Pybricks Modules and Examples

Version 2.0.0.post1

Jul 28, 2021

Table of Contents

This documentation has everything you need to install Pybricks and run your first scripts.

Step 1: Install Pybricks

To get started, go to the *EV3 Quick Start*.

Step 2: Start coding!

After you've followed the installation steps for your hub, check out the Pybricks modules in the left hand menu to see what you can do.

Step 3: Share what you made (or ask for help!)

Got questions or issues? Please share your findings on our support page so we can make Pybricks even better. *Thank you!*

CHAPTER 1

EV3 Quick Start

1.1 Installation

This page guides you through the steps to collect and install everything you need to start programming.

1.1.1 What do you need?

To get started, you'll need:

- A Windows 10 or Mac OS computer
- Internet access and administrator access

  This is required during the installation only. You will not need special access to write and run programs later on.

- A microSD card

  You'll need a card with a minimum capacity of 4GB and a maximum capacity of 32GB. This type of microSD card is also known as microSDHC. We recommend cards with Application Performance Class A1.

- A microSD card slot or card reader in your computer

  If your computer does not have a (micro)SD card slot, you can use an external USB (micro)SD card reader.

- A mini-USB cable, like the one included with your EV3 set

The typical configuration of this equipment is summarized in Figure 1.1.

1.1.2 Preparing your computer

You'll write your MicroPython programs using Visual Studio Code. Follow the steps below to download, install, and configure this application:

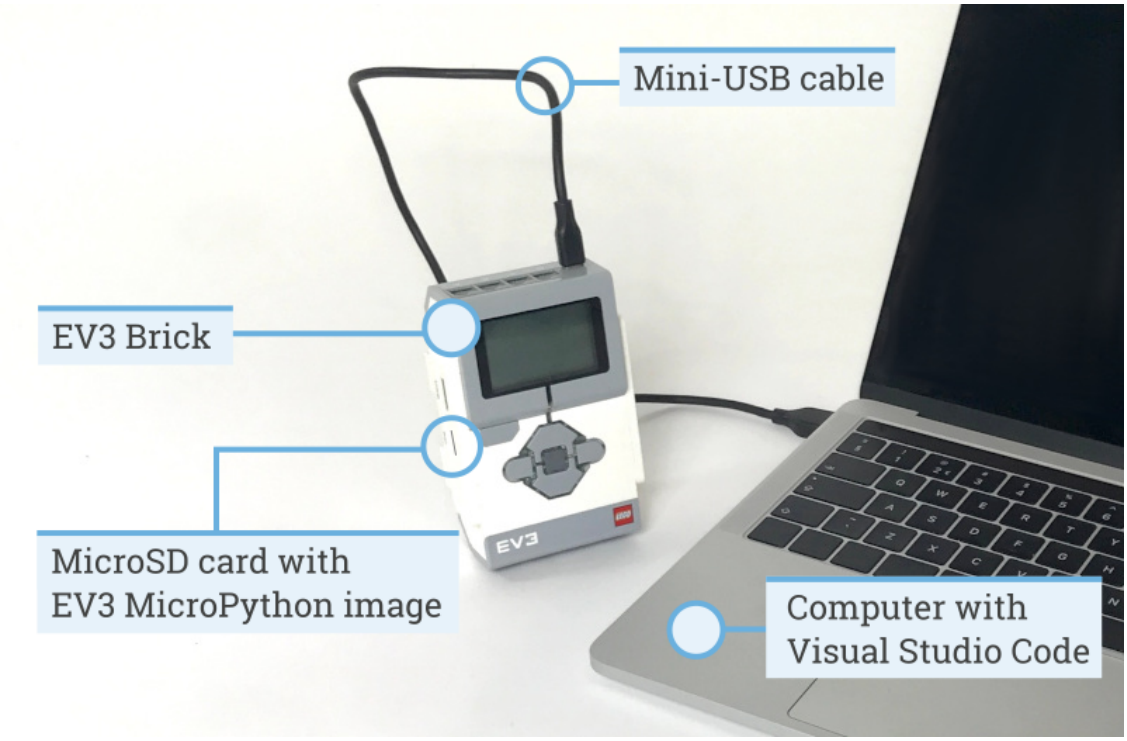


Figure 1.1: Setup overview

1. Download Visual Studio Code.
2. Follow the on-screen instructions to install the application.
3. Launch Visual Studio Code.
4. Open the extensions tab.
5. Install the EV3 MicroPython extension as shown in Figure 1.2.

1.1.3 Preparing the microSD card

To make it possible to run MicroPython programs on your EV3 Brick, you'll now learn how to install the required tools on your microSD card.

If the microSD card contains files you want to keep, make sure to create a backup of its contents first. See *managing files on the EV3* to learn how to backup your previous MicroPython programs if necessary.

This process erases everything on your microSD card, including any previous MicroPython programs on it.

To install the MicroPython tools on your microSD card:

1. Download the EV3 MicroPython microSD card image and save it in a convenient location. This file is approximately 360 MB. You do **not** need to unzip the file.

2. Download and install a microSD card flashing tool such as Etcher.
3. Insert the microSD card into your computer or card reader.
4. Launch the flashing tool and follow the steps on your screen to install the file you have just downloaded. If you use Etcher, you can follow the instructions below, as shown in Figure 1.3.

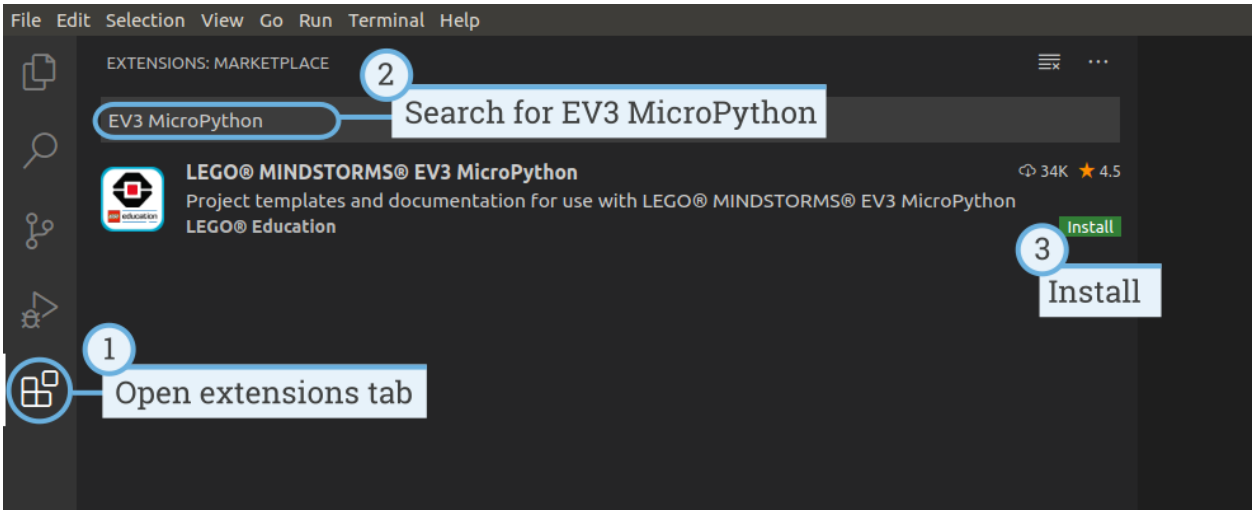


Figure 1.2: Installing the extension from the Visual Studio Code marketplace

(a) Select the EV3 MicroPython microSD card image file you have just downloaded.

(b) Select your microSD card. Make sure that the device and size correspond to your microSD card.

(c) Start the flashing process. This may take several minutes. Do not remove the card until the flashing process is complete.

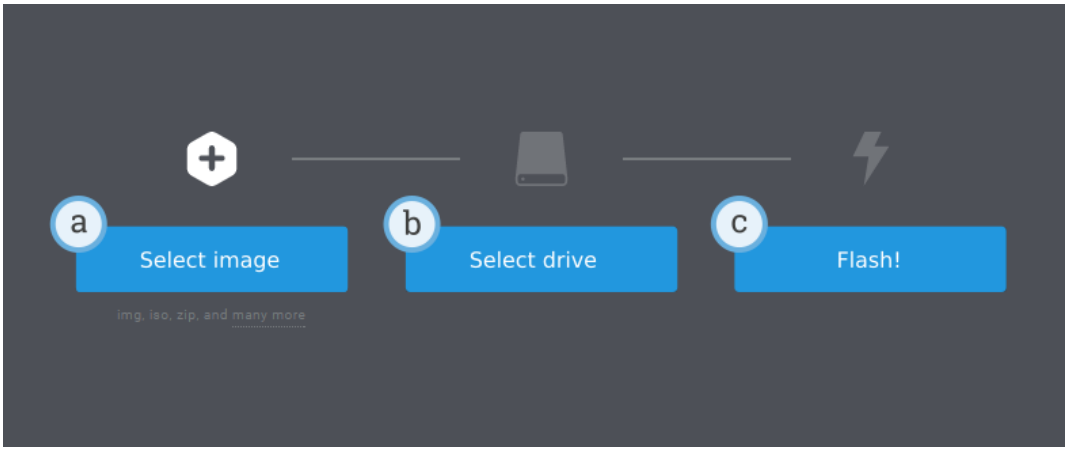


Figure 1.3: Using Etcher to flash the EV3 MicroPython microSD card image

1.1.4 Updating the microSD card

To update the microSD card, download a new image file using the link above and flash it to the microSD card as described above. Be sure to *back up any MicroPython programs you want to save*.

You do not need to erase the contents of the microSD card first. This is done automatically when you flash the new image file.

1.2 Using the EV3 Brick

Make sure the EV3 Brick is turned off. Insert the microSD card you prepared into the microSD card slot on the EV3 Brick, as shown in Figure 1.4.

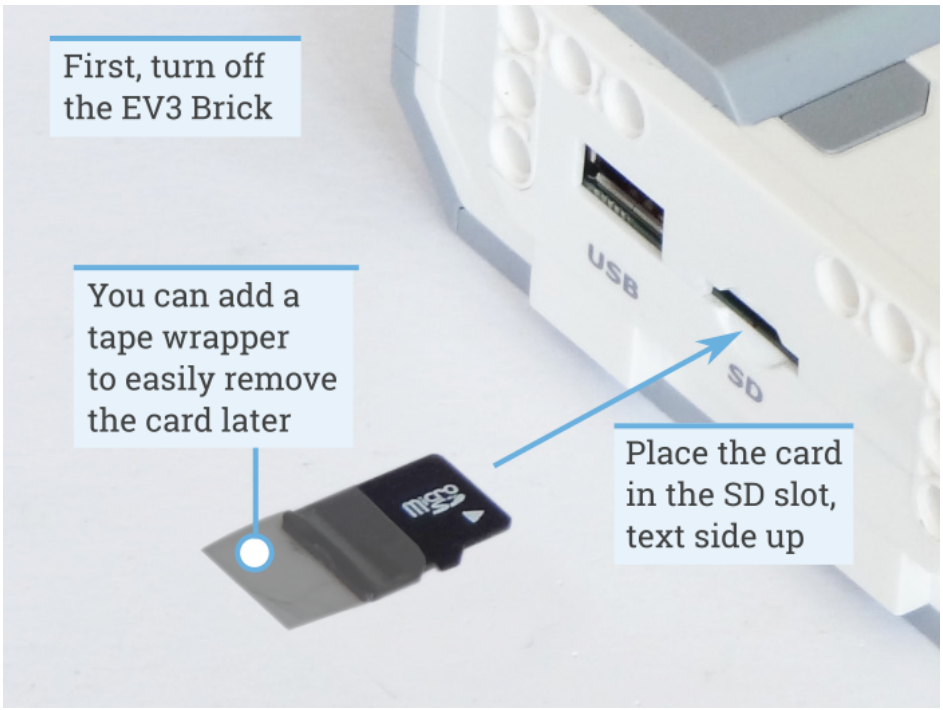


Figure 1.4: Inserting the flashed microSD card into the EV3 Brick

1.2.1 Turning the EV3 Brick on and off

Turn on the EV3 Brick by pressing the dark gray center button.

The boot process may take several minutes. While booting, the EV3 Brick status light turns orange and blinks intermittently, and you'll see a lot of text on the EV3 screen. The EV3 Brick is ready for use when the status light turns green.

To turn the EV3 Brick off, open the shutdown menu with the back button, and then select *Power Off* using the center button, as shown in Figure 1.5.

1.2.2 Viewing motor and sensor values

When you're not running a program, you can view motor and sensor values using the device browser, as shown in Figure 1.6.

1.2.3 Running a program without a computer

You can run previously downloaded programs directly from the EV3 Brick.

To do so, find the program using the *file browser* on the EV3 screen and press the center button key to start the program as shown in Figure 1.7.

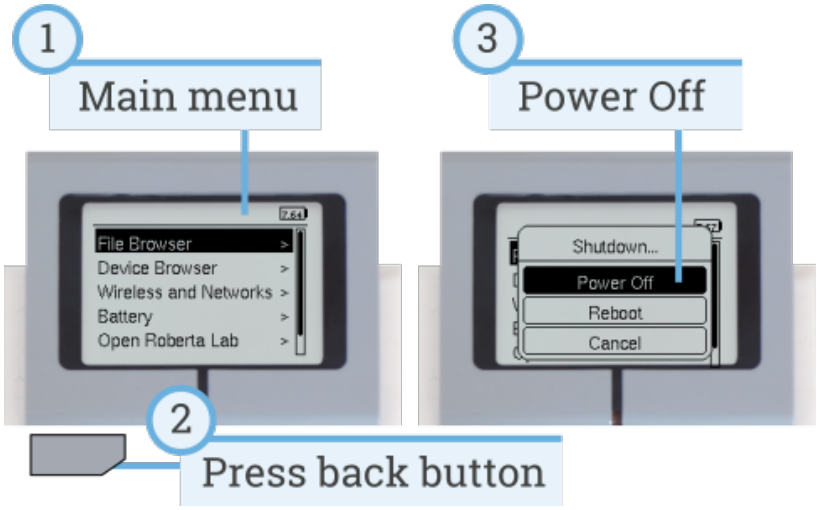


Figure 1.5: Turning the EV3 Brick off

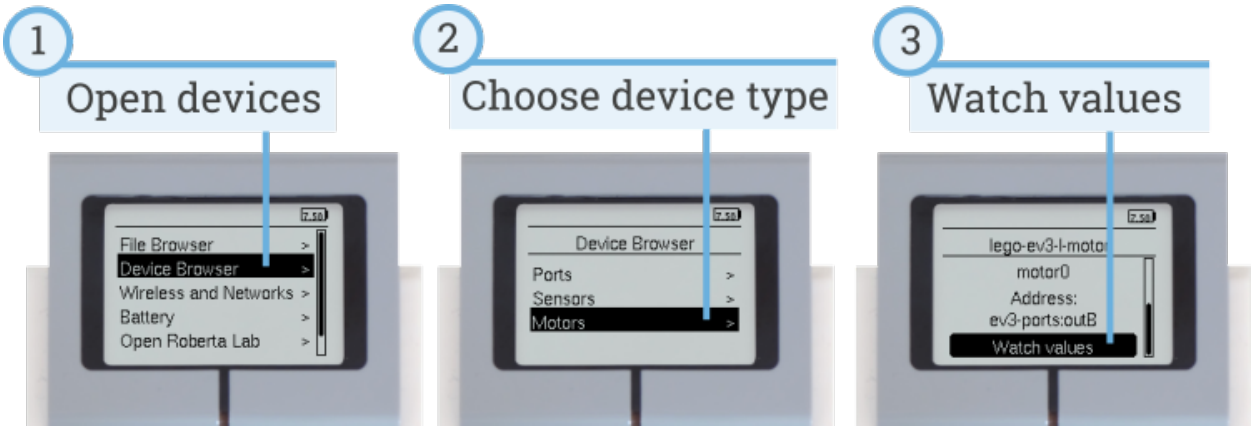


Figure 1.6: Viewing motor and sensor values

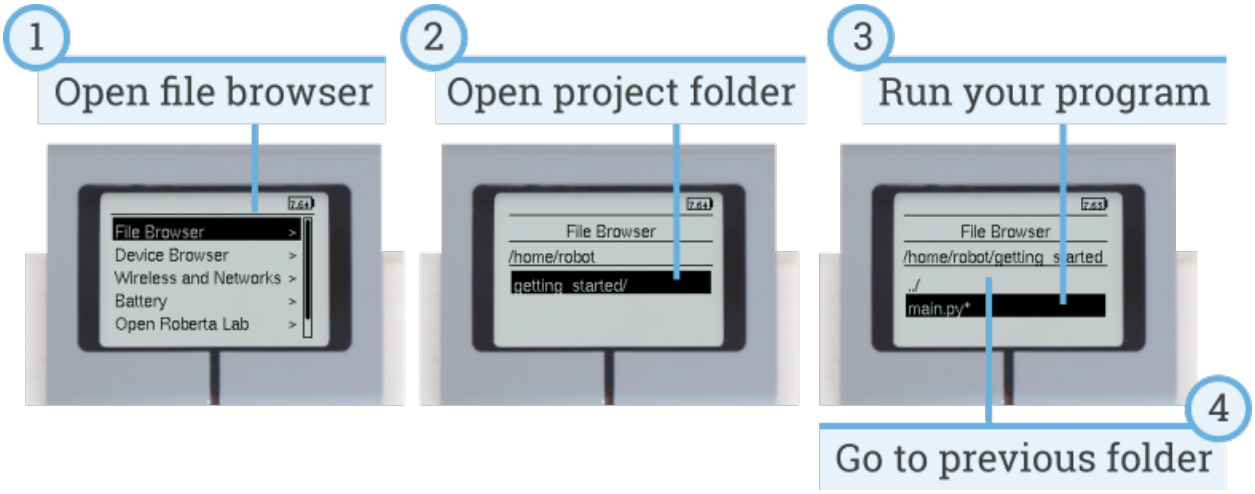


Figure 1.7: Starting a program using the buttons on the EV3 Brick

1.2.4 Going back to the original firmware

You can go back to the LEGO® firmware and your LEGO programs at any time. To do so:

1. Turn the EV3 Brick off as shown above.
2. Wait for the screen and brick status light to turn off.
3. Remove the microSD card.
4. Turn the EV3 on.

1.3 Creating and running programs

Now that you've set up your computer and EV3 Brick, you're ready to start writing programs.

To make it easier to create and manage your programs, let's first have a quick look at how MicroPython projects and programs for your EV3 robots are organized.

Programs are organized into *project folders*, as shown in Figure 1.8. A project folder is a directory on your computer that contains the main program (**main.py**) and other optional scripts or files. This project folder and all of its contents will be copied to the EV3 Brick, where the main program will be run.

This page shows you how to create such a project and how to transfer it to the EV3 Brick.

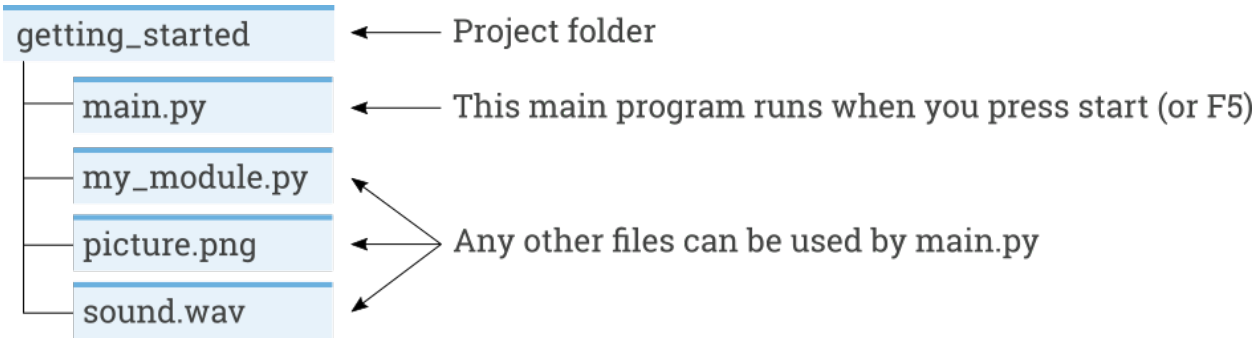


Figure 1.8: A project contains a program called **main.py** and optional resources like sounds or MicroPython modules.

1.3.1 Creating a new project

To create a new project, open the EV3 MicroPython tab and click *create a new project*, as shown in Figure 1.9. Enter a project name in the text field that appears and press *Enter*. When prompted, choose a location for this program and confirm by clicking *choose folder*.

When you create a new project, it already includes a file called *main.py*. To see its contents and to modify it, open it from the file browser as shown in Figure 1.10. This is where you'll write your programs.

If you are new to MicroPython programming, we recommend that you keep the existing code in place and add your code to it.

1.3.2 Opening an existing project

To open a project you created previously, click *File* and click *Open Folder*, as shown in Figure 1.11. Next, navigate to your previously created project folder and click *OK*. You can also open your recently used projects using the *Open Recent* menu option.

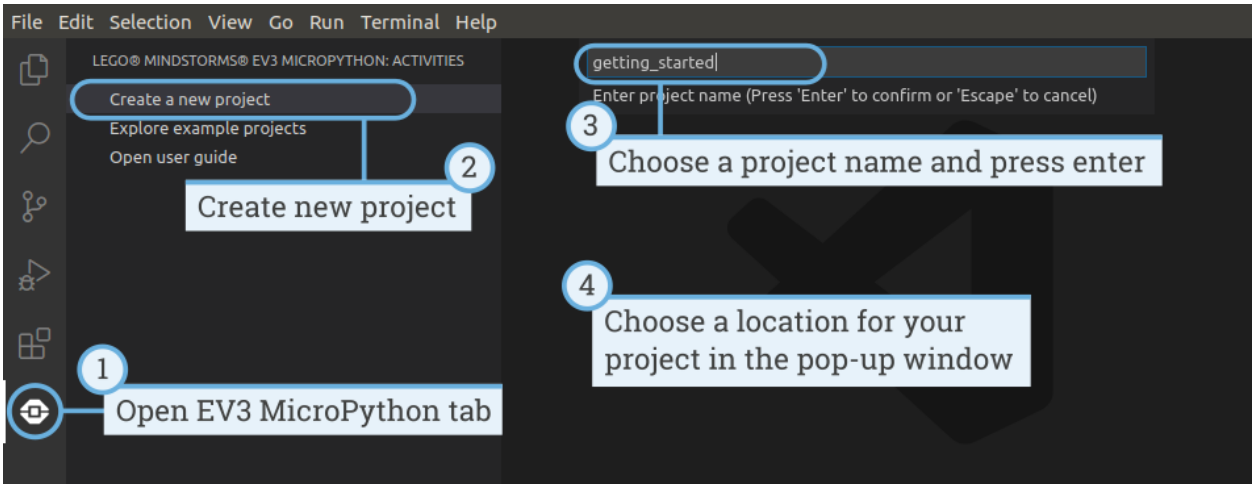


Figure 1.9: Creating a new project. This example is called *getting_started*, but you can choose any name.

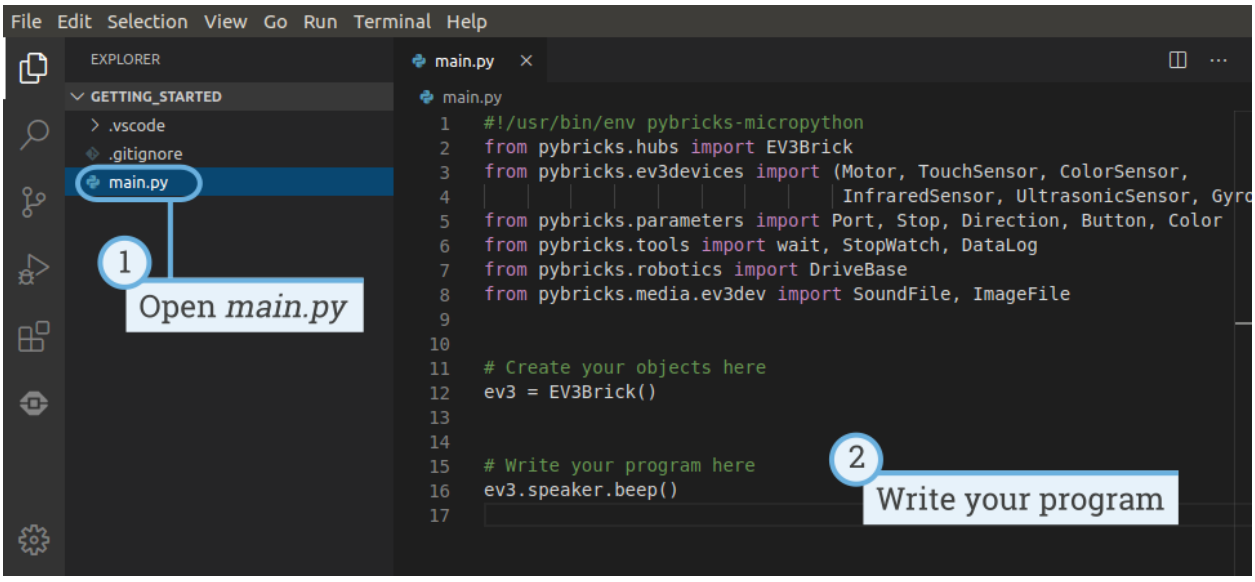


Figure 1.10: Opening the default *main.py* program.

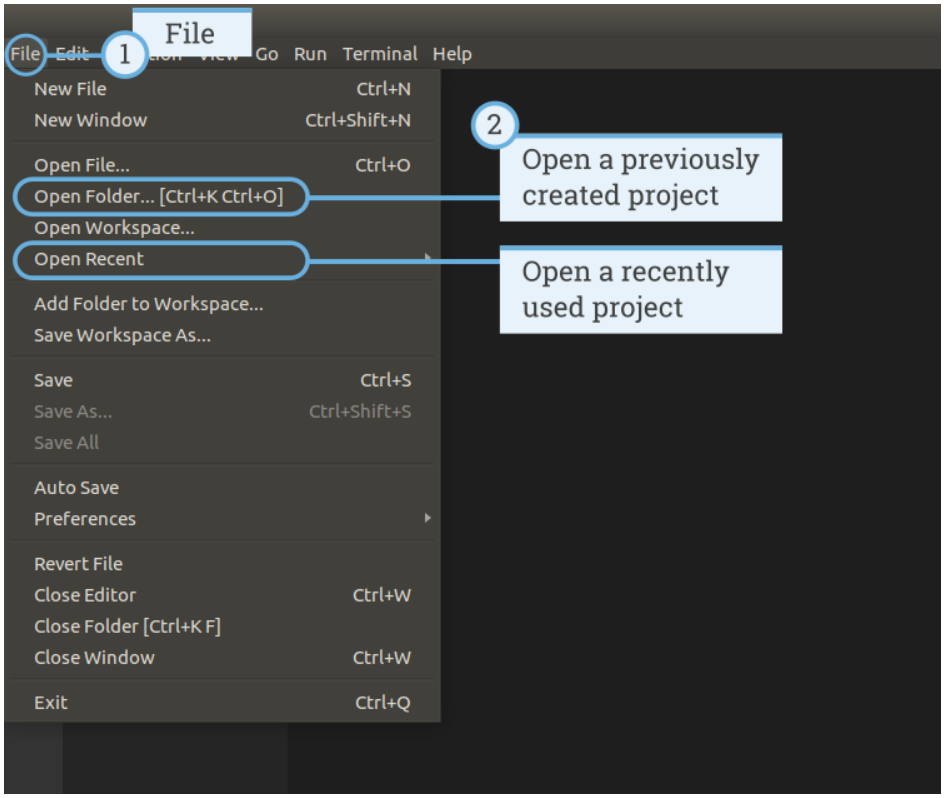


Figure 1.11: Opening a previously created project.

1.3.3 Connecting to the EV3 Brick with Visual Studio Code

To be able to transfer your code to the EV3 Brick, you'll first need to connect the EV3 Brick to your computer with the mini-USB cable and configure the connection with Visual Studio Code. To do so:

- Turn the EV3 Brick on
- Connect the EV3 Brick to your computer with the mini-USB cable
- Configure the USB connection as shown in Figure 1.12.

1.3.4 Downloading and running a program

You can press the F5 key to run the program. Alternatively, you can start it manually by going to the *debug* tab and clicking the green start arrow, as shown in Figure 1.13.

When the program starts, a pop-up toolbar allows you to stop the program if necessary. You can also stop the program at any time using the back button on the EV3 Brick.

If your program produces any output with the `print` command, this is shown in the output window.

1.3.5 Expanding the example program

Now that you've run the basic code template, you can expand the program to make a motor move. First, attach a Large Motor to Port B on the EV3 Brick, as shown in Figure 1.14.

Next, edit *main.py* to make it look like this:

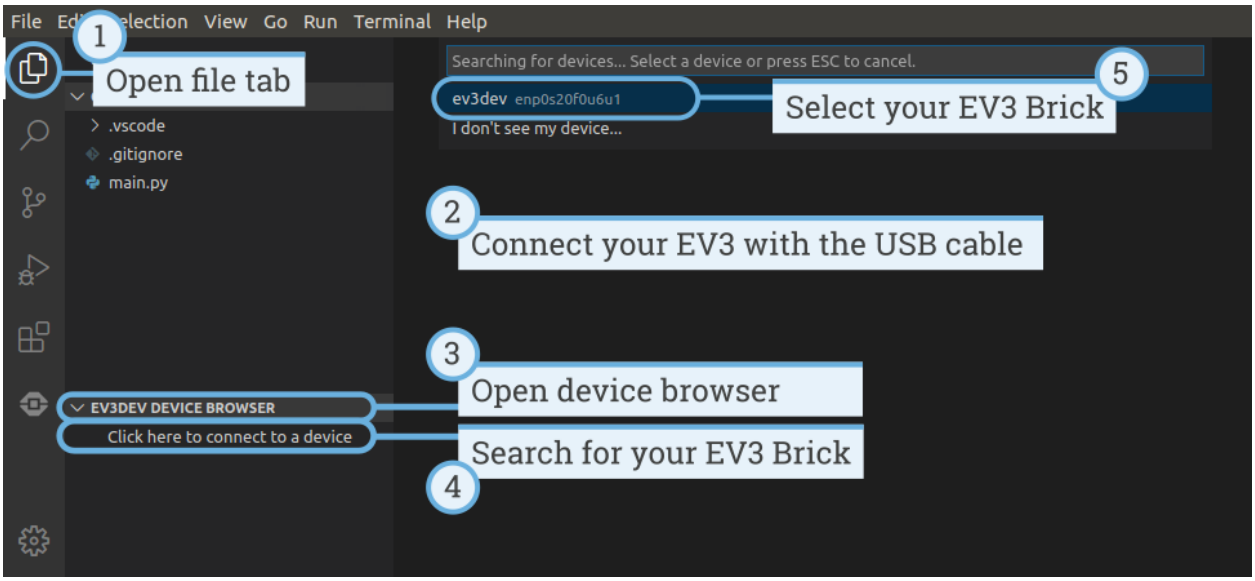


Figure 1.12: Configuring the USB connection between the computer and the EV3 Brick

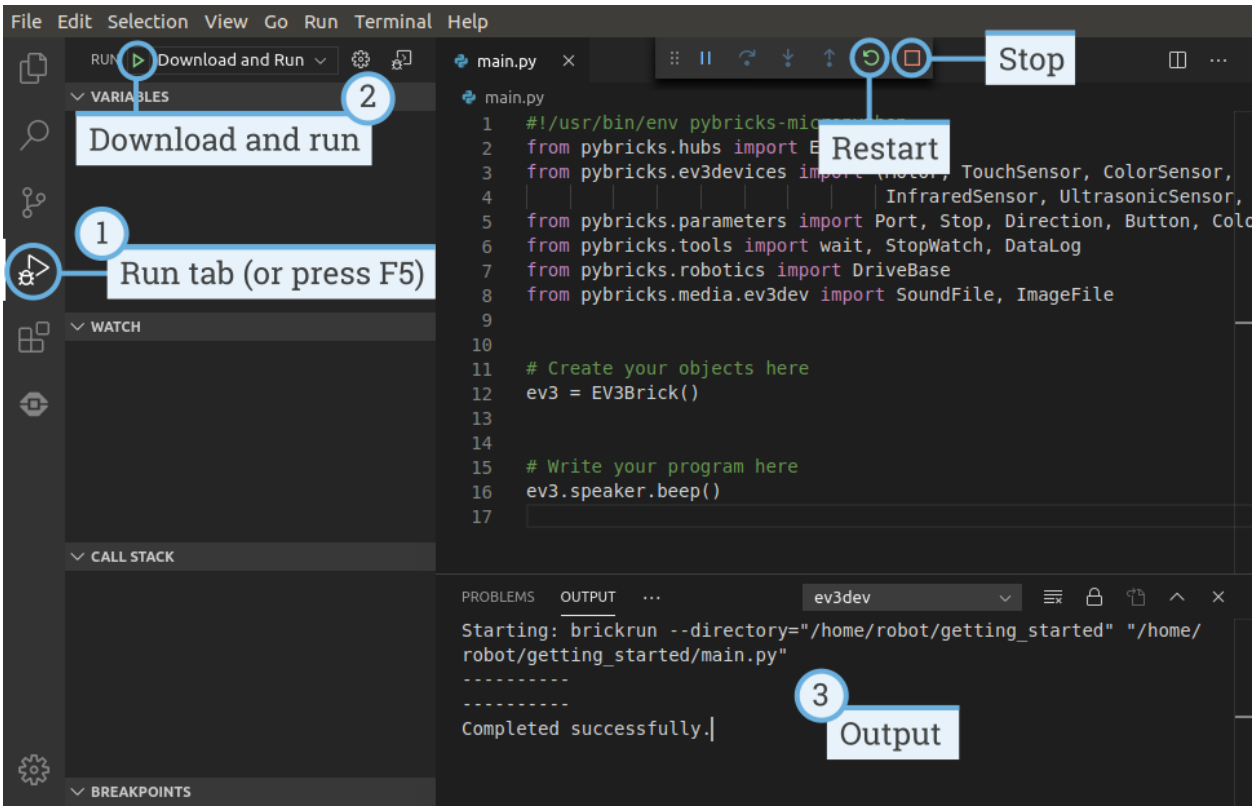


Figure 1.13: Running a program

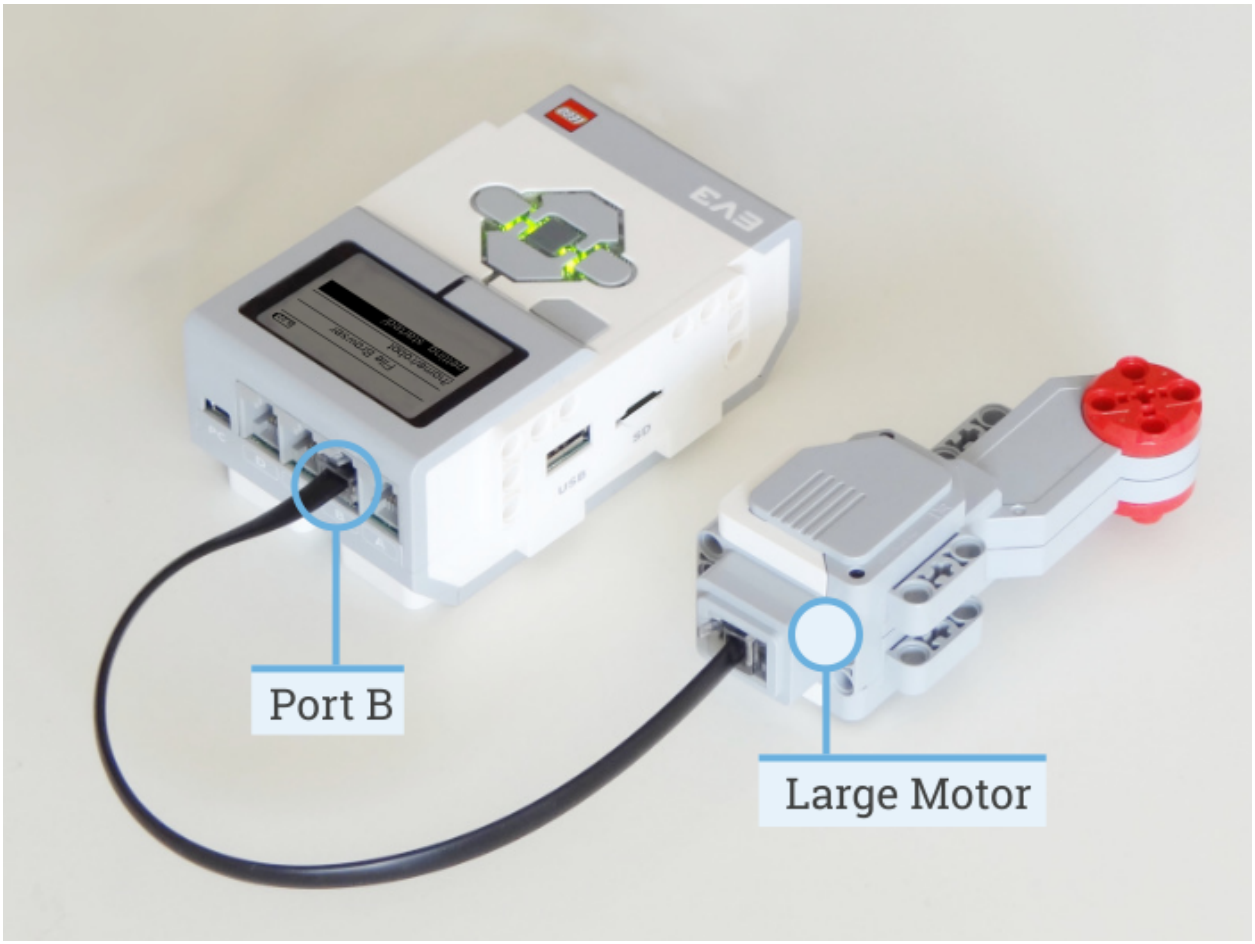


Figure 1.14: The EV3 Brick with a Large Motor attached to port B.

```
#!/usr/bin/env pybricks-micropython
from pybricks.hubs import EV3Brick
from pybricks.ev3devices import Motor
from pybricks.parameters import Port

# Create your objects here

# Initialize the EV3 Brick.
ev3 = EV3Brick()

# Initialize a motor at port B.
test_motor = Motor(Port.B)

# Write your program here

# Play a sound.
ev3.speaker.beep()

# Run the motor up to 500 degrees per second. To a target angle of 90 degrees.
test_motor.run_target(500, 90)

# Play another beep sound.
ev3.speaker.beep(frequency=1000, duration=500)
```

This program makes your robot beep, rotate the motor, and beep again with a higher pitched tone. Run the program to make sure that it works as expected.

1.3.6 Managing files on the EV3 Brick

After you've downloaded a project to the EV3 Brick, you can run, delete, or back up programs stored on it using the device browser as shown in Figure 1.15.

1.4 Accessing advanced EV3 features

MicroPython runs on top of ev3dev, which is a specific version of Linux. Linux is an *operating system*. (Other popular operating systems are Microsoft Windows and Apple macOS.) This means that your EV3 is almost like a real computer, just much smaller.

Note: *If you just want to write MicroPython programs, you can skip the remaining sections.*

The remaining sections are aimed at curious users who want go beyond MicroPython and access some of the other built-in features of Linux and ev3dev.

1.4.1 The Linux command line

Although your EV3 Brick is quite like a real computer, you do not interact with it using a big screen and a mouse. Instead, you can access files and programs on it using the *command line*. It is also called the *terminal*.

Follow the steps in Figure 1.16 to access the command line. Now you can enter commands by typing them in and pressing enter.

Running basic commands

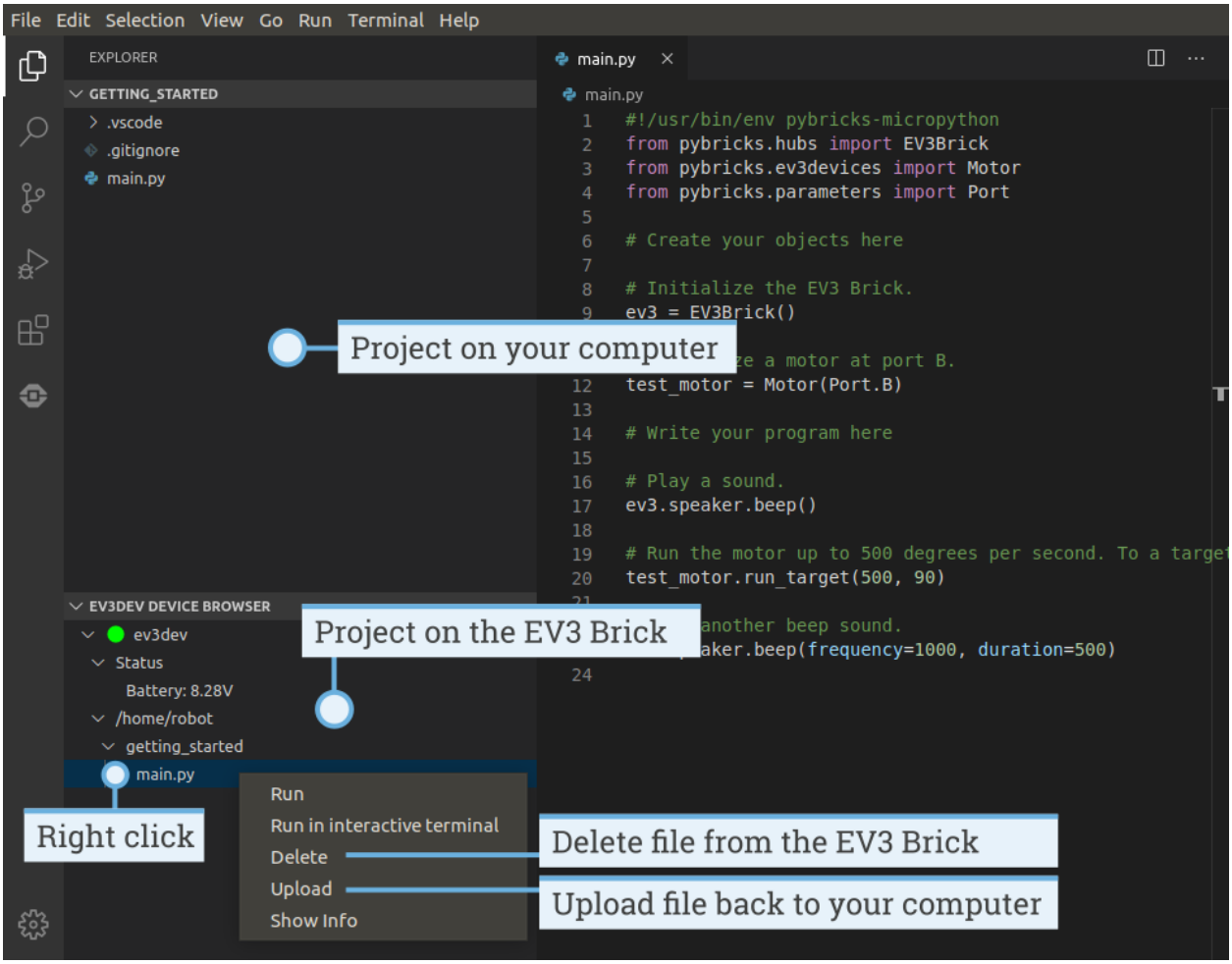


Figure 1.15: Using the EV3 device browser to manage files on your EV3 Brick

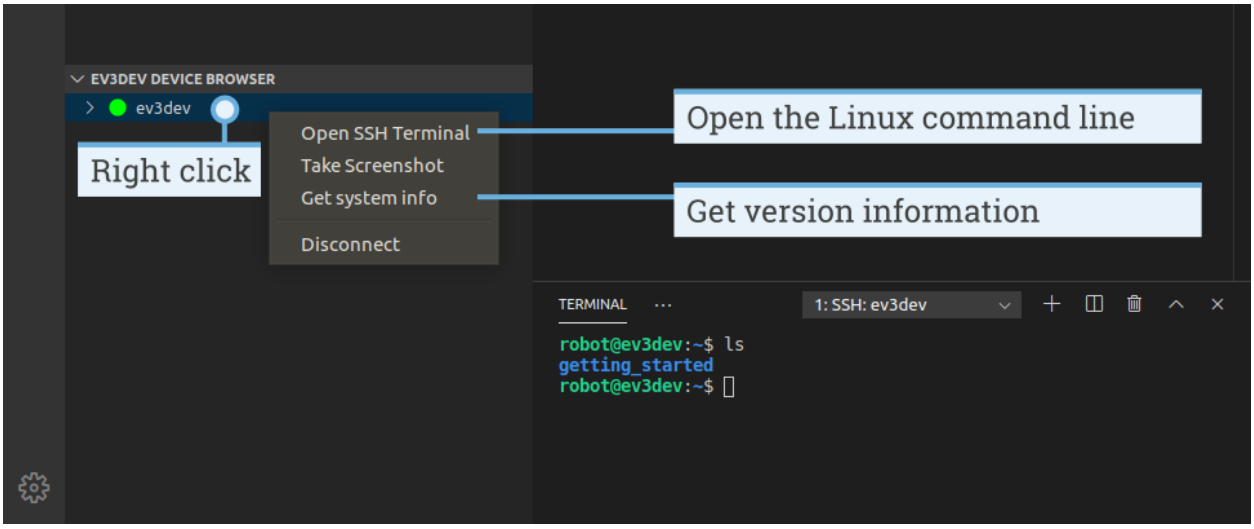


Figure 1.16: Opening the Linux command line and running the `ls` command.

For example, if you type the following command and press enter:

```
ls
```

then you will see the contents of the current folder. Figure 1.16 shows the result: it listed the project folder of the `getting_started` project that we just ran.

If you type the following command and press enter:

```
exit
```

then the command line will be closed. Alternatively, click the garbage icon shown in Figure 1.16.

You can copy text from the command line by selecting it and then pressing `ctrl shift c`. You can paste text into the command line using `ctrl shift v`.

Running commands as an administrator

Some commands require a password to run. This is similar to administrative tasks on your computer or tablet, such as installing a new app. These commands work like any other command, but you add `sudo` in front of them.

As an exercise, you can run the following command to turn the EV3 Brick off:

```
sudo poweroff
```

You will be prompted for a password. Type `maker` and then press `Enter`.

Warning: Only run commands with `sudo` if you know what you are doing.

Learning more about the command line

To learn more about the command line and many of the available commands, we recommend reading the beginner-friendly free ebook called The Linux Command Line.

To learn more about ev3dev-specific tips and tricks, visit the ev3dev website.

1.4.2 Changing the EV3 Brick name

When you search for your EV3 using Visual Studio Code, you see all EV3 Bricks listed by their name. By default, all EV3 Bricks are named *ev3dev*. Follow these steps to change that name:

1. Open Visual Studio Code and connect to your EV3 as usual.
2. Read the steps above about running commands as an administrator.
3. Think of a good name. In this example, we'll call it `autonomous-vehicle2`
4. Enter the following command and press enter:

   ```
   sudo hostnamectl set-hostname autonomous-vehicle2
   ```

5. Reboot the EV3 Brick for the change to take effect.
6. You may need to reboot your computer as well.

EV3 Brick names should only contain lowercase letters `a` through `z`, the digits `0` through `9`, and the hyphen `-`. It must start with a letter or digit. It cannot include spaces or other symbols.

1.5 Upgrading from v1.0 to v2.0

EV3 MicroPython version 2.0 was released on May 18, 2020.

This section is for users who have previously used LEGO MINDSTORMS EV3 MicroPython v1.0. We'll explain what's changed and how you can upgrade to benefit from the latest improvements.

If you are a new user and you just got started using version 2.0, you may skip this section.

1.5.1 Upgrading the microSD Card

To upgrade, download the latest microSD card file and install it using the standard *instructions*.

Note that this will erase all your existing files on the SD Card. Before you upgrade, make sure that you still have all your projects on your computer. If not, you can upload files back to your computer using *these instructions*.

As with any software update, *be careful about when you update*. For example, if you developed your code using version v1.0 and you are halfway into your robotics competition season, you may want to stick with v1.0 for now.

1.5.2 Upgrading your existing programs

Most changes in v2.0 are *new* features, like support for additional sensors. Naturally, this will not affect your existing code. However, some changes were made to existing features to improve performance.

All originally documented features in v1.0 will still work after you upgrade. This means that most programs originally made for v1.0 will work with the v2.0 microSD card image without any changes.

To try this, simply download and run your original code as you did before.

However, it is recommended that you upgrade both the microSD card and your programs at the same time to ensure everything works as expected.

The new EV3Brick() class replaces the ev3brick module

Version 2.0 introduces the *`EV3Brick()`* class. You can use it instead of the old `ev3brick` module. The old `ev3brick` module can still be used, but it is no longer recommended or documented.

The *`EV3Brick()`* class improves the speed and reliability of the EV3 screen and the EV3 speaker. It also adds functionality like speech and drawing shapes. The default font size is also bigger to make it easier to read text on the screen.

You can use the following table as a starting point to upgrade your scripts. See the *`EV3Brick()`* class documentation for complete details of all methods and arguments.

Action	v1.0	v2.0

Initialize your EV3	`from pybricks import ev3brick as brick`	`from pybricks.hubs import EV3Brick`
		
		`ev3 = EV3Brick()`

Light on	`brick.light(Color.RED)`	`ev3.light.on(Color.RED)`
Light off	`brick.light(None)`	`ev3.light.off()`
Read Buttons	`if Button.LEFT in brick.buttons():` `  print("Left is pressed.")`	`if Button.LEFT in ev3.buttons.pressed():` `  print("Left is pressed.")`
Play a beep	`brick.sound.beep()`	`ev3.speaker.beep()`
Play a sound file	`brick.sound.file(SoundFile.HELLO)`	`ev3.speaker.play_file(SoundFile.HELLO)`
Text to speech		`ev3.speaker.say("I can say anything!")`
Play notes		`ev3.speaker.play_notes(['C4/4', 'G4/4'])`
Write text at a given position	`brick.display.text("Hello!", (50, 60))`	`ev3.screen.draw_text(50, 60, "Hello!")`
Write text and scroll automatically	`brick.display.text("Hello")` `brick.display.text("world!")`	`ev3.screen.print("Hello")` `ev3.screen.print("world!")`

Change font size		`from pybricks.media.ev3dev import Font`
		
		`big_font = Font(size=24)`

		`ev3.screen.set_font(big_font)`
Display an	`from pybricks.parameters`	`from pybricks.media.`

Other internal changes to existing features

- Most methods of the *Motor()* class now have `Stop.HOLD` as the default instead of `Stop.COAST`. This improves accuracy in most applications. You can still select `Stop.COAST` if you like.
- The internal PID controllers for the motors are more accurate than before. If you give a motor command when it is already running, it smoothly adjusts the speed to the newly given command. This works even if you keep adjusting the speed in a fast loop.
- Methods to configure motor settings have changed. You can change settings using the *control* attribute now. The old settings setters continue to exist in the implementation, but they are no longer documented.
- So-called Python *keyword arguments* are now supported. Previously, you could only enter the argument *values*. For example:

  ```
  my_motor.run_angle(500, 90, Stop.HOLD, False)
  ```

  This is still possible. But you can now choose to omit optional arguments and specify others with *keywords*. This can make your code easier to read. For example:

  ```
  my_motor.run_angle(500, 90, wait=False)
  ```

- It is no longer necessary to import `pybricks.tools.print`. The `print` function is now built-in. It works just like Python or MicroPython.
- Most parameters in the *parameters* now have a specific type and representation. For example, suppose you measure a color and print the result. If you do `print(Color.RED)`, you will see the parameter instead of a technical number.
- Sound and image files have moved to a dedicated *media* module. Importing them from the old location will continue to work in this release, to make sure existing scripts will still work.

Installing an older version of the Visual Studio Code extension

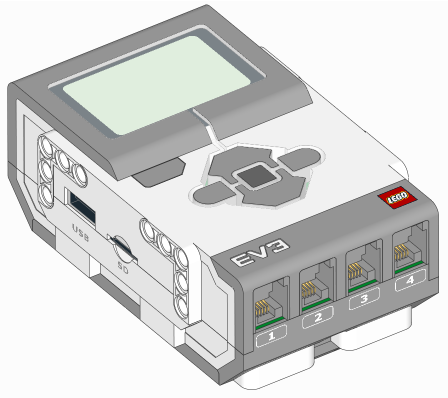
The Visual Studio Code extension and this documentation are updated automatically. You can still use your existing scripts with the updated extension. If you absolutely wish to keep the old version, look for the EV3 extension on the extension tab, click the gear icon, and click *install another version*.

CHAPTER 2

hubs – Programmable Hubs

Select your programmable hub using the buttons below.

EV3 Brick



class EV3Brick

LEGO® MINDSTORMS® EV3 Brick.

Using the buttons

buttons.**pressed**()

Checks which buttons are currently pressed.

Returns List of pressed buttons.

Return type List of *Button*

Using the brick status light

`light.`**`on`**(*color*)
Turns on the light at the specified color.

Parameters **`color`** (`Color`) – Color of the light. The light turns off if you choose `None` or a color that is not available.

Show/hide example

Example: Turn the light on and change the color.

```
#!/usr/bin/env pybricks-micropython

from pybricks.hubs import EV3Brick
from pybricks.tools import wait
from pybricks.parameters import Color

# Initialize the EV3
ev3 = EV3Brick()

# Turn on a red light
ev3.light.on(Color.RED)

# Wait
wait(1000)

# Turn the light off
ev3.light.off()
```

`light.`**`off`**()
Turns off the light.

Using the speaker

`speaker.`**`beep`**(*frequency=500*, *duration=100*)
Play a beep/tone.

Parameters

- **`frequency`** (*frequency: Hz*) – Frequency of the beep. Frequencies below 100 are treated as 100.
- **`duration`** (*time: 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, you can play: `['C4/4', 'C4/4', 'G4/4', 'G4/4']`.

Parameters

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

Show/hide musical note format

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.

speaker.**play_file**(*file_name*)

Plays a sound file.

Parameters **file_name** (*str*) – Path to the sound file, including the file extension.

speaker.**say**(*text*)

Says a given text string.

You can configure the language and voice of the text using *set_speech_options()*.

Parameters **text** (*str*) – What to say.

speaker.**set_speech_options**(*language=None*, *voice=None*, *speed=None*, *pitch=None*)

Configures speech settings used by the *say()* method.

Any option that is set to `None` will not be changed. If an option is set to an invalid value *say()* will use the default value instead.

Parameters

- **language** (*str*) – Language of the text. For example, you can choose `'en'` (English) or `'de'` (German). A list of all available languages is given below.
- **voice** (*str*) – The voice to use. For example, you can choose `'f1'` (female voice variant 1) or `'m3'` (male voice variant 3). A list of all available voices is given below.
- **speed** (*int*) – Number of words per minute.
- **pitch** (*int*) – Pitch (0 to 99). Higher numbers make the voice higher pitched and lower numbers make the voice lower pitched.

Show/hide available languages and voices

You can choose the following languages:

- `'af'`: Afrikaans
- `'an'`: Aragonese
- `'bg'`: Bulgarian
- `'bs'`: Bosnian
- `'ca'`: Catalan
- `'cs'`: Czech
- `'cy'`: Welsh
- `'da'`: Danish
- `'de'`: German

- `'el'`: Greek
- `'en'`: English (default)
- `'en-gb'`: English (United Kingdom)
- `'en-sc'`: English (Scotland)
- `'en-uk-north'`: English (United Kingdom, Northern)
- `'en-uk-rp'`: English (United Kingdom, Received Pronunciation)
- `'en-uk-wmids'`: English (United Kingdom, West Midlands)
- `'en-us'`: English (United States)
- `'en-wi'`: English (West Indies)
- `'eo'`: Esperanto
- `'es'`: Spanish
- `'es-la'`: Spanish (Latin America)
- `'et'`: Estonian
- `'fa'`: Persian
- `'fa-pin'`: Persian
- `'fi'`: Finnish
- `'fr-be'`: French (Belgium)
- `'fr-fr'`: French (France)
- `'ga'`: Irish
- `'grc'`: Greek
- `'hi'`: Hindi
- `'hr'`: Croatian
- `'hu'`: Hungarian
- `'hy'`: Armenian
- `'hy-west'`: Armenian (Western)
- `'id'`: Indonesian
- `'is'`: Icelandic
- `'it'`: Italian
- `'jbo'`: Lojban
- `'ka'`: Georgian
- `'kn'`: Kannada
- `'ku'`: Kurdish
- `'la'`: Latin
- `'lfn'`: Lingua Franca Nova
- `'lt'`: Lithuanian
- `'lv'`: Latvian

- `'mk'`: Macedonian
- `'ml'`: Malayalam
- `'ms'`: Malay
- `'ne'`: Nepali
- `'nl'`: Dutch
- `'no'`: Norwegian
- `'pa'`: Punjabi
- `'pl'`: Polish
- `'pt-br'`: Portuguese (Brazil)
- `'pt-pt'`: Portuguese (Portugal)
- `'ro'`: Romanian
- `'ru'`: Russian
- `'sk'`: Slovak
- `'sq'`: Albanian
- `'sr'`: Serbian
- `'sv'`: Swedish
- `'sw'`: Swahili
- `'ta'`: Tamil
- `'tr'`: Turkish
- `'vi'`: Vietnamese
- `'vi-hue'`: Vietnamese (Hue)
- `'vi-sgn'`: Vietnamese (Saigon)
- `'zh'`: Mandarin Chinese
- `'zh-yue'`: Cantonese Chinese

You can choose the following voices:

- `'f1'`: female variant 1
- `'f2'`: female variant 2
- `'f3'`: female variant 3
- `'f4'`: female variant 4
- `'f5'`: female variant 5
- `'m1'`: male variant 1
- `'m2'`: male variant 2
- `'m3'`: male variant 3
- `'m4'`: male variant 4
- `'m5'`: male variant 5
- `'m6'`: male variant 6

- `'m7'`: male variant 7
- `'croak'`: croak
- `'whisper'`: whisper
- `'whisperf'`: female whisper

`speaker.`**`set_volume`**(*volume*, *which='_all_'*)
Sets the speaker volume.

Parameters

- **`volume`** (*percentage: %*) – Volume of the speaker.
- **`which`** (*str*) – Which volume to set. `'Beep'` sets the volume for *beep()* and *play_notes()*. `'PCM'` sets the volume for *play_file()* and *say()*. `'_all_'` sets both at the same time.

Using the screen

`screen.`**`clear`**()
Clears the screen. All pixels on the screen will be set to *Color.WHITE*.

`screen.`**`draw_text`**(*x*, *y*, *text*, *text_color=Color.BLACK*, *background_color=None*)
Draws text on the screen.

The most recent font set using *set_font()* will be used or *Font.DEFAULT* if no font has been set yet.

Parameters

- **`x`** (*int*) – The x-axis value where the left side of the text will start.
- **`y`** (*int*) – The y-axis value where the top of the text will start.
- **`text`** (*str*) – The text to draw.
- **`text_color`** (`Color`) – The color used for drawing the text.
- **`background_color`** (`Color`) – The color used to fill the rectangle behind the text or `None` for transparent background.

`screen.`**`print`**(**args*, *sep=' '*, *end='\n'*)
Prints a line of text on the screen.

This method works like the builtin `print()` function, but it writes on the screen instead.

You can set the font using *set_font()*. If no font has been set, *Font.DEFAULT* will be used. The text is always printed used black text with a white background.

Unlike the builtin `print()`, the text does not wrap if it is too wide to fit on the screen. It just gets cut off. But if the text would go off of the bottom of the screen, the entire image is scrolled up and the text is printed in the new blank area at the bottom of the screen.

Parameters

- **`*`** (*object*) – Zero or more objects to print.
- **`sep`** (*str*) – Separator that will be placed between each object that is printed.
- **`end`** (*str*) – End of line that will be printed after the last object.

Show/hide example

Example: Say hello… in several ways.

```
#!/usr/bin/env pybricks-micropython

from pybricks.hubs import EV3Brick
from pybricks.tools import wait
from pybricks.media.ev3dev import Font

# It takes some time for fonts to load from file, so it is best to only
# load them once at the beginning of the program like this:
tiny_font = Font(size=6)
big_font = Font(size=24, bold=True)
chinese_font = Font(size=24, lang='zh-cn')


# Initialize the EV3
ev3 = EV3Brick()


# Say hello
ev3.screen.print('Hello!')

# Say tiny hello
ev3.screen.set_font(tiny_font)
ev3.screen.print('hello')

# Say big hello
ev3.screen.set_font(big_font)
ev3.screen.print('HELLO')

# Say Chinese hello
ev3.screen.set_font(chinese_font)
ev3.screen.print('你好')

# Wait some time to look at the screen
wait(5000)
```

screen.**set_font**(*font*)

Sets the font used for writing on the screen.

The font is used for both *draw_text()* and *print()*.

Parameters **font** (*Font*) – The font to use.

Example: See example in *print()*.

screen.**load_image**(*source*)

Clears this image, then draws the `source` image centered in the screen.

Parameters **source** (*Image or str*) – The source *Image*. If the argument is a string, then the `source` image is loaded from file.

Show/hide example

Example: Show an image on the screen.

```
#!/usr/bin/env pybricks-micropython

from pybricks.hubs import EV3Brick
from pybricks.tools import wait
from pybricks.media.ev3dev import Image, ImageFile

```

(continues on next page)

(continued from previous page)

```
# It takes some time to load images from the SD card, so it is best to load
# them once at the beginning of a program like this:
ev3_img = Image(ImageFile.EV3_ICON)


# Initialize the EV3
ev3 = EV3Brick()


# Show an image
ev3.screen.load_image(ev3_img)

# Wait some time to look at the image
wait(5000)
```

screen.**draw_image**(*x*, *y*, *source*, *transparent=None*)

Draws the source image on the screen.

Parameters

- **x** (*int*) – The x-axis value where the left side of the image will start.
- **y** (*int*) – The y-axis value where the top of the image will start.
- **source** (*Image or str*) – The source *Image*. If the argument is a string, then the source image is loaded from file.
- **transparent** (Color) – The color of image to treat as transparent or None for no transparency.

screen.**draw_pixel**(*x*, *y*, *color=Color.BLACK*)

Draws a single pixel on the screen.

Parameters

- **x** (*int*) – The x coordinate of the pixel.
- **y** (*int*) – The y coordinate of the pixel.
- **color** (Color) – The color of the pixel.

screen.**draw_line**(*x1*, *y1*, *x2*, *y2*, *width=1*, *color=Color.BLACK*)

Draws a line on the screen.

Parameters

- **x1** (*int*) – The x coordinate of the starting point of the line.
- **y1** (*int*) – The y coordinate of the starting point of the line.
- **x2** (*int*) – The x coordinate of the ending point of the line.
- **y2** (*int*) – The y coordinate of the ending point of the line.
- **width** (*int*) – The width of the line in pixels.
- **color** (Color) – The color of the line.

Show/hide example

Example: Draw some shapes on the screen.

```
#!/usr/bin/env pybricks-micropython

from pybricks.hubs import EV3Brick
from pybricks.tools import wait


# Initialize the EV3
ev3 = EV3Brick()


# Draw a rectangle
ev3.screen.draw_box(10, 10, 40, 40)

# Draw a solid rectangle
ev3.screen.draw_box(20, 20, 30, 30, fill=True)

# Draw a rectangle with rounded corners
ev3.screen.draw_box(50, 10, 80, 40, 5)

# Draw a circle
ev3.screen.draw_circle(25, 75, 20)

# Draw a triangle using lines
x1, y1 = 65, 55
x2, y2 = 50, 95
x3, y3 = 80, 95
ev3.screen.draw_line(x1, y1, x2, y2)
ev3.screen.draw_line(x2, y2, x3, y3)
ev3.screen.draw_line(x3, y3, x1, y1)

# Wait some time to look at the shapes
wait(5000)
```

screen.**draw_box**(*x1*, *y1*, *x2*, *y2*, *r=0*, *fill=False*, *color=Color.BLACK*)

Draws a box on the screen.

Parameters

- **x1** (*int*) – The x coordinate of the left side of the box.
- **y1** (*int*) – The y coordinate of the top of the box.
- **x2** (*int*) – The x coordinate of the right side of the box.
- **y2** (*int*) – The y coordinate of the bottom of the box.
- **r** (*int*) – The radius of the corners of the box.
- **fill** (*bool*) – If `True`, the box will be filled with `color`, otherwise only the outline of the box will be drawn.
- **color** (`Color`) – The color of the box.

Example: See example in *draw_line()*.

screen.**draw_circle**(*x*, *y*, *r*, *fill=False*, *color=Color.BLACK*)

Draws a circle on the screen.

Parameters

- **x** (*int*) – The x coordinate of the center of the circle.

- **y** (*int*) – The y coordinate of the center of the circle.
- **r** (*int*) – The radius of the circle.
- **fill** (*bool*) – If `True`, the circle will be filled with `color`, otherwise only the circumference will be drawn.
- **color** (`Color`) – The color of the circle.

Example: See example in *draw_line()*.

`screen.`**`width`**
Gets the width of the screen in pixels.

`screen.`**`height`**
Gets the height of the screen in pixels.

`screen.`**`save`**(*filename*)
Saves the screen as a `.png` file.

Parameters **filename** (*str*) – The path to the file to be saved.

Raises

- `TypeError` – `filename` is not a string.
- `OSError` – There was a problem saving the file.

Using the battery

`battery.`**`voltage`**`()`
Gets the voltage of the battery.

Returns Battery voltage.

Return type *voltage: mV*

`battery.`**`current`**`()`
Gets the current supplied by the battery.

Returns Battery current.

Return type *current: mA*

CHAPTER 3

ev3devices – EV3 Devices

LEGO® MINDSTORMS® EV3 motors and sensors.

3.1 Motors

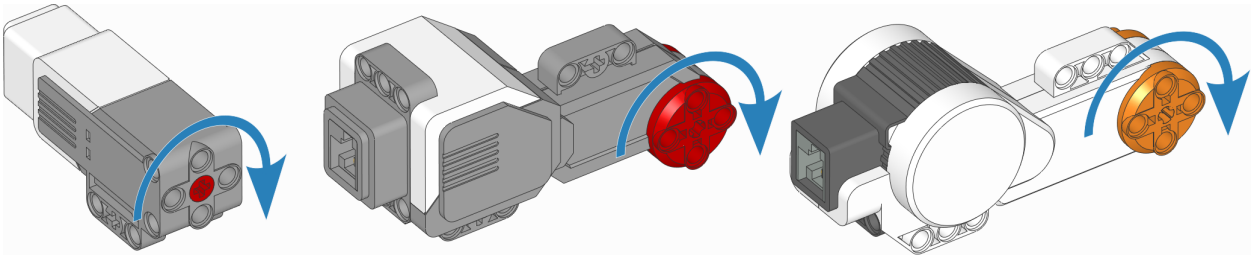


Figure 3.1: EV3-compatible motors. The arrows indicate the default positive direction.

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

Generic class to control motors with built-in 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.
- **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.

Measuring

speed()
Gets the speed of the motor.

Returns Motor speed.

Return type *rotational speed: deg/s*

angle()
Gets the rotation angle of the motor.

Returns Motor angle.

Return type *angle: deg*

reset_angle(*angle*)
Sets the accumulated rotation angle of the motor to a desired value.

Parameters **angle** (*angle: deg*) – Value to which the angle should be reset.

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.

Action

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** (*rotational speed: deg/s*) – Speed of the motor.

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** (*rotational speed: deg/s*) – Speed of the motor.
- **time** (*time: 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** (*rotational speed: deg/s*) – Speed of the motor.
- **rotation_angle** (*angle: 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 matter if speed is positive or negative.

Parameters

- **speed** (*rotational speed: deg/s*) – Speed of the motor.
- **target_angle** (*angle: 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.

run_until_stalled(*speed*, *then=Stop.COAST*, *duty_limit=None*)
Runs the motor at a constant speed until it stalls.

Parameters

- **speed** (*rotational speed: deg/s*) – Speed of the motor.
- **then** (Stop) – What to do after coming to a standstill.
- **duty_limit** (*percentage: %*) – Torque limit during this command. This is useful to avoid applying the full motor torque to a geared or lever mechanism.

Returns Angle at which the motor becomes stalled.

Return type *angle: deg*

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

This method lets you use a motor just like a simple DC motor.

Parameters **duty** (*percentage: %*) – The duty cycle (-100.0 to 100).

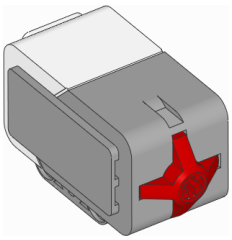
Advanced motion control

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** (*angle: deg*) – Target angle that the motor should rotate to.

control
The motors use PID control to accurately track the speed and angle targets that you specify. You can change its behavior through the `control` attribute of the motor. See *The Control Class* for an overview of available methods.

3.2 Touch Sensor



class TouchSensor(*port*)
LEGO® MINDSTORMS® EV3 Touch Sensor.

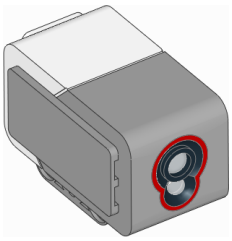
Parameters **port** (`Port`) – Port to which the sensor is connected.

pressed()
Checks if the sensor is pressed.

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

Return type bool

3.3 Color Sensor



class ColorSensor(*port*)
LEGO® MINDSTORMS® EV3 Color Sensor.

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

color()
Measures the color of a surface.

Returns `Color.BLACK`, `Color.BLUE`, `Color.GREEN`, `Color.YELLOW`, `Color.RED`, `Color.WHITE`, `Color.BROWN` or `None`.

Return type *Color*, or `None` if no color is detected.

ambient()
Measures the ambient light intensity.

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

Return type *percentage: %*

reflection()
Measures the reflection of a surface using a red light.

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

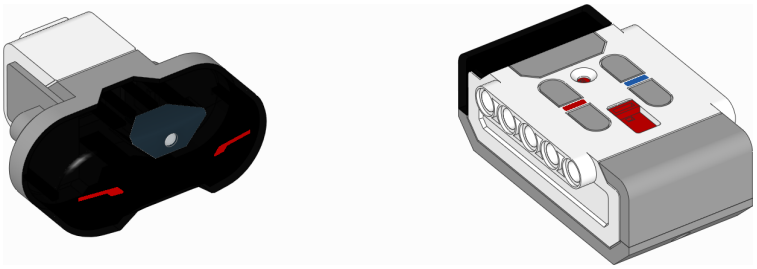
Return type *percentage: %*

rgb()
Measures the reflection of a surface using a red, green, and then a blue light.

Returns Tuple of reflections for red, green, and blue light, each ranging from 0.0 (no reflection) to 100.0 (high reflection).

Return type (*percentage: %*, *percentage: %*, *percentage: %*)

3.4 Infrared Sensor and Beacon



class InfraredSensor(*port*)
LEGO® MINDSTORMS® EV3 Infrared Sensor and Beacon.

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

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

Returns Relative distance ranging from 0 (closest) to 100 (farthest).

Return type *relative distance: %*

beacon(*channel*)
Measures the relative distance and angle between the remote and the infrared sensor.

Parameters **channel** (*int*) – Channel number of the remote.

Returns Tuple of relative distance (0 to 100) and approximate angle (-75 to 75 degrees) between remote and infrared sensor.

Return type (*relative distance: %*, *angle: deg*) or (None, None) if no remote is detected.

buttons(*channel*)
Checks which buttons on the infrared remote are pressed.

This method can detect up to two buttons at once. If you press more buttons, you may not get useful data.

Parameters **channel** (*int*) – Channel number of the remote.

Returns List of pressed buttons on the remote on selected channel.

Return type List of *Button*

keypad()
Checks which buttons on the infrared remote are pressed.

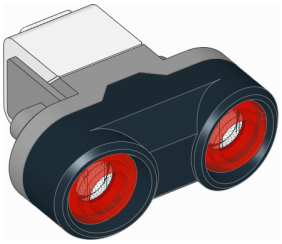
This method can independently detect all 4 up/down buttons, but it cannot detect the beacon button.

This method only works with the remote in channel 1.

Returns List of pressed buttons on the remote on selected channel.

Return type List of *Button*

3.5 Ultrasonic Sensor



class UltrasonicSensor(*port*)
LEGO® MINDSTORMS® EV3 Ultrasonic Sensor.

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

distance(*silent=False*)
Measures the distance between the sensor and an object using ultrasonic sound waves.

Parameters **silent** (*bool*) – Choose True to turn the sensor off after measuring the distance. This reduces interference with other ultrasonic sensors. If you do this too frequently, the sensor can freeze. If this happens, unplug it and plug it back in.

Returns Distance.

Return type *distance: mm*

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

If the other ultrasonic sensor is operating in silent mode, you can only detect the presence of that sensor while it is taking a measurement.

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

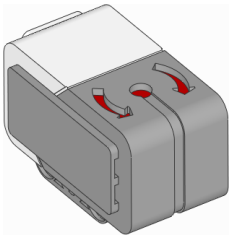
Return type bool

3.6 Gyroscopic Sensor

class GyroSensor(*port*, *positive_direction=Direction.CLOCKWISE*)
LEGO® MINDSTORMS® EV3 Gyro Sensor.

Parameters

- **port** (Port) – Port to which the sensor is connected.
- **positive_direction** (Direction) – Positive rotation direction when looking at the red dot on top of the sensor.



speed()
Gets the speed (angular velocity) of the sensor.

Returns Sensor angular velocity.

Return type *rotational speed: deg/s*

angle()
Gets the accumulated angle of the sensor.

Returns Rotation angle.

Return type *angle: deg*

If you use the `angle()` method, you cannot use the `speed()` method in the same program. Doing so would reset the sensor angle to zero every time you read the speed.

reset_angle(*angle*)
Sets the rotation angle of the sensor to a desired value.

Parameters **angle** (*angle: deg*) – Value to which the angle should be reset.

CHAPTER 4

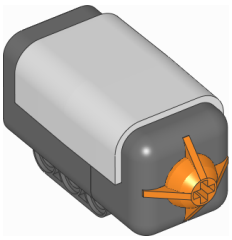
nxtdevices – NXT Devices

Use LEGO® MINDSTORMS® NXT motors and sensors with the EV3 brick.

4.1 NXT Motor

This motor works just like a LEGO MINDSTORMS EV3 Large Motor. You can use it in your programs using the *Motor* class.

4.2 NXT Touch Sensor



class TouchSensor(*port*)

LEGO® MINDSTORMS® NXT Touch Sensor.

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

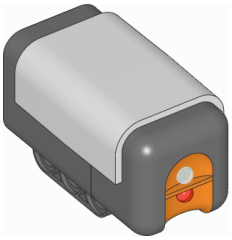
pressed()

Checks if the sensor is pressed.

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

Return type bool

4.3 NXT Light Sensor



class LightSensor(*port*)

LEGO® MINDSTORMS® NXT Color Sensor.

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

ambient()

Measures the ambient light intensity.

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

Return type *percentage: %*

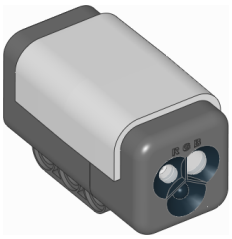
reflection()

Measures the reflection of a surface using a red light.

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

Return type *percentage: %*

4.4 NXT Color Sensor



class ColorSensor(*port*)

LEGO® MINDSTORMS® NXT Color Sensor.

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

color()

Measures the color of a surface.

Returns Color.BLACK, Color.BLUE, Color.GREEN, Color.YELLOW, Color.RED, Color.WHITE or None.

Return type *Color*, or None if no color is detected.

ambient()

Measures the ambient light intensity.

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

Return type *percentage: %*

reflection()
Measures the reflection of a surface.

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

Return type *percentage: %*

rgb()
Measures the reflection of a surface using a red, green, and then a blue light.

Returns Tuple of reflections for red, green, and blue light, each ranging from 0.0 (no reflection) to 100.0 (high reflection).

Return type (*percentage: %*, *percentage: %*, *percentage: %*)

Built-in light

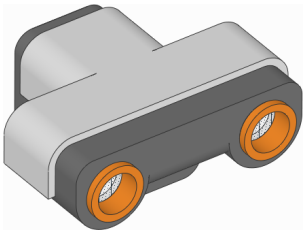
This sensor has a built-in light. You can make it red, green, blue, or turn it off.

light.**on**(*color*)
Turns on the light at the specified color.

Parameters **color** (Color) – Color of the light. The light turns off if you choose None or a color that is not available.

light.**off**()
Turns off the light.

4.5 NXT Ultrasonic Sensor



class UltrasonicSensor(*port*)
LEGO® MINDSTORMS® NXT Ultrasonic Sensor.

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

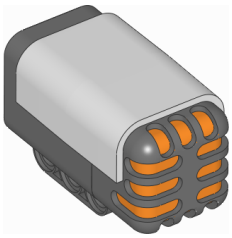
distance()
Measures the distance between the sensor and an object using ultrasonic sound waves.

Returns Distance.

Return type *distance: mm*

4.6 NXT Sound Sensor

class SoundSensor(*port*)
LEGO® MINDSTORMS® NXT Sound Sensor.



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

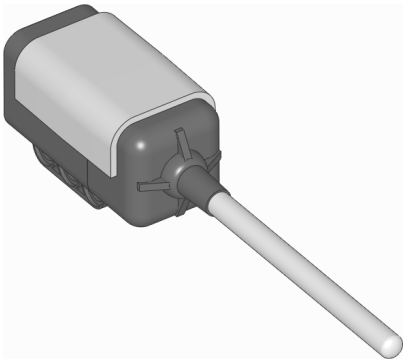
intensity(*audible_only=True*)
Measures the ambient sound intensity (loudness).

Parameters **audible_only** (*bool*) – Detect only audible sounds. This tries to filter out frequencies that cannot be heard by the human ear.

Returns Sound intensity.

Return type *percentage: %*

4.7 NXT Temperature Sensor



class TemperatureSensor(*port*)
LEGO® MINDSTORMS® NXT Temperature Sensor.

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

temperature()
Measures the temperature.

Returns Measured temperature.

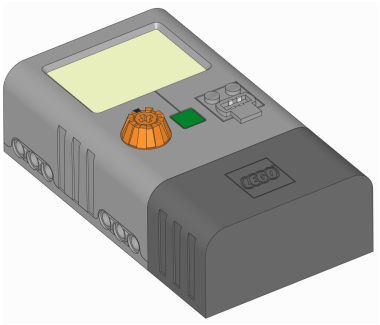
Return type *temperature: °C*

4.8 NXT Energy Meter

class EnergyMeter(*port*)
LEGO® MINDSTORMS® Education NXT Energy Meter.

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

storage()
Gets the total available energy stored in the battery.



Returns Remaining stored energy.

Return type *energy: J*

input()
Measures the electrical signals at the input (bottom) side of the energy meter. It measures the voltage applied to it and the current passing through it. The product of these two values is power. This power value is the rate at which the stored energy increases. This power is supplied by an energy source such as the provided solar panel or an externally driven motor.

Returns Voltage, current, and power measured at the input port.

Return type (*voltage: mV*, *current: mA*, *power: mW*)

output()
Measures the electrical signals at the output (top) side of the energy meter. It measures the voltage applied to the external load and the current passing to it. The product of these two values is power. This power value is the rate at which the stored energy decreases. This power is consumed by the load, such as a light or a motor.

Returns Voltage, current, and power measured at the output port.

Return type (*voltage: mV*, *current: mA*, *power: mW*)

4.9 Vernier Adapter

class VernierAdapter(*port*, *conversion=None*)
LEGO® MINDSTORMS® Education NXT/EV3 Adapter for Vernier Sensors.

Parameters

- **port** (Port) – Port to which the sensor is connected.
- **conversion** (*callable*) – Function of the format *conversion()*. This function is used to convert the raw analog voltage to the sensor-specific output value. Each Vernier Sensor has its own conversion function. The example given below demonstrates the conversion for the Surface Temperature Sensor.

voltage()
Measures the raw analog sensor voltage.

Returns Analog voltage.

Return type *voltage: mV*

conversion(*voltage*)
Converts the raw voltage (mV) to a sensor value.

If you did not provide a `conversion` function earlier, no conversion will be applied.

Parameters **voltage** (*voltage: mV*) – Analog sensor voltage

Returns Converted sensor value.

Return type float

value()

Measures the sensor *voltage()* and then applies your *conversion()* to give you the sensor value.

Returns Converted sensor value.

Return type float

Show/hide example

Example: Using the Surface Temperature Sensor.

```
#!/usr/bin/env pybricks-micropython
from pybricks.parameters import Port
from pybricks.nxtdevices import VernierAdapter

from math import log


# Conversion formula for Surface Temperature Sensor
def convert_raw_to_temperature(voltage):

    # Convert the raw voltage to the NTC resistance
    # according to the Vernier Adapter EV3 block.
    counts = voltage/5000*4096
    ntc = 15000*(counts)/(4130-counts)

    # Handle log(0) safely: make sure that ntc value is positive.
    if ntc <= 0:
        ntc = 1

    # Apply Steinhart-Hart equation as given in the sensor documentation.
    K0 = 1.02119e-3
    K1 = 2.22468e-4
    K2 = 1.33342e-7
    return 1/(K0 + K1*log(ntc) + K2*log(ntc)**3)


# Initialize the adapter on port 1
thermometer = VernierAdapter(Port.S1, convert_raw_to_temperature)

# Get the measured value and print it
temp = thermometer.value()
print(temp)
```

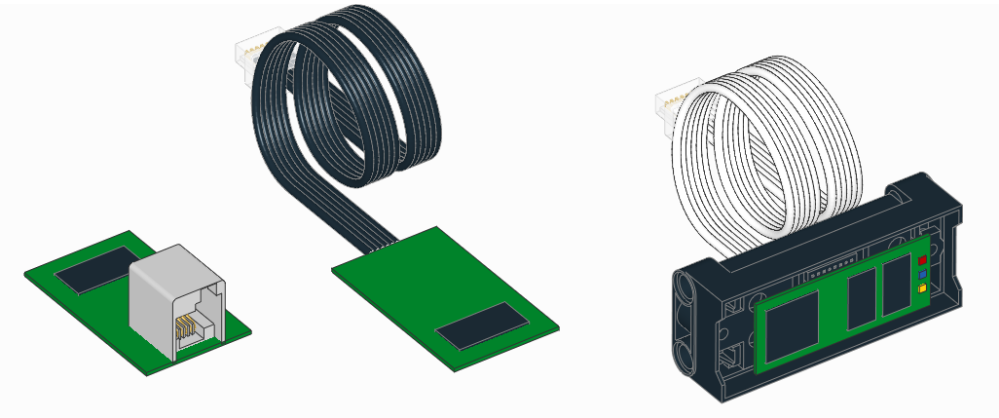
CHAPTER 5

iodevices – Generic I/O Devices

Generic input/output devices.

Note: This module provides classes to interact with unofficial motors, sensors, and other custom electronics. You should only connect custom electronics or unofficial devices if you know what you are doing. Proceed with caution.

5.1 LUMP Device



class LUMPDevice(*port*)

Devices using the LEGO UART Messaging Protocol.

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

read(*mode*)

Reads values from a given mode.

Parameters **mode** (int) – Device mode.

Returns Values read from the sensor.

Return type `tuple`

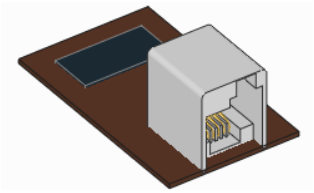
write(*mode*, *values*)
Writes values to the sensor. Only selected sensors and modes support this.

Parameters

- **mode** (`int`) – Device mode.
- **data** (`tuple`) – Values to be written.

The classes listed below are **only available on the EV3**.

5.2 Analog Sensor



class AnalogSensor(*port*)
Generic or custom analog sensor.

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

voltage()
Measures analog voltage.

Returns Analog voltage.

Return type *voltage: mV*

resistance()
Measures resistance.

This value is only meaningful if the analog device is a passive load such as a resistor or thermistor.

Returns Resistance of the analog device.

Return type *resistance: Ω*

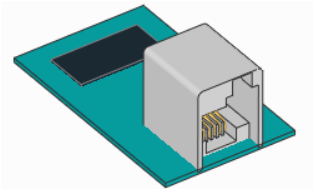
active()
Sets sensor to active mode. This sets pin 5 of the sensor port to *high*.

This is used in some analog sensors to control a switch. For example, if you use the NXT Light Sensor as a custom analog sensor, this method will turn the light on. From then on, `voltage()` returns the raw reflected light value.

passive()
Sets sensor to passive mode. This sets pin 5 of the sensor port to *low*.

This is used in some analog sensors to control a switch. For example, if you use the NXT Light Sensor as a custom analog sensor, this method will turn the light off. From then on, `voltage()` returns the raw ambient light value.

5.3 I2C Device



class I2CDevice(*port*, *address*)
Generic or custom I2C device.

Parameters

- **port** (Port) – Port to which the device is connected.
- **address** (*int*) – I2C address of the client device. See *I2C Addresses*.

read(*reg*, *length=1*)
Reads bytes, starting at a given register.

Parameters

- **reg** (int) – Register at which to begin reading: 0–255 or 0x00–0xFF.
- **length** (int) – How many bytes to read.

Returns Bytes returned from the device.

Return type bytes

write(*reg*, *data=None*)
Writes bytes, starting at a given register.

Parameters

- **reg** (int) – Register at which to begin writing: 0–255 or 0x00–0xFF.
- **data** (bytes) – Bytes to be written.

Show/hide example

Example: Read and write to an I2C device.

```
#!/usr/bin/env pybricks-micropython
from pybricks.hubs import EV3Brick
from pybricks.iodevices import I2CDevice
from pybricks.parameters import Port

# Initialize the EV3
ev3 = EV3Brick()

# Initialize I2C Sensor
device = I2CDevice(Port.S2, 0xD2 >> 1)

# Read one byte from the device.
# For this device, we can read the Who Am I
# register (0x0F) for the expected value: 211.
if 211 not in device.read(0x0F):
    raise OSError("Device is not attached")
```

(continues on next page)

(continued from previous page)

```
# To write data, create a bytes object of one
# or more bytes. For example:
# data = bytes((1, 2, 3))

# Write one byte (value 0x08) to register 0x22
device.write(0x22, bytes((0x08,)))
```

5.3.1 I2C Addresses

I2C addresses are 7-bit values. However, most vendors who make LEGO compatible sensors provide an 8-bit address in their documentation. To use those addresses, you must shift them by 1 bit. For example, if the documented address is `0xD2`, you can do `address = 0xD2 >> 1`.

5.3.2 Advanced I2C Commands

Some rudimentary I2C devices do not require a register argument or even any data. You can achieve this behavior as shown in the examples below.

Show/hide example

Example: Advanced I2C read and write techniques.

```
#!/usr/bin/env pybricks-micropython
from pybricks.hubs import EV3Brick
from pybricks.iodevices import I2CDevice
from pybricks.parameters import Port

# Initialize the EV3
ev3 = EV3Brick()

# Initialize I2C Sensor
device = I2CDevice(Port.S2, 0xD2 >> 1)

# Recommended for reading
result, = device.read(reg=0x0F, length=1)

# Read 1 byte from no particular register:
device.read(reg=None, length=1)

# Read 0 bytes from no particular register:
device.read(reg=None, length=0)

# I2C write operations consist of a register byte followed
# by a series of data bytes. Depending on your device, you
# can choose to skip the register or data as follows:

# Recommended for writing:
device.write(reg=0x22, data=b'\x08')

# Write 1 byte to no particular register:
device.write(reg=None, data=b'\x08')

# Write 0 bytes to a particular register:
device.write(reg=0x08, data=None)
```

(continues on next page)

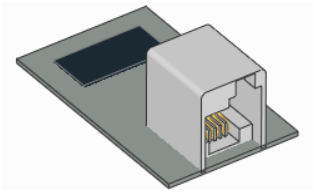
(continued from previous page)

```
# Write 0 bytes to no particular register:
device.write(reg=None, data=None)
```

Additional technical resources

The `I2CDevice` class methods call functions from the Linux SMBus driver. To find out which commands are called under the hood, check the Pybricks source code. More details about using I2C without MicroPython can be found on the ev3dev I2C page.

5.4 UART Device



class UARTDevice(*port*, *baudrate*, *timeout=None*)

Generic UART device.

Parameters

- **port** (`Port`) – Port to which the device is connected.
- **baudrate** (*int*) – Baudrate of the UART device.
- **timeout** (*time: ms*) – How long to wait during *read()* before giving up. If you choose `None`, it will wait forever.

read(*length=1*)

Reads a given number of bytes from the buffer.

Your program will wait until the requested number of bytes are received. If this takes longer than `timeout`, the `ETIMEDOUT` exception is raised.

Parameters **length** (`int`) – How many bytes to read.

Returns Bytes returned from the device.

Return type `bytes`

read_all()

Reads all bytes from the buffer.

Returns Bytes returned from the device.

Return type `bytes`

write(*data*)

Writes bytes.

Parameters **data** (`bytes`) – Bytes to be written.

waiting()

Gets how many bytes are still waiting to be read.

Returns Number of bytes in the buffer.

Return type int

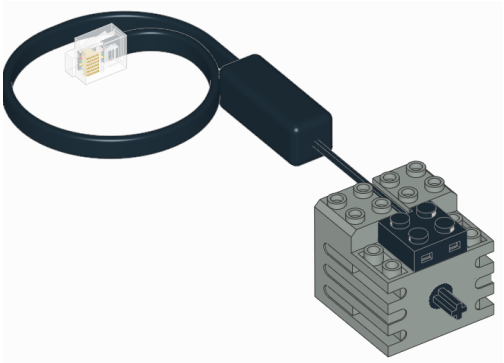
clear()
Empties the buffer.

Show/hide example

Example: Read and write to a UART device.

```
#!/usr/bin/env pybricks-micropython
from pybricks.hubs import EV3Brick
from pybricks.iodevices import UARTDevice
from pybricks.parameters import Port
from pybricks.media.ev3dev import SoundFile

# Initialize the EV3
ev3 = EV3Brick()

# Initialize sensor port 2 as a uart device
ser = UARTDevice(Port.S2, baudrate=115200)

# Write some data
ser.write(b'\r\nHello, world!\r\n')

# Play a sound while we wait for some data
for i in range(3):
    ev3.speaker.play_file(SoundFile.HELLO)
    ev3.speaker.play_file(SoundFile.GOOD)
    ev3.speaker.play_file(SoundFile.MORNING)
    print("Bytes waiting to be read:", ser.waiting())

# Read all data received while the sound was playing
data = ser.read_all()
print(data)
```

5.5 DC Motor



class DCMotor(*port*, *positive_direction=Direction.CLOCKWISE*)
Generic class to control simple motors without rotation sensors, such as train motors.

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.

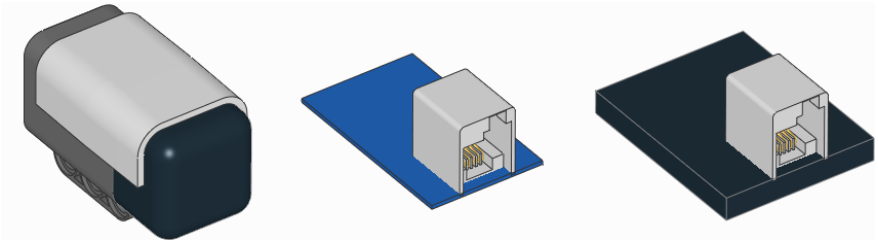
dc(*duty*)
Rotates the motor at a given duty cycle (also known as "power").

Parameters **duty** (*percentage: %*) – The duty cycle (-100.0 to 100).

stop()
Stops the motor and lets it spin freely.

The motor gradually stops due to friction.

5.6 Ev3dev sensors



EV3 MicroPython is built on top of ev3dev, which means that a sensor may be supported even if it is not listed in this documentation. If so, you can use it with the `Ev3devSensor` class. This is easier and faster than using the custom device classes given above.

To check whether you can use the `Ev3devSensor` class:

- Plug the sensor into your EV3 Brick.
- Go to the main menu of the EV3 Brick.
- Select *Device Browser* and then *Sensors*.
- If your sensor shows up, you can use it.

Now select your sensor from the menu and choose *set mode*. This shows all available modes for this sensor. You can use these mode names as the `mode` setting below.

To learn more about compatible devices and what each mode does, visit the ev3dev sensors page.

class Ev3devSensor(*port*)
Read values of an ev3dev-compatible sensor.

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

sensor_index
Index of the ev3dev sysfs lego-sensor class.

port_index
Index of the ev3dev sysfs lego-port class.

read(*mode*)
Reads values at a given mode.

Parameters **mode** (str) – Mode name.

Returns Values read from the sensor.

Return type tuple

Show/hide example: Reading values with the Ev3devSensor class

Example

In this example we use the LEGO MINDSTORMS EV3 Color Sensor with the raw RGB mode. This gives uncalibrated red, green, and blue reflection values.

```
#!/usr/bin/env pybricks-micropython
from pybricks.parameters import Port
from pybricks.tools import wait
from pybricks.iodevices import Ev3devSensor

# Initialize an Ev3devSensor.
# In this example we use the
# LEGO MINDSTORMS EV3 Color Sensor.
sensor = Ev3devSensor(Port.S3)

while True:
    # Read the raw RGB values
    r, g, b = sensor.read('RGB-RAW')

    # Print results
    print('R: {0}\t G: {1}\t B: {2}'.format(r, g, b))

    # Wait
    wait(200)
```

Show/hide example: Extending the Ev3devSensor class

Example

This example shows how to extend the `Ev3devSensor` class by accessing additional features found in the Linux system folder for this device.

```
#!/usr/bin/env pybricks-micropython
from pybricks.parameters import Port
from pybricks.iodevices import Ev3devSensor


class MySensor(Ev3devSensor):
    """Example of extending the Ev3devSensor class."""

    def __init__(self, port):
        """Initialize the sensor."""

        # Initialize the parent class.
        super().__init__(port)

        # Get the sysfs path.
        self.path = '/sys/class/lego-sensor/sensor' + str(self.sensor_index)

    def get_modes(self):
        """Get a list of mode strings so we don't have to look them up."""

        # The path of the modes file.
        modes_path = self.path + '/modes'

        # Open the modes file.
        with open(modes_path, 'r') as m:
```

(continues on next page)

(continued from previous page)

```
            # Read the contents.
            contents = m.read()

            # Strip the newline symbol, and split at every space symbol.
            return contents.strip().split(' ')


# Initialize the sensor
sensor = MySensor(Port.S3)

# Show where this sensor can be found
print(sensor.path)

# Print the available modes
modes = sensor.get_modes()
print(modes)

# Read mode 0 of this sensor
val = sensor.read(modes[0])
print(val)
```

CHAPTER 6

parameters – Parameters and Constants

Constant parameters/arguments for the Pybricks API.

class Port

Port on the programmable brick or hub.

Motor ports:

A

B

C

D

Sensor ports:

S1

S2

S3

S4

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 following diagrams:

- Clockwise direction for *EV3/NXT motors*

class Stop

Action after the motor stops: coast, brake, or hold.

COAST

Let the motor move freely.

BRAKE

Passively resist small external forces.

HOLD

Keep controlling the motor to hold it at the commanded angle. This is only available on motors with encoders.

The following table show how each stop type adds 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)`

class Color

Light or surface color.

BLACK

BLUE

GREEN

YELLOW

RED

WHITE

BROWN

ORANGE

PURPLE

class Button

Buttons on a brick or remote:

LEFT_DOWN

DOWN

RIGHT_DOWN

LEFT

CENTER

RIGHT

LEFT_UP

UP

BEACON

RIGHT_UP

LEFT_UP	UP/BEACON	RIGHT_UP
LEFT	CENTER	RIGHT
LEFT_DOWN	DOWN	RIGHT_DOWN

CHAPTER 7

tools – Timing and Data logging

Common tools for timing and data logging.

wait(*time*)
Pauses the user program for a specified amount of time.

Parameters **time** (*time: ms*) – How long to wait.

class StopWatch
A stopwatch to measure time intervals. Similar to the stopwatch feature on your phone.

time()
Gets the current time of the stopwatch.

Returns Elapsed time.

Return type *time: ms*

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).

class DataLog(**headers*, *name='log'*, *timestamp=True*, *extension='csv'*, *append=False*)
Create a file and log data.

Parameters

- **headers** (*col1*, *col2*, ...) – Column headers. These are the names of the data columns. For example, choose `'time'` and `'angle'`.

- **name** (*str*) – Name of the file.
- **timestamp** (*bool*) – Choose `True` to add the date and time to the file name. This way, your file has a unique name. Choose `False` to omit the timestamp.
- **extension** (*str*) – File extension.
- **append** (*bool*) – Choose `True` to reopen an existing data log file and append data to it. Choose `False` to clear existing data. If the file does not exist yet, an empty file will be created either way.

log(**values*)

Saves one or more values on a new line in the file.

Parameters **values** (object, object, . . .) – One or more objects or values.

By default, this class creates a `csv` file on the EV3 brick with the name `log` and the current date and time. For example, if you use this class on 13 February 2020 on 10:07 and 44.431260 seconds, the file is called `log_2020_02_13_10_07_44_431260.csv`.

See *managing files on the EV3* to learn how to upload the log file back to your computer.

Show/hide example: Logging and visualizing measurements

Example

This example shows how to log the angle of a rotating wheel as time passes.

```
#!/usr/bin/env pybricks-micropython
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import DataLog, StopWatch, wait

# Create a data log file in the project folder on the EV3 Brick.
# * By default, the file name contains the current date and time, for example:
#   log_2020_02_13_10_07_44_431260.csv
# * You can optionally specify the titles of your data columns. For example,
#   if you want to record the motor angles at a given time, you could do:
data = DataLog('time', 'angle')

# Initialize a motor and make it move
wheel = Motor(Port.B)
wheel.run(500)

# Start a stopwatch to measure elapsed time
watch = StopWatch()

# Log the time and the motor angle 10 times
for i in range(10):
    # Read angle and time
    angle = wheel.angle()
    time = watch.time()

    # Each time you use the log() method, a new line with data is added to
    # the file. You can add as many values as you like.
    # In this example, we save the current time and motor angle:
    data.log(time, angle)

    # Wait some time so the motor can move a bit
    wait(100)
```

(continues on next page)

(continued from previous page)

```
# You can now upload your file to your computer
```

In this example, the generated file has the following contents:

```
time, angle
3, 0
108, 6
212, 30
316, 71
419, 124
523, 176
628, 228
734, 281
838, 333
942, 385
```

When you upload the file to your computer as shown above, you can open it in a spreadsheet editor. You can then generate a graph of the data, as shown in Figure 7.1.

In this example, we see that the motor angle changes slowly at first. Then the angle begins to change faster, and the graph becomes a straight line. This means that the motor has reached a constant speed. You can verify that the angle increases by 500 degrees per second.

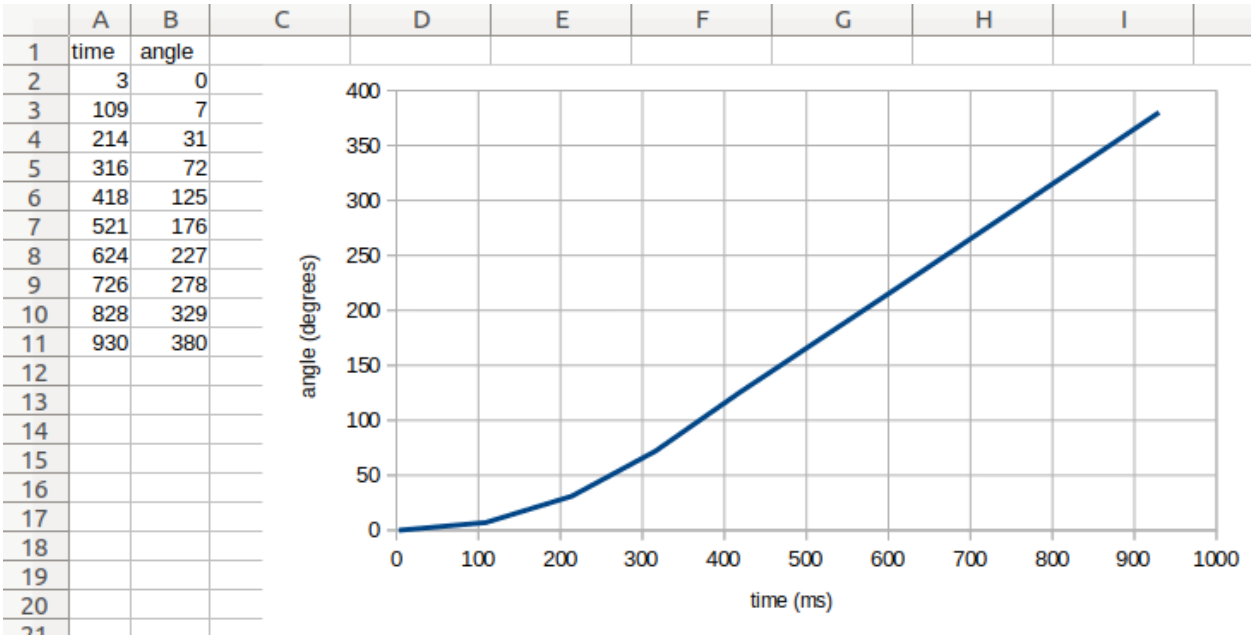


Figure 7.1: Original file contents (left) and a generated graph (right).

Show/hide example: Using the optional arguments

Example

This example shows how to log data beyond just numbers. It also shows how you can use the optional arguments of the `DataLog` class to choose the file name and extension.

In this example, `timestamp=False`, which means that the date and time are not added to the file name. This can be convenient because the file name will always be the same. However, this means that the contents of `my_file.txt` will be overwritten every time you run this script.

```
#!/usr/bin/env pybricks-micropython
from pybricks.parameters import Color
from pybricks.tools import DataLog

# Create a data log file called my_file.txt
data = DataLog('time', 'angle', name='my_file', timestamp=False, extension='txt')

# The log method uses the print() method to add a line of text.
# So, you can do much more than saving numbers. For example:
data.log('Temperature', 25)
data.log('Sunday', 'Monday', 'Tuesday')
data.log({'Kiwi': Color.GREEN}, {'Banana': Color.YELLOW})

# You can upload the file to your computer, but you can also print the data:
print(data)
```

CHAPTER 8

robotics – Robotics

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 and 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.

Parameters

- **left_motor** (Motor) – The motor that drives the left wheel.
- **right_motor** (Motor) – The motor that drives the right wheel.
- **wheel_diameter** (*dimension: mm*) – Diameter of the wheels.
- **axle_track** (*dimension: mm*) – Distance between the points where both wheels touch the ground.

Driving for a given distance or by an 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*)

Drives straight for a given distance and then stops.

Parameters **distance** (*distance: mm*) – Distance to travel.

turn(*angle*)

Turns in place by a given angle and then stops.

Parameters **angle** (*angle: deg*) – Angle of the turn.

settings(*straight_speed*, *straight_acceleration*, *turn_rate*, *turn_acceleration*)
Configures the speed and acceleration used by `straight()` and `turn()`.

If you give no arguments, this returns the current values as a tuple.

You can only change the settings while the robot is stopped. This is either before you begin driving or after you call `stop()`.

Parameters

- **straight_speed** (*speed: mm/s*) – Speed of the robot during `straight()`.
- **straight_acceleration** (*linear acceleration: mm/s/s*) – Acceleration and deceleration of the robot at the start and end of `straight()`.
- **turn_rate** (*rotational speed: deg/s*) – Turn rate of the robot during `turn()`.
- **turn_acceleration** (*rotational acceleration: deg/s/s*) – Angular acceleration and deceleration of the robot at the start and end of `turn()`.

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(*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

- **drive_speed** (*speed: mm/s*) – Speed of the robot.
- **turn_rate** (*rotational speed: deg/s*) – Turn rate of the robot.

stop()
Stops the robot by letting the motors spin freely.

Measuring

distance()
Gets the estimated driven distance.

Returns Driven distance since last reset.

Return type *distance: mm*

angle()
Gets the estimated rotation angle of the drive base.

Returns Accumulated angle since last reset.

Return type *angle: deg*

state()
Gets the state of the robot.

This returns the current `distance()`, the drive speed, the `angle()`, and the turn rate.

Returns Distance, drive speed, angle, turn rate

Return type (*distance: mm*, *speed: mm/s*, *angle: deg*, *rotational speed: deg/s*)

reset()
Resets the estimated driven distance and angle to 0.

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.

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

Suppose you make a *DriveBase* object using two `Motor` objects called `left_motor` and `right_motor`. You **cannot** use these motors individually while the DriveBase is **active**.

The DriveBase is active if it is driving, but also when it is actively holding the wheels in place after a *straight()* or *turn()* command. To deactivate the *DriveBase*, call *stop()*.

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 setttings.

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 Control Class* for an overview of available methods.

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 Control Class* for an overview of available methods.

CHAPTER 9

media – Sounds and Images

This module describes media such as sound and images that you can use in your projects. Media are divided into submodules that indicate on which platform they are available.

9.1 media.ev3dev – Sounds and Images

EV3 MicroPython is built on top of ev3dev, which comes with a variety of image and sound files. You can access them using the classes below.

You can also use your own sound and image files by placing them in your project folder.

9.1.1 Image Files

class ImageFile

Paths to standard EV3 images.

Information

ACCEPT

BACKWARD

DECLINE

FORWARD

LEFT

NO_GO

QUESTION_MARK

RIGHT

STOP_1

STOP_2

THUMBS_DOWN

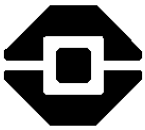
THUMBS_UP

WARNING

LEGO

EV3

EV3_ICON



Objects

TARGET

Eyes

ANGRY

AWAKE

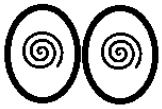
BOTTOM_LEFT

BOTTOM_RIGHT

CRAZY_1

CRAZY_2

DIZZY



DOWN

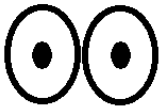
EVIL

KNOCKED_OUT

MIDDLE_LEFT

MIDDLE_RIGHT

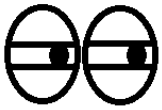
NEUTRAL



PINCHED_LEFT

PINCHED_MIDDLE

PINCHED_RIGHT



SLEEPING

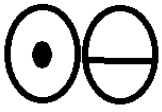
TIRED_LEFT

TIRED_MIDDLE

TIRED_RIGHT

UP

WINKING



9.1.2 Sound Files

class SoundFile

Paths to standard EV3 sounds.

Expressions

BOING

Download

BOO

Download

CHEERING

Download

CRUNCHING

Download

CRYING

Download

FANFARE

Download

KUNG_FU

Download

LAUGHING_1

Download

LAUGHING_2

Download

MAGIC_WAND

Download

OUCH

Download

SHOUTING

Download

SMACK

Download

SNEEZING

Download

SNORING

Download

UH_OH

Download

Information

ACTIVATE

Download

ANALYZE

Download

BACKWARDS

Download

COLOR

Download

DETECTED

Download

DOWN

Download

ERROR

Download

ERROR_ALARM

Download

FLASHING

Download

FORWARD

Download

LEFT

Download

OBJECT

Download

RIGHT

Download

SEARCHING

Download

START

Download

STOP

Download

TOUCH

Download

TURN

Download

UP

Download

Communication

BRAVO

Download

EV3

Download

FANTASTIC

Download

GAME_OVER

Download

GO

Download

GOOD_JOB

Download

GOOD

Download

GOODBYE

Download

HELLO

Download

HI

Download

LEGO

Download

MINDSTORMS

Download

MORNING

Download

NO

Download

OKAY

Download

OKEY_DOKEY

Download

SORRY

Download

THANK_YOU

Download

YES

Download

Movement sounds

SPEED_DOWN

Download

SPEED_IDLE

Download

SPEED_UP

Download

Colors

BLACK

Download

BLUE

Download

BROWN

Download

GREEN

Download

RED

Download

WHITE

Download

YELLOW

Download

Mechanical

AIR_RELEASE

Download

AIRBRAKE

Download

BACKING_ALERT

Download

HORN_1

Download

HORN_2

Download

LASER

Download

MOTOR_IDLE

Download

MOTOR_START

Download

MOTOR_STOP

Download

RATCHET

Download

SONAR

Download

TICK_TACK

Download

Animal sounds

CAT_PURR

Download

DOG_BARK_1

Download

DOG_BARK_2

Download

DOG_GROWL

Download

DOG_SNIFF

Download

DOG_WHINE

Download

ELEPHANT_CALL

Download

INSECT_BUZZ_1

Download

INSECT_BUZZ_2

Download

INSECT_CHIRP

Download

SNAKE_HISS

Download

SNAKE_RATTLE

Download

T_REX_ROAR

Download

Numbers

ZERO

Download

ONE

Download

TWO

Download

THREE

Download

FOUR

Download

FIVE

Download

SIX

Download

SEVEN

Download

EIGHT

Download

NINE

Download

TEN

Download

System sounds

CLICK

Download

CONFIRM

Download

GENERAL_ALERT

Download

OVERPOWER

Download

READY

Download

9.1.3 Fonts

class Font (*family=None*, *size=12*, *bold=False*, *monospace=False*, *lang=None*, *script=None*)

Object that represents a font for writing text.

The font object will be a font that is the "best" match based on the parameters given and available fonts installed.

Parameters

- **family** (*str*) – The preferred font family or `None` to use the default value.
- **size** (*int*) – The preferred font size. Most fonts have sizes between 6 and 24. This is the "point" size and not the same as *height*.
- **bold** (*bool*) – When `True`, prefer bold fonts.
- **monospace** (*bool*) – When `True` prefer monospaced fonts. This is useful for aligning multiple rows of text.
- **lang** (*str*) – A language code, such as `'en'` or `'zh-cn'` or `None` to use the default language.[1]

[1] Language codes

Note: Languages depend on installed fonts. Additional language codes are possible and some listed language codes may not have a satisfactory font.

- `'aa'`: Afar
- `'af'`: Afrikaans
- `'an'`: Aragonese
- `'av'`: Avaric
- `'ay'`: Aymara
- `'az-az'`: Azerbaijani
- `'be'`: Belarusian
- `'bg'`: Bulgarian
- `'bi'`: Bislama
- `'bm'`: Bambara
- `'br'`: Breton
- `'bs'`: Bosnian
- `'bua'`: Buriat
- `'ca'`: Catalan
- `'ce'`: Chechen

- **script** (*str*) – A unicode script identifier such as `'Runr'` or `None`.

DEFAULT = Font('Lucida', 12)
The default font.

family
Gets the family name of the font.

- `'ch'`: Chamorro
- `'co'`: Corsican
- `'crh'`: Crimean
- `'cs'`: Czech
- `'csb'`: Kashubian
- `'cy'`: Welsh
- `'da'`: Danish
- `'de'`: German
- `'ee'`: Ewe
- `'el'`: Greek
- `'en'`: English
- `'eo'`: Esperanto
- `'es'`: Spanish
- `'et'`: Estonian
- `'eu'`: Basque
- `'ff'`: Fulah
- `'fi'`: Finnish
- `'fil'`: Filipino
- `'fj'`: Fijian
- `'fo'`: Faroese
- `'fr'`: French
- `'fur'`: Friulian
- `'fy'`: Western Frisian
- `'ga'`: Irish
- `'gd'`: Gaelic
- `'gl'`: Galician
- `'gv'`: Manx
- `'ha'`: Hausa
- `'haw'`: Hawaiian
- `'he'`: Hebrew
- `'ho'`: Hiri Motu
- `'hr'`: Croatian
- `'hsb'`: Upper Sorbian
- `'ht'`: Haitian
- `'hu'`: Hungarian
- `'ia'`: Interlingua
- `'id'`: Indonesian

style
Gets a string describing the font style.

Can be “Regular” or “Bold”.

width
Gets the width of the widest character of the font.

- `'ie'`: Interlingue
- `'ik'`: Inupiaq
- `'io'`: Ido
- `'is'`: Icelandic
- `'it'`: Italian
- `'ja'`: Japanese
- `'jv'`: Javanese
- `'ki'`: Kikuyu
- `'kj'`: Kuanyama
- `'kl'`: Kalaallisut
- `'ko'`: Korean
- `'ku-tr'`: Kurdish
- `'kum'`: Kumyk
- `'kw'`: Cornish
- `'kwm'`: Kwambi
- `'la'`: Latin
- `'lb'`: Luxembourgish
- `'lez'`: Lezghian
- `'lg'`: Ganda
- `'li'`: Limburgan
- `'ln'`: Lingala
- `'lt'`: Lithuanian
- `'lv'`: Latvian
- `'mg'`: Malagasy
- `'mh'`: Marshallese
- `'mi'`: Maori
- `'mk'`: Macedonian
- `'mn-mn'`: Mongolian
- `'mo'`: Moldavian
- `'ms'`: Malay
- `'mt'`: Maltese
- `'na'`: Nauru
- `'nb'`: Norwegian Bokmål
- `'nds'`: Low German
- `'ng'`: Ndonga
- `'nl'`: Dutch
- `'nn'`: Norwegian Nynorsk

height
Gets the height of the font.

text_width(*text*)
Gets the width of the text when the text is drawn using this font.

Parameters **text** (*str*) – The text.

- `'no'`: Norwegian
- `'nr'`: South Ndebele
- `'nso'`: Northern Sotho
- `'nv'`: Navajo
- `'ny'`: Chichewa
- `'oc'`: Occitan
- `'om'`: Oromo
- `'os'`: Ossetian
- `'pap-an'`: Papiamento, Netherlands Antilles
- `'pap-aw'`: Papiamento, Aruba
- `'pl'`: Polish
- `'pt'`: Portuguese
- `'qu'`: Quechua
- `'quz'`: Cusco Quechua
- `'rm'`: Romansh
- `'rn'`: Rundi
- `'ro'`: Romanian
- `'ru'`: Russian
- `'rw'`: Kinyarwanda
- `'sc'`: Sardinian
- `'sco'`: Scots
- `'se'`: Northern Sami
- `'sel'`: Selkup
- `'sg'`: Sango
- `'sk'`: Slovak
- `'sl'`: Slovenian
- `'sm'`: Samoan
- `'sma'`: Southern Sami
- `'smj'`: Lule Sami
- `'smn'`: Inari Sami
- `'sms'`: Skolt Sami
- `'sn'`: Shona
- `'so'`: Somali
- `'sq'`: Albanian
- `'sr'`: Serbian
- `'ss'`: Swati
- `'st'`: Southern Sotho

Returns The width in pixels.

Return type int

text_height(*text*)

Gets the height of the text when the text is drawn using this font.

Parameters **text** (*str*) – The text.

Returns The height in pixels.

Return type int

Exploring more fonts

Behind the scenes, Pybricks uses Fontconfig for fonts. The Fontconfig command line tools can be used to explore available fonts in more detail. To do so, go to the ev3dev device browser, right click on your EV3 brick, and click *Open SSH Terminal*. Then you can enter one of these commands:

```
# List available font families.
fc-list :scalable=false family
# Perform lookup similar to Font.DEFAULT
fc-match :scalable=false:dpi=119:family=Lucida:size=12
```

(continues on next page)

- `'su'`: Sundanese
- `'sv'`: Swedish
- `'sw'`: Swahili
- `'tk'`: Turkmen
- `'tl'`: Tagalog
- `'tn'`: Tswana
- `'to'`: Tonga
- `'tr'`: Turkish
- `'ts'`: Tsonga
- `'ty'`: Tahitian
- `'uk'`: Ukrainian
- `'uz'`: Uzbek
- `'vo'`: Volapük
- `'vot'`: Votic
- `'wa'`: Walloon
- `'wen'`: Sorbian
- `'wo'`: Wolof
- `'xh'`: Xhosa
- `'yap'`: Yapese
- `'yi'`: Yiddish
- `'za'`: Zhuang
- `'zh-cn'`: Chinese, China
- `'zh-sg'`: Chinese, Singapore
- `'zh-tw'`: Chinese, Taiwan
- `'zu'`: Zulu

(continued from previous page)

```
# Perform lookup similar to Font(size=24,lang=zh-cn)
fc-match :scalable=false:dpi=119:size=24:lang=zh-cn
```

Pybricks only allows the use of bitmap fonts (`scalable=false`) and the screen on the EV3 has 119 pixels per inch (`dpi=119`).

9.1.4 Image Manipulation

Instead of drawing directly on the EV3 screen, you can make and interact with image files using the `Image` class given below.

class Image(*source*, *sub=False*)

Object representing a graphics image. This can either be an in-memory copy of an image or the image displayed on a screen.

Parameters

- **source** (*str or Image*) – The source of the image.

  If `source` is a string, then the image will be loaded from the file path given by the string. Only `.png` files are supported. As a special case, if the string is `_screen_`, the image will be configured to draw directly on the screen.

  If an *Image* is given, the new object will contain a copy of the `source` image object.

- **sub** (*bool*) – If `sub` is `True`, then the image object will act as a sub-image of the `source` image (this only works if the type of `source` is *Image* and not when it is a `str`).

  Additional keyword arguments `x1`, `y1`, `x2`, `y2` are needed when `sub=True`. These specify the top-left and bottom-right coordinates in the `source` image that will be used as the bounds for the sub-image.

static empty(*width=<screen width>*, *height=<screen height>*)

Creates a new empty *Image* object.

Parameters

- **width** (*int*) – The width of the image in pixels.
- **height** (*int*) – The height of the image in pixels.

Returns A new image with all pixels set to *Color.WHITE*.

Return type *Image*

Raises

- `TypeError` – `width` or `height` is not a number.
- `ValueError` – `width` or `height` is less than 1.
- `RuntimeError` – There was a problem allocating a new image.

Drawing text

There are two ways to draw text on images. *draw_text()* lets text be placed precisely on the image or *print()* can be used to automatically print text on a new line.

draw_text(*x*, *y*, *text*, *text_color=Color.BLACK*, *background_color=None*)
Draws text on this image.

The most recent font set using *set_font()* will be used or *Font.DEFAULT* if no font has been set yet.

Parameters

- **x** (*int*) – The x-axis value where the left side of the text will start.
- **y** (*int*) – The y-axis value where the top of the text will start.
- **text** (*str*) – The text to draw.
- **text_color** (Color) – The color used for drawing the text.
- **background_color** (Color) – The color used to fill the rectangle behind the text or None for transparent background.

print(**args*, *sep=' '*, *end='\n'*)
Prints a line of text on this image.

This method works like the builtin `print()` function, but it writes on this image instead.

You can set the font using *set_font()*. If no font has been set, *Font.DEFAULT* will be used. The text is always printed used black text with a white background.

Unlike the builtin `print()`, the text does not wrap if it is too wide to fit on this image. It just gets cut off. But if the text would go off of the bottom of this image, the entire image is scrolled up and the text is printed in the new blank area at the bottom of this image.

Parameters

- ***** (*object*) – Zero or more objects to print.
- **sep** (*str*) – Separator that will be placed between each object that is printed.
- **end** (*str*) – End of line that will be printed after the last object.

set_font(*font*)
Sets the font used for writing on this image.

The font is used for both *draw_text()* and *print()*.

Parameters **font** (*Font*) – The font to use.

Drawing images

A copy of another image can be drawn on an image. Also consider using sub-images to copy part of an image.

draw_image(*x*, *y*, *source*, *transparent=None*)
Draws the `source` image on this image.

Parameters

- **x** (*int*) – The x-axis value where the left side of the image will start.
- **y** (*int*) – The y-axis value where the top of the image will start.
- **source** (Image *or* *str*) – The source *Image*. If the argument is a string, then the `source` image is loaded from file.
- **transparent** (Color) – The color of `image` to treat as transparent or None for no transparency.

Drawing shapes

These are the methods to draw basic shapes, including points, lines, rectangles and circles.

draw_pixel(*x*, *y*, *color=Color.BLACK*)
Draws a single pixel on this image.

Parameters

- **x** (*int*) – The x coordinate of the pixel.
- **y** (*int*) – The y coordinate of the pixel.
- **color** (Color) – The color of the pixel.

draw_line(*x1*, *y1*, *x2*, *y2*, *width=1*, *color=Color.BLACK*)
Draws a line on this image.

Parameters

- **x1** (*int*) – The x coordinate of the starting point of the line.
- **y1** (*int*) – The y coordinate of the starting point of the line.
- **x2** (*int*) – The x coordinate of the ending point of the line.
- **y2** (*int*) – The y coordinate of the ending point of the line.
- **width** (*int*) – The width of the line in pixels.
- **color** (Color) – The color of the line.

draw_box(*x1*, *y1*, *x2*, *y2*, *r=0*, *fill=False*, *color=Color.BLACK*)
Draws a box on this image.

Parameters

- **x1** (*int*) – The x coordinate of the left side of the box.
- **y1** (*int*) – The y coordinate of the top of the box.
- **x2** (*int*) – The x coordinate of the right side of the box.
- **y2** (*int*) – The y coordinate of the bottom of the box.
- **r** (*int*) – The radius of the corners of the box.
- **fill** (*bool*) – If `True`, the box will be filled with `color`, otherwise only the outline of the box will be drawn.
- **color** (Color) – The color of the box.

draw_circle(*x*, *y*, *r*, *fill=False*, *color=Color.BLACK*)
Draws a circle on this image.

Parameters

- **x** (*int*) – The x coordinate of the center of the circle.
- **y** (*int*) – The y coordinate of the center of the circle.
- **r** (*int*) – The radius of the circle.
- **fill** (*bool*) – If `True`, the circle will be filled with `color`, otherwise only the circumference will be drawn.
- **color** (Color) – The color of the circle.

Image properties

width
Gets the width of this image in pixels.

height
Gets the height of this image in pixels.

Replacing the entire image

clear()
Clears this image. All pixels on this image will be set to *Color.WHITE*.

load_image(*source*)
Clears this image, then draws the `source` image centered in this image.

Parameters **source** (*Image or str*) – The source *Image*. If the argument is a string, then the `source` image is loaded from file.

Saving the image

save(*filename*)
Saves this image as a `.png` file.

Parameters **filename** (*str*) – The path to the file to be saved.

Raises

- `TypeError` – `filename` is not a string.
- `OSError` – There was a problem saving the file.

CHAPTER 10

`messaging` – Messaging

An EV3 Brick can send information to another EV3 Brick using Bluetooth. This page shows you how to connect multiple bricks and how to write scripts to send messages between them.

10.1 Pairing two EV3 Bricks

Before two EV3 bricks can exchange messages, they must be *paired*. You'll need to do this only the first time. First, activate bluetooth on all EV3 bricks as shown in Figure 10.1.

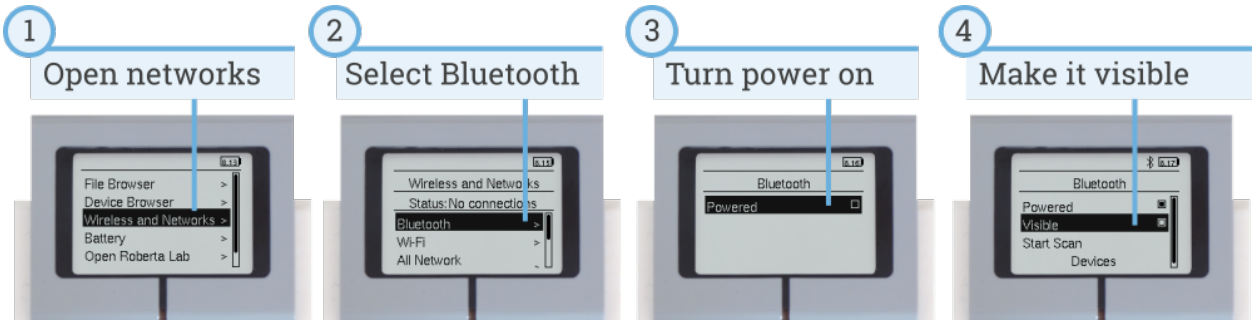


Figure 10.1: Turn on Bluetooth and make Bluetooth visible.

Now you can make one EV3 Brick search for the other and pair with it, as shown in Figure 10.2.

Once they are paired, do *not* click *connect* in the menu that appears. The connection will be made when you run your programs, as described below.

When you scan for Bluetooth devices, you'll see a list of device names. By default, all EV3 Bricks are named *ev3dev*. Click *here* to learn how to change that name. This makes it easy to tell them apart.

Repeat the steps in Figure 10.2 if you want to pair more than two EV3 Bricks.

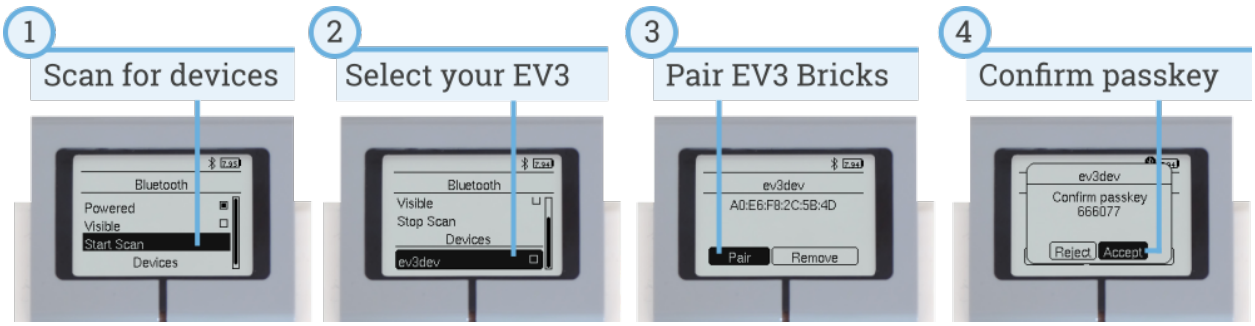


Figure 10.2: Pairing one EV3 Brick to another EV3 Brick.

10.2 Server and Client

A wireless network consists of EV3 Bricks acting as servers or clients. A example with one server and one client is shown in Figure 10.3. Messages can be sent in both ways: the server can send a message to the client, and the client can send a message to the server.

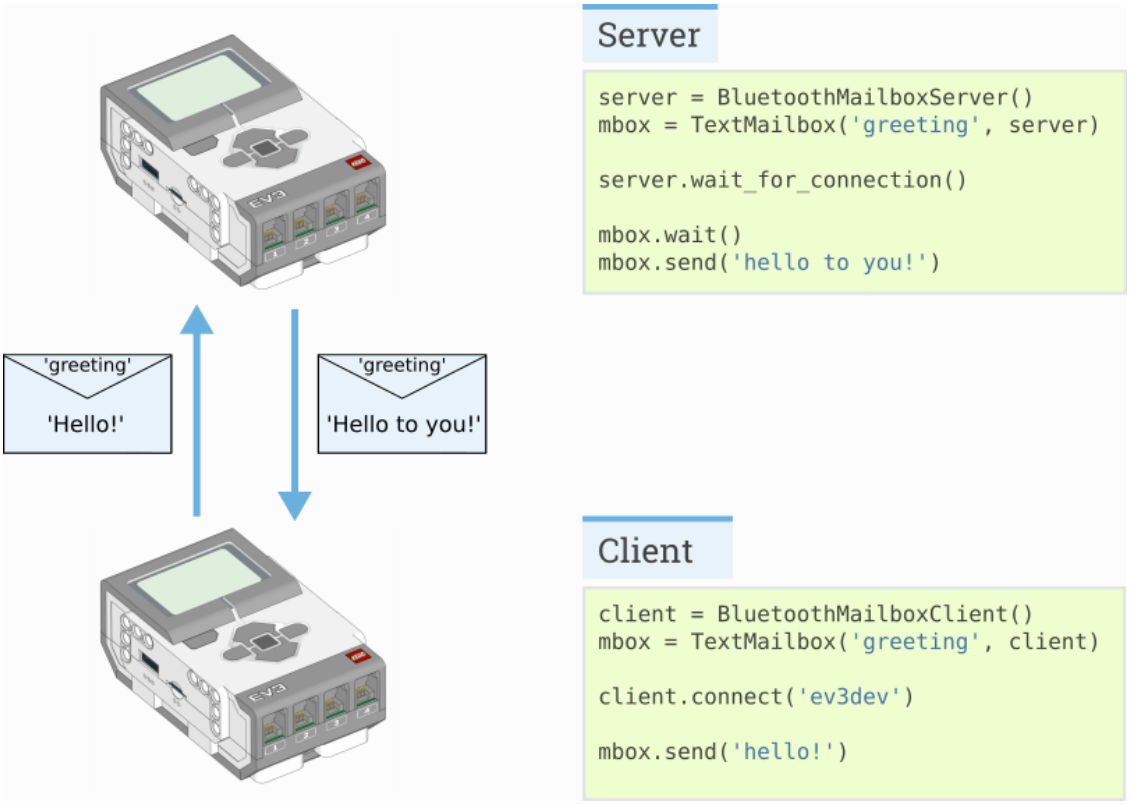


Figure 10.3: An example network with one server and one clients.

Show/hide full server example

Example: EV3 Bluetooth Server.

This is the full version of the excerpt shown in Figure 10.3.

```
#!/usr/bin/env pybricks-micropython

# Before running this program, make sure the client and server EV3 bricks are
# paired using Bluetooth, but do NOT connect them. The program will take care
# of establishing the connection.

# The server must be started before the client!

from pybricks.messaging import BluetoothMailboxServer, TextMailbox

server = BluetoothMailboxServer()
mbox = TextMailbox('greeting', server)

# The server must be started before the client!
print('waiting for connection...')
server.wait_for_connection()
print('connected!')

# In this program, the server waits for the client to send the first message
# and then sends a reply.
mbox.wait()
print(mbox.read())
mbox.send('hello to you!')
```

Show/hide full client example

Example: EV3 Bluetooth Client.

This is the full version of the excerpt shown in Figure 10.3.

```
#!/usr/bin/env pybricks-micropython

# Before running this program, make sure the client and server EV3 bricks are
# paired using Bluetooth, but do NOT connect them. The program will take care
# of establishing the connection.

# The server must be started before the client!

from pybricks.messaging import BluetoothMailboxClient, TextMailbox

# This is the name of the remote EV3 or PC we are connecting to.
SERVER = 'ev3dev'

client = BluetoothMailboxClient()
mbox = TextMailbox('greeting', client)

print('establishing connection...')
client.connect(SERVER)
print('connected!')

# In this program, the client sends the first message and then waits for the
# server to reply.
mbox.send('hello!')
mbox.wait()
print(mbox.read())
```

The only difference between the client and the server is which one initiates the connection at the beginning of the program:

- The **server** must always be started first. It uses the `BluetoothMailboxServer` class. Then it waits for clients using the `wait_for_connection` method.
- The **client** uses the `BluetoothMailboxClient` class. It connects to the server using the `connect` method.
- After that, sending and receiving messages is done in the same way on both EV3 Bricks.

class BluetoothMailboxServer

Object that represents a Bluetooth connection from one or more remote EV3s.

The remote EV3s can either be running MicroPython or the standard EV3 firmware.

A "server" waits for a "client" to connect to it.

wait_for_connection(*count=1*)

Waits for a *BluetoothMailboxClient* on a remote device to connect.

Parameters **count** (*int*) – The number of remote connections to wait for.

Raises `OSError` – There was a problem establishing the connection.

close()

Closes all connections.

class BluetoothMailboxClient

Object that represents a Bluetooth connection to one or more remote EV3s.

The remote EV3s can either be running MicroPython or the standard EV3 firmware.

A "client" initiates a connection to a waiting "server".

connect(*brick*)

Connects to an *BluetoothMailboxServer* on another device.

The remote device must be paired and waiting for a connection. See *BluetoothMailboxServer.wait_for_connection()*.

Parameters **brick** (*str*) – The name or Bluetooth address of the remote EV3 to connect to.

Raises `OSError` – There was a problem establishing the connection.

server_close()

Closes all connections.

10.3 Mailboxes

Mailboxes are used to send data to and from other EV3 Bricks.

A Mailbox has a `name`, similar to the "subject" of an email. If two EV3 Bricks have a Mailbox with the same name, they can send messages between them. Each EV3 Brick can read its own Mailbox, and send messages to the Mailbox on the other EV3 Brick.

Depending on the type of messages you would like to exchange (bytes, booleans, numbers, or text), you can choose one of the Mailboxes below.

class Mailbox(*name*, *connection*, *encode=None*, *decode=None*)

Object that represents a mailbox containing data.

You can read data that is delivered by other EV3 bricks, or send data to other bricks that have the same mailbox.

By default, the mailbox reads and send only bytes. To send other data, you can provide an `encode` function that encodes your Python object into bytes, and a `decode` function to convert bytes back to a Python object.

Parameters

- **name** (*str*) – The name of this mailbox.
- **connection** – A connection object such as *BluetoothMailboxClient*.
- **encode** (*callable*) – Function that encodes a Python object to bytes.
- **decode** (*callable*) – Function that creates a new Python object from bytes.

read()
Gets the current value of the mailbox.

Returns The current value or `None` if the mailbox is empty.

send(*value*, *brick=None*)
Sends a value to this mailbox on connected devices.

Parameters

- **value** – The value that will be delivered to the mailbox.
- **brick** (*str*) – The name or Bluetooth address of the brick or `None` to to broadcast to all connected devices.

Raises `OSError` – There is a problem with the connection.

wait()
Waits for the mailbox to be updated by remote device.

wait_new()
Waits for a new value to be delivered to the mailbox that is not equal to the current value in the mailbox.

Returns The new value.

class LogicMailbox(*name*, *connection*)
Object that represents a mailbox containing boolean data.

This works just like a regular *Mailbox*, but values must be `True` or `False`.

This is compatible with the "logic" mailbox type in EV3-G.

Parameters

- **name** (*str*) – The name of this mailbox.
- **connection** – A connection object such as *BluetoothMailboxClient*.

class NumericMailbox(*name*, *connection*)
Object that represents a mailbox containing numeric data.

This works just like a regular *Mailbox*, but values must be a number, such as `15` or `12.345`

This is compatible with the "numeric" mailbox type in EV3-G.

Parameters

- **name** (*str*) – The name of this mailbox.
- **connection** – A connection object such as *BluetoothMailboxClient*.

class TextMailbox(*name*, *connection*)
Object that represents a mailbox containing text data.

This works just like a regular *Mailbox*, but data must be a string, such as `'hello!'` or `'My name is EV3'`.

This is compatible with the "text" mailbox type in EV3-G.

Parameters

- **name** (*str*) – The name of this mailbox.
- **connection** – A connection object such as *BluetoothMailboxClient*.

10.4 Making bigger networks

The classes in this module are not limited to just two EV3 Bricks. for example, you can add more clients to your network. An example with pseudo-code is shown in Figure 10.4.

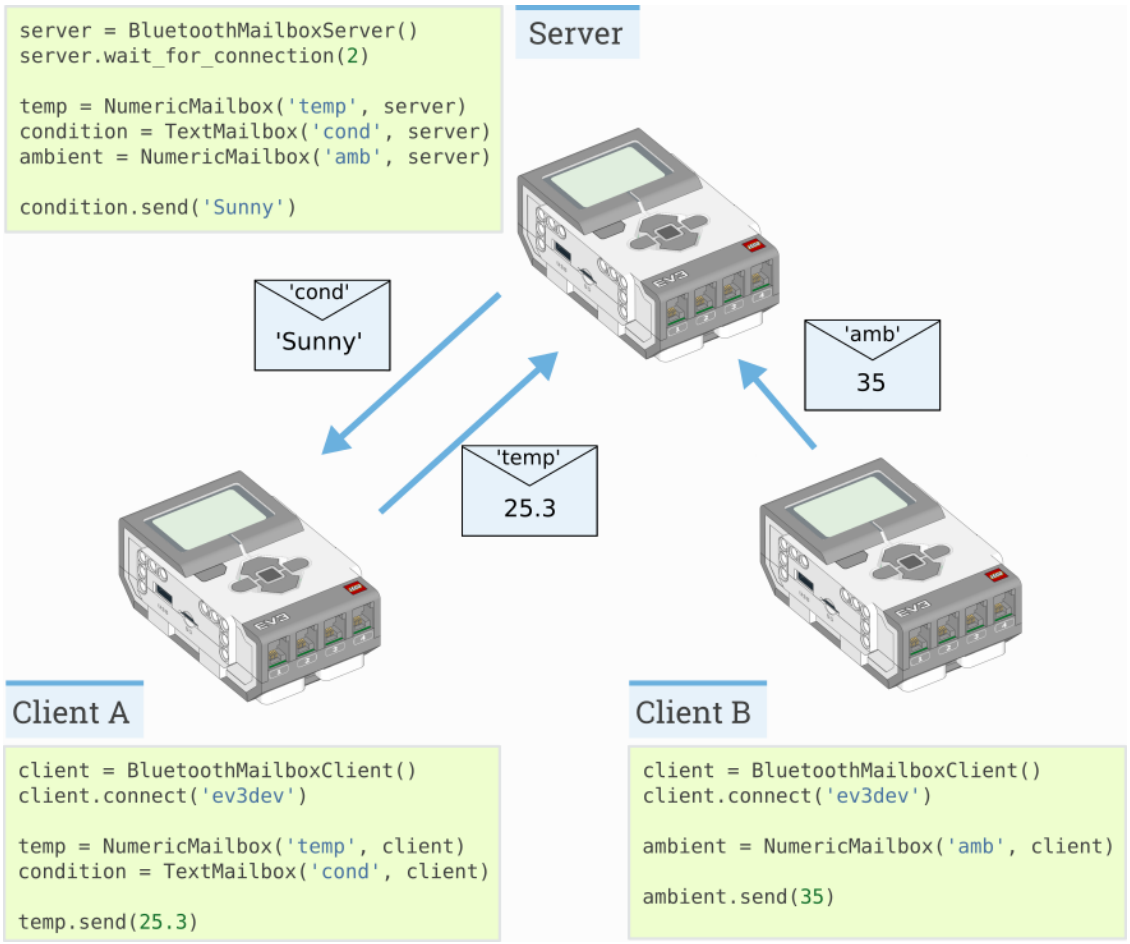


Figure 10.4: An example network with one server and two clients.

CHAPTER 11

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.

11.1 Time

11.1.1 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.

11.2 Angles and angular motion

11.2.1 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.

11.2.2 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

11.2.3 rotational acceleration: deg/s/s

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/s). 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.

11.3 Distance and linear motion

11.3.1 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

11.3.2 dimension: mm

Dimensions are expressed in millimeters (mm), just like distances.

For example, the diameter of a wheel is measured in millimeters.

11.3.3 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.

11.3.4 linear acceleration: mm/s/s

Linear acceleration describes how fast the speed changes. This is expressed as the change of the millimeters per second, during one second (deg/s/s). 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.

11.4 Approximate and relative units

11.4.1 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).

11.4.2 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.

11.4.3 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.

11.5 Force

11.5.1 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

11.6 Electricity

11.6.1 voltage: mV

Voltages are expressed in millivolt (mV).

For example, you can check the voltage of the battery.

11.6.2 current: mA

Electrical currents are expressed in milliampere (mA).

For example, you can check the current supplied by the battery.

11.6.3 energy: J

Stored energy or energy consumption can be expressed in Joules (J).

11.6.4 power: mW

Power is the rate at which energy is stored or consumed. It is expressed in milliwatt (mW).

11.7 Ambient environment

11.7.1 frequency: Hz

Sound frequencies are expressed in Hertz (Hz).

For example, you can choose the frequency of a beep to change the pitch.

11.7.2 temperature: ℃

Temperature is measured in degrees Celcius (°C). To convert to degrees Fahrenheit (°F) or Kelvin (K), you can use the following conversion formulas:

$$F = C \cdot \frac{9}{5} + 32.$$

$$K = C + 273.15.$$

CHAPTER 12

More about Motors

12.1 The Control Class

The `Motor` class uses PID control to accurately track your commanded target angles. Similarly, the `DriveBase` class uses two of such controllers: one to control the heading and one to control the traveled distance.

You can change the control settings through the following attributes, which are instances of the `Control` class given below.:

- `Motor.control`
- `DriveBase.heading_control`
- `DriveBase.distance_control`

You can only change the settings while the controller is stopped. For example, you can set the settings at the beginning of your program. Alternatively, first call `stop()` to make your `Motor` or `DriveBase` stop, and then change the settings.

class Control

Class to interact with PID controller and settings.

scale

Scaling factor between the controlled integer variable and the physical output. For example, for a single motor this is the number of encoder pulses per degree of rotation.

Status

done()

Checks if an ongoing command or maneuver is done.

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

Return type bool

stalled()
Checks if the controller is currently stalled.

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

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

Return type bool

Settings

limits(*speed*, *acceleration*, *actuation*)
Configures the maximum speed, acceleration, and actuation.

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

Parameters

- **speed** (*rotational speed: deg/s* or *speed: mm/s*) – Maximum speed. All speed commands will be capped to this value.
- **acceleration** (*rotational acceleration: deg/s/s* or *linear acceleration: mm/s/s*) – Maximum acceleration.
- **actuation** (*percentage: %*) – Maximum actuation as percentage of absolute maximum.

pid(*kp*, *ki*, *kd*, *integral_range*, *integral_rate*, *feed_forward*)
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 (or integral speed) control constant.
- **ki** (*int*) – Integral position control constant.
- **kd** (*int*) – Derivative position (or proportional speed) control constant.
- **integral_range** (*angle: deg* or *distance: mm*) – Region around the target angle or distance, in which integral control errors are accumulated.
- **integral_rate** (*rotational speed: deg/s* or *speed: mm/s*) – Maximum rate at which the error integral is allowed to grow.
- **feed_forward** (*percentage: %*) – This adds a feed forward signal to the PID feedback signal, in the direction of the speed reference. This value is expressed as a percentage of the absolute maximum duty cycle.

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

stall_tolerances(*speed*, *time*)
Gets or sets stalling tolerances.

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

Parameters

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

Python Module Index

p

Index

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W