Espressif IOT Demo

Smart Light/Plug/Sensor

Version 1.3

Espressif Systems IOT Team

Disclaimer and Copyright Notice

Table of Content

1. Preambles

Herein, we introduce a series of embedded applications based on Espressif's SoC with WiFi connectivity ESP8266. The IoT demo applications here showcase how you can develop an entire application with this single SoC, and to realize smart connectivity for three kinds of products: smart power plugs, smart lights, and humidity-temperature sensors. In addition, our Espressif cloud server enables the reverse control of devices and data collection.

With the IoT demo, users can rapidly develop a variety of similar applications.

2. Overview

The SDK provides a set of interfaces for data receive and transmit functions over the WiFi and TCP/IP layer so programmers can focus on application development on the high level. Users can easily make use of the corresponding interfaces to realize data receive and transmit.

All networking functions on the ESP8266 IoT platform are realized in the library, and are not transparent to users. Instead, users can initialize the interface in `user_main.c`.

`void user_init(void)` is the default method provided. Users can add functions like firmware initialization, network parameters setting, and timer initialization in the interface.

The SDK provides APIs to handle JSON, and users can also use self-defined data types to handle the them.

Folder `example` in `esp_iot_sdk` contains three application demos: `IOT_Demo`, `AT`, `smart_config`. In fact, all APIs provided by SDK can be called in application, for example, users can call `smart_config` APIs in `IOT_Demo`.

2.1. Source Code Layout

1. Folder "usr"

The IoT demo source code of is in the "usr" folder, and the details are as follows:

`user_main.c` – main file

`user_webserver.c` – creates a TCP server, provides REST light weighted webserver function

`user_devicefind.c` – creates a UDP transmission, provides device look-up function

`user_esp_platform.c` – communicate with Espressif Cloud

`user_json.c` – json packet processing function

`user_plug.c` – demo of plug device

`user_light.c` – demo of PWM light device

`user_humiture.c` – demo of humidity-temperature sensor device

2. Folder "include"

Header files are stored in the include folder. The "user_config.h" define variables to select a demo type to compile. Please only enable one device definition when compile.

Example:

```
PLUG_DEVICE, LIGHT_DEVICE, SENSOR_DEVICE

SENSOR_DEVICE can be sub-divided into HUMITURE_SUB_DEVICE and FLAMMABLE_GAS_SUB_DEVICE.
```

Definition below shows the flash user parameter area, users need to adjust it according to flash map which selected during compilation.

```
In user_esp_platform.h
      #define ESP_PARAM_START_SEC  0x3D   // or 0x7D, or 0xFD
For LIGHT_DEVICE, in user_light.h
      #define PRIV_PARAM_START_SEC  0x3C  // or 0x7C, or 0xFC
For PLUG_DEVICE, in user_plug.h
      #define PRIV_PARAM_START_SEC  0x3C  // or 0x7C, or 0xFC
```

More details about flash-map in documentation "2A-ESP8266__IOT_SDK_User_Manual"

3. Folder "driver"

Our divers currently supports I2C Master, SPI, external buttons, PWM and Dual UART.

4. Operating Mode

The WiFi operating mode in this IoT Demo is softAP + station -> station. It is set to softfAP + station mode in `user_esp_platform_init` . Users can connect to the LAN via the softAP interface and send commands to connect to the router via the station interface. Through the softAP interface, the user can query the station connection status. After connection to the router is successful, the device can be set to station mode. (For more details, please refer to *3.1.2 Setting Connection Parameters*)

By default, the SSID for softAP is **ESP_XXXXXX**, where **XXXXXX** are the last three bytes of the MAC address. The default encryption mode is WPA/WPA2.

In station mode, by pressing the button for 5 seconds, the device will be reset and restarted to initial softAP + station mode for reconfiguration.

2.2. Debugging Tools

The Cloud Server in the IoT Demo uses the REST architecture. When the PC is communicating with the IoT Demo device, curl commands can be used.

Download the specified version on http://curl.haxx.se/download.html . For use of curl commands here, please refer to the examples shown in "Windows curl".

If you use Linux curl or Cygwin curl, please refer to examples of the curl command in "`Linux/Cygwin curl`".

Unless otherwise specified, both can be used.

1. Common errors in Curl

Take note of the upper and lower case in the commands, otherwise the command may fail.

The number of spaces curl commands may cause it to fail.

Token generated on the Server is only for 1 device and cannot be shared.

Use the right command format for Linux/Cygwin or Windows Curl. Do not mix them up.

3. Functions for LAN

Default IP address of softAP mode is 192.168.4.1. In station mode the IP address is assigned by router. The IP address in the URL represents IP in softAP and station mode depending on which is required.

Espressif softAP needs a password to connect, it depends on `#define SOFTAP_ENCRYPT`.

The password format is as follows:

```
device's_softAPMAC_PASSWORD.
```

The password defined in `esp_iot_sdk/app/include/user_config.h` is the softAP's PASSWORD.

For example:

The "PASSWORD" defined in esp_iot_sdk_v.08 is `v*%W>L<@i&Nxe!`

The softAP MAC address of a device is `1a:fe:86:90:d5:7b`

So the connection password is `1a:fe:86:90:d5:7b_v*%W>L<@i&Nxe!`

3.1. General Functions

1. Get version information

```
curl –X GET http://ip/client?command=info
Response:
{
        "Version": {
                "hardware":"0.1",
                "software":"0.8.0"
        },
        "Device": {
                "product":"Plug",
                "manufacture",  "Espressif Systems"
        }
}
```

2. Set connection parameter

Device initial state is softAP+station mode. Connect PC to LAN provided by device softAP's interface (password described above) and send curl to control device through PC.

Set station mode

PCs send following command and connect device to the external network.

Linux/Cygwin curl:

```
curl -X POST -H "Content-Type:application/json" -d '{"Request":{"Station":
{"Connect_Station":{"SSID":"tenda", "password":"1234567890", "token":
"1234567890123456789012345678901234567890"}}}}' http://192.168.4.1/config?
command=wifi
```

Windows curl:

```
curl -X POST -H "Content-Type:application/json" -d "{\"Request\":{\"Station\":
{\"Connect_Station\":{\"ssid\":\"tenda\",\"password\":\"1234567890\",\"token\":
\"1234567890123456789012345678901234567890\"}}}}" http://192.168.4.1/config?
command=wifi
```

After the setup is completed, connect the device to the router in the command.

Note:

The red token length field is a random hexademical string of 40 bytes. Device will send this random token to server for activation and identification

Users use the same random token to apply for control access to the device from Espressif Cloud, so random token cannot be shared with other devices.

PS:

If router(AP) config is WEP HEX , the password need to convert into HEX in curl.

For example:

SSID of router is "wifi_1", the password is"tdr0123456789", encrypt as "WEP HEX", then

Linux/Cygwin curl:

```
curl -X POST -H Content-Type:application/json -d '{"Request":{"Station":
{"Connect_Station":{"ssid":"wifi_1", "password":"7464723031323333435363738 39",
"token": "1234567890123456789012345678901234567890"}}}}' http://192.168.4.1/config?
command=wifi
```

Windows curl:

```
curl -X POST -H "Content-Type:application/json" -d "{\"Request\":{\"Station\":
{\"Connect_Station\":{\"ssid\":\"wifi_1\",\"password\":\"7464723031323333435363738 39
\" , \"token\": \"1234567890123456789012345678901234567890\"}}}}" http://
192.168.4.1/config?command=wifi
```

During the connection, the command below can be sent through PC side to query the connection status of device

```
curl -X GET http://ip/client?command=status
```

Response:

```
enum {
    STATION_IDLE = 0,
    STATION_CONNECTING,
    STATION_WRONG_PASSWORD,
    STATION_NO_AP_FOUND,
    STATION_CONNECT_FAIL,
    STATION_GOT_IP
};
```

```
enum {
    DEVICE_CONNECTING = 40,
    DEVICE_ACTIVE_DONE,
    DEVICE_ACTIVE_FAIL,
    DEVICE_CONNECT_SERVER_FAIL
};
```

After connecting the PC side will send following command and change the device mode from softAP +station mode to station mode.

For the devices which support reverse control, such as switches, lights and so on, command is:

```
curl -X POST http://ip/config?command=reboot
```

For the devices which do not support reverse control, such as sensors, the command is:

```
curl -X POST http://ip/config?command=sleep
```

The sensor devices will wake up automatically after sleep 30s and mode will be changed to station mode.

Set softAP parameters

Devices send following command and set parameters for softAP, such as SSID, password and so on.

Linux/Cygwin curl:

```
curl -X POST -H "Content-Type:application/json" -d '{"Request":{"Softap":
{"Connect_Softap":{"authmode":"OPEN", "channel":6, "ssid":"ESP_IOT_SOFTAP",
"password":""}}}}' http://192.168.4.1/config?command=wifi
```

Windows curl:

```
curl -X POST -H "Content-Type:application/json" -d "{\"Request\":{\"Softap\":
{\"Connect_Softap\":{\"authmode\":\"OPEN\",\"channel\":6,\"ssid\":\"ESP_IOT_SOFTAP
\",\"password\":\"\"}}}}" http://192.168.4.1/config?command=wifi
```

The devices need to be rebooted before changes take effect.

Note:

authmode supports following modes: OPEN, WPAPSK, WPA2PSK, WPAPSK/WPA2PSK.

password must be no less than 8 bytes.

3. Transformation of wifi mode

Since esp_iot_sdk_v0.9.2, transformation of wifi mode is as follows:

- Device initial state is softAP+station mode.
- Phone APP（or PC）connects to ESP8266 softAP，sends a command to make ESP8266 station connect to router（AP）.
- Connected to router（AP），ESP8266 tries to communicate with server for authentication. If succeed，ESP8266 changes to station mode.
- After that, ESP8266 is in station mode, only if it is disconnected from the network, ESP8266 will convert into softAP + station mode again. Then restart from step (2), try to connect again.
- If fail to connect to router(AP), ESP8266 will try another router(AP) which has been recorded. This function in source code defines as "#define AP_CACHE".

3.1. Device search in LAN

Find devices by sending UDP broadcast packets to port 1025 in the LAN. The message sent is "Are You Espressif IOT Smart Device?". Devices will respond to the received UDP broadcast packets with a string.

This function can be tested by using the network debugging assistant, for example:

Response:

- **Plug**

I'm Plug.xx:xx:xx:xx:xx:xxyyy.yyy.yyy.yyy

- **Light**

I'm Light.xx:xx:xx:xx:xx:xxyyy.yyy.yyy.yyy

- **Humidity and Temperature Sensor**

I'm Humiture.xx:xx:xx:xx:xx:xxyyy.yyy.yyy.yyy

Where xx:xx:xx:xx:xx:xx is the device MAC address and yyy.yyy.yyy.yyy is the device IP address. There is no response for the wrong string.

3.2. Plug

1. Get Status

```
curl –X GET http://ip/config?command=switch
```

Response

```
{
    "Response": {
        "status": 0
    }
}
```

Status can be 0 or 1.

2. Set Status

Linux/Cygwin curl:

```
curl -X POST -H "Content-Type:application/json" -d '{"Response":{"status":1}}'
http://ip/config?command=switch
```

Windows curl:

```
curl -X POST -H "Content-Type:application/json" -d "{/"Response/":{/"status/":1}}"
http://ip/config?command=switch
```

status can be 0 or 1.

3.3. Light

1. Get Status

```
curl -X GET http://ip/config?command=light
```

Response:

```
{
    "freq": 100,
    "rgb": {
        "red": 100,
        "green": 0,
        "blue": 0
    }
}
```

Range: freq can be 1~500 while red, green, blue can be 0~255.

2. Set Status

Linux/Cygwin curl:

```
curl -X POST -H "Content-Type:application/json" -d '{"freq":100, "rgb":{"red":200,
"green":0, "blue":0}}' http://ip/config?command=light
```

Windows curl:

```
curl -X POST -H "Content-Type:application/json" -d "{\"freq\":100,\"rgb\":{\"red\":
200,\"green\":0,\"blue\":0}}" http://ip/config?command=light
```

Range: freq can be 1~500 while red, green, blue can be 0~255.

3.4. Humidity-Temperature Sensor

The humidity-temperature sensor status need to be obtained from Espressif Cloud Server through internet.

4. Functions for WAN

4.1. Espressif Cloud Server

Specifications and details of the Espressif Cloud Server, including introduction of the APIs can be found on the server itself.

Note:

- "Device " refers to the operation which that the device runs by itself and needs no user intervention.
- "PC" refers to the commands that users can send to the device to run.

master-device-key

The Espressif Cloud server is designed such that each device needs to apply for a master device key from it and burned to SPI flash.

Please refer to the document Espressif Cloud Introduction.

Activation

Device

After setting the SSID, password and random token through softAP interface, the device's station interface will connect to router for activation. Once it gets IP address from router, it will try to connect to the server automatically for activation.

Activation requires a TCP packet to be send to the server (IP address 114.215.177.97, port 8000). Format of TCP packet:

```
{"path": "/v1/device/activate/", "method": "POST", "meta": {"Authorization":
"token HERE_IS_THE_MASTER_DEVICE_KEY"}, "body": {"encrypt_method": "PLAIN",
"bSSID": "18:fe:34:70:12:00", "token": "1234567890123456789012345678901234567890"}}
```

HERE_IS_THE_MASTER_DEVICE_KEY is the device key stored in the SPI flash, and 1234567890123456789012345678901234567890 is the random token set in above section 3.1.2 set connection parameter.

Response

```
{"status": 200, "device": {device}, "key": {key}, "token": {token}}
```

PC

After PC has configured the device's SSID, password and token, it needs to be connected to a router which has access to the internet and applies for device control from server.

Linux/Cygwin curl:

```
curl -X POST -H "Authorization:token c8922638bb6ec4c18fcf3e44ce9955f19fa3ba12" -d
'{"token": "12345678901234567890123456789012345678901234567890"}' http://iot.espressif.cn/v1/
key/authorize/
```

Windows curl:

```
curl -X POST -H "Authorization:token c8922638bb6ec4c18fcf3e44ce9955f19fa3ba12" -d
"{\"token\": \"12345678901234567890123456789012345678901234567890\"}" http://iot.espressif.cn/
v1/key/authorize/
```

Response:

```
{"status": 200, "key": {"updated": "2014-05-12 21:22:03", "user_id": 1,
"product_id": 0, "name": "device activate share token", "created": "2014-05-12
21:22:03", "source_ip": "*", "visibly": 1, "id": 149, "datastream_tmpl_id": 0,
"token": "e474bba4b8e11b97b91019e61b7a018cdbaa3246", "access_methods": "*",
"is_owner_key": 1, "scope": 3, "device_id": 29, "activate_status": 1,
"datastream_id": 0, "expired_at": "2288-02-22 20:31:47"}}
```

c8922638bb6ec4c18fcf3e44ce9955f19fa3ba12 is an example of user key, and users do need to fill in their own user key which can be obtained as follows:

- Log into Espressif server http://iot.espressif.cn/
- Sign in with username password
- Click on Username at the top corner ->Set-up ->Developer.

e474bba4b8e11b97b91019e61b7a018cdbaa3246 is the device's owner key which will be used later to control the device at PC end.

1. Identification

After activation the device needs to send TCP packets to Espressif Cloud Server (IP address 115.29.202.58, port 8000). TCP packet format:

```
{"nonce": 560192812, "path": "/v1/device/identify", "method": "GET", "meta":
{"Authorization": "token HERE_IS_THE_MASTER_DEVICE_KEY"}}
```

The function of this tcp packet is to help the device to confirm its identity. Each time the device reconnects to server, it should send such a packet. "nonce" is a batch of random numbers, the string behind token is the master device key.

The server replies to the device with when its identity is successfully confirmed with a data packet:

Response:

```
{"device": {"productbatch_id": 0, "last_active": "2014-06-19 10:06:58", "ptype":
12335, "activate_status": 1, "serial": "334a8481", "id": 130, "bSSID": "18:fe:
34:97:d5:33", "last_pull": "2014-06-19 10:06:58", "last_push": "2014-06-19
10:06:58", "location": "", "metadata": "18:fe:34:97:d5:33 temperature", "status":
2, "updated": "2014-06-19 10:06:58", "description": "device-description-79eba060",
"activated_at": "2014-06-19 10:06:58", "visibly": 1, "is_private": 1,
```

```
"product_id": 1, "name": "device-name-79eba060", "created": "2014-05-28 17:43:29",
"is_frozen": 0, "key_id": 387}, "nonce": 560192812, "message": "device
identified", "status": 200}
```

The identification process is required for plugs and lights application.

2. PING Server

To maintain the connection of socket under the circumstance that the device doesn't need reverse control over a long period of time, the device needs to send TCP packets to Espressif Cloud Server (IP address 115.29.202.58, port 8000) every 50s in following format:

```
{"path": "/v1/ping/", "method": "POST", "meta": {"Authorization": "token
HERE_IS_THE_MASTER_DEVICE_KEY"}}
```

Response:

```
{"status": 200, "message": "ping success", "datetime": "2014-06-19 09:32:28",
"nonce": 977346588}
```

PING is required in devices that need reverse control like plugs and lights.

3. Plug

Device

During reverse control of devices, there are two cases:

- Device receives get command from the server which indicates that the device needs to send his own status to the server. Format of device's get command sent from server:

```
{"body": {}, "nonce": 33377242, "is_query_device": true, "get": {}, "token":
"e474bba4b8e11b97b91019e61b7a018cdbaa3246", "meta": {"Authorization": "token
e474bba4b8e11b97b91019e61b7a018cdbaa3246"}, "path": "/v1/datastreams/plug-status/
datapoint/", "post": {}, "method": "GET"}
```

Response:

```
{"status": 200, "datapoint": {"x": 0}, "nonce": 33377242, "is_query_device":
true}
```

- When device receives post command from the server, it indicates that the device needs to change his own status and server will send data packets for instructions. For example, "turn on the switch" command:

```
{"body": {"datapoint": {"x": 1}}, "nonce": 620580862, "is_query_device": true,
"get": {}, "token": "e474bba4b8e11b97b91019e61b7a018cdbaa3246", "meta":
{"Authorization": "token e474bba4b8e11b97b91019e61b7a018cdbaa3246"}, "path": "/v1/
datastreams/plug-status/datapoint/", "post": {}, "method": "POST",
"deliver_to_device": true}
```

After the switch completes instruction, it will send a successful status update response to the server in the following format. The "nonce" used to sync request and response must be in consistent to the "nonce" in the control command previously sent from the server which represents that each control and response correspond to each other.

Response:

```
{"status": 200, "datapoint": {"x": 1}, "nonce": 620580862, "deliver_to_device":
true}
```

PC

Get plug status

```
curl -X GET -H "Content-Type:application/json" -H "Authorization: token
e474bba4b8e11b97b91019e61b7a018cdbaa3246" http://iot.espressif.cn/v1/datastreams/
plug-status/datapoint/
```

Response:

```
{"status": 200, "nonce": 11432809, "datapoint": {"x": 1}, "deliver_to_device":
true}
```

Set plug status

Linux/Cygwin curl:

```
curl -X POST -H "Content-Type:application/json" -H "Authorization: token
e474bba4b8e11b97b91019e61b7a018cdbaa3246" -d '{"datapoint":{"x":1}}' http://
iot.espressif.cn/v1/datastreams/plug-status/datapoint/?deliver_to_device=true
```

Windows curl:

```
curl -X POST -H "Content-Type:application/json" -H "Authorization: token
e474bba4b8e11b97b91019e61b7a018cdbaa3246" -d "{\"datapoint\":{\"x\":1}}" http://
iot.espressif.cn/v1/datastreams/plug-status/datapoint/?deliver_to_device=true
```

Response:

```
{"status": 200, "nonce": 11432809, "datapoint": {"x": 1}, "deliver_to_device":
true}
```

4. Light

Device

When handling reverse control of devices, there are two cases:

When device receives get command from the server, it needs to send its own status to the server. The get command format as sent from server to device is as follows:

```
{"body": {}, "nonce": 8968711, "is_query_device": true, "get": {}, "token":
"e474bba4b8e11b97b91019e61b7a018cdbaa3246", "meta": {"Authorization": "token
e474bba4b8e11b97b91019e61b7a018cdbaa3246"}, "path": "/v1/datastreams/light/
datapoint/", "post": {}, "method": "GET"}
```

Response:

```
{"nonce": 5619936, "datapoint": {"x": 100, "y": 200, "z": 0, "k": 0, "l": 50},
"deliver_to_device": true}
```

When device receives post command from the server, it needs to change its own status and complete instructions according to data packets from the server. The example given here is the switch on lights command:

```
{"body": {"datapoint": {"y": 200, "x": 100, "k": 0, "z": 0, "l": 50}}, "nonce":
5619936, "is_query_device": true, "get": {}, "token":
"e474bba4b8e11b97b91019e61b7a018cdbaa3246", "meta": {"Authorization": "token
e474bba4b8e11b97b91019e61b7a018cdbaa3246"}, "path": "/v1/datastreams/light/
datapoint/", "post": {}, "method": "POST"
```

Response:

```
{"nonce": 5619936, "datapoint": {"x": 100, "y": 200, "z": 0, "k": 0, "l": 50},
"deliver_to_device": true}
```

X = frequency, range 1~500. Y (red), Z (green), K (blue) indicate that the different colours of lights, range 0~255. L parameters are reserved.

PC

Get light status

```
curl -X GET -H "Content-Type:application/json" -H "Authorization: token
e474bba4b8e11b97b91019e61b7a018cdbaa3246" http://iot.espressif.cn/v1/datastreams/
light/datapoint
```

Response:

```
{"nonce": 5619936, "datapoint": {"x": 100, "y": 200, "z": 0, "k": 0, "l": 50},
"deliver_to_device": true}
```

Set light status

Linux/Cygwin curl:

```
curl -X POST -H "Content-Type:application/json" -H "Authorization: token
e474bba4b8e11b97b91019e61b7a018cdbaa3246" -d '{"datapoint":{"x": 100, "y": 200,
"z": 0, "k": 0, "l": 50}} ' http://iot.espressif.cn/v1/datastreams/light/
datapoint/?deliver_to_device=true
```

Windows curl:

```
curl -X POST -H "Content-Type:application/json" -H "Authorization: token
e474bba4b8e11b97b91019e61b7a018cdbaa3246" -d "{\"datapoint\":{\"x\": 100, \"y\":
200, \"z\": 0, \"k\": 0, \"l\": 50}}" http://iot.espressif.cn/v1/datastreams/light/
datapoint/?deliver_to_device=true
```

Response:

```
{"nonce": 5619936,  "datapoint": {"x": 100, "y": 200, "z": 0, "k": 0, "l": 50},
"deliver_to_device": true}
```

X = frequency, range 1~500.

Y (red), Z (green), K (blue) indicate that the different colors of lights, range 0~255. L parameters are reserved.

5. Humidity and Temperature Sensor

Device

Upload data packets of this format:

```
{"nonce": 1,  "path": "/v1/datastreams/tem_hum/datapoint/",  "method": "POST",
"body": {"datapoint": {"x": 35, "y": 32}},  "meta": {"Authorization": "token
HERE_IS_THE_MASTER_DEVICE_KEY"}}
```

X = temperature, Y = humidity.

Device key after token. When upload successful, the server will respond as follows:

Response:

```
{"status": 200,  "datapoint": {"updated": "2014-05-14 18:42:54",  "created":
"2014-05-14 18:42:54",  "visibly": 1,  "datastream_id": 16,  "at": "2014-05-14
18:42:54",  "y": 32,  "x": 35,  "id": 882644}}
```

The last data update time will be contained in the response information.

PC

The PC gets sensor information through two kinds of interfaces. Fonts in red are user's owner key.

- Get the latest information:

```
curl -X GET -H "Content-Type:application/json" -H "Authorization: token
e474bba4b8e11b97b91019e61b7a018cdbaa3246" http://iot.espressif.cn/v1/datastreams/
tem_hum/datapoint
```

Notes: Above mentioned command will respond "remote device is disconnect or busy" since the sensor does not support reverse control.

- Get historical data collected by sensor devices:

```
curl -X GET -H "Content-Type:application/json" -H "Authorization: token
e474bba4b8e11b97b91019e61b7a018cdbaa3246" http://iot.espressif.cn/v1/datastreams/
tem_hum/datapoints
```

4.2. User Defined Reverse Control

The Espressif Cloud Server supports user defined reverse control actions. Send the action to the device with additional parameters and flexible reverse control can be realized. The command format is as follows:

Linux/Cygwin curl:

```
curl -X GET -H "Content-Type:application/json" -H "Authorization: token
HERE_IS_THE_OWNER_KEY" 'http://iot.espressif.cn/v1/device/rpc/?
deliver_to_device=true&action=your_custom_action&any_parameter=any_value'
```

Windows curl:

```
curl -X GET -H "Content-Type:application/json" -H "Authorization: token
HERE_IS_THE_OWNER_KEY" "http://iot.espressif.cn/v1/device/rpc/?
deliver_to_device=true&action=your_custom_action&any_parameter=any_value"
```

The user defined portion is marked red. Users can analyze the action and parameter in codes and define their own functions.

Device side receives following:

```
{"body": {}, "nonce": 872709859, "get": {"action": "your_custom_action",
"any_parameter": "any_value", "deliver_to_device": "true"}, "token":
"HERE_IS_THE_DEVICE_KEY", "meta": {"Authorization": "token HERE_IS_THE_DEVICE_KEY
"}, "path": "/v1/device/rpc/", "post": {}, "method": "GET", "deliver_to_device":
true}
```

Note: RPC commands can only realize flexible reverse control but does not save history info. For example, users can define an action to control the fan to stop turning, but server will not record how many times the fan has turned or stopped.