Growthings Documentation

Release 0.1

Paul Xu

Apr 23, 2018

Contents:

Welcome to the User Guide of Growthings, an affordable, easily programmable table-top smart greenhouse for everyone!

If you are here for the first time and looking to get started on building your smart greenhouse, please go to the *Installation* page for detailed guide on how to set up your hardware and software.

If you are looking for information on how start programming the electronics, please go to the *Quickstart Guide* page.

If you are trying to find out how to program a certain device, please click on the corresponding image below.

CHAPTER 1

Introduction

CHAPTER 2

Installation

2.1 Setting up WioLink boards

2.1.1 Downloading MicroPython Firmware

2.1.2 Downloading EsPy IDE

2.1.3 Flashing MicroPython Firmware

2.1.4 Testing Installation

2.2 Setting up the Raspberry Pi Zero W

2.2.1 PiCamera Installation

2.2.2 Oled Screen Installation

2.2.3 Downloading the Preconfigured Image

2.2.4 Writing the Image to an MicroSD card

2.2.5 Setting up Wifi

2.2.6 Setting up Node-RED App

CHAPTER 3

Quickstart Guide

CHAPTER 4

API Reference

4.1 `sensors` - Working with Sensors

The `sensors` module provides a collection of classes to interact with sensors, such as temperature, light, and soil moisture sensor.

4.1.1 Temperature/Humidity Sensor

class sensors.**TemperatureSensor**(*port[=3]*)

Allows reading temperature and humidity information from the Grove Temperature Sensor based on DHT11. We recommend that this sensor be connected to Port 3, which is the default for this class. The `port` parameter defines which port the sensor is connected to.

TemperatureSensor.**get_temperature**(*celsius[=False]*)

Returns the temperature reading in Fahrenheit. If `celsius` is set to `True` then the celsius temperature will be returned.

Note: This temperature sensor offers a resolution of 1 degree Celsius. If you want more accurate readings, please try the pro version below.

TemperatureSensor.**get_humidity**()

Returns the relative humidity in percentage.

TemperatureSensor.**show_data**(*screen*)

Shows the temperature (in Fahrenheit) and relative humidity on the specified screen object.

Example

```
from sensors import TemperatureSensor()
t = TemperatureSensor() # defines a temperature sensor at default port (Port 3)
```

```
t.get_temperature() # returns the Fahrenheit value
t.get_temperature(True) # returns the Celsius value
t.get_humidity() # returns the humidity value
```

class sensors.**TemperatureSensorPro**(*port[=3]*)

Allows reading temperature and humidity information from the Grove Temperature Sensor Pro based on DHT22 (AM2302). We recommend that this sensor be connected to Port 3, which is the default for this class. The `port` parameter defines which port the sensor is connected to.

TemperatureSensorPro.**get_temperature**(*celsius[=False]*)

Returns the temperature reading in Fahrenheit. If `celsius` is set to `True` then the celsius temperature will be returned.

Note: This temperature sensor offers a resolution of .1 degree Celsius.

TemperatureSensorPro.**get_humidity**()

Returns the relative humidity in percentage.

TemperatureSensor.**show_data**(*screen*, *line*)

Shows the temperature (in Fahrenheit) and relative humidity on the specified `screen` object on the specified `line`.

```
from sensors import TemperatureSensorPro
t = TemperatureSensor(3) # defines a temperature sensor pro at default port (Port 3)

t.get_temperature() # returns the Fahrenheit value
t.get_temperature(True) # returns the Celsius value
t.get_humidity() # returns the humidity value

# shows temperature/humidity data on the oled screen
from displays import OledScreen

screen = OledScreen(6)
t.show_data(screen, 1)
```

4.1.2 Light Sensor

class sensors.**LightSensor**(*port[=6], address[=0x29]*)

Allows reading lux values from the Grove Digital Light Sensor based on the TSL2561 I2C light sensor. The `port` parameter cannot be any other number than 6, and the sensor can be connected to the board through an I2C hub. The `address` parameter assigns the I2C address of the light sensor. `0x29 (41)` is the default for the Grove sensor. The Adafruit version has a default of `0x39 (57)`.

LightSensor.**get_lux**()

Returns the light intensity reading as lux.

Hint: Useful lux values:

- Sunlight: 107,527
- Full Daylight: 10,752
- Overcast Day: 1,075
- Very Dark Day: 107

- Twilight: 10.8
- Full Moon: .108

LightSensor.**show_data**(*screen*, *line*)
Shows the light intensity reading in lux on the specified `screen` object on the specified `line`.

Example

```
# The following code reads light value every 5 seconds,
# and if it's too dark (lux < 100), prints a warning message

from sensors import LightSensor
import time

l = LightSensor()

while True:
    lux = l.get_lux()

    if lux < 100:
        print("Too Dark!")

    time.sleep(20) # wait for 20 seconds
```

4.1.3 Moisture Sensor

class MoistureSensor(*port[=4]*)
Allows reading moisture values from the Grove Moisture Sensor. The `port` parameter cannot be any other number than 4, because the sensor is analog.

get_moisture(*port[=4]*)
Returns the raw moisture reading.

> **Warning:** Because the moisture sensor is analog, the values of the sensor readings might vary from case to case. It is a good idea to calibrate the sensor by experimenting on the soil.

LightSensor.**show_data**(*screen*, *line*)
Shows the raw moisture reading on the specified `screen` object on the specified `line`.

4.2 `actuators` - Working with Actuators

The `actuators` module provides a collection of classes to interact with actuators, such as servos, relays, and the grow light LED strip.

4.2.1 Servo

class actuators.**Servo**(*port[=2], position[=0]*)
Allows control of a Grove Servo. Default `port` for the servo is `2`. The `position` parameter sets the initial position of the servo.

`Servo.`**`set_position`**(*degree*)
Sets the `degree` position of the servo (between `0` and `180`, which is half a circle). If `degree` is greater than 180, the servo will be set at the `180` degree position. Likewise, if `degree` is less than `0`, the servo will rotate to the 0 degree position.

`Servo.`**`get_position`**()
Returns the current position of the servo in degrees.

Example:

```
from actuators import Servo

s = Servo(1, init_degree = 180) # defines a servo connected to port 1 with initial␣
↪position at 180 degrees.
s.set_position(90) # rotate the servo by 90 degrees.
```

4.2.2 Relay

class `actuators.`**`Relay`**(*port[=1]*)
Allows control of a Grove Relay. The relay by default is normally open (NO) triggered by a `high` signal.

`Relay.`**`on`**()
Activate the relay, close the circuit, and turn on whatever appliance that's connected to the relay.

`Relay.`**`off`**()
Deactivate the relay, open the circuit, and turn off whatever appliance that's connected to the relay.

`Relay.`**`is_on`**()
Returns `True` if the relay is on, or `False` if the relay is off.

4.2.3 Grow Light Strip

class **`GrowLight`**(*port[=1], n[=60]*)
Allows control of a 5V LED strip based on the WS2812b (NeoPixel). `n` specifies the number of LEDs on the strip. Default is `60`.

`on`()
Turns on the LED strip as a plant growth light that emits red and blue light.

`off`()
Turns off the LED strip.

`is_on`()
Returns `True` if the grow light is on, or `False` if it is off.

Hint: This class is a subclass of MicroPython's `neopixel.NeoPixel` class, so it can be programmed the same way as the Neo Pixel. See this page for more details and examples.

---

class **`Led`**(*port[=1]*)
Allows control of a Grove LED socket. It is possible to switch the LEDs on the socket. The LEDs have polarities. The longer leg is positive.

on (*fade[=False], duration[=None]*)
Turns on the LED. If the `fade` parameter is set to `True`, then the led will turn on gradually in the number of seconds set to the `duration` parameter.

off (*fade[=False], duration[=None]*)
Turns off the LED. If the `fade` parameter is set to `True`, then the led will turn off gradually in the number of seconds set to the `duration` parameter.

is_on()
Returns `True` if the grow light is on, or `False` if it is off.

class Button (*port[=2], pullup[=True]*)
Allows control of a Grove Button. There are two ways to use a button. First, you can access `Button.is_pressed` property or `Button.is_pressed()` method to determine if the button is pressed. Alternatively, you can also set a callback function with `Button.on_release()` method use the interrupt mechanism. Please see example of how to use the callback.

is_pressed()
Returns `True` if the button is pressed, or `False` if it is not.

on_press (*callback*)
Executes the `callback` function provided to the method when the button is pressed.

on_release (*callback*)
Executes the `callback` function provided to the method when the button is released.

Example

Controlling the LED with the Button

```
from actuators import Led, Button
led = Led(1) # Specifies an LED at Port 1
button = Button(2) # Specifies a button at Port 2

## Turns on the LED when the button is pressed

while True:

    if button.is_pressed():
        led.on()
    else:
        led.off()
```

Turns the LED on/off with a callback function

```
from actuators import Led, Button
led = Led(1) # Specifies an LED at Port 1
button = Button(2) # Specifies a button at Port 2

## Define a callback function

def turn_on_led():
    global led # Need this line to refer to the led object outside the function.

    if led.is_on():
        led.off()
    else:
        led.on()
```

```
## Set the callback function to Button.on_release method.

button.on_release(callback=turn_on_led) # Note that no () are needed.
```

4.3 `displays` - Working with Displays

The `displays` module provides a collection of classes to interact with screens. As of now only the Grove 0.96" OLED screen is supported.

4.3.1 OLED Screen

class `screens.`**`OledScreen`**(*port[=6], width[=128], height[=64], address[=0x3c]*)

Allows control of a Grove 0.96' OLED screen. It is an I2C device so it will only work on `Port 6` or an I2C hub that is connected to `Port 6`. You can specify the `width` and `height` of the screen (default 128x64). You can also specify the I2C address of the screen. If you are not sure, just use default values and everything will be taken care of.

The screen has an internal representation of the content it displays called *framebuffer*. Usually you will need to change *framebuffer* first, and call the `show()` method to change what is actually displayed on the screen. Some of the following methods only changes the buffer, while some directly modifies what is displayed on the screen. Please choose these methods accordingly.

`OledScreen.`**`clear`**`()`

Clears the *framebuffer*. Does **NOT** change what is displayed on the screen. Please call the `show()` method subsequently to see the result.

`OledScreen.`**`clear_line`**(*line*)

Clears the specified `line` in the framebuffer. Does **NOT** change what is displayed on the screen. Please call the `show()` method subsequently to see the result.

`OledScreen.`**`write_line`**(*line*, *message*)

Writes the `message` to the specified `line`. Does **NOT** change what is displayed on the screen. Please call the `show()` method subsequently to see the result.

`OledScreen.`**`show_line`**(*line*, *message*)

Writes and shows the `message` to the specified `line`. This method directly changes what is displayed on the screen.

`OledScreen.`**`show_sensor_data`**(*sensor*, *line*)

Writes the data on the specified `sensor` to the specified `line`.

Example:

```
from displays import OledScreen

screen = OledScreen(6)
screen.show_line(1, "Hello World!")
```

4.4 IoT modules

4.4.1 Raspberry Pi

4.4.2 BC Server

Index