Gigglebot Micropython Library Documentation

Dexter Industries

Jun 28, 2019

Contents:

This library is meant to be used with the Mu editor , a Micro:bit, and GiggleBot.

For step by step installation instructions and how to get started, please see the main GiggleBot website. You can also follow the instructions in this documentation by going over *here*.

For an introduction to MicroPython on the Micro:Bit, see the official tutorial.

CHAPTER 1

Getting Started

Mu is a beginners-oriented python code editor that has been built with lots of feedback from teachers and learners throughout the time. A supported board on the Mu editor is the BBC microbit board for which we have created a tailored firmware *(based on the MicroPython project)* that includes the modules documented here. The BBC microbit in turn, gets plugged into the GiggleBot.

The following sections will direct you to setting up the GiggleBot firmware on the BBC microbit and thus be able to program it with the Mu editor. At the end of these short instructions, you will be able to run the following snippet of code that lights up the eye LEDs on the GiggleBot.

```
from microbit import *
from gigglebot import *

# initialize the neopixels on the GiggleBot
init()

while True:
    set_eyes() # set GiggleBot eyes to blue (by default)
    sleep(250) # wait 250 msec
    pixels_off() # turn off the GiggleBot eyes
    sleep(250) # wait 250 msec
```

Note: There are 2 MCUs (short for *Microcontroller Unit*) sitting on the BBC micro:bit:

- One MCU has the role of flashing the HEX file that gets copied over to the other MCU.

- And the second MCU houses the actual scripts and MicroPython runtime (technically known as firmware) that you interact with (in the real world or at the terminal).

In the micropython ecosystem, people refer to it as *firmware*. In the micro:bit ecosystem, it is referred to as *micropython runtime* to differentiate it from the microbit firmware, also known as the DAL firmware. Here are some examples:

- Exhibit A from BBC micro:bit MicroPython docs.
- Exhibit B from BBC micro:bit MicroPython docs.

So in our documentation, **when we talk about the firmware, we are referring to the MicroPython firmware. And when we want to refer to the other MCU's firmware, we just call it the DAPLink firmware**. Simple, right?

1.1 Downloading the Firmware

The 1st option in downloading the firmware is to visit the direct link placed in here and download the afferent file. Make sure you download it as a `hex` file.

The latest version (which is **v0.4.0**) of the firmware can also be directly downloaded from here.

The 2nd option is head over to this project's release page and download the appropriate firmware for the appropriate documentation. As of this moment, this is the latest release.

1.2 Downloading the Modules

Apart from being able to download the firmware, the `.py` and `.mpy` modules can also be downloaded. And just like with the firmware, you can either download the modules from the release page or by accessing the following direct links for version **v0.4.0**: gigglebot.py, distance_sensor.py, thp.py, lightcolor.py, gb_diag.py.

You can also check this artifact explorer here.

Important: Downloading the modules when the *GiggleBot MicroPython Firmware* is used is redundant. Use the modules (the `py` modules) when you are using the basic version of micropython. And in that case, not all `py` modules will work - for that check this *section* and see how you can pair the modules to fit on a basic firmware.

1.3 Flashing the Firmware

Flashing the firmware is a breeze. Connect the BBC microbit to your laptop, wait until the *MICROBIT* filesystem appears and then copy-paste the GiggleBot firmware you just downloaded to the microbit.

After flashing the firmware, you will be able to import all modules listed in this documentation.

1.4 Custom Firmware in the Mu Editor

The Mu Editor comes with a default firmware for the microbit that can be overridden with the GiggleBot firmware instead. All that has to be done is to press on the *gear wheel* on the right hand side of the editor, then go to *BBC micro:bit Settings* and lastly, copy paste the path to the custom firmware (or runtime as the Mu editor likes to call).

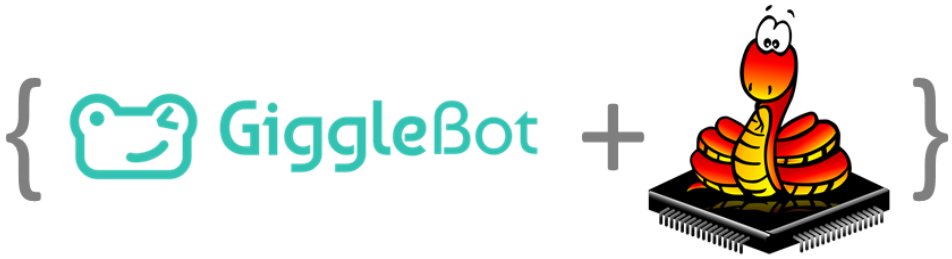
1.5 Upgrading DAPLink Firmware

There may be cases when BBC microbit fails to flash the firmware when the binary is dragged and dropped. This is generally caused by an old version of the DAPLink firmware. This DAPLink firmware provides the USB interface that allows you to drag-and-drop binaries onto the target microcontroller (the microbit).

This DAPLink firmware can be easily upgraded. Just go over this short tutorial to upgrade it.

CHAPTER 2

About the Firmware



2.1 Modules

The GiggleBot MicroPython firmware comes with the following modules:

- *gigglebot* - contains an API to interface with the actual GiggleBot: think motors, line sensor, light sensors, LEDs, servos, etc.
- *gb_diag*- still an API to interface with the GiggleBot, but slightly more advanced and useful for debugging purposes.
- *distance_sensor* - add-on sensor library for the Distance Sensor.
- *thp* - add-on sensor library for the Temperature Humidity Pressure Sensor.
- *lightcolor* - add-on sensor library for the Light and Color Sensor.

Each new release comes with the following artifacts:

- A customized BBC micro:bit firmware that includes all the above modules in it - and we are calling it the GiggleBot firmware.
- The minified modules. These modules were minified to take less space on the BBC micro:bit.
- The pre-compiled modules (bytecodes) that no longer need to be compiled when they get imported on the BBC micro:bit.

- And the modules in their natural state.

2.2 Firmware

This customized firmware that includes the modules listed in this documentation also includes a handful of features meant to make use of the available RAM on the BBC micro:bit in a more efficient manner. The downside to this, is that we had to **remove the built-in support for the micro:bit's compass and the SPI libraries**, which are least likely to be needed.

The default micropython firmware that comes with the Mu Editor fills the needs of a single microbit but does not offer enough power to easily drive extra hardware like the GiggleBot. So we have come up with a custom firmware that allows the import of precompiled python modules (to byte-code aka `.mpy` files). These `.mpy` modules can be treated like regular `.py` modules and copied in exactly the same manner over your microbit's filesystem, but they cannot be opened and edited in any way; it's not like it's necessary anyway. This has the advantage of skipping the precompiling phase on the microbit that could leave it without RAM resources during or after this process.

Still, precompiling modules still uses RAM to load them when they get imported, so the best bet is to just incorporate them into the firmware - a process called *freezing*. And this is what we have done with our firmware. Freezing the modules leads to a small usage of RAM, which is critical to the microbit, given its limited resources.

Here's a table showing what kind of modules the **GiggleBot firmware** (which is just a custom MicroPython version for the micro:bit) accepts vs the **regular firmware**:

	GiggleBot Firmware	Regular Micropython For The Microbit
regular `.py` modules	yes	yes
precompiled `.mpy` modules	yes	no
freezed gigglebot modules	yes	no

And here's what kind of modules can be used on the BBC microbit hardware:

Module	Load `.py` module	Load `.mpy` module	As freezed module
gigglebot	yes	yes	yes
gb_diag	yes	yes	yes
distance_sensor	no	no	yes
thp	no	yes	yes
lightcolor	no	yes	yes

Note: Be advised that loading `.py` modules directly to the microbit uses most of the RAM that's available to the board, so not much is left to the user to code. That's why it's better to go with `.mpy` or freezed modules (*meaning our custom firmware*) and only go with the regular `.py` when burning the custom GiggleBot MicroPython firmware to the microbit is not possible.

2.3 Using It

To download, flash and play with the firmware, follow the *Getting Started* chapter.

CHAPTER 3

API

Time for reading the GiggleBot modules' API.

3.1 GiggleBot - Regular Module

The *gigglebot* module should be the go-to module when playing around with the GiggleBot. That's why this is the first module that gets documented in this chapter.

The module, should you want to use it in another firmware instead of using the *GiggleBot MicroPython Firmware* (that already comes with all the modules in it), can be downloaded from here (version **v0.4.0**).

gigglebot.**LEFT = 0**
: Left, either a left turn, or the left motor.

gigglebot.**RIGHT = 1**
: Right, either a right turn, or the right motor.

gigglebot.**BOTH = 2**
: Indicates both motors.

gigglebot.**FORWARD = 1**
: Forward direction if the motor power is positive. Please note that if the motor power is negative, this forward would become a backward.

gigglebot.**BACKWARD = -1**
: Backward direction if the motor power is positive. Please note that if the motor power is negative, this backward would become a forward.

gigglebot.**motor_power_left = 50**
: Power to the left motor. From **-100** to **100**. Negative numbers will end up reversing the movement. Default value is **50%**.

gigglebot.**motor_power_right = 50**
Power to the right motor. From **-100 to 100**. Negative numbers will end up reversing the movement. Default value is **50%**.

gigglebot.**neopixelstrip = None**
Neopixel variable to control all neopixels. There are 9 neopixels on the Gigglebot. Pixel **0** is the right eye, pixel **1** is the left eye. Pixel **2** is the first of the rainbow pixel, on the right side. In order to control the neopixels, you must call *init()* beforehand.

gigglebot.**LINE_SENSOR = 5**
I2C command to read the line sensors.

gigglebot.**LIGHT_SENSOR = 6**
I2C command to read the light sensors.

gigglebot.**init**()
Loads up the neopixel library and sets up the neopixelstrip variable for later use.

Important: It is possible to use the Gigglebot without calling this method but the leds will not work. Should you need more RAM space, you can choose to ignore the neopixels.

Returns The neopixel strip.

gigglebot.**set_smile**(*R=25*, *G=0*, *B=0*)
Controls the color of the smile neopixels, all together.

Parameters

- **R = 25** (*int*) – Red component of the color, from **0** to **255**.
- **G = 0** (*int*) – Green component of the color, from **0** to **255**.
- **B = 0** (*int*) – Blue component of the color, from **0** to **255**.

gigglebot.**set_eyes**(*which=2*, *R=0*, *G=0*, *B=10*)
Controls the color of the two eyes, each one individually or both together.

Parameters

- **which = BOTH** (*int*) – either *LEFT* (0), *RIGHT* (1), or *BOTH* (2).
- **R = 0** (*int*) – Red component of the color, from **0** to **255**.
- **G = 0** (*int*) – Green component of the color, from **0** to **255**.
- **B = 10** (*int*) – Blue component of the color, from **0** to **255**.

gigglebot.**set_eye_color_on_start**()
Sets the eye color to blue if the batteries are good, to red if the batteries are running low.

This is called by the *init()*, usually at the start of the program. You are free to call this method whenever you want if you need to keep a closer watch on the voltage level.

gigglebot.**pixels_off**()
Turns all neopixels off, both eyes and smile.

gigglebot.**drive**(*dir=1*, *milliseconds=-1*)
This results in the Gigglebot driving *FORWARD* or *BACKWARD*.

The following snippet of code will see the gigglebot drive forward for a second:

```
from gigglebot import *
drive(FORWARD, 1000)
```

And this snippet of code will do the same thing:

```
import gigglebot
gigglebot.drive(gigglebot.FORWARD, 1000)
```

Parameters

- **dir = FORWARD** (*int*) – Possible values are *FORWARD* (1) or *BACKWARD* (-1). Please note there are no tests done on this value. One could theoretically use 2 to double the speed.
- **milliseconds = -1** (*int*) – If this parameter is omitted, or a negative value is supplied, the robot will keep on going until told to do something else, like turning or stopping. If a positive value is supplied, the robot will drive for that quantity of milliseconds.

gigglebot.**turn**(*dir=0*, *milliseconds=-1*)

Will get the gigglebot to turn left or right by temporarily removing power to one wheel.

Parameters

- **dir=LEFT** (*int*) – Either *LEFT* (0) or *RIGHT* (1) to determine the direction of the turn.
- **milliseconds=-1** (*int*) – If this parameter is omitted, or a negative value is supplied, the robot will keep on going until told to do something else, like turning or stopping. If a positive value is supplied, the robot will drive for that quantity of milliseconds.

gigglebot.**stop**()

Stops the GiggleBot right away.

gigglebot.**set_speed**(*power_left*, *power_right*)

Assigns left and right motor powers. If both are the same speed, the GiggleBot will go mostly straight.

Note: It is possible that the GiggleBot does not go straight by default. If so, you need to adjust the speed of each motor to correct the course of the robot.

---

gigglebot.**set_servo**(*which=0*, *degrees=90*)

Parameters

- **which** (*int*) – Which servo to control: *LEFT* (0), *RIGHT* (1), or *BOTH* (2).
- **degrees** (*int*) – Position of the servo, from 0 to 180.

Note: Moving the servo is not instantaneous. It is possible that you will need to give it time to reach its final position.

The following is an example that will get the servo moving from 0 to 180 degrees every second.

```
from microbit import *
from gigglebot import *

while True:
    set_servo(BOTH, 0)
    sleep(1000) # sleeps for 1000 milliseconds
    set_servo(BOTH, 180)
    sleep(1000) # sleeps for 1000 milliseconds
```

gigglebot.**servo_off**(*which*)
Removes power from the servo.

Parameters **which** (*int*) – Determines which servo, *LEFT* (0), *RIGHT* (1), *BOTH* (2).

gigglebot.**read_sensor**(*which_sensor*, *which_side*)
Reads the GiggleBot onboard sensors, light or line sensors.

Parameters

- **which_sensor** (*int*) – Reads the light sensors *LIGHT_SENSOR* (6), or the line sensors *LINE_SENSOR* (5). Values are from **0** to **1023**.
- **which_side** (*int*) – Reads *LEFT* (0), *RIGHT* (1), or *BOTH* (2) sensors. When reading both sensors, an array will be returned.

Returns Either an integer or an array of integers (right, then left).

You can read the sensors this way:

```
right, left = read_sensor(LIGHT_SENSOR, BOTH)
```

gigglebot.**read_distance_sensor**()
Read the detected range by the distance sensor. Uses the *read_range_single()* method.

Returns The distance to the object as measured in millimeters. Range is up to 2.3 meters.

Return type int

Raises

- **OSError** – When there's trouble reaching the sensor.
- **ImportError** – If this module is run from a firmware that doesn't have the *distance_sensor* module.

gigglebot.**read_thp_sensor**()
Read the temperature, the atmospheric pressure, humidity and dewpoint temperature. Uses the *TempHumPress* class to do that.

Returns In this specific order: **temp** in *Celsius*, **temp** in *Fahrenheit*, **pressure** in *Pascals* unit, **humidity** as percentage, **dewpoint** in *Celsius*, **dewpoint** in *Fahrenheit*.

Return type 6-element float tuple

Raises

- **OSError** – When there's trouble reaching the sensor.
- **ImportError** – If this module is run from a firmware that doesn't have the *thp* module.

gigglebot.**read_light_color_sensor**()
Detect the color with the Light and Color sensor. Uses the *get_color()* method.

Returns

Return type tuple(string, tuple(int,int,int))

Raises

- **OSError** – When there's trouble reaching the sensor.
- **ImportError** – If this module is run from a firmware that doesn't have the *lightcolor* module.

gigglebot.**volt**()
Returns the voltage level of the batteries.

Returns Voltage level of the batteries.

3.2 Distance Sensor

As you can see here, the Distance Sensor is the same sensor used on the GoPiGo3 and on any DexterIndustries board that we support and that has a Grove port with an *I2C* interface on it.

Because of this, we've spend lots of time trying to make the following API of the Distance Sensor identical to the one in the DI-Sensors documentation `di_sensors.distance_sensor.DistanceSensor`, so that the transition from either platform can be as seamless as possible.

The module, should you want to use it in another firmware instead of using the *GiggleBot MicroPython Firmware* (that already comes with all the modules in it), can be downloaded from here (version **v0.4.0**).

class distance_sensor.**DistanceSensor**(*address=42*, *timeout=500*)
Class for interfacing with the Distance Sensor.

__init__(*address=42*, *timeout=500*)
Constructor for initializing a *DistanceSensor* object.

Parameters

- **address = 0x2A** (*int*) – Address of the Distance Sensor. Default address of the device is **0x29**.
- **timeout = 500** (*int*) – Timeout value for when *read_range_continuous()* is used.

Important: When instantiating this object, keep in mind that the Distance Sensor has by-default, the same address as the *LightColorSensor*, but gets changed immediately after that, right within the constructor.

What still has to be done is to make sure the distance sensor is the first sensor to be connected and initialiazed and only after that the Light and Color Sensor can be plugged in and initialized too on the GiggleBot.

---

start_continuous(*period_ms=0*)
Start taking continuous measurements.

Once this method is called, then the *read_range_continuous()* method should be called periodically, depending on the value that was set to `period_ms` parameter.

Parameters **period_ms = 0** (*int*) – The time between measurements. Can be set to anywhere between **20 ms** and **5 secs**.

Raises **OSError** – When it cannot communicate with the device.

The advantage of this method over the simple `read_range_single()` method is that this method allows for faster reads. Therefore, this method should be used by those that want maximum performance from the sensor.

Also, the greater the value set to `period_ms`, the higher is the accuracy of the distance sensor.

read_range_continuous()

Read the detected range while the sensor is taking continuous measurements at the set rate.

Returns The detected range of the sensor as measured in millimeters. The range can go up to 2.3 meters.

Return type int

Raises **OSError** – When the distance sensor is not reachable or when the *start_continuous()* hasn't been called before. This exception gets raised also when the user is trying to poll data faster than how it was initially set with the *start_continuous()* method.

Important: If this method is called in a shorter timeframe than the period that was set through *start_continuous()*, an `OSError` exception is thrown.

There's also a timeout on this method that's set to **0.5 secs**. Having this timeout set to **0.5 secs** means that the `OSError` gets thrown when the `period_ms` parameter of the *start_continuous()* method is bigger than **500 ms**.

---

read_range_single()

Read the detected range with a single measurement. This is less precise/fast than its counterpart *read_range_continuous()*, but it's easier to use.

Returns The detected range of the sensor as measured in millimeters. The range can go up to 2.3 meters.

Return type int

Raises **OSError** – When the distance sensor is not reachable.

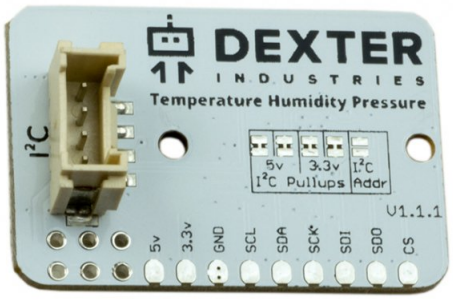
timeout_occurred()

Checks if a timeout has occurred on the *read_range_continuous()* method.

Returns Whether a timeout has occurred or not.

Return type bool

3.3 Temperature Humidity Pressure Sensor



Just like with the Distance Sensor, the Temperature Humidity Pressure Sensor is the same sensor used on the GoPiGo3 and on any other board that we support.

As a consequence, we tried making the API of the Temperature Humidity Pressure Sensor similar to the one in the DI-Sensors documentation `di_sensors.temp_hum_press.TempHumPress`, so that the transition from either platform can be as seamless as possible.

The module, should you want to use it in another firmware instead of using the *GiggleBot MicroPython Firmware* (that already comes with all the modules in it), can be downloaded from here (version **v0.4.0**).

class thp.**TempHumPress**
Class for interfacing with the Temperature Humidity Pressure Sensor.

__init__()
Constructor for initializing link with the Temperature Humidity Pressure Sensor.

Raises **OSError** – When the sensor cannot be reached.

get_temperature_celsius()
Read temperature in Celsius degrees.

Returns Temperature in Celsius degrees.

Return type float

Raises **OSError** – When the sensor cannot be reached.

get_temperature_fahrenheit()
Read temperature in Fahrenheit degrees.

Returns Temperature in Fahrenheit degrees.

Return type float

Raises **OSError** – When the sensor cannot be reached.

get_pressure()
Read the air pressure in pascals.

Returns The air pressure in pascals.

Return type float

Raises **OSError** – When the sensor cannot be reached.

Note: `get_temperature_celsius()` or `get_temperature_fahrenheit()` has to be called in order to update the temperature compensation.

---

get_humidity()
Read the relative humidity as a percentage.

Returns Percentage of the relative humidity.

Return type float

Raises **OSError** – When the sensor cannot be reached.

Note: `get_temperature_celsius()` or `get_temperature_fahrenheit()` has to be called in order to update the temperature compensation.

---

get_dewpoint_celsius()
Read dewpoint temperature in Celsius degrees.

The dewpoint represents the atmospheric temperature (varying according to pressure and humidity) below which water droplets begin to condense and dew can form.

Returns Dewpoint temperature in Celsius degrees.

Return type float

Raises **OSError** – When the sensor cannot be reached.

get_dewpoint_fahrenheit()
Read dewpoint temperature in Fahrenheit degrees.

It does the exact same thing as *get_dewpoint_celsius()*, the only difference being the used unit of measure.

Returns Dewpoint temperature in Fahrenheit degrees degrees.

Return type float

Raises **OSError** – When the sensor cannot be reached.

get_pressure_inches()
Read pressure in inches of Hg.

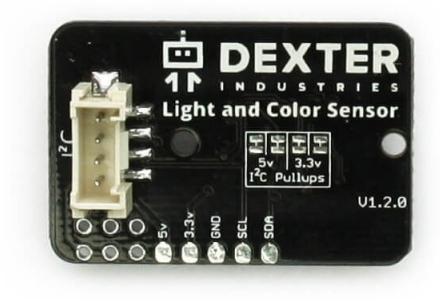
Returns The air pressure in inches of Hg.

Return type float

Raises **OSError** – When the sensor cannot be reached.

Note: `get_temperature_celsius()` or `get_temperature_fahrenheit()` has to be called in order to update the temperature compensation.

3.4 Light and Color Sensor



The module, should you want to use it in another firmware instead of using the *GiggleBot MicroPython Firmware* (that already comes with all the modules in it), can be downloaded from here (version **v0.4.0**).

class lightcolor.**LightColorSensor**(*integration_time=0.0048*, *gain=2*)

Class for interfacing with the Light and Color Sensor.

__init__(*integration_time=0.0048*, *gain=2*)

Constructor to initialize the Light and Color Sensor.

Parameters

- **integration_time = 0.0024** (*float*) – Time in seconds for each sample. **0.0024** second (2.4*ms*) increments. Clipped to the range of **0.0024** to **0.6144** *seconds*.
- **gain = GAIN_16X** (*int*) – The gain constant. Valid values are `lightcolor.GAIN_1X`, `lightcolor.GAIN_4X`, `lightcolor.GAIN_16X` and `lightcolor.GAIN_60X`.

Raises **OSError** – When the sensor cannot be reached.

set_led(*value*, *delay=True*)

Turn on/off the LED on the Light and Color Sensor.

Parameters

- **value** (*bool*) – `True` for turning it on or `False` otherwise.
- **delay = True** (*bool*) – Whether to wait for that much as it takes to take a reading before actually reading the sensor.

Raises **OSError** – When the sensor cannot be reached.

For things that emit light like monitors, the LED should be turned off, but for anything else it is recommended to have it turned on.

get_color()

Read the sensor and determine which color it resembles the most from the *known_colors* list.

This method is based off of *guess_color_hsv()* function.

Returns The detected color in string format and then a 3-element tuple describing the color in RGB format. The values of the RGB tuple are between **0** and **255**.

Return type tuple(str,(float,float,float))

Raises **OSError** – When the sensor cannot be reached.

get_raw_data(*delay=True*)
Read the **RGBA** values from the sensor.

Parameters **delay = True** (*bool*) – Whether to add delay before actually reading the sensor or not. The delay is equal to length of time it takes to take a reading.

Returns The RGBA values as a 4-tuple on a scale of 0-1.

Return type tuple(float,float,float,float)

Raises **OSError** – When the sensor cannot be reached.

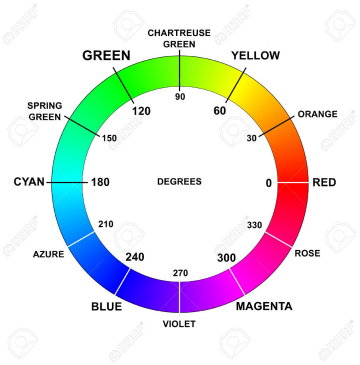
lightcolor.**known_colors = {'azure': (0, 128, 255), 'blue': (0, 0, 255), 'chartreuse-green**
12 predefined colors in RGB format that the *LightColorSensor* class and *guess_color_hsv()* function use to detect colors.

lightcolor.**known_hsv = {'azure': [(195, 65, 40), (222, 100, 100)], 'blue': [(223, 65, 40)**
The same 12 predefined colors in *known_colors*, but in HSV format. Used by *LightColorSensor* class and *guess_color_hsv()* function to detect colors.

The above colors were picked from the following color wheel - there are a total of 12 colors hand-picked. All the codenames for these colors are to be found as keys in *known_colors* and *known_hsv* dictionaries.

In the following context, the numbers encompassing the wheel represent the **Hue** of the color. You can go on and read more about **Hue** and other characteristics of HSV format here.

Note: If you want to come up with your own colour boundaries, check this thread here.



lightcolor.**translate_to_hsv**(*in_color*)
Standard algorithm to switch from one color system (**RGB**) to another (**HSV**).

Parameters **in_color** (*tuple(float,float,float)*) – The RGB tuple list that gets translated to HSV system. The values of each element of the tuple is between **0** and **1**.

Returns The translated HSV tuple list. Returned values are *H(0-360)*, *S(0-100)*, *V(0-100)*.

Return type tuple(int, int, int)

Important: For finding out the differences between **RGB** *(Red, Green, Blue)* color scheme and **HSV** *(Hue, Saturation, Value)* please check out this link.

lightcolor.**guess_color_hsv**(*in_color*)
Determines which color in_color parameter is closest to in the *known_colors* list.

This function converts the RGBA input into an HSV representation and then it determines within which color boundary described in *known_hsv* dictionary it fits in. Afterwards, it returns the candidate color in both text and RGB formats.

Parameters **in_color** (*tuple(float,float,float,float)*) – A 4-element tuple list for the *Red*, *Green*, *Blue* and *Alpha* channels. The elements are all valued between **0** and **1**.

Returns The detected color in string format and then a 3-element tuple describing the color in RGB format. The values of the RGB tuple are between **0** and **255**.

Return type tuple(str,(float,float,float))

Important: For finding out the differences between **RGB** *(Red, Green, Blue)* color scheme and **HSV** *(Hue, Saturation, Value)* please check out this link.

3.5 GiggleBot - Diagnostic Module

The module, should you want to use it in another firmware instead of using the *GiggleBot MicroPython Firmware* (that already comes with all the modules in it), can be downloaded from here (version **v0.4.0**).

gb_diag.**MOTOR_RIGHT = '\x01'**
Left motor constant.

gb_diag.**MOTOR_LEFT = '\x02'**
Right motor constant.

class gb_diag.**GiggleBot**
Class to get details and check the status of the GiggleBot hardware.

__init__()
Constructor to initialize a *GiggleBot* object.

get_details()
Gets details about the Gigglebot robot.

Returns The manufacturer name of the board, the name of the board and the firmware version of it in this specific order.

Return type tuple(string, string, int)

get_voltages()
Gets battery and rail voltages.

If the power switch is off, the value returned for the battery voltage is unequivocally close to **0**.

Returns The battery and rail voltages in this order.

Return type (float, float)

set_motor_power(*port*, *power*)
Sets the power of a single motor on the GiggleBot.

Parameters

- **port** – Either *MOTOR_LEFT* or *MOTOR_RIGHT* depending on which motor has to be controlled.
- **power** (*int*) – **-100** to **0** for reverse and from **0** to **100** for full speed ahead.

set_motor_powers(*powerLeft*, *powerRight*)
Sets the power of both motors of the GiggleBot.

Parameters

- **powerLeft** (*int*) – Anywhere between from **-100** to **100** for the left motor.
- **powerRight** (*int*) – Anywhere between from **-100** to **100** for the right motor.

get_motor_status(*port*)
Returns a report of the status of a given motor on the GiggleBot.

Parameters **port** – Either *MOTOR_LEFT* or *MOTOR_RIGHT*.

This method returns a 2-element list where the 1st element is called the `status_flag` and the 2nd one is represents the current speed set for the given motor.

The `status_flag` can have the following values:

- For **0** it means the conditions are normal (the motor can run).
- For **1** it means the battery voltage is too low or the power switch is off.

Also, if the battery voltage is **<= 3.3V**, then it's too low and the motors float. If the battery voltage is **>= 3.4V**, then it's high enought that the motors are set to run. This difference of **0.1V** is introduced in order to prevent the motors from quickly turning on and off if the battery voltage is right on the edge.

Returns A 2-element list containing the status of the motor and the current speed/power of it.

Return type list(int,int)

reset_all()
Brings the GiggleBot to a full stop, while also cutting power to the motors, so that the battery life is preserved.

CHAPTER 4

GiggleBot Tutorial

This is a tutorial on how to control your GiggleBot.

4.1 Take the GiggleBot on a stroll

This first tutorial will demonstrate how to control the robot's movements. The GiggleBot will:

1. Set its speed to 75% of its maximum power (default is 50%).
2. Go forward for a second.
3. Go backward for a second.
4. Wait for a second.
5. Turn left for half a second.
6. Turn right for half a second.

```
from microbit import *
from gigglebot import *

set_speed(75,75)
drive(FORWARD,1000)
drive(BACKWARD,1000)
sleep(1000)
turn(LEFT,500)
turn(RIGHT,500)
```

Note: There is no need to make use of the *stop()* method here because each of those timed functions will make sure the robot stops at the end of the delay.

Note: *init()* does not need to be called in this example as we are not making use of the lights on the robot.

4.2 Big Smile

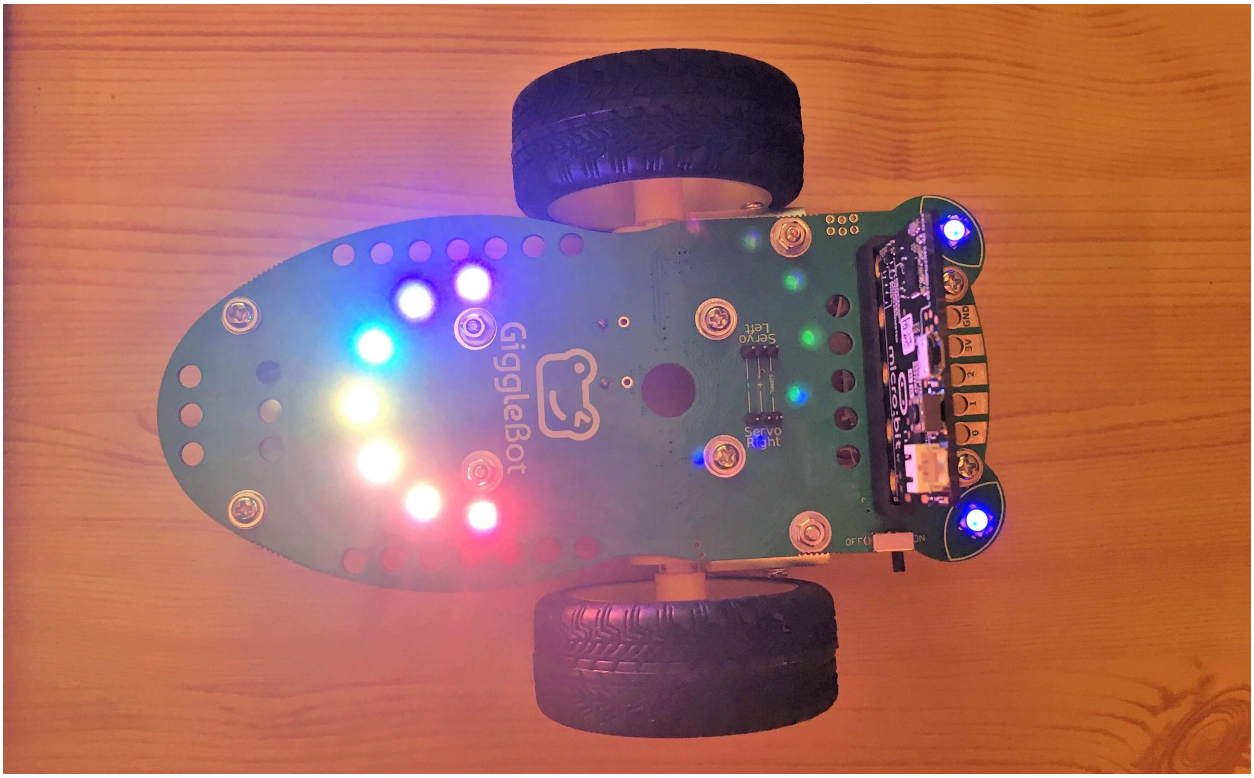
Let's use the Neopixels to turn the smile leds to a nice red, followed by green and then blue.

```
from microbit import *
from gigglebot import *

init()
while True:
    set_smile(R=100,G=0,B=0)
    sleep(500)
    set_smile(R=0,G=100,B=0)
    sleep(500)
    set_smile(R=0,G=0,B=100)
    sleep(500)
```

4.3 Rainbow Smile

You are not limited to the basic red,green,blue colors as they can be mixed. Let's create a rainbow of colors! The *init()* method returns a variable that lets you control each neopixel individually. We'll make use of this to create a rainbow.

```
from gigglebot import *

strip=init()
strip[2]=(255,0,0)
strip[2]=(248,12,18)
strip[3]=(255,68,34)
strip[4]=(255,153,51)
strip[5]=(208,195,16)
strip[6]=(34,204,170)
strip[7]=(51,17,187)
strip[8]=(68, 34, 153)
strip.show()
```



4.4 Rainbow Cycle

Here is how you can get the smile to cycle through the colours of the rainbow.

```
from microbit import *
from gigglebot import *

# first define the colors of the rainbow in an array
colors = []
colors.append((255,0,0))
colors.append((248,12,18))
colors.append((255,68,34))
colors.append((255,153,51))
colors.append((208,195,16))
colors.append((34,204,170))
colors.append((51,17,187))
colors.append((68, 34, 153))

strip=init()

# offset will let us know which colour is due to be displayed on which LED
offset = 0

# Looping forever
while True:
    offset = offset + 1

    # we might run into an issue of trying to display color 8 - which doesn't exist -␣
↪on LED 7
```

(continues on next page)

(continued from previous page)

```
    # we need to catch that case before it crashes the code.
    if offset > 7:
        offset = 0
    for i in range(7):
        if i+offset > 7:
            colind = i+offset-7
        else:
            colind = i+offset
        strip[i+2]=colors[colind]
    # display the colors
    strip.show()
    # wait a bit for the human eye to catch the colors in question
    sleep(100)
# colors were taken from http://colrd.com/palette/22198/?download=css
```

End

CHAPTER 5

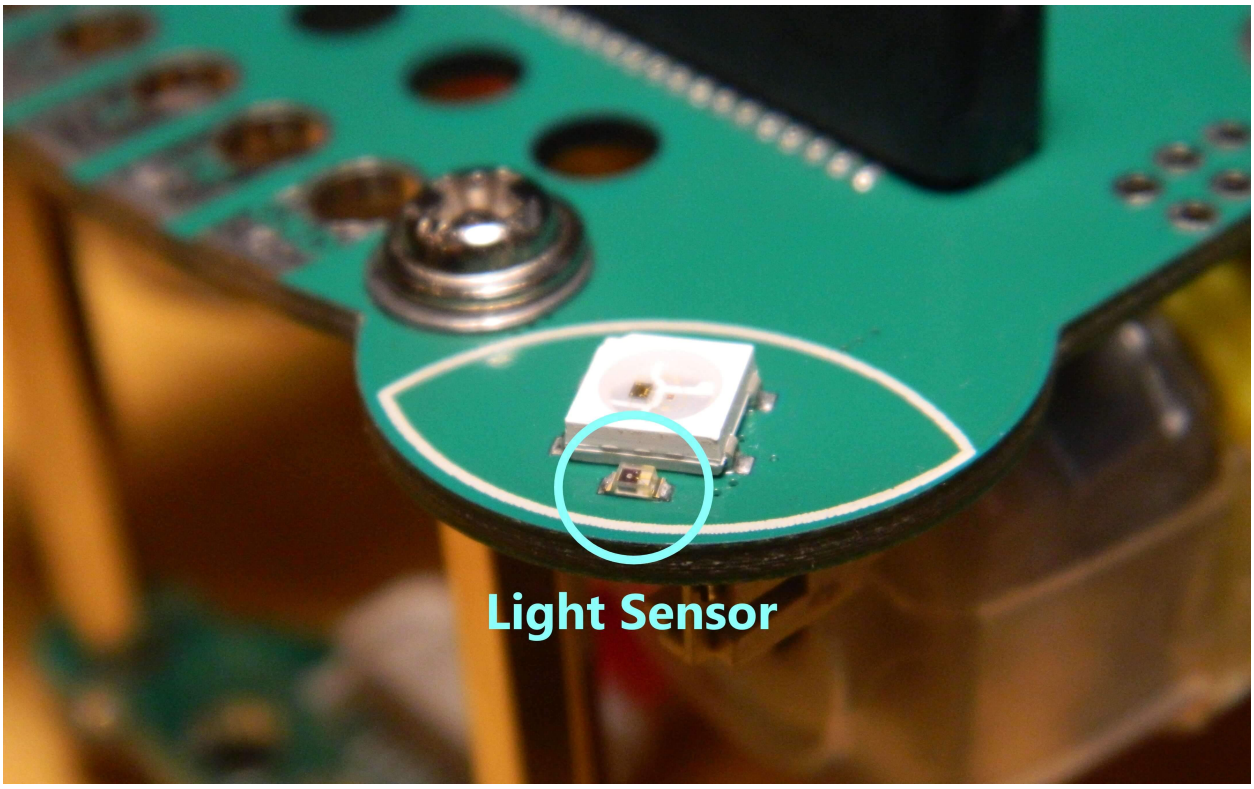
Sensors Tutorial

The GiggleBot comes with a couple of onboard sensors:

1. Two light sensors, near the front eyes LEDs.
2. Two line sensors, on the underside of the line follower.

5.1 Light Sensors

The GiggleBot comes with two light sensors right in front of the LED on each eye. They are very small and easy to miss. However, they are more versatile than the Micro:Bit's light sensor as they can be read together to detect which side is receiving more light.



The main method to query the light sensors is *read_sensor()*. The same method is used for both light sensors and line sensors.

Here's how to read both light sensors in one call:

```
right, left = read_sensor(LIGHT_SENSOR, BOTH)
```

And here's how to read just one side at a time:

```
right = read_sensor(LIGHT_SENSOR, RIGHT)
```

5.1.1 Chase the Light

This tutorial will turn the GiggleBot into a cat, following a spotlight on the floor. Many cats do that when you shine a flashlight in front of them, they will try to hunt the light spot. When running this code, you will be able to guide your GiggleBot by using a flashlight.

This is how to use this project:

1. Start GiggleBot and wait for sleepy face to appear on the Micro:Bit.
2. Press *button A* to start the Chase the Light game (heart will replace the sleepy face).
3. Chase the light as long as you want.
4. Press *button B* to stop the GiggleBot and display sleepy face again.

First, a bit of explanation on the algorithm being used here.

On starting the GiggleBot, the Micro:Bit will:

1. Assign a value to the `diff` variable (here it is using 10).

2. Display a sleepy face image.
3. Resets whatever readings from the buttons it might have had.
4. Start a forever loop.

This forever loop only waits for one thing: for the user to press *button A*, and that's when the light chasing begins.

As soon as *button A* is pressed, the Micro:Bit will display a heart as it's quite happy to be active! And then it starts a second forever loop! This second forever loop is the actual Light Chasing game. It will end when *button B* is pressed by the user.

How to Chase a Light:

1. Take reading from both light sensors.
2. Print the readings every 50 ms.
3. Compare the readings, using `diff` to allow for small variations. You will most likely get absolutely identical readings, even if the light is mostly equal. Using a differential value helps stabilize the behavior. You can adapt to your own lighting conditions by changing this value.
4. If the right sensor reads more than the left sensor plus the `diff` value, then we know it's brighter to the right. Turn right.
5. If the left sensor reads more than the right sensor plus the `diff` value, then it's brighter to the left. Turn left.
6. If there isn't that much of a difference between the two sensors, go straight.
7. If *button B* gets pressed at any time, stop the robot, change sleepy face, and get out of this internal loop. The code will fall back to the first loop, ready for another game.

```
from microbit import *
from gigglebot import *

# value for the differential between the two sensors.
# you can change this value to make it more or less sensitive.
diff = 10
# display sleepy face
display.show(Image.ASLEEP)
# the following two lines resets the 'was_pressed' info
# and discards any previous presses
button_a.was_pressed()
button_b.was_pressed()
# and also make sure the robot is stopped
stop()

# variable to control how fast
# the sensors get printed
counter = running_time()

# start first loop, waiting for user input
while True:
    # test for user input
    if button_a.was_pressed():
        # game got started! Display much love
        display.show(Image.HEART)

        # start game loop
        while True:
            # read both sensors
            right, left = read_sensor(LIGHT_SENSOR, BOTH)
```

(continues on next page)

(continued from previous page)

```
            # and print these values every 50 ms
            if running_time() - 50 > counter:
                counter = running_time()
                print((left, right))

            # test if it's brighter to the right
            if right > left+diff:
                turn(RIGHT)

            # test if it's brighter to the left
            elif left > right+diff:
                turn(LEFT)

            # both sides being equal, go straight
            else:
                drive(FORWARD)

            # oh no, the game got interrupted
            if button_b.is_pressed():
                stop()
                display.show(Image.ASLEEP)

                # this line here gets us out of the game loop
                break
```

What else can be done with the light sensors?

You could modify this code to turn the GiggleBot into a night insect? Those would avoid light instead of chasing it.

You could detect when it gets dark or bright. Imagine the GiggleBot inside your closet. When someone opens the door, the sudden light can be detected. The GiggleBot can let you know someone went through your things while you were away.

5.2 Line Sensors

In front of GiggleBot, attached to the body, there is a line follower sensor. It contains two line sensors. You can spot them from the top of the line follower by two white dots. And from the bottom, they are identified as *R* and *L* (for *right* and *left*)

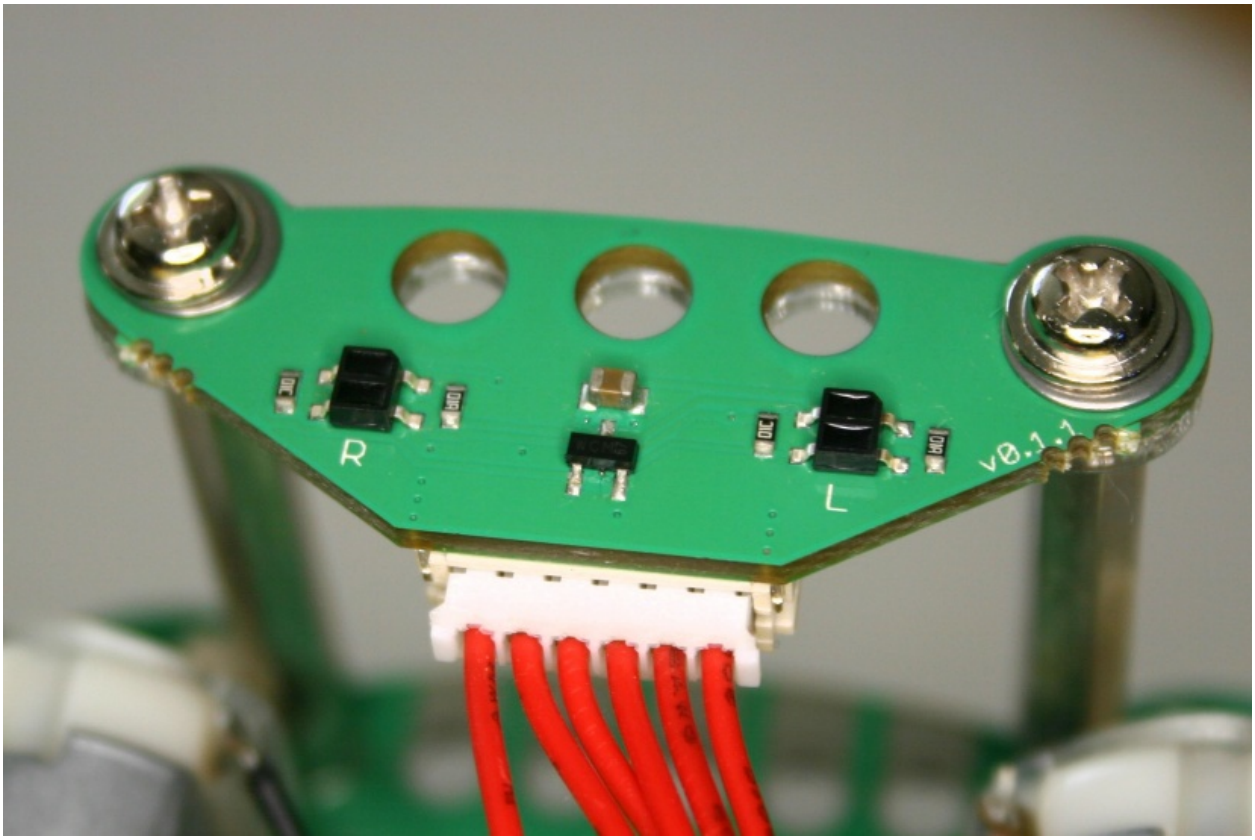


photo courtesy of Les Pounder

The easiest way of reading the sensors is as follow:

```
from gigglebot import *
right, left = read_sensor(LINE_SENSOR, BOTH)
```

The lower the number, the darker it is reading. Values can go from 0 to 1023 and depend a lot on your environment. If you want to write a line follower robot, it is best to take a few readings first, to get a good idea of what numbers will represent a black line, and what numbers represent a white line.

5.2.1 Calibrating the Line Follower

Calibrating the line follower means figuring out which numbers get returned when it's over a black line, so that you can later code an actual line follower robot.

The best approach for this is to get readings in various parts of your line, from both sensors, for both the black line and the background color.

The following code will display the values onto the microbit leds when you press *button A*, allowing you to manually position your robot around your circuit and take readings.

```
from microbit import *
from gigglebot import *

# reset all previous readings of button_a
# strictly speaking this is not necessary, it is just a safety thing
button_a.was_pressed()
```

(continues on next page)

(continued from previous page)

```
while True:
    if button_a.is_pressed():
        right, left = read_sensor(LINE_SENSOR, BOTH)
        display.scroll(left)
        display.scroll(right)
```

5.2.2 Follow the Line

Once you have gotten readings from the line sensors, you are ready to code a line follower robot.

Here we are coding for a line that is thick enough that both sensors can potentially be over the line. The robot will stop if it loses track of the line, in other words, if both sensors detect they're over the background color.

The logic will be as follow:

1. If both sensors detect a black line, forge straight ahead.
2. If neither sensor detects a black line, give up and stop.
3. If the right sensor detects a black line but not the left sensor, then steer to the right.
4. If the left sensor detects a black line but not the right sensor, then steer to the left.

We are also using the LEDs on the LED smile to indicate what is going on while we follow the line.

```
from microbit import *
from gigglebot import *

# reset all previous readings of button_a, and button_b
# strictly speaking this is not necessary, it is just a safety thing
button_a.was_pressed()
button_b.was_pressed()
display.show(Image.YES)
strip=init()
# speed needs to be set according to your line and battery level.
# do not go too fast though.
set_speed(60, 60)
# threshold is a little over the highest number you got that indicates a
# black line.
threshold = 90
while True:
    # if both buttons are pressed, run calibration code
    if button_a.is_pressed() and button_b.is_pressed():
        right, left = read_sensor(LINE_SENSOR, BOTH)
        display.scroll(left)
        display.scroll(right)
    # if button A is pressed run line following code until button B gets pressed
    # or until we're over white/background
    if button_a.is_pressed():
        while not button_b.is_pressed():
            right, left = read_sensor(LINE_SENSOR, BOTH)
            if left < threshold and right < threshold:
                # both sensors detect the line
                strip[2]=(0,255,0)
                strip[8]=(0,255,0)
                strip.show()
                drive(FORWARD)
```

(continues on next page)

(continued from previous page)

```
            elif right > threshold and left > threshold:
                # neither sensor detects the line
                stop()
                strip[2]=(255,0,0)
                strip[8]=(255,0,0)
                strip.show()
                break
            elif left > threshold and right < threshold:
               # only the right sensor detects the line
                strip[2]=(0,255,0)
                strip[8]=(0,0,0)
                strip.show()
                turn(RIGHT)
            elif right > threshold and left < threshold:
                # only the left sensor detects the line
                strip[2]=(0,0,0)
                strip[8]=(0,255,0)
                strip.show()
                turn(LEFT)
        stop()
```

photo courtesy of Lisa Rode

CHAPTER 6

Add-On Sensors Tutorial

CHAPTER 7

Indices and tables

- genindex
- search

Python Module Index

Index

Symbols

B

D

F

G

I

K

L

M

N

P

R

S

T

V