DI-Sensors Documentation

Dexter Industries

Jan 23, 2020

CONTENTS:

CHAPTER ONE

ABOUT DI-SENSORS

1.1 Who we are and what we do.

Dexter Industries is an American educational robotics company that develops robot kits that make programming accessible for everyone.

1.2 What's this documentation about.

The documentation details how to use the sensors that Dexter Industries produces and maintains - that's where the **DI** acronym comes from. All the source code for these sensors has been written in Python. Within this documentation, you will find instructions on:

- How to get started with the DI-Sensors - in general it refers to how to install them on your Raspberry Pi.
- How to get going with the examples found in our repository. In the DI_Sensors repository, you can find all the source code for our sensors and example programs.
- How to use our DI-Sensors - we offer a thorough API on the sensors we have.

CHAPTER

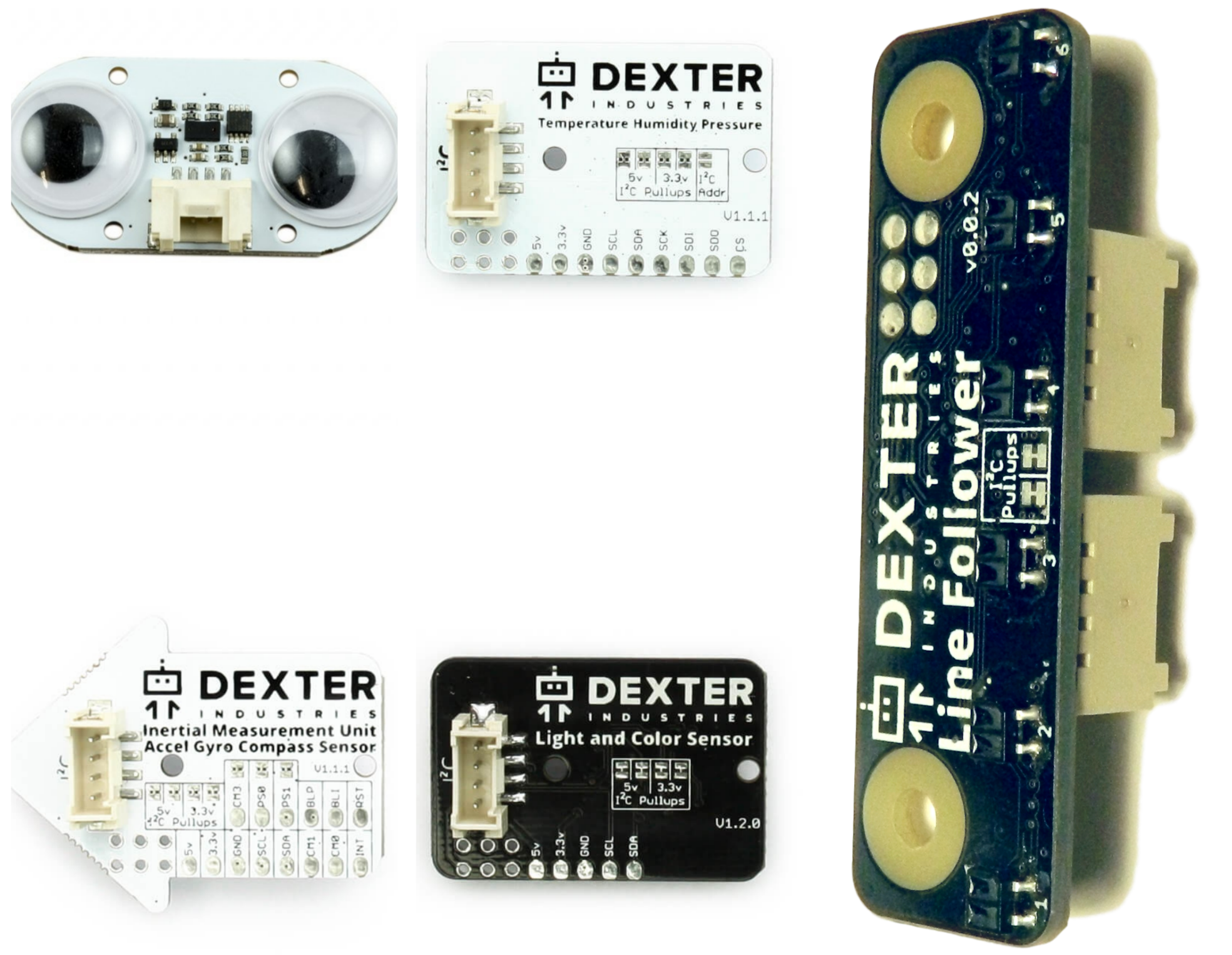
TWO

GETTING STARTED

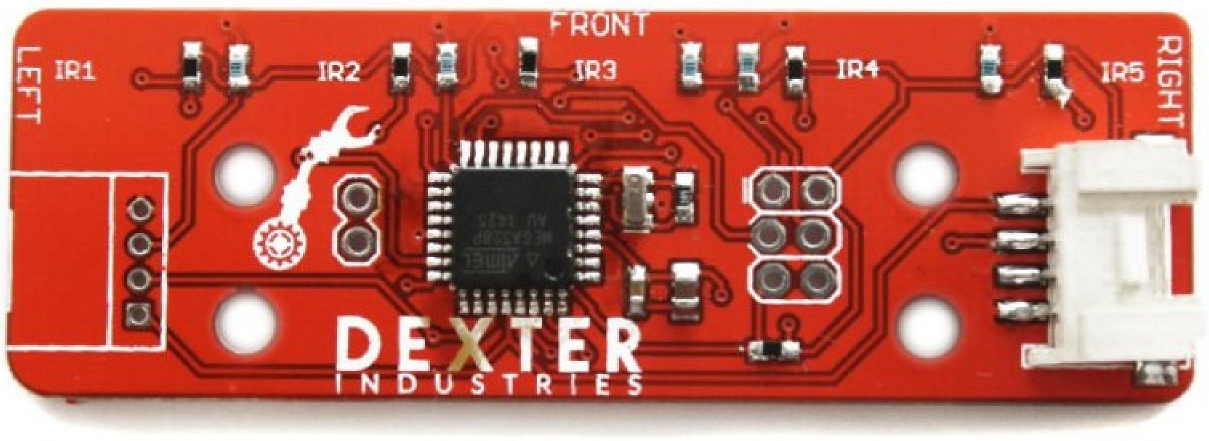
2.1 Buying our sensors

In order to run code found in this documentation, you need to head over to our online shop and get yourself one of the following sensors:

- The DI IMU Sensor.
- The DI Light and Color Sensor.
- The DI Temperature Humidity Pressure Sensor.
- The DI Distance Sensor.
- The DI Line Follower Sensor, aka the *black line follower*.



Also, apart from these sensors, the red line follower is also supported. It is the predecessor to the Line Follower Sensor. This is what it looks like:



2.2 What I can use the sensors with

All these sensors can be used along the following platforms:

- The BrickPi3.
  - Github project here.
- The GoPiGo3.
  - Github project here.
  - Documentation for the GoPiGo3 can be found here.
- The GoPiGo.
  - Github project here.
  - Predecessor of the GoPiGo3.
- The GrovePi.
  - Github project here.
  - Platform for collecting data from the environment through the use of sensors.
- The PivotPi.
  - Github project here.
  - Board to connect to a bunch of servos.

2.3 How to install the DI-Sensors

In order to install the DI-Sensors package you need to open up a terminal on your Raspberry Pi and type in the following command:

```
curl -kL dexterindustries.com/update_sensors | bash
```

Enter the command and follow the instructions given, if provided. This command can also be used for updating the package with the latest changes.

To find more about our source code, please visit the DI_Sensors repository on GitHub.

CHAPTER

THREE

EXAMPLES

This chapter revolves around the following python classes:

- *di_sensors.inertial_measurement_unit.InertialMeasurementUnit*
- *di_sensors.easy_light_color_sensor.EasyLightColorSensor*
- *di_sensors.easy_temp_hum_press.EasyTHPSensor*
- *di_sensors.distance_sensor.DistanceSensor*
- *di_sensors.easy_distance_sensor.EasyDistanceSensor*

Please make sure you have followed all the instructions found in *Getting Started* before jumping into these example programs. In all these examples, you will be required to use one of the 4 documented sensors and optionally, a GoPiGo3.

3.1 Using the Distance Sensor

3.1.1 Basic Example

Before going to the more advanced example program of using the Distance Sensor, we're going to give an example of the easiest way to read from the sensor.

The following code snippet reads values off of the Distance Sensor and prints them iteratively in the console. As you'll see, this is far easier than the following examples, which are more complex to use, but have a more granular control over the device.

In this example program, connect the Distance Sensor to an I2C port on whichever platform (GoPiGo3, GrovePi or BrickPi3) and then run the following script.

```
# import the modules
from di_sensors.easy_distance_sensor import EasyDistanceSensor
from time import sleep

# instantiate the distance object
my_sensor = EasyDistanceSensor()

# and read the sensor iteratively
while True:
  read_distance = my_sensor.read()
  print("distance from object: {} cm".format(read_distance))

  sleep(0.1)
```

The source file for this example program can be found here on github

3.1.2 Continuous-mode

Again, just like in the previous example program, connect the Distance Sensor to an I2C port on whichever platform before running the following script.

The advantage of this script over the ones in the following and previous sections is that the time taken for reading the distance can be fine-tuned by the user - for instance, it can be made to run as fast as possible (to see how fast it can read see the API of `DistanceSensor`) or it can be made to go very slow. Each fine-tune has its benefits and disadvantages, so the user has to experiment with the sensor and determine what setting suits him best.

```
import time
from di_sensors.distance_sensor import DistanceSensor

print("Example program for reading a Dexter Industries Distance Sensor on an I2C port.
↪")

# establish communication with the DistanceSensor
ds = DistanceSensor()

# set the sensor in fast-polling-mode
ds.start_continuous()

while True:
    # read the distance in millimeters
    read_distance = ds.read_range_continuous()
    print("distance from object: {} mm".format(read_distance))
```

The source code for this example program can be found here on github.

3.1.3 Single-mode

In this third example, we have the same physical arrangement as in the second one, the only difference being in how we communicate with the sensor. This time, we take single-shot readings, which for the user is simpler than having to tune the distance sensor first and then read off of it. The only disadvantage is that there's no fine-control over how fast the sensor is making the readings.

```
import time
from di_sensors.distance_sensor import DistanceSensor

print("Example program for reading a Dexter Industries Distance Sensor on an I2C port.
↪")

ds = DistanceSensor()

while True:
    # read the distance as a single-shot sample
    read_distance = ds.read_range_single()
    print("distance from object: {} mm".format(read_distance))
```

The source code for this example program can be found here on github.

3.1.4 Console Output

All 3 example scripts described in this chapter should have a console output similar to what we have next.

```
distance from object: 419 mm
distance from object: 454 mm
distance from object: 452 mm
distance from object: 490 mm
distance from object: 501 mm
distance from object: 8190 mm
distance from object: 1650 mm
distance from object: 1678 mm
distance from object: 1638 mm
distance from object: 1600 mm
```

3.2 Using the Light and Color Sensor

In this short section, we get to see how one can read data off of the Light and Color Sensor without having to fine-tune the sensor or to deal with hard-to-understand concepts. Before anything else, connect the Light and Color Sensor to an I2C port on whichever platform (be it a GoPiGo3, GrovePi or a BrickPi3) and then run the following script.

The source file for this example program can be found here on github.

```
from time import sleep
from di_sensors.easy_light_color_sensor import EasyLightColorSensor

print("Example program for reading a Dexter Industries Light Color Sensor on an I2C␣
↪port.")

my_lcs = EasyLightColorSensor(led_state = True)

while True:
    # Read the R, G, B, C color values
    red, green, blue, clear = my_lcs.safe_raw_colors()

    # Print the values
    print("Red: {:5.3f} Green: {:5.3f} Blue: {:5.3f} Clear: {:5.3f}".format(red,␣
↪green, blue, clear))

    sleep(0.02)
```

Here's how the output of the script should look like:

```
Example program for reading a Dexter Industries Light Color Sensor on an I2C port.
Red: 0.004 Green: 0.004 Blue: 0.004 Clear: 0.013
Red: 0.005 Green: 0.004 Blue: 0.004 Clear: 0.013
Red: 0.005 Green: 0.005 Blue: 0.004 Clear: 0.014
Red: 0.005 Green: 0.005 Blue: 0.004 Clear: 0.015
Red: 0.005 Green: 0.005 Blue: 0.004 Clear: 0.014
Red: 0.005 Green: 0.005 Blue: 0.004 Clear: 0.014
Red: 0.006 Green: 0.005 Blue: 0.005 Clear: 0.015
```

3.3 Temperature Humidity and Pressure Sensor

In order to run this example program, connect the Temperature Humidity and Pressure Sensor to an I2C port on whichever platform (GoPiGo3, GrovePi or BrickPi3) and then run the following script.

The source file for this example program can be found here on github.

```
from time import sleep
from di_sensors.easy_temp_hum_press import EasyTHPSensor

print("Example program for reading a Dexter Industries Temperature Humidity Pressure␣
↪Sensor on an I2C port.")

my_thp = EasyTHPSensor()

while True:
    # Read the temperature
    temp = my_thp.safe_celsius()

    # Read the relative humidity
    hum = my_thp.safe_humidity()

    # Read the pressure
    press = my_thp.safe_pressure()

    # Print the values
    print("Temperature: {:5.3f} Humidity: {:5.3f} Pressure: {:5.3f}".format(temp, hum,
↪ press))

    sleep(0.02)
```

The console output of this script should look like:

```
Example program for reading a Dexter Industries Temperature Humidity Pressure Sensor␣
↪on an I2C port.
Temperature: 28.139 Humidity: 48.687 Pressure: 101122.691
Temperature: 28.141 Humidity: 48.698 Pressure: 101122.840
Temperature: 28.145 Humidity: 48.385 Pressure: 101122.900
Temperature: 28.151 Humidity: 48.715 Pressure: 101122.889
Temperature: 28.157 Humidity: 48.436 Pressure: 101122.607
Temperature: 28.163 Humidity: 48.464 Pressure: 101122.836
Temperature: 28.171 Humidity: 48.674 Pressure: 101123.085
Temperature: 28.180 Humidity: 48.120 Pressure: 101123.114
```

3.4 Using the IMU Sensor

In order to run this example program, we need to have a GoPiGo3 because bus `"GPG3_AD1"` is used in this case and it's specific to the GoPiGo3 platform. The `"GPG3_AD1"` bus translates to port `"AD1"` on the GoPiGo3, so the IMU Sensor has to be connected to port `"AD1"`.

We could have gone with the default `"RPI_1SW"` bus so it can be used on any platform, but since this is an example, we might as-well show how it's being done with a GoPiGo3.

The source file for this example program can be found here on github.

```
import time
from di_sensors.inertial_measurement_unit import InertialMeasurementUnit

print("Example program for reading a Dexter Industries IMU Sensor on a GoPiGo3 AD1␣
↪port.")

imu = InertialMeasurementUnit(bus = "GPG3_AD1")

while True:
    # Read the magnetometer, gyroscope, accelerometer, euler, and temperature values
    mag   = imu.read_magnetometer()
    gyro  = imu.read_gyroscope()
    accel = imu.read_accelerometer()
    euler = imu.read_euler()
    temp  = imu.read_temperature()

    string_to_print = "Magnetometer X: {:.1f}  Y: {:.1f}  Z: {:.1f} " \
                      "Gyroscope X: {:.1f}  Y: {:.1f}  Z: {:.1f} " \
                      "Accelerometer X: {:.1f}  Y: {:.1f} Z: {:.1f} " \
                      "Euler Heading: {:.1f}  Roll: {:.1f}  Pitch: {:.1f} " \
                      "Temperature: {:.1f}C".format(mag[0], mag[1], mag[2],
                                                    gyro[0], gyro[1], gyro[2],
                                                    accel[0], accel[1], accel[2],
                                                    euler[0], euler[1], euler[2],
                                                    temp)
    print(string_to_print)

    time.sleep(0.1)
```

The console output of this script should look like:

```
Example program for reading a Dexter Industries IMU Sensor on a GoPiGo3 AD1 port.
Magnetometer X: 0.0  Y: 0.0  Z: 0.0 Gyroscope X: 54.9  Y: -25.4  Z: 8.8 Accelerometer␣
↪X: 9.8  Y: 9.5 Z: -3.5 Euler Heading: 0.0  Roll: 0.0  Pitch: 0.0 Temperature: 31.0C
Magnetometer X: -44.2  Y: 12.2  Z: 15.8 Gyroscope X: -38.6  Y: 12.4  Z: -116.7␣
↪Accelerometer X: -1.2  Y: 4.0 Z: -7.4 Euler Heading: 354.7  Roll: 6.3  Pitch: 13.6␣
↪Temperature: 31.0C
Magnetometer X: -44.6  Y: 15.2  Z: 18.6 Gyroscope X: -11.7  Y: 5.0  Z: 18.5␣
↪Accelerometer X: 6.5  Y: 7.0 Z: -1.4 Euler Heading: 354.2  Roll: 6.2  Pitch: 12.6␣
↪Temperature: 31.0C
Magnetometer X: -47.9  Y: 14.5  Z: 17.8 Gyroscope X: 17.1  Y: -23.1  Z: 43.0␣
↪Accelerometer X: 6.6  Y: 7.1 Z: -2.2 Euler Heading: 350.6  Roll: 8.3  Pitch: 13.2␣
↪Temperature: 31.0C
Magnetometer X: -30.5  Y: 11.0  Z: 13.0 Gyroscope X: -8.6  Y: -2.1  Z: -0.1␣
↪Accelerometer X: 6.2  Y: 5.7 Z: -3.5 Euler Heading: 2.7  Roll: 8.8  Pitch: 12.6␣
↪Temperature: 31.0C
Magnetometer X: -33.2  Y: 10.4  Z: 15.2 Gyroscope X: -87.0  Y: -29.6  Z: 141.0␣
↪Accelerometer X: 9.1  Y: 4.8 Z: -1.9 Euler Heading: 332.2  Roll: 15.8  Pitch: 2.1␣
↪Temperature: 31.0C
```

3.5 Using the Line Follower

3.5.1 Reading the Sensor

In order to run this example program, we need to have a GoPiGo3 because bus `"GPG3_AD1"` is used in this case and it's specific to the GoPiGo3 platform. The `"GPG3_AD1"` bus translates to port `"AD1"` on the GoPiGo3, so the Line Follower Sensor has to be connected to port `"AD1"`.

The default `"RPI_1SW"` bus could have been used, but since this is an example, let's do it with a GoPiGo3.

This is the exact same scenario as the one in *IMU sensor example*.

The source file for this example program can be found here on github.

```
import time
from di_sensors import line_follower

print("Example program for reading a Dexter Industries Line Follower sensor on GPG3␣
↪AD1 port")

lf = line_follower.LineFollower(bus = "GPG3_AD1")

'''
cd ~/Dexter/DI_Sensors/Python
sudo python setup.py install

python ~/Dexter/DI_Sensors/Python/Examples/LineFollower.py

'''

print("Manufacturer     : %s" % lf.get_manufacturer())
print("Name             : %s" % lf.get_board())
print("Firmware Version : %d" % lf.get_version_firmware())

while True:
    # Read the line sensors values
    values = lf.read_sensors()
    str = ""
    for v in range(len(values)):
        str += "%.3f " % values[v]
    print(str)

    time.sleep(0.1)
```

The console output of this script should look like:

```
Example program for reading a Dexter Industries Line Follower sensor on GPG3 AD1 port
Manufacturer     : Dexter Industries
Name             : Line Follower
Firmware Version : 1
0.031 0.044 0.051 0.038 0.033 0.017
0.035 0.054 0.063 0.051 0.048 0.026
0.048 0.065 0.075 0.061 0.059 0.034
0.047 0.065 0.076 0.063 0.064 0.040
0.063 0.080 0.096 0.091 0.091 0.071
0.070 0.087 0.104 0.103 0.107 0.087
0.070 0.088 0.107 0.103 0.108 0.090
```

(continues on next page)

(continued from previous page)

```
0.120 0.090 0.115 0.121 0.128 0.116
0.283 0.089 0.121 0.141 0.177 0.196
0.383 0.109 0.155 0.206 0.299 0.404
0.454 0.159 0.185 0.250 0.396 0.340
0.287 0.338 0.111 0.115 0.132 0.137
0.443 0.208 0.268 0.132 0.109 0.079
0.285 0.484 0.259 0.167 0.095 0.057
0.589 0.200 0.268 0.176 0.091 0.051
0.695 0.158 0.180 0.359 0.219 0.157
0.625 0.095 0.152 0.369 0.227 0.069
0.398 0.098 0.127 0.383 0.372 0.081
0.698 0.111 0.128 0.720 0.902 0.193
0.297 0.124 0.138 0.447 0.843 0.187
0.104 0.146 0.180 0.392 0.960 0.291
0.096 0.132 0.158 0.319 0.945 0.299
0.094 0.128 0.152 0.376 0.935 0.249
0.084 0.114 0.144 0.710 0.716 0.143
0.065 0.095 0.567 0.617 0.149 0.157
0.061 0.298 0.331 0.096 0.109 0.117
0.275 0.239 0.102 0.112 0.144 0.170
```

3.5.2 Basic Line Follower Controller

The simplest way to make the GoPiGo3 follow a line with the Line Follower Sensor is by having a controller that can output 3 different states: one for the **left**, one for the **center** when it's properly aligned and another one for the **right**. When the line that the robot is following is on the left side, then command the robot to go to the left, otherwise go to the right.

That simple! Now, let's see how that code looks like. The following example was done for the GoPiGo3 but it can easily be adapted for any other robot that has motors.

```
import easygopigo3 as easy
from di_sensors.easy_line_follower import EasyLineFollower

gpg = easy.EasyGoPiGo3()
my_linefollower = EasyLineFollower()

gpg.forward()
while not  my_linefollower.position() == "black":
    line_pos = my_linefollower.position()
    if line_pos == 'center':
        gpg.forward()
    elif line_pos == 'left':
        gpg.left()
    elif line_pos == 'right':
        gpg.right()
gpg.stop()
```

When running this in reality, the result is going to be choppy, so if you need a better control of the robot's trajectory, go on and check the next section.

3.5.3 PID-Based Controller

Also, doing something more advanced with the line follower is possible by using the `EasyLineFollower` class. With an object of such type, a estimated position of the black line can be returned and then feed this estimate into a PID controller. Consequently, a robot (such as the GoPiGo3) can be precisely controlled.

```python
from di_sensors.easy_line_follower import EasyLineFollower
from time import time, sleep

setpoint = 0.5
integralArea = 0.0
previousError = 0.0
motorBaseSpeed = 300
loopTime = 1.0 / 100

Kp = 0.0 # a value suitable for this component
Ki = 0.0 # ditto
Kd = 0.0 # ditto as above

lf = EasyLineFollower()

while True:
    start = time()
    pos, out_of_line = lf.read_sensors('weighted-avg')
    error = pos - setpoint

    integralArea += error
    correction = Kp * error + Ki * integralArea + Kd * (previousError - error)
    previousError = error

    motorA = motorBaseSpeed + correction
    motorB = motorBaseSpeed - correction

    # code for actuating the robot to follow the line
    # using the previously computed values for each motor

    # to make it run at certain frequency
    end = time()
    alreadySpent = end - start
    remainingTime = loopTime - alreadySpent
    if remainingTime > 0:
        sleep(remainingTime)
```

With something like the above code, you can make a pretty reliable control system for the line follower.

3.6 Using Mutexes

In this section, we are showing how handy mutexes are when we're trying to access the same resource (a device, for instance a Distance Sensor) simultaneously from multiple threads. All *Easy classes* are thread-safe - what one has to do is to activate the use of mutexes by passing a boolean parameter to each of the classes' constructor.

In the following example program, 2 threads are accessing the resource of an `EasyDistanceSensor` object. `use_mutex` parameter is set to `True` so that the resource can be accessed from multiple threads/processes (this is what we would call a thread-safe class). Each of these 2 threads run for `runtime` seconds - we didn't make it so one can stop the program while it's running, because that would have been more complex.

Without the mutex mechanism, accessing the same resource from multiple processes/threads would not be possible.

```python
# do the import stuff
from di_sensors.easy_distance_sensor import EasyDistanceSensor
from time import time, sleep
from threading import Thread, Event, get_ident

# instantiate the distance object
my_sensor = EasyDistanceSensor(use_mutex = True)
start_time = time()
runtime = 2.0
# create an event object for triggering the "shutdown" of each thread
stop_event = Event()

# target function for each thread
def readingSensor():
    while not stop_event.is_set():
      thread_id = get_ident()
      distance = my_sensor.read()
      print("Thread ID = {} with distance value = {}".format(thread_id, distance))
      sleep(0.001)

# create an object for each thread
thread1 = Thread(target = readingSensor)
thread2 = Thread(target = readingSensor)

# and then start them
thread1.start()
thread2.start()

# let it run for [runtime] seconds
while time() - start_time <= runtime:
    sleep(0.1)

# and then set the stop event variable
stop_event.set()

# and wait both threads to end
thread1.join()
thread2.join()
```

Important: There was no need to use mutexes in the above example, but for the sake of an example, it is a good thing. The idea is that CPython's implementation has what it's called a **GIL** (*Global Interpreter Lock*) and this only allows one thread to run at once, which is a skewed way of envisioning how threads work, but it's the reality in Python still. Ideally, a thread can run concurrently with another one. You can read more on the Global Interpreter Lock here.

Still, the implementation we have with mutexes proves to be useful when one wants to launch multiple processes at a time - at that moment, we can talk of true concurrency. This can happen when multiple instances of Python scripts are launched and when each process tries to access the same resource as the other one.

The output on the console should look like this - the thread IDs don't mean anything and they are merely just a number used to identify threads.

```
Thread ID = 1883501680 with distance value = 44
Thread ID = 1873802352 with distance value = 44
Thread ID = 1873802352 with distance value = 44
```

(continues on next page)

(continued from previous page)

```
Thread ID = 1883501680 with distance value = 44
Thread ID = 1873802352 with distance value = 46
Thread ID = 1883501680 with distance value = 46
Thread ID = 1873802352 with distance value = 45
Thread ID = 1883501680 with distance value = 45
Thread ID = 1883501680 with distance value = 44
Thread ID = 1873802352 with distance value = 44
Thread ID = 1883501680 with distance value = 45
Thread ID = 1873802352 with distance value = 45
```

CHAPTER

FOUR

ON LIBRARY & HARDWARE

4.1 Requirements

Before you check the API for the DI-Sensors, please make sure you have the `DI-Sensors` package installed. You can do this by checking with `pip` by typing the following command.

```
pip show DI-Sensors
```

Or you can check by trying to import the package in a Python console the following way:

```
import di_sensors
```

If there's nothing to be shown when `pip show`-ing or you get an import error on the `di_sensors` package, then please check the *Getting Started* section and follow the instructions.

4.2 Hardware interface

Instantiating the *4 sensors* in Python is a matter of choosing the right bus. Thus, there are 4 buses to choose from, depending on the context:

- The `"RPI_1SW"` bus - this can be used along all 5 platforms we have (the GoPiGo3, GoPiGo, BrickPi3, GrovePi & PivotPi). This bus corresponds to the `"I2C"` port.
- The `"RPI_1"` bus - this bus can be used along all 5 platforms we have (the GoPiGo3, GoPiGo, BrickPi3, GrovePi & PivotPi). Does not correspond to the `"I2C"` port.
- The `"GPG3_AD1"`/`"GPG3_AD2"` buses - these buses can **only** be used on the GoPiGo3 platform. The advantage of using these ones is that the interface between the Raspberry Pi and the sensor is more stable. These buses correspond to the `"AD1"` and `"AD2"` ports of the GoPiGo3.

Important: These notations for ports (`"RPI_1SW"`, `"RPI_1"`, `"GPG3_AD1"` and `"GPG3_AD2"`) are only required for classes that *don't start* with the **Easy** word, specifically for:

- *`DistanceSensor`*
- `InertialMeasurementUnitSensor`
- *`LightColorSensor`*
- *`TempHumPress`*

If you choose to use a sensor library *that starts* with the **Easy** word, you can use the same notations as those used and mentioned in the GoPiGo3's documentation, such as:

- `"I2C"` instead of `"RPI_1SW"`.
- `"AD1/AD2"` instead of `"GPG3_AD1/GPG3_AD2"`.

Also, you may notice that for the `"I2C"` port we only support the `"RPI_1SW"`, which is a software implementation for the I2C so that the hardware one can be avoided. The problem with the hardware implementation (the `"RPI_1"` bus) is that it's riddled with bugs and if you don't want your application to crash, use the software implemented one. The software implemented driver for the I2C is as fast the HW one and it doesn't take much CPU time at all.

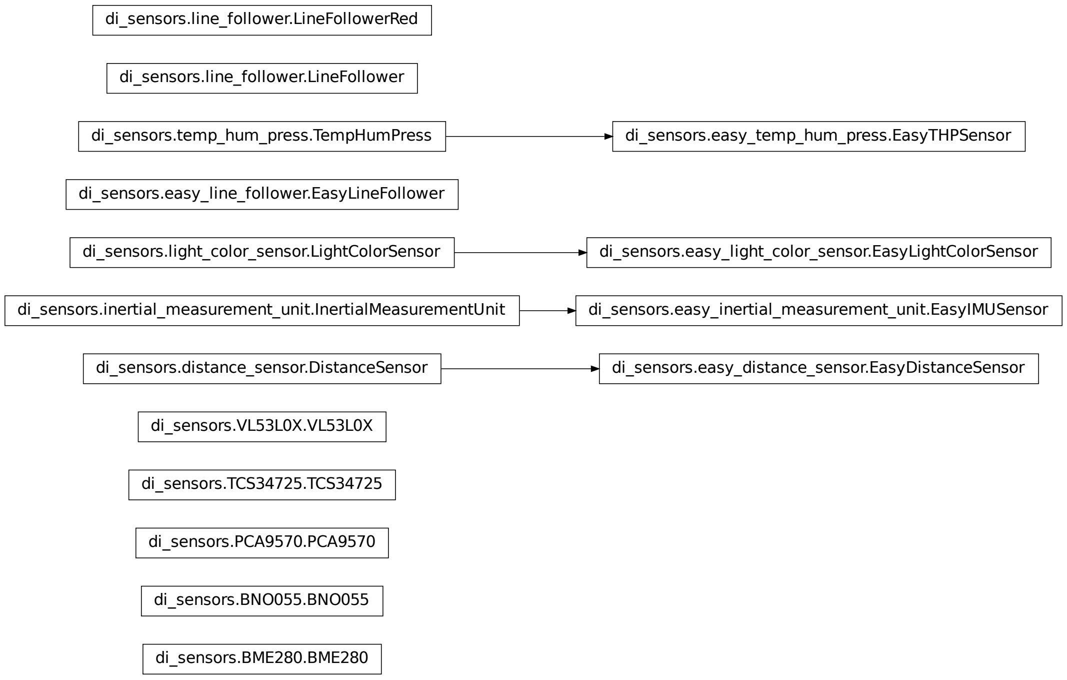
For seeing where the `"AD1"`/`"AD2"` are located on the GoPiGo3, please check the GoPiGo3's documentation.

4.3 Library Structure

4.3.1 Classes Short-List

The classes that are more likely to be of interest are graphically displayed shortly after this. In this graphic you can also notice inheritance links between different classes. We can notice 3 groups of classes:

- Those that start with the **Easy** word in them and are easier to use and may provide some high-level functionalities.
- Those that don't start with the **Easy** word and yet are related to those that are. These are generally intented for power users.
- Those that look like they might represent a model number (that belong to modules such as `di_sensors.VL53L0X`, `di_sensors.BME280`, etc). These are intented for those who want to extend the functionalities of our library and are not documented here.


di_sensors.line_follower.LineFollowerRed
di_sensors.line_follower.LineFollower
di_sensors.temp_hum_press.TempHumPress
di_sensors.easy_temp_hum_press.EasyTHPSensor
di_sensors.easy_line_follower.EasyLineFollower
di_sensors.light_color_sensor.LightColorSensor
di_sensors.easy_light_color_sensor.EasyLightColorSensor
di_sensors.inertial_measurement_unit.InertialMeasurementUnit
di_sensors.easy_inertial_measurement_unit.EasyIMUSensor
di_sensors.distance_sensor.DistanceSensor
di_sensors.easy_distance_sensor.EasyDistanceSensor
di_sensors.VL53L0X.VL53L0X
di_sensors.TCS34725.TCS34725
di_sensors.PCA9570.PCA9570
di_sensors.BNO055.BNO055
di_sensors.BME280.BME280

Note: Since this is an interactive graphic, you can click on the displayed classes and it'll take you to the documentation of a given class, if provided.

4.3.2 Functions Short-List

Here's a short summary of all classes and methods. There's a list going on for each class. We first start off by listing the **Easy** classes/methods and then we end up showing the classes/methods for power users. In this short summary, we're not covering the low-level classes that are not even documented in this documentation.

Easy - TempHumPress

di_sensors.easy_temp_hum_press.EasyTHPSensor([...])	Class for interfacing with the Temperature Humidity Pressure Sensor.
di_sensors.easy_temp_hum_press.EasyTHPSensor.__init__([...])	Constructor for initializing link with the Temperature Humidity Pressure Sensor.
di_sensors.easy_temp_hum_press.EasyTHPSensor.safe_celsius()	Read temperature in Celsius degrees.
di_sensors.easy_temp_hum_press.EasyTHPSensor.safe_fahrenheit()	Read temperature in Fahrenheit degrees.
di_sensors.easy_temp_hum_press.EasyTHPSensor.safe_pressure()	Read the air pressure in pascals.

Continued on next page

Table 1 – continued from previous page

di_sensors.easy_temp_hum_press. EasyTHPSensor.safe_humidity()	Read the relative humidity as a percentage.

Easy - Light & Color

di_sensors.easy_light_color_sensor. EasyLightColorSensor([. . .])	Class for interfacing with the Light Color Sensor.
di_sensors.easy_light_color_sensor. EasyLightColorSensor.__init__([. . .])	Constructor for initializing a link to the Light Color Sensor.
di_sensors.easy_light_color_sensor. EasyLightColorSensor. translate_to_hsv(. . .)	Standard algorithm to switch from one color system (**RGB**) to another (**HSV**).
di_sensors.easy_light_color_sensor. EasyLightColorSensor. safe_raw_colors()	Returns the color as read by the Light Color Sensor.
di_sensors.easy_light_color_sensor. EasyLightColorSensor.safe_rgb()	Detect the RGB color off of the Light Color Sensor.
di_sensors.easy_light_color_sensor. EasyLightColorSensor. guess_color_hsv(. . .)	Determines which color *in_color* parameter is closest to in the `known_colors` list.

Easy - Distance

di_sensors.easy_distance_sensor. EasyDistanceSensor([. . .])	Class for the Distance Sensor device.
di_sensors.easy_distance_sensor. EasyDistanceSensor.__init__([. . .])	Creates a `EasyDistanceSensor` object which can be used for interfacing with a distance sensor.
di_sensors.easy_distance_sensor. EasyDistanceSensor.read_mm()	Reads the distance in millimeters.
di_sensors.easy_distance_sensor. EasyDistanceSensor.read()	Reads the distance in centimeters.
di_sensors.easy_distance_sensor. EasyDistanceSensor.read_inches()	Reads the distance in inches.

Easy - IMU

di_sensors.easy_inertial_measurement_un EasyIMUSensor([. . .])	Class for interfacing with the InertialMeasurementUnit Sensor.
di_sensors.easy_inertial_measurement_un EasyIMUSensor.__init__([. . .])	Constructor for initializing link with the InertialMeasurementUnit Sensor.
di_sensors.easy_inertial_measurement_un EasyIMUSensor.reconfig_bus()	Use this method when the InertialMeasurementUnit Sensor becomes unresponsive but it's still plugged into the board.
di_sensors.easy_inertial_measurement_un EasyIMUSensor.safe_calibrate()	Once called, the method returns when the magnetometer of the InertialMeasurementUnit Sensor gets fully calibrated.

Continued on next page

Table 4 – continued from previous page

di_sensors.easy_inertial_measurement_un... EasyIMUSensor.safe_calibration_status()	Returns the calibration level of the magnetometer of the InertialMeasurementUnit Sensor.
di_sensors.easy_inertial_measurement_un... EasyIMUSensor.convert_heading(...)	This method takes in a heading in degrees and return the name of the corresponding heading.
di_sensors.easy_inertial_measurement_un... EasyIMUSensor.safe_read_euler()	Read the absolute orientation.
di_sensors.easy_inertial_measurement_un... EasyIMUSensor.safe_read_magnetometer()	Read the magnetometer values.
di_sensors.easy_inertial_measurement_un... EasyIMUSensor.safe_north_point()	Determines the heading of the north point.

Easy - Line Follower

di_sensors.easy_line_follower. EasyLineFollower([...])	Higher-level of abstraction class for either the LineFollower or LineFollowerRed.
di_sensors.easy_line_follower. EasyLineFollower.__init__([...])	Initialize a class to interface with either the LineFollower or the LineFollowerRed.
di_sensors.easy_line_follower. EasyLineFollower.set_calibration(color)	Calibrate the sensor for the given `color` and save the values to file.
di_sensors.easy_line_follower. EasyLineFollower.get_calibration(color)	Read the calibration values from the disk for the given `color`.
di_sensors.easy_line_follower. EasyLineFollower.read([...])	Read the sensors' values from either line follower.
di_sensors.easy_line_follower. EasyLineFollower.position_01()	Same as calling `read()` method like `read("bivariate")`.
di_sensors.easy_line_follower. EasyLineFollower.position_bw()	Same as calling `read()` method like `read("bivariate-str")`.
di_sensors.easy_line_follower. EasyLineFollower.position()	Returns a string telling to which side the black line that we're following is located.
di_sensors.easy_line_follower. EasyLineFollower.position_val()	Same as calling `read()` method like `read("weighted-avg")`.

TempHumPress

di_sensors.temp_hum_press. TempHumPress([bus])	Class for interfacing with the Temperature Humidity Pressure Sensor.
di_sensors.temp_hum_press. TempHumPress.__init__([bus])	Constructor for initializing link with the Temperature Humidity Pressure Sensor.
di_sensors.temp_hum_press. TempHumPress.get_temperature_celsius()	Read temperature in Celsius degrees.
di_sensors.temp_hum_press. TempHumPress.get_temperature_fahrenheit()	Read temperature in Fahrenheit degrees.
di_sensors.temp_hum_press. TempHumPress.get_pressure()	Read the air pressure in pascals.
di_sensors.temp_hum_press. TempHumPress.get_humidity()	Read the relative humidity as a percentage.
di_sensors.temp_hum_press. TempHumPress.get_humidity()	Read the relative humidity as a percentage.

Light & Color

di_sensors.light_color_sensor. LightColorSensor([…])	Class for interfacing with the Light Color Sensor.
di_sensors.light_color_sensor. LightColorSensor.__init__([…])	Constructor for initializing a link to the Light Color Sensor.
di_sensors.light_color_sensor. LightColorSensor.set_led(value)	Set the LED state.
di_sensors.light_color_sensor. LightColorSensor.get_raw_colors([delay])	Read the sensor values.

Distance

di_sensors.distance_sensor. DistanceSensor([bus])	Class for interfacing with the Distance Sensor.
di_sensors.distance_sensor. DistanceSensor.__init__([bus])	Constructor for initializing a *DistanceSensor* class.
di_sensors.distance_sensor. DistanceSensor.start_continuous([…])	Start taking continuous measurements.
di_sensors.distance_sensor. DistanceSensor.read_range_continuous()	Read the detected range while the sensor is taking continuous measurements at the set rate.
di_sensors.distance_sensor. DistanceSensor.read_range_single([…])	Read the detected range with a single measurement.
di_sensors.distance_sensor. DistanceSensor.timeout_occurred()	Checks if a timeout has occurred on the *read_range_continuous()* method.

IMU

di_sensors.inertial_measurement_unit. InertialMeasurementUnit([bus])	Class for interfacing with the InertialMeasurementUnit Sensor.
di_sensors.inertial_measurement_unit. InertialMeasurementUnit.__init__([bus])	Constructor for initializing link with the InertialMeasurementUnit Sensor.
di_sensors.inertial_measurement_unit. InertialMeasurementUnit.read_euler()	Read the absolute orientation.
di_sensors.inertial_measurement_unit. InertialMeasurementUnit. read_magnetometer()	Read the magnetometer values.
di_sensors.inertial_measurement_unit. InertialMeasurementUnit. read_gyroscope()	Read the angular velocity of the gyroscope.
di_sensors.inertial_measurement_unit. InertialMeasurementUnit. read_accelerometer()	Read the accelerometer.
di_sensors.inertial_measurement_unit. InertialMeasurementUnit. read_linear_acceleration()	Read the linear acceleration - that is, the acceleration from movement and without the gravitational acceleration in it.

Continued on next page

Table 9 – continued from previous page

di_sensors.inertial_measurement_unit.InertialMeasurementUnit.read_gravity()	Read the gravitational acceleration.
di_sensors.inertial_measurement_unit.InertialMeasurementUnit.read_quaternion()	Read the quaternion values.
di_sensors.inertial_measurement_unit.InertialMeasurementUnit.read_temperature()	Read the temperature in Celsius degrees.

Line Follower Black/Red

di_sensors.line_follower.LineFollower([bus])	Class for interfacing with the Line Follower Sensor (black board).
di_sensors.line_follower.LineFollower.__init__([bus])	Constructor for initializing an object to interface with the Line Follower Sensor (black board).
di_sensors.line_follower.LineFollower.read_sensors()	Read the line follower's values.
di_sensors.line_follower.LineFollower.get_manufacturer()	Read the manufacturer of the Line Follower Sensor (black board)'s.
di_sensors.line_follower.LineFollower.get_board()	Read the board name of the Line Follower Sensor (black board).
di_sensors.line_follower.LineFollower.get_version_firmware()	Get the firmware version currently residing on the Line Follower Sensor (black board).

di_sensors.line_follower.LineFollowerRed([bus])	Class for interfacing with the depreciated Line Follower Sensor (red board).
di_sensors.line_follower.LineFollowerRed.__init__([bus])	Constructor for initializing an object to interface with the depreciated Line Follower Sensor (red board).
di_sensors.line_follower.LineFollowerRed.read_sensors()	Read the line follower's values.

CHAPTER

FIVE

API DI-SENSORS - BASIC

5.1 EasyDistanceSensor

class `di_sensors.easy_distance_sensor.`**`EasyDistanceSensor`**(*port='I2C'*, *use_mutex=False*)

Bases: *`di_sensors.distance_sensor.DistanceSensor`*

Class for the Distance Sensor device.

This class compared to *`DistanceSensor`* uses mutexes that allows a given object to be accessed simultaneously from multiple threads/processes. Apart from this difference, there may also be functions that are more user-friendly than the latter.

`__init__`(*port='I2C'*, *use_mutex=False*)

Creates a `EasyDistanceSensor` object which can be used for interfacing with a distance sensor.

Parameters

- **`bus = "I2C"`** (*string*) – the bus for the sensor. For the GoPiGo3, options also include `"GPG3_AD1"` and `"GPG3_AD2"`.
- **`use_mutex = False`** (*bool*) – When using multiple threads/processes that access the same resource/device, mutexes should be enabled. Check the *hardware specs* for more information about the ports.

Raises **`OSError`** – When the distance sensor is not connected to the designated bus/port, where in this case it must be `"I2C"`. Most probably, this means the distance sensor is not connected at all.

To see where the ports are located on the GoPiGo3 robot, please take a look at the following diagram: Hardware Ports.

`read_mm`()

Reads the distance in millimeters.

Returns Distance from target in millimeters.

Return type int

Note:

1. Sensor's range is **5-2300** millimeters.
2. When the values are out of the range, it returns **3000**.

`read`()

Reads the distance in centimeters.

Returns Distance from target in centimeters.

Return type int

Note:

1. Sensor's range is **0-230** centimeters.
2. When the values are out of the range, it returns **300**.

`read_inches()`

Reads the distance in inches.

Returns Distance from target in inches.

Return type float with one decimal

Note:

1. Sensor's range is **0-90** inches.
2. Anything that's bigger than **90** inches is returned when the sensor can't detect any target/surface.

5.2 EasyLightColorSensor

class `di_sensors.easy_light_color_sensor.`**`EasyLightColorSensor`**(*port='I2C'*, *led_state=False*, *use_mutex=False*)

Bases: *`di_sensors.light_color_sensor.LightColorSensor`*

Class for interfacing with the Light Color Sensor. This class compared to *`LightColorSensor`* uses mutexes that allows a given object to be accessed simultaneously from multiple threads/processes. Apart from this difference, there may also be functions that are more user-friendly than the latter.

`known_colors = {'black': (0, 0, 0), 'blue': (0, 0, 255), 'cyan': (0, 255, 255), 'fu`

The 8 colors that *`guess_color_hsv()`* method may return upon reading and interpreting a new set of color values.

`__init__`(*port='I2C'*, *led_state=False*, *use_mutex=False*)

Constructor for initializing a link to the Light Color Sensor.

Parameters

- **`port = "I2C"`** (*`str`*) – The port to which the distance sensor is connected to. Can also be connected to ports `"AD1"` or `"AD2"` of the GoPiGo3. If you're passing an **invalid port**, then the sensor resorts to an `"I2C"` connection. Check the *hardware specs* for more information about the ports.
- **`led_state = False`** (*`bool`*) – The LED state. If it's set to `True`, then the LED will turn on, otherwise the LED will stay off. By default, the LED is turned off.
- **`use_mutex = False`** (*`bool`*) – When using multiple threads/processes that access the same resource/device, mutexes should be enabled.

Raises

- **`OSError`** – When the Light Color Sensor is not reachable.

- **`RuntimeError`** – When the chip ID is incorrect. This happens when we have a device pointing to the same address, but it's not a Light Color Sensor.

`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.

`safe_raw_colors`()

Returns the color as read by the Light Color Sensor. The colors detected vary depending on the lighting conditions of the nearby environment.

Returns The RGBA values from the sensor. RGBA = Red, Green, Blue, Alpha (or Clear). Range of each element is between **0** and **1**. **-1** means an error occured.

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

`safe_rgb`()

Detect the RGB color off of the Light Color Sensor.

Returns The RGB color in 8-bit format.

Return type tuple(int,int,int)

`calibrate`(*color*)

Replace the HSV centroid for a given color with the sensor reading obtained from an example of that color in the current lighting environment <color> can be one of black | white | red | green | blue | yellow | cyan | fuschia

`guess_color_hsv`(*in_color*)

Determines which color *in_color* parameter is closest to in the *`known_colors`* list. This method uses the euclidean algorithm for detecting the nearest center to it out of *`known_colors`* list. It does work exactly the same as KNN (K-Nearest-Neighbors) algorithm, where *K = 1*.

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 **1**.

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.

5.3 EasyIMUSensor

class `di_sensors.easy_inertial_measurement_unit.`**`EasyIMUSensor`**(*port='AD1', use_mutex=False*)

Bases: *`di_sensors.inertial_measurement_unit.InertialMeasurementUnit`*

Class for interfacing with the InertialMeasurementUnit Sensor.

This class compared to *`InertialMeasurementUnit`* uses mutexes that allows a given object to be accessed simultaneously from multiple threads/processes. Apart from this difference, there may also be functions that are more user-friendly than the latter.

`__init__`(*port='AD1', use_mutex=False*)

Constructor for initializing link with the InertialMeasurementUnit Sensor.

Parameters

- **`port = "AD1"`** (*`str`*) – The port to which the IMU sensor gets connected to. Can also be connected to port `"AD2"` of a GoPiGo3 robot or to any `"I2C"` port of any of our platforms. If you're passing an **invalid port**, then the sensor resorts to an `"I2C"` connection. Check the *hardware specs* for more information about the ports.
- **`use_mutex = False`** (*`bool`*) – When using multiple threads/processes that access the same resource/device, mutexes should be enabled.

Raises

- **`RuntimeError`** – When the chip ID is incorrect. This happens when we have a device pointing to the same address, but it's not a InertialMeasurementUnit Sensor.
- **`OSError`** – When the InertialMeasurementUnit Sensor is not reachable.

`reconfig_bus`()

Use this method when the InertialMeasurementUnit Sensor becomes unresponsive but it's still plugged into the board. There will be times when due to improper electrical contacts, the link between the sensor and the board gets disrupted - using this method restablishes the connection.

Note: Sometimes the sensor won't work just by calling this method - in this case, switching the port will do the job. This is something that happens very rarely, so there's no need to worry much about this scenario.

`safe_calibrate`()

Once called, the method returns when the magnetometer of the InertialMeasurementUnit Sensor gets fully calibrated. Rotate the sensor in the air to help the sensor calibrate faster.

Note: Also, this method is not used to trigger the process of calibrating the sensor (the IMU does that automatically), but its purpose is to block a given script until the sensor reports it has fully calibrated.

If you wish to block your code until the sensor calibrates and still have control over your script, use *`safe_calibration_status()`* method along with a `while` loop to continuously check it.

`safe_calibration_status`()

Returns the calibration level of the magnetometer of the InertialMeasurementUnit Sensor.

Returns Calibration level of the magnetometer. Range is **0-3** and **-1** is returned when the sensor can't be accessed.

Return type int

`convert_heading`(*in_heading*)

This method takes in a heading in degrees and return the name of the corresponding heading. :param float in_heading: the value in degree that needs to be converted to a string.

Returns The heading of the sensor as a string.

Return type str

The possible strings that can be returned are: `"North"`, `"North East"`, `"East"`, `"South East"`, `"South"`, `"South West"`, `"West"`, `"North West"`, `"North"`.

Note: First use `safe_calibrate()` or `safe_calibration_status()` methods to determine if the magnetometer sensor is fully calibrated.

`safe_read_euler`()

Read the absolute orientation.

Returns Tuple of euler angles in degrees of *heading*, *roll* and *pitch*.

Return type (float,float,float)

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

`safe_read_accelerometer`()

Read the acceleration in 3 axes.

Returns Tuple of x,y,z axes.

Return type (float,float,float)

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

`safe_read_magnetometer`()

Read the magnetometer values.

Returns Tuple containing X, Y, Z values in *micro-Teslas* units. You can check the X, Y, Z axes on the sensor itself.

Return type (float,float,float)

Note: In case of an exception occurring within this method, a tuple of 3 elements where all values are set to **0** is returned.

`safe_read_gyroscope`()

Read the angular velocity of the gyroscope.

Returns The angular velocity as a tuple of X, Y, Z values in *degrees/s*. You can check the X, Y, Z axes on the sensor itself.

Return type (float,float,float)

Note: In case of an exception occurring within this method, a tuple of 3 elements where all values are set to **0** is returned.

`safe_north_point`()

Determines the heading of the north point. This function doesn't take into account the declination.

Returns The heading of the north point measured in degrees. The north point is found at **0** degrees.

Return type int

Note: In case of an exception occurring within this method, **0** is returned.

5.4 EasyTHPSsensor

class `di_sensors.easy_temp_hum_press.`**`EasyTHPSensor`**(*port='I2C'*, *use_mutex=False*)

Bases: *`di_sensors.temp_hum_press.TempHumPress`*

Class for interfacing with the Temperature Humidity Pressure Sensor.

This class compared to *`TempHumPress`* uses mutexes that allows a given object to be accessed simultaneously from multiple threads/processes. Apart from this difference, there may also be functions that are more user-friendly than the latter.

`__init__`(*port='I2C'*, *use_mutex=False*)

Constructor for initializing link with the Temperature Humidity Pressure Sensor.

Parameters

- **`port = "I2C"`** (*str*) – The port to which the THP sensor is connected to. Can also be connected to ports `"AD1"` or `"AD2"` of the GoPiGo3. If you're passing an **invalid port**, then the sensor resorts to an `"I2C"` connection. Check the *hardware specs* for more information about the ports.
- **`use_mutex = False`** (*bool*) – When using multiple threads/processes that access the same resource/device, mutexes should be enabled.

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

`safe_celsius`()

Read temperature in Celsius degrees.

Returns Temperature in Celsius degrees.

Return type float

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

`safe_fahrenheit`()

Read temperature in Fahrenheit degrees.

Returns Temperature in Fahrenheit degrees.

Return type float

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

`safe_pressure`()

Read the air pressure in pascals.

Returns The air pressure in pascals.

Return type float

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

`safe_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.

5.5 EasyLineFollower

class `di_sensors.easy_line_follower.`**`EasyLineFollower`**(*port='I2C'*, *sensor_id=-1*, *calib_dir='/home/pi/Dexter/'*, *white_file='white_line.txt'*, *black_file='black_line.txt'*, *use_mutex=True*)

Bases: `object`

Higher-level of abstraction class for either the `LineFollower` or `LineFollowerRed`.

`__init__`(*port='I2C'*, *sensor_id=-1*, *calib_dir='/home/pi/Dexter/'*, *white_file='white_line.txt'*, *black_file='black_line.txt'*, *use_mutex=True*)

Initialize a class to interface with either the `LineFollower` or the `LineFollowerRed`.

Parameters

- **`port = "I2C"`** (*str*) – The port to which the line follower is connected. The `"I2C"` port corresponds to `"RPI_1SW"` bus. Can also choose port `"AD1"`/`"AD2"` only if it's connected to the GoPiGo3 and the line follower sensor (black board) is used. To find out more, check the *hardware specs* for more information about the ports.
- **`sensor_id = -1`** (*int*) – **-1** to automatically detect the connected line follower - this is the default value. It can also set to **1** to only use it with the line follower (red board) (`LineFollowerRed`) or to **2** for the line follower (black board) (`LineFollower`)[1].
- **`calib_dir = "/home/pi/Dexter/"`** (*str*) – Directory where the calibration files are saved. It already has a default value set.
- **`white_file = "white_line.txt"`** (*str*) – The name of the calibration file for the white line.
- **`black_file = "black_line.txt"`** (*str*) – The name of the calibration file for the black line.
- **`use_mutex = True`** (*bool*) – Whether to use a mutex on the sensor or not. Recommended when the same sensor is called from multiple threads/processes. It's meant for the I2C line and does not protect the file I/O in multi-threaded applications.

Upon instantiating an object of this class, after detecting the line follower, the calibration values are read and if they are not compatible with those required for the given line follower, default/generic calibration values will be set for both colors computed by taking the average of the 2 extremes.

Important to keep in mind is that both line followers' calibration files are incompatible, because one uses 5 sensors and the other one 6 - there are also, more factors to consider, such as the kind of sensors used in the line follower, but for the most part, the incompatibility comes from the different number of sensors.

`set_calibration`(*color*, *inplace=True*)

Calibrate the sensor for the given `color` and save the values to file.

Parameters

- **`color`** (*str*) – Either `"white"` for calibrating white or `"black"` for black.

[1] To see what module has been detected, check `_sensor_id` attribute. If it's set to 0, then no line follower has been detected.

- **inplace = True** (*bool*) – Apply the calibration values to this instantiated object too. Use `white_calibration` and `black_calibration` attributes to access the calibration values.

get_calibration(*color*, *inplace=True*)

Read the calibration values from the disk for the given `color`.

Parameters

- **color** (*str*) – Either `"white"` for reading the calibration values for white or `"black"` for black.
- **inplace = True** (*bool*) – Apply the read values to this instantiated object too. Use `white_calibration` `black_calibration` to access the calibration values.

Return type 5/6-element list depending on which line follower is used.

Returns The calibrated values for the given color.

Raises **ValueError** – When the read file is incompatible with what the line follower expects. This can happen if a line follower has been calibrated and then switched with another one of a different type (like going from the black -> red board or vice-versa).

read(*representation='raw'*)

Read the sensors' values from either line follower.

Parameters **representation="raw"** (*str*) – It's set by-default to `"raw"` , but it can also be `"bivariate"`, `"bivariate-str"` or `"weighted-avg"`.

Raises **OSError** – If the line follower sensor is not reachable.

Each of the line followers' order of the sensors' values is the same as the one in each read method of them both: *di_sensors.line_follower.LineFollower.read_sensors()* and *di_sensors.line_follower.LineFollowerRed.read_sensors()*.

For representation="raw" For this, raw values are returned from the line follower sensor. Values range between **0** and **1** and there can be 5 or 6 values returned depending on what line follower sensor is used.

For representation="bivariate" In this case, a list with the length equal to the number of sensors present on the given line follower is returned. Values are either **0** (for black) or **1** (for white). In order to get good results, make sure the line follower is properly calibrated.

For representation="bivariate-str" Same as `"bivariate"` except that **0** is replaced with letter *b* (for black) and **1** with *w* (for white).

For representation="weighted-avg" Returns a 2-element tuple. The first element is an estimated position of the line.

The estimate is computed using a weighted average of each sensor value (regardless of which line follower sensor is used), so that if the black line is on the left of the line follower, the returned value will be in the **0.0-0.5** range and if it's on the right, it's in the **0.5-1.0** range, thus making **0.5** the center point of the black line. Keep in mind that the sensor's orientation is determined by the order of the returned sensor values and not by how the sensor is positioned on the robot. Check *read_sensors()* and *read_sensors()* methods to see how the values are returned.

If the line follower sensor ends up on a surface with an homogeneous color (or shade of grey), the returned value will circle around **0.5**.

The 2nd element is an integer taking 3 values: **1** if the line follower only detects black, **2** if it only detects white and **0** for the rest of cases.

position_01()

Same as calling *read()* method like `read("bivariate")`.

Return type list(int)

Returns A list of 0s and 1s for each sensor of the line follower.

Raises Check *read()*.

position_bw()

Same as calling *read()* method like `read("bivariate-str")`.

Return type str

Returns A string with a bunch of `"w"` (for white) and `"b"` (for black) representing the detected color on each sensor.

Raises Check *read()*.

position()

Returns a string telling to which side the black line that we're following is located.

Returns String that's indicating the location of the black line.

Return type str

Raises Check *read()*.

Important: It is assumed that with this method, the line follower is properly oriented on the GoPiGo. For the red line follower, when looking forward, the **left** marking on the board is on the left and vice-versa for the **right** marking. As for the black line follower, the wiggly white arrow on the board is pointed forward.

The strings this method can return are the following:

- `"center"` - when the line is found in the middle.
- `"black"` - when the line follower sensor only detects black surfaces.
- `"white"` - when the line follower sensor only detects white surfaces.
- `"left"` - when the black line is located on the left of the sensor.
- `"right"` - when the black line is located on the right of the sensor.

position_val()

Same as calling *read()* method like `read("weighted-avg")`.

Return type int

Returns Range is between **0** and **30**. When following a line the values will be between 0 and 10. For values smaller than **5**, the black line is on the left of the robot, and for values bigger than **5** but no more than **10** the line is to the right. **5** suggests the black line is in the middle. It can also return **20** if it's all black, or **30** for all white.

Raises Check *read()*.

Warning: The Line Follower class was originally held in `easysensors` module of the GoPiGo3 library, but has been moved here. The `easygopigo3.EasyGoPiGo3.init_line_follower()` method now returns an object of the *EasyLineFollower* class instead of instantiating the original Line Follower class from `easysensors` module.

In order to prevent breaking others' code, we kept the support for the older methods that are soon-to-be-deprecated in *EasyLineFollower* class. The mapping between the old methods and the new ones is as follows:

1. `read_raw_sensors()` <=> *read()*
2. `read_binary()` <=> *position_01()*
3. `read_position()` <=> *position()*
4. `read_position_str()` <=> *position_bw()*
5. `get_white_calibration()` <=> *set_calibration()* `("white")`
6. `get_black_calibration()` <=> *set_calibration()* `("black")`

API DI-SENSORS - ADVANCED

6.1 DistanceSensor

class di_sensors.distance_sensor.**DistanceSensor**(*bus='RPI_1SW'*)
Bases: object

Class for interfacing with the Distance Sensor.

__init__(*bus='RPI_1SW'*)
Constructor for initializing a *DistanceSensor* class.

Parameters bus = "RPI_1SW" (*str*) – The bus to which the distance sensor is connected to. By default, it's set to bus "RPI_1SW". Check the *hardware specs* for more information about the ports.

Raises OSError – When the distance sensor is not connected to the designated bus/port. Most probably, this means the distance sensor is not connected at all.

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`(*safe_infinity=True*)
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.

Parameters **`safe_infinity = True`** (*`boolean`*) – As sometimes the distance sensor returns a small value when there's nothing in front of it, we need to poll again and again to confirm the presence of an obstacle. Setting `safe_infinity` to `False` will avoid that extra polling.

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

6.2 LightColorSensor

class `di_sensors.light_color_sensor.`**`LightColorSensor`**(*sensor_integration_time=0.0048*, *sensor_gain=2*, *led_state=False*, *bus='RPI_1SW'*)

Bases: `object`

Class for interfacing with the Light Color Sensor.

`__init__`(*sensor_integration_time=0.0048*, *sensor_gain=2*, *led_state=False*, *bus='RPI_1SW'*)
Constructor for initializing a link to the Light Color Sensor.

Parameters

- **`sensor_integration_time = 0.0048`** (*`float`*) – Time in seconds for each sample (aka the time needed to take a sample). Range is between 0.0024 and 0.6144 seconds. Use increments of 2.4 ms.
- **`sensor_gain = di_sensors.TCS34725.GAIN_16X`** (*`int`*) – The gain constant of the sensor. Valid values are `di_sensors.TCS34725.GAIN_1X`, `di_sensors.TCS34725.GAIN_4X`, `di_sensors.TCS34725.GAIN_16X` or `di_sensors.TCS34725.GAIN_60X`.
- **`led_state = False`** (*`bool`*) – The LED state. If it's set to `True`, then the LED will turn on, otherwise the LED will stay off. By default, the LED is turned on.

- **bus = "RPI_1SW"** (*str*) – The bus to which the distance sensor is connected to. By default, it's set to bus `"RPI_1SW"`. Check the *hardware specs* for more information about the ports.

Raises

- **OSError** – When the Light Color Sensor is not reachable.
- **RuntimeError** – When the chip ID is incorrect. This happens when we have a device pointing to the same address, but it's not a Light Color Sensor.

set_led(*value*, *delay=True*)
Set the LED state.

Parameters

- **value** (*bool*) – If set to `True`, then the LED turns on, otherwise it stays off.
- **delay = True** (*bool*) – When it's set to `True`, the LED turns on after *2 * time_to_take_sample* seconds have passed. This ensures that the next read following the LED change will be correct.

Raises **OSError** – When the Light Color Sensor is not reachable.

get_raw_colors(*delay=True*)
Read the sensor values.

Parameters **delay = True** (*bool*) – Delay for the time it takes to sample. If the delay is set to be added, then we are ensured to get fresh values on every call. Used in conjuction with the `set_led()` method.

Returns The RGBA values from the sensor. RGBA = Red, Green, Blue, Alpha (or Clear).

Return type (float,float,float,float) where the range of each element is between 0 and 1.

Raises **OSError** – If the Light Color Sensor can't be reached.

6.3 InertialMeasurementUnit

class di_sensors.inertial_measurement_unit.**InertialMeasurementUnit**(*bus='RPI_1SW'*)
Bases: `object`

Class for interfacing with the InertialMeasurementUnit Sensor.

BNO055.**get_calibration_status**()
Get calibration status of the InertialMeasurementUnit Sensor.

The moment the sensor is powered, this method should be called almost continuously until the sensor is fully calibrated. For calibrating the sensor faster, it's enough to hold the sensor for a couple of seconds on each "face" of an imaginary cube.

For each component of the system, there is a number that says how much the component has been calibrated:

- **System**, `3` = fully calibrated, `0` = not calibrated.
- **Gyroscope**, `3` = fully calibrated, `0` = not calibrated.
- **Accelerometer**, `3` = fully calibrated, `0` = not calibrated.
- **Magnetometer**, `3` = fully calibrated, `0` = not calibrated.

Returns A tuple where each member shows how much a component of the IMU is calibrated. See the above description of the method.

Return type (int,int,int,int)

Raises **`OSError`** – When the InertialMeasurementUnit Sensor is not reachable.

Important: The sensor needs a new calibration each time it's powered up.

`__init__`(*bus='RPI_1SW'*)

Constructor for initializing link with the InertialMeasurementUnit Sensor.

Parameters **`bus = "RPI_1SW"`** (*`str`*) – The bus to which the distance sensor is connected to. By default, it's set to bus `"RPI_1SW"`. Check the *hardware specs* for more information about the ports.

Raises

- **`RuntimeError`** – When the chip ID is incorrect. This happens when we have a device pointing to the same address, but it's not a InertialMeasurementUnit Sensor.
- **`OSError`** – When the InertialMeasurementUnit Sensor is not reachable.

`read_euler`()

Read the absolute orientation.

Returns Tuple of euler angles in degrees of *heading*, *roll* and *pitch*.

Return type (float,float,float)

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

`read_magnetometer`()

Read the magnetometer values.

Returns Tuple containing X, Y, Z values in *micro-Teslas* units. You can check the X, Y, Z axes on the sensor itself.

Return type (float,float,float)

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

`read_gyroscope`()

Read the angular velocity of the gyroscope.

Returns The angular velocity as a tuple of X, Y, Z values in *degrees/s*. You can check the X, Y, Z axes on the sensor itself.

Return type (float,float,float)

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

`read_accelerometer`()

Read the accelerometer.

Returns A tuple of X, Y, Z values in *meters/(second^2)* units. You can check the X, Y, Z axes on the sensor itself.

Return type (float,float,float)

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

read_linear_acceleration()
Read the linear acceleration - that is, the acceleration from movement and without the gravitational acceleration in it.

Returns The linear acceleration as a tuple of X, Y, Z values measured in *meters/(second^2)* units. You can check the X, Y, Z axes on the sensor itself.

Return type (float,float,float)

Raises `OSError` – When the sensor is not reachable.

read_gravity()
Read the gravitational acceleration.

Returns The gravitational acceleration as a tuple of X, Y, Z values in *meters/(second^2)* units. You can check the X, Y, Z axes on the sensor itself.

Return type (float,float,float)

Raises `OSError` – When the sensor is not reachable.

read_quaternion()
Read the quaternion values.

Returns The current orientation as a tuple of X, Y, Z, W quaternion values.

Return type (float,float,float,float)

Raises `OSError` – When the sensor is not present.

read_temperature()
Read the temperature in Celsius degrees.

Returns Temperature in Celsius degrees.

Return type int

Raises `OSError` – When the sensor can't be contacted.

6.4 TempHumPress

class di_sensors.temp_hum_press.**TempHumPress**(*bus='RPI_1SW'*)
Bases: `object`

Class for interfacing with the Temperature Humidity Pressure Sensor.

__init__(*bus='RPI_1SW'*)
Constructor for initializing link with the Temperature Humidity Pressure Sensor.

Parameters **bus = "RPI_1SW"** (*str*) – The bus to which the THP sensor is connected to. By default, it's set to bus `"RPI_1SW"`. Check the *hardware specs* for more information about the ports.

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.

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.

6.5 LineFollower

class di_sensors.line_follower.**LineFollower**(*bus='RPI_1SW'*)
Bases: `object`

Class for interfacing with the Line Follower Sensor (black board).

Important: This sensor is the replacement for the red one *LineFollowerRed*, which is getting retired, but we'll still support it. The improvements of this one over the red one are:

1. Much faster poll rate - **~130** times a second vs the red one at **~60Hz**.
2. More energy efficient - this one uses a **minimum** amount of power compared to the previous generation which tended to get hot to touch.
3. Sensors are much more **accurate and consistent** over the red ones.
4. **Reduced overhead** on the I2C line.

__init__(*bus='RPI_1SW'*)
Constructor for initializing an object to interface with the Line Follower Sensor (black board).

Parameters **bus = "RPI_1SW"** (*str*) – The bus to which the Line Follower Sensor (black board) is connected to. By default, it's set to `"RPI_1SW"`. Check the *hardware specs* for more information about the ports.

read_sensors()
Read the line follower's values.

Returns A 6-element tuple with line sensors 1 through 6 **(from left to right with the arrow pointing forward)** with values between **0** (black) and **1** (white).

Return type tuple

Raises **OSError** – When the Line Follower Sensor (black board) is not reachable.

get_manufacturer()
Read the manufacturer of the Line Follower Sensor (black board)'s.

Returns The name of the manufacturer.

Return type str

Raises **OSError** – When the Line Follower Sensor (black board) is not reachable.

get_board()
Read the board name of the Line Follower Sensor (black board).

Returns The name of the board.

Return type str

Raises **OSError** – When the Line Follower Sensor (black board) is not reachable.

get_version_firmware()
Get the firmware version currently residing on the Line Follower Sensor (black board).

Returns The version of the firmware.

Return type str

Raises **OSError** – When the Line Follower Sensor (black board) is not reachable.

class di_sensors.line_follower.**LineFollowerRed**(*bus='RPI_1SW'*)
Bases: `object`

Class for interfacing with the depreciated Line Follower Sensor (red board).

__init__(*bus='RPI_1SW'*)
Constructor for initializing an object to interface with the depreciated Line Follower Sensor (red board).

Parameters **bus = "RPI_1SW"** (*str*) – The bus to which the depreciated Line Follower Sensor (red board) is connected to. By default, it's set to `"RPI_1SW"`. Check the *hardware specs* for more information about the ports.

read_sensors()
Read the line follower's values.

Returns A 5-element tuple with the 1st element starting from the left of the sensor going to the right of it **(check the markings on the sensor)** with values between **0** (for black) and **1** (for white).

Return type tuple

Raises **OSError** – When the depreciated Line Follower Sensor (red board) is not reachable.

6.6 More . . .

If you wish to have a more granular control over the sensors' functionalities, then you should check the following submodules of the `DI-Sensors` package:

- di_sensors.BME280 - submodule for interfacing with the Temperature Humidity Pressure Sensor.
- di_sensors.BNO055 - submodule for interfacing with the InertialMeasurementUnit Sensor.
- di_sensors.TCS34725 - submodule for interfacing with the Light Color Sensor.
- di_sensors.VL53L0X - submodule for interfacing with the Distance Sensor.

All these submodules that are being referenced in this section were used for creating the *DistanceSensor*, *LightColorSensor*, *TempHumPress* and the *InertialMeasurementUnit* classes.

CHAPTER

SEVEN

DEVELOPER'S GUIDE

7.1 Our collaborators

The following collaborators are ordered alphabetically:

- John Cole - Github Account.
- Karan Nayan - Github Account.
- Matt Richardson - Github Account.
- Nicole Parrot - Github Account.
- Robert Lucian Chiriac - Github Account.
- Shoban Narayan - Github account.

CHAPTER EIGHT

FREQUENTLY ASKED QUESTIONS

For more questions, please head over to our Dexter Industries forum.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX