

Bella Hat Documentation

version

2024, Sunfounder

■■ 12, 2024

Contents

Bella Hat documentation	1
Bella Hat Library	1
Install	1
Examples	1
Basic test	1
Reset MCU	2
Reference	2
class Bella	2
class Pin	3
class ADC	6
class PWM	7
module motor	8
class Motors	8
class Motor	10
class I2C	11
module modules	12
class Ultrasonic	12
class Grayscale_Module	13
class DHT11	14
class LSM6DSOX	14
class Music	15
module utils	18
class _Basic_class	18
Index	21
Python Module Index	25

Bella Hat documentation

Bella Hat Library

Install

1. Update your system.

Make sure you are connected to the Internet and update your system:

```
sudo apt update
```

Note

Python3 related packages must be installed if you are installing the **Lite** version OS.

```
sudo apt install git python3-pip python3-setuptools python3-smbus
```

2. Download the source code.

```
git clone https://github.com/sunfounder/bella-hat.git
```

3. Install the package.

It is recommended to install within a Python virtual environment.

```
cd bella-hat
pip3 install ./
```

Note

if you want to install in system environment. You need add parameter “--break-system-packages”

```
pip3 install ./ --break-system-packages
```

Examples

Basic test

```
cd bella-hat
python3 examples/basic_test.py
```

```

      ↑
← → power: 020 020 020
      ↓

Battery:      ||||| | 81.6 8.08 V
Ultrasonic:   5.99 cm
DHT11:        temperature: 28°C humidity: 58%
LSM6DSOX:     acc: X:0.52, Y: -0.42, Z: 9.87 m/s^2
               gyro X:-0.01, Y: 0.00, Z: 0.01 radians/s
Grayscale:    [818, 877, 544]
Fan:          off
Btn:          released

Move: [W,A,S,D]  STOP: [X]  Honk: [Q]  Fan: [E]
```

Reset MCU

Reset the MCU on the Bella HAT.

```
bella-hat reset_mcu
```

Reference

class Bella

The class `Bella` encapsulates the usage functions of all physical interfaces on the Bella HAT and presets the corresponding pin numbers. For a more detailed example, you can refer to the `examples/basic_test.py` file.

Example

```
from bella_hat.bella import Bella

bella = Bella()
bellas.et_motors_reverse(True, False)

bella.set_eyes_led(80, 80)
bella.set_motors(20, 20)
bella.fan_on()

batVolt = bella.get_battery_voltage()
batPerc = bella.get_battery_percentage()
distance = bella.get_ultrasonic_distance()
temp = bella.get_temperature()
hum = bella.get_humidity()
grayscale = bella.get_grayscales()
acc = bella.get_acc()
gyro = bella.get_gyro()
btn_state = bella.read_btn()
print(f'''
Battery:      {batPerc:.1f} {batVolt:.2f} V
Ultrasonic:   {distance} cm
DHT11:        temperature: {temp}'C    humidity: {hum}%
LSM6DSOX:     acc: X:{acc[0]:.2f},    Y: {acc[1]:.2f},    Z: {acc[2]:.2f} m/s^2
               gyro X:{gyro[0]:.2f},   Y: {gyro[1]:.2f},   Z: {gyro[2]:.2f} radians/s
Grayscale:    {grayscale}
Fan:          {"on" if bella.fan_state else "off"}
Btn:          {"pressed" if btn_state else "released"}
''')
```

API

```
class bella_hat.bella.Bella(*args, **kwargs)
    Bases: _Basic_class
```

```
__init__(*args, **kwargs)
    Initialize the basic class
```

Parameters: `debug_level` (*str/int*) – debug level, 0(critical), 1(error), 2(warning), 3(info) or 4(debug)

```
get_battery_voltage()
    ' Get the battery voltage.
    return: float - Battery voltage in volts.
```

```
get_battery_percentage()
    Get the battery percentage.
    return: float - Battery percentage.
```

get_ultrasonic_distance ()

Get the distance from the ultrasonic sensor.

return: float - Distance in cm.

-1, timeout -2, distance limit exceeded

get_temperature ()

Get the temperature from the DHT11 sensor.

return: float - Temperature in Celsius.

get_humidity ()

Get the humidity from the DHT11 sensor.

return: float - Humidity in percentage.

get_grayscales ()

Get the grayscale sensor values.

return: list - Grayscale sensor values.

set_motors (leftPercent, rightPercent)

Set the motor percentage for left and right motors

leftPercent: int - Motor percentage for left motor, -100 to 100 rightPercent: int - Motor percentage for right motor, -100 to 100

set_motors_reverse (left_reverse, right_reverse)

Set the motors is reversed or not

left_reverse: bool - set left motor is reversed or not right_reverse: bool - set right motor is reversed or not

get_acc ()

Get acceleration data of lsm6dsox sensor.

return: list - [acc_x, acc_y, acc_z].

get_gyro ()

Get gyro data of lsm6dsox sensor.

return: list - [gyro_x, gyro_y, gyro_z].

fan_on ()

Open the fan on Bella HAT.

fan_off ()

Close the fan on Bella HAT.

read_btn ()

Read the state of btn on Bella HAT.

set_eyes_led (left_brightness, right_brightness)

Set the front board LEDs brightness.

left_brightness: int - brightness for left LEDs, 0 to 100 right_brightness: int - brightness for right LEDs, 0 to 100

class Pin

Example

Simple read or control pin:

```
# Import Pin class
from robot_hat import Pin

# Create Pin object with numeric pin numbering and default input pullup enabled
d0 = Pin(16, Pin.IN, Pin.PULL_UP)
# Create Pin object with named pin numbering
d1 = Pin(25)
```

```
# read value
value0 = d0.value()
value1 = d1.value()
print(value0, value1)

# write value
d0.value(1) # force input to output
d1.value(0)

# set pin high/low
d0.high()
d1.off()
```

Interrupt:

```
# Import Pin class
from bella_hat.pin import Pin

# set interrupt
led = Pin(16, Pin.OUT)
switch = Pin(25, Pin.IN, Pin.PULL_DOWN)
def onPressed(chn):
    led.value(not switch.value())
switch.irq(handler=onPressed, trigger=Pin.IRQ_RISING_FALLING)

while True:
    pass
```

API

`class bella_hat.pin.Pin` (pin, mode=None, pull=None, *args, **kwargs)
Bases: `_Basic_class`
Pin manipulation class

`OUT = 1`
Pin mode output

`IN = 2`
Pin mode input

`PULL_UP = 17`
Pin internal pull up

`PULL_DOWN = 18`
Pin internal pull down

`PULL_NONE = None`
Pin internal pull none

`IRQ_FALLING = 33`
Pin interrupt falling

`IRQ_RISING = 34`
Pin interrupt falling

`IRQ_RISING_FALLING = 35`
Pin interrupt both rising and falling

`__init__` (pin, mode=None, pull=None, *args, **kwargs)
Initialize a pin

Parameters:

- **pin** (*int/str*) – pin number of Raspberry Pi
- **mode** (*int*) – pin mode(IN/OUT)
- **pull** (*int*) – pin pull up/down(PUD_UP/PUD_DOWN/PUD_NONE)

setup (mode, pull=None)

Setup the pin

Parameters:

- **mode** (*int*) – pin mode(IN/OUT)
- **pull** (*int*) – pin pull up/down(PUD_UP/PUD_DOWN/PUD_NONE)

dict (_dict=None)

Set/get the pin dictionary

Parameters: **_dict** (*dict*) – pin dictionary, leave it empty to get the dictionary

Returns: pin dictionary

Return type: dict

__call__ (value)

Set/get the pin value

Parameters: **value** (*int*) – pin value, leave it empty to get the value(0/1)

Returns: pin value(0/1)

Return type: int

value (value: bool = None)

Set/get the pin value

Parameters: **value** (*int*) – pin value, leave it empty to get the value(0/1)

Returns: pin value(0/1)

Return type: int

on ()

Set pin on(high)

Returns: pin value(1)

Return type: int

off ()

Set pin off(low)

Returns: pin value(0)

Return type: int

high ()

Set pin high(1)

Returns: pin value(1)

Return type: int

low ()

Set pin low(0)

Returns: pin value(0)

Return type: int

irq (handler, trigger, bouncetime=200, pull=None)

Set the pin interrupt

Parameters:

- **handler** (*function*) – interrupt handler callback function
- **trigger** (*int*) – interrupt trigger(RISING, FALLING, RISING_FALLING)
- **bouncetime** (*int*) – interrupt bouncetime in miliseconds

name ()

Get the pin name

Returns: pin name**Return type:** str**class** ADC**Example**

```
# Import ADC class
from bella_hat.adc import ADC

# Create ADC object with numeric pin numbering
a0 = ADC(0)
# Create ADC object with named pin numbering
a1 = ADC('A1')

# Read ADC value
value0 = a0.read()
value1 = a1.read()
voltage0 = a0.read_voltage()
voltage1 = a1.read_voltage()
print(f"ADC 0 value: {value0}")
print(f"ADC 1 value: {value1}")
print(f"ADC 0 voltage: {voltage0}")
print(f"ADC 1 voltage: {voltage1}")
```

Read Battery voltage on Bella bella-hat:

The battery voltage measurement terminal is connected to ADC channel 4 and divided by 3 times

```
# Import ADC class
from bella_hat.adc import ADC

BAT_VOLT_GAIN = 3
bat = ADC('A4')

bat_value = bat.read()
bat_voltage = bat.read_voltage() * BAT_VOLT_GAIN
print(f"Battery adc value: {bat_value}, voltage: {bat_voltage:.2f} V")
```

API

```
class bella_hat.adc.ADC (chn, address=None, *args, **kwargs)
```

Bases: **I2C**

Analog to digital converter

```
__init__ (chn, address=None, *args, **kwargs)
```

Analog to digital converter

Parameters: **chn** (*int/str*) – channel number (0-7/A0-A7)**read ()**

Read the ADC value

Returns: ADC value(0-4095)

Return type: int

read_voltage ()

Read the ADC value and convert to voltage

Returns: Voltage value(0-3.3(V))

Return type: float

class PWM

Example

```
# Import PWM class
from bella_hat.pwm import PWM

# Create PWM object with numeric pin numbering and default input pullup enabled
p0 = PWM(0)
# Create PWM object with named pin numbering
p1 = PWM('P1')

# Set frequency will automatically set prescaler and period
# This is easy for device like Buzzer or LED, which you care
# about the frequency and pulse width percentage.
# this usually use with pulse_width_percent function.
# Set frequency to 1000Hz
p0.freq(1000)
print(f"Frequency: {p0.freq()} Hz")
print(f"Prescaler: {p0.prescaler()}")
print(f"Period: {p0.period()}")
# Set pulse width to 50%
p0.pulse_width_percent(50)

# Or set prescaler and period, will get a frequency from:
# frequency = PWM.CLOCK / prescaler / period
# With this setup you can tune the period as you wish.
# set prescaler to 64
p1.prescaler(64)
# set period to 4096 ticks
p1.period(4096)
print(f"Frequency: {p1.freq()} Hz")
print(f"Prescaler: {p1.prescaler()}")
print(f"Period: {p1.period()}")
# Set pulse width to 2048 which is also 50%
p1.pulse_width(2048)
```

API

`class bella_hat.pwm.PWM(channel, address=None, *args, **kwargs)`

Bases: **I2C**

Pulse width modulation (PWM)

REG_CHN = 32

Channel register prefix

REG_PSC = 64

Prescaler register prefix

REG_ARR = 68

Period register prefix

REG_PSC2 = 80
Prescaler register prefix

REG_ARR2 = 84
Period register prefix

CLOCK = 72000000.0
Clock frequency

__init__(channel, address=None, *args, **kwargs)
Initialize PWM

Parameters: **channel** (*int/str*) – PWM channel number(0-13/P0-P13)

freq(freq=None)
Set/get frequency, leave blank to get frequency

Parameters: **freq** (*float*) – frequency(0-65535)(Hz)

Returns: frequency

Return type: float

prescaler(prescaler=None)
Set/get prescaler, leave blank to get prescaler

Parameters: **prescaler** (*int*) – prescaler(0-65535)

Returns: prescaler

Return type: int

period(arr=None)
Set/get period, leave blank to get period

Parameters: **arr** (*int*) – period(0-65535)

Returns: period

Return type: int

pulse_width(pulse_width=None)
Set/get pulse width, leave blank to get pulse width

Parameters: **pulse_width** (*float*) – pulse width(0-65535)

Returns: pulse width

Return type: float

pulse_width_percent(pulse_width_percent=None)
Set/get pulse width percentage, leave blank to get pulse width percentage

Parameters: **pulse_width_percent** (*float*) – pulse width percentage(0-100)

Returns: pulse width percentage

Return type: float

module motor

class Motors

Example

Initilize

```
# Import Motor class
from bella_hat.motor import Motors
```

```
# Create Motor object
motors = Motors()
# Create Motor object with setting motors direction
# motors = Motors(left_reversed=True, right_reversed=False)
```

Control motors power.

```
#
motors.speed([50, 50])
# stop
motors.stop()
```

Setup motor direction.

```
# Go forward and see if both motor directions are correct
motors.forward(100)
# if you found a motor is running in the wrong direction
# Use these function to correct it
motors.reverse(True, True) # [left_reverse, right_reverse]
# Run forward again and see if both motor directions are correct
motors.forward(100)
```

Move functions

```
import time

motors.forward(100)
time.sleep(1)
motors.backward(100)
time.sleep(1)
motors.turn_left(100)
time.sleep(1)
motors.turn_right(100)
time.sleep(1)
motors.stop()
```

API

```
class bella_hat.motor.Motors (left_motor_pwm=18, left_motor_pwm=19, right_motor_pwm=16,
right_motor_pwm=17, left_reversed=False, right_reversed=False, *args, **kwargs)
```

Bases: `_Basic_class`

```
__init__ (left_motor_pwm=18, left_motor_pwm=19, right_motor_pwm=16,
right_motor_pwm=17, left_reversed=False, right_reversed=False, *args, **kwargs)
Initialize motors with bella_hat.motor.Motor
```

Parameters:

- **left_motor_pwm** (*int, pin number*) – pwm input for left motor
- **left_motor_pwm** (*int, pin number*) – pwm input for left motor
- **right_motor_pwm** (*int, pin number*) – pwm input for right motor
- **right_motor_pwm** (*int, pin number*) – pwm input for right motor
- **left_reversed** (*bool*) – whether to reverse left motor
- **right_reversed** (*bool*) – whether to reverse right motor

```
stop ()
Stop all motors
```

```
brake ()
Brake
```

```
reverse (reverse=None)
```

Get or set motors is reversed or not

Parameters: **reverse** (*[bool, bool]*) – [left_reverse, right_reverse]

speed (*speed=None*)

Get or Set motors speed

Parameters: **speed** (*[float/int, float/int]*) – [left_speed, right_speed] (-100.0~100.0)

forward (*speed*)

Forward

Parameters: **speed** (*float*) – Motor speed(-100.0~100.0)

backward (*speed*)

Backward

Parameters: **speed** (*float*) – Motor speed(-100.0~100.0)

turn_left (*speed*)

Left turn

Parameters: **speed** (*float*) – Motor speed(-100.0~100.0)

turn_right (*speed*)

Right turn

Parameters: **speed** (*float*) – Motor speed(-100.0~100.0)

```
class Motor
```

Example

```
# Import Motor class
from bella_hat.motor import Motor, PWM, Pin

# Create Motor object
motor = Motor(18, 19) # pwma, pwmb

# Motor clockwise at 100% speed
motor.speed(100)
# Motor counter-clockwise at 100% speed
motor.speed(-100)

# If you like to reverse the motor direction
motor.reverse(True)
```

API

```
class bella_hat.motor.Motor (pwma, pwmb, reversed=False)
```

Bases: **object**

__init__ (*pwma, pwmb, reversed=False*)

Initialize a motor

Parameters:

- **pwma** (*pin number*) – Motor speed control pwm input A pin
- **pwmb** (*pin number*) – Motor speed control pwm input B pin

Reversed: Whether to reverse the direction of motor rotation

speed (*speed=None*)

Get or set motor speed

Parameters: **speed** (*int or float*) – Motor speed(-100.0~100.0)

reverse (*reverse=None*)

Get or set motor is reversed or not

Parameters: **reverse** (*bool*) – True or False

brake ()

stop ()

class I2C

Example

```
# Import the I2C class
from bella_hat.i2c import I2C

# You can scan for available I2C devices
print([f"0x{addr:02X}" for addr in I2C().scan()])
# You should see at least one device address 0x14, which is the
# on board MCU for PWM and ADC

# Initialize a I2C object with device address, for example
# to communicate with on board MCU 0x14
mcu = I2C(0x14)
# Send ADC channel register to read ADC, 0x10 is Channel 0, 0x11 is Channel 1, etc.
mcu.write([0x10, 0x00, 0x00])
# Read 2 byte for MSB and LSB
msb, lsb = mcu.read(2)
# Convert to integer
value = (msb << 8) + lsb
# Print the value
print(value)
```

For more information on the I2C protocol, see checkout [adc.py](#) and [pwm.py](#)

API

`class bella_hat.i2c.I2C (address=None, bus=1, *args, **kwargs)`

Bases: `_Basic_class`

I2C bus read/write functions

`__init__ (address=None, bus=1, *args, **kwargs)`

Initialize the I2C bus

Parameters:

- **address** (*int*) – I2C device address
- **bus** (*int*) – I2C bus number

static scan (busnum=1, force=False)

Scan the I2C bus for devices

Returns: List of I2C addresses of devices found

Return type: list

write (data)

Write data to the I2C device

Parameters: **data** (*int/list/bytearray*) – Data to write

Raises: `ValueError` if write is not an int, list or bytearray

read (length=1)

Read data from I2C device

Parameters: **length** (*int*) – Number of bytes to receive**Returns:** Received data**Return type:** list**mem_write** (data, memaddr)

Send data to specific register address

Parameters:

- **data** (*int/list/bytearray*) – Data to send, int, list or bytearray
- **memaddr** (*int*) – Register address

Raises: **ValueError** – If data is not int, list, or bytearray**mem_read** (length, memaddr)

Read data from specific register address

Parameters:

- **length** (*int*) – Number of bytes to receive
- **memaddr** (*int*) – Register address

Returns: Received bytearray data or False if error**Return type:** list/False**is_avaliabile** ()

Check if the I2C device is available

Returns: True if the I2C device is available, False otherwise**Return type:** bool**module** modules**class** Ultrasonic**Example**

```
# Import Ultrasonic and Pin class
from bella_hat.modules import Ultrasonic, Pin

# Create Motor object
us = Ultrasonic(Pin(20), Pin(21))

# Read distance
distance = us.read()
print(f"Distance: {distance}cm")
```

API**class** bella_hat.modules.**Ultrasonic** (trig, echo, timeout=0.02)**__init__** (trig, echo, timeout=0.02)

Initialize an ultrasonic distance sensor

Parameters:

- **trig** (*pin number*) – trig pin
- **echo** (*pin number*) – echo pin
- **timeout** (*float, seconds*) – set the timeout for detecting the return of sound waves

read (times=10)
Read the distance

Parameters: **times** (*int*) – retry times

Returns: float, distance in centimeter 1, timeout or error

```
class Grayscale_Module
```

Example

```
# Import Grayscale_Module and ADC class
from bella_hat.modules import Grayscale_Module, ADC

# Create Grayscale_Module object, reference should be calculate from the value reads on v
# and black ground, then take the middle as reference
gs = Grayscale_Module(ADC(0), ADC(1), ADC(2), reference=[1000, 900, 1000])

# Read Grayscale_Module datas
datas = gs.read()
print(f"Grayscale Module datas: {datas}")
# or read a specific channel
l = gs.read(gs.LEFT)
m = gs.read(gs.MIDDLE)
r = gs.read(gs.RIGHT)
print(f"Grayscale Module left channel: {l}")
print(f"Grayscale Module middle channel: {m}")
print(f"Grayscale Module right channel: {r}")

# Read Grayscale_Module simple states
state = gs.read_status()
print(f"Grayscale_Module state: {state}")
```

API

```
class bella_hat.modules.Grayscale_Module (pin0: ADC, pin1: ADC, pin2: ADC, reference:
int = None)
```

3 channel Grayscale Module

LEFT = 2
Left Channel

MIDDLE = 1
Middle Channel

RIGHT = 1
Right Channel

__init__ (pin0: ADC, pin1: ADC, pin2: ADC, reference: int = None)
Initialize Grayscale Module

Parameters:

- **pin0** (*bella_hat.ADC/int*) – ADC object or int for channel 0
- **pin1** (*bella_hat.ADC/int*) – ADC object or int for channel 1
- **pin2** (*bella_hat.ADC/int*) – ADC object or int for channel 2
- **reference** (*1*3 list, [int, int, int]*) – reference voltage

reference (ref: list = None) → list
Get Set reference value

Parameters: **ref** (*list*) – reference value, None to get reference value

Returns: reference value

Return type: list

read_status (datas: list = None) → list
Read line status

Parameters: **datas** (*list*) – list of grayscale datas, if None, read from sensor

Returns: list of line status, 0 for white, 1 for black

Return type: list

read (channel: int = None) → list
read a channel or all datas

Parameters: **channel** (*int/None*) – channel to read, leave empty to read all. 0, 1, 2 or Grayscale_Module.LEFT, Grayscale_Module.CENTER, Grayscale_Module.RIGHT

Returns: list of grayscale data

Return type: list

class DHT11

Example

```
# Import DHT11 class
from bella_hat.modules import DHT11

# Create DHT11 object
dht11 = DHT11(19)

# Read DHT11 datas
temperature = dht11.temperature
humidity = dht11.humidity
print(f"Temperature: {temperature}'C, Humidity: {humidity}%")
```

API

class bella_hat.modules.DHT11 (pin)

__init__ (pin)
Initialize DHT11 Module

Parameters: **pin** (*pin number*) – DHT11 data pin

property **temperature**
Temperature

property **humidity**
Humidity

class LSM6DSOX

Example

```
# Import LSM6DSOX class
from bella_hat.modules import LSM6DSOX

# Create LSM6DSOX object
lsm6dsox = LSM6DSOX()

# Read LSM6DSOX datas
acc = lsm6dsox.acc
```

```
gyro = lsm6dsox.gyro
print(f"acc: X:{acc[0]:.2f},      Y: {acc[1]:.2f},      Z: {acc[2]:.2f} m/s^2")
print(f"gyro X:{gyro[0]:.2f},    Y: {gyro[1]:.2f},    Z: {gyro[2]:.2f} radians/s")
```

API

`class bella_hat.modules.LSM6DSOX`

```
__init__()
    Initialize LSM6DSOX Module

property acc
    Acceleration

property gyro
    Gyro
```

class Music

Example

Initialize

```
# Import Music class
from bella_hat.music import Music

# Create a new Music object
music = Music()
```

Play sound

```
# Play a sound
music.sound_play("file.wav", volume=50)
# Play a sound in the background
music.sound_play_threading("file.wav", volume=80)
# Get sound length
music.sound_length("file.wav")
```

Play Music

```
# Play music
music.music_play("file.mp3")
# Play music in loop
music.music_play("file.mp3", loop=0)
# Play music in 3 times
music.music_play("file.mp3", loop=3)
# Play music in starts from 2 second
music.music_play("file.mp3", start=2)
# Set music volume
music.music_set_volume(50)
# Stop music
music.music_stop()
# Pause music
music.music_pause()
# Resume music
music.music_resume()
```

API

`class bella_hat.music.Music`
 Bases: `_Basic_class`
 Play music, sound affect and note control

`NOTE_BASE_FREQ = 440`

Base note frequency for calculation (A4)

NOTE_BASE_INDEX = 69

Base note index for calculation (A4) MIDI compatible

[illegible]

Notes name, MIDI compatible

```
__init__()
```

Initialize music

```
time_signature(top: int = None, bottom: int = None)
```

Set/get time signature

Parameters:

- **top** (*int*) – top number of time signature
- **bottom** (*int*) – bottom number of time signature

Returns: time signature

Return type: tuple

```
key_signature (key: int = None)
```

Set/get key signature

Parameters: **key** (*int/str*) – key signature use KEY_XX_MAJOR or String “#”, “##”, or “bbb”, “bbbb”

Returns: key signature

Return type: int

```
tempo (tempo=None, note_value=0.25)
```

Set/get tempo beat per minute(bpm)

Parameters:

- **tempo** (*float*) – tempo
- **note_value** – note value(1, 1/2, Music.HALF_NOTE, etc)

Returns: tempo

Return type: int

beat (beat)

Calculate beat delay in seconds from tempo

Parameters: **beat** (*float*) – beat index

Returns: beat delay

Return type: float

note (note, natural=False)

Get frequency of a note

Parameters:

- **note_name** (*string*) – note name(See NOTES)
- **natural** (*bool*) – if natural note

Returns: frequency of note

Return type: float

```
sound_play(filename, volume=None)
```

Play sound effect file

Parameters: **filename** (*str*) – sound effect file name

sound_play_threading (filename, volume=None)

Play sound effect in thread(in the background)

Parameters:

- **filename** (*str*) – sound effect file name
- **volume** (*int*) – volume 0-100, leave empty will not change volume

music_play (filename, loops=1, start=0.0, volume=None)

Play music file

Parameters:

- **filename** (*str*) – sound file name
- **loops** (*int*) – number of loops, 0:loop forever, 1:play once, 2:play twice, ...
- **start** (*float*) – start time in seconds
- **volume** (*int*) – volume 0-100, leave empty will not change volume

music_set_volume (value)

Set music volume

Parameters: **value** (*int*) – volume 0-100

music_stop ()

Stop music

music_pause ()

Pause music

music_resume ()

Resume music

music_unpause ()

Unpause music(resume music)

sound_length (filename)

Get sound effect length in seconds

Parameters: **filename** (*str*) – sound effect file name

Returns: length in seconds

Return type: float

get_tone_data (freq: float, duration: float)

Get tone data for playing

Parameters:

- **freq** (*float*) – frequency
- **duration** (*float*) – duration in seconds

Returns: tone data

Return type: list

play_tone_for (freq, duration)

Play tone for duration seconds

Parameters:

- **freq** (*float*) – frequency, you can use NOTES to get frequency
- **duration** (*float*) – duration in seconds

module `utils`

`bella_hat.utils.set_volume (value)`
Set volume

Parameters: `value (int)` – volume(0~100)

`bella_hat.utils.run_command (cmd)`
Run command and return status and output

Parameters: `cmd (str)` – command to run

Returns: status, output

Return type: tuple

`bella_hat.utils.is_installed (cmd)`
Check if command is installed

Parameters: `cmd (str)` – command to check

Returns: True if installed

Return type: bool

`bella_hat.utils.mapping (x, in_min, in_max, out_min, out_max)`
Map value from one range to another range

Parameters:

- `x (float/int)` – value to map
- `in_min (float/int)` – input minimum
- `in_max (float/int)` – input maximum
- `out_min (float/int)` – output minimum
- `out_max (float/int)` – output maximum

Returns: mapped value

Return type: float/int

`bella_hat.utils.get_ip (ifaces=['wlan0', 'eth0'])`
Get IP address

Parameters: `ifaces (list)` – interfaces to check

Returns: IP address or False if not found

Return type: str/False

`bella_hat.utils.reset_mcu ()`

Reset mcu on BL100 Hat.

This is helpful if the mcu somehow stuck in a I2C data transfer loop, and Raspberry Pi getting IOError while Reading ADC, manipulating PWM, etc.

`bella_hat.utils.get_battery_voltage ()`
Get battery voltage

Returns: battery voltage(V)

Return type: float

class `_Basic_class`

`_Basic_class` is a logger class for all class to log, so if you want to see logs of a class, just add a debug argument to it.

Example

```
# See PWM log
from bella_hat.pwm import PWM

# init the class with a debug argument
```

```
pwm = PWM(0, debug_level="debug")  
  
# run some functions and see logs  
pwm.freq(1000)  
pwm.pulse_width_percent(100)
```

API

```
class bella_hat.basic._Basic_class(debug_level='warning')  
    Basic Class for all classes  
    with debug function
```

```
DEBUG_LEVELS = {'critical': 50, 'debug': 10, 'error': 40, 'info': 20, 'warning': 30}  
    Debug level
```

```
DEBUG_NAMES = ['critical', 'error', 'warning', 'info', 'debug']  
    Debug level names
```

```
__init__(debug_level='warning')  
    Initialize the basic class
```

Parameters: **debug_level** (*str/int*) – debug level, 0(critical), 1(error), 2(warning), 3(info) or 4(debug)

```
property debug_level  
    Debug level
```


Index

`__call__()` (bella_hat.pin.Pin method)
`__init__()` (bella_hat.adc.ADC method)
(bella_hat.basic._Basic_class method)
(bella_hat.bella.Bella method)
(bella_hat.i2c.I2C method)
(bella_hat.modules.DHT11 method)
(bella_hat.modules.Grayscale_Module method)
(bella_hat.modules.LSM6DSOX method)
(bella_hat.modules.Ultrasonic method)
(bella_hat.motor.Motor method)
(bella_hat.motor.Motors method)
(bella_hat.music.Music method)
(bella_hat.pin.Pin method)
(bella_hat.pwm.PWM method)
`_Basic_class` (class in bella_hat.basic)

A

`acc` (bella_hat.modules.LSM6DSOX property)
`ADC` (class in bella_hat.adc)

B

`backward()` (bella_hat.motor.Motors method)
`beat()` (bella_hat.music.Music method)
`Bella` (class in bella_hat.bella)

bella_hat

module

bella_hat.utils

module

`brake()` (bella_hat.motor.Motor method)
(bella_hat.motor.Motors method)

C

`CLOCK` (bella_hat.pwm.PWM attribute)

D

`debug_level` (bella_hat.basic._Basic_class property)
`DEBUG_LEVELS` (bella_hat.basic._Basic_class attribute)
`DEBUG_NAMES` (bella_hat.basic._Basic_class attribute)
`DHT11` (class in bella_hat.modules)

`dict()` (bella_hat.pin.Pin method)

F

`fan_off()` (bella_hat.bella.Bella method)
`fan_on()` (bella_hat.bella.Bella method)
`forward()` (bella_hat.motor.Motors method)
`freq()` (bella_hat.pwm.PWM method)

G

`get_acc()` (bella_hat.bella.Bella method)
`get_battery_percentage()` (bella_hat.bella.Bella method)
`get_battery_voltage()` (bella_hat.bella.Bella method)
(in module bella_hat.utils)
`get_grayscales()` (bella_hat.bella.Bella method)
`get_gyro()` (bella_hat.bella.Bella method)
`get_humidity()` (bella_hat.bella.Bella method)
`get_ip()` (in module bella_hat.utils)
`get_temperature()` (bella_hat.bella.Bella method)
`get_tone_data()` (bella_hat.music.Music method)
`get_ultrasonic_distance()` (bella_hat.bella.Bella method)
`Grayscale_Module` (class in bella_hat.modules)
`gyro` (bella_hat.modules.LSM6DSOX property)

H

`high()` (bella_hat.pin.Pin method)
`humidity` (bella_hat.modules.DHT11 property)

I

`I2C` (class in bella_hat.i2c)
`IN` (bella_hat.pin.Pin attribute)
`irq()` (bella_hat.pin.Pin method)
`IRQ_FALLING` (bella_hat.pin.Pin attribute)
`IRQ_RISING` (bella_hat.pin.Pin attribute)
`IRQ_RISING_FALLING` (bella_hat.pin.Pin attribute)
`is_avaiable()` (bella_hat.i2c.I2C method)
`is_installed()` (in module bella_hat.utils)

K

`key_signature()` (bella_hat.music.Music method)

L

`LEFT` (bella_hat.modules.Grayscale_Module attribute)

low() (bella_hat.pin.Pin method)
LSM6DSOX (class in bella_hat.modules)

M

mapping() (in module bella_hat.utils)
mem_read() (bella_hat.i2c.I2C method)
mem_write() (bella_hat.i2c.I2C method)
MIDDLE (bella_hat.modules.Grayscale_Module attribute)
module
 bella_hat
 bella_hat.utils
Motor (class in bella_hat.motor)
Motors (class in bella_hat.motor)
Music (class in bella_hat.music)
music_pause() (bella_hat.music.Music method)
music_play() (bella_hat.music.Music method)
music_resume() (bella_hat.music.Music method)
music_set_volume() (bella_hat.music.Music method)
music_stop() (bella_hat.music.Music method)
music_unpause() (bella_hat.music.Music method)

N

name() (bella_hat.pin.Pin method)
note() (bella_hat.music.Music method)
NOTE_BASE_FREQ (bella_hat.music.Music attribute)
NOTE_BASE_INDEX (bella_hat.music.Music attribute)
NOTES (bella_hat.music.Music attribute)

O

off() (bella_hat.pin.Pin method)
on() (bella_hat.pin.Pin method)
OUT (bella_hat.pin.Pin attribute)

P

period() (bella_hat.pwm.PWM method)
Pin (class in bella_hat.pin)
play_tone_for() (bella_hat.music.Music method)
prescaler() (bella_hat.pwm.PWM method)
PULL_DOWN (bella_hat.pin.Pin attribute)
PULL_NONE (bella_hat.pin.Pin attribute)
PULL_UP (bella_hat.pin.Pin attribute)
pulse_width() (bella_hat.pwm.PWM method)
pulse_width_percent() (bella_hat.pwm.PWM method)

PWM (class in bella_hat.pwm)

R

read() (bella_hat.adc.ADC method)
 (bella_hat.i2c.I2C method)
 (bella_hat.modules.Grayscale_Module method)
 (bella_hat.modules.Ultrasonic method)
read_btn() (bella_hat.bella.Bella method)
read_status() (bella_hat.modules.Grayscale_Module method)
read_voltage() (bella_hat.adc.ADC method)
reference() (bella_hat.modules.Grayscale_Module method)
REG_ARR (bella_hat.pwm.PWM attribute)
REG_ARR2 (bella_hat.pwm.PWM attribute)
REG_CHN (bella_hat.pwm.PWM attribute)
REG_PSC (bella_hat.pwm.PWM attribute)
REG_PSC2 (bella_hat.pwm.PWM attribute)
reset_mcu() (in module bella_hat.utils)
reverse() (bella_hat.motor.Motor method)
 (bella_hat.motor.Motors method)
RIGHT (bella_hat.modules.Grayscale_Module attribute)
run_command() (in module bella_hat.utils)

S

scan() (bella_hat.i2c.I2C static method)
set_eyes_led() (bella_hat.bella.Bella method)
set_motors() (bella_hat.bella.Bella method)
set_motors_reverse() (bella_hat.bella.Bella method)
set_volume() (in module bella_hat.utils)
setup() (bella_hat.pin.Pin method)
sound_length() (bella_hat.music.Music method)
sound_play() (bella_hat.music.Music method)
sound_play_threading() (bella_hat.music.Music method)
speed() (bella_hat.motor.Motor method)
 (bella_hat.motor.Motors method)
stop() (bella_hat.motor.Motor method)
 (bella_hat.motor.Motors method)

T

temperature (bella_hat.modules.DHT11 property)
tempo() (bella_hat.music.Music method)

[time_signature\(\)](#) ([bella_hat.music.Music](#) method)

[turn_left\(\)](#) ([bella_hat.motor.Motors](#) method)

[turn_right\(\)](#) ([bella_hat.motor.Motors](#) method)

U

[Ultrasonic](#) (class in [bella_hat.modules](#))

V

[value\(\)](#) ([bella_hat.pin.Pin](#) method)

W

[write\(\)](#) ([bella_hat.i2c.I2C](#) method)

Python Module Index

b

[bella_hat](#)

[bella_hat.utils](#)