SunFounder Robot HAT

www.sunfounder.com

Aug 03, 2023

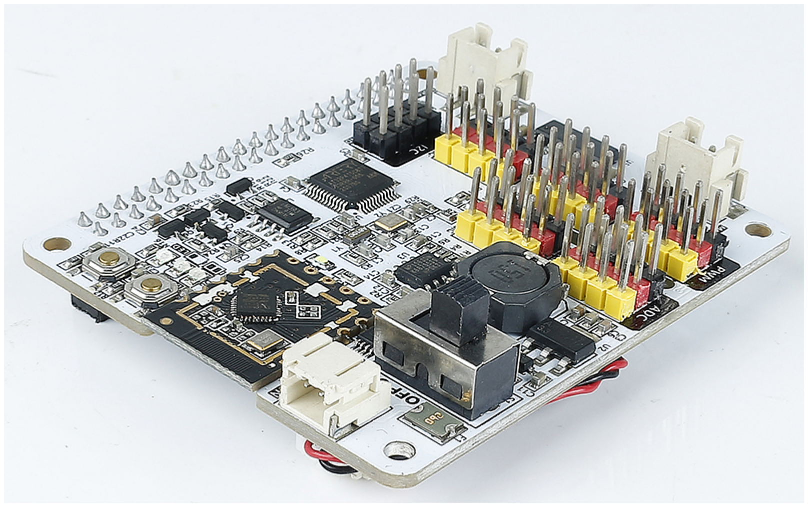
CONTENTS

Robot HAT is a multifunctional expansion board that allows Raspberry Pi to be quickly turned into a robot. An MCU is on board to extend the PWM output and ADC input for the Raspberry Pi, as well as a motor driver chip, Bluetooth module, I2S audio module and mono speaker. As well as the GPIOs that lead out of the Raspberry Pi itself.

It also comes with a Speaker, which can be used to play background music, sound effects and implement TTS functions to make your project more interesting.

Accepts 7-12V PH2.0 2pin power input with 2 power indicators. The board also has a user available LED and a button for you to quickly test some effects.

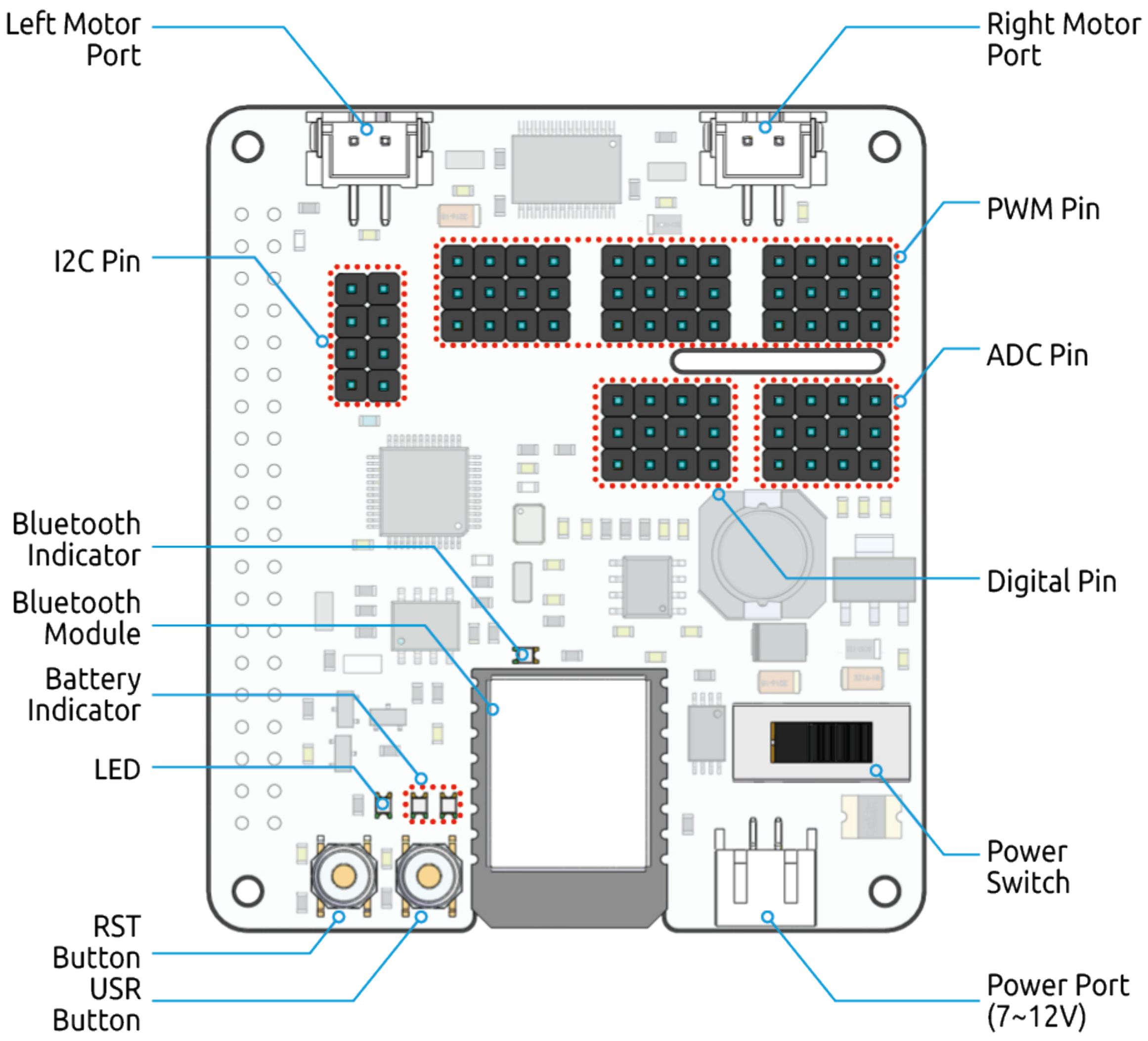
In this document, you will get a full understanding of the interface functions of the Robot HAT and the usage of these interfaces through the Python `robot-hat` library provided by SunFounder.



CHAPTER

ONE

HARDWARE INTRODUCTION



Left/Right Motor Port

- 2-channel XH2.54 motor ports.
- The left port is connected to GPIO 4 and the right port is connected to GPIO 5.
- API: *class Motor - motor control*, `0` for left motor port, `1` for right motor port.

I2C Pin

- 2-channel I2C pins from Raspberry Pi.
- API: *class i2c - IIC Bus*

PWM Pin

- 12-channel PWM pins, P0-P12.
- API: *class PWM - pulse width modulation*

ADC Pin

- 4-channel ADC pins, A0-A3.
- API: *class ADC - analog to digital converter*

Digital Pin

- 4-channel digital pins, D0-D3.
- API: *class Pin - control I/O pins*

Battery Indicator

- Two LEDs light up when the voltage is higher than 7.8V.
- One LED lights up in the 6.7V to 7.8V range.
- Below 6.7V, both LEDs turn off.

LED

- Set by your program. (Outputting 1 turns the LED on; Outputting 0 turns it off.)
- API: *class Pin - control I/O pins*, you can use `Pin("LED")` to define it.

RST Button

- Short pressing RST Button causes program resetting.
- Long press RST Button till the LED lights up then release, and you will disconnect the Bluetooth.

USR Button

- The functions of USR Button can be set by your programming. (Pressing down leads to a input "0"; releasing produces a input "1".)
- API: *class Pin - control I/O pins*, you can use `Pin("SW")` to define it.

Power Switch

- Turn on/off the power of the robot HAT.
- When you connect power to the power port, the Raspberry Pi will boot up. However, you will need to switch the power switch to ON to enable Robot HAT.

Power Port

- 7-12V PH2.0 2pin power input.
- Powering the Raspberry Pi and Robot HAT at the same time.

Bluetooth Module

- Since the Raspberry Pi comes with Bluetooth in slave mode, there will be pairing problems when connecting with cell phones. To make it easier for the Raspberry Pi to connect to the Ezblock Studio, we added a separate Bluetooth module.

- Ezblock Studio is a custom graphical programming application developed by SunFounder for Raspberry Pi, for more information please refer to: Ezblock Studio 3.

Bluetooth Indicator

- The Bluetooth indicator keeps turning on at a well Bluetooth connection, blink at a Bluetooth disconnection, blink fast at a signal transmission.

CHAPTER

TWO

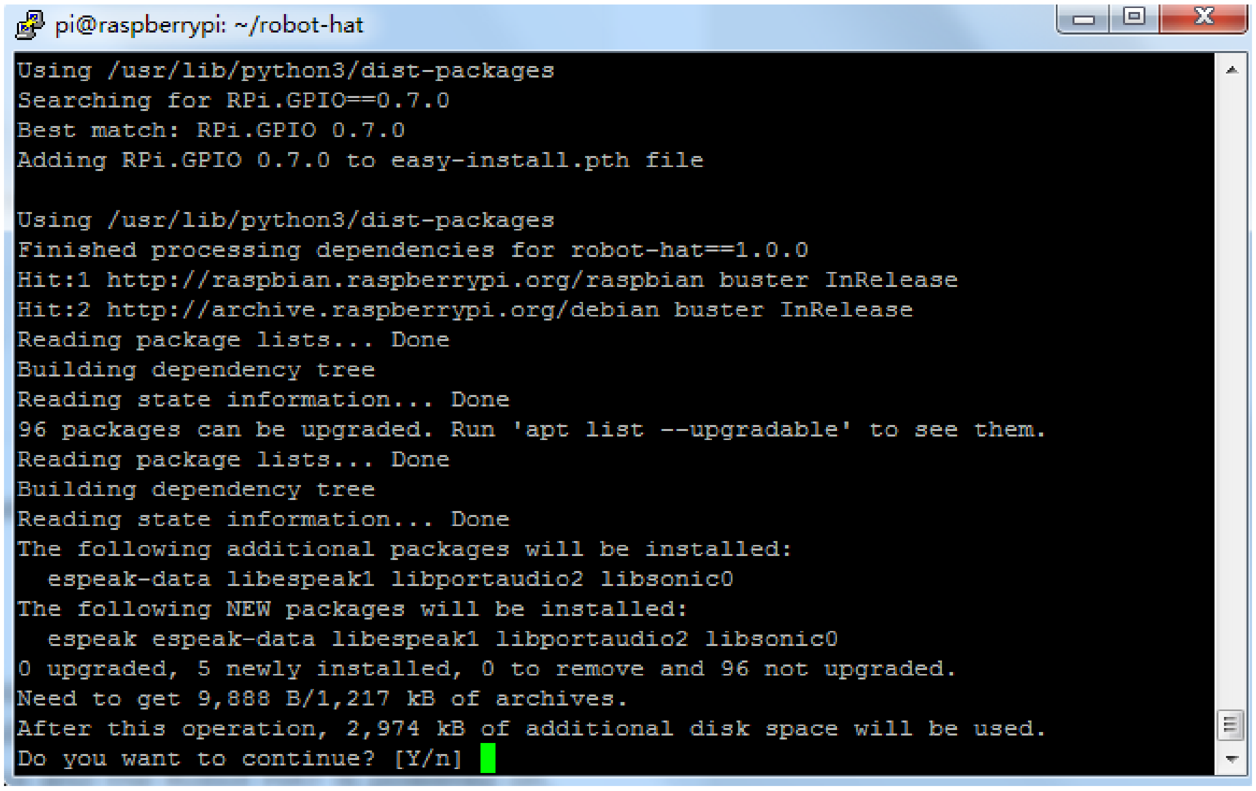
INSTALL THE ROBOT-HAT

`robot-hat` is the supported library for the Robot HAT.

Type this command into the terminal to install the Robot HAT package.

```
git clone https://github.com/sunfounder/robot-hat.git
cd robot-hat
sudo python3 setup.py install
```

Note: Run setup.py to download some necessary components. You may have a network problem and the download may fail. At this point you may need to download again. In the following cases, type Y and press Enter to continue the process.



```
Using /usr/lib/python3/dist-packages
Searching for RPi.GPIO==0.7.0
Best match: RPi.GPIO 0.7.0
Adding RPi.GPIO 0.7.0 to easy-install.pth file

Using /usr/lib/python3/dist-packages
Finished processing dependencies for robot-hat==1.0.0
Hit:1 http://raspbian.raspberrypi.org/raspbian buster InRelease
Hit:2 http://archive.raspberrypi.org/debian buster InRelease
Reading package lists... Done
Building dependency tree
Reading state information... Done
96 packages can be upgraded. Run 'apt list --upgradable' to see them.
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  espeak-data libespeak1 libportaudio2 libsonic0
The following NEW packages will be installed:
  espeak espeak-data libespeak1 libportaudio2 libsonic0
0 upgraded, 5 newly installed, 0 to remove and 96 not upgraded.
Need to get 9,888 B/1,217 kB of archives.
After this operation, 2,974 kB of additional disk space will be used.
Do you want to continue? [Y/n]
```

CHAPTER

THREE

ROBOT HAT API REFERENCE

3.1 Basic

3.1.1 class `Pin` - control I/O pins

Usage

```
from robot_hat import Pin

pin = Pin("D0")                      # create a Pin object from a digital pin
val = pin.value()                    # read the value on the digital pin

pin.value(0)                         # set the digital pin to low level
```

Constructors

`class robot_hat.Pin(value)` : The `Pin()` is the basic object to control I/O pins. It has methods to set the pin mode (input, output, etc.) and methods to get or set the level of a digital pin.

Methods

- `value` - Read or set the value on the digital pin, the value is 0/1.

```
Pin.value() # Without parameters, read gpio level, high level returns 1, low level␣
↪returns 0.

Pin.value(0)  # set to low level
Pin.value(1)  # set to high level
```

- Set the value to the digital pin, same as `value`.

```
Pin.on() # set to high level
#off() # set to low level
#high() # set to high level
#low() # set to low level
```

- `mode` - Set the mode of GPIO to `IN` or `OUT`.

```
Pin.mode(GPIO.IN)  # set gipo to INPUT mode
```

Availble Pins

- `"D0"`: The Digital pin 0.
- `"D1"`: The Digital pin 1.

- `"D2"`: The Digital pin 2.
- `"D3"`: The Digital pin 3.
- `"D4"`: The left motor pin.
- `"D5"`: The right motor pin.
- `"D6"`
- `"D7"`
- `"D8"`
- `"D9"`
- `"SW"`: The USR button.
- `"LED"`: The LED on the board.

3.1.2 class `ADC` - analog to digital converter

Usage

```
from robot_hat import ADC

adc = ADC("A0")                     # create an analog object from a pin
val = adc.read()                    # Read values from analog pins
```

Constructors

`class robot_hat.ADC(pin)`: Create an ADC object associated with the given pin. This allows you to then read analog values on that pin.

Methods

- `read` - Read the value on the analog pin and return it. The returned value will be between 0 and 4095.

```
ADC.read()
```

3.1.3 class `PWM` - pulse width modulation

Usage

```
from robot_hat import PWM

pwm = PWM('P0')                                 # create a PWM object from a pin
PWM.freq(*freq)                                 #Frequency (0-65535, unit Hz)
PWM.prescaler(*prescaler)                       #Prescaler
PWM.period(*arr)
PWM.pulse_width(*pulse_width)                   # Pulse width (pulse_width < period)
PWM.pulse_width_percent(*pulse_width_percent)   # Duty cycle (0-100)
```

Constructors

`class robot_hat.PWM(channel)`: Create a PWM object associated with the given pin. This allows you set up the pwm function on that pin.

Methods

- `freq` - Set the frequency of the pwm channel.

```
PWM.freq(50)
```

- `prescaler` - Set the prescaler for the pwm channel.

```
PWM.prescaler(50)
```

- `period` - Set the period of the pwm channel.

```
PWM.period(100)
```

- `pulse_width` - Set the pulse width of the pwm channel.

```
PWM.pulse_width(10)
```

- `pulse_width_percent` - Sets the pulse width percentage of the pwm channel.

```
PWM.pulse_width_percent(50)
```

3.1.4 class `i2c` - IIC Bus

Usage

```
from robot_hat import I2C

i2c = I2C(1)                          # create on bus 1
i2c = I2C(1, I2C.MASTER)              # create and init as a master

i2c.send('abc')      # send 3 bytes
i2c.send(0x42)       # send a single byte, given by the number
data = i2c.recv(3)   # receive 3 bytes

i2c.is_ready(0x42)           # check if slave 0x42 is ready
i2c.scan()                   # scan for slaves on the bus, returning a list of valid␣
↪addresses
i2c.mem_read(3, 0x42, 2)     # read 3 bytes from memory of slave 0x42, starting at␣
↪address 2 in the slave
i2c.mem_write('abc', 0x42, 2, timeout=1000) # write 'abc' (3 bytes) to memory of␣
↪slave 0x42, starting at address 2 in the slave, timeout after 1 second.
```

Constructors

`class robot_hat.I2C(num)`: Create an I2C object associated with the given `num`. This allows you to use i2c on that device.

Methods

- `is_ready` - Check if slave 0x42 is ready.

```
I2C.is_ready(addr)
```

- `scan` - Scan for slaves on the bus, returning.

```
I2C.scan()
```

- `send` - Send several bytes to slave with address.

```
I2C.send(send,addr,timeout)
```

- `recv` - Receive one or several bytes.

```
data = i2c.recv(recv,addr,timeout)    # receive 3 bytes
```

- `mem_write` - Write to the memory of an I2C device.

```
I2C.mem_write(data, addr, memaddr, timeout)
```

- `mem_read` - Read from the memory of an I2C device.

```
I2C.mem_read(data, addr, memaddr, timeout)
```

3.1.5 class `Music` - notes and beats

Usage

```
from robot_hat import Music, Buzzer

m = Music()         # create a music object
buzzer = Buzzer("P0")
m.tempo(120)            # set current tempo to 120 beat per minute

# play middle C, D, E, F ,G, A, B every 1 beat.
buzzer.play(m.note("Middle C"), m.beat(1))
buzzer.play(m.note("Middle D"), m.beat(1))
buzzer.play(m.note("Middle E"), m.beat(1))
buzzer.play(m.note("Middle F"), m.beat(1))
buzzer.play(m.note("Middle G"), m.beat(1))
buzzer.play(m.note("Middle A"), m.beat(1))
buzzer.play(m.note("Middle B"), m.beat(1))

song = './music/test.wav'

m.music_set_volume(80)
print('Music duration',m.sound_length(file_name))
m.sound_play(song)
```

Constructors

`class robot_hat.Music()`: Create a Music object. This allows you to then get or control music!

Methods

- `note` - Gets the frequency of the note. The input string must be the constant `NOTE`.

```
Music.note("Middle D")
Music.note("High A#")
```

- `beat` - Get milisecond from beats. Input value can be float, like `0.5` as half beat, or `0.25` as quarter beat.

```
Music.beat(0.5)
Music.beat(0.125)
```

- `tempo` - Get/set the tempo, input value is in bmp(beat per second).

```
Music.tempo()
Music.tempo(120)
```

- `play_tone_for` - Play tone. Input is note and beat, like `Music.note("Middle D")`, `Music.beat(0.5)`.

```
Music.play_tone_for(Music.note("Middle D"), Music.beat(0.5))
```

- `sound_play` - Play music files.

```
sound_play(file_name)
```

- `background_music` - Background music playback (file name, number of loops, starting position of music file, volume).

```
background_music(file_name, loops=-1, start=0.0, volume=50)
```

- `music_set_volume` - Set volume

```
music_set_volume(value=50)
```

- `music_stop` - stop

```
music_stop()
```

- `music_pause` - pause

```
music_pause()
```

- `music_unpause` - unpause

```
music_unpause()
```

- `sound_length` - Return the duration of the music file.

```
len = sound_length(file_name)
```

3.1.6 class `TTS` - text to speech

Usage

```
from robot_hat import *

tts = TTS()                        # create a TTS object
tts.say('hello')                   # write word

# tts.write('hi')                   # write word
tts.lang('en-GB')                   # change language

tts.supported_lang()               # return language
```

Constructors

`class robot_hat.TTS(engine)`: Create a TTS object, `engine` could be `"espeak"` as Espeak, `"gtts"` as Google TTS and `polly` as AWS Polly.

Methods

- say - Write word on TTS.

```
TTS.say(words)
```

- lang - Change on TTS.

```
TTS.lang(language)
```

- supported_lang - Inquire all supported language.

```
TTS.supported_lang()
```

- parameter adjustment

```
# amp: amplitude, volume
# speed: speaking speed
# gap: gap
# pitch: pitch
def espeak_params(self, amp=None, speed=None, gap=None, pitch=None)
```

3.1.7 class Motor - motor control

Usage

```
from robot_hat import Motor

motor = Motor()                    # Create a motor object
motor = motor.wheel(100)           # Set the motor speed to 100
```

Constructors

class robot_hat.Motor(): Create a motor object, you can use it to control the motors.

Methods

- wheel - Control the speed and direction of the motor.

```
# The speed range is -100 to 100, and the positive and negative values represent the␣
↪direction of rotation of the motor.
motor.wheel(100)

# The second parameter, filled with 0 or 1, is used to control a specific motor.
motor.wheel(100,1)
```

3.1.8 class Servo - 3-wire pwm servo driver

Usage

```
from robot_hat import Servo, PWM

pin = PWM("P0")
ser = Servo(pin)                    # create a Servo object
val = ser.angle(60)                 # set the servo angle
```

Constructors

`class robot_hat.Servo(pin)`: Create a Servo object associated with the given pin. This allows you to set the angle values.

Methods

- `angle` - set the angle values between -90 and 90.

```
Servo.angle(90)
```

3.2 Modules

3.2.1 class `ADXL345` - accelemeter

Usage

```
from robot_hat import ADXL345

accel = ADXL345()                    # create an ADXL345 object
x_val = accel.read(accel.X)          # read an X(0) value
y_val = accel.read(1)                # read an Y(1) value
z_val = accel.read(2)                # read a Z(2) value
```

Constructors

`class robot_hat.ADXL345(address=0x53)`: Create an ADXL345 object. This allows you to then read adxl345 accelerator values.

Methods

- `read` - Read the value with the axis and return it. The unit is gravity acceleration(about 9.8m/s2).

```
ADXL345.read(axis)
```

Constants

- `X` - x axis
- `Y` - y axis
- `Z` - z axis

3.2.2 class `Buzzer` - passive buzzer

Usage

```
from robot_hat import PWM, Buzzer, Music

pwm = PWM("A0")                 # create a pwm object
buzzer = Buzzer(pwm)            # create a Buzzer object with PWM object
music = Music()                 # create music object

buzzer.play(music.note("Low C"), music.beat(1))    # play low C for 1 beat
buzzer.play(music.note("Middle C#"))               # play middle C sharp
buzzer.off()                                       # turn buzzer off
```

Constructors

`class robot_hat.Buzzer(pwm)`: Create a Buzzer object associated with the given pwm object. This will allow you to control the buzzer.

Methods

- `on` - Turn the buzzer on with a square wave.

```
Buzzer.on()
```

- `off` - Turn the buzzer off.

```
Buzzer.off()
```

- `freq` - Set the frequency of the square wave.

```
Buzzer.freq(frequency)
```

- `play` - Make the buzzer sound at a specific frequency and stop at a ms delay time.

```
Buzzer.play(freq, ms)
Buzzer.play(freq)
```

3.2.3 class `DS18X20` - ds18x20 series temperature sensor

Usage

```
from robot_hat import Pin, DS18X20

pin = Pin("D0")              # create pin object
ds = DS18X20(pin)            # create a DS18X20 object

ds.read(ds.C)                # read temperature in celsius(1)
ds.read(0)                   # read temperature in Fahrenheit(0)
```

Constructors

`class robot_hat.DS18X20(pin)`: Create a DS18X20 object associated with the given pin object. This allows you to read temperature from DS18X20.

Methods

- `DS18X20.read` - Read temperature with the giving unit. It returns a `float` value if only one ds18x20 is connected to the pin, or it will return a list of all sensor values.

```
DS18X20.read(unit)
```

- `DS18X20.scan` - Scan the connected DS18X20 and return the roms address list.

```
DS18X20.scan()
```

- `DS18X20.read_temp` - Read temperature(s) with the giving rom(s).

```
DS18X20.read_temp(rom)
```

Constants

- `DS18X20.C` - Celsius

- `DS18X20.F` - Fahrenheit

3.2.4 class `Joystick` - 3-axis joystick

Usage

```
from robot_hat import Joystick, ADC, Pin

x_pin = ADC("A0")
y_pin = ADC("A1")
btn_pin = Pin("D1")

joystick = Joystick(x_pin, y_pin, btn_pin)          # create a Joystick object
val = joystick.read(0)                              # read an axis value
status = joystick.read_status()                     # read the status of joystick
```

Constructors

`class robot_hat.Joystick(pin)`: Create a Joystick object associated with the given pin. This allows you to read values from Joystick.

Methods

- `read` - Read the value on the given pins and return them.

```
Joystick.read(Xpin, Ypin, Btpin)
```

- `read_status` - read the status of joystick.

```
Joystick.read_status()
```

3.2.5 class `RGB_LED` - rgb LED

Usage

```
from robot_hat import PWM, RGB_LED

r = PWM("P0")
g = PWM("P1")
b = PWM("P2")

rgb = RGB_LED(r, g, b)                        # Create a RGB_LED object
val = rgb.write('#FFFFFF')                    # Write in the color in hexadecimal.
```

Constructors

`class robot_hat.RGB_LED(Rpin, Gpin, Bpin)`: Create a `RGB_LED` object associated with the given pin. This allows you set the color of RGB LED. Input `Rpin`, `Gpin`, `Bpin` must be `PWM` object from `robot_hat.PWM`.

Methods

- `write` - Writing a specific color to the RGB LED, the color value is represented by hexadecimal for red, green and blue (RGB). Each color has a minimum value of 0 (00 in hexadecimal) and a maximum value of 255 (FF in hexadecimal). Hexadecimal values are written with a # sign followed by three or six hexadecimal characters.

```
RGB_LED.write(color)
```

3.2.6 class `Ultrasonic` - ultrasonic ranging sensor

Usage

```
from robot_hat import Ultrasonic, Pin

trig = Pin("D0")
echo = Pin("D1")

ultrasonic = Ultrasonic(trig, echo)             # create an Ultrasonic object
val = ultrasonic.read()                         # read an analog value
```

Constructors

`class robot_hat.Ultrasonic(trig, echo)`: Create a Ultrasonic object associated with the given pin. This allows you to then read distance value.

Methods

- `read` - Read distance values.

  ```
  Ultrasonic.read(trig, echo)
  ```

3.2.7 class `Sound` - sound sensor

Usage

```
from robot_hat import Sound, ADC

pin = ADC("A0")
sound = Sound(pin)                          # create a Sound object from a pin
val = sound.read_raw()                      # read an analog value

average_val = sound.read_raw(time = 100)    # read an average analog value
```

Constructors

`class robot_hat.Sound(pin)`: Create a Sound object associated with the given pin. This allows you to read analog values on that pin.

Methods

- `read_raw` - Read the value on the analog pin and return it. The returned value will be between 0 and 4095.

```
Sound.read_raw()
```