ESP-TOUCH User Guide

Version 2.0

About This Guide

Release Notes

Date	Version	Release notes
2015.12	V1.0	First release.
2016.04	V1.1	Updated Chapter 2 and Chapter 3.
2018.06	V2.0	Updated Chapter 3 with the support for ESP32.

Documentation Change Notification

Espressif provides email notifications to keep customers updated on changes to technical documentation. Please subscribe *here*.

Certificates

Please download the product certificate(s) from *here*.

Table of Contents

1. Technology Overview

Espressif's ESP-TOUCH protocol implements the Smart Config technology to help users connect ESP8266EX- and ESP32-embedded devices (hereinafter referred to as the device) to a Wi-Fi network through simple configuration on a smartphone.

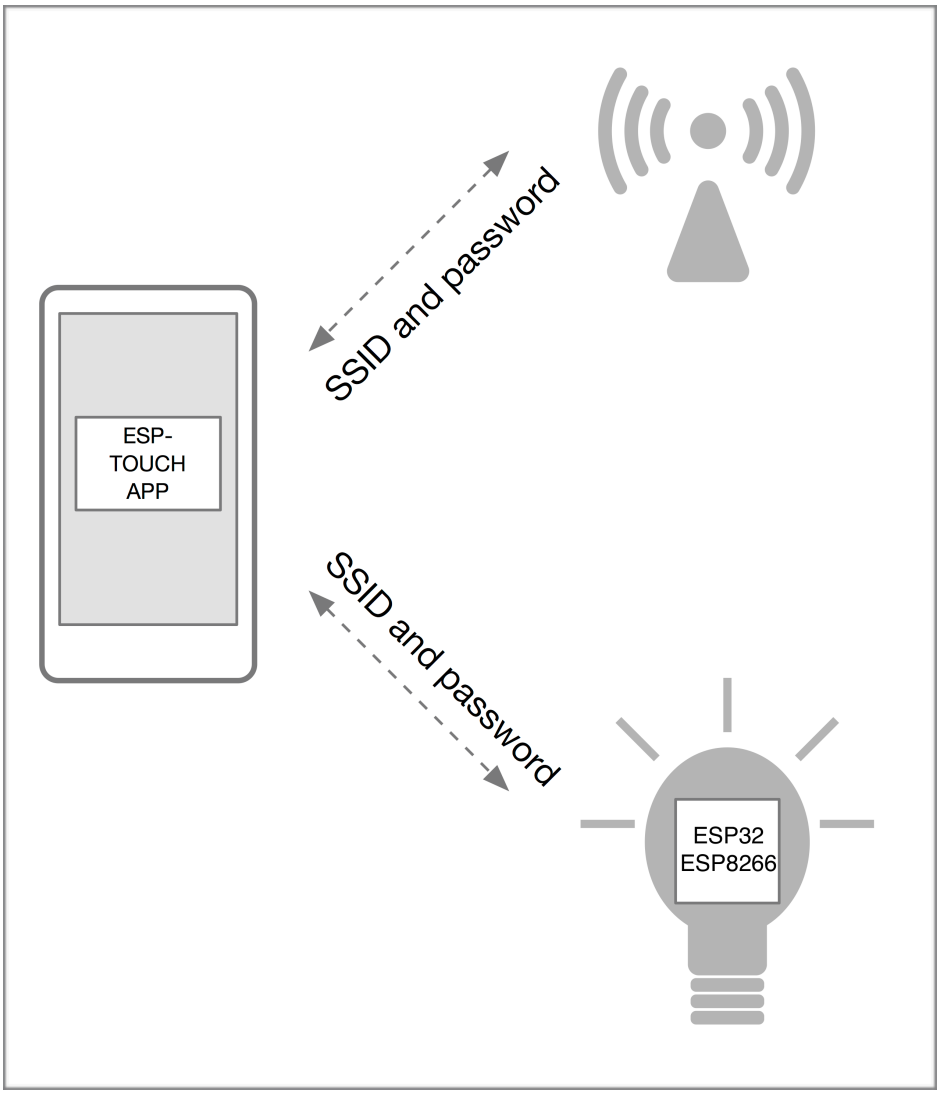

Figure 1-1 Typical ESP-TOUCH Application

Since the device is not connected to the network at the beginning, the ESP-TOUCH application cannot send any information to the device directly. With the ESP-TOUCH communication protocol, a device with Wi-Fi access capabilities, such as a smartphone, can send a series of UDP packets to the Wi-Fi Access Point (AP), encoding the SSID and password into the Length field of each of these UDP packets. The device can then reach the UDP packets, obtaining and parsing out the required information. The data packet structure is shown in Figure 1-2:

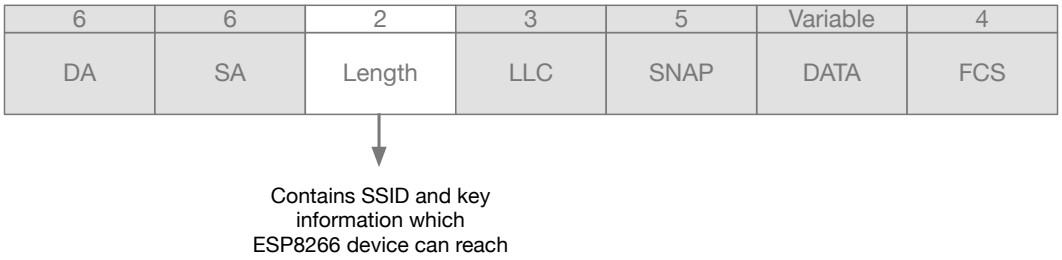

Figure 1-2 Data Packet Structure

2. ESP-TOUCH Operations

2.1. ESP-TOUCH Functional Overview

The ESP8266 OS SDK and NONOS SDK, as well as ESP -IDF, all support ESP-TOUCH.

The SDKs also integrate the AirKiss protocol developed by Wechat, so that users can configure the device either via the ESP-TOUCH App or on the Wechat client-side.

> 📖 ***Note:***
>
> *Users can download the ESP-TOUCH App source code from: https://github.com/espressifAPP.*

2.2. ESP-TOUCH Operation Process

1. Prepare a device that supports ESP-TOUCH, and enable its Smart Config function.
2. Connect your smartphone to the router.
3. Open the ESP-TOUCH App installed on the smartphone.
4. Input the router's SSID and password to connect the device to the router. Please, leave a blank in the password box if the router is not encrypted.

📖 ***Note:***

- *It only takes a few seconds for the device to connect to the router if the two are close. With greater distance between them, it will take longer to establish their connection.*
- *Make sure the router is powered on before configuration, so that the device is able to scan the APs around it.*
- *The series of data transmitted from the ESP-TOUCH App has a timeout limit. If the device cannot connect to the router within a specified period of time, the App will return a configuration failure message (for further details, please refer to the App source code). Similarly, the time that the device needs to obtain the SSID and password will be calculated. If the timeout limit is reached before the SSID and password information is obtained, the device will automatically start the next round of Smart Config process.Users can set the timeout limit through* `esptouch_set_timeout(uint8 time_s) or esp_smartconfig_set_timeout(uint8 time_s)`.
- *Sniffer mode should be enabled during the Smart Config process. Station and soft-AP modes of the device should be disabled. Other APIs should not be called. Station and soft-AP modes are allowed to be enabled at the same time for ESP32 during the Smart Config process.*
- *After the configuration process is completed, the transmitter will get the IP of the device, and the device will return the IP of the transmitter. If the user wants to customize the information exchange between the transmitter and the device, the IP information can be used for performing LAN communication.*
- *If the AP isolation mode is enabled on the router, the App may not get a configuration success message, even if the connection has been established.*
- *Users can configure multiple devices to connect to the same router simultaneously by configuring the app to receive multiple return values.*
- *ESP8266 and ESP32 do not support 5G and 11AC modes. As a result, Smart Config is not supported by routers using 5G or 11AC mode.*

3. API Development

Users can call the following APIs to realize the ESP-TOUCH configuration. Please, use the latest App and firmware. The SDKs and ESP-IDF provide an ESP-TOUCH demo for your reference.

3.1. ESP8266 API

3.1.1. smartconfig_start

Function: configure the device and connect it to the AP.

> ⚠ ***Notice:***
>
> - *This API can be called in the Station mode only.*
> - *Call* `smartconfig_stop` *to stop the Smart Config process,before repeating it or calling other APIs.*

Defintion: bool smartconfig_start(sc_callback_t cb, uint8 log)

Parameters:

`sc_callback_t cb`	The Smart Config callback is executed when the smart-config status changes. The parameter status of this callback shows the status of Smart Config: • When the status is `SC_STATUS_GETTING_SSID_PSWD`, parameter void *pdata is a pointer of sc_type, meaning that the Smart Config type is AirKiss or ESP-TOUCH. • When the status is `SC_STATUS_LINK`, parameter void *pdata is a pointer of struct station_config. • When the status is `SC_STATUS_LINK_OVER`, parameter void *pdata is a pointer of mobile IP address (4 bytes). It applies to the ESP-TOUCH configuration process only; otherwise, parameter void *pdata will be NULL. • When the status is otherwise, parameter void *pdata is NULL.
uint8 log	"1" indicates that the connection process will be printed out via the UART interface. Otherwise, only the connection result will be printed. For example: • `smartconfig_start(smartconfig_done,1)`: DEBUG information during the connection process will be printed out via the serial port. • `smartconfig_start(smartconfig_done)`: DEBUG information will not be printed out; only the connection result will be printed.

Returned Value:

TRUE	Succeeded
FALSE	Failed

Example:

```
void ICACHE_FLASH_ATTR
smartconfig_done(sc_status status, void *pdata)
{
    switch(status) {
        case SC_STATUS_WAIT:
            os_printf("SC_STATUS_WAIT\n");
            break;
        case SC_STATUS_FIND_CHANNEL:
            os_printf("SC_STATUS_FIND_CHANNEL\n");
            break;
        case SC_STATUS_GETTING_SSID_PSWD:
            os_printf("SC_STATUS_GETTING_SSID_PSWD\n");
            sc_type *type = pdata;
            if (*type == SC_TYPE_ESPTOUCH) {
                os_printf("SC_TYPE:SC_TYPE_ESPTOUCH\n");
            } else {
                os_printf("SC_TYPE:SC_TYPE_AIRKISS\n");
            }
            break;
        case SC_STATUS_LINK:
            os_printf("SC_STATUS_LINK\n");
            struct station_config *sta_conf = pdata;
            wifi_station_set_config(sta_conf);
            wifi_station_disconnect();
            wifi_station_connect();
            break;
        case SC_STATUS_LINK_OVER:
            os_printf("SC_STATUS_LINK_OVER\n");
            if (pdata != NULL) {
                uint8 phone_ip[4] = {0};
                memcpy(phone_ip, (uint8*)pdata, 4);
                os_printf("Phone ip: %d.%d.%d.
                    %d\n",phone_ip[0],phone_ip[1],phone_ip[2],phone_ip[3]);
            }
            smartconfig_stop();
```

```
                break;
            }
        }
        smartconfig_start(smartconfig_done);
```

3.1.2. smartconfig_stop

Function: stop the Smart Config process, and free the buffer taken by `smartconfig_start`.

> 📖 ***Note:***
>
> *After the connection has been established, users can call this API to free the memory taken by* `smartconfig_start`.

Defintion: bool smartconfig_stop(void)

Parameters: Null

Returned Value:

TRUE	Succeeded
FALSE	Failed

3.1.3. smartconfig_set_type

Function: set the protocol type used in `smartconfig_start` mode.

> 📖 ***Note:***
>
> *Please call this API before* `smartconfig_start`。

Definition: bool smartconfig_set_type(sc_type type)

Parameters:

```
    typedef enum {
        SC_TYPE_ESPTOUCH = 0,
        SC_TYPE_AIRKISS,
        SC_TYPE_ESPTOUCH_AIRKISS,
    } sc_type;
```

Returned Value:

TRUE	Succeeded
FALSE	Failed

3.1.4. Struct

```
        typedef enum {
```

```
    SC_STATUS_WAIT = 0,
    SC_STATUS_FIND_CHANNEL = 0,
    SC_STATUS_GETTING_SSID_PSWD,
    SC_STATUS_LINK,
    SC_STATUS_LINK_OVER,
} sc_status;
```

> ⚠ ***Notice:***
>
> `SC_STATUS_FIND_CHANNEL` *status: Users may open the App for configuration only when the device starts scanning the channels.*

```
typedef enum {
    SC_TYPE_ESPTOUCH = 0,
    SC_TYPE_AIRKISS,
    SC_TYPE_ESPTOUCH_AIRKISS,
} sc_type;
```

3.2. ESP32 API

3.2.1. esp_smartconfig_start

Function: configure the device so that it connects to the AP.

> ⚠ ***Notice:***
>
> - *This API can be called in the Station mode or Station + soft-AP mode.*
> - *Call* `esp_smartconfig_stop` *to stop the Smart Config process before repeating* `esp_smartconfig_start` *or calling other APIs.*

Defintion: esp_err_t esp_smartconfig_start(sc_callback_t cb, ...)

Parameters:

`sc_callback_t cb`	The Smart Config callback is executed when the smart-config status changes. The parameter status of this callback shows the status of Smart Config: • When the status is `SC_STATUS_GETTING_SSID_PSWD`, parameter void *pdata is a pointer of `smartconfig_type_t`, meaning that the Smart Config type is AirKiss or ESP-TOUCH. • When the status is SC_STATUS_LINK, parameter void *pdata is a pointer of `wifi_config_t`. • When the status is `SC_STATUS_LINK_OVER`, parameter void *pdata is the pointer of a mobile IP address (4 bytes). It applies to the ESP-TOUCH configuration process only; otherwise, parameter void *pdata will be NULL. • When the status is otherwise, parameter void *pdata is NULL.
`...`	"1" indicates that the connection process will be printed out via the UART interface. Otherwise, only the connection result will be printed. For example, • `esp_smartconfig_start(smartconfig_done,1)`: DEBUG information during the connection process will be printed out via the serial port. • `esp_smartconfig_start(smartconfig_done)` DEBUG information will not be printed out; only the connection result will be printed.

Returned Value:

`ESP_OK`	Succeeded
`others`	Failed

Example:

```
void sc_callback(smartconfig_status_t status, void *pdata)
{
    switch(status) {
        case SC_STATUS_WAIT:
            printf("SC_STATUS_WAIT\n");
            break;
        case SC_STATUS_FIND_CHANNEL:
            printf("SC_STATUS_FIND_CHANNEL\n");
            break;
        case SC_STATUS_GETTING_SSID_PSWD:
            printf("SC_STATUS_GETTING_SSID_PSWD\n");
            smartconfig_type_t *type = pdata;
            if (*type == SC_TYPE_ESPTOUCH) {
```

```
                printf("SC_TYPE:SC_TYPE_ESPTOUCH\n");
            } else {
                printf("SC_TYPE:SC_TYPE_AIRKISS\n");
            }
            break;
        case SC_STATUS_LINK:
            printf("SC_STATUS_LINK\n");
            wifi_config_t *wifi_conf = pdata;
            esp_wifi_disconnect();
            esp_wifi_set_config(ESP_IF_WIFI_STA, wifi_conf);
            esp_wifi_connect();
            break;
        case SC_STATUS_LINK_OVER:
            printf("SC_STATUS_LINK_OVER\n");
            if (pdata != NULL) {
                uint8 phone_ip[4] = {0};
                memcpy(phone_ip, (uint8*)pdata, 4);
                printf("Phone ip: %d.%d.%d.
                    %d\n",phone_ip[0],phone_ip[1],phone_ip[2],phone_ip[3]);
            }
            esp_smartconfig_stop();
            break;
    }
}
esp_smartconfig_start(sc_callback);
```

3.2.2. esp_smartconfig_stop

Function: stop the Smart Config process, and free the buffer taken by `esp_smartconfig_start`.

> 📖 ***Note:***
>
> *After the connection has been established, users can call this API to free the memory.*

Defintion: esp_err_t esp_smartconfig_stop(void)

Parameters: Null

Returned Value:

`ESP_OK`	Succeed
`others`	Failed

3.2.3. esp_smartconfig_set_timeout

Function: set a timeout limit to `Smart Config`.

> 📖 ***Note:***
>
> - *Call this API first before calling* `esp_smartconfig_start`.
> - *The timeout limit is calculated by getting the state of* `SC_STATUS_FIND_CHANNEL`*, and Smart Config will be restarted after the timeout limit is reached.*

Defintion: esp_err_t esp_smartconfig_set_timeout(uint8_t time_s)

Parameters:

time_s	time-out period, ranging from 15 to 255s.

Returned Value:

`ESP_OK`	Succeed
`others`	Failed

3.2.4. esp_smartconfig_set_type

Function: set the protocol type used in `esp_smartconfig_start` mode.

> 📖 ***Note:***
>
> *Call this API first before calling* `esp_smartconfig_start`.

Defintion: esp_err_t esp_smartconfig_set_type(smartconfig_type_t type)

Parameters:

```
typedef enum {
    SC_TYPE_ESPTOUCH = 0,
    SC_TYPE_AIRKISS,
    SC_TYPE_ESPTOUCH_AIRKISS,
} smartconfig_type_t;
```

Returned Value:

`ESP_OK`	Succeed
`others`	Failed

3.2.5. esp_smartconfig_fast_mode

Function: enable/disable the Smart Config fast mode.

> 📖 ***Note:***
>
> - *Call this API first before calling* `esp_smartconfig_start`*;*
> - *To make the fast mode fully functional, a corresponding App is required.*

Definition: esp_err_t esp_smartconfig_fast_mode(bool enable)

Note:

enable	True: enable fast mode; False: disable fast mode.

Returned Value:

`ESP_OK`	Succeed
`others`	Failed

3.2.6. Type

Typedef void (*sc_callback_t)(smartconfig_status_t status, void *pdata)

Function: input the callback of esp_smartconfig_start.

Parameters:

smartconfig_status_t	Status of the device.
Void*	• When the status is `SC_STATUS_GETTING_SSID_PSWD`, parameter void *pdata is a pointer of `smartconfig_type_t`, meaning that the Smart Config type is AirKiss or ESP-TOUCH. • When the status is `SC_STATUS_LINK`, parameter void *pdata is the pointer of a `wifi_config_t`. • When the status is `SC_STATUS_LINK_OVER`, parameter void *pdata is a pointer of mobile IP address (4 bytes). It applies to the ESP-TOUCH configuration process only ; otherwise, parameter void *pdata will be NULL. • When the status is otherwise, parameter void *pdata is NULL.

3.2.7. Struct

```
typedef enum {
    SC_STATUS_WAIT = 0,
    SC_STATUS_FIND_CHANNEL = 0,
    SC_STATUS_GETTING_SSID_PSWD,
    SC_STATUS_LINK,
    SC_STATUS_LINK_OVER,
} smartconfig_status_t;
```

> ⚠ ***Notice:***
>
> `SC_STATUS_FIND_CHANNEL` *status: users may open the App for configuration only when the device starts scanning the channels.*

```
typedef enum {
    SC_TYPE_ESPTOUCH = 0,
    SC_TYPE_AIRKISS,
    SC_TYPE_ESPTOUCH_AIRKISS,
} smartconfig_type_t;
```

4. ESP-TOUCH Performance Analysis

The communication model on which the ESP-TOUCH technology is based can be understood as a unidirectional channel with a certain error rate. However, this error rate differs depending on the bandwidth. The packet error rate for a 20 MHz bandwidth is 0 - 5%, and the packet error rate for a 40 MHz bandwidth is 0 - 17%. Assuming that the maximum length of data to be transmitted is 104 bytes, if an error-correcting algorithm is not used, it is difficult to ensure that the data can be transmitted over limited rounds of data transfer.

ESP-TOUCH uses a cumulative error-correcting algorithm to achieve completing the transmission process over limited rounds of data transfer. The theoretical basis of the cumulative error-correcting algorithm is that the probability of an error occurring on the same bit of data during multiple rounds of data transmission is very low. Therefore, it is possible to accumulate results of multiple rounds of data transfer, where there is great potential for one bit of erroneous data in one round to lead to the correct corresponding value in other rounds, thus ensuring the completion of data transfer within a limited amount of time.

The success rate of data transmission can be generalized as $[1-[1-p]^k]^l$ (p : packet success rate, k : round of transmission, l : length of transmitted data).

Assuming that the length of data to be transmitted is 104 bytes and 72 bytes, the success rate can reach 0.95 when the bandwidth is 20 MHz, and 0.83 when the bandwidth is 40 MHz.

The tables below show the probability of the data transmission success rate and the transmission time when the cumulative error-correcting algorithm is adopted.

Table 4-1. ESP-TOUCH Error Correcting Analysis (20 MHz Bandwidth)

Round	Length: 104 Bytes		Length: 72 Bytes	
	Transmission Time (s)	Success Rate	Transmission Time (s)	Success Rate
1	4.68	0.0048	3.24	0.0249
2	9.36	0.771	6.48	0.835
3	14.04	0.987	9.72	0.991
4	18.72	0.9994	12.9	0.9996
5	23.40	0.99997	16.2	0.99998
6	28.08	0.999998	19.4	0.99999

Table 4-2 ESP-TOUCH Error Correcting Analysis (40 MHz Bandwidth)

Round	Length: 104 Bytes		Length: 72 Bytes	
	Transmission Time (s)	Success Rate	Transmission Time (s)	Success Rate
1	4.68	3.84e-9	3.24	1.49e-6
2	9.36	0.0474	6.48	0.121
3	14.04	0.599	9.72	0.701
4	18.72	0.917	12.9	0.942
5	23.40	0.985	16.2	0.989
6	28.08	0.997	19.4	0.998

Espressif IoT Team

www.espressif.com