Pybricks Modules and Examples

Version v3.2.0

Dec 20, 2022

TABLE OF CONTENTS

Pybricks is Python coding for smart LEGO® hubs. Run MicroPython scripts directly on the hub, and get full control of your motors and sensors.

Pybricks runs on LEGO® BOOST, City, Technic, MINDSTORMS®, and SPIKE®. You can code using Windows, Mac, Linux, Chromebook, and Android.

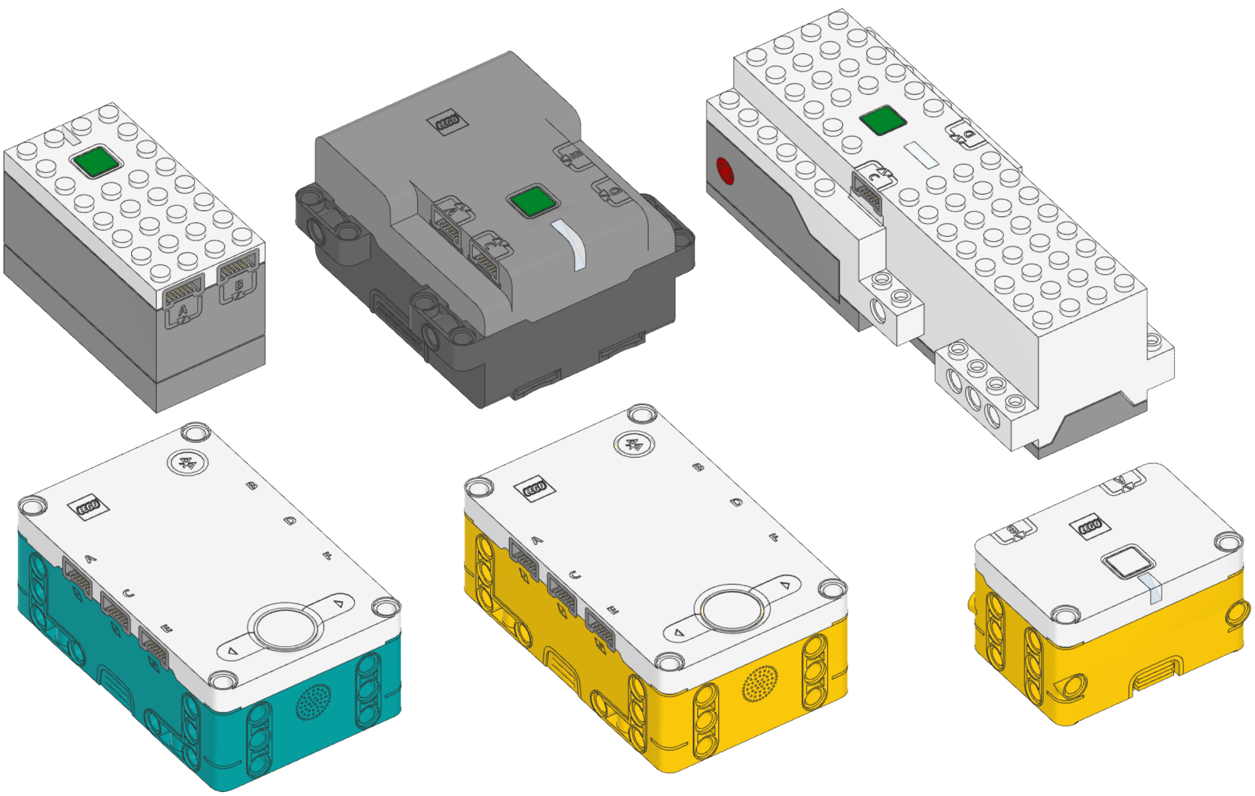
Click on any device below to see its documentation. Use the menu on the left to find documentation for additional modules. You may need to click the icon above to reveal this menu.

Note: You are viewing the stand-alone version of the documentation. To learn more about Pybricks and to start coding, visit the Pybricks website

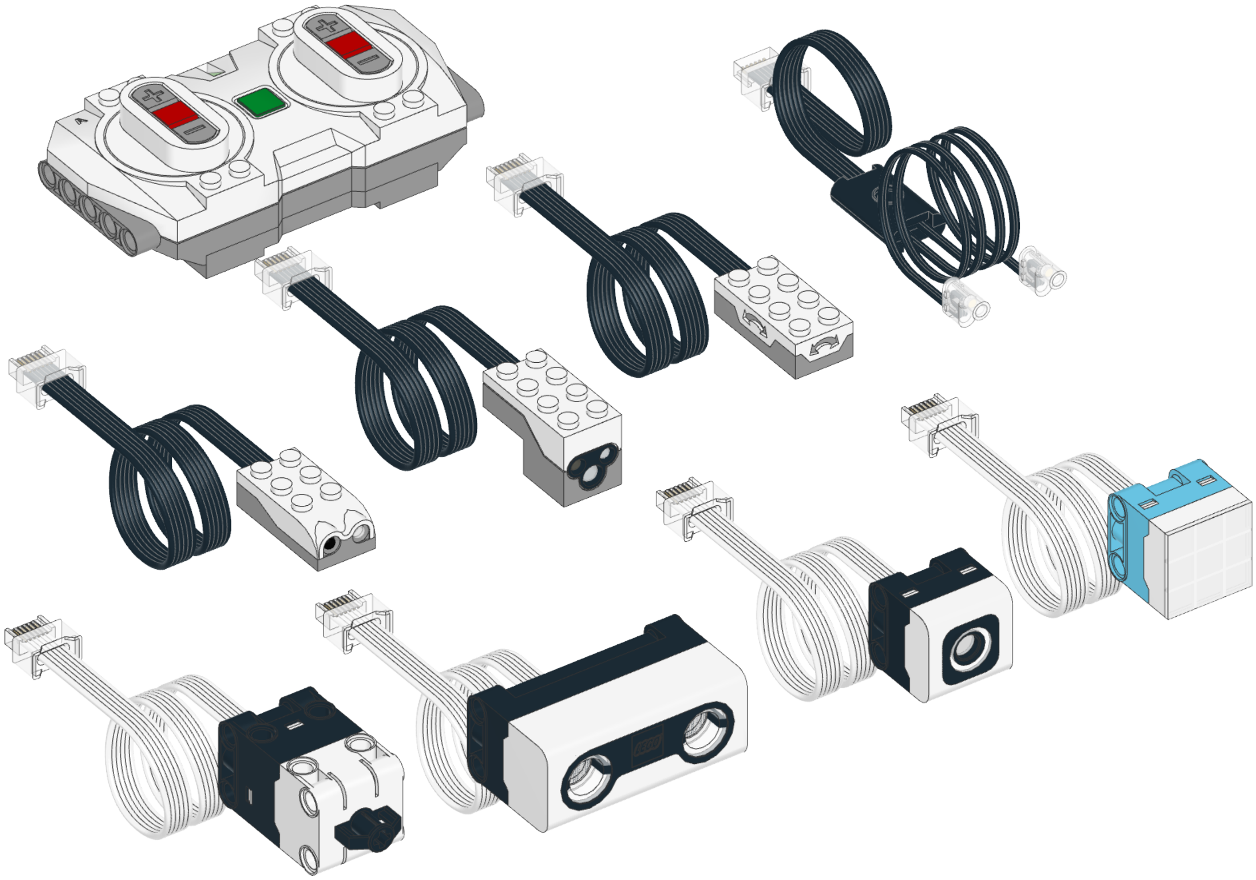
---

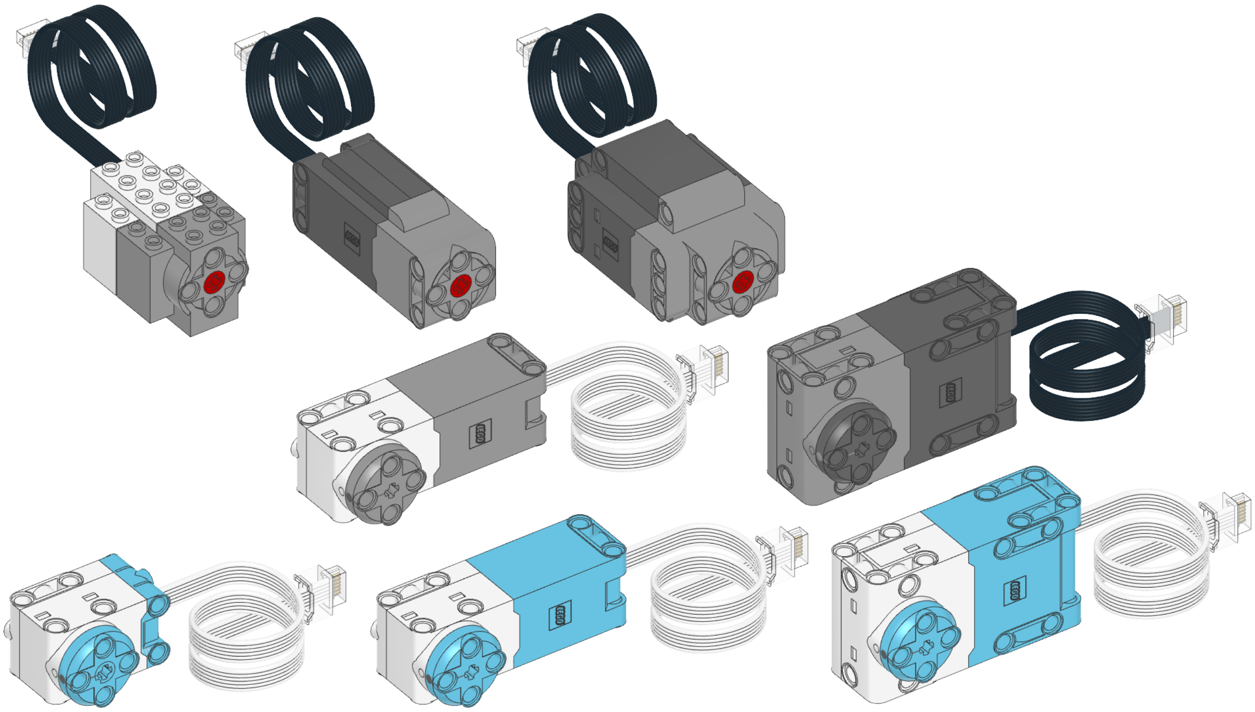
Note: Are you using LEGO MINDSTORMS EV3? Check out the EV3 documentation instead.

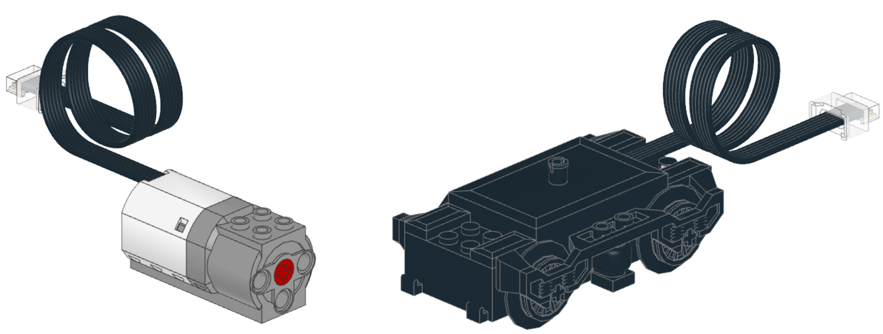
Programmable hubs



Powered Up motors and sensors





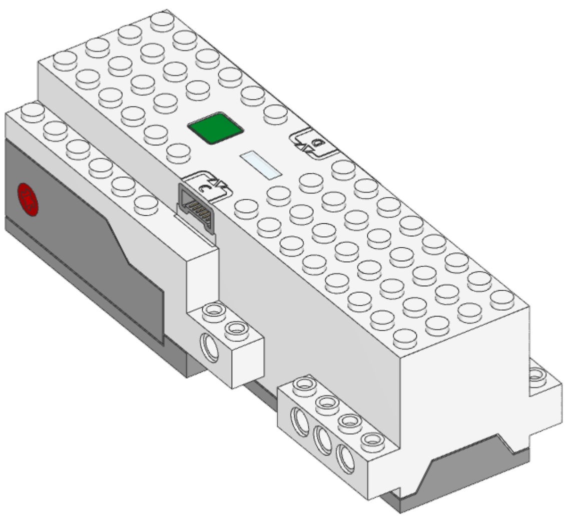


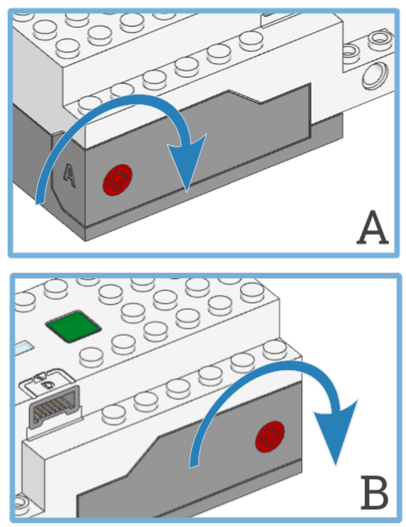
CHAPTER

ONE

HUBS – BUILT-IN HUB FUNCTIONS

1.1 Move Hub





class MoveHub

LEGO® BOOST Move Hub.

Using the hub status light

`light.on(color)`

Turns on the light at the specified color.

Parameters
color (Color) – Color of the light.

`light.off()`

Turns off the light.

`light.blink(color, durations)`

Blinks the light at a given color by turning it on and off for given durations.

The light keeps blinking indefinitely while the rest of your program keeps running.

This method provides a simple way to make basic but useful patterns. For more generic and multi-color patterns, use `animate()` instead.

Parameters

- **color** (Color) – Color of the light.
- **durations** (list) – Sequence of time values of the form `[on_1, off_1, on_2, off_2, ...]`.

`light.animate(colors, interval)`

Animates the light with a sequence of colors, shown one by one for the given interval.

The animation runs in the background while the rest of your program keeps running. When the animation completes, it repeats.

Parameters

- **colors** (list) – Sequence of *Color* values.
- **interval** (Number, *ms*) – Time between color updates.

Using the IMU

`imu.up()` → *Side*

Checks which side of the hub currently faces upward.

Returns
`Side.TOP`, `Side.BOTTOM`, `Side.LEFT`, `Side.RIGHT`, `Side.FRONT` or `Side.BACK`.

`imu.acceleration()` → Tuple[int, int, int]: mm/s^2

Gets the acceleration of the device.

Returns
Acceleration along all three axes.

Changed in version 3.2: Changed acceleration units from m/s^2 to mm/s^2.

Using the battery

`battery.voltage()` → int: mV

Gets the voltage of the battery.

Returns
Battery voltage.

`battery.current()` → int: mA

Gets the current supplied by the battery.

Returns
Battery current.

Button and system control

`button.pressed()` → Collection[*Button*]

Checks which buttons are currently pressed.

Returns
Set of pressed buttons.

`system.set_stop_button(`*button*`)`

Sets the button or button combination that stops a running script.

Normally, the center button is used to stop a running script. You can change or disable this behavior in order to use the button for other purposes.

Parameters
button (Button) – A button such as *Button.CENTER*, or a tuple of multiple buttons. Choose `None` to disable the stop button altogether.

`system.name()` → *str*

Gets the hub name. This is the name you see when connecting via Bluetooth.

Returns
The hub name.

`system.storage(`*self*, *offset*, *write=*`)`

`system.storage(`*self*, *offset*, *read=*`)` → *bytes*

Reads or writes binary data to persistent storage.

This lets you store data that can be used the next time you run the program.

The data will be saved to flash memory when you turn the hub off normally. It will not be saved if the batteries are removed *while* the hub is still running.

Once saved, the data will remain available even after you remove the batteries.

Parameters

- **offset** (int) – The offset from the start of the user storage memory, in bytes.
- **read** (int) – The number of bytes to read. Omit this argument when writing.
- **write** (bytes) – The bytes to write. Omit this argument when reading.

Returns
The bytes read if reading, otherwise `None`.

Raises
ValueError – If you try to read or write data outside of the allowed range.

You can store up to 128 bytes of data on this hub. The data is cleared when you update the Pybricks firmware or if you restore the original firmware.

`system.shutdown()`

Stops your program and shuts the hub down.

`system.reset_reason()` → *int*

Finds out how and why the hub (re)booted. This can be useful to diagnose some problems.

Returns

- `0` if the hub was previously powered off normally.
- `1` if the hub rebooted automatically, like after a firmware update.

- 2 if the hub previously crashed due to a watchdog timeout, which indicates a firmware issue.

1.1.1 Status light examples

Turning the light on and off

```
from pybricks.hubs import MoveHub
from pybricks.parameters import Color
from pybricks.tools import wait

# Initialize the hub.
hub = MoveHub()

# Turn the light on and off 5 times.
for i in range(5):

    hub.light.on(Color.RED)
    wait(1000)

    hub.light.off()
    wait(500)
```

Making the light blink

```
from pybricks.hubs import MoveHub
from pybricks.parameters import Color
from pybricks.tools import wait

# Initialize the hub
hub = MoveHub()

# Keep blinking red on and off.
hub.light.blink(Color.RED, [500, 500])

wait(10000)

# Keep blinking green slowly and then quickly.
hub.light.blink(Color.GREEN, [500, 500, 50, 900])

wait(10000)
```

1.1.2 IMU examples

Testing which way is up

```
from pybricks.hubs import MoveHub
from pybricks.parameters import Color, Side
from pybricks.tools import wait

# Initialize the hub.
hub = MoveHub()

# Define colors for each side in a dictionary.
SIDE_COLORS = {
    Side.TOP: Color.RED,
    Side.BOTTOM: Color.BLUE,
    Side.LEFT: Color.GREEN,
    Side.RIGHT: Color.YELLOW,
    Side.FRONT: Color.MAGENTA,
    Side.BACK: Color.BLACK,
}

# Keep updating the color based on detected up side.
while True:

    # Check which side of the hub is up.
    up_side = hub.imu.up()

    # Change the color based on the side.
    hub.light.on(SIDE_COLORS[up_side])

    # Also print the result.
    print(up_side)
    wait(50)
```

Reading acceleration

```
from pybricks.hubs import MoveHub
from pybricks.tools import wait

# Initialize the hub.
hub = MoveHub()

# Get the acceleration tuple.
print(hub.imu.acceleration())

while True:
    # Get individual acceleration values.
    x, y, z = hub.imu.acceleration()
    print(x, y, z)

    # Wait so we can see what we printed.
    wait(100)
```

1.1.3 Button and system examples

Using the stop button during your program

```
from pybricks.hubs import MoveHub
from pybricks.parameters import Color, Button
from pybricks.tools import wait, StopWatch

# Initialize the hub.
hub = MoveHub()

# Disable the stop button.
hub.system.set_stop_button(None)

# Check the button for 5 seconds.
watch = StopWatch()
while watch.time() < 5000:

    # Set light to green if pressed, else red.
    if hub.button.pressed():
        hub.light.on(Color.GREEN)
    else:
        hub.light.on(Color.RED)

# Enable the stop button again.
hub.system.set_stop_button(Button.CENTER)

# Now you can press the stop button as usual.
wait(5000)
```

Turning the hub off

```
from pybricks.hubs import MoveHub
from pybricks.tools import wait

# Initialize the hub.
hub = MoveHub()

# Say goodbye and give some time to send it.
print("Goodbye!")
wait(100)

# Shut the hub down.
hub.system.shutdown()
```

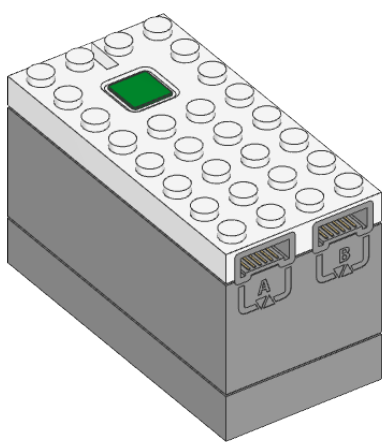
Making random numbers

The Move Hub does not include the *urandom* module. If you need random numbers in your application, you can try a variation of the following example.

To make it work better, change the initial value of `_rand` to something that is truly random in your application. You could use the IMU acceleration or a sensor value, for example.

```
from pybricks.hubs import MoveHub

# Initialize the hub.
hub = MoveHub()

# Initialize "random" seed.
_rand = hub.battery.voltage() + hub.battery.current()


# Return a random integer N such that a <= N <= b.
def randint(a, b):
    global _rand
    _rand = 75 * _rand % 65537  # Lehmer
    return _rand * (b - a + 1) // 65537 + a


# Generate a few example numbers.
for i in range(5):
    print(randint(0, 1000))
```

1.2 City Hub



class CityHub

LEGO® City Hub.

Using the hub status light

`light.on(color)`

Turns on the light at the specified color.

Parameters
color (Color) – Color of the light.

`light.off()`

Turns off the light.

`light.blink(color, durations)`

Blinks the light at a given color by turning it on and off for given durations.

The light keeps blinking indefinitely while the rest of your program keeps running.

This method provides a simple way to make basic but useful patterns. For more generic and multi-color patterns, use `animate()` instead.

Parameters

- **color** (Color) – Color of the light.
- **durations** (list) – Sequence of time values of the form `[on_1, off_1, on_2, off_2, ...]`.

`light.animate(colors, interval)`

Animates the light with a sequence of colors, shown one by one for the given interval.

The animation runs in the background while the rest of your program keeps running. When the animation completes, it repeats.

Parameters

- **colors** (list) – Sequence of *Color* values.
- **interval** (Number, *ms*) – Time between color updates.

Using the battery

`battery.voltage()` → int: mV

Gets the voltage of the battery.

Returns
Battery voltage.

`battery.current()` → int: mA

Gets the current supplied by the battery.

Returns
Battery current.

Button and system control

`button.pressed()` → Collection[*Button*]

Checks which buttons are currently pressed.

Returns
Set of pressed buttons.

`system.set_stop_button(button)`

Sets the button or button combination that stops a running script.

Normally, the center button is used to stop a running script. You can change or disable this behavior in order to use the button for other purposes.

Parameters
button (Button) – A button such as *Button.CENTER*, or a tuple of multiple buttons. Choose None to disable the stop button altogether.

`system.name()` → *str*

Gets the hub name. This is the name you see when connecting via Bluetooth.

Returns
The hub name.

`system.storage(self, offset, write=)`

`system.storage(self, offset, read=)` → *bytes*

Reads or writes binary data to persistent storage.

This lets you store data that can be used the next time you run the program.

The data will be saved to flash memory when you turn the hub off normally. It will not be saved if the batteries are removed *while* the hub is still running.

Once saved, the data will remain available even after you remove the batteries.

Parameters

- **offset** (int) – The offset from the start of the user storage memory, in bytes.
- **read** (int) – The number of bytes to read. Omit this argument when writing.
- **write** (bytes) – The bytes to write. Omit this argument when reading.

Returns
The bytes read if reading, otherwise None.

Raises
ValueError – If you try to read or write data outside of the allowed range.

You can store up to 128 bytes of data on this hub. The data is cleared when you update the Pybricks firmware or if you restore the original firmware.

`system.shutdown()`

Stops your program and shuts the hub down.

`system.reset_reason()` → *int*

Finds out how and why the hub (re)booted. This can be useful to diagnose some problems.

Returns

- 0 if the hub was previously powered off normally.
- 1 if the hub rebooted automatically, like after a firmware update.

- 2 if the hub previously crashed due to a watchdog timeout, which indicates a firmware issue.

1.2.1 Status light examples

Turning the light on and off

```
from pybricks.hubs import CityHub
from pybricks.parameters import Color
from pybricks.tools import wait

# Initialize the hub.
hub = CityHub()

# Turn the light on and off 5 times.
for i in range(5):

    hub.light.on(Color.RED)
    wait(1000)

    hub.light.off()
    wait(500)
```

Changing brightness and using custom colors

```
from pybricks.hubs import CityHub
from pybricks.parameters import Color
from pybricks.tools import wait

# Initialize the hub.
hub = CityHub()

# Show the color at 30% brightness.
hub.light.on(Color.RED * 0.3)

wait(2000)

# Use your own custom color.
hub.light.on(Color(h=30, s=100, v=50))

wait(2000)

# Go through all the colors.
for hue in range(360):
    hub.light.on(Color(hue))
    wait(10)
```

Making the light blink

```
from pybricks.hubs import CityHub
from pybricks.parameters import Color
from pybricks.tools import wait

# Initialize the hub
hub = CityHub()

# Keep blinking red on and off.
hub.light.blink(Color.RED, [500, 500])

wait(10000)

# Keep blinking green slowly and then quickly.
hub.light.blink(Color.GREEN, [500, 500, 50, 900])

wait(10000)
```

Creating light animations

```
from pybricks.hubs import CityHub
from pybricks.parameters import Color
from pybricks.tools import wait
from umath import sin, pi

# Initialize the hub.
hub = CityHub()

# Make an animation with multiple colors.
hub.light.animate([Color.RED, Color.GREEN, Color.NONE], interval=500)

wait(10000)

# Make the color RED grow faint and bright using a sine pattern.
hub.light.animate([Color.RED * (0.5 * sin(i / 15 * pi) + 0.5) for i in range(30)], 40)

wait(10000)

# Cycle through a rainbow of colors.
hub.light.animate([Color(h=i * 8) for i in range(45)], interval=40)

wait(10000)
```

1.2.2 Button and system examples

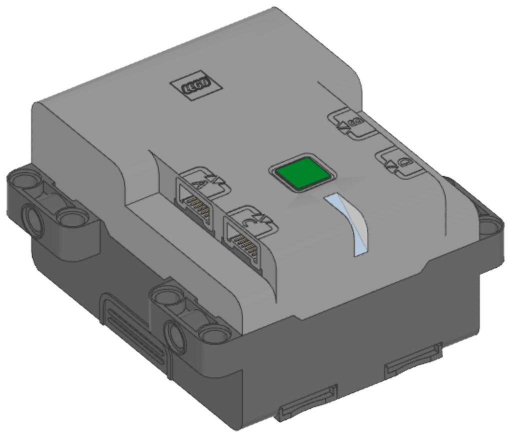
Using the stop button during your program

```
from pybricks.hubs import CityHub
from pybricks.parameters import Color, Button
from pybricks.tools import wait, StopWatch

# Initialize the hub.
hub = CityHub()

# Disable the stop button.
hub.system.set_stop_button(None)

# Check the button for 5 seconds.
watch = StopWatch()
while watch.time() < 5000:

    # Set light to green if pressed, else red.
    if hub.button.pressed():
        hub.light.on(Color.GREEN)
    else:
        hub.light.on(Color.RED)

# Enable the stop button again.
hub.system.set_stop_button(Button.CENTER)

# Now you can press the stop button as usual.
wait(5000)
```

Turning the hub off

```
from pybricks.hubs import CityHub
from pybricks.tools import wait

# Initialize the hub.
hub = CityHub()

# Say goodbye and give some time to send it.
print("Goodbye!")
wait(100)

# Shut the hub down.
hub.system.shutdown()
```

1.3 Technic Hub



class TechnicHub(*top_side=Axis.Z*, *front_side=Axis.X*)

LEGO® Technic Hub.

Initializes the hub. Optionally, specify how the hub is *placed in your design* by saying in which direction the top side (with the button) and front side (with the light) are pointing.

Parameters

- **top_side** (Axis) – The axis that passes through the *top side* of the hub.
- **front_side** (Axis) – The axis that passes through the *front side* of the hub.

Using the hub status light

light.**on**(*color*)

Turns on the light at the specified color.

Parameters

color (Color) – Color of the light.

light.**off**()

Turns off the light.

light.**blink**(*color*, *durations*)

Blinks the light at a given color by turning it on and off for given durations.

The light keeps blinking indefinitely while the rest of your program keeps running.

This method provides a simple way to make basic but useful patterns. For more generic and multi-color patterns, use `animate()` instead.

Parameters

- **color** (Color) – Color of the light.
- **durations** (list) – Sequence of time values of the form `[on_1, off_1, on_2, off_2, ...]`.

`light.animate(colors, interval)`

Animates the light with a sequence of colors, shown one by one for the given interval.

The animation runs in the background while the rest of your program keeps running. When the animation completes, it repeats.

Parameters

- **colors** (`list`) – Sequence of *Color* values.
- **interval** (`Number, ms`) – Time between color updates.

Using the IMU

`imu.up()` → *Side*

Checks which side of the hub currently faces upward.

Returns
`Side.TOP`, `Side.BOTTOM`, `Side.LEFT`, `Side.RIGHT`, `Side.FRONT` or `Side.BACK`.

`imu.tilt()` → Tuple[*int*, *int*]

Gets the pitch and roll angles. This is relative to the *user-specified neutral orientation*.

The order of rotation is pitch-then-roll. This is equivalent to a positive rotation along the robot y-axis and then a positive rotation along the x-axis.

Returns
Tuple of pitch and roll angles.

`imu.acceleration(axis)` → float: mm/s^2

`imu.acceleration()` → vector: mm/s^2

Gets the acceleration of the device along a given axis in the *robot reference frame*.

Parameters
axis (`Axis`) – Axis along which the acceleration should be measured.

Returns
Acceleration along the specified axis. If you specify no axis, this returns a vector of accelerations along all axes.

`imu.angular_velocity(axis)` → float: deg/s

`imu.angular_velocity()` → vector: deg/s

Gets the angular velocity of the device along a given axis in the *robot reference frame*.

Parameters
axis (`Axis`) – Axis along which the angular velocity should be measured.

Returns
Angular velocity along the specified axis. If you specify no axis, this returns a vector of accelerations along all axes.

`imu.heading()` → float: deg

Gets the heading angle relative to the starting orientation. It is a positive rotation around the *z-axis in the robot frame*, prior to applying any tilt rotation.

For a vehicle viewed from the top, this means that a positive heading value corresponds to a counterclockwise rotation.

Note: This method is not yet implemented.

Returns
Heading angle relative to starting orientation.

`imu.reset_heading(angle)`

Resets the accumulated heading angle of the robot.

Note: This method is not yet implemented.

Parameters
angle (Number, *deg*) – Value to which the heading should be reset.

Using the battery

`battery.voltage()` → int: mV

Gets the voltage of the battery.

Returns
Battery voltage.

`battery.current()` → int: mA

Gets the current supplied by the battery.

Returns
Battery current.

Button and system control

`button.pressed()` → Collection[*Button*]

Checks which buttons are currently pressed.

Returns
Set of pressed buttons.

`system.set_stop_button(button)`

Sets the button or button combination that stops a running script.

Normally, the center button is used to stop a running script. You can change or disable this behavior in order to use the button for other purposes.

Parameters
button (Button) – A button such as `Button.CENTER`, or a tuple of multiple buttons. Choose `None` to disable the stop button altogether.

`system.name()` → *str*

Gets the hub name. This is the name you see when connecting via Bluetooth.

Returns
The hub name.

`system.storage(self, offset, write=)`

`system.storage(self, offset, read=)` → *bytes*

Reads or writes binary data to persistent storage.

This lets you store data that can be used the next time you run the program.

The data will be saved to flash memory when you turn the hub off normally. It will not be saved if the batteries are removed *while* the hub is still running.

Once saved, the data will remain available even after you remove the batteries.

Parameters

- **offset** (int) – The offset from the start of the user storage memory, in bytes.
- **read** (int) – The number of bytes to read. Omit this argument when writing.
- **write** (bytes) – The bytes to write. Omit this argument when reading.

Returns

The bytes read if reading, otherwise `None`.

Raises

ValueError – If you try to read or write data outside of the allowed range.

You can store up to 128 bytes of data on this hub. The data is cleared when you update the Pybricks firmware or if you restore the original firmware.

`system.shutdown()`

Stops your program and shuts the hub down.

`system.reset_reason()` → *int*

Finds out how and why the hub (re)booted. This can be useful to diagnose some problems.

Returns

- `0` if the hub was previously powered off normally.
- `1` if the hub rebooted automatically, like after a firmware update.
- `2` if the hub previously crashed due to a watchdog timeout, which indicates a firmware issue.

1.3.1 Status light examples

Turning the light on and off

```
from pybricks.hubs import TechnicHub
from pybricks.parameters import Color
from pybricks.tools import wait

# Initialize the hub.
hub = TechnicHub()

# Turn the light on and off 5 times.
for i in range(5):

    hub.light.on(Color.RED)
    wait(1000)
```

(continues on next page)

(continued from previous page)

```
    hub.light.off()
    wait(500)
```

Changing brightness and using custom colors

```
from pybricks.hubs import TechnicHub
from pybricks.parameters import Color
from pybricks.tools import wait

# Initialize the hub.
hub = TechnicHub()

# Show the color at 30% brightness.
hub.light.on(Color.RED * 0.3)

wait(2000)

# Use your own custom color.
hub.light.on(Color(h=30, s=100, v=50))

wait(2000)

# Go through all the colors.
for hue in range(360):
    hub.light.on(Color(hue))
    wait(10)
```

Making the light blink

```
from pybricks.hubs import TechnicHub
from pybricks.parameters import Color
from pybricks.tools import wait

# Initialize the hub
hub = TechnicHub()

# Keep blinking red on and off.
hub.light.blink(Color.RED, [500, 500])

wait(10000)

# Keep blinking green slowly and then quickly.
hub.light.blink(Color.GREEN, [500, 500, 50, 900])

wait(10000)
```

Creating light animations

```
from pybricks.hubs import TechnicHub
from pybricks.parameters import Color
from pybricks.tools import wait
from umath import sin, pi

# Initialize the hub.
hub = TechnicHub()

# Make an animation with multiple colors.
hub.light.animate([Color.RED, Color.GREEN, Color.NONE], interval=500)

wait(10000)

# Make the color RED grow faint and bright using a sine pattern.
hub.light.animate([Color.RED * (0.5 * sin(i / 15 * pi) + 0.5) for i in range(30)], 40)

wait(10000)

# Cycle through a rainbow of colors.
hub.light.animate([Color(h=i * 8) for i in range(45)], interval=40)

wait(10000)
```

1.3.2 IMU examples

Testing which way is up

```
from pybricks.hubs import TechnicHub
from pybricks.parameters import Color, Side
from pybricks.tools import wait

# Initialize the hub.
hub = TechnicHub()

# Define colors for each side in a dictionary.
SIDE_COLORS = {
    Side.TOP: Color.RED,
    Side.BOTTOM: Color.BLUE,
    Side.LEFT: Color.GREEN,
    Side.RIGHT: Color.YELLOW,
    Side.FRONT: Color.MAGENTA,
    Side.BACK: Color.BLACK,
}

# Keep updating the color based on detected up side.
while True:

    # Check which side of the hub is up.
    up_side = hub.imu.up()
```

(continues on next page)

(continued from previous page)

```
    # Change the color based on the side.
    hub.light.on(SIDE_COLORS[up_side])

    # Also print the result.
    print(up_side)
    wait(50)
```

Reading the tilt value

```
from pybricks.hubs import TechnicHub
from pybricks.tools import wait

# Initialize the hub.
hub = TechnicHub()

while True:
    # Read the tilt values.
    pitch, roll = hub.imu.tilt()

    # Print the result.
    print(pitch, roll)
    wait(200)
```

Using a custom hub orientation

```
from pybricks.hubs import TechnicHub
from pybricks.tools import wait
from pybricks.geometry import Axis

# Initialize the hub. In this case, specify that the hub is mounted with the
# top side facing forward and the front side facing to the right.
# For example, this is how the hub is mounted in BLAST in the 51515 set.
hub = TechnicHub(top_side=Axis.X, front_side=-Axis.Y)

while True:
    # Read the tilt values. Now, the values are 0 when BLAST stands upright.
    # Leaning forward gives positive pitch. Leaning right gives positive roll.
    pitch, roll = hub.imu.tilt()

    # Print the result.
    print(pitch, roll)
    wait(200)
```

Reading acceleration and angular velocity vectors

```
from pybricks.hubs import TechnicHub
from pybricks.tools import wait

# Initialize the hub.
hub = TechnicHub()

# Get the acceleration vector in g's.
print(hub.imu.acceleration() / 9810)

# Get the angular velocity vector.
print(hub.imu.angular_velocity())

# Wait so we can see what we printed
wait(5000)
```

Reading acceleration and angular velocity on one axis

```
from pybricks.hubs import TechnicHub
from pybricks.tools import wait
from pybricks.geometry import Axis

# Initialize the hub.
hub = TechnicHub()

# Get the acceleration or angular_velocity along a single axis.
# If you need only one value, this is more memory efficient.
while True:

    # Read the forward acceleration.
    forward_acceleration = hub.imu.acceleration(Axis.X)

    # Read the yaw rate.
    yaw_rate = hub.imu.angular_velocity(Axis.Z)

    # Print the yaw rate.
    print(yaw_rate)
    wait(100)
```

1.3.3 Button and system examples

Using the stop button during your program

```
from pybricks.hubs import TechnicHub
from pybricks.parameters import Color, Button
from pybricks.tools import wait, StopWatch

# Initialize the hub.
```

(continues on next page)

(continued from previous page)

```
hub = TechnicHub()

# Disable the stop button.
hub.system.set_stop_button(None)

# Check the button for 5 seconds.
watch = StopWatch()
while watch.time() < 5000:

    # Set light to green if pressed, else red.
    if hub.button.pressed():
        hub.light.on(Color.GREEN)
    else:
        hub.light.on(Color.RED)

# Enable the stop button again.
hub.system.set_stop_button(Button.CENTER)

# Now you can press the stop button as usual.
wait(5000)
```

Turning the hub off

```
from pybricks.hubs import TechnicHub
from pybricks.tools import wait

# Initialize the hub.
hub = TechnicHub()

# Say goodbye and give some time to send it.
print("Goodbye!")
wait(100)

# Shut the hub down.
hub.system.shutdown()
```

1.4 Prime Hub / Inventor Hub

class InventorHub

This class is the same as the `PrimeHub` class, shown below. Both classes work on both hubs.

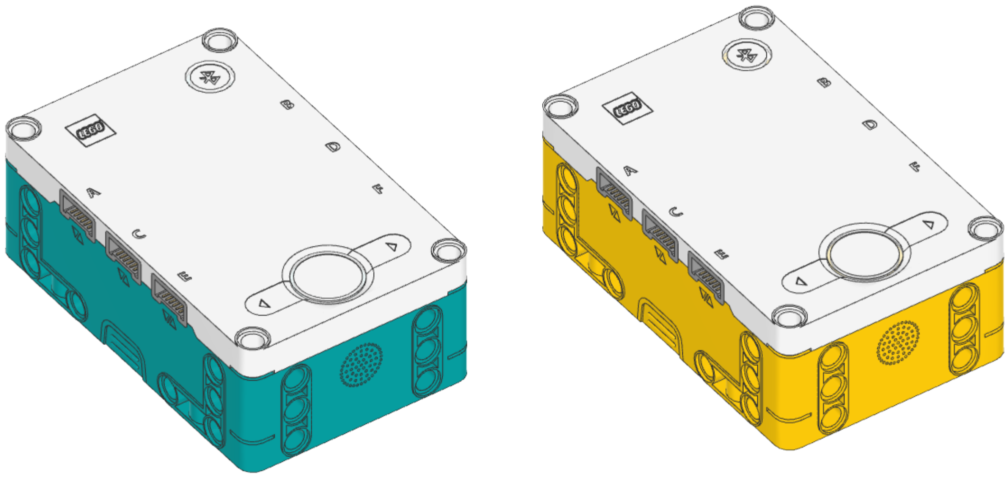
These hubs are completely identical. They use the same Pybricks firmware.

class PrimeHub(*top_side=Axis.Z*, *front_side=Axis.X*)

LEGO® SPIKE Prime Hub.

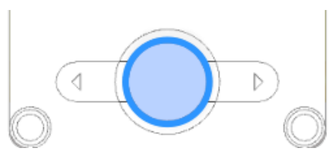
Initializes the hub. Optionally, specify how the hub is *placed in your design* by saying in which direction the top side (with the buttons) and front side (with the USB port) are pointing.

Parameters



- **top_side** (`Axis`) – The axis that passes through the *top side* of the hub.
- **front_side** (`Axis`) – The axis that passes through the *front side* of the hub.

Using the hub status light



`light.on(`*color*`)`

Turns on the light at the specified color.

Parameters
color (`Color`) – Color of the light.

`light.off()`

Turns off the light.

`light.blink(`*color*, *durations*`)`

Blinks the light at a given color by turning it on and off for given durations.

The light keeps blinking indefinitely while the rest of your program keeps running.

This method provides a simple way to make basic but useful patterns. For more generic and multi-color patterns, use `animate()` instead.

Parameters

- **color** (`Color`) – Color of the light.
- **durations** (`list`) – Sequence of time values of the form `[on_1, off_1, on_2, off_2, ...]`.

`light.animate(`*colors*, *interval*`)`

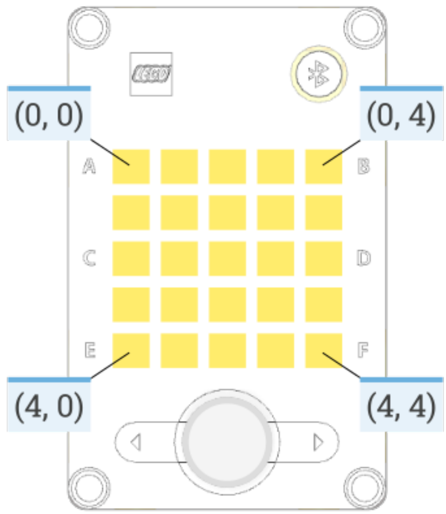
Animates the light with a sequence of colors, shown one by one for the given interval.

The animation runs in the background while the rest of your program keeps running. When the animation completes, it repeats.

Parameters

- **colors** (list) – Sequence of *Color* values.
- **interval** (Number, *ms*) – Time between color updates.

Using the light matrix display



`display.orientation(up)`

Sets the orientation of the light matrix display.

Only new displayed images and pixels are affected. The existing display contents remain unchanged.

Parameters
top (Side) – Which side of the light matrix display is "up" in your design. Choose `Side.TOP`, `Side.LEFT`, `Side.RIGHT`, or `Side.BOTTOM`.

`display.off()`

Turns off all the pixels.

`display.pixel(row, column, brightness=100)`

Turns on one pixel at the specified brightness.

Parameters

- **row** (Number) – Vertical grid index, starting at 0 from the top.
- **column** (Number) – Horizontal grid index, starting at 0 from the left.
- **brightness** (Number *brightness: %*) – Brightness of the pixel.

`display.icon(icon)`

Displays an icon, represented by a matrix of *brightness: %* values.

Parameters
icon (Matrix) – Matrix of intensities (*brightness: %*). A 2D list is also accepted.

`display.animate(matrices, interval)`

Displays an animation made using a list of images.

Each image has the same format as above. Each image is shown for the given interval. The animation repeats forever while the rest of your program keeps running.

Parameters

- **matrices** (*iter*) – Sequence of *Matrix* of intensities.
- **interval** (Number, *ms*) – Time to display each image in the list.

`display.`**`number`**`(`*number*`)`

Displays a number in the range -99 to 99.

A minus sign (-) is shown as a faint dot in the center of the display. Numbers greater than 99 are shown as >. Numbers less than -99 are shown as <.

Parameters

number (int) – The number to be displayed.

`display.`**`char`**`(`*char*`)`

Displays a character or symbol on the light grid. This may be any letter (a–z), capital letter (A–Z) or one of the following symbols: `!"#$%&'()*+,-./:;<=>?@[\]^_`{|}`.

Parameters

character (str) – The character or symbol to be displayed.

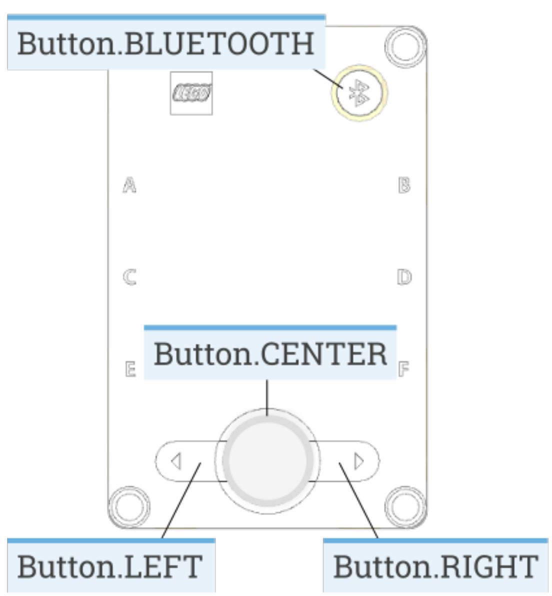
`display.`**`text`**`(`*text*, *on=500*, *off=50*`)`

Displays a text string, one character at a time, with a pause between each character. After the last character is shown, all lights turn off.

Parameters

- **text** (str) – The text to be displayed.
- **on** (Number, *ms*) – For how long a character is shown.
- **off** (Number, *ms*) – For how long the display is off between characters.

Using the buttons



`buttons.`**`pressed`**`()` → Collection[*Button*]

Checks which buttons are currently pressed.

Returns
Set of pressed buttons.

Using the IMU

`imu.up()` → *Side*

Checks which side of the hub currently faces upward.

Returns
`Side.TOP`, `Side.BOTTOM`, `Side.LEFT`, `Side.RIGHT`, `Side.FRONT` or `Side.BACK`.

`imu.tilt()` → Tuple[*int*, *int*]

Gets the pitch and roll angles. This is relative to the *user-specified neutral orientation*.

The order of rotation is pitch-then-roll. This is equivalent to a positive rotation along the robot y-axis and then a positive rotation along the x-axis.

Returns
Tuple of pitch and roll angles.

`imu.acceleration(`*axis*`)` → float: mm/s^2

`imu.acceleration()` → vector: mm/s^2

Gets the acceleration of the device along a given axis in the *robot reference frame*.

Parameters
axis (`Axis`) – Axis along which the acceleration should be measured.

Returns
Acceleration along the specified axis. If you specify no axis, this returns a vector of accelerations along all axes.

`imu.angular_velocity(`*axis*`)` → float: deg/s

`imu.angular_velocity()` → vector: deg/s

Gets the angular velocity of the device along a given axis in the *robot reference frame*.

Parameters
axis (`Axis`) – Axis along which the angular velocity should be measured.

Returns
Angular velocity along the specified axis. If you specify no axis, this returns a vector of accelerations along all axes.

`imu.heading()` → float: deg

Gets the heading angle relative to the starting orientation. It is a positive rotation around the *z-axis in the robot frame*, prior to applying any tilt rotation.

For a vehicle viewed from the top, this means that a positive heading value corresponds to a counterclockwise rotation.

Note: This method is not yet implemented.

Returns
Heading angle relative to starting orientation.

`imu.reset_heading(angle)`

Resets the accumulated heading angle of the robot.

Note: This method is not yet implemented.

Parameters
angle (Number, *deg*) – Value to which the heading should be reset.

Using the speaker

`speaker.volume(volume)`

`speaker.volume()` → int: %

Gets or sets the speaker volume.

If no volume is given, this method returns the current volume.

Parameters
volume (Number, *%*) – Volume of the speaker in the 0-100 range.

`speaker.beep(frequency=500, duration=100)`

Play a beep/tone.

Parameters

- **frequency** (Number, *Hz*) – Frequency of the beep in the 64-24000 Hz range.
- **duration** (Number, *ms*) – Duration of the beep. If the duration is less than 0, then the method returns immediately and the frequency play continues to play indefinitely.

`speaker.play_notes(notes, tempo=120)`

Plays a sequence of musical notes. For example: `["C4/4", "C4/4", "G4/4", "G4/4"]`.

Each note is a string with the following format:

- The first character is the name of the note, `A` to `G` or `R` for a rest.
- Note names can also include an accidental `#` (sharp) or `b` (flat). `B#/Cb` and `E#/Fb` are not allowed.
- The note name is followed by the octave number `2` to `8`. For example `C4` is middle C. The octave changes to the next number at the note C, for example, `B3` is the note below middle C (`C4`).
- The octave is followed by `/` and a number that indicates the size of the note. For example `/4` is a quarter note, `/8` is an eighth note and so on.
- This can optionally followed by a `.` to make a dotted note. Dotted notes are 1-1/2 times as long as notes without a dot.
- The note can optionally end with a `_` which is a tie or a slur. This causes there to be no pause between this note and the next note.

Parameters

- **notes** (*iter*) – A sequence of notes to be played.
- **tempo** (int) – Beats per minute. A quarter note is one beat.

Using the battery

`battery.voltage()` → int: mV

Gets the voltage of the battery.

Returns
Battery voltage.

`battery.current()` → int: mA

Gets the current supplied by the battery.

Returns
Battery current.

Getting the charger status

`charger.connected()` → *bool*

Checks whether a charger is connected via USB.

Returns
`True` if a charger is connected, `False` if not.

`charger.current()` → int: mA

Gets the charging current.

Returns
Charging current.

`charger.status()` → *int*

Gets the status of the battery charger, represented by one of the following values. This corresponds to the battery light indicator right next to the USB port.

0. Not charging (light is off).
1. Charging (light is red).
2. Charging is complete (light is green).
3. There is a problem with the charger (light is yellow).

Returns
Status value.

System control

`system.set_stop_button(button)`

Sets the button or button combination that stops a running script.

Normally, the center button is used to stop a running script. You can change or disable this behavior in order to use the button for other purposes.

Parameters
button (`Button`) – A button such as *`Button.CENTER`*, or a tuple of multiple buttons. Choose `None` to disable the stop button altogether.

`system.name()` → *str*

Gets the hub name. This is the name you see when connecting via Bluetooth.

Returns
The hub name.

`system.storage(self, offset, write=)`

`system.storage(self, offset, read=)` → *bytes*

Reads or writes binary data to persistent storage.

This lets you store data that can be used the next time you run the program.

The data will be saved to flash memory when you turn the hub off normally. It will not be saved if the batteries are removed *while* the hub is still running.

Once saved, the data will remain available even after you remove the batteries.

Parameters

- **offset** (int) – The offset from the start of the user storage memory, in bytes.
- **read** (int) – The number of bytes to read. Omit this argument when writing.
- **write** (bytes) – The bytes to write. Omit this argument when reading.

Returns
The bytes read if reading, otherwise `None`.

Raises
ValueError – If you try to read or write data outside of the allowed range.

You can store up to 512 bytes of data on this hub.

`system.shutdown()`

Stops your program and shuts the hub down.

`system.reset_reason()` → *int*

Finds out how and why the hub (re)booted. This can be useful to diagnose some problems.

Returns

- `0` if the hub was previously powered off normally.
- `1` if the hub rebooted automatically, like after a firmware update.
- `2` if the hub previously crashed due to a watchdog timeout, which indicates a firmware issue.

Note: The examples below use the `PrimeHub` class. The examples work fine on both hubs because they are the identical. If you prefer, you can change this to `InventorHub`.

1.4.1 Status light examples

Turning the light on and off

```
from pybricks.hubs import PrimeHub
from pybricks.parameters import Color
from pybricks.tools import wait

# Initialize the hub.
hub = PrimeHub()

# Turn the light on and off 5 times.
for i in range(5):

    hub.light.on(Color.RED)
    wait(1000)

    hub.light.off()
    wait(500)
```

Changing brightness and using custom colors

```
from pybricks.hubs import PrimeHub
from pybricks.parameters import Color
from pybricks.tools import wait

# Initialize the hub.
hub = PrimeHub()

# Show the color at 30% brightness.
hub.light.on(Color.RED * 0.3)

wait(2000)

# Use your own custom color.
hub.light.on(Color(h=30, s=100, v=50))

wait(2000)

# Go through all the colors.
for hue in range(360):
    hub.light.on(Color(hue))
    wait(10)
```

Making the light blink

```
from pybricks.hubs import PrimeHub
from pybricks.parameters import Color
from pybricks.tools import wait

# Initialize the hub
hub = PrimeHub()

# Keep blinking red on and off.
hub.light.blink(Color.RED, [500, 500])

wait(10000)

# Keep blinking green slowly and then quickly.
hub.light.blink(Color.GREEN, [500, 500, 50, 900])

wait(10000)
```

Creating light animations

```
from pybricks.hubs import PrimeHub
from pybricks.parameters import Color
from pybricks.tools import wait
from umath import sin, pi

# Initialize the hub.
hub = PrimeHub()

# Make an animation with multiple colors.
hub.light.animate([Color.RED, Color.GREEN, Color.NONE], interval=500)

wait(10000)

# Make the color RED grow faint and bright using a sine pattern.
hub.light.animate([Color.RED * (0.5 * sin(i / 15 * pi) + 0.5) for i in range(30)], 40)

wait(10000)

# Cycle through a rainbow of colors.
hub.light.animate([Color(h=i * 8) for i in range(45)], interval=40)

wait(10000)
```

1.4.2 Matrix display examples

Displaying images

```
from pybricks.hubs import PrimeHub
from pybricks.tools import wait
from pybricks.parameters import Icon

# Initialize the hub.
hub = PrimeHub()

# Display a big arrow pointing up.
hub.display.icon(Icon.UP)

# Wait so we can see what is displayed.
wait(2000)

# Display a heart at half brightness.
hub.display.icon(Icon.HEART / 2)

# Wait so we can see what is displayed.
wait(2000)
```

Displaying numbers

```
from pybricks.hubs import PrimeHub
from pybricks.tools import wait

# Initialize the hub.
hub = PrimeHub()

# Count from 0 to 99.
for i in range(100):
    hub.display.number(i)
    wait(200)
```

Displaying text

```
from pybricks.hubs import PrimeHub
from pybricks.tools import wait

# Initialize the hub.
hub = PrimeHub()

# Display the letter A for two seconds.
hub.display.char("A")
wait(2000)

# Display text, one letter at a time.
hub.display.text("Hello, world!")
```

Displaying individual pixels

```
from pybricks.hubs import PrimeHub
from pybricks.tools import wait

# Initialize the hub.
hub = PrimeHub()

# Turn on the pixel at row 1, column 2.
hub.display.pixel(1, 2)
wait(2000)

# Turn on the pixel at row 2, column 4, at 50% brightness.
hub.display.pixel(2, 4, 50)
wait(2000)

# Turn off the pixel at row 1, column 2.
hub.display.pixel(1, 2, 0)
wait(2000)
```

Changing the display orientation

```
from pybricks.hubs import PrimeHub
from pybricks.tools import wait
from pybricks.parameters import Side

# Initialize the hub.
hub = PrimeHub()

# Rotate the display. Now right is up.
hub.display.orientation(up=Side.RIGHT)

# Display a number. This will be shown sideways.
hub.display.number(23)

# Wait so we can see what is displayed.
wait(10000)
```

```
from pybricks.hubs import PrimeHub
from pybricks.parameters import Icon
from pybricks.tools import wait

# Initialize the hub.
hub = PrimeHub()

while True:

    # Check which side of the hub is up.
    up_side = hub.imu.up()

    # Use this side to set the display orientation.
```

(continues on next page)

(continued from previous page)

```
    hub.display.orientation(up_side)

    # Display something, like an arrow.
    hub.display.icon(Icon.UP)

    wait(10)
```

Making your own images

```
from pybricks.hubs import PrimeHub
from pybricks.tools import wait
from pybricks.geometry import Matrix

# Initialize the hub.
hub = PrimeHub()

# Make a square that is bright on the outside and faint in the middle.
SQUARE = Matrix(
    [
        [100, 100, 100, 100, 100],
        [100, 50, 50, 50, 100],
        [100, 50, 0, 50, 100],
        [100, 50, 50, 50, 100],
        [100, 100, 100, 100, 100],
    ]
)

# Display the square.
hub.display.icon(SQUARE)
wait(3000)

# Make an image using a Python list comprehension. In this image, the
# brightness of each pixel is the sum of the row and column index. So the
# light is faint in the top left and bright in the bottom right.
GRADIENT = Matrix([[(r + c) for c in range(5)] for r in range(5)]) * 12.5

# Display the generated gradient.
hub.display.icon(GRADIENT)
wait(3000)
```

Combining icons to make expressions

```
from pybricks.hubs import PrimeHub
from pybricks.parameters import Icon, Side
from pybricks.tools import wait

from urandom import randint

# Initialize the hub.
```

(continues on next page)

(continued from previous page)

```
hub = PrimeHub()
hub.display.orientation(up=Side.RIGHT)

while True:

    # Start with random left brow: up or down.
    if randint(0, 100) < 70:
        brows = Icon.EYE_LEFT_BROW * 0.5
    else:
        brows = Icon.EYE_LEFT_BROW_UP * 0.5

    # Add random right brow: up or down.
    if randint(0, 100) < 70:
        brows += Icon.EYE_RIGHT_BROW * 0.5
    else:
        brows += Icon.EYE_RIGHT_BROW_UP * 0.5

    for i in range(3):
        # Display eyes open plus the random brows.
        hub.display.icon(Icon.EYE_LEFT + Icon.EYE_RIGHT + brows)
        wait(2000)

        # Display eyes blinked plus the random brows.
        hub.display.icon(Icon.EYE_LEFT_BLINK * 0.7 + Icon.EYE_RIGHT_BLINK * 0.7 + brows)
        wait(200)
```

Displaying animations

```
from pybricks.hubs import PrimeHub
from pybricks.parameters import Icon
from pybricks.tools import wait

# Initialize the hub.
hub = PrimeHub()

# Turn the hub light off (optional).
hub.light.off()

# Create a list of intensities from 0 to 100 and back.
brightness = list(range(0, 100, 4)) + list(range(100, 0, -4))

# Create an animation of the heart icon with changing brightness.
hub.display.animate([Icon.HEART * i / 100 for i in brightness], 30)

# The animation repeats in the background. Here we just wait.
while True:
    wait(100)
```

1.4.3 Button examples

Detecting button presses

```
from pybricks.hubs import PrimeHub
from pybricks.parameters import Button, Icon
from pybricks.tools import wait

# Initialize the hub.
hub = PrimeHub()

# Wait for any button to be pressed, and save the result.
pressed = []
while not any(pressed):
    pressed = hub.buttons.pressed()
    wait(10)

# Display a circle.
hub.display.icon(Icon.CIRCLE)

# Wait for all buttons to be released.
while any(hub.buttons.pressed()):
    wait(10)

# Display an arrow to indicate which button was pressed.
if Button.LEFT in pressed:
    hub.display.icon(Icon.ARROW_LEFT_DOWN)
elif Button.RIGHT in pressed:
    hub.display.icon(Icon.ARROW_RIGHT_DOWN)
elif Button.BLUETOOTH in pressed:
    hub.display.icon(Icon.ARROW_RIGHT_UP)

wait(3000)
```

1.4.4 IMU examples

Testing which way is up

```
from pybricks.hubs import PrimeHub
from pybricks.parameters import Color, Side
from pybricks.tools import wait

# Initialize the hub.
hub = PrimeHub()

# Define colors for each side in a dictionary.
SIDE_COLORS = {
    Side.TOP: Color.RED,
    Side.BOTTOM: Color.BLUE,
    Side.LEFT: Color.GREEN,
    Side.RIGHT: Color.YELLOW,
```

(continues on next page)

(continued from previous page)

```
    Side.FRONT: Color.MAGENTA,
    Side.BACK: Color.BLACK,
}

# Keep updating the color based on detected up side.
while True:

    # Check which side of the hub is up.
    up_side = hub.imu.up()

    # Change the color based on the side.
    hub.light.on(SIDE_COLORS[up_side])

    # Also print the result.
    print(up_side)
    wait(50)
```

Reading the tilt value

```
from pybricks.hubs import PrimeHub
from pybricks.tools import wait

# Initialize the hub.
hub = PrimeHub()

while True:
    # Read the tilt values.
    pitch, roll = hub.imu.tilt()

    # Print the result.
    print(pitch, roll)
    wait(200)
```

Using a custom hub orientation

```
from pybricks.hubs import PrimeHub
from pybricks.tools import wait
from pybricks.geometry import Axis

# Initialize the hub. In this case, specify that the hub is mounted with the
# top side facing forward and the front side facing to the right.
# For example, this is how the hub is mounted in BLAST in the 51515 set.
hub = PrimeHub(top_side=Axis.X, front_side=-Axis.Y)

while True:
    # Read the tilt values. Now, the values are 0 when BLAST stands upright.
    # Leaning forward gives positive pitch. Leaning right gives positive roll.
    pitch, roll = hub.imu.tilt()

```

(continues on next page)

(continued from previous page)

```
    # Print the result.
    print(pitch, roll)
    wait(200)
```

Reading acceleration and angular velocity vectors

```
from pybricks.hubs import PrimeHub
from pybricks.tools import wait

# Initialize the hub.
hub = PrimeHub()

# Get the acceleration vector in g's.
print(hub.imu.acceleration() / 9810)

# Get the angular velocity vector.
print(hub.imu.angular_velocity())

# Wait so we can see what we printed
wait(5000)
```

Reading acceleration and angular velocity on one axis

```
from pybricks.hubs import PrimeHub
from pybricks.tools import wait
from pybricks.geometry import Axis

# Initialize the hub.
hub = PrimeHub()

# Get the acceleration or angular_velocity along a single axis.
# If you need only one value, this is more memory efficient.
while True:

    # Read the forward acceleration.
    forward_acceleration = hub.imu.acceleration(Axis.X)

    # Read the yaw rate.
    yaw_rate = hub.imu.angular_velocity(Axis.Z)

    # Print the yaw rate.
    print(yaw_rate)
    wait(100)
```

1.4.5 System examples

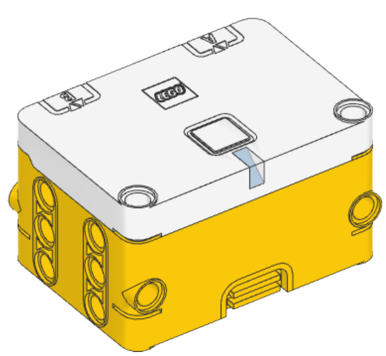
Changing the stop button combination

```
from pybricks.hubs import PrimeHub
from pybricks.parameters import Button

# Initialize the hub.
hub = PrimeHub()

# Configure the stop button combination. Now, your program stops
# if you press the center and Bluetooth buttons simultaneously.
hub.system.set_stop_button((Button.CENTER, Button.BLUETOOTH))

# Now we can use the center button as a normal button.
while True:

    # Play a sound if the center button is pressed.
    if Button.CENTER in hub.buttons.pressed():
        hub.speaker.beep()
```

Turning the hub off

```
from pybricks.hubs import PrimeHub
from pybricks.tools import wait

# Initialize the hub.
hub = PrimeHub()

# Say goodbye and give some time to send it.
print("Goodbye!")
wait(100)

# Shut the hub down.
hub.system.shutdown()
```

1.5 Essential Hub



`class EssentialHub(top_side=Axis.Z, front_side=Axis.X)`

LEGO® SPIKE Essential Hub.

Initializes the hub. Optionally, specify how the hub is *placed in your design* by saying in which direction the top side (with the button) and the front side (with the USB port, and I/O ports A and B) are pointing.

Parameters

- **top_side** (Axis) – The axis that passes through the *top side* of the hub.
- **front_side** (Axis) – The axis that passes through the *front side* of the hub.

Using the hub status light

`light.on(color)`

Turns on the light at the specified color.

Parameters
color (Color) – Color of the light.

`light.off()`

Turns off the light.

`light.blink(color, durations)`

Blinks the light at a given color by turning it on and off for given durations.

The light keeps blinking indefinitely while the rest of your program keeps running.

This method provides a simple way to make basic but useful patterns. For more generic and multi-color patterns, use `animate()` instead.

Parameters

- **color** (Color) – Color of the light.
- **durations** (list) – Sequence of time values of the form `[on_1, off_1, on_2, off_2, ...]`.

`light.animate(colors, interval)`

Animates the light with a sequence of colors, shown one by one for the given interval.

The animation runs in the background while the rest of your program keeps running. When the animation completes, it repeats.

Parameters

- **colors** (list) – Sequence of *Color* values.
- **interval** (Number, *ms*) – Time between color updates.

Using the button

`button.pressed()` → Collection[*Button*]

Checks which buttons are currently pressed.

Returns
Set of pressed buttons.

Using the IMU

`imu.up()` → *Side*

Checks which side of the hub currently faces upward.

Returns
`Side.TOP`, `Side.BOTTOM`, `Side.LEFT`, `Side.RIGHT`, `Side.FRONT` or `Side.BACK`.

`imu.tilt()` → Tuple[*int*, *int*]

Gets the pitch and roll angles. This is relative to the *user-specified neutral orientation*.

The order of rotation is pitch-then-roll. This is equivalent to a positive rotation along the robot y-axis and then a positive rotation along the x-axis.

Returns
Tuple of pitch and roll angles.

`imu.acceleration(`*axis*`)` → float: mm/s^2

`imu.acceleration()` → vector: mm/s^2

Gets the acceleration of the device along a given axis in the *robot reference frame*.

Parameters
axis (`Axis`) – Axis along which the acceleration should be measured.

Returns
Acceleration along the specified axis. If you specify no axis, this returns a vector of accelerations along all axes.

`imu.angular_velocity(`*axis*`)` → float: deg/s

`imu.angular_velocity()` → vector: deg/s

Gets the angular velocity of the device along a given axis in the *robot reference frame*.

Parameters
axis (`Axis`) – Axis along which the angular velocity should be measured.

Returns
Angular velocity along the specified axis. If you specify no axis, this returns a vector of accelerations along all axes.

`imu.heading()` → float: deg

Gets the heading angle relative to the starting orientation. It is a positive rotation around the *z-axis in the robot frame*, prior to applying any tilt rotation.

For a vehicle viewed from the top, this means that a positive heading value corresponds to a counterclockwise rotation.

Note: This method is not yet implemented.

Returns
Heading angle relative to starting orientation.

`imu.reset_heading(`*angle*`)`

Resets the accumulated heading angle of the robot.

Note: This method is not yet implemented.

Parameters
`angle` (`Number`, *deg*) – Value to which the heading should be reset.

Using the battery

`battery.voltage()` → int: mV

Gets the voltage of the battery.

Returns
Battery voltage.

`battery.current()` → int: mA

Gets the current supplied by the battery.

Returns
Battery current.

Getting the charger status

`charger.connected()` → *bool*

Checks whether a charger is connected via USB.

Returns
`True` if a charger is connected, `False` if not.

`charger.current()` → int: mA

Gets the charging current.

Returns
Charging current.

`charger.status()` → *int*

Gets the status of the battery charger, represented by one of the following values. This corresponds to the battery light indicator right next to the USB port.

0. Not charging (light is off).
1. Charging (light is red).
2. Charging is complete (light is green).
3. There is a problem with the charger (light is yellow).

Returns
Status value.

System control

`system.set_stop_button(button)`

Sets the button or button combination that stops a running script.

Normally, the center button is used to stop a running script. You can change or disable this behavior in order to use the button for other purposes.

Parameters
`button` (`Button`) – A button such as `Button.CENTER`, or a tuple of multiple buttons. Choose `None` to disable the stop button altogether.

`system.name()` → *str*

Gets the hub name. This is the name you see when connecting via Bluetooth.

Returns
The hub name.

`system.storage(self, offset, write=)`

`system.storage(self, offset, read=)` → *bytes*

Reads or writes binary data to persistent storage.

This lets you store data that can be used the next time you run the program.

The data will be saved to flash memory when you turn the hub off normally. It will not be saved if the batteries are removed *while* the hub is still running.

Once saved, the data will remain available even after you remove the batteries.

Parameters

- **offset** (int) – The offset from the start of the user storage memory, in bytes.
- **read** (int) – The number of bytes to read. Omit this argument when writing.
- **write** (bytes) – The bytes to write. Omit this argument when reading.

Returns
The bytes read if reading, otherwise `None`.

Raises
ValueError – If you try to read or write data outside of the allowed range.

You can store up to 512 bytes of data on this hub.

`system.shutdown()`

Stops your program and shuts the hub down.

`system.reset_reason()` → *int*

Finds out how and why the hub (re)booted. This can be useful to diagnose some problems.

Returns

- `0` if the hub was previously powered off normally.
- `1` if the hub rebooted automatically, like after a firmware update.
- `2` if the hub previously crashed due to a watchdog timeout, which indicates a firmware issue.

1.5.1 Status light examples

Turning the light on and off

```
from pybricks.hubs import EssentialHub
from pybricks.parameters import Color
from pybricks.tools import wait

# Initialize the hub.
hub = EssentialHub()

# Turn the light on and off 5 times.
```

(continues on next page)

(continued from previous page)

```
for i in range(5):

    hub.light.on(Color.RED)
    wait(1000)

    hub.light.off()
    wait(500)
```

Changing brightness and using custom colors

```
from pybricks.hubs import EssentialHub
from pybricks.parameters import Color
from pybricks.tools import wait

# Initialize the hub.
hub = EssentialHub()

# Show the color at 30% brightness.
hub.light.on(Color.RED * 0.3)

wait(2000)

# Use your own custom color.
hub.light.on(Color(h=30, s=100, v=50))

wait(2000)

# Go through all the colors.
for hue in range(360):
    hub.light.on(Color(hue))
    wait(10)
```

Making the light blink

```
from pybricks.hubs import EssentialHub
from pybricks.parameters import Color
from pybricks.tools import wait

# Initialize the hub
hub = EssentialHub()

# Keep blinking red on and off.
hub.light.blink(Color.RED, [500, 500])

wait(10000)

# Keep blinking green slowly and then quickly.
hub.light.blink(Color.GREEN, [500, 500, 50, 900])
```

(continues on next page)

(continued from previous page)

```
wait(10000)
```

Creating light animations

```
from pybricks.hubs import EssentialHub
from pybricks.parameters import Color
from pybricks.tools import wait
from umath import sin, pi

# Initialize the hub.
hub = EssentialHub()

# Make an animation with multiple colors.
hub.light.animate([Color.RED, Color.GREEN, Color.NONE], interval=500)

wait(10000)

# Make the color RED grow faint and bright using a sine pattern.
hub.light.animate([Color.RED * (0.5 * sin(i / 15 * pi) + 0.5) for i in range(30)], 40)

wait(10000)

# Cycle through a rainbow of colors.
hub.light.animate([Color(h=i * 8) for i in range(45)], interval=40)

wait(10000)
```

1.5.2 IMU examples

Testing which way is up

```
from pybricks.hubs import EssentialHub
from pybricks.parameters import Color, Side
from pybricks.tools import wait

# Initialize the hub.
hub = EssentialHub()

# Define colors for each side in a dictionary.
SIDE_COLORS = {
    Side.TOP: Color.RED,
    Side.BOTTOM: Color.BLUE,
    Side.LEFT: Color.GREEN,
    Side.RIGHT: Color.YELLOW,
    Side.FRONT: Color.MAGENTA,
    Side.BACK: Color.BLACK,
}
```

(continues on next page)

(continued from previous page)

```
# Keep updating the color based on detected up side.
while True:

    # Check which side of the hub is up.
    up_side = hub.imu.up()

    # Change the color based on the side.
    hub.light.on(SIDE_COLORS[up_side])

    # Also print the result.
    print(up_side)
    wait(50)
```

Reading the tilt value

```
from pybricks.hubs import EssentialHub
from pybricks.tools import wait

# Initialize the hub.
hub = EssentialHub()

while True:
    # Read the tilt values.
    pitch, roll = hub.imu.tilt()

    # Print the result.
    print(pitch, roll)
    wait(200)
```

Using a custom hub orientation

```
from pybricks.hubs import EssentialHub
from pybricks.tools import wait
from pybricks.geometry import Axis

# Initialize the hub. In this case, specify that the hub is mounted with the
# top side facing forward and the front side facing to the right.
# For example, this is how the hub is mounted in BLAST in the 51515 set.
hub = EssentialHub(top_side=Axis.X, front_side=-Axis.Y)

while True:
    # Read the tilt values. Now, the values are 0 when BLAST stands upright.
    # Leaning forward gives positive pitch. Leaning right gives positive roll.
    pitch, roll = hub.imu.tilt()

    # Print the result.
    print(pitch, roll)
    wait(200)
```

Reading acceleration and angular velocity vectors

```
from pybricks.hubs import EssentialHub
from pybricks.tools import wait

# Initialize the hub.
hub = EssentialHub()

# Get the acceleration vector in g's.
print(hub.imu.acceleration() / 9810)

# Get the angular velocity vector.
print(hub.imu.angular_velocity())

# Wait so we can see what we printed
wait(5000)
```

Reading acceleration and angular velocity on one axis

```
from pybricks.hubs import EssentialHub
from pybricks.tools import wait
from pybricks.geometry import Axis

# Initialize the hub.
hub = EssentialHub()

# Get the acceleration or angular_velocity along a single axis.
# If you need only one value, this is more memory efficient.
while True:

    # Read the forward acceleration.
    forward_acceleration = hub.imu.acceleration(Axis.X)

    # Read the yaw rate.
    yaw_rate = hub.imu.angular_velocity(Axis.Z)

    # Print the yaw rate.
    print(yaw_rate)
    wait(100)
```

1.5.3 System examples

Using the stop button during your program

```
from pybricks.hubs import EssentialHub
from pybricks.parameters import Color, Button
from pybricks.tools import wait, StopWatch

# Initialize the hub.
```

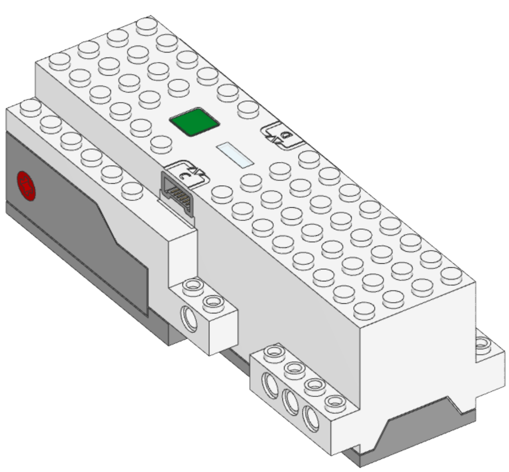
(continues on next page)

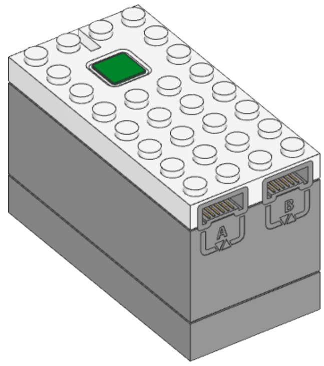
(continued from previous page)

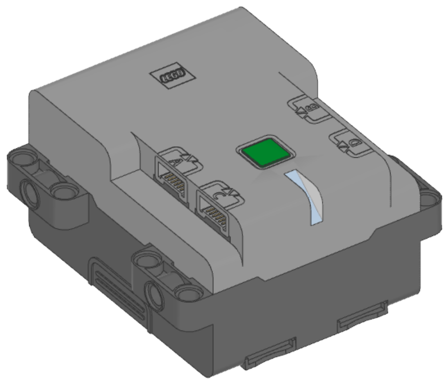
```
hub = EssentialHub()

# Disable the stop button.
hub.system.set_stop_button(None)

# Check the button for 5 seconds.
watch = StopWatch()
while watch.time() < 5000:

    # Set light to green if pressed, else red.
    if hub.button.pressed():
        hub.light.on(Color.GREEN)
    else:
        hub.light.on(Color.RED)

# Enable the stop button again.
hub.system.set_stop_button(Button.CENTER)

# Now you can press the stop button as usual.
wait(5000)
```

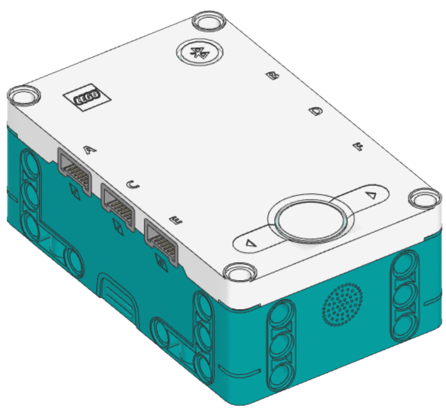
Turning the hub off

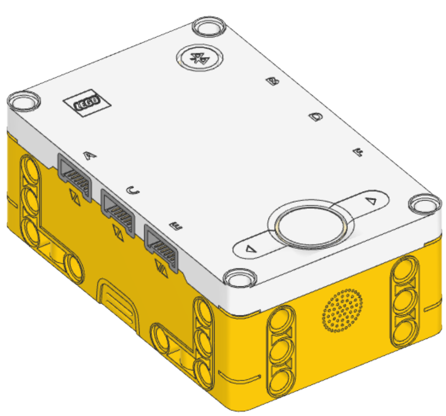
```
from pybricks.hubs import EssentialHub
from pybricks.tools import wait

# Initialize the hub.
hub = EssentialHub()

# Say goodbye and give some time to send it.
print("Goodbye!")
wait(100)

# Shut the hub down.
hub.system.shutdown()
```

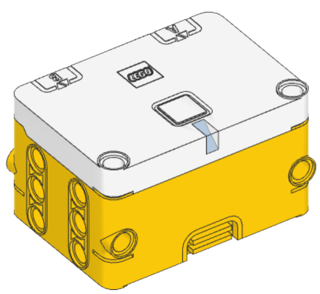












CHAPTER

TWO

PUPDEVICES – MOTORS, SENSORS, LIGHTS

LEGO® Powered Up motor, sensors, and lights.

2.1 Motors without rotation sensors

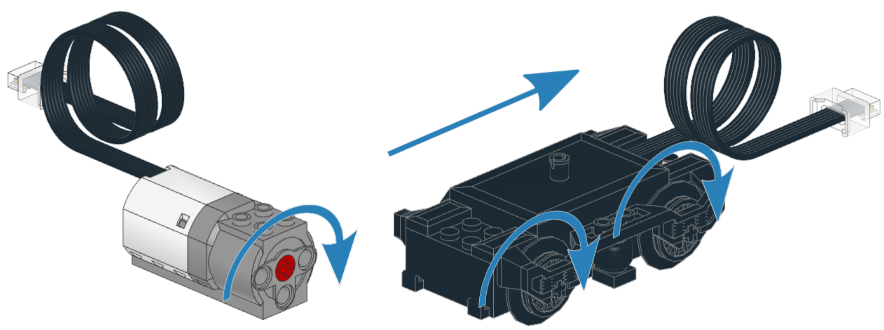


Figure 2.1: Powered Up motors without rotation sensors. The arrows indicate the default positive direction.

class DCMotor(*port*, *positive_direction=Direction.CLOCKWISE*)

LEGO® Powered Up motor without rotation sensors.

Parameters

- **port** (Port) – Port to which the motor is connected.
- **positive_direction** (Direction) – Which direction the motor should turn when you give a positive duty cycle value.

dc(*duty*)

Rotates the motor at a given duty cycle (also known as "power").

Parameters

duty (Number, %) – The duty cycle (-100.0 to 100).

stop()

Stops the motor and lets it spin freely.

The motor gradually stops due to friction.

brake()

Passively brakes the motor.

The motor stops due to friction, plus the voltage that is generated while the motor is still moving.

settings(*max_voltage*)

settings() → Tuple[*int*]

Configures motor settings. If no arguments are given, this returns the current values.

Parameters

max_voltage (Number, *mV*) – Maximum voltage applied to the motor during all motor commands.

2.1.1 Examples

Making a train drive forever

```
from pybricks.pupdevices import DCMotor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize the motor.
train_motor = DCMotor(Port.A)

# Choose the "power" level for your train. Negative means reverse.
train_motor.dc(50)

# Keep doing nothing. The train just keeps going.
while True:
    wait(1000)
```

Making the motor move back and forth

```
from pybricks.pupdevices import DCMotor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize a motor without rotation sensors on port A.
example_motor = DCMotor(Port.A)

# Make the motor go clockwise (forward) at 70% duty cycle ("70% power").
example_motor.dc(70)

# Wait for three seconds.
wait(3000)

# Make the motor go counterclockwise (backward) at 70% duty cycle.
example_motor.dc(-70)

# Wait for three seconds.
wait(3000)
```

Changing the positive direction

```
from pybricks.pupdevices import DCMotor
from pybricks.parameters import Port, Direction
from pybricks.tools import wait

# Initialize a motor without rotation sensors on port A,
# with the positive direction as counterclockwise.
example_motor = DCMotor(Port.A, Direction.COUNTERCLOCKWISE)

# When we choose a positive duty cycle, the motor now goes counterclockwise.
example_motor.dc(70)

# This is useful when your (train) motor is mounted in reverse or upside down.
# By changing the positive direction, your script will be easier to read,
# because a positive value now makes your train/robot go forward.

# Wait for three seconds.
wait(3000)
```

Starting and stopping

```
from pybricks.pupdevices import DCMotor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize a motor without rotation sensors on port A.
example_motor = DCMotor(Port.A)

# Start and stop 10 times.
for count in range(10):
    print("Counter:", count)

    example_motor.dc(70)
    wait(1000)

    example_motor.stop()
    wait(1000)
```

2.2 Motors with rotation sensors

class Motor(*port*, *positive_direction=Direction.CLOCKWISE*, *gears=None*, *reset_angle=True*)

LEGO® Powered Up motor with rotation sensors.

Parameters

- **port** (Port) – Port to which the motor is connected.
- **positive_direction** (Direction) – Which direction the motor should turn when you give a positive speed value or angle.

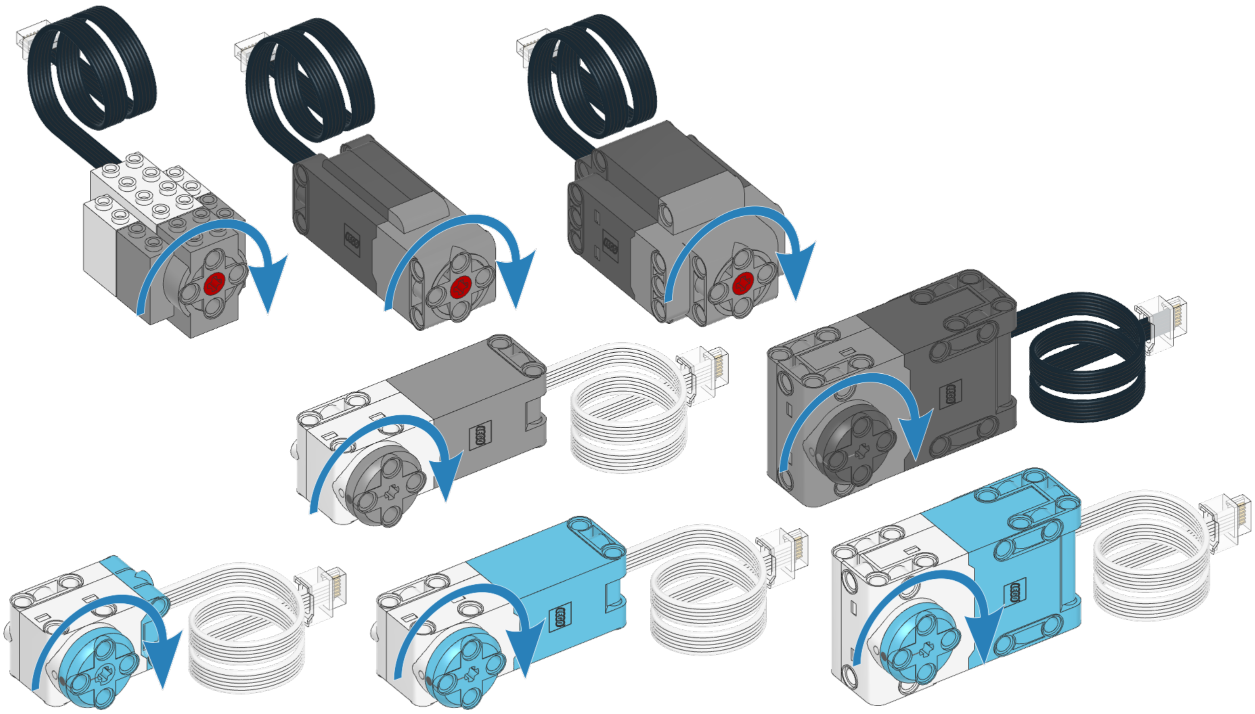


Figure 2.2: Powered Up motors with rotation sensors. The arrows indicate the default positive direction. See the *hubs* module for default directions of built-in motors.

- **gears** (list) – List of gears linked to the motor.

  For example: `[12, 36]` represents a gear train with a 12-tooth and a 36-tooth gear. Use a list of lists for multiple gear trains, such as `[[12, 36], [20, 16, 40]]`.

  When you specify a gear train, all motor commands and settings are automatically adjusted to account for the resulting gear ratio. The motor direction remains unchanged by this.

- **reset_angle** (bool) – Choose `True` to reset the rotation sensor value to the absolute marker angle (between -180 and 179). Choose `False` to keep the current value, so your program knows where it left off last time.

Measuring

speed() → int: deg/s

Gets the speed of the motor.

Returns
Motor speed.

angle() → int: deg

Gets the rotation angle of the motor.

Returns
Motor angle.

reset_angle(*angle=None*)

Sets the accumulated rotation angle of the motor to a desired value.

If you don't specify an angle, the absolute angle will be used if your motor supports it.

Parameters
`angle` (`Number`, *deg*) – Value to which the angle should be reset.

`load()` → int: mNm

Estimates the load that holds back the motor when it tries to move.

Returns
The load torque.

`stalled()` → *bool*

Checks if the motor is currently stalled.

It is stalled when it cannot reach the target speed or position, even with the maximum actuation signal.

Returns
`True` if the motor is stalled, `False` if not.

Stopping

`stop()`

Stops the motor and lets it spin freely.

The motor gradually stops due to friction.

`brake()`

Passively brakes the motor.

The motor stops due to friction, plus the voltage that is generated while the motor is still moving.

`hold()`

Stops the motor and actively holds it at its current angle.

Running forever

`run(speed)`

Runs the motor at a constant speed.

The motor accelerates to the given speed and keeps running at this speed until you give a new command.

Parameters
`speed` (`Number`, *deg/s*) – Speed of the motor.

`dc(duty)`

Rotates the motor at a given duty cycle (also known as "power").

Parameters
`duty` (`Number`, *%*) – The duty cycle (-100.0 to 100).

Running by a fixed amount

run_time(*speed*, *time*, *then=Stop.HOLD*, *wait=True*)

Runs the motor at a constant speed for a given amount of time.

The motor accelerates to the given speed, keeps running at this speed, and then decelerates. The total maneuver lasts for exactly the given amount of `time`.

Parameters

- **speed** (Number, *deg/s*) – Speed of the motor.
- **time** (Number, *ms*) – Duration of the maneuver.
- **then** (Stop) – What to do after coming to a standstill.
- **wait** (bool) – Wait for the maneuver to complete before continuing with the rest of the program.

run_angle(*speed*, *rotation_angle*, *then=Stop.HOLD*, *wait=True*)

Runs the motor at a constant speed by a given angle.

Parameters

- **speed** (Number, *deg/s*) – Speed of the motor.
- **rotation_angle** (Number, *deg*) – Angle by which the motor should rotate.
- **then** (Stop) – What to do after coming to a standstill.
- **wait** (bool) – Wait for the maneuver to complete before continuing with the rest of the program.

run_target(*speed*, *target_angle*, *then=Stop.HOLD*, *wait=True*)

Runs the motor at a constant speed towards a given target angle.

The direction of rotation is automatically selected based on the target angle. It does not matter if `speed` is positive or negative.

Parameters

- **speed** (Number, *deg/s*) – Speed of the motor.
- **target_angle** (Number, *deg*) – Angle that the motor should rotate to.
- **then** (Stop) – What to do after coming to a standstill.
- **wait** (bool) – Wait for the motor to reach the target before continuing with the rest of the program.

track_target(*target_angle*)

Tracks a target angle. This is similar to *run_target()*, but the usual smooth acceleration is skipped: it will move to the target angle as fast as possible. This method is useful if you want to continuously change the target angle.

Parameters

target_angle (Number, *deg*) – Target angle that the motor should rotate to.

run_until_stalled(*speed*, *then=Stop.COAST*, *duty_limit=None*) → int: deg

Runs the motor at a constant speed until it stalls.

Parameters

- **speed** (Number, *deg/s*) – Speed of the motor.

- **then** (Stop) – What to do after coming to a standstill.
- **duty_limit** (Number, *%*) – Duty cycle limit during this command. This is useful to avoid applying the full motor torque to a geared or lever mechanism. If it is None, the duty limit won't be changed during this command.

Returns
Angle at which the motor becomes stalled.

done() → *bool*

Checks if an ongoing command or maneuver is done.

Returns
True if the command is done, False if not.

Motor settings

settings(*max_voltage*)

settings() → Tuple[*int*]

Configures motor settings. If no arguments are given, this returns the current values.

Parameters
max_voltage (Number, *mV*) – Maximum voltage applied to the motor during all motor commands.

Control settings

control.**limits**(*speed*, *acceleration*, *torque*)

control.**limits**() → Tuple[*int*, *int*, *int*]

Configures the maximum speed, acceleration, and torque.

If no arguments are given, this will return the current values.

The new acceleration and speed limit will become effective when you give a new motor command. Ongoing maneuvers are not affected.

Parameters

- **speed** (Number, *deg/s or* Number, *mm/s*) – Maximum speed. All speed commands will be capped to this value.
- **acceleration** (Number, *deg/s^2 or* Number, *mm/s^2*) – Slope of the speed curve when accelerating or decelerating. Use a tuple to set acceleration and deceleration separately. If one value is given, it is used for both.
- **torque** (*torque: mNm*) – Maximum feedback torque during control.

control.**pid**(*kp*, *ki*, *kd*, *reserved*, *integral_rate*)

control.**pid**() → Tuple[*int*, *int*, *int*, None, *int*]

Gets or sets the PID values for position and speed control.

If no arguments are given, this will return the current values.

Parameters

- **kp** (int) – Proportional position control constant. It is the feedback torque per degree of error: μNm/deg.

- **ki** (int) – Integral position control constant. It is the feedback torque per accumulated degree of error: μNm/(deg s).
- **kd** (int) – Derivative position (or proportional speed) control constant. It is the feedback torque per unit of speed: μNm/(deg/s).
- **reserved** – This setting is not used.
- **integral_rate** (Number, *deg/s or* Number, *mm/s*) – Maximum rate at which the error integral is allowed to grow.

control.**target_tolerances**(*speed*, *position*)

control.**target_tolerances**() → Tuple[*int*, *int*]

Gets or sets the tolerances that say when a maneuver is done.

If no arguments are given, this will return the current values.

Parameters

- **speed** (Number, *deg/s or* Number, *mm/s*) – Allowed deviation from zero speed before motion is considered complete.
- **position** (Number, deg or *distance: mm*) – Allowed deviation from the target before motion is considered complete.

control.**stall_tolerances**(*speed*, *time*)

control.**stall_tolerances**() → Tuple[*int*, *int*]

Gets or sets stalling tolerances.

If no arguments are given, this will return the current values.

Parameters

- **speed** (Number, *deg/s or* Number, *mm/s*) – If the controller cannot reach this speed for some `time` even with maximum actuation, it is stalled.
- **time** (Number, *ms*) – How long the controller has to be below this minimum `speed` before we say it is stalled.

control.**scale**

Number of degrees that the motor turns to complete one degree at the output of the gear train. This is the gear ratio determined from the `gears` argument when initializing the motor.

Changed in version 3.2: The *done()*, *stalled()* and *load()* methods have been moved.

2.2.1 Initialization examples

Making the motor move back and forth

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize a motor on port A.
example_motor = Motor(Port.A)

# Make the motor run clockwise at 500 degrees per second.
example_motor.run(500)
```

(continues on next page)

(continued from previous page)

```
# Wait for three seconds.
wait(3000)

# Make the motor run counterclockwise at 500 degrees per second.
example_motor.run(-500)

# Wait for three seconds.
wait(3000)
```

Initializing multiple motors

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize motors on port A and B.
track_motor = Motor(Port.A)
gripper_motor = Motor(Port.B)

# Make both motors run at 500 degrees per second.
track_motor.run(500)
gripper_motor.run(500)

# Wait for three seconds.
wait(3000)
```

Setting the positive direction as counterclockwise

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port, Direction
from pybricks.tools import wait

# Initialize a motor on port A with the positive direction as counterclockwise.
example_motor = Motor(Port.A, Direction.COUNTERCLOCKWISE)

# When we choose a positive speed value, the motor now goes counterclockwise.
example_motor.run(500)

# This is useful when your motor is mounted in reverse or upside down.
# By changing the positive direction, your script will be easier to read,
# because a positive value now makes your robot/mechanism go forward.

# Wait for three seconds.
wait(3000)
```

Using gears

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port, Direction
from pybricks.tools import wait

# Initialize a motor on port A with the positive direction as counterclockwise.
# Also specify one gear train with a 12-tooth and a 36-tooth gear. The 12-tooth
# gear is attached to the motor axle. The 36-tooth gear is at the output axle.
geared_motor = Motor(Port.A, Direction.COUNTERCLOCKWISE, [12, 36])

# Make the output axle run at 100 degrees per second. The motor speed
# is automatically increased to compensate for the gears.
geared_motor.run(100)

# Wait for three seconds.
wait(3000)
```

2.2.2 Measurement examples

Measuring the angle and speed

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize a motor on port A.
example_motor = Motor(Port.A)

# Start moving at 300 degrees per second.
example_motor.run(300)

# Display the angle and speed 50 times.
for i in range(100):

    # Read the angle (degrees) and speed (degrees per second).
    angle = example_motor.angle()
    speed = example_motor.speed()

    # Print the values.
    print(angle, speed)

    # Wait some time so we can read what is displayed.
    wait(200)
```

Resetting the measured angle

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port

# Initialize a motor on port A.
example_motor = Motor(Port.A)

# Reset the angle to 0.
example_motor.reset_angle(0)

# Reset the angle to 1234.
example_motor.reset_angle(1234)

# Reset the angle to the absolute angle.
# This is only supported on motors that have
# an absolute encoder. For other motors, this
# will raise an error.
example_motor.reset_angle()
```

Getting the absolute angle

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize a motor on port A.
example_motor = Motor(Port.A)

while True:

    # Get the default angle value.
    angle = example_motor.angle()

    # Get the angle between 0 and 360.
    absolute_angle = example_motor.angle() % 360

    # Get the angle between -180 and 179.
    wrapped_angle = (example_motor.angle() + 180) % 360 - 180

    # Print the results.
    print(angle, absolute_angle, wrapped_angle)
    wait(100)
```

2.2.3 Movement examples

Basic usage of all run methods

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize a motor on port A.
example_motor = Motor(Port.A)

# Run at 500 deg/s and then stop by coasting.
print("Demo of run")
example_motor.run(500)
wait(1500)
example_motor.stop()
wait(1500)

# Run at 70% duty cycle ("power") and then stop by coasting.
print("Demo of dc")
example_motor.dc(50)
wait(1500)
example_motor.stop()
wait(1500)

# Run at 500 deg/s for two seconds.
print("Demo of run_time")
example_motor.run_time(500, 2000)
wait(1500)

# Run at 500 deg/s for 90 degrees.
print("Demo of run_angle")
example_motor.run_angle(500, 90)
wait(1500)

# Run at 500 deg/s back to the 0 angle
print("Demo of run_target to 0")
example_motor.run_target(500, 0)
wait(1500)

# Run at 500 deg/s back to the -90 angle
print("Demo of run_target to -90")
example_motor.run_target(500, -90)
wait(1500)

# Run at 500 deg/s until the motor stalls
print("Demo of run_until_stalled")
example_motor.run_until_stalled(500)
print("Done")
wait(1500)
```

Stopping ongoing movements in different ways

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize a motor on port A.
example_motor = Motor(Port.A)

# Run at 500 deg/s and then stop by coasting.
example_motor.run(500)
wait(1500)
example_motor.stop()
wait(1500)

# Run at 500 deg/s and then stop by braking.
example_motor.run(500)
wait(1500)
example_motor.brake()
wait(1500)

# Run at 500 deg/s and then stop by holding.
example_motor.run(500)
wait(1500)
example_motor.hold()
wait(1500)

# Run at 500 deg/s and then stop by running at 0 speed.
example_motor.run(500)
wait(1500)
example_motor.run(0)
wait(1500)
```

Using the `then` argument to change how a run command stops

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port, Stop
from pybricks.tools import wait

# Initialize a motor on port A.
example_motor = Motor(Port.A)

# By default, the motor holds the position. It keeps
# correcting the angle if you move it.
example_motor.run_angle(500, 360)
wait(1000)

# This does exactly the same as above.
example_motor.run_angle(500, 360, then=Stop.HOLD)
wait(1000)
```

(continues on next page)

(continued from previous page)

```
# You can also brake. This applies some resistance
# but the motor does not move back if you move it.
example_motor.run_angle(500, 360, then=Stop.BRAKE)
wait(1000)

# This makes the motor coast freely after it stops.
example_motor.run_angle(500, 360, then=Stop.COAST)
wait(1000)
```

2.2.4 Stall examples

Running a motor until a mechanical endpoint

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port

# Initialize a motor on port A.
example_motor = Motor(Port.A)

# We'll use a speed of 200 deg/s in all our commands.
speed = 200

# Run the motor in reverse until it hits a mechanical stop.
# The duty_limit=30 setting means that it will apply only 30%
# of the maximum torque against the mechanical stop. This way,
# you don't push against it with too much force.
example_motor.run_until_stalled(-speed, duty_limit=30)

# Reset the angle to 0. Now whenever the angle is 0, you know
# that it has reached the mechanical endpoint.
example_motor.reset_angle(0)

# Now make the motor go back and forth in a loop.
# This will now work the same regardless of the
# initial motor angle, because we always start
# from the mechanical endpoint.
for count in range(10):
    example_motor.run_target(speed, 180)
    example_motor.run_target(speed, 90)
```

Centering a steering mechanism

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize a motor on port A.
example_motor = Motor(Port.A)

# Please have a look at the previous example first. This example
# finds two endspoints and then makes the middle the zero point.

# The run_until_stalled gives us the angle at which it stalled.
# We want to know this value for both endpoints.
left_end = example_motor.run_until_stalled(-200, duty_limit=30)
right_end = example_motor.run_until_stalled(200, duty_limit=30)

# We have just moved to the rightmost endstop. So, we can reset
# this angle to be half the distance between the two endpoints.
# That way, the middle corresponds to 0 degrees.
example_motor.reset_angle((right_end - left_end) / 2)

# From now on we can simply run towards zero to reach the middle.
example_motor.run_target(200, 0)

wait(1000)
```

2.2.5 Parallel movement examples

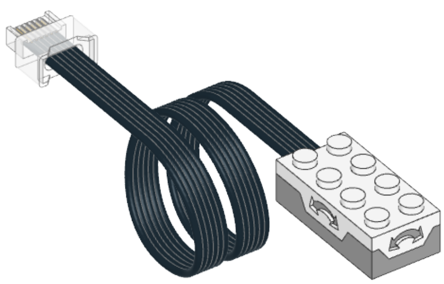
Using the `wait` argument to run motors in parallel

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port

# Initialize motors on port A and B.
track_motor = Motor(Port.A)
gripper_motor = Motor(Port.B)

# Make the track motor start moving,
# but don't wait for it to finish.
track_motor.run_angle(500, 360, wait=False)

# Now make the gripper motor rotate. This
# means they move at the same time.
gripper_motor.run_angle(200, 720)
```

Waiting for two parallel actions to complete

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize motors on port A and B.
track_motor = Motor(Port.A)
gripper_motor = Motor(Port.B)

# Make both motors perform an action with wait=False
track_motor.run_angle(500, 360, wait=False)
gripper_motor.run_angle(200, 720, wait=False)

# While one or both of the motors are not done yet,
# do something else. In this example, just wait.
while not track_motor.done() or not gripper_motor.done():
    wait(10)

print("Both motors are done!")
```

2.3 Tilt Sensor



class TiltSensor(*port*)

LEGO® Powered Up Tilt Sensor.

Parameters
port (Port) – Port to which the sensor is connected.

tilt() → Tuple[int, int]: deg

Measures the tilt relative to the horizontal plane.

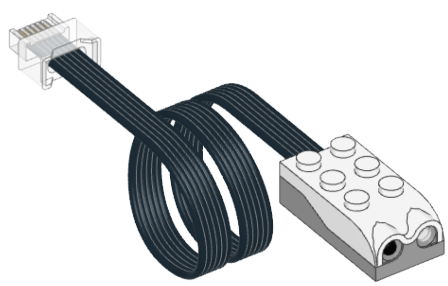
Returns
Tuple of pitch and roll angles.

2.3.1 Examples

Measuring pitch and roll

```
from pybricks.pupdevices import TiltSensor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize the sensor.
accel = TiltSensor(Port.A)

while True:
    # Read the tilt angles relative to the horizontal plane.
    pitch, roll = accel.tilt()

    # Print the values
    print("Pitch:", pitch, "Roll:", roll)

    # Wait some time so we can read what is printed.
    wait(100)
```

2.4 Infrared Sensor



class InfraredSensor(*port*)

LEGO® Powered Up Infrared Sensor.

Parameters

port (Port) – Port to which the sensor is connected.

distance() → int: %

Measures the relative distance between the sensor and an object using infrared light.

Returns

Distance ranging from 0% (closest) to 100% (farthest).

reflection() → int: %

Measures the reflection of a surface using an infrared light.

Returns

Measured reflection, ranging from 0% (no reflection) to 100% (high reflection).

count() → *int*

Counts the number of objects that have passed by the sensor.

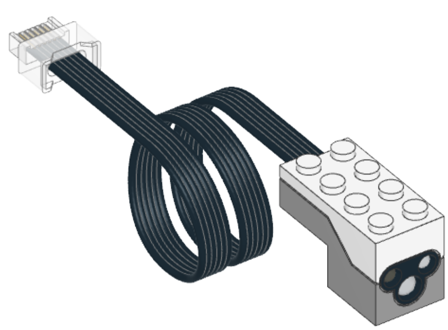
Returns
Number of objects counted.

2.4.1 Examples

Measuring distance, object count, and reflection

```
from pybricks.pupdevices import InfraredSensor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize the sensor.
ir = InfraredSensor(Port.A)

while True:
    # Read all the information we can get from this sensor.
    dist = ir.distance()
    count = ir.count()
    ref = ir.reflection()

    # Print the values
    print("Distance:", dist, "Count:", count, "Reflection:", ref)

    # Move the sensor around and move your hands in front
    # of it to see what happens to the values.

    # Wait some time so we can read what is printed.
    wait(200)
```

2.5 Color and Distance Sensor



class ColorDistanceSensor(*port*)

LEGO® Powered Up Color and Distance Sensor.

Parameters
port (Port) – Port to which the sensor is connected.

color() → *Color*

Scans the color of a surface.

You choose which colors are detected using the `detectable_colors()` method. By default, it detects `Color.RED`, `Color.YELLOW`, `Color.GREEN`, `Color.BLUE`, `Color.WHITE`, or `Color.NONE`.

Returns
Detected color.

reflection() → int: %

Measures how much a surface reflects the light emitted by the sensor.

Returns
Measured reflection, ranging from 0% (no reflection) to 100% (high reflection).

ambient() → int: %

Measures the ambient light intensity.

Returns
Ambient light intensity, ranging from 0% (dark) to 100% (bright).

distance() → int: %

Measures the relative distance between the sensor and an object using infrared light.

Returns
Distance ranging from 0% (closest) to 100% (farthest).

hsv() → *Color*

Scans the color of a surface.

This method is similar to `color()`, but it gives the full range of hue, saturation and brightness values, instead of rounding it to the nearest detectable color.

Returns
Measured color. The color is described by a hue (0–359), a saturation (0–100), and a brightness value (0–100).

detectable_colors(*colors*)

detectable_colors() → Collection[*Color*]

Configures which colors the `color()` method should detect.

Specify only colors that you wish to detect in your application. This way, the full-color measurements are rounded to the nearest desired color, and other colors are ignored. This improves reliability.

If you give no arguments, the currently chosen colors will be returned.

Parameters
colors (`list` *or* `tuple`) – List of *Color* objects: the colors that you want to detect. You can pick standard colors such as `Color.MAGENTA`, or provide your own colors like `Color(h=348, s=96, v=40)` for even better results. You measure your own colors with the `hsv()` method.

Built-in light

This sensor has a built-in light. You can make it red, green, blue, or turn it off. If you use the sensor to measure something afterwards, the light automatically turns back on at the default color for that sensing method.

light.**on**(*color*)

Turns on the light at the specified color.

Parameters
color (`Color`) – Color of the light.

`light.off()`

Turns off the light.

2.5.1 Examples

Measuring color

```
from pybricks.pupdevices import ColorDistanceSensor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorDistanceSensor(Port.A)

while True:
    # Read the color.
    color = sensor.color()

    # Print the measured color.
    print(color)

    # Move the sensor around and see how
    # well you can detect colors.

    # Wait so we can read the value.
    wait(100)
```

Waiting for a color

```
from pybricks.pupdevices import ColorDistanceSensor
from pybricks.parameters import Port, Color
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorDistanceSensor(Port.A)


# This is a function that waits for a desired color.
def wait_for_color(desired_color):
    # While the color is not the desired color, we keep waiting.
    while sensor.color() != desired_color:
        wait(20)


# Now we use the function we just created above.
while True:

    # Here you can make your train/vehicle go forward.

    print("Waiting for red ...")
```

(continues on next page)

(continued from previous page)

```
    wait_for_color(Color.RED)

    # Here you can make your train/vehicle go backward.

    print("Waiting for blue ...")
    wait_for_color(Color.BLUE)
```

Measuring distance and blinking the light

```
from pybricks.pupdevices import ColorDistanceSensor
from pybricks.parameters import Port, Color
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorDistanceSensor(Port.A)

# Repeat forever.
while True:

    # If the sensor sees an object nearby.
    if sensor.distance() <= 40:

        # Then blink the light red/blue 5 times.
        for i in range(5):
            sensor.light.on(Color.RED)
            wait(30)
            sensor.light.on(Color.BLUE)
            wait(30)
    else:
        # If the sensor sees nothing
        # nearby, just wait briefly.
        wait(10)
```

Reading hue, saturation, value

```
from pybricks.pupdevices import ColorDistanceSensor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorDistanceSensor(Port.A)

while True:
    # The standard color() method always "rounds" the
    # measurement to the nearest "whole" color.
    # That's useful for most applications.

    # But you can get the original hue, saturation,
    # and value without "rounding", as follows:
```

(continues on next page)

(continued from previous page)

```
    color = sensor.hsv()

    # Print the results.
    print(color)

    # Wait so we can read the value.
    wait(500)
```

Changing the detectable colors

By default, the sensor is configured to detect red, yellow, green, blue, white, or no color, which suits many applications.

For better results in your application, you can measure your desired colors in advance, and tell the sensor to look only for those colors. Be sure to measure them at the **same distance and light conditions** as in your final application. Then you'll get very accurate results even for colors that are otherwise hard to detect.

```
from pybricks.pupdevices import ColorDistanceSensor
from pybricks.parameters import Port, Color
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorDistanceSensor(Port.A)

# First, decide which objects you want to detect, and measure their HSV values.
# You can do that with the hsv() method as shown in the previous example.
#
# Use your measurements to override the default colors, or add new colors:
Color.GREEN = Color(h=132, s=94, v=26)
Color.MAGENTA = Color(h=348, s=96, v=40)
Color.BROWN = Color(h=17, s=78, v=15)
Color.RED = Color(h=359, s=97, v=39)

# Put your colors in a list or tuple.
my_colors = (Color.GREEN, Color.MAGENTA, Color.BROWN, Color.RED, Color.NONE)

# Save your colors.
sensor.detectable_colors(my_colors)

# color() works as usual, but now it returns one of your specified colors.
while True:
    color = sensor.color()

    # Print the color.
    print(color)

    # Check which one it is.
    if color == Color.MAGENTA:
        print("It works!")

    # Wait so we can read it.
    wait(100)
```

2.6 Power Functions

The *ColorDistanceSensor* can send infrared signals to control Power Functions infrared receivers. You can use this technique to control medium, large, extra large, and train motors. The infrared range is limited to about 30 cm, depending on the angle and ambient conditions.

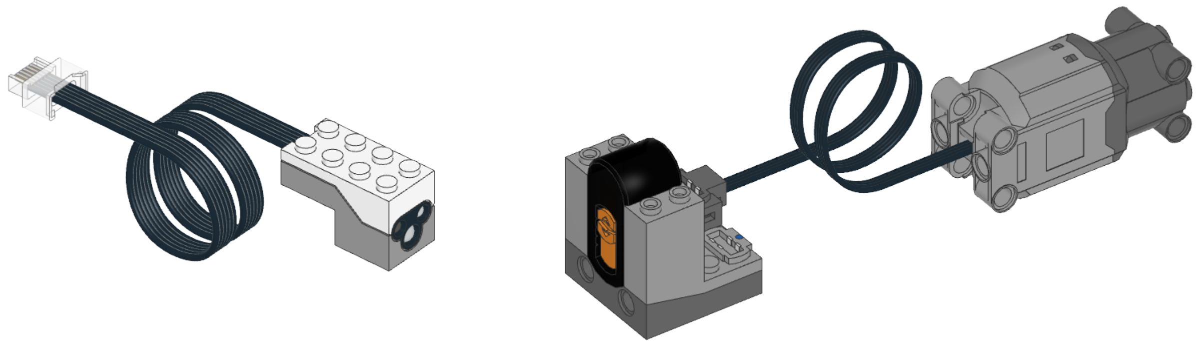


Figure 2.3: Powered Up *ColorDistanceSensor* (left), Power Functions infrared receiver (middle), and a Power Functions motor (right). Here, the receiver uses channel 1 with a motor on the red port.

class PFMotor(*sensor*, *channel*, *color*, *positive_direction=Direction.CLOCKWISE*)

Control Power Functions motors with the infrared functionality of the *ColorDistanceSensor*.

Parameters

- **sensor** (ColorDistanceSensor) – Sensor object.
- **channel** (int) – Channel number of the receiver: 1, 2, 3, or 4.
- **color** (Color) – Color marker on the receiver: *Color.BLUE* or *Color.RED*
- **positive_direction** (Direction) – Which direction the motor should turn when you give a positive duty cycle value.

dc(*duty*)

Rotates the motor at a given duty cycle (also known as "power").

Parameters
duty (Number, %) – The duty cycle (-100.0 to 100).

stop()

Stops the motor and lets it spin freely.

The motor gradually stops due to friction.

brake()

Passively brakes the motor.

The motor stops due to friction, plus the voltage that is generated while the motor is still moving.

2.6.1 Examples

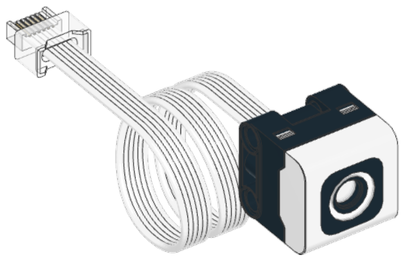
Control a Power Functions motor

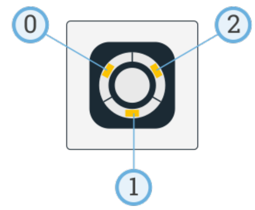
```
from pybricks.pupdevices import ColorDistanceSensor, PFMotor
from pybricks.parameters import Port, Color
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorDistanceSensor(Port.B)

# Initialize a motor on channel 1, on the red output.
motor = PFMotor(sensor, 1, Color.RED)

# Rotate and then stop.
motor.dc(100)
wait(1000)
motor.stop()
wait(1000)

# Rotate the other way at half speed, and then stop.
motor.dc(-50)
wait(1000)
motor.stop()
```

Controlling multiple Power Functions motors

```
from pybricks.pupdevices import ColorDistanceSensor, PFMotor
from pybricks.parameters import Port, Color, Direction
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorDistanceSensor(Port.B)

# You can use multiple motors on different channels.
arm = PFMotor(sensor, 1, Color.BLUE)
wheel = PFMotor(sensor, 4, Color.RED, Direction.COUNTERCLOCKWISE)

# Accelerate both motors. Only these values are available.
# Other values will be rounded down to the nearest match.
for duty in [15, 30, 45, 60, 75, 90, 100]:
    arm.dc(duty)
    wheel.dc(duty)
    wait(1000)

# To make the signal more reliable, there is a short
# pause between commands. So, they change speed and
# stop at a slightly different time.

# Brake both motors.
arm.brake()
wheel.brake()
```

2.7 Color Sensor





class **ColorSensor**(*port*)

LEGO® SPIKE Color Sensor.

Parameters
port (Port) – Port to which the sensor is connected.

color(*surface=True*) → *Color*

Scans the color of a surface or an external light source.

You choose which colors are detected using the `detectable_colors()` method. By default, it detects `Color.RED`, `Color.YELLOW`, `Color.GREEN`, `Color.BLUE`, `Color.WHITE`, or `Color.NONE`.

Parameters
surface (bool) – Choose `true` to scan the color of objects and surfaces. Choose `false` to scan the color of screens and other external light sources.

Returns
Detected color.`

reflection() → int: %

Measures how much a surface reflects the light emitted by the sensor.

Returns
Measured reflection, ranging from 0% (no reflection) to 100% (high reflection).

ambient() → int: %

Measures the ambient light intensity.

Returns
Ambient light intensity, ranging from 0% (dark) to 100% (bright).

Advanced color sensing

hsv(*surface=True*) → *Color*

Scans the color of a surface or an external light source.

This method is similar to `color()`, but it gives the full range of hue, saturation and brightness values, instead of rounding it to the nearest detectable color.

Parameters
surface (bool) – Choose `true` to scan the color of objects and surfaces. Choose `false` to scan the color of screens and other external light sources.

Returns
Measured color. The color is described by a hue (0–359), a saturation (0–100), and a brightness value (0–100).

detectable_colors(*colors*)

detectable_colors() → Collection[*Color*]

Configures which colors the `color()` method should detect.

Specify only colors that you wish to detect in your application. This way, the full-color measurements are rounded to the nearest desired color, and other colors are ignored. This improves reliability.

If you give no arguments, the currently chosen colors will be returned.

Parameters

colors (`list or tuple`) – List of *Color* objects: the colors that you want to detect. You can pick standard colors such as `Color.MAGENTA`, or provide your own colors like `Color(h=348, s=96, v=40)` for even better results. You measure your own colors with the `hsv()` method.

Built-in lights

This sensor has 3 built-in lights. You can adjust the brightness of each light. If you use the sensor to measure something, the lights will be turned on or off as needed for the measurement.

`lights.`**on**(*brightness*)

Turns on the lights at the specified brightness.

Parameters

brightness (`Number or tuple, %`) – Use a single value to set the brightness of all lights at the same time. Use a tuple of three values to set the brightness of each light individually.

`lights.`**off**()

Turns off all the lights.

2.7.1 Examples

Measuring color and reflection

```
from pybricks.pupdevices import ColorSensor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorSensor(Port.A)

while True:
    # Read the color and reflection
    color = sensor.color()
    reflection = sensor.reflection()

    # Print the measured color and reflection.
    print(color, reflection)

    # Move the sensor around and see how
    # well you can detect colors.

    # Wait so we can read the value.
    wait(100)
```

Waiting for a color

```
from pybricks.pupdevices import ColorSensor
from pybricks.parameters import Port, Color
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorSensor(Port.A)


# This is a function that waits for a desired color.
def wait_for_color(desired_color):
    # While the color is not the desired color, we keep waiting.
    while sensor.color() != desired_color:
        wait(20)


# Now we use the function we just created above.
while True:

    # Here you can make your train/vehicle go forward.

    print("Waiting for red ...")
    wait_for_color(Color.RED)

    # Here you can make your train/vehicle go backward.

    print("Waiting for blue ...")
    wait_for_color(Color.BLUE)
```

Reading *reflected* hue, saturation, and value

```
from pybricks.pupdevices import ColorSensor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorSensor(Port.A)

while True:
    # The standard color() method always "rounds" the
    # measurement to the nearest "whole" color.
    # That's useful for most applications.

    # But you can get the original hue, saturation,
    # and value without "rounding", as follows:
    color = sensor.hsv()

    # Print the results.
    print(color)
```

(continues on next page)

(continued from previous page)

```
    # Wait so we can read the value.
    wait(500)
```

Changing the detectable colors

By default, the sensor is configured to detect red, yellow, green, blue, white, or no color, which suits many applications.

For better results in your application, you can measure your desired colors in advance, and tell the sensor to look only for those colors. Be sure to measure them at the **same distance and light conditions** as in your final application. Then you'll get very accurate results even for colors that are otherwise hard to detect.

```
from pybricks.pupdevices import ColorSensor
from pybricks.parameters import Port, Color
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorSensor(Port.A)

# First, decide which objects you want to detect, and measure their HSV values.
# You can do that with the hsv() method as shown in the previous example.
#
# Use your measurements to override the default colors, or add new colors:
Color.GREEN = Color(h=132, s=94, v=26)
Color.MAGENTA = Color(h=348, s=96, v=40)
Color.BROWN = Color(h=17, s=78, v=15)
Color.RED = Color(h=359, s=97, v=39)

# Put your colors in a list or tuple.
my_colors = (Color.GREEN, Color.MAGENTA, Color.BROWN, Color.RED, Color.NONE)

# Save your colors.
sensor.detectable_colors(my_colors)

# color() works as usual, but now it returns one of your specified colors.
while True:
    color = sensor.color()

    # Print the color.
    print(color)

    # Check which one it is.
    if color == Color.MAGENTA:
        print("It works!")

    # Wait so we can read it.
    wait(100)
```

Reading *ambient* hue, saturation, value, and color

```
from pybricks.pupdevices import ColorSensor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorSensor(Port.A)

# Repeat forever.
while True:

    # Get the ambient color values. Instead of scanning the color of a surface,
    # this lets you scan the color of light sources like lamps or screens.
    hsv = sensor.hsv(surface=False)
    color = sensor.color(surface=False)

    # Get the ambient light intensity.
    ambient = sensor.ambient()

    # Print the measurements.
    print(hsv, color, ambient)

    # Point the sensor at a computer screen or colored light. Watch the color.
    # Also, cover the sensor with your hands and watch the ambient value.

    # Wait so we can read the printed line
    wait(100)
```

Blinking the built-in lights

```
from pybricks.pupdevices import ColorSensor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorSensor(Port.A)

# Repeat forever.
while True:

    # Turn on one light at a time, at half the brightness.
    # Do this for all 3 lights and repeat that 5 times.
    for i in range(5):
        sensor.lights.on([50, 0, 0])
        wait(100)
        sensor.lights.on([0, 50, 0])
        wait(100)
        sensor.lights.on([0, 0, 50])
        wait(100)

```

(continues on next page)

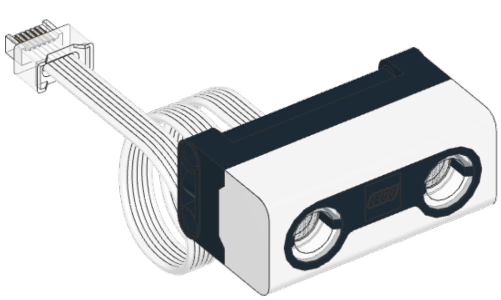
(continued from previous page)

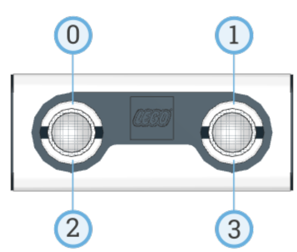
```
    # Turn all lights on at maximum brightness.
    sensor.lights.on(100)
    wait(500)

    # Turn all lights off.
    sensor.lights.off()
    wait(500)
```

Turning off the lights when the program ends

```
from pybricks.parameters import Port
from pybricks.pupdevices import ColorSensor
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorSensor(Port.A)


def main():
    # Run the main code.
    while True:
        print(sensor.color())
        wait(500)


# Wrap the main code in try/finally so that the cleanup code always runs
# when the program ends, even if an exception was raised.
try:
    main()
finally:
    # The cleanup code goes here.
    print("Cleaning up.")
    sensor.lights.off()
```

2.8 Ultrasonic Sensor





class UltrasonicSensor(*port*)

LEGO® SPIKE Color Sensor.

Parameters
port (Port) – Port to which the sensor is connected.

distance() → int: mm

Measures the distance between the sensor and an object using ultrasonic sound waves.

Returns
Measured distance. If no valid distance was measured, it returns 2000 mm.

presence() → *bool*

Checks for the presence of other ultrasonic sensors by detecting ultrasonic sounds.

Returns
`True` if ultrasonic sounds are detected, `False` if not.

Built-in lights

This sensor has 4 built-in lights. You can adjust the brightness of each light.

lights.**on**(*brightness*)

Turns on the lights at the specified brightness.

Parameters
brightness (Number *or* tuple, *%*) – Use a single value to set the brightness of all lights at the same time. Use a tuple of four values to set the brightness of each light individually. The order of the lights is shown in the image above.

lights.**off**()

Turns off all the lights.

2.8.1 Examples

Measuring distance and switching on the lights

```
from pybricks.pupdevices import UltrasonicSensor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize the sensor.
eyes = UltrasonicSensor(Port.A)

while True:
    # Print the measured distance.
    print(eyes.distance())

    # If an object is detected closer than 500mm:
    if eyes.distance() < 500:
        # Turn the lights on.
        eyes.lights.on(100)
    else:
        # Turn the lights off.
        eyes.lights.off()
```

(continues on next page)

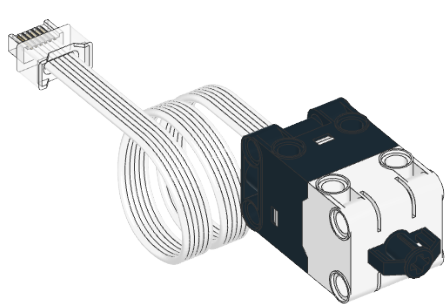
(continued from previous page)

```
    # Wait some time so we can read what is printed.
    wait(100)
```

Gradually change the brightness of the lights

```
from pybricks.pupdevices import UltrasonicSensor
from pybricks.parameters import Port
from pybricks.tools import wait, StopWatch

from umath import pi, sin

# Initialize the sensor.
eyes = UltrasonicSensor(Port.A)

# Initialize a timer.
watch = StopWatch()

# We want one full light cycle to last three seconds.
PERIOD = 3000

while True:
    # The phase is where we are in the unit circle now.
    phase = watch.time() / PERIOD * 2 * pi

    # Each light follows a sine wave with a mean of 50, with an amplitude of 50.
    # We offset this sine wave by 90 degrees for each light, so that all the
    # lights do something different.
    brightness = [sin(phase + offset * pi / 2) * 50 + 50 for offset in range(4)]

    # Set the brightness values for all lights.
    eyes.lights.on(brightness)

    # Wait some time.
    wait(50)
```

2.9 Force Sensor



class ForceSensor(*port*)

LEGO® SPIKE Force Sensor.

Parameters
port (Port) – Port to which the sensor is connected.

force() → float: N

Measures the force exerted on the sensor.

Returns
Measured force (up to approximately 10.00 N).

distance() → float: mm

Measures by how much the sensor button has moved.

Returns
Movement up to approximately 8.00 mm.

pressed(*force=3*) → *bool*

Checks if the sensor button is pressed.

Parameters
force (Number, *N*) – Minimum force to be considered pressed.

Returns
`True` if the sensor is pressed, `False` if it is not.

touched() → *bool*

Checks if the sensor is touched.

This is similar to *pressed()*, but it detects slight movements of the button even when the measured force is still considered zero.

Returns
`True` if the sensor is touched or pressed, `False` if it is not.

2.9.1 Examples

Measuring force and movement

```
from pybricks.pupdevices import ForceSensor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize the sensor.
button = ForceSensor(Port.A)

while True:
    # Read all the information we can get from this sensor.
    force = button.force()
    dist = button.distance()
    press = button.pressed()
    touch = button.touched()

    # Print the values
    print("Force", force, "Dist:", dist, "Pressed:", press, "Touched:", touch)

    # Push the sensor button see what happens to the values.
```

(continues on next page)

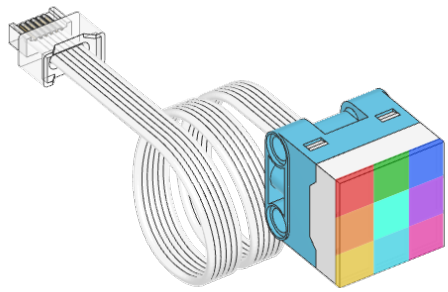
(continued from previous page)

```
    # Wait some time so we can read what is printed.
    wait(200)
```

Measuring peak force

```
from pybricks.pupdevices import ForceSensor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize the sensor.
button = ForceSensor(Port.A)


# This function waits until the button is pushed. It keeps track of the maximum
# detected force until the button is released. Then it returns the maximum.
def wait_for_force():

    # Wait for a force, by doing nothing for as long the force is nearly zero.
    print("Waiting for force.")
    while button.force() <= 0.1:
        wait(10)

    # Now we wait for the release, by waiting for the force to be zero again.
    print("Waiting for release.")

    # While we wait for that to happen, we keep reading the force and remember
    # the maximum force. We do this by initializing the maximum at 0, and
    # updating it each time we detect a bigger force.
    maximum = 0
    force = 10
    while force > 0.1:
        # Read the force.
        force = button.force()

        # Update the maximum if the measured force is larger.
        if force > maximum:
            maximum = force

        # Wait and then measure again.
        wait(10)

    # Return the maximum force.
    return maximum


# Keep waiting for the sensor button to be pushed. When it is, display
# the peak force and repeat.
while True:
    peak = wait_for_force()
    print("Released. Peak force: {0} N\n".format(peak))
```

2.10 Color Light Matrix



class ColorLightMatrix(*port*)

LEGO® SPIKE 3x3 Color Light Matrix.

Parameters
port (Port) – Port to which the device is connected.

on(*colors*)

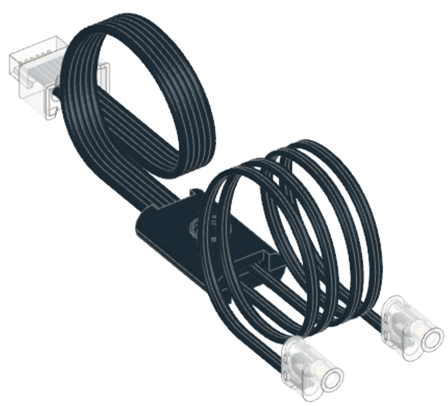
Turns the lights on.

Parameters
colors (Color *or* list) – If a single *Color* is given, then all 9 lights are set to that color. If a list of colors is given, then each light is set to that color.

off()

Turns all lights off.

2.11 Light



class Light(*port*)

LEGO® Powered Up Light.

Parameters
port (Port) – Port to which the device is connected.

on(*brightness=100*)

Turns on the light at the specified brightness.

Parameters
brightness (Number, %) – Brightness of the light.

off()
Turns off the light.

2.11.1 Examples

Making the light blink

```
from pybricks.pupdevices import Light
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize the light.
light = Light(Port.A)

# Blink the light forever.
while True:
    # Turn the light on at 100% brightness.
    light.on(100)
    wait(500)

    # Turn the light off.
    light.off()
    wait(500)
```

Gradually change the brightness

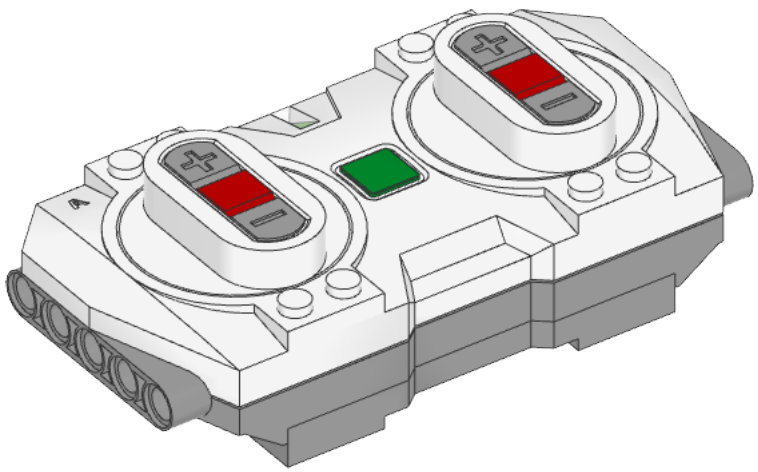
```
from pybricks.pupdevices import Light
from pybricks.parameters import Port
from pybricks.tools import wait, StopWatch

from umath import pi, cos

# Initialize the light and a StopWatch.
light = Light(Port.A)
watch = StopWatch()

# Cosine pattern properties.
PERIOD = 2000
MAX = 100

# Make the brightness fade in and out.
while True:
    # Get phase of the cosine.
    phase = watch.time() / PERIOD * 2 * pi

    # Evaluate the brightness.
    brightness = (0.5 - 0.5 * cos(phase)) * MAX
```

(continues on next page)

(continued from previous page)

```
    # Set light brightness and wait a bit.
    light.on(brightness)
    wait(10)
```

2.12 Remote Control



class Remote(*name=None*, *timeout=10000*)

LEGO® Powered Up Bluetooth Remote Control.

When you instantiate this class, the hub will search for a remote and connect automatically.

The remote must be on and ready for a connection, as indicated by a white blinking light.

Parameters

- **name** (`str`) – Bluetooth name of the remote. If no name is given, the hub connects to the first remote that it finds.
- **timeout** (`Number, ms`) – How long to search for the remote.

name(*name*)

name() → *str*

Sets or gets the Bluetooth name of the remote.

Parameters
name (`str`) – New Bluetooth name of the remote. If no name is given, this method returns the current name.

light.**on**(*color*)

Turns on the light at the specified color.

Parameters
color (`Color`) – Color of the light.

light.**off**()

Turns off the light.

`buttons.pressed()` → Collection[*Button*]

Checks which buttons are currently pressed.

Returns
Set of pressed buttons.

2.12.1 Examples

Checking which buttons are pressed

```
from pybricks.pupdevices import Remote
from pybricks.parameters import Button
from pybricks.tools import wait

# Connect to the remote.
my_remote = Remote()

while True:
    # Check which buttons are pressed.
    pressed = my_remote.buttons.pressed()

    # Show the result.
    print("pressed:", pressed)

    # Check a specific button.
    if Button.CENTER in pressed:
        print("You pressed the center button!")

    # Wait so we can see the result.
    wait(100)
```

Changing the remote light color

```
from pybricks.pupdevices import Remote
from pybricks.parameters import Color
from pybricks.tools import wait

# Connect to the remote.
remote = Remote()

while True:
    # Set the color to red.
    remote.light.on(Color.RED)
    wait(1000)

    # Set the color to blue.
    remote.light.on(Color.BLUE)
    wait(1000)
```

Changing the light color using the buttons

```
from pybricks.pupdevices import Remote
from pybricks.parameters import Button, Color


def button_to_color(buttons):

    # Return a color depending on the button.
    if Button.LEFT_PLUS in buttons:
        return Color.RED
    if Button.LEFT_MINUS in buttons:
        return Color.GREEN
    if Button.LEFT in buttons:
        return Color.ORANGE
    if Button.RIGHT_PLUS in buttons:
        return Color.BLUE
    if Button.RIGHT_MINUS in buttons:
        return Color.YELLOW
    if Button.RIGHT in buttons:
        return Color.CYAN
    if Button.CENTER in buttons:
        return Color.VIOLET

    # Return no color by default.
    return Color.NONE


# Connect to the remote.
remote = Remote()

while True:
    # Wait until a button is pressed.
    pressed = ()
    while not pressed:
        pressed = remote.buttons.pressed()

    # Convert button code to color.
    color = button_to_color(pressed)

    # Set the remote light color.
    remote.light.on(color)

    # Wait until all buttons are released.
    while pressed:
        pressed = remote.buttons.pressed()
```

Using the timeout setting

You can use the `timeout` argument to change for how long the hub searches for the remote. If you choose `None`, it will search forever.

```
from pybricks.pupdevices import Remote

# Connect to any remote. Search forever until we find one.
my_remote = Remote(timeout=None)

print("Connected!")
```

If the remote was not found within the specified `timeout`, an *OSError* is raised. You can catch this exception to run other code if the remote is not available.

```
from pybricks.pupdevices import Remote

try:
    # Search for a remote for 5 seconds.
    my_remote = Remote(timeout=5000)

    print("Connected!")

    # Here you can write code that uses the remote.

except OSError:

    print("Could not find the remote.")

    # Here you can make your robot do something
    # without the remote.
```

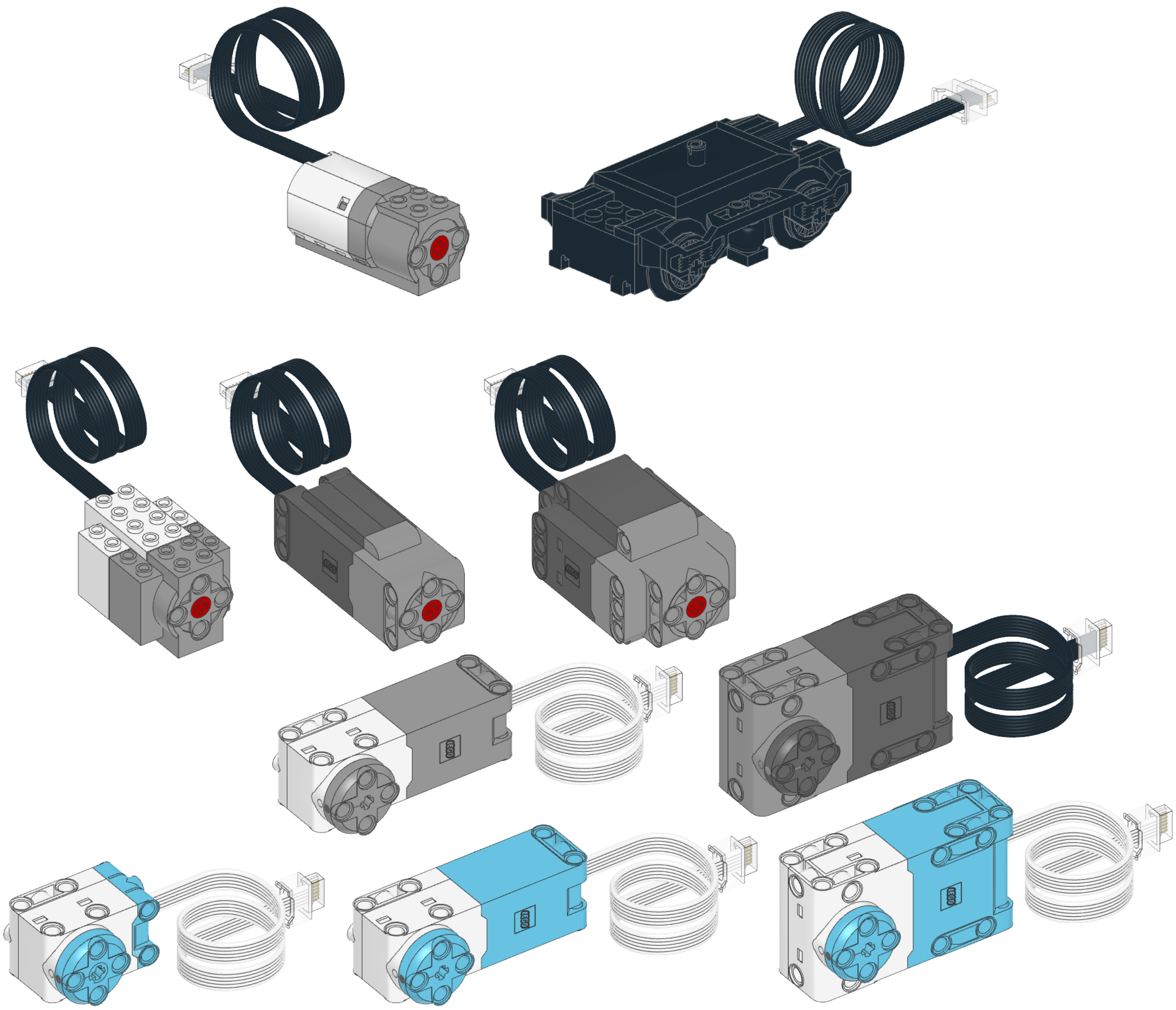
Changing the name of the remote

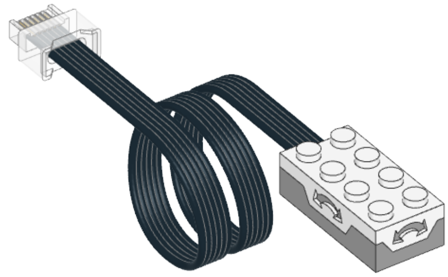
You can change the Bluetooth name of the remote. The factory default name is `Handset`.

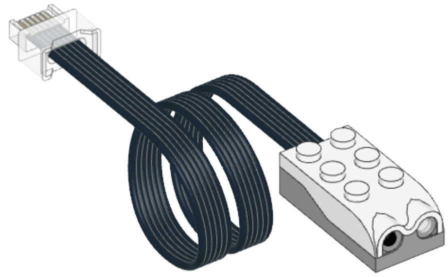
```
from pybricks.pupdevices import Remote

# Connect to any remote.
my_remote = Remote()

# Print the current name of the remote.
print(my_remote.name())

# Choose a new name.
my_remote.name("truck2")

print("Done!")
```

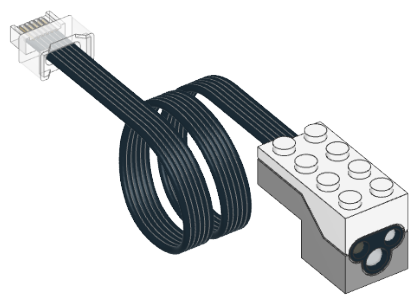
You can specify this name when connecting to the remote. This lets you pick the right one if multiple remotes are nearby.

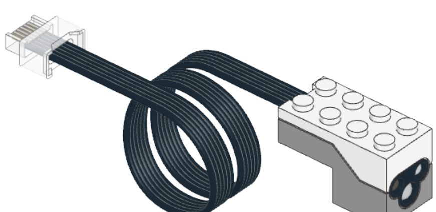
```
from pybricks.pupdevices import Remote
from pybricks.tools import wait

# Connect to a remote called truck2.
truck_remote = Remote("truck2", timeout=None)

print("Connected!")

wait(2000)
```

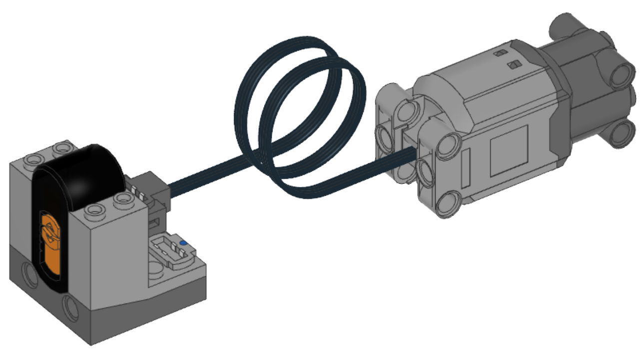


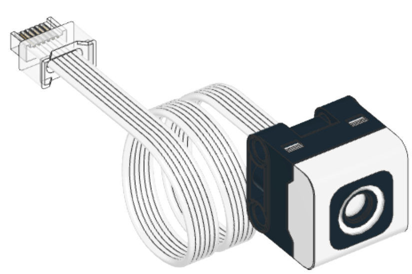


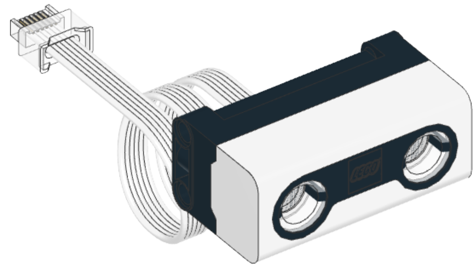


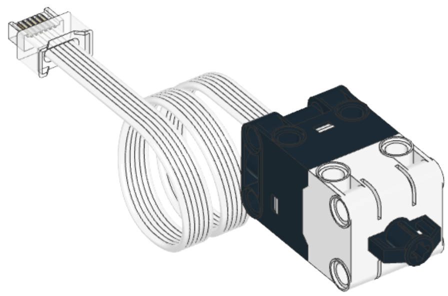


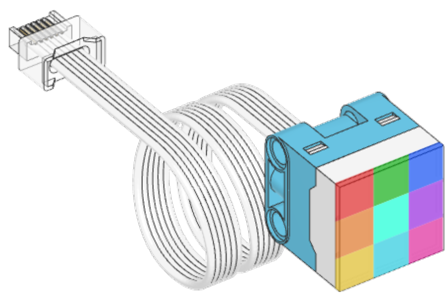


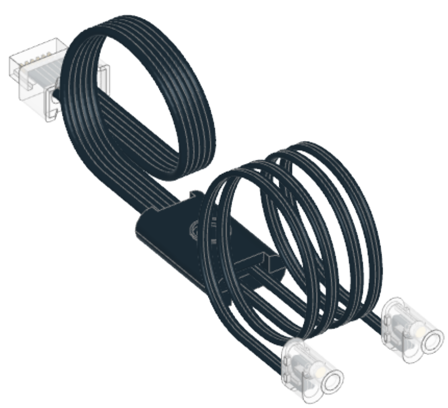


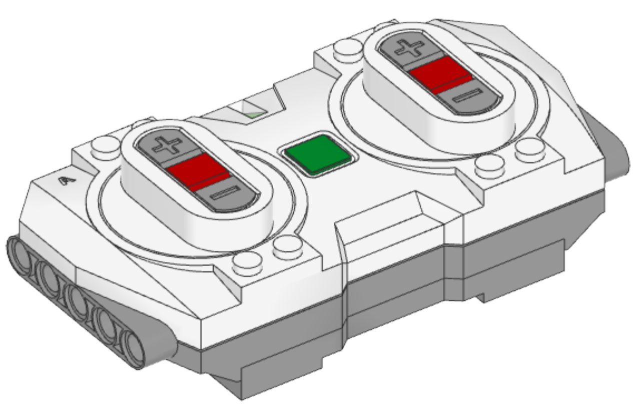










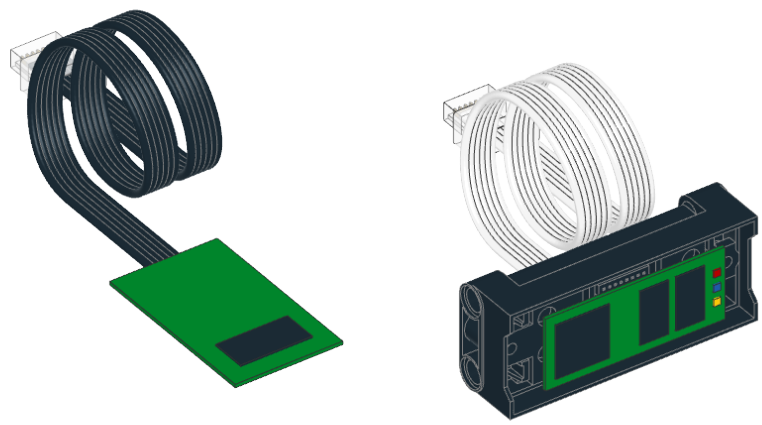


CHAPTER

THREE

IODEVICES – CUSTOM DEVICES

3.1 Powered Up Device



class PUPDevice(*port*)

Powered Up motor or sensor.

Parameters

port (Port) – Port to which the device is connected.

info() → Dict

Gets information about the device.

Returns

Dictionary with information, such as the device `id`.

read(*mode*) → Tuple

Reads values from a given mode.

Parameters

mode (int) – Device mode.

Returns

Values read from the sensor.

write(*mode*, *data*)

Writes values to the sensor. Only selected sensors and modes support this.

Parameters

- **mode** (int) – Device mode.

- **data** (tuple) – Values to be written.

3.1.1 Examples

Detecting devices

```
from pybricks.iodevices import PUPDevice
from pybricks.parameters import Port
from uerrno import ENODEV

# Dictionary of device identifiers along with their name.
device_names = {
    34: "Wedo 2.0 Tilt Sensor",
    35: "Wedo 2.0 Infrared Sensor",
    37: "BOOST Color Distance Sensor",
    38: "BOOST Interactive Motor",
    46: "Technic Large Motor",
    47: "Technic Extra Large Motor",
    48: "SPIKE Medium Angular Motor",
    49: "SPIKE Large Angular Motor",
    61: "SPIKE Color Sensor",
    62: "SPIKE Ultrasonic Sensor",
    63: "SPIKE Force Sensor",
    75: "Technic Medium Angular Motor",
    76: "Technic Large Angular Motor",
}

# Make a list of known ports.
ports = [Port.A, Port.B]

# On hubs that support it, add more ports.
try:
    ports.append(Port.C)
    ports.append(Port.D)
except AttributeError:
    pass

# On hubs that support it, add more ports.
try:
    ports.append(Port.E)
    ports.append(Port.F)
except AttributeError:
    pass

# Go through all available ports.
for port in ports:

    # Try to get the device, if it is attached.
    try:
        device = PUPDevice(port)
    except OSError as ex:
        if ex.args[0] == ENODEV:
```

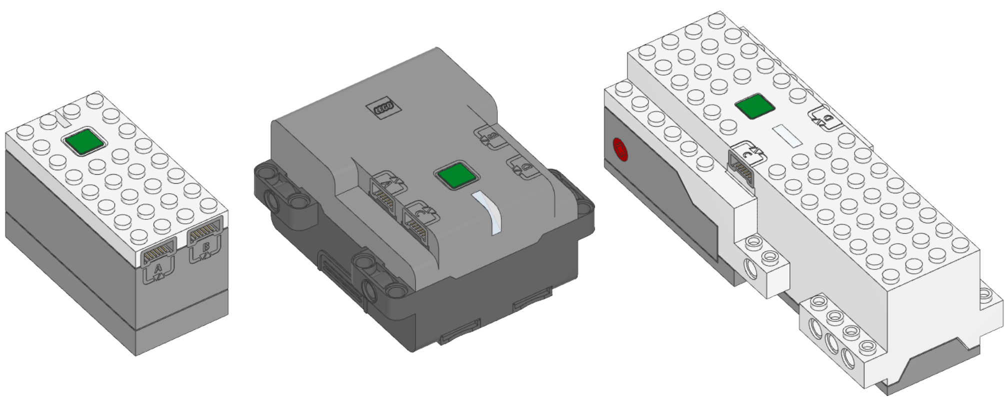
(continues on next page)

(continued from previous page)

```
            # No device found on this port.
            print(port, ": ---")
            continue
        else:
            raise

    # Get the device id
    id = device.info()["id"]

    # Look up the name.
    try:
        print(port, ":", device_names[id])
    except KeyError:
        print(port, ":", "Unknown device with ID", id)
```

3.2 LEGO Wireless Protocol v3 device

Warning: This is an experimental class. It has not been well tested and may be changed in future.



class LWP3Device(*hub_kind*, *name=None*, *timeout=10000*)

Connects to a hub running official LEGO firmware using the LEGO Wireless Protocol v3

Parameters

- **hub_kind** (int) – The hub type identifier of the hub to connect to.
- **name** (str) – The name of the hub to connect to or None to connect to any hub.
- **timeout** (int) – The time, in milliseconds, to wait for a connection before raising an exception.

name(*name*)

name() → *str*

Sets or gets the Bluetooth name of the device.

Parameters

name (str) – New Bluetooth name of the device. If no name is given, this method returns the current name.

write(*buf*)

Sends a message to the remote hub.

Parameters

buf (bytes) – The raw binary message to send.

read() → *bytes*

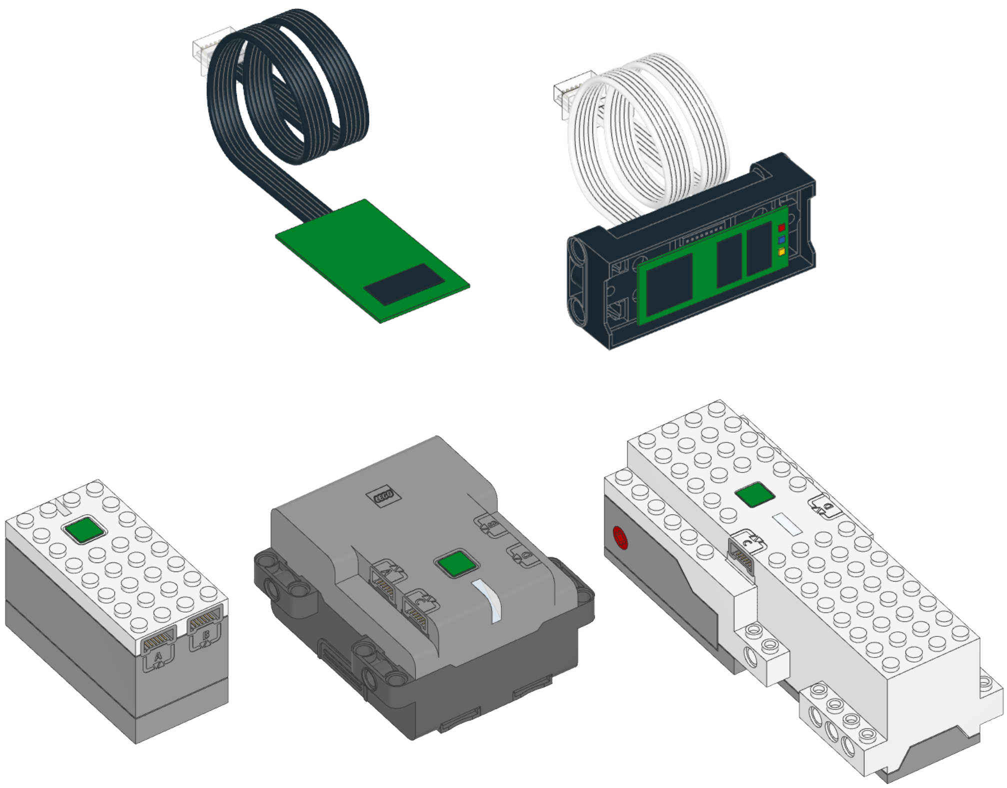
Retrieves the most recent message received from the remote hub.

If a message has not been received since the last read, the method will block until a message is received.

Returns

The raw binary message.

This module has classes for generic and custom input/output devices.



CHAPTER

FOUR

PARAMETERS – PARAMETERS AND CONSTANTS

Constant parameters/arguments for the Pybricks API.

4.1 Button

class Button

Buttons on a hub or remote.

LEFT_DOWN

LEFT_MINUS

DOWN

RIGHT_DOWN

RIGHT_MINUS

LEFT

CENTER

RIGHT

LEFT_UP

LEFT_PLUS

UP

BEACON

RIGHT_UP

RIGHT_PLUS

4.2 Color

`class Color(h, s=100, v=100)`

Light or surface color.

Parameters

- **h** (Number, *deg*) – Hue.
- **s** (Number, *%*) – Saturation.
- **v** (Number, *%*) – Brightness value.

Saturated colors

These colors have maximum saturation and brightness value. They differ only in hue.

`RED: Color = Color(h=0, s=100, v=100)`

`ORANGE: Color = Color(h=30, s=100, v=100)`

`YELLOW: Color = Color(h=60, s=100, v=100)`

`GREEN: Color = Color(h=120, s=100, v=100)`

`CYAN: Color = Color(h=180, s=100, v=100)`

`BLUE: Color = Color(h=240, s=100, v=100)`

`VIOLET: Color = Color(h=270, s=100, v=100)`

`MAGENTA: Color = Color(h=300, s=100, v=100)`

Unsaturated colors

These colors have zero hue and saturation. They differ only in brightness value.

When detecting these colors using sensors, their values depend a lot on the distance to the object. If the distance between the sensor and the object is not constant in your robot, it is better to use only one of these colors in your programs.

`WHITE: Color = Color(h=0, s=0, v=100)`

`GRAY: Color = Color(h=0, s=0, v=50)`

`BLACK: Color = Color(h=0, s=0, v=10)`

This represents dark objects that still reflect a very small amount of light.

`NONE: Color = Color(h=0, s=0, v=0)`

This is total darkness, with no reflection or light at all.

Making your own colors

This example shows the basics of color properties, and how to define new colors.

```
from pybricks.parameters import Color

# You can print colors. Colors may be obtained from the Color class, or
# from sensors that return color measurements.
print(Color.RED)

# You can read hue, saturation, and value properties.
print(Color.RED.h, Color.RED.s, Color.RED.v)

# You can make your own colors. Saturation and value are 100 by default.
my_green = Color(h=125)
my_dark_green = Color(h=125, s=80, v=30)

# When you print custom colors, you see exactly how they were defined.
print(my_dark_green)

# You can also add colors to the builtin colors.
Color.MY_DARK_BLUE = Color(h=235, s=80, v=30)

# When you add them like this, printing them only shows its name. But you can
# still read h, s, v by reading its attributes.
print(Color.MY_DARK_BLUE)
print(Color.MY_DARK_BLUE.h, Color.MY_DARK_BLUE.s, Color.MY_DARK_BLUE.v)
```

This example shows more advanced use cases of the `Color` class.

```
from pybricks.parameters import Color

# Two colors are equal if their h, s, and v attributes are equal.
if Color.BLUE == Color(240, 100, 100):
    print("Yes, these colors are the same.")

# You can scale colors to change their brightness value.
red_dark = Color.RED * 0.5

# You can shift colors to change their hue.
red_shifted = Color.RED >> 30

# Colors are immutable, so you can't change h, s, or v of an existing object.
try:
    Color.GREEN.h = 125
except AttributeError:
    print("Sorry, can't change the hue of an existing color object!")

# But you can override builtin colors by defining a whole new color.
Color.GREEN = Color(h=125)

# You can access and store colors as class attributes, or as a dictionary.
print(Color.BLUE)
print(Color["BLUE"])
```

(continues on next page)

(continued from previous page)

```
print(Color["BLUE"] is Color.BLUE)
print(Color)
print([c for c in Color])

# This allows you to update existing colors in a loop.
for name in ("BLUE", "RED", "GREEN"):
    Color[name] = Color(1, 2, 3)
```

4.3 Direction

class Direction

Rotational direction for positive speed or angle values.

CLOCKWISE

A positive speed value should make the motor move clockwise.

COUNTERCLOCKWISE

A positive speed value should make the motor move counterclockwise.

`positive_direction =`	Positive speed:	Negative speed:
`Direction.CLOCKWISE`	clockwise	counterclockwise
`Direction.COUNTERCLOCKWISE`	counterclockwise	clockwise

In general, clockwise is defined by **looking at the motor shaft, just like looking at a clock**. Some motors have two shafts. If in doubt, refer to the diagram in the `Motor` class documentation.

4.4 Icon

class Icon

Icons to display on a light matrix.

Each of the following attributes are matrices. This means you can scale icons to adjust the brightness or add icons to make composites.

See the *Making your own images* section for examples.

UP: *Matrix* = Ellipsis

DOWN: *Matrix* = Ellipsis

LEFT: *Matrix* = Ellipsis

RIGHT: *Matrix* = Ellipsis

ARROW_RIGHT_UP: *Matrix* = Ellipsis

ARROW_RIGHT_DOWN: *Matrix* = Ellipsis

ARROW_LEFT_UP: *Matrix* = Ellipsis

ARROW_LEFT_DOWN: *Matrix* = Ellipsis

ARROW_UP: *Matrix* = Ellipsis

ARROW_DOWN: *Matrix* = Ellipsis

ARROW_LEFT: *Matrix* = Ellipsis

ARROW_RIGHT: *Matrix* = Ellipsis

HAPPY: *Matrix* = Ellipsis

SAD: *Matrix* = Ellipsis

`EYE_LEFT: Matrix = Ellipsis`

`EYE_RIGHT: Matrix = Ellipsis`

`EYE_LEFT_BLINK: Matrix = Ellipsis`

`EYE_RIGHT_BLINK: Matrix = Ellipsis`

`EYE_RIGHT_BROW: Matrix = Ellipsis`

`EYE_LEFT_BROW: Matrix = Ellipsis`

EYE_LEFT_BROW_UP: *Matrix* = Ellipsis

EYE_RIGHT_BROW_UP: *Matrix* = Ellipsis

HEART: *Matrix* = Ellipsis

PAUSE: *Matrix* = Ellipsis

EMPTY: *Matrix* = Ellipsis

FULL: *Matrix* = Ellipsis

`SQUARE: Matrix = Ellipsis`

`TRIANGLE_RIGHT: Matrix = Ellipsis`

`TRIANGLE_LEFT: Matrix = Ellipsis`

`TRIANGLE_UP: Matrix = Ellipsis`

`TRIANGLE_DOWN: Matrix = Ellipsis`

`CIRCLE: Matrix = Ellipsis`

`CLOCKWISE: Matrix = Ellipsis`

`COUNTERCLOCKWISE: Matrix = Ellipsis`

`TRUE: Matrix = Ellipsis`

`FALSE: Matrix = Ellipsis`

4.5 Port

class Port

Input and output ports:

A

B

C

D

E

F

4.6 Side

class Side

Side of a hub or a sensor. These devices are mostly rectangular boxes with six sides:

TOP

BOTTOM

FRONT

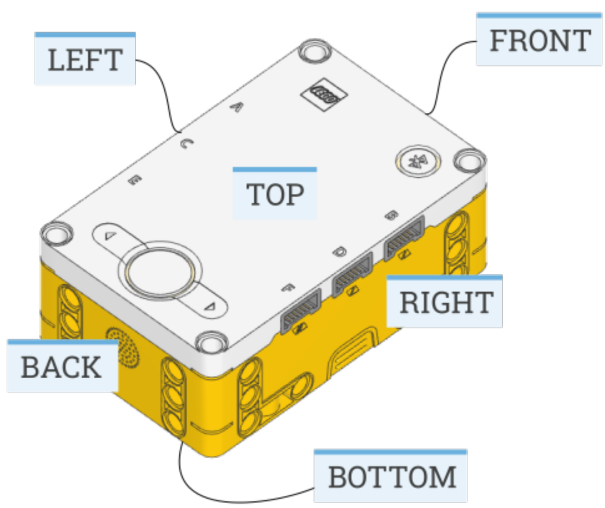
BACK

LEFT

RIGHT

Screens or light matrices have only four sides. For those, `TOP` is treated the same as `FRONT`, and `BOTTOM` is treated the same as `BACK`. The diagrams below define the sides for relevant devices.

Prime Hub



Inventor Hub

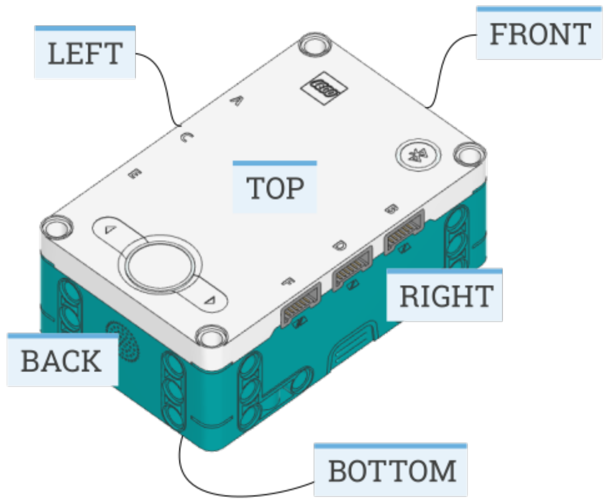
Essential Hub

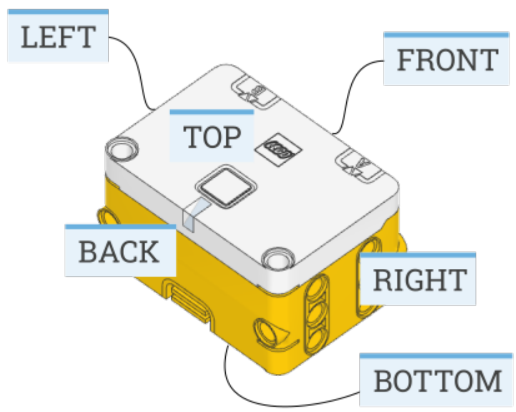
Move Hub

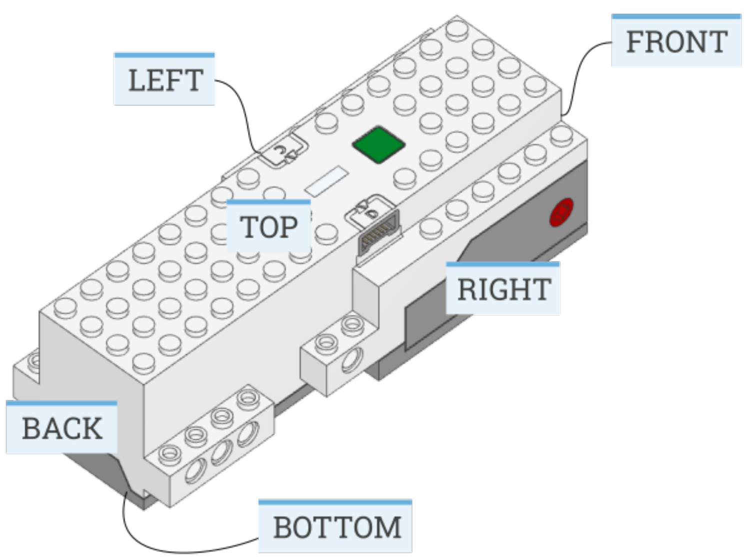
Technic Hub

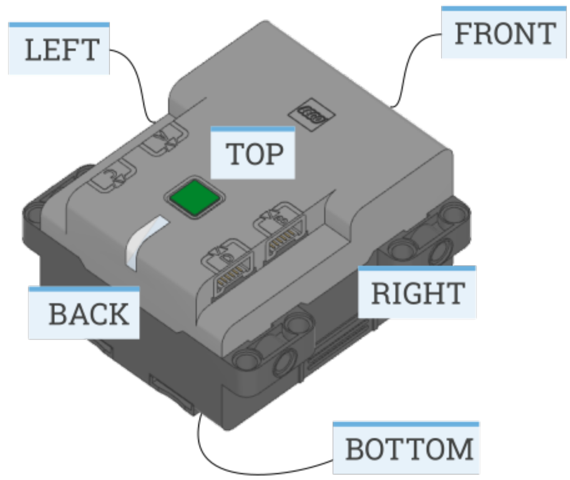
Changed in version 3.2: Changed which side is the front.

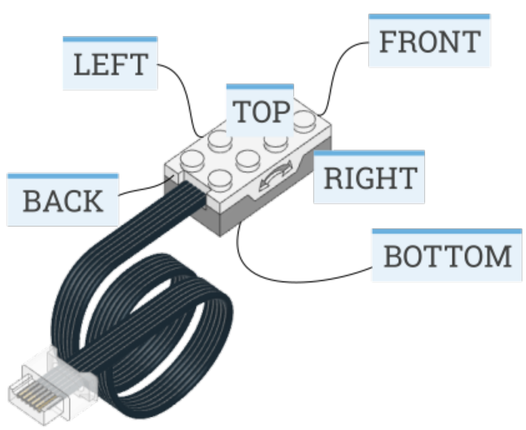
Tilt Sensor


LEFT
FRONT
TOP
RIGHT
BACK
BOTTOM


LEFT
FRONT
TOP
BACK
RIGHT
BOTTOM


FRONT
LEFT
TOP
RIGHT
BACK
BOTTOM


LEFT
FRONT
TOP
BACK
RIGHT
BOTTOM


LEFT
FRONT
TOP
BACK
RIGHT
BOTTOM

4.7 Stop

`class Stop`

Action after the motor stops.

`COAST`

Let the motor move freely.

`COAST_SMART`

Let the motor move freely. For the next relative angle maneuver, take the last target angle (instead of the current angle) as the new starting point. This reduces cumulative errors. This will apply only if the current angle is less than twice the configured position tolerance.

`BRAKE`

Passively resist small external forces.

`HOLD`

Keep controlling the motor to hold it at the commanded angle.

`NONE`

Do not decelerate when approaching the target position. This can be used to concatenate multiple motor or drive base maneuvers without stopping. If no further commands are given, the motor will proceed to run indefinitely at the given speed.

The following table shows how each of the basic stop types add an extra level of resistance to motion. In these examples, `m` is a *Motor* and and `d` is a *DriveBase*. The examples also show how running at zero speed compares to these stop types.

Type	Friction	Back EMF	Speed kept at 0	Angle kept at target	Examples
Coast	•				`m.stop()` `m.run_target(500, 90, Stop.COAST)`
Brake	•	•			`m.brake()` `m.run_target(500, 90, Stop.BRAKE)`
	•	•	•		`m.run(0)` `d.drive(0, 0)`
Hold	•	•	•	•	`m.hold()` `m.run_target(500, 90, Stop.HOLD)` `d.straight(0)` `d.straight(100)`

CHAPTER

FIVE

TOOLS – TIMING TOOLS

Common tools for timing and data logging.

`wait`(*time*)

Pauses the user program for a specified amount of time.

Parameters
`time` (`Number`, *ms*) – How long to wait.

`class StopWatch`

A stopwatch to measure time intervals. Similar to the stopwatch feature on your phone.

`time`() → int: ms

Gets the current time of the stopwatch.

Returns
Elapsed time.

`pause`()

Pauses the stopwatch.

`resume`()

Resumes the stopwatch.

`reset`()

Resets the stopwatch time to 0.

The run state is unaffected:

- If it was paused, it stays paused (but now at 0).
- If it was running, it stays running (but starting again from 0).

CHAPTER SIX

`ROBOTICS` – ROBOTICS AND DRIVE BASES

Robotics module for the Pybricks API.

class DriveBase(*left_motor*, *right_motor*, *wheel_diameter*, *axle_track*)

A robotic vehicle with two powered wheels and an optional support wheel or caster.

By specifying the dimensions of your robot, this class makes it easy to drive a given distance in millimeters or turn by a given number of degrees.

Positive distances, radii, or drive speeds mean driving **forward**. **Negative** means **backward**.

Positive angles and turn rates mean turning **right**. **Negative** means **left**. So when viewed from the top, positive means clockwise and negative means counterclockwise. If desired, you can flip this convention by reversing the `left_motor` and `right_motor` when you initialize this class.

See the *measuring* section for tips to measure and adjust the diameter and axle track values.

Parameters

- **left_motor** (`Motor`) – The motor that drives the left wheel.
- **right_motor** (`Motor`) – The motor that drives the right wheel.
- **wheel_diameter** (`Number`, *mm*) – Diameter of the wheels.
- **axle_track** (`Number`, *mm*) – Distance between the points where both wheels touch the ground.

Driving by a given distance or angle

Use the following commands to drive a given distance, or turn by a given angle.

This is measured using the internal rotation sensors. Because wheels may slip while moving, the traveled distance and angle are only estimates.

straight(*distance*, *then=Stop.HOLD*, *wait=True*)

Drives straight for a given distance and then stops.

Parameters

- **distance** (`Number`, *mm*) – Distance to travel
- **then** (`Stop`) – What to do after coming to a standstill.
- **wait** (`bool`) – Wait for the maneuver to complete before continuing with the rest of the program.

turn(*angle*, *then=Stop.HOLD*, *wait=True*)

Turns in place by a given angle and then stops.

Parameters

- **angle** (Number, *deg*) – Angle of the turn.
- **then** (Stop) – What to do after coming to a standstill.
- **wait** (bool) – Wait for the maneuver to complete before continuing with the rest of the program.

curve(*radius*, *angle*, *then=Stop.HOLD*, *wait=True*)

Drives an arc along a circle of a given radius, by a given angle.

Parameters

- **radius** (Number, *mm*) – Radius of the circle.
- **angle** (Number, *deg*) – Angle along the circle.
- **then** (Stop) – What to do after coming to a standstill.
- **wait** (bool) – Wait for the maneuver to complete before continuing with the rest of the program.

settings(*straight_speed*, *straight_acceleration*, *turn_rate*, *turn_acceleration*)

settings() → Tuple[*int*, *int*, *int*, *int*]

Configures the speed and acceleration used by `straight()`, `turn()`, and `curve()`.

If you give no arguments, this returns the current values as a tuple.

The default values are automatically configured based on your wheel diameter and axle track. They are selected such that your robot drives at about 40% of its maximum speed.

Parameters

- **straight_speed** (Number, *mm/s*) – Straight-line speed of the robot.
- **straight_acceleration** (Number, *mm/s*^{2}) – Straight-line acceleration and deceleration of the robot.
- **turn_rate** (Number, *deg/s*) – Turn rate of the robot.
- **turn_acceleration** (Number, *deg/s*^{2}) – Angular acceleration and deceleration of the robot.

done() → *bool*

Checks if an ongoing command or maneuver is done.

Returns

`True` if the command is done, `False` if not.

Drive forever

Use `drive()` to begin driving at a desired speed and steering.

It keeps going until you use `stop()` or change course by using `drive()` again. For example, you can drive until a sensor is triggered and then stop or turn around.

drive(*speed*, *turn_rate*)

Starts driving at the specified speed and turn rate. Both values are measured at the center point between the wheels of the robot.

Parameters

- **speed** (`Number, mm/s`) – Speed of the robot.
- **turn_rate** (`Number, deg/s`) – Turn rate of the robot.

stop()

Stops the robot by letting the motors spin freely.

Measuring

distance() → int: mm

Gets the estimated driven distance.

Returns
Driven distance since last reset.

angle() → int: deg

Gets the estimated rotation angle of the drive base.

Returns
Accumulated angle since last reset.

state() → Tuple[*int*, *int*, *int*, *int*]

Gets the state of the robot.

Returns
Tuple of distance, drive speed, angle, and turn rate of the robot.

reset()

Resets the estimated driven distance and angle to 0.

stalled() → *bool*

Checks if the drive base is currently stalled.

It is stalled when it cannot reach the target speed or position, even with the maximum actuation signal.

Returns
`True` if the drivebase is stalled, `False` if not.

Measuring and validating the robot dimensions

As a first estimate, you can measure the `wheel_diameter` and the `axle_track` with a ruler. Because it is hard to see where the wheels effectively touch the ground, you can estimate the `axle_track` as the distance between the midpoint of the wheels.

If you don't have a ruler, you can use a LEGO beam to measure. The center-to-center distance of the holes is 8 mm. For some tyres, the diameter is printed on the side. For example, 62.4 x 20 means that the diameter is 62.4mm and that the width is 20 mm.

In practice, most wheels compress slightly under the weight of your robot. To verify, make your robot drive 1000 mm using `my_robot.straight(1000)` and measure how far it really traveled. Compensate as follows:

- If your robot drives **not far enough**, **decrease** the `wheel_diameter` value slightly.
- If your robot drives **too far**, **increase** the `wheel_diameter` value slightly.

Motor shafts and axles bend slightly under the load of the robot, causing the ground contact point of the wheels to be closer to the midpoint of your robot. To verify, make your robot turn 360 degrees using `my_robot.turn(360)` and check that it is back in the same place:

- If your robot turns **not far enough**, **increase** the `axle_track` value slightly.
- If your robot turns **too far**, **decrease** the `axle_track` value slightly.

When making these adjustments, always adjust the `wheel_diameter` first, as done above. Be sure to test both turning and driving straight after you are done.

Using the DriveBase motors individually

After creating a *`DriveBase`* object, you can still use its two motors individually. If you start one motor, the other motor will automatically stop. Likewise, if a motor is already running and you make the drive base move, the original maneuver is cancelled and the drive base will take over.

Advanced settings

The *`settings()`* method is used to adjust commonly used settings like the default speed and acceleration for straight maneuvers and turns. Use the following attributes to adjust more advanced control settings.

You can only change the settings while the robot is stopped. This is either before you begin driving or after you call *`stop()`*.

`distance_control`

The traveled distance and drive speed are controlled by a PID controller. You can use this attribute to change its settings. See the *motor control* attribute for an overview of available methods. The `distance_control` attribute has the same functionality, but the settings apply to every millimeter driven by the drive base, instead of degrees turned by one motor.

`heading_control`

The robot turn angle and turn rate are controlled by a PID controller. You can use this attribute to change its settings. See the *motor control* attribute for an overview of available methods. The `heading_control` attribute has the same functionality, but the settings apply to every degree of rotation of the whole drive base (viewed from the top) instead of degrees turned by one motor.

Changed in version 3.2: The *`done()`* and *`stalled()`* methods have been moved.

6.1 Examples

6.1.1 Driving straight and turning in place

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port, Direction
from pybricks.robotics import DriveBase

# Initialize both motors. In this example, the motor on the
# left must turn counterclockwise to make the robot go forward.
left_motor = Motor(Port.A, Direction.COUNTERCLOCKWISE)
right_motor = Motor(Port.B)

# Initialize the drive base. In this example, the wheel diameter is 56mm.
# The distance between the two wheel-ground contact points is 112mm.
drive_base = DriveBase(left_motor, right_motor, wheel_diameter=56, axle_track=112)

# Drive forward by 500mm (half a meter).
drive_base.straight(500)

# Turn around clockwise (180 degrees)
drive_base.turn(180)

# Drive forward again to drive back.
drive_base.straight(500)

# Turn around counterclockwise.
drive_base.turn(-180)
```

CHAPTER

SEVEN

GEOMETRY – GEOMETRY AND ALGEBRA

class Matrix(*rows*)

Mathematical representation of a matrix. It supports addition (`A + B`), subtraction (`A - B`), and matrix multiplication (`A * B`) for matrices of compatible size.

It also supports scalar multiplication (`c * A` or `A * c`) and scalar division (`A / c`).

A *`Matrix`* object is immutable.

Parameters
rows (`list`) – List of rows. Each row is itself a list of numbers.

T

Returns a new *`Matrix`* that is the transpose of the original.

shape

Returns a tuple (`m`, `n`), where `m` is the number of rows and `n` is the number of columns.

vector(*x*, *y*) → *Matrix*

vector(*x*, *y*, *z*) → *Matrix*

Convenience function to create a *`Matrix`* with the shape (`2`, `1`) or (`3`, `1`).

Parameters

- **x** (`float`) – x-coordinate of the vector.
- **y** (`float`) – y-coordinate of the vector.
- **z** (`float`) – z-coordinate of the vector (optional).

Returns
A matrix with the shape of a column vector.

class Axis

Unit axes of a coordinate system.

X = vector(1, 0, 0)

Y = vector(0, 1, 0)

Z = vector(0, 0, 1)

7.1 Reference frames

The Pybricks module and this documentation use the following conventions:

- X: Positive means forward. Negative means backward.
- Y: Positive means to the left. Negative means to the right.
- Z: Positive means upward. Negative means downward.

To make sure that all hub measurements (such as acceleration) have the correct value and sign, you can specify how the hub is mounted in your creation. This adjust the measurements so that it is easy to see how your *robot* is moving, rather than how the *hub* is moving.

For example, the hub may be mounted upside down in your design. If you configure the settings as shown in Figure 7.1, the hub measurements will be adjusted accordingly. This way, a positive acceleration value in the X direction means that your *robot* accelerates forward, even though the *hub* accelerates backward.

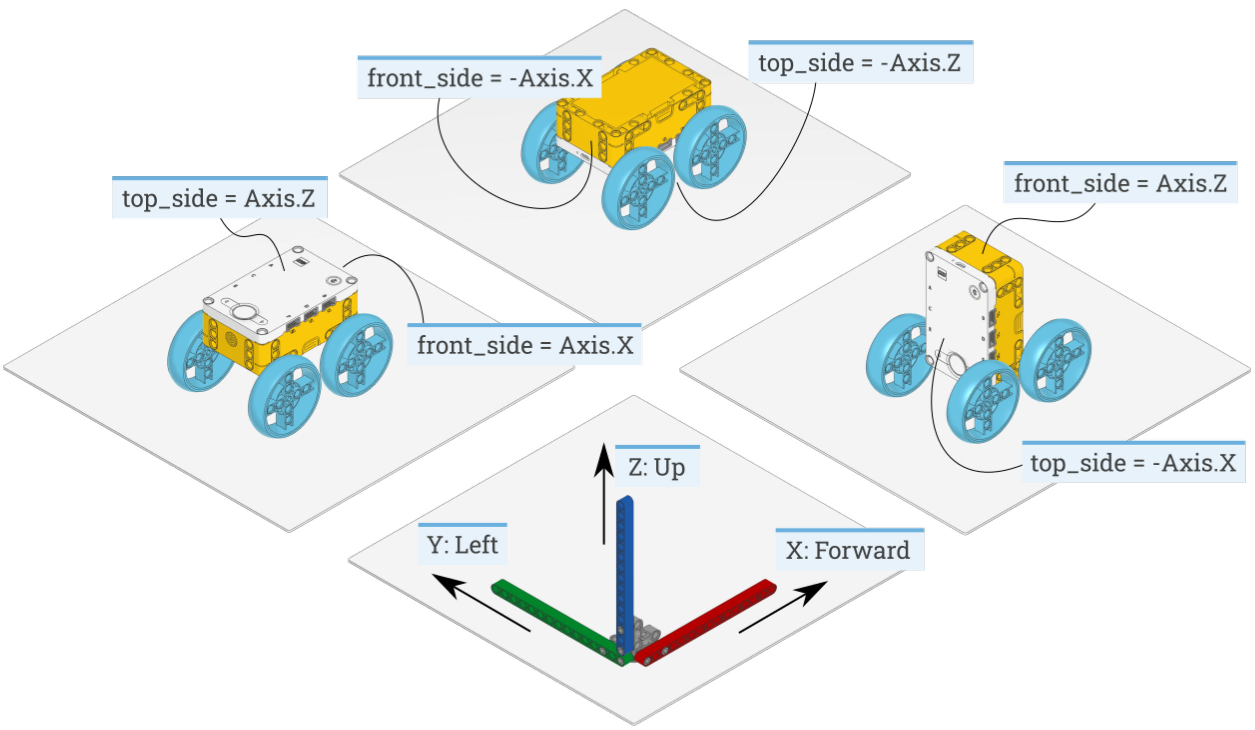


Figure 7.1: How to configure the `top_side` and `front_side` settings for three different robot designs. The same technique can be applied to other hubs and other creations, by noting which way the top and front `Side` of the hub are pointing. The example on the left is the default configuration.

CHAPTER

EIGHT

SIGNALS AND UNITS

Many commands allow you to specify arguments in terms of well-known physical quantities. This page gives an overview of each quantity and its unit.

8.1 Numbers

`Number`

Numbers can be represented as integers or floating point values:

- Integers (*`int`*) are whole numbers like `15` or `-123`.
- Floating point values (*`float`*) are decimal numbers like `3.14` or `-123.45`.

If you see *`Number`* as the argument type, both *`int`* and *`float`* may be used.

For example, *`wait(15)`* and *`wait(15.75)`* are both allowed. In most functions, however, your input value will be truncated to a whole number anyway. In this example, either command makes the program pause for just 15 milliseconds.

Note: The BOOST Move hub doesn't support floating point numbers due to limited system resources. Only integers can be used on that hub.

alias of `Union`[`int`, `float`]

8.2 Time

8.2.1 time: ms

All time and duration values are measured in milliseconds (ms).

For example, the duration of motion with `run_time`, and the duration of *`wait`* are specified in milliseconds.

8.3 Angles and angular motion

8.3.1 angle: deg

All angles are measured in degrees (deg). One full rotation corresponds to 360 degrees.

For example, the angle values of a `Motor` or the `GyroSensor` are expressed in degrees.

8.3.2 rotational speed: deg/s

Rotational speed, or *angular velocity* describes how fast something rotates, expressed as the number of degrees per second (deg/s).

For example, the rotational speed values of a `Motor` or the `GyroSensor` are expressed in degrees per second.

While we recommend working with degrees per second in your programs, you can use the following table to convert between commonly used units.

	deg/s	rpm
1 deg/s =	1	1/6=0.167
1 rpm =	6	1

8.3.3 rotational acceleration: deg/s^2

Rotational acceleration, or *angular acceleration* describes how fast the rotational speed changes. This is expressed as the change of the number of degrees per second, during one second (deg/s^2). This is also commonly written as deg/s^2.

For example, you can adjust the rotational acceleration setting of a `Motor` to change how smoothly or how quickly it reaches the constant speed set point.

8.4 Distance and linear motion

8.4.1 distance: mm

Distances are expressed in millimeters (mm) whenever possible.

For example, the distance value of the `UltrasonicSensor` is measured in millimeters.

While we recommend working with millimeters in your programs, you can use the following table to convert between commonly used units.

	mm	cm	inch
1 mm =	1	0.1	0.0394
1 cm =	10	1	0.394
1 inch =	25.4	2.54	1

8.4.2 dimension: mm

Dimensions are expressed in millimeters (mm), just like distances.

For example, the diameter of a wheel is measured in millimeters.

8.4.3 speed: mm/s

Linear speeds are expressed as millimeters per second (mm/s).

For example, the speed of a robotic vehicle is expressed in mm/s.

8.4.4 linear acceleration: mm/s^2

Linear acceleration describes how fast the speed changes. This is expressed as the change of the millimeters per second, during one second (mm/s^2). This is also commonly written as mm/s^2.

For example, you can adjust the acceleration setting of a *DriveBase* to change how smoothly or how quickly it reaches the constant speed set point.

8.5 Approximate and relative units

8.5.1 percentage: %

Some signals do not have specific units. They range from a minimum (0%) to a maximum (100%). Specifics type of percentages are *relative distances* or *brightness*.

Another example is the sound volume, which ranges from 0% (silent) to 100% (loudest).

8.5.2 relative distance: %

Some distance measurements do not provide an accurate value with a specific unit, but they range from very close (0%) to very far (100%). These are referred to as relative distances.

For example, the distance value of the `InfraredSensor` is a relative distance.

8.5.3 brightness: %

The perceived brightness of a light is expressed as a percentage. It is 0% when the light is off and 100% when the light is fully on. When you choose 50%, this means that the light is perceived as approximately half as bright to the human eye.

8.6 Force and torque

8.6.1 force: N

Force values are expressed in newtons (N).

While we recommend working with newtons in your programs, you can use the following table to convert to and from other units.

	mN	N	lbf
1 mN =	1	0.001	$2.248 \cdot 10^{-4}$
1 N =	1000	1	0.2248
1 lbf =	4448	4.448	1

8.6.2 torque: mNm

Torque values are expressed in millinewtonmeter (mNm) unless stated otherwise.

8.7 Electricity

8.7.1 voltage: mV

Voltages are expressed in millivolt (mV).

For example, you can check the voltage of the battery.

8.7.2 current: mA

Electrical currents are expressed in milliampere (mA).

For example, you can check the current supplied by the battery.

8.7.3 energy: J

Stored energy or energy consumption can be expressed in Joules (J).

8.7.4 power: mW

Power is the rate at which energy is stored or consumed. It is expressed in milliwatt (mW).

8.8 Ambient environment

8.8.1 frequency: Hz

Sound frequencies are expressed in Hertz (Hz).

For example, you can choose the frequency of a beep to change the pitch.

8.8.2 temperature: °C

Temperature is measured in degrees Celcius (°C). To convert to degrees Fahrenheit (°F) or Kelvin (K), you can use the following conversion formulas:

$$^{\circ}F = {^{\circ}C} \cdot \frac{9}{5} + 32.$$

$$K = {^{\circ}C} + 273.15.$$

8.8.3 hue: deg

Hue of a color (0-359 degrees).

CHAPTER

NINE

BUILT-IN CLASSES AND FUNCTIONS

The classes and functions shown on this page can be used without importing anything.

9.1 Input and output

input() → *str*

input(*prompt*) → *str*

Gets input from the user in the terminal window. It waits until the user presses `Enter`.

Parameters

prompt (`str`) – If given, this is printed in the terminal window first. This can be used to ask a question so the user knows what to type.

Returns

Everything the user typed before pressing `Enter`.

print(**objects*, *sep=' '*, *end='\n'*, *file=usys.stdin*)

Prints text or other objects in the terminal window.

Parameters

objects – Zero or more objects to print.

Keyword Arguments

- **sep** (`str`) – This is printed between objects, if there is more than one.
- **end** (`str`) – This is printed after the last object.
- **file** (`FileIO`) – By default, the result is printed in the terminal window. This argument lets you print it to a file instead, if files are supported.

9.2 Basic types

class bool()

class bool(*x*)

Creates a boolean value, which is either `True` or `False`.

The input value is converted using the standard truth testing procedure. If no input is given, it is assumed to be `False`.

Parameters

x – Value to be converted.

Returns
Result of the truth-test.

class complex(*string*)

class complex(*a=0*, *b=0*)

Creates a complex number from a string or from a pair of numbers.

If a string is given, it must be of the form `'1+2j'`. If a pair of numbers is provided, the result is computed as: `a + b * j`.

Parameters

- **string** (`str`) – A string of the form `'1+2j'` .
- **a** (`float or complex`) – A real-valued or complex number.
- **b** (`float or complex`) – A real-valued or complex number.

Returns
The resulting complex number.

class dict(***kwargs*)

class dict(*mapping*, ***kwargs*)

class dict(*iterable*, ***kwargs*)

Creates a dictionary object.

See the standard Python documentation for a comprehensive reference with examples.

class float(*x=0.0*)

Creates a floating point number from a given object.

Parameters
x (`int or float or str`) – Number or string to be converted.

class int(*x=0*)

Creates an integer.

Parameters
x (`int or float or str`) – Object to be converted.

to_bytes(*length*, *byteorder*) → *bytes*

Get a `bytes` representation of the integer.

Parameters

- **length** (`int`) – How many bytes to use.
- **byteorder** (`str`) – Choose `"big"` to put the most significant byte first. Choose `"little"` to put the least significant byte first.

Returns
Byte sequence that represents the integer.

classmethod from_bytes(*bytes*, *byteorder*) → *int*

Convert a byte sequence to the number it represents.

Parameters

- **bytes** (`bytes`) – The bytes to convert.
- **byteorder** (`str`) – Choose `"big"` if the most significant byte is the first element. Choose `"little"` if the least significant byte is the first element.

Returns
The number represented by the bytes.

`class object`

Creates a new, featureless object.

`class type(object)`

Gets the type of an object. This can be used to check if an object is an instance of a particular class.

Parameters
object – Object of which to check the type.

9.3 Sequences

`class bytearray()`

`class bytearray(integer)`

`class bytearray(iterable)`

`class bytearray(string)`

Creates a new `bytearray` object, which is a sequence of integers in the range $0 \leq x \leq 255$. This object is *mutable*, which means that you *can* change its contents after you create it.

If no argument is given, this creates an empty `bytearray` object.

Parameters

- **integer** (`int`) – If the argument is a single integer, this creates a `bytearray` object of zeros. The argument specifies how many.
- **iterable** (`iter`) – If the argument is a `bytearray`, `bytes` object, or some other iterable of integers, this creates a `bytearray` object with the same byte sequence as the argument.
- **string** (`str`) – If the argument is a string, this creates a `bytearray` object containing the encoded string.

`class bytes()`

`class bytes(integer)`

`class bytes(iterable)`

`class bytes(string, encoding)`

Creates a new `bytes` object, which is a sequence of integers in the range $0 \leq x \leq 255$. This object is *immutable*, which means that you *cannot* change its contents after you create it.

If no argument is given, this creates an empty `bytes` object.

Parameters

- **integer** (`int`) – If the argument is a single integer, this creates a `bytes` object of zeros. The argument specifies how many.
- **iterable** (`iter`) – If the argument is a `bytearray`, `bytes` object, or some other iterable of integers, this creates a `bytes` object with the same byte sequence as the argument.
- **string** (`str`) – If the argument is a string, this creates a `bytes` object containing the encoded string.
- **encoding** (`str`) – Specifies which encoding to use for the `string` argument. Only `"utf-8"` is supported.

`len(s)` → *int*

Gets the length (the number of items) of an object.

Parameters
`s` (*Sequence*) – The sequence of which to get the length.

Returns
The length.

`class list()`

`class list(iterable)`

Creates a new list. If no argument is given, this creates an empty `list` object.

A list is *mutable*, which means that you *can* change its contents after you create it.

Parameters
`iterable` (*iter*) – Iterable from which to build the list.

`class slice()`

Creating instances of this class is not supported.

Use indexing syntax instead. For example: `a[start:stop:step]` or `a[start:stop, i]`.

`class str()`

`class str(object)`

`class str(object, encoding)`

Gets the string representation of an object.

If no argument is given, this creates an empty `str` object.

Parameters

- `object` – If only this argument is given, this returns the stirng representation of the object.
- `encoding` (`str`) – If the first argument is a `bytearray` or `bytes` object and the encoding argument is `"utf-8"`, this will decode the byte data to get a string representation.

`class tuple()`

`class tuple(iterable)`

Creates a new tuple. If no argument is given, this creates an empty `tuple` object.

A tuple is *immutable*, which means that you *cannot* change its contents after you create it.

Parameters
`iterable` (*iter*) – Iterable from which to build the tuple.

9.4 Iterators

`all(x)` → *bool*

Checks if all elements of the iterable are true.

Equivalent to:

```
def all(x):
    for element in x:
        if not element:
            return False
    return True
```

Parameters
`x` (*Iterable*) – The iterable to be checked.

Returns
`True` if the iterable `x` is empty or if all elements are true. Otherwise `False`.

any(*x*) → *bool*

Checks if at least one elements of the iterable is true.

Equivalent to:

```
def any(x):
    for element in x:
        if element:
            return True
    return False
```

Parameters
`x` (*Iterable*) – The iterable to be checked.

Returns
`True` if at least one element in `x` is true. Otherwise `False`.

class enumerate(*iterable*, *start=0*)

Enumerates an existing iterator by adding a numeric index.

This function is equivalent to:

```
def enumerate(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
        n += 1
```

iter(*object*) → Iterator

Gets the iterator of the object if available.

Parameters
`object` – Object for which to get the iterator.

Returns
The iterator.

map(*function*, *iterable*) → Iterator

map(*function*, *iterable1*, *iterable2...*) → Iterator

Creates a new iterator that applies the given function to each item in the given iterable and yields the results.

Parameters

- `function` (*callable*) – Function that computes a result for one item in the iterable(s). The number of arguments to this function must match the number of iterables given.
- `iterable` (*iter*) – One or more source interables from which to draw data. With multiple iterables, the iterator stops when the shortest iterable is exhausted.

Returns
The new, mapped iterator.

next(*iterator*) → Any

Retrieves the next item from the iterator by calling its `__next__()` method.

Parameters
`iterator` (*iter*) – Initialized generator object from which to draw the next value.

Returns
The next value from the generator.

class range(*stop*)

class range(*start*, *stop*)

class range(*start*, *stop*, *step*)

Creates a generator that yields values from `start` up to `stop`, with increments of `step`.

Parameters

- **start** (int) – Starting value. Defaults to `0` if only one argument is given.
- **stop** (int) – Endpoint. This value is *not* included.
- **step** (int) – Increment between values. Defaults to `1` if only one or two arguments are given.

reversed(*seq*) → Iterator

Gets an iterator that yields the values from the sequence in the reverse, if supported.

Parameters
seq – Sequence from which to draw samples.

Returns
Iterator that yields values in reverse order, starting with the last value.

sorted(*iterable: Iterable*, *key=None*, *reverse=False*) → List

Sorts objects.

Parameters

- **iterable** (*iter*) – Objects to be sorted. This can also be a generator that yield a finite number of objects.
- **key** (*callable*) – Function `def(item) -> int` that maps an object to a numerical value. This is used to figure out the order of the sorted items.
- **reverse** (bool) – Whether to sort in reverse, putting the highest value first.

Returns
A new list with the sorted items.

zip(*iter_a*, *iter_b*, *...*) → Iterable[Tuple]

Returns an iterator of tuples, where the *i*-th tuple contains the *i*-th element from each of the argument sequences or iterables. The iterator stops when the shortest input iterable is exhausted.

With a single iterable argument, it returns an iterator of 1-tuples. With no arguments, it returns an empty iterator.

This functionality is equivalent to:

```
def zip(*iterables):
    sentinel = object()
    iterators = [iter(it) for it in iterables]
    while iterators:
        result = []
```

(continues on next page)

(continued from previous page)

```
        for it in iterators:
            elem = next(it, sentinel)
            if elem is sentinel:
                return
            result.append(elem)
        yield tuple(result)
```

Parameters

- **iter_a** (*iter*) – The first iterable. This provides the first value for each of the yielded tuples.
- **iter_b** (*iter*) – The second iterable. This provides the second value in each of the yielded tuples. And so on.

Returns

A new iterator that yields tuples containing the values of the individual iterables.

9.5 Conversion functions

bin(*x*) → *str*

Converts an integer to its binary representation. The result is a string prefixed with `0b`. The result is a valid Python expression. For example, `bin(5)` gives `"0b101"`.

Parameters

x (`int`) – Value to be converted.

Returns

A string representing the binary form of the input.

chr(*x*) → *str*

Returns the string representing a character whose Unicode code is the integer `x`. This is the inverse of *ord()*. For example, `chr(97)` gives `"a"`.

Parameters

x (`int`) – Value to be converted (0-255).

Returns

A string with one character, corresponding to the given Unicode value.

hex(*x*) → *str*

Converts an integer to its hexadecimal representation. The result is a lowercase string prefixed with `0x`. The result is a valid Python expression. For example, `hex(25)` gives `"0x19"`.

Parameters

x (`int`) – Value to be converted.

Returns

A string representing the hexadecimal form of the input.

oct(*x*) → *str*

Converts an integer to its octal representation. The result is a string prefixed with `0o`. The result is a valid Python expression. For example, `oct(25)` gives `"0o31"`.

Parameters

x (`int`) – Value to be converted.

Returns
A string representing the octal form of the input.

ord(*c*) → *int*

Converts a string consisting of one Unicode character to the corresponding number. This is the inverse of `chr()`.

Parameters
c (`str`) – Character to be converted.

Returns
Number that represents the character (0–255).

repr(*object*) → *str*

Gets the string that represents an object.

Parameters
x (`object`) – Object to be converted.

Returns
String representation implemented by the object's `__repr__` method.

9.6 Math functions

See also `umath` for floating point math operations.

abs(*x*) → Any

Returns the absolute value of a number.

The argument may be an integer, a floating point number, or any object implementing `__abs__()`. If the argument is a complex number, its magnitude is returned.

Parameters
x (*Any*) – The value.

Returns
Absolute value of `x`.

divmod(*a*, *b*) → Tuple[*int*, *int*]

Gets the quotient and remainder for dividing two integers.

See the standard Python divmod documentation for the expected behavior when `a` or `b` are floating point numbers instead.

Parameters

- **a** (`int`) – Numerator.
- **b** (`int`) – Denominator.

Returns
A tuple with the quotient `a // b` and the remainder `a % b`.

max(*iterable*) → Any

max(*arg1*, *arg2*, ...) → Any

Gets the object with largest value.

The argument may be a single iterable, or any number of objects.

Returns
The object with the largest value.

min(*iterable*) → Any

min(*arg1*, *arg2*, *....*) → Any

Gets the object with smallest value.

The argument may be a single iterable, or any number of objects.

Returns
The object with the smallest value.

pow(*base*, *exp*) → *Number*

Raises the base to the given exponent: base^{exp}.

This is the same as doing `base ** exp`.

Parameters

- **base** (`Number`) – The base.
- **exp** (`Number`) – The exponent.

Returns
The result.

round(*number*) → *int*

round(*number*, *ndigits*) → *float*

Round a number to a given number of digits after the decimal point.

If `ndigits` is omitted or `None`, it returns the nearest integer.

Rounding with one or more digits after the decimal point will not always truncate trailing zeros. To print numbers nicely, format strings instead:

```
# print two decimal places
print('my number: %.2f' % number) print('my number:
{:.2f}'.format(number))
```

Parameters

- **number** (`float`) – The number to be rounded.
- **ndigits** (`int`) – The number of digits remaining after the decimal point.

sum(*iterable*) → *Number*

sum(*iterable*, *start*) → *Number*

Sums the items from the iterable and the start value.

Parameters

- **iterable** (*iter*) – Values to be summed, starting with the first value.
- **start** (`Number`) – Value added to the total.

Returns
The total sum.

9.7 Runtime functions

eval(*expression*) → Any

eval(*expression*, *globals*) → Any

eval(*expression*, *globals*, *locals*) → Any

Evaluates the result of an expression.

Syntax errors are reported as exceptions.

Parameters

- **expression** (`str`) – Expression to evaluate result of.
- **globals** (`dict`) – If given, this controls what functions are available for use in the expression. By default the global scope is accessible.
- **locals** (`dict`) – If given, this controls what functions are available for use in the expression. Defaults to the same as `globals`.

Returns

The value obtained by executing the expression.

exec(*expression*)

exec(*expression*, *globals*) → None

exec(*expression*, *globals*, *locals*) → None

Executes MicroPython code.

Syntax errors are reported as exceptions.

Parameters

- **expression** (`str`) – Code to be executed.
- **globals** (`dict`) – If given, this controls what functions are available for use in the expression. By default the global scope is accessible.
- **locals** (`dict`) – If given, this controls what functions are available for use in the expression. Defaults to the same as `globals`.

globals() → *dict*

Gets a dictionary representing the current global symbol table.

Returns

The dictionary of globals.

hash(*object*) → *int*

Gets the hash value of an object, if the object supports it.

Parameters

object – Object for which to get a hash value.

Returns

The hash value.

help()

help(*object*) → None

Get information about an object.

If no arguments are given, this function prints instructions to operate the REPL. If the argument is `"modules"`, it prints the available modules.

Parameters
`object` – Object for which to print help information.

`id`(*object*) → *int*

Gets the *identity* of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime.

Parameters
`object` – Object of which to get the identifier.

Returns
The identifier.

`locals`() → *dict*

Gets a dictionary representing the current local symbol table.

Returns
The dictionary of locals.

9.8 Class functions

`callable`(*object*) → *bool*

Checks if an object is callable.

Parameters
`object` – Object to check.

Returns
`True` if the object argument appears callable, `False` if not.

`dir`() → List[*str*]

`dir`(*object*) → List[*str*]

Gets a list of attributes of an object.

If no object argument is given, this function gets the list of names in the current local scope.

Parameters
`object` – Object to check for valid attributes.

Returns
List of object attributes or list of names in current local scope.

`getattr`(*object*, *name*) → Any

`getattr`(*object*, *name*, *default*) → Any

Looks up the attribute called `name` in the given `object`.

Parameters

- **`object`** – Object in which to look for the attribute.
- **`name`** (`str`) – Name of the attribute.
- **`default`** – Object to return if the attribute is not found.

Returns
Returns the value of the named attribute.

hasattr(*object*, *name*) → *bool*

Checks if an attribute exists on an object.

Parameters

- **object** – Object in which to look for the attribute.
- **name** (`str`) – Name of the attribute.

Returns

`True` if an attribute by that name exists, `False` if not.

isinstance(*object*, *classinfo*) → *bool*

Checks if an object is an instance of a certain class.

Parameters

- **object** – Object to check the type of.
- **classinfo** (`type` *or* `tuple`) – Class information.

Returns

`True` if the `object` argument is an instance of the `classinfo` argument, or of a subclass thereof.

issubclass(*cls*, *classinfo*) → *bool*

Checks if one class is a subclass of another class.

Parameters

- **cls** – Class type.
- **classinfo** (`type` *or* `tuple`) – Class information.

Returns

`True` if `cls` is a subclass of `classinfo`.

setattr(*object*, *name*, *value*)

Assigns a value to an attribute, provided that the object allows it.

This is the counterpart of `getattr()`.

Parameters

- **object** – Object in which to store the attribute.
- **name** (`str`) – Name of the attribute.
- **value** – Value to store.

super() → *type*

super(*type*) → *type*

super(*type*, *object_or_type*) → *type*

Gets an object that delegates method calls to a parent, or a sibling class of the given type.

Returns

The matching *super()* object.

9.9 Method decorators

`@classmethod`

Transforms a method into a class method.

`@staticmethod`

Transforms a method into a static method.

CHAPTER

TEN

EXCEPTIONS AND ERRORS

This section lists all available exceptions in alphabetical order.

class ArithmeticError
The base class for those built-in exceptions that are raised for various arithmetic errors.

class AssertionError
Raised when an assert statement fails.

class AttributeError
Raised when an attribute reference or assignment fails.

class BaseException
The base class for all built-in exceptions.

It is not meant to be directly inherited by user-defined classes (for that, use *Exception*).

class EOFError
Raised when the *input()* function hits an end-of-file condition (EOF) without reading any data.

class Exception
All built-in exceptions are derived from this class.

All user-defined exceptions should also be derived from this class.

class GeneratorExit
Raised when a generator or coroutine is closed.

class ImportError
Raised when the `import` statement is unable to load a module.

class IndentationError
Base class for syntax errors related to incorrect indentation.

class IndexError
Raised when a sequence subscript is out of range.

class KeyboardInterrupt
Raised when the user hits the interrupt key (normally `Ctrl C`).

class KeyError
Raised when a mapping (dictionary) key is not found in the set of existing keys.

class LookupError
The base class for the exceptions that are raised when a key or index used on a mapping or sequence is invalid.

`class MemoryError`

Raised when an operation runs out of memory.

`class NameError`

Raised when a local or global name is not found.

`class NotImplementedError`

In user defined base classes, abstract methods should raise this exception when they require derived classes to override the method, or while the class is being developed to indicate that the real implementation still needs to be added.

`class OSError`

This exception is raised by the firmware, which is the Operating System that runs on the hub. For *example*, it raises an `OSError` if you call `Motor(Port.A)` when there is no motor on port A.

`errno: int`

Specifies which kind of `OSError` occurred, as listed in the `uerrno` module.

`class OverflowError`

Raised when the result of an arithmetic operation is too large to be represented.

`class RuntimeError`

Raised when an error is detected that doesn't fall in any of the other categories.

The associated value is a string indicating what precisely went wrong.

`class StopIteration`

Raised by built-in function `next()` and an iterator's `__next__()` method to signal that there are no further items produced by the iterator.

Generator functions should return instead of raising this directly.

`class SyntaxError`

Raised when the parser encounters a syntax error.

`class SystemExit`

Raised when you press the stop button on the hub or in the Pybricks Code app.

`class TypeError`

Raised when an operation or function is applied to an object of inappropriate type.

`class ValueError`

Raised when an operation or function receives an argument that has the right type but an inappropriate value. This is used when the situation is not described by a more precise exception such as `IndexError`.

`class ZeroDivisionError`

Raised when the second argument of a division or modulo operation is zero.

10.1 Examples

10.1.1 Debugging in the REPL terminal

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize the motor.
test_motor = Motor(Port.A)

# Start moving at 500 deg/s.
test_motor.run(500)

# If you click on the terminal window and press CTRL+C,
# you can continue debugging in this terminal.
wait(5000)

# You can also do this to exit the script and enter the
# terminal. Variables in the global scope are still available.
raise KeyboardInterrupt

# For example, you can copy the following line to the terminal
# to get the angle, because test_motor is still available.
test_motor.angle()
```

10.1.2 Running code when the stop button is pressed

```
from pybricks.tools import wait

print("Started!")

try:

    # Run your script here as you normally would. In this
    # example we just wait forever and do nothing.
    while True:
        wait(1000)

except SystemExit:
    # This code will run when you press the stop button.
    # This can be useful to "clean up", such as to move
    # the motors back to their starting positions.
    print("You pressed the stop button!")
```

10.1.3 Detecting devices using OSError

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port

from uerrno import ENODEV

try:
    # Try to initialize a motor.
    my_motor = Motor(Port.A)

    # If all goes well, you'll see this message.
    print("Detected a motor.")
except OSError as ex:
    # If an OSError was raised, we can check what
    # kind of error this was, like ENODEV.
    if ex.errno == ENODEV:
        # ENODEV is short for "Error, no device."
        print("There is no motor this port.")
    else:
        print("Another error occurred.")
```

CHAPTER ELEVEN

MICROPYTHON – MICROPYTHON INTERNALS

Access and control MicroPython internals.

const(*value*) → Any

Declares the value as a constant, which makes your code more efficient.

To reduce memory usage further, prefix its name with an underscore (`_ORANGES`). This constant can only be used within the same file.

If you want to import the value from another module, use a name without an underscore (`APPLES`). This uses a bit more memory.

Parameters

value (`int` *or* `float` *or* `str` *or* `tuple`) – The literal to be made constant.

Returns

The constant value.

heap_lock()

Locks the heap. When locked, no memory allocation can occur. A `MemoryError` will be raised if any heap allocation is attempted.

heap_unlock() → *int*

Unlocks the heap. Memory allocation is now allowed again.

If `heap_lock()` was called multiple times, `heap_unlock()` must be called the same number of times to make the heap available again.

Returns

The lock depth after unlocking. It is `0` once it is unlocked.

kbd_intr(*chr*)

Sets the character that triggers a `KeyboardInterrupt` exception when you type it in the input window. By default it is set to `3`, which corresponds to pressing `Ctrl` `C`.

Parameters

chr (`int`) – Character that should raise the `KeyboardInterrupt`. Choose `-1` to disable this feature.

mem_info()

mem_info(*verbose*) → None

Prints information about stack and heap memory usage.

Parameters

verbose – If any value is given, it also prints out the entire heap. This indicates which blocks are used and which are free.

opt_level() → *int*

opt_level(*level: int*) → None

Sets the optimization level for code compiled on the hub:

0. Assertion statements are enabled. The built-in `__debug__` variable is `True`. Script line numbers are saved, so they can be reported when an Exception occurs.
1. Assertions are ignored and `__debug__` is `False`. Script line numbers are saved.
2. Assertions are ignored and `__debug__` is `False`. Script line numbers are saved.
3. Assertions are ignored and `__debug__` is `False`. Script line numbers are *not* saved.

This applies only to code that you run in the REPL, because regular scripts are already compiled before they are sent to the hub.

Parameters
level (int) – The level to be set.

Returns
If no argument is given, this returns the current optimization level.

qstr_info()

qstr_info(*verbose*) → None

Prints how many strings are interned and how much RAM they use.

MicroPython uses string interning to save both RAM and ROM. This avoids having to store duplicate copies of the same string.

Parameters
verbose – If any value is given, it also prints out the names of all RAM-interned strings.

stack_use() → *int*

Checks the amount of stack that is being used. This can be used to compute differences in stack usage at different points in a script.

Returns
The amount of stack in use.

11.1 Examples

11.1.1 Using constants for efficiency

```
from micropython import const

# This value can be used here. Other files can import it too.
APPLES = const(123)

# These values can only be used within this file.
_ORANGES = const(1 << 8)
_BANANAS = const(789 + _ORANGES)

# You can read the constants as normal values. The compiler
# will just insert the numeric values for you.
fruit = APPLES + _ORANGES + _BANANAS
print(fruit)
```

11.1.2 Checking free RAM

```
from micropython import mem_info

# Print memory usage.
mem_info()
```

This prints information in the format shown below. In this example for the SPIKE Prime Hub, there are 257696 bytes (251 KB) worth of memory remaining for the variables in your code.

```
stack: 372 out of 40184
GC: total: 258048, used: 352, free: 257696
No. of 1-blocks: 4, 2-blocks: 2, max blk sz: 8, max free sz: 16103
```

11.1.3 Getting more memory statistics

```
from micropython import const, opt_level, mem_info, qstr_info, stack_use

# Get stack at start.
stack_start = stack_use()

# Print REPL compiler optimization level.
print("level", opt_level())

# Print memory usage.
mem_info()

# Print memory usage and a memory map.
mem_info(True)

# Print interned string information.
qstr_info()

# Print interned string information and their names.
APPLES = const(123)
_ORANGES = const(456)
qstr_info(True)


def test_stack():
    return stack_use()


# Check the stack.
print("Stack diff: ", test_stack() - stack_start)
```

CHAPTER

TWELVE

UERRNO – ERROR CODES

The errno attribute of an *OSError* indicates why this exception was raised. This attribute has one of the following values. See also *this example*.

EAGAIN: *int*

The operation is not complete and should be tried again soon.

EBUSY: *int*

The device or resource is busy and cannot be used right now.

ECANCELED: *int*

The operation was canceled.

EINVAL: *int*

An invalid argument was given. Usually ValueError is used instead.

EIO: *int*

An unspecified error occurred.

ENODEV: *int*

Device was not found. For example, a sensor or motor is not plugged in the correct port.

EOPNOTSUPP: *int*

The operation is not supported on this hub or on the connected device.

EPERM: *int*

The operation cannot be performed in the current state.

ETIMEDOUT: *int*

The operation timed out.

errorcode: Dict[*int*, *str*]

Dictionary that maps numeric error codes to strings with symbolic error code.

CHAPTER

THIRTEEN

UIO – INPUT/OUTPUT STREAMS

This module contains `stream` objects that behave like files.

class BytesIO()

class BytesIO(*data*)

class BytesIO(*alloc_size*)

A binary stream using an in-memory bytes buffer.

Parameters

- **data** (`bytes` *or* `bytearray`) – Optional bytes-like object that contains initial data.
- **alloc_size** (`int`) – Optional number of preallocated bytes. This parameter is unique to MicroPython. It is not recommended to use it in end-user code.

getvalue() → *bytes*

Gets the contents of the underlying buffer.

class StringIO()

class StringIO(*string*)

class StringIO(*alloc_size*)

A stream using an in-memory string buffer.

Parameters

- **string** (`str`) – Optional string with initial data.
- **alloc_size** (`int`) – Optional number of preallocated bytes. This parameter is unique to MicroPython. It is not recommended to use it in end-user code.

getvalue() → *str*

Gets the contents of the underlying buffer.

class FileIO

This type represents a file opened in binary mode with `open(name, 'rb')`. You should not instantiate this class directly.

CHAPTER FOURTEEN

UJSON – JSON ENCODING AND DECODING

Convert between Python objects and the JSON data format.

dump(*object*, *stream*, *separators=(', ', ': ')*)

Serializes an object to a JSON string and write it to a stream.

Parameters

- **obj** – Object to serialize.
- **stream** – Stream to write the output to.
- **separators** (tuple) – An `(item_separator, key_separator)` tuple to specify how elements should be separated.

dumps(*object*, *separators=(', ', ': ')*)

Serializes an object to JSON and return it as a string

Parameters

- **obj** – Object to serialize.
- **separators** (tuple) – An `(item_separator, key_separator)` tuple to specify how elements should be separated.

Returns
The JSON string.

load(*stream*)

Parses the stream to interpret and deserialize the JSON data to a MicroPython object.

Parsing continues until end-of-file is encountered. A `ValueError` is raised if the data in stream is not correctly formed.

Parameters
stream – Stream from which to read the JSON string.

Returns
The deserialized MicroPython object.

loads(*string*)

Parses the string to interpret and deserialize the JSON data to a MicroPython object.

A `ValueError` is raised if the string is not correctly formed.

Parameters
string (str) – JSON string to decode.

Returns
The deserialized MicroPython object.

CHAPTER

FIFTEEN

UMATH – MATH FUNCTIONS

This MicroPython module is similar to the math module in Python.

See also the *built-in math functions* that can be used without importing anything.

15.1 Rounding and sign

ceil(*x*) → *int*

Rounds up.

Parameters
`x` (`float`) – The value to be rounded.

Returns
Value rounded towards positive infinity.

floor(*x*) → *int*

Rounds down.

Parameters
`x` (`float`) – The value to be rounded.

Returns
Value rounded towards negative infinity.

trunc(*x*) → *int*

Truncates decimals to get the integer part of a value.

This is the same as rounding towards `0`.

Parameters
`x` (`float`) – The value to be truncated.

Returns
Integer part of the value.

fmod(*x*, *y*) → *float*

Gets the remainder of `x / y`.

Not to be confused with `modf()`.

Parameters

- `x` (`float`) – The numerator.
- `y` (`float`) – The denominator.

Returns
Remainder after division

`fabs`(*x*) → *float*

Gets the absolute value.

Parameters
x (`float`) – The value.

Returns
Absolute value of `x`.

`copysign`(*x*, *y*) → *float*

Gets `x` with the sign of `y`.

Parameters

- **x** (`float`) – Determines the magnitude of the return value.
- **y** (`float`) – Determines the sign of the return value.

Returns
`x` with the sign of `y`.

15.2 Powers and logarithms

`e = 2.718282`

The mathematical constant e.

`exp`(*x*) → *float*

Gets `e` raised to the power of `x`.

Parameters
x (`float`) – The exponent.

Returns
`e` raised to the power of `x`.

`pow`(*x*, *y*) → *float*

Gets `x` raised to the power of `y`.

Parameters

- **x** (`float`) – The base number.
- **y** (`float`) – The exponent.

Returns
`x` raised to the power of `y`.

`log`(*x*) → *float*

Gets the natural logarithm.

Parameters
x (`float`) – The value.

Returns
The natural logarithm of `x`.

sqrt(*x*) → *float*

Gets the square root.

Parameters
x (float) – The value x.

Returns
The square root of x.

15.3 Trigonomety

pi = 3.141593

The mathematical constant .

degrees(*x*) → *float*

Converts an angle from radians to degrees.

Parameters
x (float) – Angle in radians.

Returns
Angle in degrees.

radians(*x*) → *float*

Converts an angle from degrees to radians.

Parameters
x (float) – Angle in degrees.

Returns
Angle in radians.

sin(*x*) → *float*

Gets the sine of an angle.

Parameters
x (float) – Angle in radians.

Returns
Sine of x.

asin(*x*) → *float*

Applies the inverse sine operation.

Parameters
x (float) – Opposite / hypotenuse.

Returns
Arcsine of x, in radians.

cos(*x*) → *float*

Gets the cosine of an angle.

Parameters
x (float) – Angle in radians.

Returns
Cosine of x.

`acos`(x) → *float*

Applies the inverse cosine operation.

Parameters
x (`float`) – Adjacent / hypotenuse.

Returns
Arccosine of `x`, in radians.

`tan`(x) → *float*

Gets the tangent of an angle.

Parameters
x (`float`) – Angle in radians.

Returns
Tangent of `x`.

`atan`(x) → *float*

Applies the inverse tangent operation.

Parameters
x (`float`) – Opposite / adjacent.

Returns
Arctangent of `x`, in radians.

`atan2`(b, a) → *float*

Applies the inverse tangent operation on `b / a`, and accounts for the signs of `b` and `a` to produce the expected angle.

Parameters

- **b** (`float`) – Opposite side of the triangle.
- **a** (`float`) – Adjacent side of the triangle.

Returns
Arctangent of `b / a`, in radians.

15.4 Other math functions

`isfinite`(x) → *bool*

Checks if a value is finite.

Parameters
x (`float`) – The value to be checked.

Returns
`True` if `x` is finite, else `False`.

`isinfinite`(x) → *bool*

Checks if a value is infinite.

Parameters
x (`float`) – The value to be checked.

Returns
`True` if `x` is infinite, else `False`.

isnan(x) → *bool*

Checks if a value is not-a-number.

Parameters
x (`float`) – The value to be checked.

Returns
`True` if `x` is not-a-number, else `False`.

modf(x) → Tuple[*float*, *float*]

Gets the fractional and integral parts of `x`, both with the same sign as `x`.

Not to be confused with *fmod()*.

Parameters
x (`float`) – The value to be decomposed.

Returns
Tuple of fractional and integral parts.

frexp(x) → Tuple[*float*, *float*]

Decomposes a value `x` into a tuple `(m, p)`, such that `x == m * (2 ** p)`.

Parameters
x (`float`) – The value to be decomposed.

Returns
Tuple of `m` and `p`.

ldexp(m, p) → *float*

Computes `m * (2 ** p)`.

Parameters

- **m** (`float`) – The value.
- **p** (`float`) – The exponent.

Returns
Result of `m * (2 ** p)`.

CHAPTER SIXTEEN

URANDOM – PSEUDO-RANDOM NUMBERS

This module implements pseudo-random number generators.

All functions in this module should be used with positional arguments. Keyword arguments are not supported.

Basic random numbers

randint(*a*, *b*) → *int*

Gets a random integer N satisfying $a \leq N \leq b$.

Parameters

- **a** (int) – Lowest value. This value *is* included in the range.
- **b** (int) – Highest value. This value *is* included in the range.

Returns
The random integer.

random() → *float*

Gets a random value x satisfying $0 \leq x < 1$.

Returns
The random value.

Random numbers from a range

getrandbits(*k*) → *int*

Gets a random integer N satisfying $0 \leq N < 2^{\text{bits}}$.

Parameters
k (int) – How many bits to use for the result.

randrange(*stop*) → *int*

randrange(*start*, *stop*) → *int*

randrange(*start*, *stop*, *step*) → *int*

Returns a randomly selected element from `range(start, stop, step)`.

For example, `randrange(1, 7, 2)` returns random numbers from `1` up to (but excluding) `7`, in increments of `2`. In other words, it returns `1`, `3`, or `5`.

Parameters

- **start** (int) – Lowest value. Defaults to `0` if only one argument is given.
- **stop** (int) – Highest value. This value is *not* included in the range.

- **step** (`int`) – Increment between values. Defaults to 1 if only one or two arguments are given.

Returns
The random number.

uniform(*a*, *b*) → *float*

Gets a random floating point value x satisfying $a \leq x \leq b$.

Parameters

- **a** (`float`) – Lowest value.
- **b** (`float`) – Highest value.

Returns
The random value.

Random elements from a sequence

choice(*sequence*) → Any

Gets a random element from a sequence such as a tuple or list.

Parameters
sequence – Sequence from which to select a random element.

Returns
The randomly selected element.

Raises
IndexError – If the sequence is empty.

Updating the random seed

seed(*value=None*)

Initializes the random number generator.

This gets called when the module is imported, so normally you do not need to call this.

Parameters
value – Seed value. When using `None`, the system timer will be used.

CHAPTER

SEVENTEEN

USELECT – WAIT FOR EVENTS

This module provides functions to efficiently wait for events on multiple streams.

Poll instance and class

poll() → *Poll*

Creates an instance of the `Poll` class.

Returns
The `Poll` instance.

class Poll

register(*object*, *eventmask=POLLOUT | POLLOUT*)

Register a stream object for polling. The stream object will now be monitored for events. If an event happens, it becomes part of the return value of `poll()`.

If this method is called again for the same stream object, the object will not be registered again, but the `eventmask` flags will be updated, as if calling `modify()`.

Parameters

- **object** (`FileIO`) – Stream to be registered for polling.
- **eventmask** (`int`) – Which events to use. Should be `POLLIN`, `POLLOUT`, or their logical disjunction: `POLLIN | POLLOUT`.

unregister(*poll*)

Unregister an object from polling.

Parameters
object (`FileIO`) – Stream to be unregistered from polling.

modify(*object*, *eventmask*)

Modifies the event mask for the stream object.

Parameters

- **object** (`FileIO`) – Stream to be registered for polling.
- **eventmask** (`int`) – Which events to use.

Raises
OSError – If the object is not registered. The error is `ENOENT`.

poll(*timeout=- 1*) → List[Tuple[*FileIO*, *int*]]

Wait until at least one of the registered objects has a new event or exceptional condition ready to be handled.

Parameters

timeout (int) – Timeout in milliseconds. Choose `0` to return immediately or choose `-1` to wait indefinitely.

Returns

A list of tuples. There is one (`object`, `eventmask`, ...) tuple for each object with an event, or no tuples if there are no events to be handled. The `eventmask` value is a combination of poll flags to indicate what happened. This may include `POLLERR` and `POLLHUP` even if they were not registered.

ipoll(*timeout=- 1*, *flags=1*) → Iterator[Tuple[*FileIO*, *int*]]

First, just like *poll()*, wait until at least one of the registered objects has a new event or exceptional condition ready to be handled.

But instead of a list, this method returns an iterator for improved efficiency. The iterator yields one (`object`, `eventmask`, ...) tuple at a time, and overwrites it when yielding the next value. If you need the values later, make sure to copy them explicitly.

Parameters

- **timeout** (int) – Timeout in milliseconds. Choose `0` to return immediately or choose `-1` to wait indefinitely.
- **flags** (int) – If set to `1`, one-shot behavior for events is employed. This means that streams for which events happened will have their event masks automatically reset using `poll.modify(obj, 0)`. This way, new events for such a stream won't be processed until a new mask is set with *modify()*, which is useful for asynchronous I/O schedulers.

Event mask flags

POLLIN: *int*

Data is available for reading.

POLLOUT: *int*

More data can be written.

POLLERR: *int*

Error condition happened on the associated stream. Should be handled explicitly or else further invocations of *poll()* may return right away.

POLLHUP: *int*

Hang up happened on the associated stream. Should be handled explicitly or else further invocations of *poll()* may return right away.

17.1 Examples

See the projects website for a demo that uses this module.

CHAPTER

EIGHTEEN

USTRUCT – PACK AND UNPACK BINARY DATA

This module provides functions to convert between Python values and C-like data structs.

calcsize(*format: str*) → *int*

Gets the data size corresponding to a format string

Parameters
`format` (`str`) – Data format string.

Returns
The number of bytes needed to represent this format.

pack(*format*, *value1*, *value2*, *...*)

Packs the values using the given format.

Parameters
`format` (`str`) – Data format string.

Returns
The data encoded as bytes.

pack_into(*format*, *buffer*, *offset*, *value1*, *value2*, *...*)

Encode the values using the given format and write them to a given buffer.

Parameters

- `format` (`str`) – Data format string.
- `buffer` (`bytearray`) – Buffer to store the encoded data.
- `offset` (`int`) – Offset from the start of the buffer. Use a negative value to count from the end of the buffer.

unpack(*format*, *data*) → Tuple

Decodes the binary data using the given format.

Parameters

- `format` (`str`) – Data format string.
- `data` (`bytes` *or* `bytearray`) – Data to unpack.

Returns
The decoded data as a tuple of values.

unpack_from(*format*, *data*, *offset*) → Tuple

Decodes binary data from a buffer using the given format.

Parameters

- **format** (str) – Data format string.
- **data** (bytes *or* bytearray) – Data buffer to unpack.
- **offset** (int) – Offset from the start of the data. Use a negative value to count from the end of the data.

Returns
The decoded data as a tuple of values.

The following byte orders are supported:

Character	Byte order	Size	Alignment
@	native	native	native
<	little-endian	standard	none
>	big-endian	standard	none
!	network (= big-endian)	standard	none

The following data types are supported:

Format	C Type	Python type	Standard size
b	signed char	integer	1
B	unsigned char	integer	1
h	short	integer	2
H	unsigned short	integer	2
i	int	integer	4
I	unsigned int	integer	4
l	long	integer (1)	4
L	unsigned long	integer (1)	4
q	long long	integer (1)	8
Q	unsigned long long	integer (1)	8
f	float	float	4
d	double	float	8
s	char[]	bytes	
P	void *	integer	

- (1) Supports values up to +/-1073741823

CHAPTER

NINETEEN

USYS – SYSTEM SPECIFIC FUNCTIONS

This MicroPython module is a subset of the sys module in Python.

Input and output streams

stdin

This is a stream object (*uio.FileIO*) that receives input from a connected terminal, if any.

Also see *kbd_intr* to disable KeyboardInterrupt when passing binary data via stdin.

stdout

This is a stream object (*uio.FileIO*) that sends output to a connected terminal, if any.

stderr

Alias for *stdout*.

Version info

implementation

MicroPython version tuple. See format and example below.

version

Python compatibility version, Pybricks version, and build date. See format and example below.

version_info

Python compatibility version. See format and example below.

19.1 Examples

19.1.1 Version information

```
from pybricks import version

# ('essentialhub', '3.2.0b5', 'v3.2.0b5 on 2022-11-11')
print(version)
```

```
import usys

# ('micropython', (1, 19, 1), 'SPIKE Essential Hub with STM32F413RG', 6)
print(usys.implementation)

# '3.4.0; Pybricks MicroPython v3.2.0b5 on 2022-11-11'
print(usys.version)

# (3, 4, 0)
print(usys.version_info)
```

19.1.2 Standard input and output

The stdin stream can be used to capture input via the Pybricks Code input/output window. See the keyboard input project to learn how this works. This approach can be extended to exchange data with any other device as well.

PYTHON MODULE INDEX

INDEX

D

E

F

G

H

P

Q

R

S

T

U

V

W

Y

Z