# DeviceHive ESP8266 Firmware User Guide.

Created by Nikolay Khabarov

v0.2, 12 Aug 2015

# 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://exmaple.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.

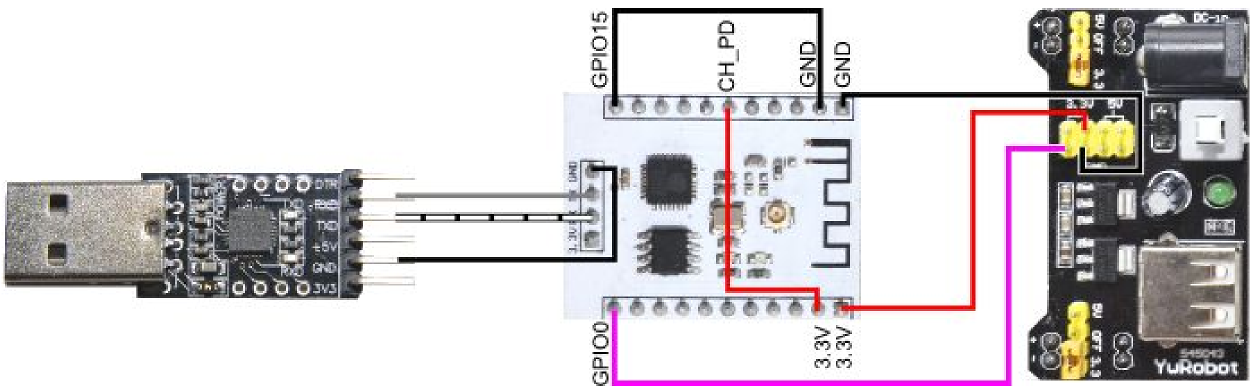
The latest version can be found in git project repository. There are sources codes and binary images: https://github.com/devicehive/esp8266-firmware

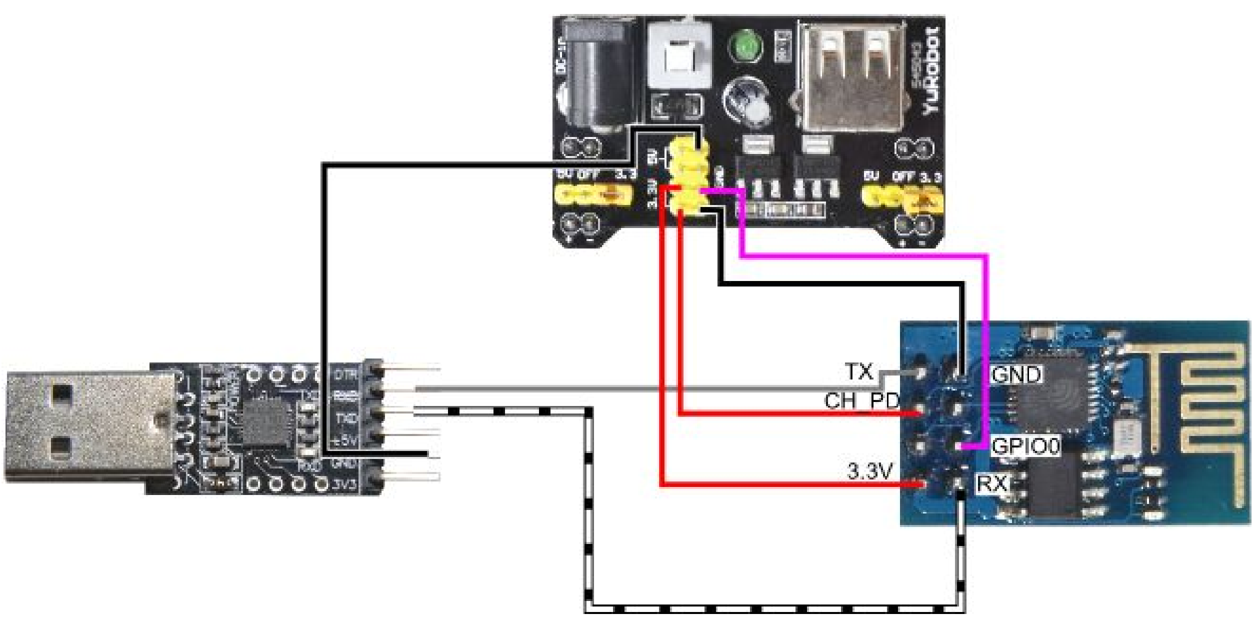
The purpose of this firmware is to provide easy tool for building IoT solutions for developers which used to program on unsuitable for microcontroller programming languages. You can easily use AngularJS framework for example to implement your idea. Also, considering chip price, DeviceHive usability and 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.

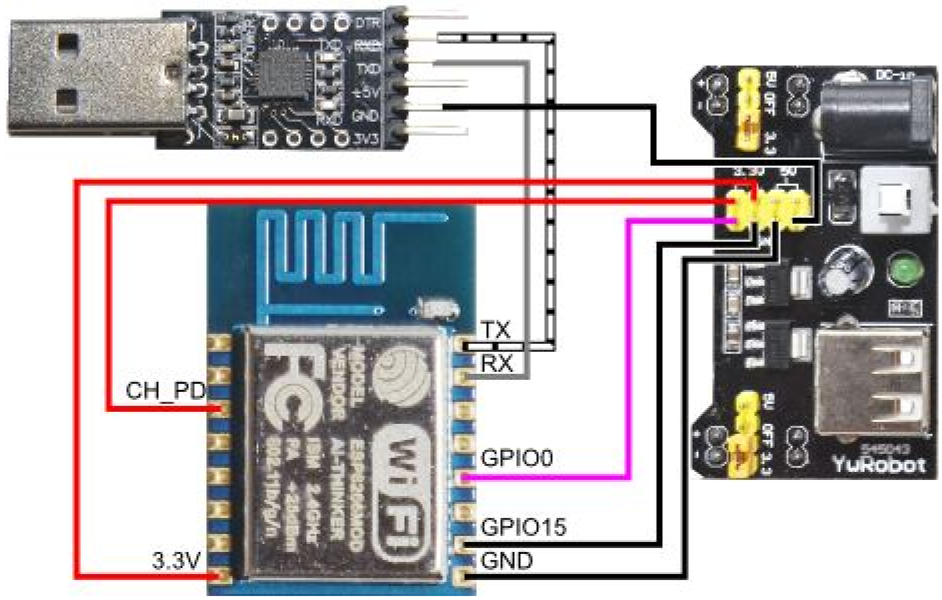
## 2. Getting started

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.

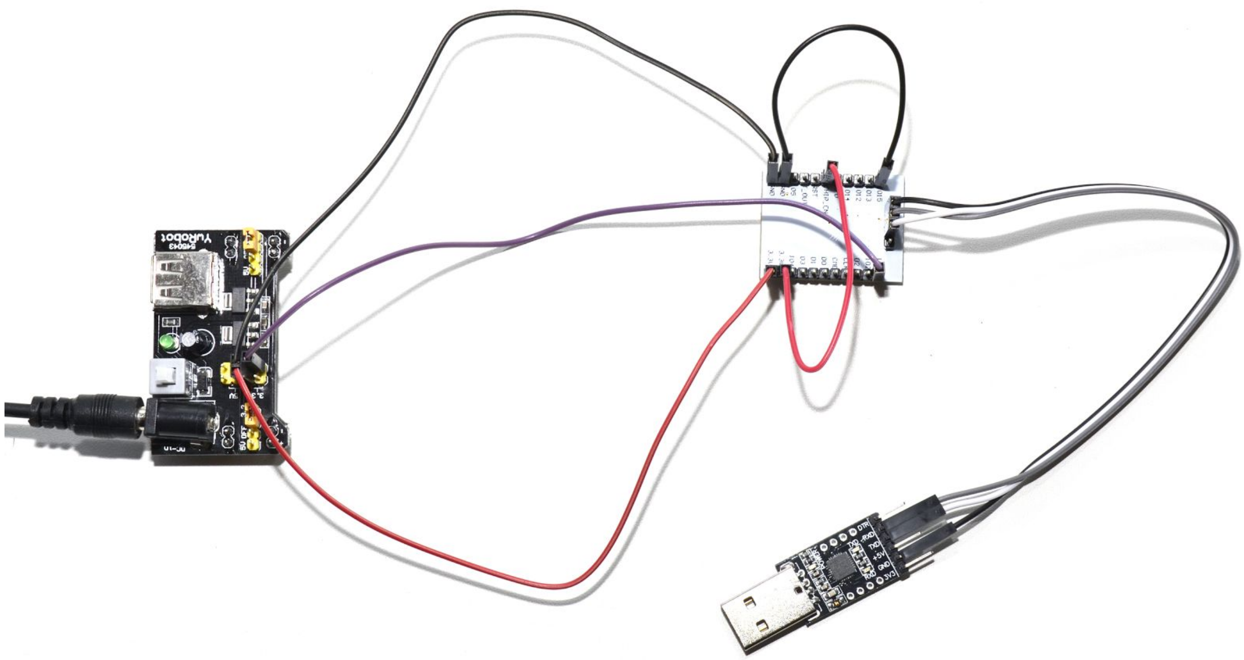
For flashing chip needs to be connected to computer via USB-UART converter, CH_PD pin have to be connected to Vcc, GPIO15 and GPIO0 have to be connected to ground and 3.3 power supply should be used. Sample for most popular pre soldered modules connection is below:






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-<your-os>' in terminal. Flasher automatically detects serial port and use '0x00000.bin' and '0x40000.bin' files for flashing. Successful flasher output is below:

```
Detecting device...
Device found on COM2 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 0x00000 image1.bin 0x40000 image2.bin
Trying to flash with defaults:
0x00000 <- 0x00000.bin
0x40000 <- 0x40000.bin
Flashing 0x00000.bin at 0x00000000
Total block count 44, block size 1024
Blocks wrote 44/44 at 0x000000000
Flashing 0x40000.bin at 0x00040000
Total block count 177, block size 1024
Blocks wrote 177/177 at 0x000400000
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 with 'esp-terminal-<your-os>' util. You can also use any other util that can connect to terminal via UART and support escape sequences, PuTTY or GNU 'screen' for example. Port parameters are: 115200 8N1.

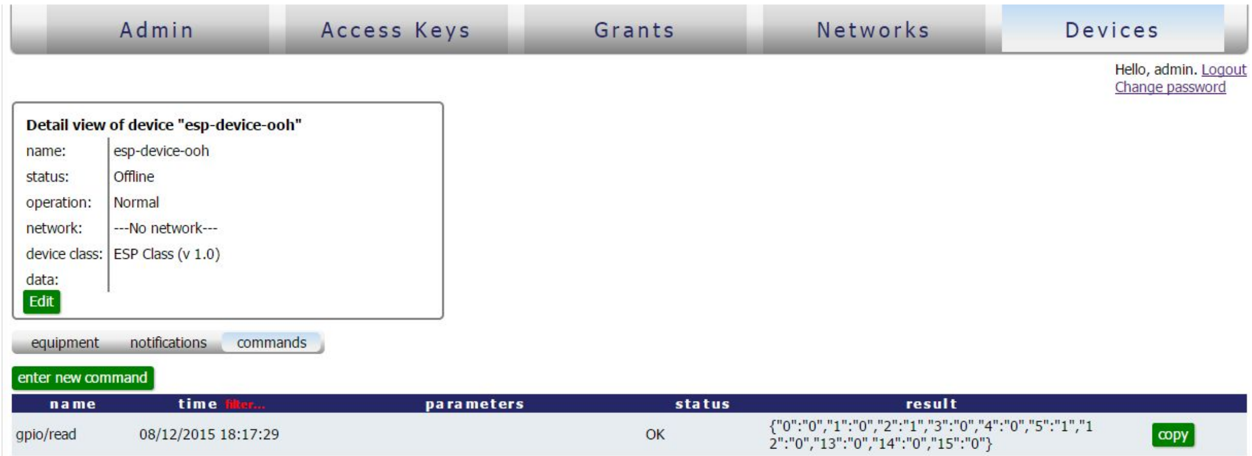
Firmware terminal is a unix like terminal with few commands. It exists for chip configuring and debugging. To see debug output type 'dmesg'. To configure run 'configure' command. Follow instructions in terminal. You need to now DeviceHive server credentials for configuring.

*For the very beginning or DIY purpose you can use DeviceHive free playground located here: [http://devicehive.com/playground](http://devicehive.com/playground) 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](http://devicehive.com)*

Configuring sample is below:

```
$ configure
Welcome to the DeviceHive setup utility. Use Ctrl+C to interrupt.
Enter Wi-Fi network SSID
> MyHomeWiFi
Enter Wi-Fi network password. Leave empty to keep current
> *************************
Enter DeviceHive server URL
> http://nn8354.pg.devicehive.com/api
Enter DeviceHive DeviceId. Press Tab button to generate random uuid
> esp-demo-device
Enter DeviceHive DeviceKey. Press Tab button to generate random key (will be
shown on screen). Leave empty to keep current
> ********************************************
Configuring complete, store settings...OK
Rebooting...
```

After rebooting you are free to 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:




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

# 3. Pin definition

| Pin name in commands | Function | ESP8266 pin number |
|---|---|---|
| *GPIO* | | |
| "0" | GPIO0 | 15 |
| "1" | GPIO1, UART0TX | 26 |
| "2" | GPIO2 | 14 |
| "3" | GPIO3, UART0RX | 25 |
| "4" | GPIO4 | 16 |
| "5" | GPIO5 | 24 |
| "12" | GPIO12 | 10 |
| "13" | GPIO13 | 12 |
| "14" | GPIO14 | 9 |
| "15" | GPIO15 | 13 |
| *ADC* | | |
| “0” | ADC0 | 6 |
| *Common* | | |
| "all" | all pins in current group | |

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

# 4. GPIO

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

## 4.1 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.

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

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

## 4.2 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"
}
```

## 4.3 gpio/int

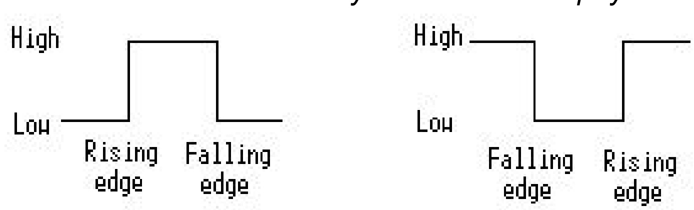
Enable or disable notification on pin state changes(enable interruptions).

*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.

*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 and current(on notification creating moment, after timeout) gpio inputs(all of them) state in ‘state’ field:


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

# 5. 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.

## 5.1 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"
}
```

## 5.2 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.

# 6. 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.

## 6.1 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 4000 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.*

# 7. 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.

## 7.1 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.

## 7.2 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).

## 7.3 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.

# 8. I2C

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

## 8.1 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.

## 8.2 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 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":"122"
	"data":"YWI="
}
```

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

# 9. License

## The MIT License (MIT):



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