this smart config status.

**Prototype**:

sc\_status get\_smartconfig\_status(void)

**Parameter**: null

**Return**:

typedef enum { SC\_STATUS\_WAIT = 0, SC\_STATUS\_FIND\_CHANNEL,

SC\_STATUS\_GETTING\_SSID\_PSWD, SC\_STATUS\_GOT\_SSID\_PSWD, SC\_STATUS\_LINK, SC\_STATUS\_LINK\_OVER,

} sc\_status;

**Note**:

Use APP to start connection when get\_smartconfig\_status is SC\_STATUS\_FIND\_CHANNEL.

## SNTP APIs

### sntp\_setserver

**Function**:

Set SNTP server by IP address, support 3 SNTP server at most

**Prototype**:

void sntp\_setserver(unsigned char idx, ip\_addr\_t \*addr)

**Parameter**:

unsigned char idx : SNTP server index， support 3 SNTP server at most（0 ~ 2）；index 0 is the main server，index 1 and 2 are as backup.

ip\_addr\_t \*addr : IP address; users need to ensure that it’s a SNTP server

**Return**:

none

### sntp\_getserver

**Function**:

Get IP address of SNTP server which set by sntp\_setserver

**Prototype**:

ip\_addr\_t sntp\_getserver(unsigned char idx)

**Parameter**:

unsigned char idx : SNTP server index, support 3 SNTP server at most（0 ~ 2）

**Return**:

IP address

### sntp\_setservername

**Function**:

Set SNTP server by domain name, support 3 SNTP server at most

**Prototype**:

void sntp\_setservername(unsigned char idx, char \*server)

**Parameter**:

unsigned char idx : SNTP server index， support 3 SNTP server at most（0 ~ 2）；index 0 is the main server，index 1 and 2 are as backup.

char \*server : domain name； users need to ensure that it’s a SNTP server

**Return**:

none

### sntp\_getservername

**Function**:

Get domain name of SNTP server which set by sntp\_setservername

**Prototype**:

char \* sntp\_getservername(unsigned char idx)

**Parameter**:

unsigned char idx : SNTP server index， support 3 SNTP server at most（0 ~ 2)

**Return**:

domain name

### sntp\_init

**Function**:

SNTP initialize

**Prototype**:

void sntp\_init(void)

**Parameter**: none

**Return**:

none

### sntp\_stop

**Function**:

Stop SNTP

**Prototype**:

void sntp\_stop(void)

**Parameter**: none

**Return**:

none

### sntp\_get\_current\_timestamp

**Function**:

Get current timestamp from basic time（1970.01.01 00：00：00 GMT）, uint:second

**Prototype**:

uint32 sntp\_get\_current\_timestamp()

**Parameter**: none

**Return**:

time stamp

### sntp\_get\_real\_time

**Function**:

Get real time（GMT + 8）

**Prototype**:

char\* sntp\_get\_real\_time(long t)

**Parameter**:

long t - time stamp

**Return**:

real time

### SNTP Example

ip\_addr\_t \*addr = (ip\_addr\_t \*)os\_zalloc(sizeof(ip\_addr\_t)); sntp\_setservername(0, ”us.pool.ntp.org”); // set server 0 by domain name

sntp\_setservername(1, ”ntp.sjtu.edu.cn”); // set server 1 by domain name ipaddr\_aton(“210.72.145.44”, addr);

sntp\_setserver(2, addr); // set server 2 by IP address sntp\_init();

os\_free(addr);

uint32 current\_stamp;

current\_stamp = sntp\_get\_current\_timestamp();

os\_printf(“sntp: %d, %s \n”,current\_stamp, sntp\_get\_real\_time(current\_stamp));

# TCP/UDP APIs

Found in esp\_iot\_sdk/include/espconn.h. The network APIs can be grouped into the following types:

* **General APIs**: APIs can be used for both TCP and UDP .
* **TCP APIs**: APIs that are only used for TCP.
* **UDP APIs**: APIs that are only used for UDP.
* **mDNS APIs**：APIs that related to mDNS

## Generic TCP/UDP APIs

### espconn\_delete

**Function**:

Delete a transmission.

**Note**:

Corresponding creation API : TCP: espconn\_accept, UDP: espconn\_create

**Prototype**:

sint8 espconn\_delete(struct espconn \*espconn)

**Parameter**:

struct espconn \*espconn : corresponding connected control block structure

**Return**:

0 : succeed

Non-0 : error, return error code ESPCONN\_ARG - illegal argument，can’t find network transmission according to structure espconn

### espconn\_gethostbyname

**Function**: DNS

**Prototype:**

err\_t espconn\_gethostbyname( struct espconn \*pespconn, const char \*hostname, ip\_addr\_t \*addr, dns\_found\_callback found

)

**Parameters**:

struct espconn \*espconn : corresponding connected control block structure const char \*hostname : domain name string pointer

ip\_addr\_t \*addr : IP address dns\_found\_callback found : callback

**Return**:

err\_t : ESPCONN\_OK - succeed

ESPCONN\_INPROGRESS - error code : already connected ESPCONN\_ARG - error code : illegal argument，can’t find network

transmission according to structure espconn

**Example as follows. Pls refer to source code of IoT\_Demo:**

ip\_addr\_t esp\_server\_ip;

LOCAL void ICACHE\_FLASH\_ATTR

user\_esp\_platform\_dns\_found(const char \*name, ip\_addr\_t \*ipaddr, void \*arg)

{

struct espconn \*pespconn = (struct espconn \*)arg; os\_printf(user\_esp\_platform\_dns\_found %d.%d.%d.%d/n,

\*((uint8 \*)&ipaddr->addr), \*((uint8 \*)&ipaddr->addr + 1),

\*((uint8 \*)&ipaddr->addr + 2), \*((uint8 \*)&ipaddr->addr + 3));

}

void dns\_test(void) { espconn\_gethostbyname(pespconn,iot.espressif.cn, &esp\_server\_ip,

user\_esp\_platform\_dns\_found);

}

### espconn\_port

**Function**: get void ports

**Prototype:**

uint32 espconn\_port(void)

**Parameter**: null

**Return**:

uint32 : id of the port you get

### espconn\_regist\_sentcb

**Function**:

Register data sent function which will be called back when data are successfully sent.

**Prototype:**

sint8 espconn\_regist\_sentcb( struct espconn \*espconn, espconn\_sent\_callback sent\_cb

)

**Parameters**:

struct espconn \*espconn : corresponding connected control block structure espconn\_sent\_callback sent\_cb : registered callback function

**Return**:

0 : succeed

Non-0 : error code ESPCONN\_ARG - illegal argument，can’t find network transmission according to structure espconn

### espconn\_regist\_recvcb

**Function**:

register data receive function which will be called back when data are received

**Prototype:**

sint8 espconn\_regist\_recvcb( struct espconn \*espconn, espconn\_recv\_callback recv\_cb

)

**Parameters**:

struct espconn \*espconn : corresponding connected control block structure espconn\_connect\_callback connect\_cb : registered callback function

**Return**:

0 : succeed

Non-0 : error code ESPCONN\_ARG - illegal argument，can’t find network transmission according to structure espconn

### espconn\_sent\_callback

**Function**:

Callback after the data are sent

**Prototype:**

void espconn\_sent\_callback (void \*arg)

**Parameters**:

void \*arg : call back function parameters

**Return**:

null

### espconn\_recv\_callback

**Function**:

callback after data are received

**Prototype:**

void espconn\_recv\_callback ( void \*arg,

char \*pdata, unsigned short len

)

**Parameters**:

void \*arg : callback function parameters char \*pdata : received data entry parameters unsigned short len : received data length

**Return**:

null

### espconn\_sent

**Function:**

Send data through WiFi

**Note**:

Please call espconn\_sent after espconn\_sent\_callback of the pre-packet.

**Prototype:**

sint8 espconn\_sent(

struct espconn \*espconn, uint8 \*psent,

uint16 length

)

**Parameters**:

struct espconn \*espconn : corresponding connected control block structure uint8 \*psent : sent data pointer

uint16 length : sent data length

**Return**:

0 : succeed Non-0 : error code

ESPCONN\_MEM - Out of memory

ESPCONN\_ARG - illegal argument，can’t find network transmission according to structure espconn

## TCP APIs

TCP APIs act only on TCP connections and do not affect nor apply to UDP connections.

### espconn\_accept

**Function**:

Creates a TCP server (i.e. accepts connections.)

**Prototype:**

sint8 espconn\_accept(struct espconn \*espconn)

**Parameter**:

struct espconn \*espconn : corresponding connected control block structure

**Return**:

0 : succeed Non-0 : error code

ESPCONN\_MEM - Out of memory ESPCONN\_ISCONN - Already connected

ESPCONN\_ARG - illegal argument，can’t find TCP connection according to

structure espconn

### espconn\_secure\_accept

**Function**:

Creates an SSL TCP server.

**Note:**

(1) Only created one SSL server is allowed，this API can be called only once， and only one SSL client is allowed to connect.

1. If SSL encrypted packet size is larger than ESP8266 SSL buffer size (default 2KB, set by espconn\_secure\_set\_size), SSL connection will fail, will enter espconn\_reconnect\_callback
2. SSL related APIs named as espconn\_secure\_XXX are different from normal TCP APIs, so please don’t mixed use. In SSL connection, only espconn\_secure\_XXX APIs, espconn\_regist\_XXX APIs and espconn\_port can be used.

**Prototype:**

sint8 espconn\_secure\_accept(struct espconn \*espconn)

**Parameter**:

struct espconn \*espconn : corresponding connected control block structure

**Return**:

0 : succeed Non-0 : error code

ESPCONN\_MEM - Out of memory ESPCONN\_ISCONN - Already connected

ESPCONN\_ARG - illegal argument，can’t find TCP connection according to

structure espconn

### espconn\_regist\_time

**Function**:

register timeout interval of ESP8266 TCP server.

**Note**:

Call this API after espconn\_accept.

If timeout is set to 0, timeout will be disable and ESP8266 TCP server will not disconnect TCP clients has stopped communication. This usage of timeout=0, is deprecated.

**Prototype:**

sint8 espconn\_regist\_time(

struct espconn \*espconn, uint32 interval,

uint8 type\_flag

)

**Parameters**:

struct espconn \*espconn : corresponding connected control block structure uint32 interval : timeout interval, unit: second, maximum: 7200 seconds uint8 type\_flag : 0, set all connections; 1, set a single connection

**Return**:

0 : succeed

Non-0 : error code ESPCONN\_ARG - illegal argument，can’t find TCP connection according to structure espconn

### espconn\_get\_connection\_info

**Function**:

Get a connection’s info in TCP multi-connection case

**Prototype:**

sint8 espconn\_get\_connection\_info( struct espconn \*espconn, remot\_info \*\*pcon\_info, uint8 typeflags

)

**Parameters**:

struct espconn \*espconn : corresponding connected control block structure remot\_info \*\*pcon\_info : connect to client info

uint8 typeflags : 0, regular server;1, ssl server

**Return**:

0 : succeed

Non-0 : error code ESPCONN\_ARG - illegal argument，can’t find TCP connection according to structure espconn

### espconn\_connect

**Function**:

Connect to a TCP server (ESP8266 acting as TCP client).

**Prototype:**

sint8 espconn\_connect(struct espconn \*espconn)

**Parameters**:

struct espconn \*espconn : corresponding connected control block structure

**Return**:

0 : succeed Non-0 : error code

ESPCONN\_RTE - Routing Problem ESPCONN\_MEM - Out of memory ESPCONN\_ISCONN - Already connected

ESPCONN\_ARG - illegal argument，can’t find TCP connection according to structure espconn

### espconn\_connect\_callback

**Function**: successful listening (ESP8266 as TCP server) or connection (ESP8266 as TCP client) callback, register by espconn\_regist\_connectcb

**Prototype:**

void espconn\_connect\_callback (void \*arg)

**Parameter**:

void \*arg : callback function parameters

**Return**:

null

### espconn\_set\_opt

**Function**: Set option of TCP connection

**Prototype**:

sint8 espconn\_set\_opt(

struct espconn \*espconn, uint8 opt

)

**Structure：**

enum espconn\_option{

ESPCONN\_START = 0x00， ESPCONN\_REUSEADDR = 0x01， ESPCONN\_NODELAY = 0x02， ESPCONN\_COPY = 0x04， ESPCONN\_KEEPALIVE = 0x08， ESPCONN\_END

}

**Parameter**:

struct espconn \*espconn : corresponding connected control structure uint8 opt : Option of TCP connection, refer to espconn\_option

bit 0: 1: free memory after TCP disconnection happen need not wait 2 minutes;

bit 1: 1: disable nalgo algorithm during TCP data transmission, quiken the data transmission.

bit 2: 1: use 2920 bytes write buffer for the data espconn\_sent sending.

bit 3: 1: enable TCP keep alive

**Return**:

0 : succeed

Non-0 : error code ESPCONN\_ARG - illegal argument，can’t find TCP connection according to structure espconn

**Note**:

In general, we need not call this API;

If call espconn\_set\_opt, please call it in espconn\_connect\_callback.

### espconn\_clear\_opt

**Function**:

Clear option of TCP connection.

**Prototype**:

sint8 espconn\_clear\_opt(

struct espconn \*espconn, uint8 opt

)

**Structure：**

enum espconn\_option{

ESPCONN\_START = 0x00， ESPCONN\_REUSEADDR = 0x01， ESPCONN\_NODELAY = 0x02， ESPCONN\_COPY = 0x04， ESPCONN\_KEEPALIVE = 0x08， ESPCONN\_END

}

**Parameters**:

struct espconn \*espconn : corresponding connected control block structure uint8 opt : option of TCP connection,refer to espconn\_option

**Return**: 0

Non-0

: succeed

: error code ESPCONN\_ARG - illegal argument，can’t find TCP

connection according to structure espconn

### espconn\_set\_keepalive

**Function**:

Set configuration of TCP keep alive .

Prototype**:**

sint8 espconn\_set\_keepalive(struct espconn \*espconn, uint8 level, void\* optarg)

#### Structure：

enum espconn\_level{ ESPCONN\_KEEPIDLE, ESPCONN\_KEEPINTVL, ESPCONN\_KEEPCNT

}

#### Parameters:

struct espconn \*espconn : corresponding connected control block structure

uint8 level : Default to do TCP keep-alive detection every ESPCONN\_KEEPIDLE, if there in no response, retry ESPCONN\_KEEPCNT times every ESPCONN\_KEEPINTVL. If still no response, considers it as TCP connection broke, goes into espconn\_reconnect\_callback .

description：

ESPCONN\_KEEPIDLE - TCP keep-alive interval，unit：500 millisecond ESPCONN\_KEEPINTVL - packet interval during TCP keep-alive，unit：500

millisecond

ESPCONN\_KEEPCNT - maximum packet count of TCP keep-alive void\* optarg ：value of parameter

#### Return:

0 : succeed

Non-0 : error code ESPCONN\_ARG - illegal argument，can’t find TCP connection according to structure espconn

#### Note:

In general, we need not call this API;

If needed, please call it in espconn\_connect\_callback and call espconn\_set\_opt to enable keep alive first.

### espconn\_get\_keepalive

**Function**:

Get value of TCP keep-alive parameter

**Prototype:**

sint8 espconn\_set\_keepalive(struct espconn \*espconn, uint8 level, void\* optarg)

**Structure：**

enum espconn\_level{ ESPCONN\_KEEPIDLE, ESPCONN\_KEEPINTVL, ESPCONN\_KEEPCNT

}

**Parameter**:

struct espconn \*espconn : corresponding connected control block structure uint8 level :

ESPCONN\_KEEPIDLE - TCP keep-alive interval，unit：500 millisecond

ESPCONN\_KEEPINTVL - packet interval during TCP keep-alive，unit：500 millisecond

ESPCONN\_KEEPCNT - maximum packet count of TCP keep-alive

void\* optarg ：value of parameter

**Return**: 0

Non-0

: succeed

: error code ESPCONN\_ARG - illegal argument，can’t find TCP

connection according to structure espconn

### espconn\_disconnect

**Function**:

disconnect a TCP connection

**Prototype:**

sint8 espconn\_disconnect(struct espconn \*espconn)

**Parameters**:

struct espconn \*espconn : corresponding connected control structure

**Return**:

0 : succeed

Non-0 : error code ESPCONN\_ARG - illegal argument，can’t find TCP connection according to structure espconn

### espconn\_regist\_connectcb

**Function**:

Register connection function which will be called back under successful TCP connection

**Prototype:**

sint8 espconn\_regist\_connectcb( struct espconn \*espconn,

espconn\_connect\_callback connect\_cb

)

**Parameters**:

struct espconn \*espconn : corresponding connected control block structure espconn\_connect\_callback connect\_cb : registered callback function

**Return**:

0 : succeed

Non-0 : error code ESPCONN\_ARG - illegal argument，can’t find TCP connection according to structure espconn

### espconn\_regist\_reconcb

**Function**:

Register reconnect callback

**Note**:

Reconnect callback is more like a network error handler; it handles errors that occurred in any phase of the connection. For instance, if espconn\_sent fails, reconnect callback will be called because the network is broken.

**Prototype:**

sint8 espconn\_regist\_reconcb( struct espconn \*espconn,

espconn\_connect\_callback recon\_cb

)

**Parameters**:

struct espconn \*espconn : corresponding connected control block structure espconn\_connect\_callback connect\_cb : registered callback function

**Return**:

0 : succeed

Non-0 : error code ESPCONN\_ARG - illegal argument，can’t find TCP connection according to structure espconn

### espconn\_regist\_disconcb

**Function**: register disconnection function which will be called back under successful TCP disconnection

**Prototype:**

sint8 espconn\_regist\_disconcb( struct espconn \*espconn,

espconn\_connect\_callback discon\_cb

)

**Parameters**:

struct espconn \*espconn : corresponding connected control block structure espconn\_connect\_callback connect\_cb : registered callback function

**Return**:

0 : succeed

Non-0 : error code ESPCONN\_ARG - illegal argument，can’t find TCP connection according to structure espconn

### espconn\_regist\_write\_finish

**Function**:

Register a callback which will be called when all sending data is completely write into write buffer or sent.

**Prototype**:

sint8 espconn\_regist\_write\_finish ( struct espconn \*espconn,

espconn\_connect\_callback write\_finish\_fn

)

**Parameters**:

struct espconn \*espconn : corresponding connected control block structure espconn\_connect\_callback write\_finish\_fn : registered callback function

**Return**:

0 : succeed

Non-0 : error code ESPCONN\_ARG - illegal argument，can’t find TCP connection according to structure espconn

### espconn\_secure\_set\_size

**Function**:

Set buffer size of encrypted data (SSL)

**Note:**

Buffer size default to be 2Kbytes. If need to change, please call this API before espconn\_secure\_accept (ESP8266 as TCP SSL server) or espconn\_secure\_connect (ESP8266 as TCP SSL client)

**Prototype**:

bool espconn\_secure\_set\_size (uint8 level, uint16 size)

**Parameters**:

uint8 level : set buffer for ESP8266 SSL server/client：

0x01 SSL client；0x02 SSL server；0x03 SSL client and SSL server uint16 size : buffer size，range：1 ~ 8192，unit：byte，default to be 2048

**Return**:

true : succeed false : fail

### espconn\_secure\_get\_size

**Function**:

Get buffer size of encrypted data (SSL)

**Prototype**:

sint16 espconn\_secure\_get\_size (uint8 level)

**Parameters**:

uint8 level : buffer for ESP8266 SSL server/client：

0x01 SSL client；0x02 SSL server；0x03 SSL client and SSL server

**Return**:

buffer size

### espconn\_secure\_connect

**Function**:

Secure connect (SSL) to a TCP server (ESP8266 is acting as TCP client.)

**Note:**

1. Only one connection is allowed when ESP8266 as SSL client, this API can be called only once, or call espconn\_secure\_disconnect to disconnect first, then call this API to create another SSL connection.
2. If SSL encrypted packet size is larger than ESP8266 SSL buffer size (default 2KB, set by espconn\_secure\_set\_size), SSL connection will fail, will enter espconn\_reconnect\_callback

(3) SSL related APIs named as espconn\_secure\_XXX are different from normal TCP APIs, so please don’t mixed use. In SSL connection, only espconn\_secure\_XXX APIs, espconn\_regist\_XXX APIs and espconn\_port can be used.

**Prototype**:

sint8 espconn\_secure\_connect (struct espconn \*espconn)

**Parameters**:

struct espconn \*espconn : corresponding connected control block structure

**Return**:

0 : succeed

Non-0 : error code ESPCONN\_ARG - illegal argument，can’t find TCP connection according to structure espconn

### espconn\_secure\_sent

**Function**: send encrypted data (SSL)

**Note**:

Please call espconn\_secure\_sent after espconn\_sent\_callback of the pre-packet.

**Prototype**:

sint8 espconn\_secure\_sent ( struct espconn \*espconn, uint8 \*psent,

uint16 length

)

**Parameters**:

struct espconn \*espconn : corresponding connected control block structure uint8 \*psent : sent data pointer

uint16 length : sent data length

**Return**:

0 : succeed

Non-0 : error code ESPCONN\_ARG - illegal argument，can’t find TCP connection according to structure espconn

### espconn\_secure\_disconnect

**Function**: secure TCP disconnection(SSL)

**Prototype:**

sint8 espconn\_secure\_disconnect(struct espconn \*espconn)

**Parameters**:

struct espconn \*espconn : corresponding connected control block structure

**Return**:

0 : succeed

Non-0 : error code ESPCONN\_ARG - illegal argument，can’t find TCP connection according to structure espconn

### espconn\_tcp\_get\_max\_con

**Function**:

Get maximum number of how many TCP connection is allowed.

**Prototype**:

uint8 espconn\_tcp\_get\_max\_con(void)

**Parameter**: null

**Return**:

Maximum number of how many TCP connection is allowed.

### espconn\_tcp\_set\_max\_con

**Function**:

Set the maximum number of how many TCP connection is allowed.

**Prototype**:

sint8 espconn\_tcp\_set\_max\_con(uint8 num)

**Parameter**:

uint8 num : Maximum number of how many TCP connection is allowed.

**Return**:

0 : succeed

Non-0 : error code ESPCONN\_ARG - illegal argument，can’t find TCP connection according to structure espconn

### espconn\_tcp\_get\_max\_con\_allow

**Function**:

Get the maximum number of TCP clients which are allowed to connect to ESP8266 TCP server.

**Prototype**:

sint8 espconn\_tcp\_get\_max\_con\_allow(struct espconn \*espconn)

**Parameter**:

struct espconn \*espconn : corresponding connected control structure

**Return**:

> 0 : Maximum number of TCP clients which are allowed.

< 0 : error code ESPCONN\_ARG - illegal argument，can’t find TCP connection according to structure espconn

### espconn\_tcp\_set\_max\_con\_allow

**Function**:

Set the maximum number of TCP clients allowed to connect to ESP8266 TCP server.

**Prototype**:

sint8 espconn\_tcp\_set\_max\_con\_allow(struct espconn \*espconn, uint8 num)

**Parameter**:

struct espconn \*espconn : corresponding connected control structure uint8 num : Maximum number of TCP clients which are allowed.

**Return**:

0 : succeed

Non-0 : error code ESPCONN\_ARG - illegal argument，can’t find TCP connection according to structure espconn

### espconn\_recv\_hold

**Function**:

Puts in a request to block the TCP receive function.

**Note**:

The function does not act immediately; we recommend calling it while reserving 5\*1460 bytes of memory.

This API can be called more than once.

**Prototype**:

sint8 espconn\_recv\_hold(struct espconn \*espconn)

**Parameter**:

struct espconn \*espconn : corresponding connected control structure

**Return**:

0 : succeed

Non-0 : error code ESPCONN\_ARG - illegal argument，can’t find TCP connection according to structure espconn

### espconn\_recv\_unhold

**Function**:

Unblock TCP receiving data (i.e. undo espconn\_recv\_hold).

**Note**:

This API takes effect immediately.

**Prototype**:

sint8 espconn\_recv\_unhold(struct espconn \*espconn)

**Parameter**:

struct espconn \*espconn : corresponding connected control structure

**Return**:

0 : succeed

Non-0 : error code ESPCONN\_ARG - illegal argument，can’t find TCP connection according to structure espconn

## UDP APIs

### espconn\_create

**Function**: create UDP transmission.

**Prototype**:

sin8 espconn\_create(struct espconn \*espconn)

**Parameter**:

struct espconn \*espconn : corresponding connected control block structure

**Return**:

0 : succeed Non-0 : error code

ESPCONN\_ISCONN - Already connected ESPCONN\_MEM - Out of memory

ESPCONN\_ARG - illegal argument，can’t find UDP transmission according to

structure espconn

### espconn\_igmp\_join

**Function**:

Join a multicast group

**Prototype**:

sint8 espconn\_igmp\_join(ip\_addr\_t \*host\_ip, ip\_addr\_t \*multicast\_ip)

**Parameters**:

ip\_addr\_t \*host\_ip : IP of host

ip\_addr\_t \*multicast\_ip : IP of multicast group

**Return**:

0 : succeed

Non-0 : error code ESPCONN\_MEM - Out of memory

### espconn\_igmp\_leave

**Function**:

Quit a multicast group

**Prototype**:

sint8 espconn\_igmp\_leave(ip\_addr\_t \*host\_ip, ip\_addr\_t \*multicast\_ip)

**Parameters**:

ip\_addr\_t \*host\_ip

: IP of host

ip\_addr\_t \*multicast\_ip : IP of multicast group

**Return**:

0 : succeed

Non-0 : error code ESPCONN\_MEM - Out of memory

## mDNS APIs

### espconn\_mdns\_init

**Function**:

mDNS initialization

**Note**：

1. Only ESP8266 station support mDNS, please get IP address of ESP8266 station first，then call this API to initial mDNS；
2. txt\_data has to be set as “ key = value ”，as **Example**；

**Structure**:

struct mdns\_info{ char \*host\_name; char \*server\_name;

uint16 server\_port; unsigned long ipAddr;

char \*txt\_data;

};

**Prototype**:

void espconn\_mdns\_init(struct mdns\_info \*info)

**Parameter**:

struct mdns\_info \*info : mdns information

**Return**:

none

**Example：**

struct mdns\_info \*info = (struct mdns\_info \*)os\_zalloc(sizeof(struct mdns\_info));

info->host\_name = “espressif”;

info->ipAddr = station\_ipconfig.ip.addr; //ESP8266 station IP info->server\_name = “iot”;

info->server\_port = 8080;

info->txt\_data = “version = now”; espconn\_mdns\_init(info);

### espconn\_mdns\_close

**Function**:

close mDNS, corresponding creation API : espconn\_mdns\_init

**Prototype**:

void espconn\_mdns\_close(void)

**Parameter**: none

**Return**:

none

### espconn\_mdns\_server\_register

**Function**:

register mDNS server

**Prototype**:

void espconn\_mdns\_server\_register(void)

**Parameter**: none

**Return**:

none

### espconn\_mdns\_server\_unregister

**Function**:

unregister mDNS server

**Prototype**:

void espconn\_mdns\_server\_unregister(void)

**Parameter**: none

**Return**:

none

### espconn\_mdns\_get\_servername

**Function**:

Get mDNS server name

**Prototype**:

char\* espconn\_mdns\_get\_servername(void)

**Parameter**: none

**Return**:

server name

### espconn\_mdns\_set\_servername

**Function**:

Set mDNS server name

**Prototype**:

void espconn\_mdns\_set\_servername(const char \*name)

**Parameter**:

const char \*name - server name

**Return**:

none

### espconn\_mdns\_set\_hostname

**Function**:

Set mDNS host name

**Prototype**:

void espconn\_mdns\_set\_hostname(char \*name)

**Parameter**:

char \*name - host name

**Return**:

none

### espconn\_mdns\_get\_hostname

**Function**:

Get mDNS host name

**Prototype**:

char\* espconn\_mdns\_get\_hostname(void)

**Parameter**: none

**Return**:

host name

### espconn\_mdns\_disable

**Function**:

Disable mDNS , corresponding creation API : espconn\_mdns\_enable

**Prototype**:

void espconn\_mdns\_disable(void)

**Parameter**: none

**Return**:

none

### espconn\_mdns\_enable

**Function**:

Enable mDNS

**Prototype**:

void espconn\_mdns\_enable(void)

**Parameter**: none

**Return**:

none

# Application Related

## AT APIs

for AT APIs examples, refer to esp\_iot\_sdk/examples/at/user/user\_main.c.

### at\_response\_ok

**Function**:

Output OK to AT Port (UART0)

**Prototype**:

void at\_response\_ok(void)

**Parameter**: null

**Return**:

null

### at\_response\_error

**Function**:

output ERROR to AT Port (UART0)

**Prototype**:

void at\_response\_error(void)

**Parameter**: null

**Return**:

null

### at\_cmd\_array\_regist

**Function**:

register user-define AT commands.

Can be called only once to register all user-define AT commands.

**Prototype**:

void at\_cmd\_array\_regist (

at\_function \* custom\_at\_cmd\_arrar, uint32 cmd\_num

)

**Parameter**:

at\_function \* custom\_at\_cmd\_arrar : Array of user-define AT commands uint32 cmd\_num : Number counts of user-define AT commands

**Return**:

null

**Example**:

refer to esp\_iot\_sdk/examples/at/user/user\_main.c

### at\_get\_next\_int\_dec

**Function**:

parse int from AT command

**Prototype**:

bool at\_get\_next\_int\_dec (char \*\*p\_src,int\* result,int\* err)

**Parameter**:

char \*\*p\_src : \*p\_src is the AT command that need to be parsed int\* result : int number parsed from the AT command

int\* err : 1: no number is found; 3: only ’-’ is found.

**Return**:

true: parser succeeds (NOTE: if no number is found, it will return True, but returns error code 1)

false: parser is unable to parse string; some probable causes are: int number more than 10 bytes; string contains termination characters '/r'; string contains only '-'.

**Example**:

refer to esp\_iot\_sdk/examples/at/user/user\_main.c

### at\_data\_str\_copy

**Function**: parse string from AT command

**Prototype**:

int32 at\_data\_str\_copy (char \* p\_dest, char \*\* p\_src,int32 max\_len)

**Parameter**:

char \* p\_dest : string parsed from the AT command

char \*\* p\_src : \*p\_src is the AT command that need to be parsed int32 max\_len : max string length that allowed

**Return**:

length of string:

>=0: succeed and returns the length of the string

<0 : fail and returns -1

**Example**:

refer to esp\_iot\_sdk/examples/at/user/user\_main.c

### at\_init

**Function**:

AT initialize

**Prototype**:

void at\_init (void)

**Parameter**: null

**Return**:

null

**Example**:

refer to esp\_iot\_sdk/examples/at/user/user\_main.c

### at\_port\_print

**Function**:

output string to AT PORT(UART0)

**Prototype**:

void at\_port\_print(const char \*str)

**Parameter**:

const char \*str : string that need to output

**Return**:

null

**Example**:

refer to esp\_iot\_sdk/examples/at/user/user\_main.c

### at\_set\_custom\_info

**Function**:

User-define version info of AT which can be got by AT+GMR.

**Prototype**:

void at\_set\_custom\_info (char \*info)

**Parameter**:

char \*info : version info

**Return**:

null

### at\_enter\_special\_state

**Function**:

Enter processing state. In processing state, AT core will return busy for any further AT commands.

**Prototype**:

void at\_enter\_special\_state (void)

**Parameter**: null

**Return**:

null

### at\_leave\_special\_state

**Function**:

Exit from AT processing state.

**Prototype**:

void at\_leave\_special\_state (void)

**Parameter**: null

**Return**:

null

### at\_get\_version

**Function**:

Get Espressif AT lib version.

**Prototype**:

uint32 at\_get\_version (void)

**Parameter**: null

**Return**:

Espressif AT lib version

### at\_register\_uart\_rx\_intr

**Function**:

Set UART0 to be used by user or AT commands.

**Note:**

This API can be called multiple times.

Running AT，UART0 default to be used by AT commands.

**Prototype**:

void at\_register\_uart\_rx\_intr(at\_custom\_uart\_rx\_intr rx\_func)

**Parameter**:

at\_custom\_uart\_rx\_intr : register a UART0 RX interrupt handler so that UART0 can be used by the customer, while if it’s NULL, UART0 is assigned to AT commands.

**Return**:

null

**Example**:

void user\_uart\_rx\_intr(uint8\* data, int32 len)

{

// UART0 rx for user os\_printf(“len=%d \r\n”,len); os\_printf(data);

// change UART0 for AT at\_register\_uart\_rx\_intr(NULL);

}

void user\_init(void){ at\_register\_uart\_rx\_intr(user\_uart\_rx\_intr); }

## Related JSON APIs

Found in : esp\_iot\_sdk/include/json/jsonparse.h & jsontree.h

### jsonparse\_setup

**Function**:

json initialize parsing

**Prototype:**

void jsonparse\_setup(

struct jsonparse\_state \*state, const char \*json,

int len

)

**Parameters**:

struct jsonparse\_state \*state : json parsing pointer const char \*json : json parsing character string int len : character string length

**Return**:

null

### jsonparse\_next

**Function**:

Returns jsonparse next object

**Prototype**:

int jsonparse\_next(struct jsonparse\_state \*state)

**Parameters**:

struct jsonparse\_state \*state : json parsing pointer

**Return**:

int : parsing result

### jsonparse\_copy\_value

**Function**:

Copies current parsing character string to a certain buffer

**Prototype**:

int jsonparse\_copy\_value(

struct jsonparse\_state \*state, char \*str,

int size

)

**Parameters**:

struct jsonparse\_state \*state : json parsing pointer char \*str : buffer pointer

int size : buffer size

**Return**:

int : copy result

### jsonparse\_get\_value\_as\_int

**Function**:

Parses json to get integer

**Prototype**:

int jsonparse\_get\_value\_as\_int(struct jsonparse\_state \*state)

**Parameters**:

struct jsonparse\_state \*state : json parsing pointer

**Return**:

int : parsing result

### jsonparse\_get\_value\_as\_long

**Function**:

Parses json to get long integer

**Prototype**:

long jsonparse\_get\_value\_as\_long(struct jsonparse\_state \*state)

**Parameters**:

struct jsonparse\_state \*state : json parsing pointer

**Return**:

long : parsing result

### jsonparse\_get\_len

**Function**:

Gets parsed json length

**Prototype**:

int jsonparse\_get\_value\_len(struct jsonparse\_state \*state)

**Parameters**:

struct jsonparse\_state \*state : json parsing pointer

**Return**:

int : parsed jason length

### jsonparse\_get\_value\_as\_type

**Function**:

Parses json data type

**Prototype**:

int jsonparse\_get\_value\_as\_type(struct jsonparse\_state \*state)

**Parameters**:

struct jsonparse\_state \*state : json parsing pointer

**Return**:

int : parsed json data type

### jsonparse\_strcmp\_value

**Function**:

Compares parsed json and certain character string

**Prototype:**

int jsonparse\_strcmp\_value(struct jsonparse\_state \*state, const char \*str)

**Parameters**:

struct jsonparse\_state \*state : json parsing pointer const char \*str : character buffer

**Return**:

int : comparison result

### jsontree\_set\_up

**Function**:

Creates json data tree

**Prototype**:

void jsontree\_setup(

struct jsontree\_context \*js\_ctx, struct jsontree\_value \*root,

int (\* putchar)(int)

)

**Parameters**:

struct jsontree\_context \*js\_ctx : json tree element pointer struct jsontree\_value \*root : root element pointer

int (\* putchar)(int) : input function

**Return**:

null

### jsontree\_reset

**Function**:

Resets json tree

**Prototype**:

void jsontree\_reset(struct jsontree\_context \*js\_ctx)

**Parameters**:

struct jsontree\_context \*js\_ctx : json data tree pointer

**Return**:

null

### jsontree\_path\_name

**Function**:

get json tree parameters

**Prototype**:

const char \*jsontree\_path\_name(

const struct jsontree\_cotext \*js\_ctx, int depth

)

**Parameters**:

struct jsontree\_context \*js\_ctx : json tree pointer int depth : json tree depth

**Return**:

char\* : parameter pointer

### jsontree\_write\_int

**Function**:

write integer to json tree

**Prototype**:

void jsontree\_write\_int(

const struct jsontree\_context \*js\_ctx, int value

)

**Parameters**:

struct jsontree\_context \*js\_ctx : json tree pointer int value : integer value

**Return**:

null

### jsontree\_write\_int\_array

**Function**:

Writes integer array to json tree

**Prototype**:

void jsontree\_write\_int\_array(

const struct jsontree\_context \*js\_ctx, const int \*text,

uint32 length

)

**Parameters**:

struct jsontree\_context \*js\_ctx : json tree pointer int \*text : array entry address

uint32 length : array length

**Return**:

null

### jsontree\_write\_string

**Function**:

Writes string to json tree

**Prototype**:

void jsontree\_write\_string(

const struct jsontree\_context \*js\_ctx, const char \*text

)

**Parameters**:

struct jsontree\_context \*js\_ctx : json tree pointer const char\* text : character string pointer

**Return**:

null

### jsontree\_print\_next

**Function**:

json tree depth

**Prototype**:

int jsontree\_print\_next(struct jsontree\_context \*js\_ctx)

**Parameters**:

struct jsontree\_context \*js\_ctx : json tree pointer

**Return**:

int : json tree depth

### jsontree\_find\_next

**Function**:

find json tree element

**Prototype**:

struct jsontree\_value \*jsontree\_find\_next( struct jsontree\_context \*js\_ctx, int type

)

**Parameters**:

struct jsontree\_context \*js\_ctx : json tree pointer int : type

**Return**:

struct jsontree\_value \* : json tree element pointer

# Definition of Structures

## Timer

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;

## WiFi Related Structures

### Station Related

struct station\_config { uint8 ssid[32]; uint8 password[64]; uint8 bssid\_set; uint8 bssid[6];

};

**Note**:

BSSID as MAC address of AP, will be used when several APs have the same SSID.

If station\_config.bssid\_set==1 , station\_config.bssid has to be set, otherwise, the connection will fail.

In general, station\_config.bssid\_set need to be 0.

### soft-AP related

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 ssid\_len;

uint8 channel; // support 1 ~ 13

uint8 authmode; // Don’t support AUTH\_WEP in soft-AP mode uint8 ssid\_hidden; // default 0

uint8 max\_connection; // default 4, max 4

uint16 beacon\_interval; // 100 ~ 60000 ms, default 100

};

**Note**:

If softap\_config.ssid\_len==0, check ssid till find a termination characters; otherwise, it depends on softap\_config.ssid\_len.

### scan related

struct scan\_config { uint8 \*ssid; uint8 \*bssid; uint8 channel;

uint8 show\_hidden; // Scan APs which are hiding their SSID or not.

};

struct bss\_info { STAILQ\_ENTRY(bss\_info) next; u8 bssid[6];

u8 ssid[32]; u8 channel; s8 rssi;

u8 authmode;

uint8 is\_hidden; // SSID of current AP is hidden or not.

};

typedef void (\* scan\_done\_cb\_t)(void \*arg, STATUS status);

### WiFi event related structure

enum {

EVENT\_STAMODE\_CONNECTED = 0, EVENT\_STAMODE\_DISCONNECTED, EVENT\_STAMODE\_AUTHMODE\_CHANGE, EVENT\_STAMODE\_GOT\_IP, EVENT\_SOFTAPMODE\_STACONNECTED,

EVENT\_SOFTAPMODE\_STADISCONNECTED,

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| }; | EVENT\_MAX |  |  |  | | |
| enum | {  REASON\_UNSPECIFIED | = | 1, |
|  | REASON\_AUTH\_EXPIRE | = | 2, |
|  | REASON\_AUTH\_LEAVE | = | 3, |
|  | REASON\_ASSOC\_EXPIRE | = | 4, |
|  | REASON\_ASSOC\_TOOMANY | = | 5, |
|  | REASON\_NOT\_AUTHED | = | 6, |
|  | REASON\_NOT\_ASSOCED | = | 7, |
|  | REASON\_ASSOC\_LEAVE | = | 8, |
|  | REASON\_ASSOC\_NOT\_AUTHED | = | 9, |
|  | REASON\_DISASSOC\_PWRCAP\_BAD | = | 10, | /\* | 11h | \*/ |
|  | REASON\_DISASSOC\_SUPCHAN\_BAD | = | 11, | /\* | 11h | \*/ |
|  | REASON\_IE\_INVALID | = | 13, | /\* | 11i | \*/ |
|  | REASON\_MIC\_FAILURE | = | 14, | /\* | 11i | \*/ |
|  | REASON\_4WAY\_HANDSHAKE\_TIMEOUT | = | 15, | /\* | 11i | \*/ |
|  | REASON\_GROUP\_KEY\_UPDATE\_TIMEOUT | = | 16, | /\* | 11i | \*/ |
|  | REASON\_IE\_IN\_4WAY\_DIFFERS | = | 17, | /\* | 11i | \*/ |
|  | REASON\_GROUP\_CIPHER\_INVALID | = | 18, | /\* | 11i | \*/ |
|  | REASON\_PAIRWISE\_CIPHER\_INVALID | = | 19, | /\* | 11i | \*/ |
|  | REASON\_AKMP\_INVALID | = | 20, | /\* | 11i | \*/ |
|  | REASON\_UNSUPP\_RSN\_IE\_VERSION | = | 21, | /\* | 11i | \*/ |
|  | REASON\_INVALID\_RSN\_IE\_CAP | = | 22, | /\* | 11i | \*/ |
|  | REASON\_802\_1X\_AUTH\_FAILED | = | 23, | /\* | 11i | \*/ |
|  | REASON\_CIPHER\_SUITE\_REJECTED | = | 24, | /\* | 11i | \*/ |
|  | REASON\_BEACON\_TIMEOUT | = | 200, |  |  |  |
| }; | REASON\_NO\_AP\_FOUND | = | 201, |  |  |  |

typedef struct {

uint8 ssid[32]; uint8 ssid\_len; uint8 bssid[6]; uint8 channel;

} Event\_StaMode\_Connected\_t;

typedef struct {

uint8 ssid[32]; uint8 ssid\_len; uint8 bssid[6]; uint8 reason;

} Event\_StaMode\_Disconnected\_t;

typedef struct {

uint8 old\_mode; uint8 new\_mode;

} Event\_StaMode\_AuthMode\_Change\_t;

typedef struct {

struct ip\_addr ip; struct ip\_addr mask; struct ip\_addr gw;

} Event\_StaMode\_Got\_IP\_t;

typedef struct {

uint8 mac[6]; uint8 aid;

} Event\_SoftAPMode\_StaConnected\_t;

typedef struct {

uint8 mac[6]; uint8 aid;

} Event\_SoftAPMode\_StaDisconnected\_t;

typedef union {

Event\_StaMode\_Connected\_t connected;

Event\_StaMode\_Disconnected\_t disconnected; Event\_StaMode\_AuthMode\_Change\_t auth\_change; Event\_StaMode\_Got\_IP\_t got\_ip; Event\_SoftAPMode\_StaConnected\_t sta\_connected; Event\_SoftAPMode\_StaDisconnected\_t sta\_disconnected;

} Event\_Info\_u;

typedef struct \_esp\_event {

uint32 event; Event\_Info\_u event\_info;

} System\_Event\_t;

### smart config structure

typedef enum {

SC\_STATUS\_WAIT = 0, // Please don’t start connection in this phase SC\_STATUS\_FIND\_CHANNEL, // Start connection by APP in this phase SC\_STATUS\_GETTING\_SSID\_PSWD,

SC\_STATUS\_GOT\_SSID\_PSWD, SC\_STATUS\_LINK,

SC\_STATUS\_LINK\_OVER, // Got IP, connect to AP successfully

} sc\_status; typedef enum {

SC\_TYPE\_ESPTOUCH = 0, SC\_TYPE\_AIRKISS,

} sc\_type;

## JSON Related Structure

### json structure

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];

};

### json macro definition

#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 }

## espconn parameters

### callback 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);

### espconn

typedef void\* espconn\_handle; typedef struct \_esp\_tcp {

int remote\_port; int local\_port; uint8 local\_ip[4]; uint8 remote\_ip[4];

espconn\_connect\_callback connect\_callback; espconn\_reconnect\_callback reconnect\_callback; espconn\_connect\_callback disconnect\_callback; espconn\_connect\_callback write\_finish\_fn;

} esp\_tcp;

typedef struct \_esp\_udp { int remote\_port;

int local\_port; uint8 local\_ip[4]; uint8 remote\_ip[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

};

enum espconn\_option{

ESPCONN\_START = 0x00， ESPCONN\_REUSEADDR = 0x01， ESPCONN\_NODELAY = 0x02， ESPCONN\_COPY = 0x04， ESPCONN\_KEEPALIVE = 0x08， ESPCONN\_END

}

enum espconn\_level{

ESPCONN\_KEEPIDLE, ESPCONN\_KEEPINTVL, ESPCONN\_KEEPCNT

}

/\*\* 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; uint8 link\_cnt;

void \*reverse; // reversed for customer use

};

# Peripheral Related Drivers

## GPIO Related APIs

Please refer to /user/user\_plug.c.

### PIN Related Macros

The following macros are used to control the GPIO pins' status.

PIN\_PULLUP\_DIS(PIN\_NAME)

Disable pin pull up

PIN\_PULLUP\_EN(PIN\_NAME)

Enable pin pull up

PIN\_FUNC\_SELECT(PIN\_NAME, FUNC)

Select pin function

**Example**:

PIN\_FUNC\_SELECT(PERIPHS\_IO\_MUX\_MTDI\_U, FUNC\_GPIO12); // Use MTDI pin as GPIO12.

### gpio\_output\_set

**Function**: set gpio property

**Prototype:**

void gpio\_output\_set( uint32 set\_mask, uint32 clear\_mask, uint32 enable\_mask, uint32 disable\_mask

)

**Input Parameters**:

uint32 set\_mask : set high output; 1:high output; 0:no status change uint32 clear\_mask : set low output; 1:low output; 0:no status change uint32 enable\_mask : enable outpout bit

uint32 disable\_mask : enable input bit

**Return**:

null

**Example**:

gpio\_output\_set(BIT12, 0, BIT12, 0): Set GPIO12 as high-level output; gpio\_output\_set(0, BIT12, BIT12, 0):

Set GPIO12 as low-level output gpio\_output\_set(BIT12, BIT13, BIT12|BIT13, 0):

Set GPIO12 as high-level output, GPIO13 as low-level output. gpio\_output\_set(0, 0, 0, BIT12):

Set GPIO12 as input

### GPIO input and output macro

GPIO\_OUTPUT\_SET(gpio\_no, bit\_value)

Set gpio\_no as output bit\_value, the same as the output example in 5.1.2

GPIO\_DIS\_OUTPUT(gpio\_no)

Set gpio\_no as input, the same as the input example in 5.1.2.

GPIO\_INPUT\_GET(gpio\_no)

Get the level status of gpio\_no.

### GPIO interrupt

ETS\_GPIO\_INTR\_ATTACH(func, arg)

Register GPIO interrupt control function

ETS\_GPIO\_INTR\_DISABLE()

Disable GPIO interrupt

ETS\_GPIO\_INTR\_ENABLE()

Enable GPIO interrupt

### gpio\_pin\_intr\_state\_set

**Function**:

set GPIO interrupt state

**Prototype:**

void gpio\_pin\_intr\_state\_set( uint32 i,

GPIO\_INT\_TYPE intr\_state

)

|  |  |  |
| --- | --- | --- |
| 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 Interrupt Handler

**Input Parameters**:

uint32 i : GPIO pin ID, if you want to set GPIO14, pls use GPIO\_ID\_PIN(14); GPIO\_INT\_TYPE intr\_state : interrupt type as the following:

typedef enum {

} GPIO\_INT\_TYPE;

**Return**:

null

Follow the steps below to clear interrupt status in GPIO interrupt processing function:

uint32 gpio\_status;

gpio\_status = GPIO\_REG\_READ(GPIO\_STATUS\_ADDRESS);

//clear interrupt status GPIO\_REG\_WRITE(GPIO\_STATUS\_W1TC\_ADDRESS, gpio\_status);

## UART Related APIs

By default, UART0 is debug output interface. In the case of dual Uart, UART0 works as data receive and transmit interface, and UART1as debug output interface.

Please make sure all hardware are correctly connected.

### uart\_init

**Function**:

Initializes baud rates of the two uarts

**Prototype:**

void uart\_init( UartBautRate uart0\_br, UartBautRate uart1\_br

)

**Parameters**:

UartBautRate uart0\_br : uart0 baud rate UartBautRate uart1\_br : uart1 baud rate

|  |  |  |
| --- | --- | --- |
| 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 |

### uart0\_tx\_buffer

**Baud Rates**:

typedef enum {

} UartBautRate;

**Return**:

null

**Function**:

Sends user-defined data through UART0

**Prototype:**

void uart0\_tx\_buffer(uint8 \*buf, uint16 len)

**Parameter**:

uint8 \*buf : data to send later

uint16 len : the length of data to send later

**Return**:

null

### uart0\_rx\_intr\_handler

**Function**:

UART0 interrupt processing function. Users can add the processing of received data in this function. (Receive buffer size: 0x100; if the received data are more than 0x100, pls handle them yourselves.)

**Prototype:**

void uart0\_rx\_intr\_handler(void \*para)

**Parameter**:

void \*para : the pointer pointing to RcvMsgBuff structure

**Return**:

null

## I2C Master Related APIs

### i2c\_master\_gpio\_init

**Function**:

Set GPIO in I2C master mode

**Prototype:**

void i2c\_master\_gpio\_init (void)

**Input Parameters**: null

**Return**:

null

### i2c\_master\_init

**Function**:

Initialize I2C

**Prototype:**

void i2c\_master\_init(void)

**Input Parameters**: null

**Return**:

null

### i2c\_master\_start

**Function**: configures I2C to start sending data

**Prototype:**

void i2c\_master\_start(void)

**Input Parameters**: null

**Return**:

null

### i2c\_master\_stop

**Function**:

configures I2C to stop sending data

**Prototype:**

void i2c\_master\_stop(void)

**Input Parameters**: null

**Return**:

null

### i2c\_master\_send\_ack

**Function**:

Sends I2C ACK

**Prototype:**

void i2c\_master\_send\_ack (void)

**Input Parameters**: null

**Return**:

null

### i2c\_master\_send\_nack

**Function**:

Sends I2C NACK

**Prototype:**

void i2c\_master\_send\_nack (void)

**Input Parameters**: null

**Return**:

null

### i2c\_master\_checkAck

**Function**:

Checks ACK from slave

**Prototype:**

bool i2c\_master\_checkAck (void)

**Input Parameters**: null

**Return**:

true: get I2C slave ACK false: get I2C slave NACK

### i2c\_master\_readByte

**Function**:

Read one byte from I2C slave

**Prototype:**

uint8 i2c\_master\_readByte (void)

**Input Parameters**: null

**Return**:

uint8 : the value that was read

### i2c\_master\_writeByte

**Function**:

Write one byte to slave

**Prototype:**

void i2c\_master\_writeByte (uint8 wrdata)

**Input Parameters**:

uint8 wrdata : data to write

**Return**:

null

## PWM Related

ESP9266 supports 4x PWM outputs. More details can be found in pwm.h. It is possible to increase the number of PWM outputs, but it is beyond the scope of this document.

### pwm\_init

**Function**:

Initialize PWM function, including GPIO selection, frequency and duty cycle.

**Prototype:**

void pwm\_init(uint16 freq, uint8 \*duty)

**Input Parameters**:

uint16 freq : PWM frequency;

uint8 \*duty : duty cycle of each output

**Return**:

null

### pwm\_start

**Function**:

Starts PWM. This function needs to be called after PWM config is changed.

**Prototype**:

void pwm\_start (void)

**Parameter**: null

**Return**:

null

### pwm\_set\_duty

**Function**:

Sets duty cycle of an output

**Prototype:**

void pwm\_set\_duty(uint8 duty, uint8 channel)

**Input Parameters**:

uint8 duty : duty cycle uint8 channel : an output

**Return**:

null

### pwm\_set\_freq

**Function**:

Sets PWM frequency

**Prototype:**

void pwm\_set\_freq(uint16 freq)

**Input Parameters**:

uint16 freq : PWM frequency

**Return**:

null

### pwm\_get\_duty

**Function**:

Gets duty cycle of PWM output

**Prototype:**

uint8 pwm\_get\_duty(uint8 channel)

**Input Parameters**:

uint8 channel : channel of which to get duty cycle

**Return**:

uint8 : duty cycle

### pwm\_get\_freq

**Function**:

Gets PWM frequency.

**Prototype:**

uint16 pwm\_get\_freq(void)

**Input Parameters**: null

**Return**:

uint16 : frequency

# Appendix

## ESPCONN Programming

### TCP Client Mode

Notes

* ESP8266, работающий в режиме станции, начнет клиентское соединение при получении IP-адреса.
* ESP8266, работающая в режиме soft-AP, начнет клиентское подключение, когда устройствам, подключенным к ESP8266, будет присвоен IP-адрес.

Steps

* Initialize espconn parameters according to protocols.
* Зарегистрируйте функцию обратного вызова connect и зарегистрируйте функцию обратного вызова reconnect.
  + (Call espconn\_regist\_connectcb and espconn\_regist\_reconcb )
* Call espconn\_connect function and set up the connection with TCP Server.
* Зарегистрированная подключенная функция обратного вызова будет вызвана после успешного подключения, которая зарегистрирует соответствующую функцию обратного вызова. Рекомендую зарегистрировать функцию обратного вызова отключения.
  + (Call espconn\_regist\_recvcb , espconn\_regist\_sentcb and espconn\_regist\_disconcb

in connected callback)

* При использовании функции приема обратного вызова или функции обратного вызова для запуска отключения рекомендуется установить временную задержку, чтобы убедиться, что все функции прошивки завершены.

### TCP Server Mode

Notes

* Если ESP8266 находится в режиме станции, он начнет прослушивание сервера при получении IP-адреса.
* Если ESP8266 находится в режиме soft-AP, он начнет прослушивание сервера.

Steps

* Initialize espconn parameters according to protocols.
* Register connect callback and reconnect callback function.
  + (Call espconn\_regist\_connectcb and espconn\_regist\_reconcb )
* Call espconn\_accept function to listen to the connection with host.
* Registered connect function will be called after successful connection, which will register corresponding callback function.
  + (Call espconn\_regist\_recvcb , espconn\_regist\_sentcb and espconn\_regist\_disconcb

in connected callback)

## RTC APIs Example

Demo code below shows how to get RTC time and to read and write to RTC memory.

void user\_init(void) {

os\_printf(clk cal : %d /n/r,system\_rtc\_clock\_cali\_proc()>>12);

uint32 rtc\_time = 0, rtc\_reg\_val = 0,stime = 0,rtc\_time2 = 0,stime2 = 0; rtc\_time = system\_get\_rtc\_time();

stime = system\_get\_time();

os\_printf(rtc time : %d /n/r,rtc\_time); os\_printf(system time : %d /n/r,stime);

if( system\_rtc\_mem\_read(0, &rtc\_reg\_val, 4)) { os\_printf(rtc mem val : 0x%08x/n/r,rtc\_reg\_val);

} else {

os\_printf(rtc mem val error/n/r);

}

rtc\_reg\_val++;

os\_printf(rtc mem val write/n/r); system\_rtc\_mem\_write(0, &rtc\_reg\_val, 4) ;

if( system\_rtc\_mem\_read(0, &rtc\_reg\_val, 4) ){ os\_printf(rtc mem val : 0x%08x/n/r,rtc\_reg\_val);

} else {

os\_printf(rtc mem val error/n/r);

}

rtc\_time2 = system\_get\_rtc\_time(); stime2 = system\_get\_time();

os\_printf(rtc time : %d /n/r,rtc\_time2); os\_printf(system time : %d /n/r,stime2); os\_printf(delta time rtc: %d /n/r,rtc\_time2-rtc\_time);

os\_printf(delta system time rtc: %d /n/r,stime2-stime); os\_printf(clk cal : %d /n/r,system\_rtc\_clock\_cali\_proc()>>12);

os\_delay\_us(500000); system\_restart();

}

## Sniffer Structure Introduction

ESP8266 can enter promiscuous mode (sniffer) and capture IEEE 802.11 packets in the air. The following HT20 packets are support:

* 802.11b
* 802.11g
* 802.11n (from MCS0 to MCS7)
* AMPDU types of packets The following are not supported:
* HT40
* LDPC

Although ESP8266 can not completely decipher these kinds of IEEE80211 packets completely, it can still obtain the length of these special packets.

In summary, while in sniffer mode, ESP8266 can either capture completely the packets or obtain the length of the packet:

* Packets that ESP8266 can decipher completely; ESP8266 returns with the
  + MAC address of the both side of communication and encryption type and
  + the length of entire packet.
* Packets that ESP8266 can only partial decipher; ESP8266 returns with
  + the length of packet.

Structure RxControl and sniffer\_buf are used to represent these two kinds of packets. Structure

sniffer\_buf contains structure RxControl.

struct RxControl { signed rssi:8; unsigned rate:4; unsigned is\_group:1;

unsigned:1;

// signal intensity of packet

unsigned sig\_mode:2; // 0:is 11n packet; 1:is not 11n packet; unsigned legacy\_length:12; // if not 11n packet, shows length of packet. unsigned damatch0:1;

unsigned damatch1:1; unsigned bssidmatch0:1; unsigned bssidmatch1:1;

unsigned MCS:7; // if is 11n packet, shows the modulation

// and code used (range from 0 to 76) unsigned CWB:1; // if is 11n packet, shows if is HT40 packet or not unsigned HT\_length:16;// if is 11n packet, shows length of packet. unsigned Smoothing:1;

unsigned Not\_Sounding:1; unsigned:1;

unsigned Aggregation:1; unsigned STBC:2;

unsigned FEC\_CODING:1; // if is 11n packet, shows if is LDPC packet or not. unsigned SGI:1;

unsigned rxend\_state:8; unsigned ampdu\_cnt:8;

unsigned channel:4; //which channel this packet in. unsigned:12;

};

struct LenSeq{

u16 len; // length of packet

u16 seq; // serial number of packet, the high 12bits are serial number,

// low 14 bits are Fragment number (usually be 0) u8 addr3[6]; // the third address in packet

};

struct sniffer\_buf{

struct RxControl rx\_ctrl;

u8 buf[36 ]; // head of ieee80211 packet u16 cnt; // number count of packet struct LenSeq lenseq[1]; //length of packet

};

Callback wifi\_promiscuous\_rx has two parameters ( buf and len). len means the length of buf, len = 12 or len ≥ 60:

Case of LEN ≥ 60

* buf contains structure sniffer\_buf: this structure is reliable, data packets represented by it has been verified by CRC.
* sniffer\_buf.cnt means the count of packets in buf. The value of len depends on

sniffer\_buf.cnt.

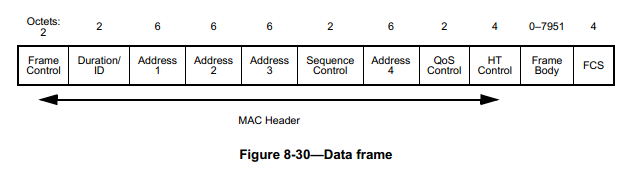
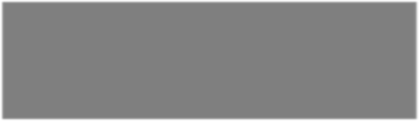
* + sniffer\_buf.cnt==0, invalid buf; otherwise, len = 50 + cnt \* 10
* sniffer\_buf.buf contains the first 36 bytes of ieee80211 packet. Starting from sniffer\_buf.lenseq[0], each structure lenseq represent a length information of packet. lenseq[0] represents the length of first packet. If there are two packets where (sniffer\_buf.cnt == 2), lenseq[1] represents the length of second packet.
* If sniffer\_buf.cnt > 1, it is a AMPDU packet, head of each MPDU packets are similar, so we only provide the length of each packet (from head of MAC packet to FCS)
* This structure contains: length of packet, MAC address of both sides of communication, length of the head of packet.

Case of LEN==12

* buf contains structure RxControl; but this structure is not reliable, we can not get neither MAC address of both sides of communication nor length of the head of packet.
* For AMPDU packet, we can not get the count of packets or the length of packet.
* This structure contains: length of packet, rssi and FEC\_CODING.
* RSSI and FEC\_CODING are used to guess if the packets are sent from same device.

Summary

We should not take too long to process the packets. Otherwise, other packets may be lost. The diagram below shows the format of a ieee80211 packet:



* The first 24 bytes of MAC Header of data packet are needed:
  + Address 4 field depends on FromDS and ToDS which is in Frame Control;
  + QoS Control field depends on Subtype which is in Frame Control;
  + HT Control field depends on Order Field which is in Frame Control;
  + More details are found in IEEE Std 80211-2012.
* For WEP packets, MAC Header is followed by 4 bytes IV and before FCS there are 4 bytes ICV.
* For TKIP packet, MAC Header is followed by 4 bytes IV and 4 bytes EIV, and before FCS there are 8 bytes MIC and 4 bytes ICV.
* For CCMP packet, MAC Header is followed by 8 bytes CCMP header, and before FCS there are 8 bytes MIC.

## ESP8266 ADC & VDD33

SARADC in ESP8266 provides the following two applications, but cannot be used at the same time:

1. Measure the power voltage of VDD3P3 pin 3 and 4：

The function to measure PA\_VDD pin’s voltage: uint16 system\_get\_vdd33(void)；

1. Measure the Input voltage of TOUT pin 6：

The function to measure the voltage of TOUT pin：uint16 system\_adc\_read(void). The described parameter of RF\_init is esp\_init\_data\_default.bin.

### Applications Scenario 1：Measure power voltage of VDD3P3 pin 3 and 4

**Hardware design:** TOUT pin must be suspended.

**RF\_init parameter：**

The 107th byte in esp\_init\_data\_default.bin（0〜～127byte）is “vdd33\_const“ , and it must be set as 0xFF， that is 255；

**RF Calibration work-in-progress：**

Optimize the operating of RF circuit, after testing the power voltage of VDD3P3 pin 3 and 4.

**User software**：system\_get\_vdd33 is available；system\_adc\_read is disabled.

### Applications Scenario 2：Input Voltage of TOUT pin 6

**Hardware design**: wire TOUT pin to external circuitry，Input Voltage Range restricted to 0~1.0V.

**RF\_init parameter**：

The 107th byte in esp\_init\_data\_default.bin（0〜～127byte）is“vdd33\_const”,and it must be set as the real power voltage of VDD3P3 pin 3 and 4.

The range of operating voltage of ESP8266 is 1.8V~3.6V，the unit of vdd33\_const is 0.1V，so effective value range of vdd33\_const is 18~36.

It will dynamically vary if the voltage is unstable, please input the Minimum\*10 of the voltage’s variation of vdd33\_const.

**RF Calibration work-in-progress**：

Optimize the operating of RF circuit, according to the value of 107th byte in“vdd33\_const”, the tolerance is +-0.2V.

**User software**：system\_get\_vdd33 is disable；system\_adc\_read is available.

### Notice

Notice 1 The parameter of RF\_ini, that is the 107th byte in esp\_init\_data\_default.bin（0〜～127）

should be named as vdd33\_const， and the explanations are as follows：

1. Optimize the operating of RF circuit, after testing the power voltage of VDD3P3 pin 3 and 4 inside ESP8266 RF Calibration, when vdd33\_const = 0xFF.
2. Optimize the operating of RF circuit with （vdd33\_const/10）inside ESP8266 RF Calibration,when 18 <= vdd33\_const <= 36.
3. Optimize the operating of RF circuit with the default value of 3.0V inside ESP8266 RF Calibration, when vdd33\_const < 18 or 36 < vdd33\_const < 255.

Notice 2 When measure the power voltage of VDD3P3 pin3 and 4 with function(system\_get\_vdd3) :

1. Tout pin must be suspended, the 107th byte (vdd33\_const) in RF\_init parameter must be equal to 0xFF.
2. Only the 107th byte (vdd33\_const) in RF\_init parameter is equal to 0xFF, can the return values from function(system\_get\_vdd33) are valid, or function(system\_get\_vdd33) returns to 0xFFFF.
3. The value returned is 1/1024 V.

Notice 3 When measure the voltage of Tout pin 6 with function(system\_adc\_read):

1. The 107th byte (vdd33\_const) in RF\_init parameter must be set as the real power voltage.
2. Only the 107th byte (vdd33\_const) in RF\_init parameter isn’t equal to0xFF, can the return values from function(system\_adc\_read) are valid; or function(system\_adc\_read) returns to 0xFFFF.
3. The value returned is 1/1024 V.

## ESP8266 soft-AP and station channel configuration

Even though ESP8266 can be in soft-AP + station mode，it actually has only one hardware channel.

So in soft-AP + station mode, ESP8266 soft-AP will adjust its channel configuration to be as same as ESP8266 station.

This limitation may cause some inconvenience in softAP + station mode users need to pay attention，for example:

Case 1.

1. If user connect ESP8266 station to a router(e.g. router is in channel 6)
2. Then set ESP8266 softAP by wifi\_softap\_set\_config
3. The API may return true, but channel will always be channel 6. Because we have only one hardware channel.

Case 2.

1. If user set ESP8266 softAP a channel number(e.g. channel 5) by wifi\_softap\_set\_config
2. Some stations connected to ESP8266 softAP.
3. Then connect ESP8266 station to a router of which channel number is different (e.g. channel 6) .
4. ESP8266 softAP has to adjust its channel to be as same as ESP8266 station , in this case, is channel 6.
5. So the stations that connected to ESP8266 softAP in step 2 will be disconnected because of the channel change.