

DeviceHive

DeviceHive ESP8266 Firmware User Guide.

Created by Nikolay Khabarov

- [1. Overview](#)
- [2. Getting started](#)
- [3. Wireless configuring](#)
- [4. Pin definition](#)
- [5. GPIO](#)
 - [5.1 gpio/write](#)
 - [5.2 gpio/read](#)
 - [5.3 gpio/int](#)
- [6. ADC](#)
 - [6.1 adc/read](#)
 - [6.2 adc/int](#)
- [7. PWM](#)
 - [7.1 pwm/control](#)
- [8. UART](#)
 - [8.1 uart/write](#)
 - [8.2 uart/int](#)
 - [8.3 uart/terminal](#)
- [9. I2C](#)
 - [9.1 i2c/master/read](#)
 - [9.2 i2c/master/write](#)
- [10. SPI](#)
 - [10.1 spi/master/read](#)
 - [10.2 spi/master/write](#)
- [11. Onewire](#)
 - [11.1 onewire/master/read](#)
 - [11.2 onewire/master/write](#)
 - [11.3 onewire/master/int](#)
 - [11.4 onewire/master/search](#)
 - [11.5 onewire/master/alarm](#)
 - [11.6 onewire/dht/read](#)
- [12. License](#)

1. Overview

DeviceHive firmware for ESP8266 is suppose to connect simple electronic devices with common interfaces to DeviceHive cloud service. Having this firmware on ESP8266 there is no need to write programs for hardware. Chip connects to specified remote DeviceHive server(which can be deployed in local network or in some cloud service) and waits for commands. Each command suppose to use some electronic interface to control some hardware connected to chip. It can be simple interaction with GPIO pins for turning on/off simple LED, relay or waiting some changes on pin from sensor. Also it is possible to connected more complicated sensors and actuators using UART or I2C bus. In other words, this firmware move the most commonly used interfaces right into cloud service where you can control your devices or even network of devices from any distance, from any point.

DeviceHive server have two endpoints to send command commands or receive notifications: HTTP and Websocket. Developers can use any programming languages, any developer tools, any SDK that they prefer to interact with server and as a result with hardware. It is possible because all commands represent simple JSON string which sends via HTTP or Websocket. For example sending command with JavaScript looks like:

```
var xmlhttp = new XMLHttpRequest();
xmlhttp.open('POST','http://example.com/api/device/deviceId/command', true);
xmlhttp.setRequestHeader('Authorization', 'Basic ' + btoa('user:password'));
xmlhttp.setRequestHeader('Content-type', 'application/json; charset=UTF-8');
var myjson = {};
myjson['command'] = 'uart/write';
myjson['parameters'] = {'data': 'SGVsbG8sIHdvcmxkIQ=='};
xmlhttp.send(JSON.stringify(myjson));
```

After receiving this command chip prints 'Hello, world!' in UART bus. The same idea with other interfaces. To check if your hardware device is suitable with firmware check which interface your devices has and then check if this interface is supported by DeviceHive ESP8266 firmware.

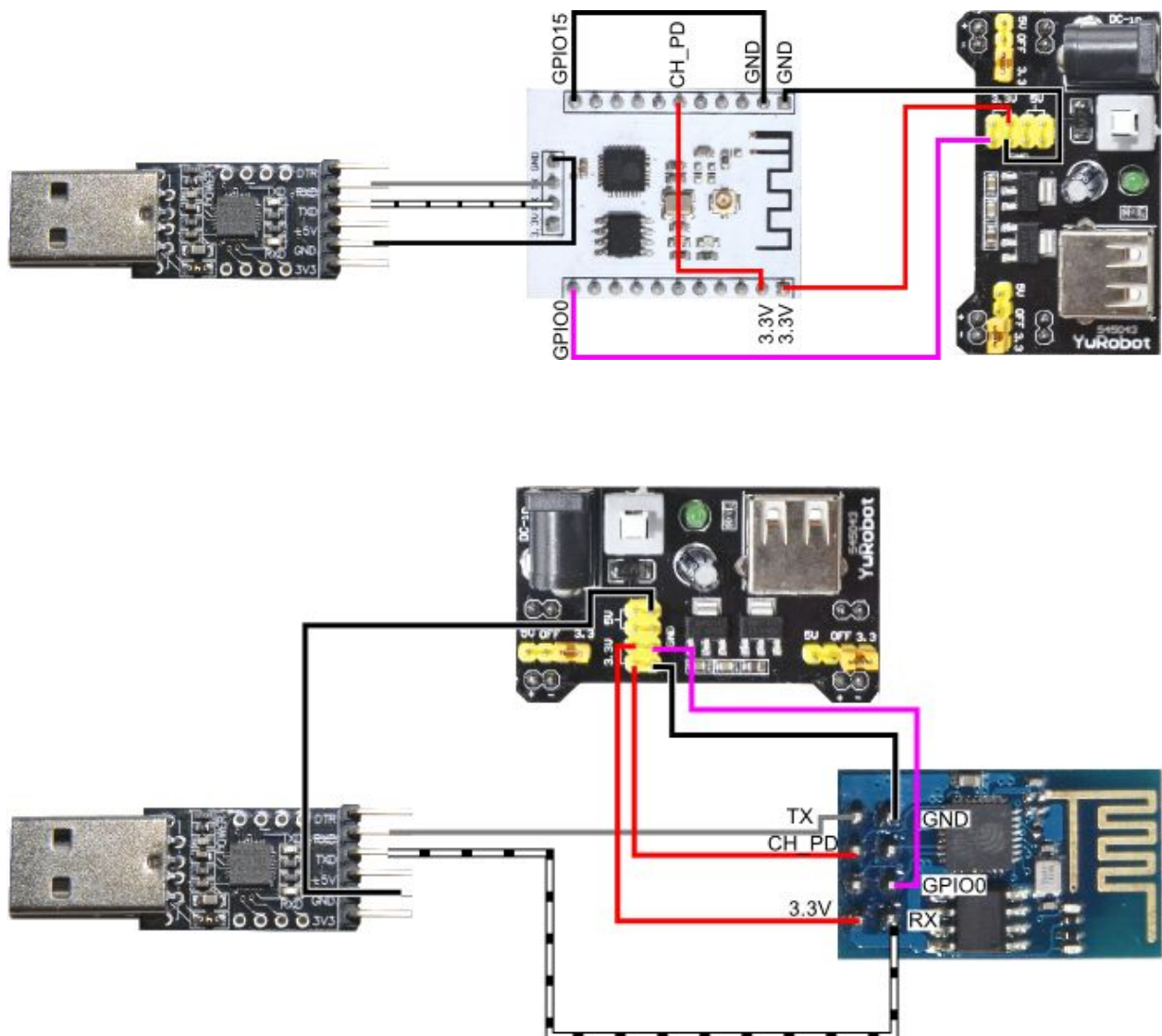
The latest version can be found in git project repository. There are sources codes and binary images: <https://github.com/devicehive/esp8266-firmware>

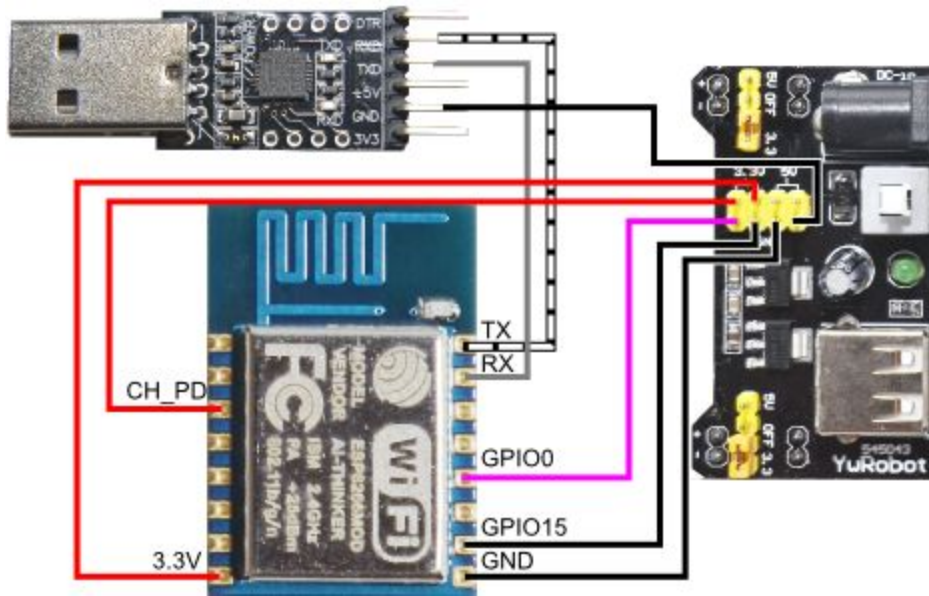
The purpose of this firmware is to provide easy tool for building IoT solutions for developers which used to program on unsuitable for microcontroller programming languages. You can easily use AngularJS framework for example to implement your idea. Also, considering chip price, DeviceHive usability and plenty modules on market which are not require soldering, it looks like perfect tool for prototyping. DIY developers also may find this firmware very useful for their project.

2. Getting started

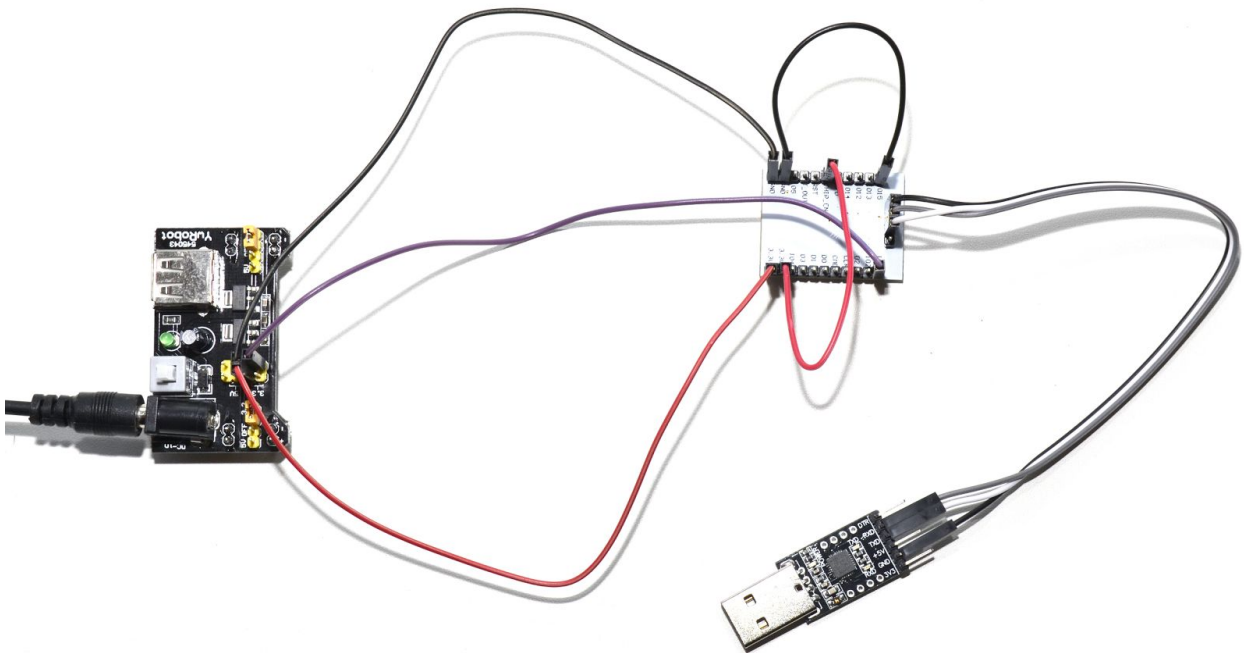
First at all firmware have to be flashed into chip memory and chip have to be configured for using specified Wi-Fi network and DeviceHive server. Developers can build firmware and all tools for flashing and configuring by himself from sources. Or it can be downloaded from git repository: go to <https://github.com/devicehive/esp8266-firmware/tree/master/release> and download archive with the latest version.

For flashing chip needs to be connected to computer via USB-UART converter, CH_PD pin have to be connected to Vcc, GPIO15 and GPIO0 have to be connected to ground and 3.3 power supply should be used. Sample for most popular pre soldered modules connection is below:





Real connection sample:



After assembling, connect it to computer. Install driver for your USB->UART converter. The most popular chip and official sites with drivers below:

CP210x: <http://www.silabs.com/products/mcu/pages/usbtouartbridgevcpdrivers.aspx>

PL230x: <http://www.prolific.com.tw/US/ShowProduct.aspx?pcid=41>

FTDI: <http://www.ftdichip.com/Drivers/VCP.htm>

CH341: http://www.wch.cn/index.php?s=/page-search_content-keyword-CH341SER.html

Make sure that virtual serial port is available in your system(virtual COM is present on Windows OS, '/dev/ttyUSB*' on Linux, '/dev/tty.*' on OS X). Unpack archive with firmware and flash it running 'esp-flasher' in terminal. Flasher automatically detects serial port and use '0x000000.bin' and '0x400000.bin' files for flashing. Successful flasher output is below:

```
Detecting device...
Device found on /dev/ttyUSB1 and successfully synced
No image file were specified. You can specify it in args by pairs
hex adress and image file name, for example:
esp-flasher 0x000000 image1.bin 0x400000 image2.bin
Trying to flash with defaults:
0x000000 <- 0x000000.bin
0x400000 <- 0x400000.bin
0x7C000 <- default configuration
0x7E000 <- 4K of 0xFF
Flashing 0x000000.bin at 0x00000000
Total block count 47, block size 1024
Blocks wrote 47/47 at 0x00000000
Flashing 0x400000.bin at 0x00040000
Total block count 199, block size 1024
Blocks wrote 199/199 at 0x00040000
Flashing default configuration at 0x0007C000
Total block count 1, block size 1024
Blocks wrote 1/1 at 0x0007C000
Flashing 4K of 0xFF at 0x0007E000
Total block count 4, block size 1024
Blocks wrote 4/4 at 0x0007E000
Flashing done.
Press ENTER for exit.
```

Now remove wire from GPIO0(live it float or connect to high), reboot device and connect to firmware with 'esp-terminal' util. You can also use any other tool that can connect to terminal via UART and support escape sequences, PuTTY or GNU 'screen' for example. Port parameters are: 115200 8N1.

Notice: you can avoid configuring firmware with terminal and use wireless configuring procedure described in [paragraph 3](#) instead. Wireless configuring procedure also can be used for end-user devices with this firmware.

Firmware terminal is a unix like terminal with few commands. It exists for chip configuring and debugging. To see debug output type 'dmesg'. To configure run 'configure' command. Follow instructions in terminal. You need to know DeviceHive server credentials for configuring.

For the very beginning or DIY purpose you can use DeviceHive free playground located here: <http://playground.devicehive.com/> Register there and you will have your own DeviceHive server instance. DeviceHive server can be deployed in local network or on some cloud hosting services. Follow for DeviceHive server deployment instructions on <http://devicehive.com>

Configuring sample is below:

```
$ configure
Welcome to the DeviceHive setup utility. Use Ctrl+C to interrupt.
Enter Wi-Fi network SSID.
> MyWiFi
Enter Wi-Fi network password. Leave empty to keep current.
> *****
Enter DeviceHive API URL.
> http://playground.devicehive.com/api/rest
Enter DeviceHive DeviceId. Press Tab button to generate random.
Allowed chars are A-Za-z0-9_-
> esp-device-00h
Enter DeviceHive AccessKey. Leave empty to keep current.
Allowed chars are A-Za-z0-9/+*=
> *****
Configuring complete, store settings...OK
Rebooting...
```

After rebooting you can send commands to DeviceHive server and ESP8266 perform them. List of accepted command is in this document. You can use DeviceHive web admin control panel to send command for test purpose or learning. Go in web admin, 'Devices' tab, 'commands' subtab, 'enter new command'. Type command and parameters and press 'push'. After ESP8266 perform your command you can press 'refresh' button to see result. For example 'gpio/read' command would look in admin control panel as below:

ACCESS KEYS

GRANTS

NETWORKS

DEVICES

NIKOLAY@DEVICEHIVE.COM LOGOUT

Detail view of device "esp-device-00h"

name:

esp-device-00h

status:

Online

operation:

Normal

network:

Sample network for nikolay@devicehive.com

device class:

ESP Class (v 0.4)

data:

EDIT

commands

notifications

ENTER NEW COMMAND

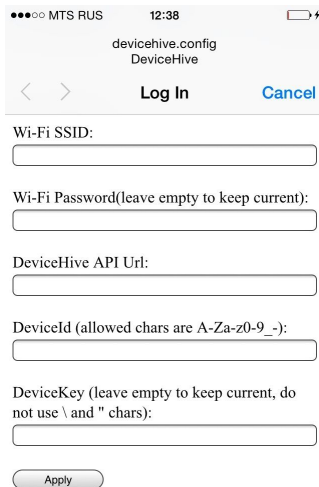
| NAME | TIME (UTC) | PARAMETERS | STATUS | RESULT |
|-----------|---------------------|------------|--------|---|
| gpio/read | 10/31/2015 13:50:56 | | OK | {0:1,1:0,2:1,3:0,4:1,5:1,12:0,13:0,14:0,15:0} |

COPY

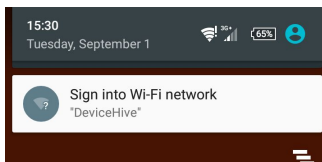
Now you can start writing your own program to create your own IoT devices with your favorite language and frameworks using DeviceHive RESTful API: <http://devicehive.com/restful> which you can transmit with HTTP(S) or Websockets. List of accepted command for ESP8266 is listed in this document.

3. Wireless configuring

Since DeviceHive ESP8266 firmware flashed into chip, it can be configured without any special devices or software. So this mode can be used in end user projects to providing easy way for configuring device. To enter configuration mode just reset device three times with chip RESET pin. Intervals between resets should be more than half seconds and less than 3 seconds, i.e. simply reset device three times leisurely. If board has LED connected to TX pin, it turns on. ESP8266 will operate as Wi-Fi access point providing open wireless network with SSID 'DeviceHive'. Connect to this network with your laptop/phone/tablet or other device with Wi-Fi support. Device with iOS and OS X automatically will show configuration page like below:



Android devices will show notification 'Sign into Wi-Fi network' in bar:



Tap on it to open the same configuration page. In case of using other devices, just open browser and type URL: <http://devicehive.config>

Also you can type any URL with http scheme in it, you will be redirected on URL above anyway while you are connected to ESP8266 in configuration mode.

Type your configuration in form and click 'Apply' button. Device will be rebooted in 10 seconds in normal mode with new configuration.

Notice: You can automate configuration process for your devices. Actually to configure firmware you need to send HTTP POST request like below to 192.168.2.1:80

POST / HTTP/1.0

Host: devicehive.config

Content-Type: .application/x-www-form-urlencoded

Content-Length: 80

ssid=ssid&pass=pass&url=http%3A%2F%2Fexample.com%2Fapi&id=deviceid&key=accesskey

4. Pin definition

| Pin name in commands | Function | ESP8266 number | pin |
|----------------------|---------------------------|-------------------|-----|
| <i>GPIO</i> | | | |
| "0" | GPIO0 | 15 | |
| "1" | GPIO1, UART_TX | 26 | |
| "2" | GPIO2 | 14 | |
| "3" | GPIO3, UART_RX | 25 | |
| "4" | GPIO4 | 16 | |
| "5" | GPIO5 | 24 | |
| "12" | GPIO12, SPI_MISO | 10 | |
| "13" | GPIO13, SPI_MOSI | 12 | |
| "14" | GPIO14, SPI_CLK | 9 | |
| "15" | GPIO15 | 13 | |
| <i>ADC</i> | | | |
| "0" | ADC0 | 6 | |
| <i>Common</i> | | | |
| "all" | all pins in current group | | |

Notes:

GPIO6-GPIO11 usually connected to on-module EEPROM, that is why no API for this pins.

5. GPIO

Each ESP8266 pin can be loaded up to 12 mA. Pins also have overvoltage and reverse protection.

5.1 gpio/write

Sets gpio pins state according to the specified parameters. Pins will be automatically initialized to output. All pins will be setted up simultaneously. Unlisted pins will not be touched.

Parameters:

Json with set of key-value, where key is pin name and value '0', '1' or 'x'. '0' means low level, '1' means high level, 'x' means dummy request which added for compatibility and easy string json generation. Sample below, sets gpio10 to low level and gpio11 to high level.

Example:

```
{
  "10": "0",
  "11": "1",
  "12": "x"
}
```

Return 'OK' in status on success or 'Error' and description in result on error.

5.2 gpio/read

Read all gpio pins state. Pins will not be initialized as input. if pins were not specified in parameters.

Parameters:

Can be empty.

Json with set of key-value, where key is pin name and value can be:

"init" - all pins are initialized as input by default, if pin was used as output or any other peripheral module before, pass this argument to reinit pin before reading. Pullup state will not be touched.

"pullup" - init pin as input and enable pullup

"nopull" - init pin as input and disable pullup

Example:

```
{
  "10": "init",
  "11": "pullup",
  "12": "nopull"
}
```

Note: pull up and pull down are the SoC feature that allow to put input to high level or low level through resistor with very high resistance. By default each pin in float ('Z') condition which state not determined and reading will return random value if pin doesn't connect to high or low source. Enabling pull up feature helps to have very weak high level on input pin by default and pull down sets very weak low level.

Return 'OK' in status and json like below in result on success. Or 'Error' and description in result on error.

```
{
  "0": "0",
  "1": "1"
  ....
  "16": "0"
}
```

5.3 gpio/int

Enable or disable notification on pin state changes(enable interruptions).

Parameters:

Json with set of key-value, where key is pin name and value can be:

"disable" - disable interruption if it was enabled before

"rising" - send notification on rising edge

"falling" - send notification on falling edge

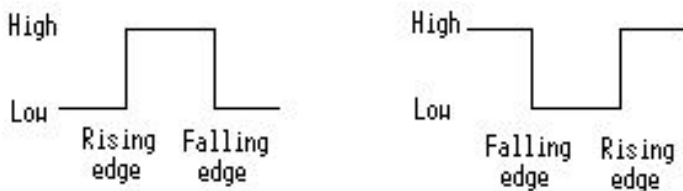
"both" - send notification on rising and falling edge

'timeout' - additional parameters. Notification will be sent only after this period of time. Minimum is 50 ms. Maximum is 8388607 ms. If not specified previous timeout will be used. Default is 250 ms.

Mnemonic "all" can be used as key to set up something for all pins.

Notes:

Timeout feature can be very useful with real physical switches for contact bounce filtering.



Example:

```
{
  "all": "disable",
  "11": "rising",
  "12": "falling",
  "13": "both",
}
```

```
    "timeout": "100"  
  }
```

Return 'OK' in status. Or 'Error' and description in result on error.

Notifications will be sent with name 'gpio/int'. Each notification will contain list of gpio which caused interruption in 'caused' field, current(on notification creating moment, after timeout) gpio inputs(all of them) state in 'state' field and tick count on gpio read moment in microseconds based on internal on chip time:

```
{  
  "caused": ["0", "1"],  
  "state": {  
    "0": "0",  
    "1": "1"  
    ....  
    "16": "0"  
  }  
  "tick": "123456"  
}
```

6. ADC

ESP8266 has just one ADC channel. This channel connected to dedicated pin #6 - 'TOUT'. ADC can measure voltage in range 0..1 Volts with 10 bit resolution.

6.1 adc/read

Reads ADC channels values. ESP8266 has just one channel - '0'.

Parameters:

Can be empty, all channels value will be sent in this case.

Json with set of key-value, where key is ADC channel and value can be:

"read" - read channel current value

Example:

```
{
  "all": "read",
  "0": "read"
}
```

Return 'OK' in status and json like below in result on success. Each entry contains channel number and value in volts. 'Error' and description will be send in result on error.

```
{
  "0": "0.6"
}
```

6.2 adc/int

Subscribes on notifications with ADC value with some period.

Parameters:

Json with set of key-value, where key is ADC channel and value is period in milliseconds or 'disable' for disabling. '0' value also means disable. Period can be from 250 till 8388607 ms.

Example:

```
{
  "0": "1000",
  "0": "disable"
}
```

In this example value of channel 0 will be sent every 1 second.

Return 'OK' in status. Or 'Error' and description in result on error. Notification will have name 'adc/int' and following format:

```
{
  "0": "0.0566"
}
```

Where "0" channel number, and "0.0566" current voltage in volts.

7. PWM

PWM implementation is software. PWM has just one channel, but this channel can control all GPIO outputs with different duty cycle. It also means that all outputs are synchronized and work with the same frequency. PWM depth is 100. PWM can be used as pulse generator with specified number of pulses.

7.1 pwm/control

Enable or disable PWM.

Parameters:

Json with set of key-value, where key is pin name and value is duty cycle. Duty cycle is an integer between 0..100, ie percent. Mnemonic pin 'all' also can be used to control all GPIO pins simultaneously. To disable PWM for one of the outputs, just set value to 'disable' or '0'. PWM can be also disabled for pin if command 'gpio/write' or 'gpio/read'(only with some pins for init) is called for pin.

There are also additional parameters:

'frequency' - set PWM base frequency, if this parameter was omitted, previous frequency will be used. 'frequency' also can be set while PWM working or before command with pins duty cycles. Default frequency is 1 kHz. Minimum frequency is 0.0005 Hz, maximum is 2000 Hz

'count' - the number of pulses that PWM will generate after command, maximum is 4294967295, 0 means never stop. Pins with 100% duty cycle will be switched to low level when pwm stops.

Example:

```
{
  "0": "50",
  "frequency": "1000",
  "count": "10000"
}
```

This example starts PWM with square pulses for 10 seconds, duty cycle is 50%, frequency 1 kHz.

Return 'OK' in status. Or 'Error' and description in result on error.

Notes:

PWM is can be used to generate single or multiple pulses with specific length:

{ "0": "100", "frequency": "1000", "count": "100" } - generate single pulse 100 milliseconds length
{ "0": "30", "frequency": "0.1", "count": "4" } - generate 4 pulses 3 seconds length, 7 seconds interval between pulses.

8. UART

ESP8266 has one UART interface. RX pin is 25(GPIO3), TX pin is 26(GPIO1). All read operations have to be done with notifications.

8.1 uart/write

Send data via UART.

Parameters:

"mode" - UART speed which can be in range 300..230400. After speed may contains space and UART framing parameters: number of bits(5-8), parity mode(none - "N", odd - "O" or even - "E"), stop bits(one - "1", two - "2"). Framing can be omitted, 8N1 will be used in this case. If this parameter specified UART will be reinit with specified mode. If this parameter is omitted, port will use current settings("115200 8N1" by default) and will not reinit port.

"data" - data string encoded with base64. Original data size have to be equal or less than 264 bytes.

Example:

```
{
  "mode": "115200",
  "data": "SGVsbG8sIHdvcmxkIQ=="
}
```

Return 'OK' in status. Or 'Error' and description in result on error.

8.2 uart/int

Subscribe on notification which contains data that was read from UART. Firmware starts wait for data from and each time when byte is received byte puts into buffer (264 bytes len), then firmware starts wait for the next byte with some timeout. When timeout reached or buffer is full firmware sends notification.

Parameters:

"mode" - the same "mode" parameter as in "uart/write" command, see description there. It also can be omitted to keep current parameters. Additionally this parameter can be "disable" or "0" for disabling notifications.

"timeout" - timeout for notifications. Default is 250 ms.

Example:

```
{
  "mode": "38400 8E2"
}
```

Return 'OK' in status. Or 'Error' and description in result on error. Notifications with name 'uart/int' will have following format:

```
{  
  "data": "SGVsbG8sIHdvcmxkIQ=="  
}
```

Where "data" key name is always used and value is string with base64 encoded data(264 or less bytes).

8.3 uart/terminal

Resume terminal on UART interface. If UART's pins were used by another feature(i.e. for GPIO or custom UART protocol) this command resume UART terminal back and disables UART notifications. Port will be reinit with 115200 8N1.

Parameters:

No parameters.

Return 'OK' in status. Or 'Error' and description in result on error.

9. I2C

There is software implementation of I2C protocol. Any GPIO pin can be SDA or SCL.

9.1 i2c/master/read

Read specified number of bytes from bus. This command also can set up pins that will be used for I2C protocol. Pins will be init with open-drain output mode and on-board pull up will be enabled.

Parameters:

"address" - I2C slave device address, hex value. Can start with "0x".

"count" - number of bytes that should be read. If not specified, 2 bytes will be read. Can not be 0.

"data" - base64 encoded data that should be sent before reading operation. Repeated START will be applied for bus if this field specified. Maximum size of data is 264 bytes.

"SDA" - GPIO port number for SDA data line. If not specified, previous pins will be used. Default is "0".

"SCL" - GPIO port number for SCL data line. If not specified, previous pins will be used. Default is "2".

Example:

```
{
  "SDA": "4",
  "SCL": "5",
  "address": "78",
  "count": "1",
  "data": "YWI="
}
```

Notes:

Very common situation when slave device needs to be written with register address and data can be readed after repeated START. Using this command with "data" field allow to organise repeated START for reading.

Return 'OK' in status and json like below in result on success. Or 'Error' and description in result on error.

```
{
  "data": "YWE="
}
```

"data" field is base64 encoded data that was read from bus.

9.2 i2c/master/write

Write data to I2C bus.

Parameters:

"address" - I2C slave device address, decimal integer value.

"data" - base64 encoded data that should be sent. Maximum size of data is 264 bytes.

"SDA" - GPIO port number for SDA data line. If not specified, previous pin will be used. Default is "0".

"SCL" - GPIO port number for SCL data line. If not specified, previous pin will be used. Default is "2".

Example:

```
{  
    "SDA": "4",  
    "SCL": "5",  
    "address": "122",  
    "data": "YWI="
```

Return 'OK' in status. Or 'Error' and description in result on error.

10. SPI

ESP8266 has hardware SPI module. MISO pin is 10 (GPIO12), MOSI pin is 12(GPIO13), CLK is 9(GPIO14), CS can be specified as any other pin. Clock divider is fixed and equal 80, i.e. SPI clock is 1 MHz.

10.1 spi/master/read

Read data from SPI bus.

Parameters:

"count" - number of bytes that should be read. If not specified, 2 bytes will be read. Can not be 0.

"data" - base64 encoded data that should be sent before reading. Maximum size of data is 264 bytes.

"mode" - Select SPI clock mode. Can be:

0 - Low clock polarity, front edge

1 - Low clock polarity, rear edge

2 - High clock polarity, front edge

3 - High clock polarity, rear edge

If not specified, previous mode will be used. Default is 0.

"CS" - GPIO port number for CS(chip select) line. If not specified, previous pin will be used. Can be "x" for disabling CS usage. Default is "x". Can not be the same pin as used for other SPI data communication.

Example:

```
{
  "data": "YWI=",
  "CS": "15",
  "count": "2",
  "mode": "0"
}
```

Return 'OK' in status and json like below in result on success. Or 'Error' and description in result on error.

```
{
  "data": "YWE="
}
```

"data" field is base64 encoded data that was read from bus.

10.2 spi/master/write

Write data to SPI bus.

Parameters:

"data" - base64 encoded data that should be sent. Maximum size of data is 264 bytes.

"mode" - Select SPI clock mode. Can be:

0 - Low clock polarity, front edge

1 - Low clock polarity, rear edge

2 - High clock polarity, front edge

3 - High clock polarity, rear edge

If not specified, previous mode will be used. Default is 0.

"CS" - GPIO port number for CS(chip select) line. If not specified, previous pin will be used. Can be "x" for disabling CS usage. Default is "x". Can not be the same pin as used for other SPI data communication.

Example:

```
{  
    "data": "YWI=",  
    "CS": "15",  
    "mode": "0"  
}
```

Return 'OK' in status. Or 'Error' and description in result on error.

11. Onewire

There is a software implementation of some onewire protocols. Any GPIO pin can be used for connection. Master operate as 1-wire master. Though selected pin will have on chip pull up, additional pull up resistor may require. Typically 4.7k Ohm.

11.1 onewire/master/read

Read specified number of bytes from onewire bus. Onewire pin can also be specified with this for this command. Selected pin will be init with open-drain output mode and on-board pull up will be enabled.

Parameters:

"count" - number of bytes that should be read. Can not be 0.

"data" - base64 encoded data that have to be sent before reading operation for initialize device for sending some data. Cannot be empty. Maximum size of data is 264 bytes.

"pin" - GPIO port number for onewire data line. If not specified, previous pins will be used. Default is "0".

Example:

```
{
  "count": "1",
  "data": "YWI=",
  "pin": "2"
}
```

Return 'OK' in status and json like below in result on success. Or 'Error' and description in result on error.

```
{
  "data": "YWE="
}
```

"data" field is base64 encoded data that was read from bus.

11.2 onewire/master/write

Read specified data to onewire bus. Onewire pin can also be specified with this for this command. Selected pin will be init with open-drain output mode and on-board pull up will be enabled.

Parameters:

"data" - base64 encoded data that have to be sent before reading operation for initialize device for sending some data. Maximum size of data is 264 bytes.

"pin" - GPIO port number for onewire data line. If not specified, previous pins will be used. Default is "0".

Example:

```
{
  "data": "YWI=",
  "pin": "2",
}
```

Return 'OK' in status. Or 'Error' and description in result on error.

11.3 onewire/master/int

Enable or disable notifications for event on one wires buses.

Parameters:

Json with set of key-value, where key is pin name and value can be:

"disable" - disable interruption if it was enabled before

"presence" - scan bus and send notification with scan result when device send PRESENCE signal on bus. 1-wire devices sends this signal after powering on, so it can be used to read iButton devices serial.

Mnemonic "all" can be used as key to set up something for all pins.

Example:

```
{
  "2": "presence",
}
```

Return 'OK' in status. Or 'Error' and description in result on error.

Notifications will be sent with name 'onewire/master/int'. Each notification will contain list of device's serial numbers and pin where it was found:

```
{
  "found": ["5800000A08DB5E01", "700000288D214B01"],
  "pin": "2"
}
```

11.4 onewire/master/search

Search bus for serial numbers of all attached devices.

Parameters:

"pin" - GPIO port number for onewire data line. If not specified, previous pins will be used. Default is "0".

Example:

```
{  
  "pin": "2",  
}
```

Return 'OK' in status and result with list as below. Or 'Error' and description in result on error.

```
{  
  "found": ["5800000A08DB5E01", "700000288D214B01"],  
  "pin": "2"  
}
```

11.5 onewire/master/alarm

Search bus for serial numbers of attached devices which are in alarm state.

Parameters:

"pin" - GPIO port number for onewire data line. If not specified, previous pins will be used. Default is "0".

Example:

```
{  
  "pin": "0",  
}
```

Return 'OK' in status and result with list as below. Or 'Error' and description in result on error.

```
{  
  "found": ["5800000A08DB5E01", "700000288D214B01"],  
  "pin": "2"  
}
```

11.6 onewire/dht/read

Read data from DHT11/DHT22/AM2302 or device with the same protocol. Number of readed data depends on device, but can not be more that 264. Any checksums will not be checked.

Parameters:

"pin" - GPIO port number for onewire data line. If not specified, previous pins will be used. Default is "0".

Example:

```
{  
    "pin": "0",  
}
```

Return 'OK' in status and json like below in result on success. Or 'Error' and description in result on error.

```
{  
    "data": "YWE=",  
}
```

"data" field is base64 encoded data that was read from bus.

12. License

The MIT License (MIT):

Copyright (c) 2015 DeviceHive

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

DeviceHive firmware is based on Espressif IoT SDK which has ESPRESSIF MIT License:

ESPRESSIF MIT License

Copyright (c) 2015 <ESPRESSIF SYSTEMS (SHANGHAI) PTE LTD>

Permission is hereby granted for use on ESPRESSIF SYSTEMS ESP8266 only, in which case, it is free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

乐鑫 MIT 许可证

版权 (c) 2015 <乐鑫信息科技（上海）有限公司>

该许可证授权仅限于乐鑫信息科技 ESP8266 产品的应用开发。在此情况下，该许可证免费授权任何获得该软件及其相关文档（统称为“软件”）的人无限制地经营该软件，包括无限制的使用、复制、修改、合并、出版发行、散布、再授权、及贩售软件及软件副本的权利。被授权人在享受这些权利的同时，需服从下面的条件：

在软件和软件的所有副本中都必须包含以上的版权声明和授权声明。

该软件按本来的样子提供，没有任何明确或暗示的担保，包括但不限于关于试销性、适合某一特定用途和非侵权的保证。作者和版权持有人在任何情况下均不就由软件或软件使用引起的以合同形式、民事侵权或其它方式提出的任何索赔、损害或其它责任负责。

All trademarks, service marks, trade names, trade dress, product names and logos appearing on the firmware repository are the property of their respective owners.