ev3_dc Documentation

Release 0.9.9

Christoph Gaukel

Feb 18, 2022

Contents:

Use python3 to program your LEGO Mindstorms EV3. The program runs on the local host and sends direct commands to the EV3 device. It communicates via Bluetooth, WiFi or Usb.

There is no need to boot the EV3 device from an SD Card or manipulate its software. You can use it as it is, the EV3 is designed to execute commands which come from outside.

If you like to code from scratch, then read this blog.

CHAPTER 1

Installation

Use pip to install module **ev3_dc**:

```
python3 -m pip install --user ev3_dc
```

or:

```
pip3 install --user ev3_dc
```

pip also allows to upgrade module **ev3_dc**:

```
python3 -m pip install --upgrade --user ev3_dc
```

ev3_dc supports text to speech. You need to install ffmpeg to get it working. On a Windows machine, you have to download an archive (e.g. this) and unpack it. Finally you have to add the directory with the binary files to your environment variable *path*. MacOS and Unix provide installation packages (MacOS: *brew install ffmpeg*, Ubuntu: *sudo apt-get install ffmpeg*).

Before you can use Bluetooth, you need to couple the computer (that executes the python programs) and the EV3 brick.

If you own a compatible WiFi dongle, and you want to use it, you have to connect the EV3 brick with the WiFi network.

Protocol USB is the fastest option, both to establish the connection and to communicate. But using USB may need some additional installations.

Read more in section *Connect with the EV3 device*.

CHAPTER 2

Examples

2.1 EV3

Class *EV3* is the base class of ev3_dc. The constructor of EV3 establishes a connection between your computer and your EV3 brick. Its properties allow to get some information about the EV3's state. A few of them allow to change its behaviour. But the power of this class comes from its methods `send_direct_cmd()` and `send_system_cmd()` which send bytestrings to your EV3 brick. If these bytestrings are well formed, your EV3 brick will understand and execute their operations. If a bytestring requests it, the EV3 brick answers with another bytestring, which contains the return data. For using these methods, you need to know the details of direct and system commands.

To establish a connection is a requirement for using class *EV3*. This is, what the next section deals with.

2.1.1 Connect with the EV3 device

We test all three connection protocols, which the EV3 device provides. If you don't own a WiFi dongle, you still can use *USB* and *Bluetooth*.

USB

In the background python modules often use programs, written in C. This means: the module is a thin python layer, which calls a compiled library, often named backend. In case of USB devices, there exists a number of different backends, e.g. libusb0.1, libusb1.0, OpenUSB.

For the installation process of the software, this says: You have to install some python code, which automatically has been done, when module ev3_dc was installed. But when ev3_dc tries to connect via USB, it needs to load a backend and it may happen, that it does not find any.

Let's look at the preparation steps.

Linux

On my Ubuntu system, I first installed backend libusb1.0 with this terminal command:

```
sudo apt-get install libusb-1.0-0
```

Second I made shure to have the permission to connect the EV3 device. I added this udev rule (as file */etc/udev/rules.d/90-legoev3.rules*):

```
ATTRS{idVendor}=="0694",ATTRS{idProduct}=="0005",MODE="0666",GROUP="christoph"
```

This gives all members of group *christoph* (me allone) read and write permissions to product *0005* (EV3 devices) of vendor *0694* (LEGO group).

Reboot the system or alternatively run this terminal command:

```
sudo udevadm control --reload-rules && udevadm trigger
```

Windows 10

Download libusb1.0 from the libusb project (I additionally had to install program 7 zip).

Copy file *libusb-1.0.dll* from *libusb-1.0.xy.7z\MinGW64\dll\* into directory *C:\Windows\System32*.

Follow this instruction and replace *Xin-Mo Programmer* by *EV3* (when I did it, I clicked the *Install Now* button in the Inf-Wizard and it was successfully installing).

MacOS

In case of MacOS, ev3_dc imports and uses module hidapi (whereas on Linux or Windows systems it imports module pyusb). As far as I know, the connection works out of the box (I don't own a Mac).

Test USB

Take an USB cable and connect your EV3 device (the 2.0 Mini-B port, titled PC) with your computer. Then run this program.

```
import ev3_dc as ev3

with ev3.EV3(protocol=ev3.USB) as my_robot:
    print(my_robot)
```

If everything is o.k., you will see an output like:

```
USB connected EV3 00:16:53:42:2B:99 (Hugo)
```

It needs a communication between the program and the EV3 device to know my EV3's name (*Hugo*) and its MAC-address (*00:16:53:42:2B:99*). The MAC-address also is known as serial number or pysical address and you can read it from your EV3's display under Brick Info / ID. Therefore the result documents, the connection was successfully established.

Bluetooth

On Windows systems, Bluetooth works from Python 3.9 upwards. This says: your operating system can't be Windows 7 or earlier. Maybe you need to install a newer python3 version. This can be done from Python Releases for Windows.

On Linux systems, Bluetooth AutoEnable needs to be deactivated. I (my computer has an Ubuntu 20.10 operating system) had to comment out the last line in file */etc/bluetooth/main.conf* (which needs superuser access rights):

```
# AutoEnable defines option to enable all controllers when they are found.
# This includes adapters present on start as well as adapters that are plugged
# in later on. Defaults to 'false'.
# AutoEnable=true
```

Couple (only steps 1 - 12) your computer and your EV3 device via Bluetooth and call the EV3 constructor with **protocol=ev3.BLUETOOTH**. This says: replace MAC-address `00:16:53:42:2B:99` with the one of your EV3, then run this program:

```
import ev3_dc as ev3

with ev3.EV3(protocol=ev3.BLUETOOTH, host='00:16:53:42:2B:99') as my_robot:
    print(my_robot)
```

My output was:

```
Bluetooth connected EV3 00:16:53:42:2B:99 (Hugo)
```

Hopefully, you will see something similar. If so, your Bluetooth connection works.

WiFi

If you own a WiFi dongle, you can connect (only steps 1 - 12) your EV3 device via WiFi with your local network. If your computer also is connected (either via WiFi or via Ethernet), they can communicate. If these conditions are fulfilled, you can call the EV3 constructor with **protocol=ev3.WIFI**. Replace MAC-address `00:16:53:42:2B:99` with the one of your EV3, then start this program:

```
import ev3_dc as ev3

with ev3.EV3(protocol=ev3.WIFI, host='00:16:53:42:2B:99') as my_robot:
    print(my_robot)
```

As you may have expected, my program's output was:

```
WiFi connected EV3 00:16:53:42:2B:99 (Hugo)
```

I hope you can connect at least one protocol, if it's really only one and this is *USB*, you have no wireless connection, which is a restriction. If you have more than one option, you are lucky. *USB* is fast connected and fast in data transfer. When you start your EV3 device, *USB* is ready without any coupling. I prefer it for developing.

2.1.2 EV3's properties

The properties of class *EV3* provide easy access to the state of the EV3 device. They e.g. describe the battery status, the free memory space or the connected sensors and motors. I will present some short programs to show their usage.

A few of the properties also allow to change the state of the EV3 device, you can e.g. easily change the sound volume or the EV3's name.

name

Property `name` allows to read and change the name of the EV3 device. This is the one, you see in the first line of your EV3's display, which you can change under menu item *Brick Name*. Replace MAC-address `00:16:53:42:2B:99` with the one of your EV3 device and select the protocol you prefer, then start this program:

```
import ev3_dc as ev3

with ev3.EV3(protocol=ev3.BLUETOOTH, host='00:16:53:42:2B:99') as my_ev3:
    print('This is', my_ev3.name)
```

My program's output was:

```
This is Hugo
```

Now let's change the name of the EV3 device with this program:

```
import ev3_dc as ev3

with ev3.EV3(protocol=ev3.BLUETOOTH, host='00:16:53:42:2B:99') as my_ev3:
    my_ev3.name = 'Evelyn'
```

Control your EV3's display, if the name really did change.

sleep

Property `sleep` allows to read and change the timespan (in minutes), the EV3 waits in idle state before it automatically shuts down. You can change this timespan under menu item **Sleep**. Your display allows the following values: *2 min.*, *5 min.*, *10 min.*, *30 min.*, *60 min.* and *never*.

Replace MAC-address `00:16:53:42:2B:99` with the one of your EV3 device and select the protocol you prefer, then start this program:

```
import ev3_dc as ev3

with ev3.EV3(protocol=ev3.BLUETOOTH, host='00:16:53:42:2B:99') as my_ev3:
    print(f'Currently sleep is set to {my_ev3.sleep} min.')
```

My program's output was:

```
Currently sleep is set to 30 min.
```

We change the sleeping time of the EV3 device with this program:

```
import ev3_dc as ev3

with ev3.EV3(protocol=ev3.BLUETOOTH, host='00:16:53:42:2B:99') as my_ev3:
    my_ev3.sleep = 12
```

Your EV3 device accepts all values from 0 to 120, but your EV3's display will not present them correctly and is blocked for any further changes of the sleeping time. Therefore change it once again to one of the above mentioned values (*never* is value 0).

volume

Property `volume` allows to read and change the sound volume. You can also change the sound volume under menu item **Volume**. Your display allows the following values: *0 %*, *10 %*, *20 %*, . . . , *100 %*.

Replace MAC-address `00:16:53:42:2B:99` with the one of your EV3 device and select the protocol you prefer, then start this program:

```
import ev3_dc as ev3

with ev3.EV3(protocol=ev3.BLUETOOTH, host='00:16:53:42:2B:99') as my_ev3:
    print(f'Currently the sound volume is set to {my_ev3.volume} %')
```

My program's output was:

```
Currently the sound volume is set to 10 %.
```

We change the sound volume of the EV3 device with this program:

```
import ev3_dc as ev3

with ev3.EV3(protocol=ev3.BLUETOOTH, host='00:16:53:42:2B:99') as my_ev3:
    my_ev3.volume = 18
```

Your EV3 device accepts all values from 0 to 100, but your EV3's display will not present all of them correctly and will be partly blocked. Therefore change it once again to one of the above mentioned values.

battery

Property `battery` allows to get informations about the EV3's battery state. You get its voltage, its current and its state of charge.

Replace MAC-address `00:16:53:42:2B:99` with the one of your EV3 device, select the protocol you prefer, then start this program:

```
import ev3_dc as ev3

with ev3.EV3(protocol=ev3.BLUETOOTH, host='00:16:53:42:2B:99') as my_ev3:
    print(my_ev3.battery)
```

My program's output was:

```
Battery(voltage=7.123220920562744, current=0.19781701266765594, percentage=5)
```

The voltage is in Volt, the current in Ampère. You can also access the single values:

```
import ev3_dc as ev3

with ev3.EV3(protocol=ev3.BLUETOOTH, host='00:16:53:42:2B:99') as my_ev3:
    bat = my_ev3.battery
    print(f'the power consumption is {bat.voltage * bat.current:4.2f} Watt')
```

Don't code `{my_ev3.battery.voltage * my_ev3.battery.current:4.2f}`, this would result in two request-reply-cycles, because the battery state is requested again whenever you reference property *battery*.

My program's output was:

```
the power consumption is 1.44 Watt
```

Maybe you like to recalculate the power consumption, when some motors are running. The value above is without motor movement and is typical for ARM architecture computers.

sensors

Property *sensors* informs about the sensor types (motors also are sensors), which are connected to the EV3 brick.

Replace MAC-address `00:16:53:42:2B:99` with the one of your EV3 device, select the protocol you prefer, then start this program:

```
import ev3_dc as ev3

with ev3.EV3(protocol=ev3.USB, host='00:16:53:42:2B:99') as my_ev3:
    print(my_ev3.sensors)
```

My program's output was:

```
Sensors(Port_1=16, Port_2=33, Port_3=5, Port_4=1, Port_A=7, Port_B=8, Port_C=None,␣
↪Port_D=7)
```

Read chapter 5 *Device type list* of document EV3 Firmware Developer Kit, which lists the EV3 sensors. Each sensor is identified by an integer number:

- NXT_TOUCH = 1
- NXT_LIGHT = 2
- NXT_SOUND = 3
- NXT_COLOR = 4
- NXT_ULTRASONIC = 5
- NXT_TEMPERATURE = 6
- EV3_LARGE_MOTOR = 7
- EV3_MEDIUM_MOTOR = 8
- EV3_TOUCH = 16
- EV3_COLOR = 29
- EV3_ULTRASONIC = 30
- EV3_GYRO = 32
- EV3_IR = 33

Your EV3 brick names its sensor ports by numbers 1 to 4 and its motor ports by characters A to D.

sensors_as_dict

Property *sensors_as_dict* provides the same information as property *sensors* but presents it in a form, which supports automatic handling.

Replace MAC-address `00:16:53:42:2B:99` with the one of your EV3 device, select the protocol you prefer, then start this program:

```
import ev3_dc as ev3

with ev3.EV3(protocol=ev3.USB, host='00:16:53:42:2B:99') as my_ev3:
    sensors = my_ev3.sensors_as_dict

    assert sensors[ev3.PORT_1] == ev3.EV3_TOUCH, \
      'no EV3 touch connected at port 1'
    assert sensors[ev3.PORT_2] == ev3.EV3_IR, \
      'no EV3 infrared connected at port 2'
    assert sensors[ev3.PORT_3] == ev3.NXT_ULTRASONIC, \
      'no NXT ultrasonic connected at port 3'
    assert sensors[ev3.PORT_4] == ev3.NXT_TOUCH, \
      'no NXT touch connected at port 4'
    assert sensors[ev3.PORT_A_SENSOR] == ev3.EV3_LARGE_MOTOR, \
      'no large motor connected at port A'
    assert sensors[ev3.PORT_B_SENSOR] == ev3.EV3_MEDIUM_MOTOR, \
      'no medium motor connected at port B'
    assert sensors[ev3.PORT_D_SENSOR] == ev3.EV3_LARGE_MOTOR, \
      'no large motor connected at port D'

    print('everything is as expected')
```

Some remarks:

- Adapt this program to your connected sensor combination.
- Using constants for the ports and sensors helps for readability.
- Motors can be addressed as sensors or as motors, this is why we use two different constants for the sensor context and the movement context. If you use a motor as sensor, address it by e.g. constant PORT_A_SENSOR.

system

Property *system* tells some informations about the EV3's operating system version, firmware version and hardware version. Operating system and firmware additionally know their build numbers.

Replace MAC-address `00:16:53:42:2B:99` with the one of your EV3 device, select the protocol you prefer, then start this program:

```
import ev3_dc as ev3

with ev3.EV3(protocol=ev3.BLUETOOTH, host='00:16:53:42:2B:99') as my_ev3:
    print(my_ev3.system)
```

My program's output was:

```
System(os_version='Linux 2.6.33-rc', os_build='1212131117', fw_version='V1.09H', fw_
↪build='1512030906', hw_version='V0.60')
```

The operating system is Linux, which runs a lot of devices like smart TVs, routers, etc. On my EV3 device, the Linux version is 2, the major revision is 6, the minor revision is 33 and it's a *release candidate*. This says, it stems from a time before 24 February 2010. If you need it more precisely, you also get the build number of the operating system version.

The firmware is the software, which LEGO® developped, it allows to e.g. control the display, communicate with sensors and motors or run programs. My EV3 has been updated to version V1.09H and its hardware version is V0.60.

network

Property `network` allows to get informations about the WiFi connection of the EV3 device. Therefore it only works if the connection protocol is *WIFI*.

Replace MAC-address `00:16:53:42:2B:99` with the one of your EV3 device, connect your EV3 device via WiFi with your local network, then start this program:

```
import ev3_dc as ev3

with ev3.EV3(protocol=ev3.WIFI, host='00:16:53:42:2B:99') as my_ev3:
    print(my_ev3.network)
```

My program's output was:

```
Network(name='NetOfTheSix', ip_adr='192.168.178.35', mac_adr='44:49:94:4F:FC:C2')
```

This says:

- The name of the WiFi network is *NetOfTheSix*, which must operate on 2.4 GHz (the EV3 device does not support 5 GHz WiFi).
- In this network, my EV3 device got the IPv4 address *192.168.178.35*.
- My WiFi dongle (this is the device, which connects to the network) has the mac-address *44:49:94:4F:FC:C2*, which is different from the mac-address of the EV3 device.

If you prefer to access the single values directly, then do:

```
import ev3_dc as ev3

with ev3.EV3(protocol=ev3.WIFI, host='00:16:53:42:2B:99') as my_ev3:
    print(f'name of the network:         {my_ev3.network.name}')
    print(f'ip_adr of the EV3 device:    {my_ev3.network.ip_adr}')
    print(f'mac_adr of the WiFi dongle: {my_ev3.network.mac_adr}')
```

This program's output was:

```
name of the network:         NetOfTheSix
ip_adr of the EV3 device:    192.168.178.35
mac_adr of the WiFi dongle: 44:49:94:4F:FC:C2
```

memory

Property `memory` informs about EV3's memory space.

Replace MAC-address `00:16:53:42:2B:99` with the one of your EV3 device, select the protocol you prefer, then start this program:

```
import ev3_dc as ev3

with ev3.EV3(protocol=ev3.USB, host='00:16:53:42:2B:99') as my_ev3:
    print(f'{my_ev3.memory.free} of {my_ev3.memory.total} kB memory are free')
```

My program's output was:

```
4572 of 6000 kB memory are free
```

This says, 6 MB is the total user memory space of my EV3 device, which seems to be small, but is large enough for the things I really do on this device.

protocol

Property *protocol* tells the protocol type of the EV3's connection. This sounds weird because we explicitly set it, when we create an EV3 instance and we can't change it. But think of the situation, when you call a function or method, which you did not code and it returns an EV3 instance. Maybe you want to know, how this instance is connected.

Replace MAC-address `00:16:53:42:2B:99` with the one of your EV3 device, select the protocol you prefer, then start this program:

```
import ev3_dc as ev3

with ev3.EV3(protocol=ev3.USB, host='00:16:53:42:2B:99') as my_ev3:
    print(f'this EV3 device is connected via {my_ev3.protocol}')
```

This program's output:

```
this EV3 device is connected via USB
```

host

Property *host* tells the MAC-address of the EV3 device. As above this is thought for EV3 instances, you got from somewhere.

Replace MAC-address `00:16:53:42:2B:99` with the one of your EV3 device, select the protocol you prefer, then start this program:

```
import ev3_dc as ev3

with ev3.EV3(protocol=ev3.USB, host='00:16:53:42:2B:99') as my_ev3:
    print(f'{my_ev3.host} is the MAC-address of this EV3 device')
```

This program's output:

```
00:16:53:42:2B:99 is the MAC-address of this EV3 device
```

verbosity

Setting property *verbosity* to a value greater than zero allows to see the communication data between the program and the connected EV3 device.

Replace MAC-address `00:16:53:42:2B:99` with the one of your EV3 device, select the protocol you prefer, then start this program:

```
import ev3_dc as ev3

with ev3.EV3(protocol=ev3.USB, host='00:16:53:42:2B:99') as my_ev3:
    my_ev3.verbosity = 1
    bat = my_ev3.battery
```

This program's output:

```
19:45:30.891798 Sent 0x|0E:00|2A:00|00|09:00|81:01:60:81:02:64:81:12:68|
19:45:30.898732 Recv 0x|0C:00|2A:00|02|7C:03:F1:40:40:07:3B:3E:64|
```

Some remarks:

- Referencing the battery property by `bat = my_ev3.battery` initiates a request-response-cycle which asks for the current state of the battery and gets some data back.
- Easy to understand are the timestamps. Between the request and the response lies a timespan of 7 ms.
- The request and response themselves are quite cryptic! If you want to understand them, read section *Direct commands*

sync_mode

Property `sync_mode` has a very special meaning for direct commands. It influences the way, how requests are handled. If its value is *SYNC*, then all requests will be answered and the calling program will always wait until the response did arrive, even if the direct command does not return any data. If its value is *ASYNC*, then method `send_direct_cmd()` never will wait until a response comes back. Instead it will return the message counter and it is the responsibility of the programmer to call method `wait_for_reply()`. This allows to continue with processing until the response is needed and then wait and get it. The third value *STD* will only wait for replies, if the direct command returns data.

Replace MAC-address `00:16:53:42:2B:99` with the one of your EV3 device, select the protocol you prefer, then start this program:

```python
import ev3_dc as ev3

with ev3.EV3(protocol=ev3.USB, host='00:16:53:42:2B:99') as my_ev3:
    print(f"protocol USB's default sync_mode is {my_ev3.sync_mode}")
    my_ev3.name = 'Evelyn'
    my_ev3.verbosity = 1
    my_ev3.name = 'Hugo'
```

This program's output:

```
protocol USB's default sync_mode is SYNC
19:28:11.184508 Sent 0x|0D:00|2B:00|00|00:00|D4:08:84:48:75:67:6F:00|
19:28:11.193370 Recv 0x|03:00|2B:00|02|
```

Protocol *USB* is that fast, that sometimes the EV3 device is not able to handle all direct commands correctly. *sync_mode = SYNC* guaranties, that each direct command has finished, before the next one is sent. Therefore protol *USB's* default snc_mode is *SYNC*.

The direct command, which changes EV3's name does not reply anything, but our program had to wait about 9 ms until the response did arrive.

sync_mode *SYNC's* 2nd advantage is, that errors can't occur silently. Every direct command replies and every reply contains the return code of the direct command.

Now let's change the program and explicitly set *sync_mode = STD*:

```python
import ev3_dc as ev3

with ev3.EV3(protocol=ev3.USB, host='00:16:53:42:2B:99') as my_ev3:
    print(f"protocol USB's default sync_mode is {my_ev3.sync_mode}")
    my_ev3.name = 'Evelyn'
```

(continues on next page)

(continued from previous page)

```
    my_ev3.sync_mode = ev3.STD
    my_ev3.verbosity = 1
    my_ev3.name = 'Hugo'
```

This program's output:

```
protocol USB's default sync_mode is SYNC
19:34:35.935427 Sent 0x|0D:00|2B:00|80|00:00|D4:08:84:48:75:67:6F:00|
```

With *sync_mode* = *STD*, the EV3 device does not reply this direct command.

2.1.3 Direct commands

Document EV3 Firmware Developer Kit is the reference book of LEGO EV3 direct commands and will help you to understand the details.

The art of doing nothing

We send the idle operation of the EV3 device to test the communication speed.

Replace MAC-address `00:16:53:42:2B:99` with the one of your EV3 device, then run this program:

```
import ev3_dc as ev3

with ev3.EV3(protocol=ev3.USB, host='00:16:53:42:2B:99') as my_ev3:
    my_ev3.verbosity = 1
    my_ev3.sync_mode = ev3.SYNC
    ops = ev3.opNop
    my_ev3.send_direct_cmd(ops)
```

If everything is o.k., you will see an output like:

```
20:09:32.162156 Sent 0x|06:00|2A:00|00|00:00|01|
20:09:32.168082 Recv 0x|03:00|2A:00|02|
```

Some remarks:

- Both lines start with a timestamp. A bit shorter than 6 ms was the timespan of this request-reply-cycle.
- The first line shows the sent data in a binary format. We separate bytes by colons ":" or vertical bars "|". Vertical bars separate these groups of bytes:
  - **Length of the message** (bytes 0, 1): The first two bytes are not part of the direct command itself. They are part of the communication protocol. The length is coded as a 2-byte unsigned integer in little endian format, 0x|06:00| therefore stands for the value 6.
  - **Message counter** (bytes 2, 3): This is the footprint of the direct command. The message counter will be included in the corresponding reply and allows to match the direct command and its reply. This too is a 2-byte unsigned integer in little endian format. The EV3 class starts counting with 0x|2A:00|, which is the value 42.
  - **Message type** (byte 4): For direct commands it may have the following two values:
    * DIRECT_COMMAND_REPLY = 0x|00|
    * DIRECT_COMMAND_NO_REPLY = 0x|80|

In our case we did set sync_mode=SYNC, which means: we want the EV3 to reply all messages.

  - **Header** (bytes 5, 6): These two bytes, the last in front of the first operation are the header. It includes a combination of two numbers, which define the memory sizes of the direct command (yes, its plural, there are two memories, a local and a global one). Our command does not need any memory, therefore the header was set to 0x|00:00|.
  - **Operations** (starting at byte 7): Here one single byte, that stands for: opNOP = 0x|01|, do nothing, the idle operation of the EV3.

- The second line shows the received data:
  - **Length of the message** (bytes 0, 1), here 3 bytes.
  - **Message counter** (bytes 2, 3): This fits the message counter of the corresponding request.
  - **Return status** (byte 4): For direct commands it may have the following two values:
    - DIRECT_REPLY = 0x|02|: the direct command was successfully operated.
    - DIRECT_REPLY_ERROR = 0x|04|: the direct command ended with an error.

If we had set the global memory to a value larger than 0 (e.g. calling `send_direct_cmd()` with a keyword argument `global_mem=1`, we would have seen some additional data after the return status.

Replace the protocol by **ev3.WIFI** and **ev3.BLUETOOTH** and start the program again. The time gaps between request and reply will show the communication speeds. USB is the fastest, then comes WIFI, BLUETOOTH is the slowest. Compared with human communication, all three of them are quite fast.

Tell your EV3 what to do

Direct commands allow to send instructions with arguments.

Changing LED colors

There are some light effects on the EV3 brick. You can change the colors of the LEDs and this is done by operation *opUI_Write* with CMD *LED*.

opUI_Write = 0x|82| with CMD LED = 0x|1B| needs one argument:

- PATTERN: GREEN = 0x|01|, RED = 0x|02|, etc.

Take an USB cable and connect your EV3 brick with your computer. Replace the MAC-address by the one of your EV3 brick, then start the program.

```
import ev3_dc as ev3
from time import sleep

my_ev3 = ev3.EV3(protocol=ev3.USB, host='00:16:53:42:2B:99')
my_ev3.verbosity = 1

ops = b''.join((
    ev3.opUI_Write,  # operation
    ev3.LED,  # CMD
    ev3.LED_RED_FLASH  # PATTERN
))
my_ev3.send_direct_cmd(ops)

sleep(5)
```

(continues on next page)

(continued from previous page)

```
ops = b''.join((
    ev3.opUI_Write,
    ev3.LED,
    ev3.LED_GREEN
))
my_ev3.send_direct_cmd(ops)
```

This program sends two direct commands with a timespan of 5 sec. between them. The first one changes the LED color to a red flashing, the second sets the well known green color.

The output:

```
10:43:38.601015 Sent 0x|08:00|2A:00|00|00:00|82:1B:05|
10:43:38.616028 Recv 0x|03:00|2A:00|02|
10:43:43.620023 Sent 0x|08:00|2B:00|00|00:00|82:1B:01|
10:43:43.630105 Recv 0x|03:00|2B:00|02|
```

Some remarks:

- The default *sync_mode* of the USB protocol is *SYNC*. This is why both direct commands were replied.
- EV3 increments the message counter. The first command got 0x|2A:00|, which is the value 42, the second command got 0x|2B:00| (value 43).
- 0x|82| is the bytecode of operation *opUI_Write*.
- 0x|1B| is the bytecode of CMD *LED*.
- 0x|05| is the bytecode of *LED_RED_FLASH*.
- 0x|01| is the bytecode of *LED_GREEN*.

If we replace *protocol=ev3.USB* by *protocol=ev3.BLUETOOTH*, we get this output:

```
10:44:47.266688 Sent 0x|08:00|2A:00|80|00:00|82:1B:05|
10:44:52.272881 Sent 0x|08:00|2B:00|80|00:00|82:1B:01|
```

The *message type* changed from 0x|00| (DIRECT_COMMAND_REPLY) to 0x|80| (DIRECT_COMMAND_NO_REPLY) and the EV3 brick indeed did not reply. This happens because *protocol* BLUETOOTH defaults to *sync_mode* STD.

Setting EV3's brickname

You can change the name of your EV3 brick by sending a direct command.

opCom_Set = 0x|D4| with CMD SET_BRICKNAME = 0x|08| needs one argument:

- NAME: (DATA8) – First character in character string

Some more explanations of argument NAME will follow. The text above is, what the LEGO documentation says.

The program:

```
import ev3_dc as ev3

my_ev3 = ev3.EV3(protocol=ev3.WIFI, host='00:16:53:42:2B:99')
my_ev3.verbosity = 1

ops = b''.join((
```

(continues on next page)

(continued from previous page)

```
    ev3.opCom_Set,  # operation
    ev3.SET_BRICKNAME,  # CMD
    ev3.LCS("myEV3")  # NAME
))
my_ev3.send_direct_cmd(ops)
```

Direct commands are built as byte strings. Multiple operations can be concatenated. Here a single operation is sent. The combination of operation *opCom_Set* and CMD *SET_BRICKNAME* sets the brickname. This command needs a single string argument and does not produce any output. We let *sync_mode* be *STD*, which omits replies if the global memory (space for return data) is unused.

The output of the program:

```
10:49:13.012039 Sent 0x|0E:00|2A:00|80|00:00|D4:08:84:6D:79:45:56:33:00|
```

Some remarks:

- 0x|D4| is the bytecode of operation *opCom_Set*.
- 0X|08| is the bytecode of CMD *SET_BRICKNAME*.
- 0x|84| is the bytecode of the leading identification byte of `LCS()` character strings (in binary notation, it is: 0b 1000 0100). If any argument is a string, it will be sent as an LCS, which says a leading and a trailing byte must be added.
- 0x|6D:79:45:56:33| is the ascii bytecode of the string *myEV3*.
- 0x|00| terminates LCS character strings.

Maybe you're not familiar with this vocabulary. Document EV3 Firmware Developer Kit will help you. Read the details about the leading identification byte in section *3.4 Parameter encoding*.

Starting programs

Direct commands allow to start programs, which normally is done by pressing buttons of the EV3 device. A program is a file, that exists in the filesystem of the EV3. We will start /home/root/lms2012/apps/Motor Control/Motor Control.rbf. This needs two operations:

```
import ev3_dc as ev3

my_ev3 = ev3.EV3(protocol=ev3.BLUETOOTH, host='00:16:53:42:2B:99')

ops = b''.join((
    ev3.opFile,
    ev3.LOAD_IMAGE,
    ev3.LCX(1),  # SLOT
    ev3.LCS('../apps/Motor Control/Motor Control.rbf'),  # NAME
    ev3.LVX(0),  # SIZE
    ev3.LVX(4),  # IP*
    ev3.opProgram_Start,
    ev3.LCX(1),  # SLOT
    ev3.LVX(0),  # SIZE
    ev3.LVX(4),  # IP*
    ev3.LCX(0)  # DEBUG
))
my_ev3.send_direct_cmd(ops, local_mem=8)
```

The first operation is the loader. It places a program into memory and prepares it for execution. The second operation starts the program. The return values of the first operation are SIZE and IP*. We use `LVX()` to write them to the local memory at addresses 0 and 4. The second operation reads its arguments SIZE and IP* from the local memory. It's arguments SLOT and DEBUG are given as constant values.

Paths can be absolute or relative. Relative paths, like the above one, are relative to */home/root/lms2012/sys/*. We don't set verbosity and the command does not use any global memory, therefore it sends the direct command and ends silently. But the display of the EV3 device will show, that the program has been started.

Playing Sound Files

Take an USB cable and connect your EV3 brick with your computer. Replace the MAC-address by the one of your EV3 brick, then start the program.

```
import ev3_dc as ev3

my_ev3 = ev3.EV3(protocol=ev3.USB, host='00:16:53:42:2B:99')
my_ev3.verbosity = 1

ops = b''.join((
    ev3.opSound,  # operation
    ev3.PLAY,  # CMD
    ev3.LCX(100),  # VOLUME
    ev3.LCS('./ui/DownloadSucces')  # NAME
))
my_ev3.send_direct_cmd(ops)
```

The output:

```
10:20:05.004355 Sent␣
↪0x|1E:00|2A:00|00|00:00|94:02:81:64:84:2E:2F:75:69:2F:44:6F:77:6E:6C:6F:61:64:53:75:63:63:65:73:00
10:20:05.022584 Recv 0x|03:00|2A:00|02|
```

opSound with CMD *PLAY* needs two arguments:

- volume in percent as an integer value [0 - 100]
- name of the sound file (without extension ".rsf") as absolute path, or relative to */home/root/lms2012/sys/*

The default *sync_mode* of the USB protocol is *SYNC*. This is why the direct command was replied.

Playing Sound Files repeatedly

As above, take an USB cable, connect your EV3 brick with your computer and replace MAC-address by the one of your EV3 brick, then start this program.

```
import ev3_dc as ev3
import time

my_ev3 = ev3.EV3(protocol=ev3.USB, host='00:16:53:42:2B:99')
my_ev3.verbosity = 1

ops = b''.join((
    ev3.opSound,  # operation
    ev3.REPEAT,  # CMD
    ev3.LCX(100),  # VOLUME
```

(continues on next page)

(continued from previous page)

```
    ev3.LCS('./ui/DownloadSucces')   # NAME
))
my_ev3.send_direct_cmd(ops)

time.sleep(5)
ops = b''.join((
    ev3.opSound,
    ev3.BREAK
))
my_ev3.send_direct_cmd(ops)
```

This program sends two direct commands with a timespan of 5 sec. between them. The first one starts the repeated playing of a sound file, the second stops the playing.

The output:

```
10:26:20.466604 Sent␣
↪0x|1E:00|2A:00|00|00:00|94:03:81:64:84:2E:2F:75:69:2F:44:6F:77:6E:6C:6F:61:64:53:75:63:63:65:73:00
10:26:20.481941 Recv 0x|03:00|2A:00|02|
10:26:25.487598 Sent 0x|07:00|2B:00|00|00:00|94:00|
10:26:25.500652 Recv 0x|03:00|2B:00|02|
```

EV3 increments the message counter. The first command got 0x|2A:00|, which is the value 42, the second command got 0x|2B:00| (value 43).

Playing Tones

We send a direct command, that plays a flourish in c, which consists of four tones:

- c' (262 Hz)
- e' (330 Hz)
- g' (392 Hz)
- c" (523 Hz)

```
import ev3_dc as ev3

my_ev3 = ev3.EV3(protocol=ev3.USB, host='00:16:53:42:2B:99')

ops = b''.join((
    ev3.opSound,   # operation
    ev3.TONE,   # CMD
    ev3.LCX(1),   # volume
    ev3.LCX(262),   # frequency
    ev3.LCX(1000),   # duration
    ev3.opSound_Ready,   # operation
    ev3.opSound,
    ev3.TONE,
    ev3.LCX(1),
    ev3.LCX(330),
    ev3.LCX(1000),
    ev3.opSound_Ready,
    ev3.opSound,
    ev3.TONE,
    ev3.LCX(1),
```

(continues on next page)

(continued from previous page)

```
    ev3.LCX(392),
    ev3.LCX(1000),
    ev3.opSound_Ready,
    ev3.opSound,
    ev3.TONE,
    ev3.LCX(2),
    ev3.LCX(523),
    ev3.LCX(2000)
))
my_ev3.send_direct_cmd(ops)
```

The single direct command consists of 7 operations. *opSound_Ready* prevents interruption. Without it, only the last tone could be heard. The duration is in milliseconds.

Drawing and Timers

Contolling time is an important aspect in real time programs. We have seen how to wait until a tone ended and we waited in the python program until we stopped the repeated playing of a sound file. The operation set of the EV3 includes timer operations which allow to wait in the execution of a direct command. This needs the following two operations:

opTimer_Wait = 0x|85| with two arguments:

- (Data32) TIME: Time to wait (in milliseconds)
- (Data32) TIMER: Variable used for timing

This operation writes a 4-bytes timestamp into the local or global memory.

opTimer_Ready = 0x|86| with one argument:

- (Data32) TIMER: Variable used for timing

This operation reads a timestamp and waits until the actual time reaches the value of this timestamp.

We test the timer operations with a program that draws a triangle. This needs operation *opUI_Draw* with CMD *LINE* three times.

opUI_Draw = 0x|84| with CMD LINE = 0x|03| and the arguments:

- (Data8) COLOR: Specify either black or white, [0: White, 1: Black]
- (Data16) X0: Specify X start point, [0 - 177]
- (Data16) Y0: Specify Y start point, [0 - 127]
- (Data16) X1: Specify X end point
- (Data16) Y1: Specify Y end point

The program:

```
import ev3_dc as ev3

my_ev3 = ev3.EV3(protocol=ev3.USB, host='00:16:53:42:2B:99')

ops = b''.join((
    ev3.opUI_Draw,
    ev3.TOPLINE,
    ev3.LCX(0),   # ENABLE
```

(continues on next page)

(continued from previous page)

```
    ev3.opUI_Draw,
    ev3.FILLWINDOW,
    ev3.LCX(0),  # COLOR
    ev3.LCX(0),  # Y0
    ev3.LCX(0),  # Y1
    ev3.opUI_Draw,
    ev3.UPDATE,
    ev3.opTimer_Wait,
    ev3.LCX(2000),
    ev3.LVX(0),
    ev3.opTimer_Ready,
    ev3.LVX(0),
    ev3.opUI_Draw,
    ev3.LINE,
    ev3.LCX(1),  # COLOR
    ev3.LCX(2),  # X0
    ev3.LCX(125),  # Y0
    ev3.LCX(88),  # X1
    ev3.LCX(2),  # Y1
    ev3.opUI_Draw,
    ev3.UPDATE,
    ev3.opTimer_Wait,
    ev3.LCX(1000),
    ev3.LVX(0),
    ev3.opTimer_Ready,
    ev3.LVX(0),
    ev3.opUI_Draw,
    ev3.LINE,
    ev3.LCX(1),  # COLOR
    ev3.LCX(88),  # X0
    ev3.LCX(2),  # Y0
    ev3.LCX(175),  # X1
    ev3.LCX(125),  # Y1
    ev3.opUI_Draw,
    ev3.UPDATE,
    ev3.opTimer_Wait,
    ev3.LCX(1000),
    ev3.LVX(0),
    ev3.opTimer_Ready,
    ev3.LVX(0),
    ev3.opUI_Draw,
    ev3.LINE,
    ev3.LCX(1),  # COLOR
    ev3.LCX(175),  # X0
    ev3.LCX(125),  # Y0
    ev3.LCX(2),  # X1
    ev3.LCX(125),  # Y1
    ev3.opUI_Draw,
    ev3.UPDATE
))
my_ev3.send_direct_cmd(ops, local_mem=4)
```

This program cleans the display, then waits for two seconds, draws a line, waits for one second, draws another line, waits and finally draws a third line. It needs 4 bytes of local memory, which are multiple times written and red. *opTimer_Wait* writes a timestamp to local memory address 0 and *opTimer_Ready* reads it from local memory address 0.

Obviously the timing can be done in the local program or in the direct command. We change the program:

```
import ev3_dc as ev3
from time import sleep

my_ev3 = ev3.EV3(protocol=ev3.USB, host='00:16:53:42:2B:99')

ops = b''.join((
    ev3.opUI_Draw,
    ev3.TOPLINE,
    ev3.LCX(0),   # ENABLE
    ev3.opUI_Draw,
    ev3.FILLWINDOW,
    ev3.LCX(0),   # COLOR
    ev3.LCX(0),   # Y0
    ev3.LCX(0),   # Y1
    ev3.opUI_Draw,
    ev3.UPDATE
))
my_ev3.send_direct_cmd(ops)

sleep(2)
ops = b''.join((
    ev3.opUI_Draw,
    ev3.LINE,
    ev3.LCX(1),   # COLOR
    ev3.LCX(2),   # X0
    ev3.LCX(125),   # Y0
    ev3.LCX(88),   # X1
    ev3.LCX(2),   # Y1
    ev3.opUI_Draw,
    ev3.UPDATE
))
my_ev3.send_direct_cmd(ops)

sleep(1)
ops = b''.join((
    ev3.opUI_Draw,
    ev3.LINE,
    ev3.LCX(1),   # COLOR
    ev3.LCX(88),   # X0
    ev3.LCX(2),   # Y0
    ev3.LCX(175),   # X1
    ev3.LCX(125),   # Y1
    ev3.opUI_Draw,
    ev3.UPDATE
))
my_ev3.send_direct_cmd(ops)

sleep(1)
ops = b''.join((
    ev3.opUI_Draw,
    ev3.LINE,
    ev3.LCX(1),   # COLOR
    ev3.LCX(175),   # X0
    ev3.LCX(125),   # Y0
    ev3.LCX(2),   # X1
    ev3.LCX(125),   # Y1
```

(continues on next page)

(continued from previous page)

```
    ev3.opUI_Draw,
    ev3.UPDATE
))
my_ev3.send_direct_cmd(ops)
```

Both alternatives result in the same behaviour of the display but are different. The first version needs less communication but blocks the EV3 device for four seconds (until the direct command ends its execution). The second version needs four direct commands but does not block the EV3 brick. All its direct commands need a short execution time and allow to send other direct commands in between.

Simulating Button presses

In this example, we shut down the EV3 brick by simulating button presses. We use two operations:

opUI_Button = 0x|83| with CMD *PRESS* = 0x|05| needs one argument:

- BUTTON
  - NO_BUTTON = 0x|00|
  - UP_BUTTON = 0x|01|
  - ENTER_BUTTON = 0x|02|
  - DOWN_BUTTON = 0x|03|
  - RIGHT_BUTTON = 0x|04|
  - LEFT_BUTTON = 0x|05|
  - BACK_BUTTON = 0x|06|
  - ANY_BUTTON = 0x|07|

opUI_Button = 0x|83| with CMD *WAIT_FOR_PRESS* = 0x|03| needs no argument.

To prevent interruption, we need to wait until the initiated operations are finished. This is done by the second operation.

The program:

```
import ev3_dc as ev3

my_ev3 = ev3.EV3(protocol=ev3.USB, host='00:16:53:42:2B:99')

ops = b''.join((
    ev3.opUI_Button,  # operation
    ev3.PRESS,  # CMD
    ev3.BACK_BUTTON,
    ev3.opUI_Button,  # operation
    ev3.WAIT_FOR_PRESS,  # CMD
    ev3.opUI_Button,
    ev3.PRESS,
    ev3.RIGHT_BUTTON,
    ev3.opUI_Button,
    ev3.WAIT_FOR_PRESS,
    ev3.opUI_Button,
    ev3.PRESS,
    ev3.ENTER_BUTTON
))
my_ev3.send_direct_cmd(ops)
```

Reading data from EV3's sensors

Direct commands allow to read data from your EV3 device. The most important operation for reading data is:

opInput_Device = 0x|99| with CMD *READY_RAW* = 0x|1C|

Arguments

- (Data8) LAYER: Specify chain layer number [0-3]
- (Data8) NO: Port number
- (Data8) TYPE: Specify device type (0 = Don't change type)
- (Data8) MODE: Device mode [0-7] (-1 = Don't change mode)
- (Data8) VALUES: Number of return values

Returns

- (Data32) VALUE1: First value received from sensor in the specified mode

There are two siblings, that read data a bit different:

- *opInput_Device* = 0x|99| with CMD *READY_PCT* = 0x|1B| reads integer data in the range [0 - 100], that must be interpreted as a percentage.
- *opInput_Device* = 0x|99| with CMD *READY_SI* = 0x|1D| reads floating point data.

Return data can be written to the local or global memory. Use function `LVX()` to address the local memory and `GVX()` to address the global memory (e.g. GVX(0) addresses the first byte of the global memory).

Another operation, that may be important for sensors, resets the sensor at a specific port. This sets the sensor to its initial state and clears its counters.

opInput_Device = 0x|99| with CMD *CLR_CHANGES* = 0x|1A|

Arguments

- (Data8) LAYER: Specify chain layer number [0-3]
- (Data8) NO: Port number

Introspection

There is an operation, that asks for the type and mode of a sensor at a specified port.

opInput_Device = 0x|99| with CMD *GET_TYPEMODE* = 0x|05|

Arguments

- (Data8) LAYER: chain layer number
- (Data8) NO: port number

Returns

- (Data8) TYPE: device type
- (Data8) MODE: device mode

Please connect some sensors to your sensor ports and some motors to your motor ports. Then connect your EV3 brick and your computer with an USB cable. Replace MAC-address by the one of your EV3 brick. The following program sends two direct commands, the first asks for the sensors, the second for the motors.

```
import ev3_dc as ev3
import struct

my_ev3 = ev3.EV3(protocol=ev3.USB, host='00:16:53:42:2B:99')
my_ev3.verbosity = 1


def create_ops(ports: tuple, motors=False):
    if motors:
        ports = tuple(ev3.port_motor_input(port) for port in ports)
    ops = b''
    for i in range(4):
        ops += b''.join((
            ev3.opInput_Device,  # operation
            ev3.GET_TYPEMODE,  # CMD
            ev3.LCX(0),  # LAYER
            ports[i],  # NO
            ev3.GVX(2*i),  # TYPE (output)
            ev3.GVX(2*i + 1)  # MODE (output)
        ))
    return ops


def print_table(port_names: tuple, answer: tuple):
    print('-'*20)
    print('port | type | mode |')
    print('-'*20)
    for i in range(4):
        print(
            '   {} |'.format(
                port_names[i]
            ),
            end=''
        )
        if answer[2*i] == 126:
            print('    - |    - |')
        else:
            print(
                '  {:3d} |  {:3d} |'.format(
                    answer[2*i],
                    answer[2*i + 1]
                )
            )
    print('-'*20)
    print()


# sensors
ports = (ev3.PORT_1, ev3.PORT_2, ev3.PORT_3, ev3.PORT_4)
ops = create_ops(ports)
reply = my_ev3.send_direct_cmd(ops, global_mem=8)
answer = struct.unpack('8B', reply)

print()
print('Sensor ports:')
print_table(
    ('1', '2', '3', '4'),
```

(continues on next page)

(continued from previous page)

```
    answer
)

# motors
ports = (ev3.PORT_A, ev3.PORT_B, ev3.PORT_C, ev3.PORT_D)
ops = create_ops(ports, motors=True)
reply = my_ev3.send_direct_cmd(ops, global_mem=8)
answer = struct.unpack('8B', reply)

print()
print('Motor ports:')
print_table(
    ('A', 'B', 'C', 'D'),
    answer
)
```

Some Remarks:

- Each operation *opInput_Device* with CMD *GET_TYPEMODE* answers with two bytes of data, one byte for the type, another for the mode.
- It's the python program that decides, how to place the data into the global memory. Every *GVX()* directs some output data to an address of the global memory.
- *reply* is a byte string of 8 bytes length, *answer* is a tuple of 8 byte numbers.
- struct is the tool of choice to translate binary data into python data types.
- *port_motor_input()* allows to use the same motor port constants for input and output.
- type *126* stands for *no sensor connected*.

The output:

```
09:25:12.400013 Sent␣
↪0x|1D:00|2A:00|00|08:00|99:05:00:00:60:61:99:05:00:01:62:63:99:05:00:02:64:65:99:05:00:03:66:67|
09:25:12.410124 Recv 0x|0B:00|2A:00|02|10:00:1D:00:21:00:7E:00|

Sensor ports:
--------------------
port | type | mode |
--------------------
   1 |   16 |    0 |
   2 |   29 |    0 |
   3 |   33 |    0 |
   4 |    - |    - |
--------------------

09:25:12.411241 Sent␣
↪0x|1D:00|2B:00|00|08:00|99:05:00:10:60:61:99:05:00:11:62:63:99:05:00:12:64:65:99:05:00:13:66:67|
09:25:12.417945 Recv 0x|0B:00|2B:00|02|07:00:7E:00:08:00:07:00|

Motor ports:
--------------------
port | type | mode |
--------------------
   A |    7 |    0 |
   B |    - |    - |
   C |    8 |    0 |
```

(continues on next page)

(continued from previous page)

```
   D |     7 |     0 |
--------------------
```

Section 5 Device type list in EV3 Firmware Developer Kit lists the sensor types and modes of the EV3 device and helps to understand these numbers.

Touch mode of the Touch Sensor

We use operation *opInput_Device* to ask the touch sensor if it currently is touched. Connect your touch sensor with port 1, take an USB-cable and connect your computer with your EV3 brick, then run this program:

```
import ev3_dc as ev3
import struct

my_ev3 = ev3.EV3(protocol=ev3.USB)
my_ev3.verbosity = 1

# touch sensor at port 1
ops = b''.join((
    ev3.opInput_Device,  # operation
    ev3.READY_SI,  # CMD
    ev3.LCX(0),  # LAYER
    ev3.PORT_1,  # NO
    ev3.LCX(16),  # TYPE (EV3-Touch)
    ev3.LCX(0),  # MODE (Touch)
    ev3.LCX(1),  # VALUES
    ev3.GVX(0)  # VALUE1 (output)
))
reply = my_ev3.send_direct_cmd(ops, global_mem=4)
touched = struct.unpack('<f', reply)[0]

print()
print(
        'The sensor is',
        ('not touched', 'touched')[int(touched)]
)
```

Some remarks:

- The single return value of *opInput_Device* with CMD *READY_SI* is a floating point number of 4 bytes length in little endian notation.
- With GVX(0) we write it to the global memory address 0. This says, it takes the first 4 bytes of the global memory.
- Method *send_direct_cmd()* skips the leading bytes of the reply and returns the global memory only.
- struct is the tool of choice to translate the packed binary little endian data into python data format. `struct.unpack()` returns a tuple, from where we pick the first (and only) item.

The output:

```
09:35:17.516913 Sent 0x|0D:00|2A:00|00|04:00|99:1D:00:00:10:00:01:60|
09:35:17.524934 Recv 0x|07:00|2A:00|02|00:00:80:3F|

The sensor is touched
```

0x|00:00:80:3F| is the little endian notation of the floating point number 1.0.

Bump mode of the Touch Sensor

The bump mode of the touch sensor counts the number of touches since the last reset. The following program resets the counter of the touch sensor, waits for five seconds, then asks about the number of touches.

If you own a WiFi dongle and both, you computer and your EV3 brick are connected to the WiFi, then you can start the following program. If not, replace the protocol by USB or by BLUETOOTH.

```python
import ev3_dc as ev3
import struct
from time import sleep

my_ev3 = ev3.EV3(protocol=ev3.WIFI)
my_ev3.verbosity = 1

# clear port 1
ops = b''.join((
    ev3.opInput_Device,  # operation
    ev3.CLR_CHANGES,  # CMD
    ev3.LCX(0),  # LAYER
    ev3.PORT_1  # NO
))
my_ev3.send_direct_cmd(ops)

print('\ncounting starts now ...\n')
sleep(5)

# touch sensor at port 1
ops = b''.join((
    ev3.opInput_Device,  # operation
    ev3.READY_SI,  # CMD
    ev3.LCX(0),  # LAYER
    ev3.PORT_1,  # NO
    ev3.LCX(16),  # TYPE (EV3-Touch)
    ev3.LCX(1),  # MODE (Bump)
    ev3.LCX(1),  # VALUES
    ev3.GVX(0)  # VALUE1 (output)
))
reply = my_ev3.send_direct_cmd(ops, global_mem=4)
touched = struct.unpack('<f', reply)[0]

print()
print(
        'The sensor was touched',
        int(touched),
        'times'
)
```

The output:

```
09:37:04.402440 Sent 0x|09:00|2A:00|80|00:00|99:1A:00:00|

counting starts now ...

09:37:09.418332 Sent 0x|0D:00|2B:00|00|04:00|99:1D:00:00:10:01:01:60|
```

(continues on next page)

(continued from previous page)

```
09:37:09.435870 Recv 0x|07:00|2B:00|02|00:00:40:41|

The sensor was touched 12 times
```

If you compare the two direct commands, you will realize some differences:

- The length is different.
- The message counter has been incremented.
- The message types are different, the first one is *DIRECT_COMMAND_NO_REPLY*, the second one is *DIRECT_COMMAND_REPLY*. Consequently, the first command does not get a reply. If you use protocol USB, this will change and all direct commands will be replied.
- The header is different. The first direct command does not use any global or local memory, the second needs 4 bytes of global memory.
- The operations are different, which is not surprising.

Measure distances

Use operation *opInput_Device* to read data of the infrared sensor. Connect your EV3 infrared sensor with port 3, take an USB-cable and connect your computer with your EV3 brick, then run this program:

```python
import ev3_dc as ev3
import struct

my_ev3 = ev3.EV3(protocol=ev3.USB)
my_ev3.verbosity = 1

# infrared sensor at port 3
ops = b''.join((
    ev3.opInput_Device,
    ev3.READY_SI,
    ev3.LCX(0),   # LAYER
    ev3.PORT_3,   # NO
    ev3.LCX(33),   # TYPE - EV3-IR
    ev3.LCX(0),   # MODE - Proximity
    ev3.LCX(1),   # VALUES
    ev3.GVX(0)   # VALUE1
))
reply = my_ev3.send_direct_cmd(ops, global_mem=4)
distance = struct.unpack('<f', reply)[0]

print('\nSomething detected at a distance of {:2.0f} cm.'.format(distance))
```

The output:

```
09:45:34.223216 Sent 0x|0E:00|2A:00|00|04:00|99:1D:00:02:81:21:00:01:60|
09:45:34.229976 Recv 0x|07:00|2A:00|02|00:00:D0:41|

Something detected at a distance of 26 cm.
```

Seeker and Beacon

Combining the EV3 infrared sensor and the EV3 beacon identifies the position of one to four beacons. A beacon send signals on one of four channels and the infrared sensor measures its own position relative to the position the beacon.

Connect your EV3 infrared sensor with port 3, take an USB-cable and connect your computer with your EV3 brick, select a channel, place it in front of the infrared sensor, then run this program:

```
import ev3_dc as ev3
import struct

my_ev3 = ev3.EV3(protocol=ev3.USB)
my_ev3.verbosity = 1

ops_read = b''.join((
    ev3.opInput_Device,  # operation
    ev3.READY_RAW,  # CMD
    ev3.LCX(0),  # LAYER
    ev3.PORT_3,  # NO
    ev3.LCX(33),  # TYPE - IR
    ev3.LCX(1),  # MODE - Seeker
    ev3.LCX(8),  # VALUES
    ev3.GVX(0),  # VALUE1 - heading   channel 1
    ev3.GVX(4),  # VALUE2 - proximity channel 1
    ev3.GVX(8),  # VALUE3 - heading   channel 2
    ev3.GVX(12),  # VALUE4 - proximity channel 2
    ev3.GVX(16),  # VALUE5 - heading   channel 3
    ev3.GVX(20),  # VALUE6 - proximity channel 3
    ev3.GVX(24),  # VALUE7 - heading   channel 4
    ev3.GVX(28)  # VALUE8 - proximity channel 4
))
reply = my_ev3.send_direct_cmd(ops_read, global_mem=32)
answer = struct.unpack('<8i', reply)

for i in range(4):
    # proximity (little endian) == 0x|00:00:00:80| means no signal
    if answer[2*i + 1] == -2147483648:
        continue

    print(
        '\nchannel: {}, heading: {}, proximity: {}'.format(
            i + 1,
            answer[2*i],
            answer[2*i + 1]
        )
    )
```

Some remarks:

- Type 33 (IR) with Mode 1 (Seeker) writes 8 data values, heading and proximity of four channels.
- In case of CMD *READY_RAW*, these are 8 integer values, each of four bytes length. This needs 32 bytes of global memory.
- struct translates the packed binary little endian data of the global memory and returns a tuple of eight integer values.
- A proximity of 0x|00:00:00:80| (little endian, the heighest bit is 1, all others are 0) has a special meaning. It says, on this channel the infrared sensor did not receive a signal. Interpeted as a signed litlle endian integer,

0x|00:00:00:80| becomes $-2,147,483,648 = -2^{31}$, the smallest of all values.

- Using a single beacon means, three channels without signal, one channel with. Channels without signal are sorted out.

The output:

```
10:05:43.514714 Sent␣
↪0x|15:00|2A:00|00|20:00|99:1C:00:02:81:21:01:08:60:64:68:6C:70:74:78:7C|
10:05:44.629441 Recv␣
↪0x|23:00|2A:00|02|00:00:00:00:00:00:00:80:EB:FF:FF:FF:1B:00:00:00:00:00:00:00:00:00:80:00:00:00

channel: 2, heading: -21, proximity: 27
```

Some remarks:

- Heading is in the range [-25 - 25], negative values stand for the left, 0 for straight, positive for the right side.
- Proximity is in the range [0 - 100] and measures in cm.
- In my case, the beacon was far left, 27 cm apart and sended on channel 2.

Reading the color

We use operation *opInput_Device* to read data of the color sensor. Connect your color sensor with port 2, take an USB-cable and connect your computer with your EV3 brick, then run this program:

```python
import ev3_dc as ev3
import struct

my_ev3 = ev3.EV3(protocol=ev3.USB)
my_ev3.verbosity = 1

# color sensor at port 2
ops = b''.join((
    ev3.opInput_Device,  # operation
    ev3.READY_RAW,  # CMD
    ev3.LCX(0),  # LAYER
    ev3.PORT_2,  # NO
    ev3.LCX(29),  # TYPE (EV3-Color)
    ev3.LCX(2),  # MODE (Color)
    ev3.LCX(1),  # VALUES
    ev3.GVX(0)  # VALUE1 (output)
))
reply = my_ev3.send_direct_cmd(ops, global_mem=4)
color_nr = struct.unpack('<i', reply)[0]

color_str = (
    'none',
    'black',
    'blue',
    'green',
    'yellow',
    'red',
    'white',
    'brown'
)[color_nr]
print('\nThis color is', color_str)
```

The output:

```
09:49:32.461804 Sent 0x|0D:00|2A:00|00|04:00|99:1C:00:01:1D:02:01:60|
09:49:32.467874 Recv 0x|07:00|2A:00|02|03:00:00:00|

This color is green
```

There are some more color sensor modes, maybe you like to test these:

- Mode 0 (Reflected) - switches on the red light and measures the inensity of the reflection, which is dependent from distance, color and the reflection factor of the surface.
- Mode 1 (Ambient) - switches on the blue light (why?) and measures the intensity of the ambient light.
- Mode 4 (RGB-Raw) - switches on red, green and blue light and measures the intensity of the reflected light.

Reading the current position of motors

If two large motors are connected with ports A and D, you can start this program:

```
import ev3_dc as ev3
import struct

my_ev3 = ev3.EV3(protocol=ev3.BLUETOOTH, host='00:16:53:42:2B:99')

ops = b''.join((
    ev3.opInput_Device,
    ev3.READY_SI,
    ev3.LCX(0),  # LAYER
    ev3.port_motor_input(ev3.PORT_A),  # NO
    ev3.LCX(7),  # TYPE (EV3-Large-Motor)
    ev3.LCX(0),  # MODE (Degree)
    ev3.LCX(1),  # VALUES
    ev3.GVX(0),  # VALUE1
    ev3.opInput_Device,
    ev3.READY_RAW,
    ev3.LCX(0),  # LAYER
    ev3.port_motor_input(ev3.PORT_D),  # NO
    ev3.LCX(7),  # TYPE
    ev3.LCX(0),  # MODE
    ev3.LCX(1),  # VALUES
    ev3.GVX(4)  # VALUE1
))
reply = my_ev3.send_direct_cmd(ops, global_mem=8)
pos_a, pos_d = struct.unpack('<fi', reply)
print(
    "positions in degrees (ports A and D): {} and {}".format(
        pos_a,
        pos_d
    )
)
```

Section *5 Device type list* in *EV3 Firmware Developer Kit* lists the sensors of the EV3 device. If you want to read the positions of large motors in degrees, you will set TYPE=7 and MODE=0. We read one value from each.

For demonstration pupose only, we use two different CMDs, *READY_SI* and *READY_RAW*. Both of them read the current position of a motor, but the first writes floating point data, the second integer data. We use 8 bytes of global memory. The first 4 bytes hold the position of motor A as a floating point number. The next 4 bytes hold the position of motor D as an integer. Module struct is the tool of choice to translate the packed binary little endian data into a float and an int.

Moving motors

A number of operations is used for motor movements.

Exact movements, blocking the EV3 brick

Exact and smooth movements of a mootor are our first theme. We start with using four operations:

opOutput_Reset = 0x|A2|

Arguments

- (Data8) LAYER: chain layer number
- (Data8) NOS: port number (or a combination of port numbers)

The EV3 brick tracks exact movements and does some corrections of overshooting or manual movements. *opOutput_Reset* resets these tracking informations. It does not clear the counter.

opOutput_Step_Speed = 0x|AE|

Arguments

- (Data8) LAYER: chain layer number
- (Data8) NOS: port number (or a combination of port numbers)
- (Data8) SPEED: direction (sign) and speed of movement [-100, 100]
- (Data32) STEP1: length of acceleration
- (Data32) STEP2: length of constant speed movement
- (Data32) STEP3: length of deceleration
- (Data8) BRAKE: flag if ending with floating motor or active break [0: Float, 1: Break]

This operation defines a smooth and exact movement of one or multiple motors. Dependent from the mode, *STEP1*, *STEP2* and *STEP3* are in degrees (default) or rotations.

opOutput_Ready = 0x|AA|

Arguments

- (Data8) LAYER: chain layer number
- (Data8) NOS: port number (or a combination of port numbers)

Starts the movement and waits until the movement has finished.

opOutput_Stop = 0x|A3|

Arguments

- (Data8) LAYER: chain layer number
- (Data8) NOS: port number (or a combination of port numbers)
- (Data8) BRAKE: flag if ending with floating motor or active break [0: Float, 1: Break]

Stops the current movement of one or multiple motors.

Connect your EV3 medium motor with port B, connect your computer via Bluetooth with your EV3 brick, replace MAC-address with the one of your EV3 brick, then run this program:

```python
import ev3_dc as ev3
from time import sleep


my_ev3 = ev3.EV3(protocol=ev3.BLUETOOTH, host='00:16:53:42:2B:99')
my_ev3.verbosity = 1

jukebox = ev3.Jukebox(ev3_obj=my_ev3)
jukebox.song(ev3.FRERE_JACQUES).start()


def reset():
    ops = b''.join((
        ev3.opOutput_Reset,
        ev3.LCX(0),  # LAYER
        ev3.LCX(ev3.PORT_B)  # NOS
    ))
    my_ev3.send_direct_cmd(ops, sync_mode=ev3.SYNC)


def step_speed(speed: int):
    ops_step_speed = b''.join((
        ev3.opOutput_Step_Speed,
        ev3.LCX(0),  # LAYER
        ev3.LCX(ev3.PORT_B),  # NOS
        ev3.LCX(speed),  # SPEED
        ev3.LCX(15),  # STEP1
        ev3.LCX(60),  # STEP2
        ev3.LCX(15),  # STEP3
        ev3.LCX(1)  # BRAKE - yes
    ))
    ops_ready = b''.join((
        ev3.opOutput_Ready,
        ev3.LCX(0),  # LAYER
        ev3.LCX(ev3.PORT_B)  # NOS
    ))
    my_ev3.send_direct_cmd(ops_step_speed + ops_ready, sync_mode=ev3.SYNC)


def stop():
    ops = b''.join((
        ev3.opOutput_Stop,
        ev3.LCX(0),  # LAYER
        ev3.LCX(ev3.PORT_B),  # NOS
        ev3.LCX(0)  # BRAKE - no
    ))
    my_ev3.send_direct_cmd(ops)


speed = 5

reset()
for i in range(5):
    step_speed(speed)
    step_speed(-speed)
```

(continues on next page)

(continued from previous page)

```
sleep(.2)
stop()
```

Some remarks:

- Function `reset()` resets the tracking information of the motor at port B.
- Function `step_speed()` does a 90 ° smooth movement of the motor at port B. Dependent from the sign of SPEED the movement is forwards or backwards. The three numbers STEP1, STEP2 and STEP3 define the lengths of the acceleration, the constant speed and the deceleration phase, all of them in degrees. The movement ends with an active break, which holds the motor in a defined position. It waits until the movement has finished.
- Function `stop()` releases the brake. This is done 0.2 sec. after the last movement has finished.
- There are 10 slow and smooth movements of the motor, 5 times forwards and backwards. If you fix an infrared sensor on top of the shaft, this looks like headshaking. Changing the speed will change the character of the headshaking.
- Setting *sync_mode=SYNC* allows to get the reply just when the movement has finished.
- The program plays the song *Frère Jacques* parallel to the motor movement.
- Using two classes *EV3* and *Jukebox* is not necessary. *Jukebox* as a subclass of *EV3* would have done the job alone. But this example demonstrates, how specialized subclasses of *EV3* can handle specific tasks, like *Jukebox* handles sound. And multiple subclasses of *EV3* can work together.

The output:

```
11:52:26.168681 Sent 0x|08:00|2A:00|00|00:00|A2:00:02|
11:52:26.247070 Recv 0x|03:00|2A:00|02|
11:52:26.248399 Sent 0x|11:00|2D:00|00|00:00|AE:00:02:05:0F:81:3C:0F:01:AA:00:02|
11:52:27.402000 Recv 0x|03:00|2D:00|02|
11:52:27.403093 Sent 0x|11:00|2F:00|00|00:00|AE:00:02:3B:0F:81:3C:0F:01:AA:00:02|
11:52:28.578030 Recv 0x|03:00|2F:00|02|
11:52:28.578578 Sent 0x|11:00|30:00|00|00:00|AE:00:02:05:0F:81:3C:0F:01:AA:00:02|
11:52:29.735028 Recv 0x|03:00|30:00|02|
11:52:29.736302 Sent 0x|11:00|33:00|00|00:00|AE:00:02:3B:0F:81:3C:0F:01:AA:00:02|
11:52:30.929957 Recv 0x|03:00|33:00|02|
11:52:30.930941 Sent 0x|11:00|35:00|00|00:00|AE:00:02:05:0F:81:3C:0F:01:AA:00:02|
11:52:32.089839 Recv 0x|03:00|35:00|02|
11:52:32.091088 Sent 0x|11:00|38:00|00|00:00|AE:00:02:3B:0F:81:3C:0F:01:AA:00:02|
11:52:33.220884 Recv 0x|03:00|38:00|02|
11:52:33.221437 Sent 0x|11:00|39:00|00|00:00|AE:00:02:05:0F:81:3C:0F:01:AA:00:02|
11:52:34.366040 Recv 0x|03:00|39:00|02|
11:52:34.367271 Sent 0x|11:00|3C:00|00|00:00|AE:00:02:3B:0F:81:3C:0F:01:AA:00:02|
11:52:35.536879 Recv 0x|03:00|3C:00|02|
11:52:35.537949 Sent 0x|11:00|3E:00|00|00:00|AE:00:02:05:0F:81:3C:0F:01:AA:00:02|
11:52:36.735035 Recv 0x|03:00|3E:00|02|
11:52:36.735600 Sent 0x|11:00|3F:00|00|00:00|AE:00:02:3B:0F:81:3C:0F:01:AA:00:02|
11:52:37.870978 Recv 0x|03:00|3F:00|02|
11:52:38.071796 Sent 0x|09:00|43:00|80|00:00|A3:00:02:00|
```

The movement of the motor is the expected, but the song is not! The movements last more than a second each and for this timespan, the EV3 brick is blocked because operation *opOutput_Ready* lets the EV3 brick wait. If you look at the message counters, you find some gaps, where direct commands of the sond were sent.

What we heave learned: *If the timing is done in the direct command, this limits parallel execution.*

Exact Movements, not blocking

We modify the program and replace *opOutput_Ready* by *opOutput_Start*. While the movement takes place, we ask frequently if it still is in progress or has finished (done by *opOutput_Test*). This means more data traffic, but none of the requests will block the EV3 brick. We use these new operations:

opOutput_Start = 0x|A6|

Arguments

- (Data8) LAYER: chain layer number
- (Data8) NOS: port number (or a combination of port numbers)

Starts the movement and does not wait until the movement has finished.

opOutput_Test = 0x|A9|

Arguments

- (Data8) LAYER: chain layer number
- (Data8) NOS: port number (or a combination of port numbers)

Returns

- (Data8) BUSY: flag if motor is busy [0 = Ready, 1 = Busy]

Tests if a motor is currently busy.

Connect your EV3 medium motor with port B, connect your computer via Bluetooth with your EV3 brick, replace MAC-address with the one of your EV3 brick, then run this program:

```
import ev3_dc as ev3
import struct
from time import sleep


my_ev3 = ev3.EV3(protocol=ev3.BLUETOOTH, host='00:16:53:42:2B:99')
my_ev3.verbosity = 1

jukebox = ev3.Jukebox(ev3_obj=my_ev3)
jukebox.song(ev3.FRERE_JACQUES).start()


def reset():
    ops = b''.join((
        ev3.opOutput_Reset,
        ev3.LCX(0),  # LAYER
        ev3.LCX(ev3.PORT_B)  # NOS
    ))
    my_ev3.send_direct_cmd(ops, sync_mode=ev3.SYNC)


def step_speed(speed: int):
    ops_step_speed = b''.join((
        ev3.opOutput_Step_Speed,
        ev3.LCX(0),  # LAYER
        ev3.LCX(ev3.PORT_B),  # NOS
        ev3.LCX(speed),  # SPEED
        ev3.LCX(15),  # STEP1
        ev3.LCX(60),  # STEP2
```

(continues on next page)

(continued from previous page)

```
        ev3.LCX(15),  # STEP3
        ev3.LCX(1)  # BRAKE - yes
    ))
    ops_start = b''.join((
        ev3.opOutput_Start,
        ev3.LCX(0),  # LAYER
        ev3.LCX(ev3.PORT_B)  # NOS
    ))
    my_ev3.send_direct_cmd(ops_step_speed + ops_start)


def test():
    ops = b''.join((
        ev3.opOutput_Test,
        ev3.LCX(0),  # LAYER
        ev3.LCX(ev3.PORT_B),  # NOS
        ev3.GVX(0)  # BUSY
    ))
    reply = my_ev3.send_direct_cmd(ops, global_mem=4)
    return struct.unpack('<i', reply)[0]


def stop():
    ops = b''.join((
        ev3.opOutput_Stop,
        ev3.LCX(0),  # LAYER
        ev3.LCX(ev3.PORT_B),  # NOS
        ev3.LCX(0)  # BRAKE - no
    ))
    my_ev3.send_direct_cmd(ops)


speed = 5

reset()
for i in range(5):
    step_speed(speed)
    sleep(.2)
    while test():
        sleep(.2)

    step_speed(-speed)
    sleep(.2)
    while test():
        sleep(.2)

sleep(.2)
stop()
```

Some remarks:

- *opOutput_Ready* has been replaced by *opOutput_Start*. This starts the movement, but does not wait for its end.
- Instead of waiting, this program uses *opOutput_Test* to ask frequently, if the movement is still in progress.
- If still your song is not played correctly, use protocols USB or WiFi instead of Bluetooth, because these are faster and speed helps to prevent conflicts.

The output:

```
12:21:08.851739 Sent 0x|08:00|2A:00|00|00:00|A2:00:02|
12:21:08.903092 Recv 0x|03:00|2A:00|02|
12:21:08.904440 Sent 0x|11:00|2D:00|80|00:00|AE:00:02:05:0F:81:3C:0F:01:A6:00:02|
12:21:09.105336 Sent 0x|09:00|2E:00|00|01:00|A9:00:02:60|
12:21:09.174974 Recv 0x|04:00|2E:00|02|01|
12:21:09.375951 Sent 0x|09:00|2F:00|00|01:00|A9:00:02:60|
12:21:09.444917 Recv 0x|04:00|2F:00|02|01|
12:21:09.645735 Sent 0x|09:00|31:00|00|01:00|A9:00:02:60|
12:21:09.715081 Recv 0x|04:00|31:00|02|01|
12:21:09.916029 Sent 0x|09:00|32:00|00|01:00|A9:00:02:60|
12:21:09.991093 Recv 0x|04:00|32:00|02|01|
12:21:10.191946 Sent 0x|09:00|34:00|00|01:00|A9:00:02:60|
12:21:10.262916 Recv 0x|04:00|34:00|02|00|
12:21:10.263476 Sent 0x|11:00|35:00|80|00:00|AE:00:02:3B:0F:81:3C:0F:01:A6:00:02|
12:21:10.464500 Sent 0x|09:00|36:00|00|01:00|A9:00:02:60|
12:21:10.535111 Recv 0x|04:00|36:00|02|01|
12:21:10.736109 Sent 0x|09:00|38:00|00|01:00|A9:00:02:60|
12:21:10.777892 Recv 0x|04:00|38:00|02|01|
12:21:10.978716 Sent 0x|09:00|39:00|00|01:00|A9:00:02:60|
12:21:11.044970 Recv 0x|04:00|39:00|02|01|
12:21:11.245923 Sent 0x|09:00|3A:00|00|01:00|A9:00:02:60|
12:21:11.303016 Recv 0x|04:00|3A:00|02|01|
12:21:11.504236 Sent 0x|09:00|3D:00|00|01:00|A9:00:02:60|
12:21:11.575097 Recv 0x|04:00|3D:00|02|00|
12:21:11.575639 Sent 0x|11:00|3E:00|80|00:00|AE:00:02:05:0F:81:3C:0F:01:A6:00:02|
12:21:11.776573 Sent 0x|09:00|3F:00|00|01:00|A9:00:02:60|
12:21:11.842046 Recv 0x|04:00|3F:00|02|01|
12:21:12.043106 Sent 0x|09:00|41:00|00|01:00|A9:00:02:60|
12:21:12.112103 Recv 0x|04:00|41:00|02|01|
12:21:12.313026 Sent 0x|09:00|42:00|00|01:00|A9:00:02:60|
12:21:12.375051 Recv 0x|04:00|42:00|02|01|
12:21:12.575968 Sent 0x|09:00|44:00|00|01:00|A9:00:02:60|
12:21:12.637077 Recv 0x|04:00|44:00|02|01|
12:21:12.838115 Sent 0x|09:00|45:00|00|01:00|A9:00:02:60|
12:21:12.908110 Recv 0x|04:00|45:00|02|00|
12:21:12.908696 Sent 0x|11:00|46:00|80|00:00|AE:00:02:3B:0F:81:3C:0F:01:A6:00:02|
12:21:13.109496 Sent 0x|09:00|48:00|00|01:00|A9:00:02:60|
12:21:13.121873 Recv 0x|04:00|48:00|02|01|
12:21:13.322848 Sent 0x|09:00|49:00|00|01:00|A9:00:02:60|
12:21:13.402117 Recv 0x|04:00|49:00|02|01|
12:21:13.603152 Sent 0x|09:00|4A:00|00|01:00|A9:00:02:60|
12:21:13.657882 Recv 0x|04:00|4A:00|02|01|
12:21:13.858904 Sent 0x|09:00|4D:00|00|01:00|A9:00:02:60|
12:21:13.899888 Recv 0x|04:00|4D:00|02|01|
12:21:14.100762 Sent 0x|09:00|4E:00|00|01:00|A9:00:02:60|
12:21:14.144913 Recv 0x|04:00|4E:00|02|00|
12:21:14.145297 Sent 0x|11:00|4F:00|80|00:00|AE:00:02:05:0F:81:3C:0F:01:A6:00:02|
12:21:14.346331 Sent 0x|09:00|51:00|00|01:00|A9:00:02:60|
12:21:14.389892 Recv 0x|04:00|51:00|02|01|
12:21:14.590822 Sent 0x|09:00|52:00|00|01:00|A9:00:02:60|
12:21:14.657997 Recv 0x|04:00|52:00|02|01|
12:21:14.858864 Sent 0x|09:00|54:00|00|01:00|A9:00:02:60|
12:21:14.944139 Recv 0x|04:00|54:00|02|01|
12:21:15.145073 Sent 0x|09:00|55:00|00|01:00|A9:00:02:60|
12:21:15.206087 Recv 0x|04:00|55:00|02|01|
12:21:15.407067 Sent 0x|09:00|56:00|00|01:00|A9:00:02:60|
12:21:15.476913 Recv 0x|04:00|56:00|02|00|
```

(continues on next page)

(continued from previous page)

```
12:21:15.477296 Sent 0x|11:00|57:00|80|00:00|AE:00:02:3B:0F:81:3C:0F:01:A6:00:02|
12:21:15.678152 Sent 0x|09:00|58:00|00|01:00|A9:00:02:60|
12:21:15.746237 Recv 0x|04:00|58:00|02|01|
12:21:15.947113 Sent 0x|09:00|59:00|00|01:00|A9:00:02:60|
12:21:16.008946 Recv 0x|04:00|59:00|02|01|
12:21:16.209772 Sent 0x|09:00|5C:00|00|01:00|A9:00:02:60|
12:21:16.286122 Recv 0x|04:00|5C:00|02|01|
12:21:16.488816 Sent 0x|09:00|5D:00|00|01:00|A9:00:02:60|
12:21:16.611171 Recv 0x|04:00|5D:00|02|01|
12:21:16.812098 Sent 0x|09:00|5F:00|00|01:00|A9:00:02:60|
12:21:16.895091 Recv 0x|04:00|5F:00|02|00|
12:21:16.895637 Sent 0x|11:00|60:00|80|00:00|AE:00:02:05:0F:81:3C:0F:01:A6:00:02|
12:21:17.096654 Sent 0x|09:00|61:00|00|01:00|A9:00:02:60|
12:21:17.138906 Recv 0x|04:00|61:00|02|01|
12:21:17.339764 Sent 0x|09:00|63:00|00|01:00|A9:00:02:60|
12:21:17.400990 Recv 0x|04:00|63:00|02|01|
12:21:17.601883 Sent 0x|09:00|64:00|00|01:00|A9:00:02:60|
12:21:17.638926 Recv 0x|04:00|64:00|02|01|
12:21:17.839940 Sent 0x|09:00|65:00|00|01:00|A9:00:02:60|
12:21:17.910139 Recv 0x|04:00|65:00|02|01|
12:21:18.111050 Sent 0x|09:00|66:00|00|01:00|A9:00:02:60|
12:21:18.176911 Recv 0x|04:00|66:00|02|00|
12:21:18.177386 Sent 0x|11:00|67:00|80|00:00|AE:00:02:3B:0F:81:3C:0F:01:A6:00:02|
12:21:18.378438 Sent 0x|09:00|68:00|00|01:00|A9:00:02:60|
12:21:18.454102 Recv 0x|04:00|68:00|02|01|
12:21:18.655531 Sent 0x|09:00|6B:00|00|01:00|A9:00:02:60|
12:21:18.699933 Recv 0x|04:00|6B:00|02|01|
12:21:18.900855 Sent 0x|09:00|6C:00|00|01:00|A9:00:02:60|
12:21:18.956985 Recv 0x|04:00|6C:00|02|01|
12:21:19.158315 Sent 0x|09:00|6F:00|00|01:00|A9:00:02:60|
12:21:19.205918 Recv 0x|04:00|6F:00|02|01|
12:21:19.406850 Sent 0x|09:00|71:00|00|01:00|A9:00:02:60|
12:21:19.455956 Recv 0x|04:00|71:00|02|00|
12:21:19.456500 Sent 0x|11:00|72:00|80|00:00|AE:00:02:05:0F:81:3C:0F:01:A6:00:02|
12:21:19.657513 Sent 0x|09:00|73:00|00|01:00|A9:00:02:60|
12:21:19.722027 Recv 0x|04:00|73:00|02|01|
12:21:19.923390 Sent 0x|09:00|75:00|00|01:00|A9:00:02:60|
12:21:19.961935 Recv 0x|04:00|75:00|02|01|
12:21:20.162824 Sent 0x|09:00|76:00|00|01:00|A9:00:02:60|
12:21:20.234139 Recv 0x|04:00|76:00|02|01|
12:21:20.435034 Sent 0x|09:00|78:00|00|01:00|A9:00:02:60|
12:21:20.480964 Recv 0x|04:00|78:00|02|01|
12:21:20.681819 Sent 0x|09:00|79:00|00|01:00|A9:00:02:60|
12:21:20.735111 Recv 0x|04:00|79:00|02|00|
12:21:20.735661 Sent 0x|11:00|7A:00|80|00:00|AE:00:02:3B:0F:81:3C:0F:01:A6:00:02|
12:21:20.936434 Sent 0x|09:00|7D:00|00|01:00|A9:00:02:60|
12:21:20.985048 Recv 0x|04:00|7D:00|02|01|
12:21:21.185991 Sent 0x|09:00|7E:00|00|01:00|A9:00:02:60|
12:21:21.255167 Recv 0x|04:00|7E:00|02|01|
12:21:21.456068 Sent 0x|09:00|80:00|00|01:00|A9:00:02:60|
12:21:21.519136 Recv 0x|04:00|80:00|02|01|
12:21:21.720515 Sent 0x|09:00|82:00|00|01:00|A9:00:02:60|
12:21:21.780126 Recv 0x|04:00|82:00|02|01|
12:21:21.981291 Sent 0x|09:00|84:00|00|01:00|A9:00:02:60|
12:21:22.033996 Recv 0x|04:00|84:00|02|00|
12:21:22.235006 Sent 0x|09:00|86:00|80|00:00|A3:00:02:00|
```

Some remarks:

- Much more data traffic, but smooth and correct execution of movements, tones and LED lights.
- All these direct commands block the EV3 brick only for a very short timespan, short enough to be not recognized.
- As before, the message counters show gaps, where the direct commands of the song have been sent. But now, they were sent with a correct timing.

You can easily imagine, how adding some more motors or sensors will complicate the code. Therefore it's good practice to separate the tasks. Here the song has been separated as a thread task object and we didn't care about its internals.

Exact Movements as a Thread Task

We modify this program once more and create a thread task object for both, the motor movement and the song, which can be started and stopped. Encapsulating activities into thread task objects helps to code applications of more and more parallel actions.

Connect your EV3 medium motor with port B, connect your computer via Bluetooth with your EV3 brick, replace MAC-address with the one of your EV3 brick, then run this program:

```
import ev3_dc as ev3
import struct
from thread_task import Task, Periodic, Repeated, Sleep
from time import sleep


my_ev3 = ev3.EV3(protocol=ev3.BLUETOOTH, host='00:16:53:42:2B:99')
my_ev3.verbosity = 1

jukebox = ev3.Jukebox(ev3_obj=my_ev3)


def reset():
    my_ev3.send_direct_cmd(
        b''.join((
            ev3.opOutput_Reset,
            ev3.LCX(0),  # LAYER
            ev3.LCX(ev3.PORT_B)  # NOS
        )),
        sync_mode=ev3.SYNC
    )


def step_speed(speed: int):
    my_ev3.send_direct_cmd(
        b''.join((
            ev3.opOutput_Step_Speed,
            ev3.LCX(0),  # LAYER
            ev3.LCX(ev3.PORT_B),  # NOS
            ev3.LCX(speed),  # SPEED
            ev3.LCX(15),  # STEP1
            ev3.LCX(60),  # STEP2
            ev3.LCX(15),  # STEP3
            ev3.LCX(1),  # BRAKE - yes
            ev3.opOutput_Start,
```

(continues on next page)

(continued from previous page)

```
            ev3.LCX(0),  # LAYER
            ev3.LCX(ev3.PORT_B)  # NOS
        ))
    )


def test():
    ops = b''.join((
        ev3.opOutput_Test,
        ev3.LCX(0),  # LAYER
        ev3.LCX(ev3.PORT_B),  # NOS
        ev3.GVX(0)  # BUSY
    ))
    reply = my_ev3.send_direct_cmd(ops, global_mem=1)
    busy = struct.unpack('<b', reply)[0]
    return False if busy else True


def stop():
    my_ev3.send_direct_cmd(
        b''.join((
            ev3.opOutput_Stop,
            ev3.LCX(0),  # LAYER
            ev3.LCX(ev3.PORT_B),  # NOS
            ev3.LCX(0)  # BRAKE - no
        ))
    )


speed = 5

t_song = jukebox.song(ev3.FRERE_JACQUES)

t_forwards = (
    Task(step_speed, args=(speed,), duration=.2) +
    Periodic(.2, test)
)
t_forwards.action_stop = stop

t_backwards = (
    Task(step_speed, args=(-speed,), duration=.2) +
    Periodic(.2, test)
)

t = (
    Task(t_song.start) +
    Task(reset) +
    Repeated(
        t_forwards + t_backwards,
        num=5
    ) +
    Sleep(.2) +
    Task(stop)
)

t.start()
```

(continues on next page)

(continued from previous page)

```
sleep(8)
t.stop()
```

Some remarks:

- periodic ends, when its action returns True. This is why function `test()` returns the opposite of the expected.
- Nearly all of the program is about creating t as a thread task object. Its execution is only the few lines at the end. You can easily imagine to hide the creation behind the public API of a class.
- The parallel execution of motor movements and playing a song is handled inside of t.
- Stopping is quite easy. The logic, how to stop the activities is hidden insite the thread task.
- This thread task is not perfect because its continuation logic is not proper coded.

The output:

```
12:48:40.569302 Sent 0x|08:00|2A:00|00|00:00|A2:00:02|
12:48:40.648679 Recv 0x|03:00|2A:00|02|
12:48:40.649948 Sent 0x|11:00|2D:00|80|00:00|AE:00:02:05:0F:81:3C:0F:01:A6:00:02|
12:48:40.849774 Sent 0x|09:00|2E:00|00|01:00|A9:00:02:60|
12:48:40.896519 Recv 0x|04:00|2E:00|02|01|
12:48:41.050036 Sent 0x|09:00|2F:00|00|01:00|A9:00:02:60|
12:48:41.098628 Recv 0x|04:00|2F:00|02|01|
12:48:41.250406 Sent 0x|09:00|31:00|00|01:00|A9:00:02:60|
12:48:41.318686 Recv 0x|04:00|31:00|02|01|
12:48:41.450778 Sent 0x|09:00|32:00|00|01:00|A9:00:02:60|
12:48:41.494671 Recv 0x|04:00|32:00|02|01|
12:48:41.651188 Sent 0x|09:00|33:00|00|01:00|A9:00:02:60|
12:48:41.703649 Recv 0x|04:00|33:00|02|01|
12:48:41.851603 Sent 0x|09:00|35:00|00|01:00|A9:00:02:60|
12:48:41.943683 Recv 0x|04:00|35:00|02|00|
12:48:41.944486 Sent 0x|11:00|36:00|80|00:00|AE:00:02:3B:0F:81:3C:0F:01:A6:00:02|
12:48:42.144719 Sent 0x|09:00|37:00|00|01:00|A9:00:02:60|
12:48:42.211513 Recv 0x|04:00|37:00|02|01|
12:48:42.344999 Sent 0x|09:00|38:00|00|01:00|A9:00:02:60|
12:48:42.385481 Recv 0x|04:00|38:00|02|01|
12:48:42.545455 Sent 0x|09:00|3A:00|00|01:00|A9:00:02:60|
12:48:42.598677 Recv 0x|04:00|3A:00|02|01|
12:48:42.745691 Sent 0x|09:00|3B:00|00|01:00|A9:00:02:60|
12:48:42.799659 Recv 0x|04:00|3B:00|02|01|
12:48:42.946066 Sent 0x|09:00|3C:00|00|01:00|A9:00:02:60|
12:48:43.031706 Recv 0x|04:00|3C:00|02|01|
12:48:43.146600 Sent 0x|09:00|3F:00|00|01:00|A9:00:02:60|
12:48:43.207665 Recv 0x|04:00|3F:00|02|00|
12:48:43.208658 Sent 0x|11:00|40:00|80|00:00|AE:00:02:05:0F:81:3C:0F:01:A6:00:02|
12:48:43.409048 Sent 0x|09:00|41:00|00|01:00|A9:00:02:60|
12:48:43.482677 Recv 0x|04:00|41:00|02|01|
12:48:43.609350 Sent 0x|09:00|43:00|00|01:00|A9:00:02:60|
12:48:43.659614 Recv 0x|04:00|43:00|02|01|
12:48:43.809783 Sent 0x|09:00|44:00|00|01:00|A9:00:02:60|
12:48:43.867547 Recv 0x|04:00|44:00|02|01|
12:48:44.009999 Sent 0x|09:00|45:00|00|01:00|A9:00:02:60|
12:48:44.067605 Recv 0x|04:00|45:00|02|01|
12:48:44.210310 Sent 0x|09:00|47:00|00|01:00|A9:00:02:60|
12:48:44.295689 Recv 0x|04:00|47:00|02|01|
12:48:44.410610 Sent 0x|09:00|48:00|00|01:00|A9:00:02:60|
12:48:44.472626 Recv 0x|04:00|48:00|02|00|
```

(continues on next page)

(continued from previous page)

```
12:48:44.473215 Sent 0x|11:00|49:00|80|00:00|AE:00:02:3B:0F:81:3C:0F:01:A6:00:02|
12:48:44.673718 Sent 0x|09:00|4A:00|00|01:00|A9:00:02:60|
12:48:44.773517 Recv 0x|04:00|4A:00|02|01|
12:48:44.874162 Sent 0x|09:00|4C:00|00|01:00|A9:00:02:60|
12:48:44.950694 Recv 0x|04:00|4C:00|02|01|
12:48:45.074450 Sent 0x|09:00|4D:00|00|01:00|A9:00:02:60|
12:48:45.124658 Recv 0x|04:00|4D:00|02|01|
12:48:45.274793 Sent 0x|09:00|4E:00|00|01:00|A9:00:02:60|
12:48:45.322584 Recv 0x|04:00|4E:00|02|01|
12:48:45.475099 Sent 0x|09:00|51:00|00|01:00|A9:00:02:60|
12:48:45.528592 Recv 0x|04:00|51:00|02|01|
12:48:45.675549 Sent 0x|09:00|52:00|00|01:00|A9:00:02:60|
12:48:45.732762 Recv 0x|04:00|52:00|02|00|
12:48:45.733656 Sent 0x|11:00|53:00|80|00:00|AE:00:02:05:0F:81:3C:0F:01:A6:00:02|
12:48:45.934072 Sent 0x|09:00|54:00|00|01:00|A9:00:02:60|
12:48:45.985610 Recv 0x|04:00|54:00|02|01|
12:48:46.134651 Sent 0x|09:00|56:00|00|01:00|A9:00:02:60|
12:48:46.183608 Recv 0x|04:00|56:00|02|01|
12:48:46.334950 Sent 0x|09:00|57:00|00|01:00|A9:00:02:60|
12:48:46.399693 Recv 0x|04:00|57:00|02|01|
12:48:46.535293 Sent 0x|09:00|58:00|00|01:00|A9:00:02:60|
12:48:46.579582 Recv 0x|04:00|58:00|02|01|
12:48:46.735896 Sent 0x|09:00|5A:00|00|01:00|A9:00:02:60|
12:48:46.788654 Recv 0x|04:00|5A:00|02|01|
12:48:46.936190 Sent 0x|09:00|5B:00|00|01:00|A9:00:02:60|
12:48:46.992702 Recv 0x|04:00|5B:00|02|00|
12:48:46.993371 Sent 0x|11:00|5C:00|80|00:00|AE:00:02:3B:0F:81:3C:0F:01:A6:00:02|
12:48:47.193783 Sent 0x|09:00|5D:00|00|01:00|A9:00:02:60|
12:48:47.263712 Recv 0x|04:00|5D:00|02|01|
12:48:47.394209 Sent 0x|09:00|5E:00|00|01:00|A9:00:02:60|
12:48:47.439528 Recv 0x|04:00|5E:00|02|01|
12:48:47.594571 Sent 0x|09:00|5F:00|00|01:00|A9:00:02:60|
12:48:47.652589 Recv 0x|04:00|5F:00|02|01|
12:48:47.794863 Sent 0x|09:00|62:00|00|01:00|A9:00:02:60|
12:48:47.874742 Recv 0x|04:00|62:00|02|01|
12:48:47.995327 Sent 0x|09:00|63:00|00|01:00|A9:00:02:60|
12:48:48.067742 Recv 0x|04:00|63:00|02|01|
12:48:48.195556 Sent 0x|09:00|64:00|00|01:00|A9:00:02:60|
12:48:48.242525 Recv 0x|04:00|64:00|02|00|
12:48:48.243357 Sent 0x|11:00|65:00|80|00:00|AE:00:02:05:0F:81:3C:0F:01:A6:00:02|
12:48:48.443723 Sent 0x|09:00|67:00|00|01:00|A9:00:02:60|
12:48:48.498720 Recv 0x|04:00|67:00|02|01|
12:48:48.578092 Sent 0x|09:00|6A:00|80|00:00|A3:00:02:00|
```

Some remarks:

- Until the interruption, the direct commands were the same as before.
- The stopping occured during the seventh movement.
- The last direct command stopped the motor. This is what *t_forwards.action_stop* = *stop* meant.

Moving a motor to a Specified Position

Connect your EV3 medium motor with port B, connect your computer and your EV3 brick with an USB cable, replace MAC-address `00:16:53:42:2B:99` with the one of your EV3 brick, then run this program:

```
import ev3_dc as ev3
import struct
from math import copysign


my_ev3 = ev3.EV3(
    protocol=ev3.USB,
    host='00:16:53:42:2B:99'
)
my_ev3.verbosity = 1

speed = 10
to_position = 90
port = ev3.PORT_B
brake = 0

ops1 = b''.join((
    ev3.opInput_Device,
    ev3.READY_SI,
    ev3.LCX(0),  # LAYER
    ev3.port_motor_input(port),  # NO
    ev3.LCX(8),  # TYPE (EV3-Medium-Motor)
    ev3.LCX(0),  # MODE (Degree)
    ev3.LCX(1),  # VALUES
    ev3.GVX(0)  # VALUE1
))
reply = my_ev3.send_direct_cmd(ops1, global_mem=4)
from_position = struct.unpack('<f', reply)[0]

diff = to_position - round(from_position)
speed *= round(copysign(1, diff))
steps = abs(diff)

ops2 = b''.join((
    ev3.opOutput_Reset,
    ev3.LCX(0),  # LAYER
    ev3.LCX(port),  # NOS

    ev3.opOutput_Step_Speed,
    ev3.LCX(0),  # LAYER
    ev3.LCX(port),  # NOS
    ev3.LCX(speed),  # SPEED
    ev3.LCX(0),  # STEP1
    ev3.LCX(steps),  # STEP2
    ev3.LCX(0),  # STEP3
    ev3.LCX(brake),  # BRAKE - 1 (yes) or 0 (no)

    ev3.opOutput_Start,
    ev3.LCX(0),  # LAYER
    ev3.LCX(port)  # NOS
))
my_ev3.send_direct_cmd(ops2)
```

Please move the motor by hand and then run the program again. The motor will return to the defined position of 90 degrees. We use 4 already known operations and it's obvious, that this algorithm can easily be encapsulated into a method of a motor class.

The output:

```
13:19:05.149392 Sent 0x|0D:00|2A:00|00|04:00|99:1D:00:11:07:00:01:60|
13:19:05.155311 Recv 0x|07:00|2A:00|02|00:00:04:C2|
13:19:05.155969 Sent␣
↪0x|14:00|2B:00|00|00:00|A2:00:02:AE:00:02:0A:00:81:7B:00:00:A6:00:02|
13:19:05.161331 Recv 0x|03:00|2B:00|02|
```

Direct Commands are Machine Code Programs

There are operations for calculations and much more. Direct commands are little machine code programs. Let's write a single direct command, that does the same thing.

```python
import ev3_dc as ev3


my_ev3 = ev3.EV3(
    protocol=ev3.USB,
    host='00:16:53:42:2B:99'
)
my_ev3.verbosity = 1

speed = 10
to_position = 90
port = ev3.PORT_B
brake = 0

ops = b''.join((
    ev3.opInput_Device,
    ev3.READY_SI,
    ev3.LCX(0),  # LAYER
    ev3.port_motor_input(port),  # NO
    ev3.LCX(8),  # TYPE (EV3-Medium-Motor)
    ev3.LCX(0),  # MODE (Degree)
    ev3.LCX(1),  # VALUES
    ev3.LVX(0),  # VALUE1 - from_position (DATAF)

    ev3.opMove32_F,
    ev3.LCX(to_position),  # SOURCE
    ev3.LVX(4),  # DESTINATION - to_position (DATAF)

    ev3.opSubF,
    ev3.LVX(4),  # SOURCE1 - to_position (DATAF)
    ev3.LVX(0),  # SOURCE2 - from_position (DATAF)
    ev3.LVX(0),  # DESTINATION - diff (DATAF)

    ev3.opMath,
    ev3.ABS,  # CMD
    ev3.LVX(0),  # DATA X - diff (DATAF)
    ev3.LVX(4),  # RESULT - abs(diff) (DATAF)

    ev3.opDivF,
    ev3.LVX(0),  # SOURCE1 - diff (DATAF)
    ev3.LVX(4),  # SOURCE2 - abs(diff) (DATAF)
    ev3.LVX(0),  # DESTINATION - sign of diff (DATAF)

    ev3.opMove32_F,
```

(continues on next page)

(continued from previous page)

```
    ev3.LCX(speed),  # SOURCE
    ev3.LVX(8),  # DESTINATION - speed (DATAF)

    ev3.opMulF,
    ev3.LVX(0),  # SOURCE1 - sign of diff (DATAF)
    ev3.LVX(8),  # SOURCE2 - speed (DATAF)
    ev3.LVX(0),  # DESTINATION - signed_speed (DATAF)

    ev3.opMoveF_32,
    ev3.LVX(4),  # SOURCE - abs(diff) (DATAF)
    ev3.LVX(4),  # DESTINATION - abs(diff) (DATA32)

    ev3.opMoveF_8,
    ev3.LVX(0),  # SOURCE - signed_speed (DATAF)
    ev3.LVX(0),  # DESTINATION - signed_speed (DATA8)

    ev3.opOutput_Reset,
    ev3.LCX(0),  # LAYER
    ev3.LCX(port),  # NOS

    ev3.opOutput_Step_Speed,
    ev3.LCX(0),  # LAYER
    ev3.LCX(port),  # NOS
    ev3.LVX(0),  # SPEED - signed_speed (DATA8)
    ev3.LCX(0),  # STEP1
    ev3.LVX(4),  # STEP2 - abs(diff) (DATA32)
    ev3.LCX(0),  # STEP3
    ev3.LCX(brake),  # BRAKE - 1 (yes) or 0 (no)

    ev3.opOutput_Start,
    ev3.LCX(0),  # LAYER
    ev3.LCX(port)  # NOS
))
my_ev3.send_direct_cmd(ops, local_mem=12)
```

Some remarks:

- This direct command allocates 12 bytes of local memory for its intermediate results. Most of these are 4-bytes-numbers, therefore the referenced addresses are LVX(0), LVX(4) and LVX(8).
- We need to be carefull with the data formats, here we use numbers in three formats:
  - *DATA8* (1 byte integer),
  - *DATA32* (4 bytes integer) and
  - *DATAF* (4 bytes floating point).
- We have to translate some of the formats:
  - *opMove32_F* translates a 4 bytes integer into a floating point number,
  - *opMoveF_32* does the opposite,
  - *opMoveF_8* translates a floating point number into a 1 byte integer.
- We do the calculations with floating point numbers and use:
  - *opDivF* for division,
  - *opMulF* for multiplication and

  - *opMath* with CMD *ABS* to get the absolute value of a floating point number.

That's machine code, welcome to the sixties! Think a minute about coding complex algorithms this way and realize what the apollo program meant for the software developers in these times. But keep in mind, coding machine code is great for performance. Here the communication is reduced from 2 direct commands to one. In case of protocol *USB*, this means some 0.05 sec.

2.2 Touch

`Touch` is a subclass of `EV3`. You can use it to read values from a single touch sensor without any knowledge of direct command syntax.

To use multiple touch sensors, you can create multiple instances of class Touch.

2.2.1 Asking for the current state

Property `touched` is of type bool and tells, if the sensor currently is touched.

Connect your EV3 device with your local network via WiFi and make sure, it's the only EV3 devices in the network. Connect a touch sensor (it may be an EV3-Touch or a NXT-Touch) with PORT 1, then start this program.

```
import ev3_dc as ev3

with ev3.Touch(ev3.PORT_1, protocol=ev3.WIFI) as my_touch:
    print(str(my_touch) + ':', end=' ')
    print('touched' if my_touch.touched else 'not touched')
```

Some remarks:

- You already know, how to modify the program, when using protocols Bluetooth or USB.
- As the output line shows, the class knows it's sensor type.
- Run the program multiple times with touched and not touched sensor.
- Test what happens, when no sensor is connected to PORT 1.
- Test what happens, when another sensor type is connected to PORT 1.
- Switch on verbosity and you will see the communication data.

2.2.2 Multiple instances of class Touch

Connect an additional touch sensor (again it may be an EV3-Touch or a NXT-Touch) with PORT 4, then start this program.

```
import ev3_dc as ev3

with ev3.Touch(ev3.PORT_1, protocol=ev3.WIFI) as touch_left:
    touch_right = ev3.Touch(ev3.PORT_4, ev3_obj=touch_left)
    print(str(touch_left) + ':', end=' ')
    print('touched' if touch_left.touched else 'not touched')
    print(str(touch_right) + ':', end=' ')
    print('touched' if touch_right.touched else 'not touched')
```

Some remarks:

- Both touch sensors share the same EV3 device. Therefore only the first instance is initialized with keyword argument *protocol*. The second instance is initialized with keyword argument *ev3_obj* instead.
- *touch_left* owns the connection, *touch_right* is its joint user.
- A single EV3 device controls up to four sensors and additionally up to four motors. You will deal with more than two objects, when you make use of EV3's full capacity.
- Both sensors are handled independently, therefore the communication is not optimized. The request of both sensors' state could have been done in a single direct command, but here it needs two instead.

2.2.3 Bump-Mode

Touch sensors provide two modes, touch and bump (see sections *Touch mode of the Touch Sensor* and *Bump mode of the Touch Sensor*). The touch-mode is, what we have seen above: the sensor replies it's current state. The bump-mode counts the number of bumps since the last sensor clearing.

Connect your EV3 device with your local network via WiFi. Replace the MAC-address by the one of your EV3 brick, connect a touch sensor (it may be an EV3-Touch or a NXT-Touch) with PORT 1, then start this program.

```
import ev3_dc as ev3
from time import sleep

my_touch = ev3.Touch(
        ev3.PORT_1,
        protocol=ev3.WIFI,
        host='00:16:53:42:2B:99'
)

print('\n' + 'countdown ...' + '\n')
for n in range(10, 0, -1):
    print('\r' + f'{n:2d} ', end='', flush=True)
    sleep(1)

print('\r' + '** go ** ', end='', flush=True)

my_touch.bumps = 0
sleep(5)

print('\r' + 'number of bumps:', my_touch.bumps)
```

Some remarks:

- This program counts the number of bumps for a timespan of 5 sec.
- To prevent jumping the start, the sensor clearing is done at the end of the countdown.
- Instead of setting property *bumps = 0*, you alternatively can call method `clear()`.
- Compare the version above with the manually coded direct commands from section *Bump mode of the Touch Sensor* and you will realize the handiness of sensor classes.

2.3 Infrared

Class `Infrared` is a subclass of `EV3`. You can use it to read values from a single infrared sensor without any knowledge of direct command syntax.

The infrared sensor sends and receives infrared light signals. It is able to calculate distances by analyzing reflected light. It also is able to communicate with the EV3 beacon device. This allows to determine the current position of the beacon and it allows to use the bacon as a remote control.

To use multiple infrared sensors simultaneously, you can create multiple instances of this class.

2.3.1 Asking for the distance from a surface

Class `Infrared` has an attribute `distance`, which is of type float and tells, if the the sensor currently *sees* some surface in front of the sensor and in a distance closer than 1.00 m.

Connect your EV3 device with your local network via WiFi. Replace the MAC-address by the one of your EV3 brick, connect an infrared sensor with PORT 2, then start this program.

```
import ev3_dc as ev3

with ev3.Infrared(
        ev3.PORT_2,
        protocol=ev3.WIFI,
        host='00:16:53:42:2B:99'
) as my_infrared:
    dist = my_infrared.distance
    if dist:
        print(f'distance: {dist:3.2f} m')
    else:
        print('seen nothing')
```

Some remarks:

- You already know, how to change the program for using protocols Bluetooth or USB.
- Run the program multiple times with different surfaces and distances.
- Test what happens, when no sensor is connected to PORT 2.
- Test what happens, when another sensor type is connected to PORT 2.
- Test with distances larger than 1.00 m.
- Every time, you refrence attribute `distance`, you again start a communication between your program and the EV3 device.
- Switch on verbosity by setting attribute `verbosity` to value 1 and you will see the communication data.

2.3.2 Asking for a beacon's position

Class `Infrared` has an attribute `beacon`, which returns a named tuple of type *Beacon*. It tells, if the sensor currently *sees* an active beacon, which is sending on the requested channel.

Connect your EV3 device with your local network via WiFi. Replace the MAC-address by the one of your EV3 brick. Connect an infrared sensor with PORT 2, place a beacon somewhere in front of the sensor, select channel 3 and switch on the beacon, then start this program.

```
import ev3_dc as ev3

with ev3.Infrared(
        ev3.PORT_2,
        channel=3,
```

(continues on next page)

(continued from previous page)

```
        protocol=ev3.WIFI,
        host='00:16:53:42:2B:99'
) as my_infrared:
    print(my_infrared)
    print(f'beacon on channel {my_infrared.channel}: {my_infrared.beacon}')
```

Some remarks:

- If you prefer protocols Bluetooth or USB, you know how to change the program.
- The named tuple *Beacon* has two items, *heading* and *distance*, where *heading* is between -25 and 25, and *distance* is in meters.
- The meaning of the *heading* values:
  - -25: far left
  - 0: straight forwards
  - 25: far right

The output of my program was:

The beacon was positioned left ahead in a distance of 23 cm.

2.3.3 Using up to four beacons

If you need to identify the exact orientation and position of your EV3 device, you can use multiple beacons. Because they send on four different channels, you can simultaneously up to four of them. Attribute `beacons` allows to ask for their positions at once.

As before, connect your EV3 device with your local network via WiFi. Replace the MAC-address by the one of your EV3 brick. Connect an infrared sensor with PORT 2, place up to four beacons somewhere in front of the sensor, select different channels and switch on the beacons, then start this program.

```
import ev3_dc as ev3

with ev3.Infrared(
        ev3.PORT_2,
        protocol=ev3.WIFI,
        host='00:16:53:42:2B:99'
) as my_infrared:
    print(f'beacons: {my_infrared.beacons}')
```

The output of my program run:

```
beacons: (None, Beacon(heading=5, distance=0.32), None, None)
```

Some remarks:

- This was a single beacon, sending on channel 2, which was positioned right ahead in a distance of 32 cm.
- The returned data is a tuple of four items, one per channel.
- If no beacon was found, the channel's item is set to *None*.
- If a beacon was found, the channel's item is of type *Beacon*.

2.3.4 Using the beacon as a remote control

Class `Infrared` has an attribute `remote`, which returns a named tuple of type *Remote*. It tells, which of the beacon's buttons currently were pushed.

Connect your EV3 device with your local network via WiFi. Replace the MAC-address by the one of your EV3 brick. Connect an infrared sensor with PORT 2, place a beacon somewhere in front of the sensor, select channel 3 and switch on the beacon, then start this program.

```python
import ev3_dc as ev3
from time import sleep

with ev3.Infrared(
        ev3.PORT_2,
        channel=3,
        protocol=ev3.WIFI,
        host='00:16:53:42:2B:99'
) as my_infrared:
    while True:
        remote_state = my_infrared.remote
        if remote_state is not None:
            break
        sleep(0.1)

    print(f'state of the remote on channel {my_infrared.channel}: {remote_state}')
```

Some remarks:

- Every 100 ms, the state of the remote is requested, which means request and reply communication between program and EV3 device ten times per second.
- The state of the remote control is stored in variable *remote_state*. This allows to use it to end the loop as well as for the printing.
- You will easily imagine, how to define different actions for different states of the remote data.

The output of my program's execution:

```
state of the remote on channel 3: Remote(permanent=False, red_up=False, red_down=True,
↪ blue_up=True, blue_down=False)
```

This says, someone pushed two of the buttons simultaneously. The communication does not handle triple pushes and double pushes are restricted to the buttons *red_up*, *red_down*, *blue_up* and *blue_down*. Altogether, we can distinguish 11 different states plus none pushes.

2.3.5 Reading multiple remote control channels simultaneously

If you try to use multiple beacons simultaneously as remote controls, you can do that with attribute `remotes`, which returns a tuple of four items, one per channel. As you will have expected, each of them may be *None* or of type *Remote*.

As before, connect your EV3 device with your local network via WiFi. Replace the MAC-address by the one of your EV3 brick. Connect an infrared sensor with PORT 2, then start the program. After some time push any button of a beacon.

```python
import ev3_dc as ev3
import time

```

(continues on next page)

(continued from previous page)

```
with ev3.Infrared(
        ev3.PORT_2,
        protocol=ev3.WIFI,
        host='00:16:53:42:2B:99'
) as my_infrared:
    print(f'started at {time.strftime("%H:%M:%S", time.localtime())}')

    def any_remote():
        for remote in my_infrared.remotes:
            if remote:
                return remote

    while True:
        the_active_one = any_remote()
        if the_active_one:
            break
        time.sleep(0.1)

    print(the_active_one)
    print(f'stopped at {time.strftime("%H:%M:%S", time.localtime())}')
```

The output of my program's execution:

```
started at 18:32:01
Remote(permanent=False, red_up=False, red_down=True, blue_up=True, blue_down=False)
stopped at 18:32:09
```

Some remarks:

- Eight seconds after the program's start, someone simultaneously pressed two buttons of a beacon. These buttons were *red_down* and *blue_up*.
- This program does not care about channels. Function *any_remote* loops over all four channels and if it finds one unequal *None*, this one is returned.
- May be, your program only supports one beacon as a remote control but you do not trust the user to select the correct channel. This may be the solution: you read all four channels and then select the correct one.
- May be your program is thought for multiple users and every user has his own beacon. Then any of them can end the program.

2.4 Ultrasonic

Class `Ultrasonic` is a subclass of *EV3*. You can use it to read values from a single ultrasonic sensor without any knowledge of direct command syntax.

The ultrasonic sensor sends and receives ultrasonic sound signals. It is able to calculate distances by analyzing reflected sound. This is a subset of the infrared sensors functionality,

The ultrasonic sensor returns a distance of 2.55 m, when it does not detect anything. Class *Ultrasonic* replaces this by value *None*.

If you like to use multiple ultrasonic sensors simultaneously, you can create more than one instance of this class.

2.4.1 Asking for the distance from a surface

Class `Ultrasonic` has an attribute `distance`, which is of type float and tells, if the the sensor currently *sees* some surface in front and in a distance closer than 2.55 m.

Take an USB cable and connect your EV3 device with your computer. Replace MAC-address `00:16:53:42:2B:99` by the one of your EV3 brick, connect an ultrasonic sensor (it may be of type ev3.NXT_ULTRASONIC or ev3.EV3_ULTRASONIC) with PORT 3, then start this program:

```
from time import sleep
import ev3_dc as ev3

my_ultrasonic = ev3.Ultrasonic(
        ev3.PORT_3,
        protocol=ev3.USB,
        host='00:16:53:42:2B:99'
)

while True:
    dist = my_ultrasonic.distance
    if dist:
        break
    sleep(0.1)

print(f'something seen {dist:3.2f} m ahead')
```

Some remarks:

- You already know, how to change the program for using protocols Bluetooth or WiFi.
- Run the program multiple times with different surfaces and distances.
- Test what happens, when no sensor is connected to PORT 2.
- Test what happens, when another sensor type is connected to PORT 2.
- Test for the maximum distance and determine if this depends on the surface material.
- Every reference of property *distance* starts a new communication between the program and the EV3 device.
- Switch on verbosity by setting attribute `verbosity` to value 1 and you will see the communication data.

2.5 Color

Class `Color` is a subclass of `EV3`. You can use it to read values from a single color sensor without any knowledge of direct command syntax. The color sensor measures light intensity or colors. This may be reflected light. If you like to use multiple color sensors simultaneously, then create more than one instance of this class.

2.5.1 The reflected intensity of red light

Class `Color` has an attribute `reflected`, which is of type int and tells the intensity of the reflected red light in percent. This says: it switches on red light and then measures the reflection from a surface. From then on the red light shines permanently. With constant surface type and color, this allows to measure small distances (this needs calibration). Alternatively, when the distance is constant, it allows to distinguish dark from bright surfaces (e.g. for line followers). Because of the red light, it tends to categorize red surfaces as bright and green surfaces as dark.

Take an USB cable and connect your EV3 device with your computer. Replace MAC-address `00:16:53:42:2B:99` by the one of your EV3 brick, connect a color sensor (it may be of type ev3.NXT_COLOR or ev3.EV3_COLOR) with PORT 1, then start this program:

```
import ev3_dc as ev3

my_color = ev3.Color(
        ev3.PORT_1,
        protocol=ev3.USB,
        host='00:16:53:42:2B:99'
)

print(f'reflected intensity is {my_color.reflected} %')
```

Some remarks:

- You already know, how to change the program for using protocols Bluetooth or WiFi.
- Run the program multiple times with different surface colors and distances.
- Test what happens, when no sensor is connected to PORT 1.
- Test what happens, when another sensor type is connected to PORT 1.
- Every reference of property *reflected* starts a new communication between the program and the EV3 device.
- Switch on verbosity by setting attribute `verbosity` to value 1 and you will see the communication data.

My program's output was:

```
reflected intensity is 17 %
```

2.5.2 Recognize colors

Class *`Color`* has an attribute *`color`*, which is of type int and tells the color of the surface in front of the sensor. This ist done when the sensor shines white.

Take an USB cable and connect your EV3 device with your computer. Replace MAC-address `00:16:53:42:2B:99` by the one of your EV3 brick, connect a color sensor (it may be of type ev3.NXT_COLOR or ev3.EV3_COLOR) with PORT 1, then start this program:

```
import ev3_dc as ev3

my_color = ev3.Color(
        ev3.PORT_1,
        protocol=ev3.USB,
        host='00:16:53:42:2B:99'
)

color = (
    'none',
    'black',
    'blue',
    'green',
    'yellow',
    'red',
    'white',
    'brown'
```

(continues on next page)

(continued from previous page)

```
)[my_color.color]
print('the color is', color)
```

Some remarks:

- You already know, how to change the program for using protocols Bluetooth or WiFi.
- Run the program multiple times with different surface colors in front of the sensor.
- Test what happens, when no sensor is connected to PORT 1.
- Test what happens, when another sensor type is connected to PORT 1.
- Every reference of property *color* starts a new communication between the program and the EV3 device.
- Switch on verbosity by setting attribute `verbosity` to value 1 and you will see the communication data.
- The light emission is permanent. Therefore the sensor permanently switches on white light.
- NXT-Color does never answer with 0 or 7, it therefore will never see *none* or *brown*.
- You can use the constants *ev3.NONECOLOR*, *ev3.BLACKCOLOR*, etc. if your program asks for specific colors.

My program's output was:

```
the color is green
```

2.5.3 Red green blue Color Intensities

Class `Color` has an attribute `rgb_raw`, which is a tuple of three int type values and tells the color of the surface in front of the sensor. This ist done when the sensor shines with all three led colors on.

Take an USB cable and connect your EV3 device with your computer, connect a color sensor (it must be of type ev3.EV3_COLOR) with PORT 1, then start this program:

```
import ev3_dc as ev3

my_color = ev3.Color(
        ev3.PORT_1,
        protocol=ev3.USB
)
print(my_color.rgb_raw)
```

Some remarks:

- You already know, how to change the program for using protocols Bluetooth or WiFi.
- Run the program multiple times with different surface colors in front of the sensor. Also vary the distance from the surface.
- Be aware, that every reference of property *rgb_raw* starts a new communication between the program and the EV3 device.
- Switch on verbosity by setting attribute `verbosity` to value 1 and you will see the communication data.
- The light emission is permanent. Therefore the sensor permanently switches on all its LED light colors.

My program's output with a white surface in front of the sensor:

```
RawRGBColor(red=253, green=292, blue=183)
```

Some remarks:

- Maybe you did not expect to get values higher than 255. Keep in mind, that sensor ev3.EV3_COLOR is normed to values from 0 to 1024.
- The measurement is done with reflected light and consequently depends on the color spectrum of the light source. The blue part of the light source's frequencies is under-represented, the green part is over-represented and this is what you find in the result above.
- The result from the white surface can be used for a color balance, which is well known from photography or image processing.
- Geometry also plays its role in the measured intensities. The blue light source is closest to the sensor, the green one is most distant. As a consequence, smaller distances between sensor and surface result in more balanced results.

2.5.4 Balanced red green blue Color Intensities

Class `Color` has an attribute `rgb`, which is very similar to attribute `rgb_raw`, but is white balanced.

Take an USB cable and connect your EV3 device with your computer, connect a color sensor (it must be of type ev3.EV3_COLOR) with PORT 1, then start this program:

```
import ev3_dc as ev3

my_color = ev3.Color(ev3.PORT_1, protocol=ev3.USB)
my_color.rgb_white_balance = (253, 292, 183)
print(my_color.rgb)
```

Some remarks:

- This program uses the raw values, measured on a white surface to do the white balance.
- Replace the values for the white balance by the result of your own measurement.
- Run the program multiple times with different surface colors in front of the sensor. Also vary the distance from the surface.

My program's output with a green surface in front of the sensor:

```
RGBColor(red=43, green=114, blue=54)
```

Some remarks:

- Attribute `rgb` is normed to values between 0 and 255. This is what you know as rgb colors.
- Use a color picker, like this one to control your results.

2.5.5 Ambient light intensity

Class `Color` has an attribute `ambient`, which is of type int and tells the intensity of the ambient light in percent. One would expect, that this ist done without any light emission. Surprisingly the EV3_Color sensor switches on its blue light, when it measures ambient light. The NXT-Color sensor behaves as expected, it switches its light off.

Take an USB cable and connect your EV3 device with your computer. Replace MAC-address `00:16:53:42:2B:99` by the one of your EV3 brick, connect a color sensor (it may be of type ev3.NXT_COLOR or ev3.EV3_COLOR) with PORT 1, then start this program:

```
import ev3_dc as ev3

my_color = ev3.Color(
        ev3.PORT_1,
        protocol=ev3.USB,
        host='00:16:53:42:2B:99'
)

print(f'ambient intensity is {my_color.ambient} %')
```

Some remarks:

- You already know, how to change the program for using protocols Bluetooth or WiFi.
- Run the program multiple times with different light intensity in front of the sensor.
- Test what happens, when no sensor is connected to PORT 1.
- Test what happens, when another sensor type is connected to PORT 1.
- Every reference of property *ambient* starts a new communication between the program and the EV3 device.
- Switch on verbosity by setting attribute `verbosity` to value 1 and you will see the communication data.
- The light emission is permanent. Therefore EV3-Color permanently changes to blue light, NXT-Color permanently switches its light off.

My program's output was:

```
ambient intensity is 9 %
```

2.6 Gyro

Class *`Gyro`* is a subclass of *`EV3`*. You can use it to read values from a single LEGO gyro sensor (a gyroscope) without any knowledge of direct command syntax.

The gyro sensor is used to measure one-dimensional orientation and angular velocity. These attributes allow to get the measurements:

- *`angle`* measures the current orientation as an angle, which is an integer representing the sensor's clockwise rotation.
- *`rate`* is an integer, representing the sensor's clockwise rotation rate (or angular velocity) in degree per second.
- *`state`* holds both, the current angle and the current rotation rate. It is of type *GyroState*.

If you like to use multiple gyro sensors simultaneously (e.g. to receive rotations along multiple axes), you can create more than one instance of this class.

2.6.1 Asking for the current orientation angle

Choose settings (protocol and host) to connect the EV3 to your computer. Replace the settings in the program below, connect a gyro sensor to PORT_1, run this program and rotate the gyro sensor:

```
from time import sleep
import ev3_dc as ev3

settings = {"protocol":ev3.BLUETOOTH, "host":"00:16:53:81:D7:E2"}
```

(continues on next page)

(continued from previous page)

```
with ev3.Gyro(ev3.PORT_1, **settings) as gyro:
    while True:
        current_angle = gyro.angle
        print(f"\rThe current angle is {current_angle:4d} °", end='')
        if current_angle >= 360:
            print("\n" + "The sensor made a full clockwise turn!")
            break
        elif current_angle <= -360:
            print("\n" + "The sensor made a full counterclockwise turn!")
            break
        sleep(0.05)
```

Some remarks:

- Mathematically, clockwise rotation is measured with negative values, counterclockwise rotation with positive ones. LEGO's gyro sensor does not follow this convention! If you face the the red icon on its top, then clockwise rotation measures positive.
- In the moment, when class *Gyro* is initiated, the sensor's current rotation angle becomes value zero.
- What a LEGO gyro sensor measures is not really orientation. Instead it measures the orientation angle between an original orientation and the current one. If the sensor made multiple full rotations, then *angle* will correctly show it. Modify the program and break the loop when at least 2 full turns have been made.
- Every reference of property *angle* starts a new communication between the program and the EV3 device.
- Method *reset()* allows to reset the zero position at any other time or to set the current angle to any other value.
- Printing '\r' (carriage return) returns to the beginning of the current line. This allows to print the same line again and again.
- Switch on verbosity by setting attribute `verbosity` to value 1 and you will see the communication data. This will show you, that the measurements use mode 3, which is *EV3-Gyro-Rate & Angle* and get angle and rate as results.

2.6.2 Asking for the current rotation rate

Connect your EV3 device and your computer via USB cable, connect a gyro sensor to PORT_1, then run this program and rotate the gyro sensor:

```
from time import sleep
import ev3_dc as ev3

with ev3.Gyro(ev3.PORT_1, protocol=ev3.USB) as gyro:
    min_rate, max_rate = 0, 0
    print('for 10 sec. do some rotation movements')
    for i in range(100):
        cur_rate = gyro.rate
        min_rate = min(min_rate, cur_rate)
        max_rate = max(max_rate, cur_rate)
        sleep(0.1)
print(f'max. rate: {max_rate} °/s, min. rate: {min_rate} °/s')
```

Some remarks:

- Every reference of property *rate* starts a new communication between the program and the EV3 device. This is why we use variable *cur_rate* (current rate) to hold the values.

- Switch on verbosity by setting attribute `verbosity` to value 1 and you will see the communication data.

2.6.3 Asking for the current state (angle and rate)

Connect your EV3 device and your computer via USB cable, connect a gyro sensor to PORT_1, then run this program and rotate the gyro sensor:

```
import time import sleep
import ev3_dc as ev3

with ev3.Gyro(ev3.PORT_1, protocol=ev3.USB) as gyro:
    cs = gyro.state
    print(f'angle: {cs.angle:4d} °, rate: {cs.rate:4d} °/s', end='')
    for i in range(10):
        sleep(1)
        cs = gyro.state
        print('\r' + f'angle: {cs.angle:4d} °, rate: {cs.rate:4d} °/s', end='')
    print()
```

Some remarks:

- Every reference of property *state* starts a new communication between the program and the EV3 device. This is why we use variable *cs* (current state) to hold the values.
- Porperty *state* is of type *GyroState*, which has two attributes: *angle* and *rate*.
- Printing '\r' (carriage return) returns to the beginning of the current line. This allows to print the same line again and again.
- Switch on verbosity by setting attribute `verbosity` to value 1 and you will see the communication data.

2.6.4 Reset the original orientation

Sometimes the orientation in the moment of class initialization is not the best point of reference. E.g. an algorithm for a balancing device is clearer coded, when the perfect balance becomes the point of reference. Method *reset()* allows to do exactly that.

Connect your EV3 device and your computer via USB cable, connect a gyro sensor to PORT_1, then run this program and don't rotate the gyro sensor:

```
from time import sleep
import ev3_dc as ev3

with ev3.Gyro(ev3.PORT_1, protocol = ev3.USB) as gyro:
        print(f"The current angle is {gyro.angle} °")
        sleep(5)

        gyro.reset(angle=90)
        print(f"After resetting: The current angle is {gyro.angle} °")
        sleep(5)

        gyro.reset(angle=180)
        print(f"After another resetting: The current angle is {gyro.angle} °")
        sleep(5)

        gyro.reset()
        print(f"After resetting again: The current angle is {gyro.angle} °")
```

The output:

Some remarks:

- Run the program again and do some rotation movements of the sensor while the sleeping. You will see the very same output, why?
- Modify the program and do the sleeping between the resets and the measurements. Then start the program again and do some rotation movements of the sensor.

2.7 Sound

Sound is a subclass of *FileSystem*. It provides higher order methods to play tones and sound files.

2.7.1 Play a Tone

Method *tone()* of class *Sound* plays tones with given frequencies. Connect your EV3 brick and your computer with an USB cable, then start this program:

```
import ev3_dc as ev3

with ev3.Sound(protocol=ev3.USB) as sound:
    sound.verbosity = 1
    sound.tone(250, duration=1, volume=100)
```

This plays a tone with frequency 250 Hz for one second at maximum volume. If no duration is given, method *tone()* plays the tone unlimited. If no volume is given it takes the volume, which can be set as a property of class Sound. If neither was set, it takes the volume from the EV3 device. The frequency must be in a range of 250 - 10.000 Hz, the volume must be in a range of 1 - 100.

The output:

```
13:01:25.162280 Sent 0x|0F:00|2B:00|00|00:00|94:01:81:64:82:FA:00:82:E8:03|
13:01:25.168008 Recv 0x|03:00|2B:00|02|
```

2.7.2 Play a Sound File

Robot sound files are a special pulse code modulation format for audio signals. The single channel signal has a sample rate of 8 kHz and an 8 bit resolution. This blog describes, how to add the header information and create robot sound files. Some sound files can be found on the EV3 device. Method *play_sound()* allows to play these sound files.

Connect your EV3 brick and your computer via Bluetooth, replace the MAC-address by the one of your EV3 brick, then start this program:

```
import ev3_dc as ev3
from time import sleep

hugo = ev3.Sound(protocol=ev3.BLUETOOTH, host='00:16:53:42:2B:99', volume=100)
hugo.verbosity = 1

hugo.play_sound(
    './ui/DownloadSucces.rsf',
    repeat=True
)
```

(continues on next page)

(continued from previous page)

```
sleep(5)

hugo.stop_sound()
```

Some remarks:

- The program plays a sound file repeatedly and stops the sound after 5 sec. This is exactly, what program *Playing Sound Files repeatedly* does.
- The timing is done by the program.
- It needs to call method `stop_sound()` to stop the playing, otherwise it would last forever.

The output:

```
13:45:30.663648 Sent␣
↪0x|1E:00|2A:00|80|00:00|94:03:81:64:84:2E:2F:75:69:2F:44:6F:77:6E:6C:6F:61:64:53:75:63:63:65:73:00
13:45:35.669587 Sent 0x|07:00|2B:00|80|00:00|94:00|
```

2.7.3 Play a Sound File as a Thread Task

thread_task objects allow to define the timing beforehand, when the thread task is created. Starting thread tasks allows to do multiple things parallel.

Connect your EV3 brick and your computer via Bluetooth, replace the MAC-address by the one of your EV3 brick, then start this program:

```
import ev3_dc as ev3

hugo = ev3.Sound(protocol=ev3.BLUETOOTH, host='00:16:53:42:2B:99', volume=100)
hugo.verbosity = 1

t_sound = hugo.sound(
    './ui/DownloadSucces.rsf',
    duration=5,
    repeat=True
)

t_sound.start()
```

Some remarks:

- method `sound()` returns a thread task object, we name it *t_sound*.
- *t_sound* can be started, stopped, continued and restarted. We only start it.
- *t_sound* runs in the background. If you add some more commands to this program, you will realize, they are executed parallel to *t_sound*.
- the timing is done inside the thread task object.
- stopping the sound also is done by the task object.
- thread task objects encapsulate program logik behind a simple public API.

The output:

```
14:06:40.170520 Sent␣
↪0x|1E:00|2A:00|80|00:00|94:03:81:64:84:2E:2F:75:69:2F:44:6F:77:6E:6C:6F:61:64:53:75:63:63:65:73:00
14:06:45.170841 Sent 0x|07:00|2B:00|80|00:00|94:00|
```

2.7.4 Play a local Sound File

If you combine method `load_file()` from class `FileSystem()` with the above described functionality, you can also play local sound files.

Find the location of LEGO's sound files, which in my case was: .../Program Files (x86)/LEGO Software/LEGO MINDSTORMS EV3 Home Edition/Resources/BrickResources/Retail/Sounds/files (I copied this directory to a location with a shorter path). Modify the program by replacing the file location. Take an USB cable and connect your EV3 brick with your computer then start the following program.

```
import ev3_dc as ev3

with ev3.Sound(protocol=ev3.USB, volume=20) as hugo:
    hugo.sound(
        '../Sound/Expressions/Laughing 2.rsf',
        local=True
    ).start(thread=False)
print('all done')
```

Some remarks:

- keyword argument *local* makes the distinction between local sound files and sound files on the EV3 device. In this case, the sound file exists in the file system of the machine, which runs the program and the relative path is from the directory, where this python program is located.
- Starting a Thread Task with *thread=False* lets it behave traditional, it does its actions and your program continues with execution, when they are done.

2.8 Jukebox

`Jukebox` is a subclass of `EV3`. You can use it to play tones or to change the LED color without internal knowledge of direct commands.

But Jukebox is more than that. It combines direct commands with thread_task. This allows to use sounds and light effects parallel to other activities.

2.8.1 Change Color

Instead of coding a direct command, like we did in *Changing LED colors*, you can do the same thing a bit more comfortable.

Connect your EV3 brick and your computer via Bluetooth, replace the MAC-address by the one of your EV3 brick, then start this program

```
import ev3_dc as ev3
from time import sleep

jukebox = ev3.Jukebox(protocol=ev3.BLUETOOTH, host='00:16:53:42:2B:99')
jukebox.verbosity = 1
```

(continues on next page)

(continued from previous page)

```
jukebox.change_color(ev3.LED_RED_FLASH)

sleep(5)

jukebox.change_color(ev3.LED_GREEN)
```

For five seconds, the LED will flah red, then it will become green again. Calling method `change_color()` hides the technical details behind a more user friendly API. We set the verbosity because we like to see the communication between our program an the EV3 brick. The output:

```
13:09:05.718859 Sent 0x|08:00|2A:00|80|00:00|82:1B:05|
13:09:10.724762 Sent 0x|08:00|2B:00|80|00:00|82:1B:01|
```

Obviously, `Jukebox` sends direct commands.

2.8.2 Play Tone

Playing tones is the other competence of class `Jukebox`. Different from method `tone()` of class `Sound`, these tones are defined by their musical names and not by their frequencies. Method `play_tone()` allows to name them *c*, *d*, *e*, *c'*, *d'* or *e'*. One does not need to know their frequencies.

Connect your EV3 brick and your computer via Bluetooth, replace the MAC-address by the one of your EV3 brick, then start this program

```
import ev3_dc as ev3

jukebox = ev3.Jukebox(protocol=ev3.BLUETOOTH, host='00:16:53:42:2B:99')
jukebox.verbosity = 1

jukebox.play_tone("f'''", duration=1, volume=100)
```

This plays f^3 for one second at maximum volume (f^3 is the highest tone of Mozart's *Queen of the night aria*). If no duration is given, method `play_tone()` plays the tone unlimited. If no volume is given, it takes the volume, which was set as an optional argument of class Jukebox's creation. If neither was set, it takes the volume from the device.

The output:

```
13:13:49.071839 Sent 0x|0F:00|2A:00|80|00:00|94:01:81:64:82:75:05:82:E8:03|
```

2.8.3 Playing the EU-Antemn

Connect your EV3 brick and your computer via Bluetooth, replace the MAC-address by the one of your EV3 brick, then start this program:

```
import ev3_dc as ev3

with ev3.Jukebox(protocol=ev3.BLUETOOTH, host='00:16:53:42:2B:99') as jukebox:
    jukebox.song(ev3.EU_ANTEMN).start()
```

Some remarks:

- Method `song()` returns a thread_task.Task object, which can be started, stopped and continued. It plays tones and changes the LED-colors.

- Starting the thread task does not block the program nor does it block the EV3 brick. It runs in the background and allows to do additional things parallel.

EU_ANTEMN is a dictionary:

```
EU_ANTEMN = {
    "tempo": 100,
    "beats_per_bar": 4,
    "led_sequence": (
        LED_ORANGE,
        LED_GREEN,
        LED_RED,
        LED_GREEN
    ),
    "tones": (
        ("a'", 1),
        ("a'", 1),
        ("bb'", 1),
        ("c''", 1),

...

        ("g'", 1.5),
        ("f'", .5),
        ("f'", 1)
    )
}
```

Some remarks:

- *tempo* is beats per minute.
- *led_sequence* is the color sequence, which changes per bar.
- *tones* are the tones to play, the duration is not in seconds, but in beats.

2.8.4 Combine Happy Birthday with the Triad

Connect your EV3 brick and your computer via Bluetooth, replace the MAC-address by the one of your EV3 brick, then start this program:

```
import ev3_dc as ev3
from thread_task import Sleep

with ev3.Jukebox(protocol=ev3.BLUETOOTH, host='00:16:53:42:2B:99') as jukebox:
    (
        jukebox.song(ev3.TRIAD) +
        Sleep(1) +
        jukebox.song(ev3.HAPPY_BIRTHDAY) +
        Sleep(1) +
        jukebox.song(ev3.TRIAD)
    ).start()
```

The program builds a chain of tasks, which also is a thread_task object. It demonstrates how to build tasks of growing complexity, which still keep their simple public API.

2.8.5 Singing Canon with an EV3 brick

Connect your EV3 brick and your computer via Bluetooth, replace the MAC-address by the one of your EV3 brick, then start this program:

```
import ev3_dc as ev3
from thread_task import Repeated

with ev3.Jukebox(protocol=ev3.BLUETOOTH, host='00:16:53:42:2B:99') as jukebox:
    Repeated(
        jukebox.song(ev3.FRERE_JACQUES),
        num=3
    ).start()
```

Class Repeated plays the canon three times.

2.9 Voice

Voice is a subclass of *Sound*. You can use it to get your EV3 device speaking. This is done by supportting text to speech. Method *speak()* generates robot sound files (rsf) from text strings, copies them to the EV3 device and plays them.

2.9.1 Get your EV3 Device Speaking

Take an USB cable and connect your EV3 brick with your computer, then start this program:

```
import ev3_dc as ev3

with ev3.Voice(protocol=ev3.USB, lang='it') as voice:
    (
        voice.speak(
            '''
            Bona sera, cara Francesca! Come stai?
            Non vedo l'ora di venire in Italia.
            Stasera è una bella serata.
            '''
        ) + voice.speak(
            '''
            Hello Brian,
            this is your LEGO EV3 device.
            I speak english and hope, you can understand me.
            If not so, select another language please.
            ''',
            lang='en'
        ) + voice.speak(
            '''
            Guten Abend, lieber Kurt! Wie geht es Dir?
            Hier regnet es viel, wie schon den ganzen März und April.
            ''',
            lang='de'
        )
    ).start(thread=False)
```

Some remarks:

- Per call of method `speak()` this program does the following steps in the background:
  - It calls google's tts server, which answers with mp3 data.
  - It calls the system's program *ffmpeg* to convert mp3 into pcm data of the requested sample rate and resolution.
  - It splits the pcm into parts, adds headers and sends part by part to the EV3 device, where they are played.
- This voice was defined as italian speaking. The first call of method `speak()` is without argument *lang*. The second and third calls are with an explicit *lang* argument.
- The timing is automatic, each text gets the time it needs.

2.9.2 Use Voice for your User Interface

Class *Voice* allows to design user interfaces with spoken elements. We run a little program, which uses the EV3 device as a competitve game tool. Two players have 5 seconds time to push a touch sensor as often they can.

Take an USB cable and connect your EV3 brick with your computer, connect two touch sensors to ports 1 and 4, then start this program:

```
import ev3_dc as ev3
from time import sleep

with ev3.Voice(protocol=ev3.USB, volume=100) as voice:
    left_touch = ev3.Touch(ev3.PORT_1, ev3_obj=voice)
    right_touch = ev3.Touch(ev3.PORT_4, ev3_obj=voice)

    voice.speak('Ready', duration=2).start(thread=False)
    voice.speak('Steady', duration=2).start(thread=False)
    voice.speak('Go').start()
    left_touch.bumps = 0
    right_touch.bumps = 0
    sleep(5)

    cnt_left = left_touch.bumps
    cnt_right = right_touch.bumps
    voice.speak(
        f'''
        Stop,
        {cnt_left} on the left side and
        {cnt_right} on the right side
        '''
    ).start(thread=False)
```

Some remarks:

- Compare with program *Bump-Mode*, which uses the display for a simular user interface.
- Keyword argument *ev3_obj* allows the three objects, *voice*, *left_touch* and *right_touch* to share a single connection. *voice* owns the connection and shares it with *left_touch* and *right_touch*.
- Optional argument *duration* lets a task wait some additional time until the duration time is over. This helps for precise timing.
- speaking *Go* executes parallel in its own thread. This says: the five seconds timespan starts when the speaking starts.
- resetting *bumps* prevents from jump starts.

- the formatted multiline string makes *cnt_left* and *cnt_right* part of the spoken text.

2.9.3 Combine Text to Speech with existing Sound Files

Class `Voice` is a subclass of `Sound` and inherits all their methods. Therefore it is straight forward to combine the playing of existing sound files with the speaking of individual texts.

Find the location of LEGO's sound files, which in my case was: *./Program Files (x86)/LEGO Software/LEGO MINDSTORMS EV3 Home Edition/Resources/BrickResources/Retail/Sounds/files* (I, on my Unix system, created a soft link named *Sound*, to get easy access). Modify the program by replacing the file locations. Take an USB cable and connect your EV3 device with your computer then start the following program.

```
import ev3_dc as ev3
from thread_task import Periodic

with ev3.Voice(protocol=ev3.USB, volume=20, lang='en') as hugo:
    (
        Periodic(
            2,  # interval
            hugo.sound(
                '../Sound/Animals/Dog bark 1.rsf',
                local=True
            ),
            num=2,
            duration=3
        ) +
        hugo.speak("Don't panic, she plays only", volume=100) +
        hugo.sound(
            '../Sound/Animals/Dog bark 2.rsf',
            local=True,
            volume=100
        )
    ).start(thread=False)
```

Some remarks:

- All the sound files still exist on the local machine or are produced on the local machine. From there, they are loaded to the EV3 device and played.
- The first barking is wrapped in a Periodic, which repeats it 2 times in an interval of 2 seconds and sets the duration to 3 seconds.
- The speaking, which follows the first barking, takes the language from its *Voice* object, but overwrites the volume.
- The second barking is straight forward. Its not repeated and it reads its duration from the header of the sound file.

2.10 Motor

`Motor` is a subclass of `EV3`. You can use it to move a single motor without any knowledge of direct command syntax. Class Motor uses thread_task.Task, which allows to move motors parallel to other activities.

To use multiple motors, you can create multiple instances of class Motor.

2.10.1 Properties of Class Motor

Beside the properties, which it inherits from its parent class *EV3*, class *Motor* provides some additional properties.

busy

Read only property *busy* tells if the motor currently is busy, which means, it is actively moving.

Take an USB cable and connect your EV3 brick with your computer. Connect a motor (medium or large) with PORT B, then start this program.

```
import ev3_dc as ev3

with ev3.Motor(
    ev3.PORT_B,
    protocol=ev3.USB
) as my_motor:
    if my_motor.busy:
        print('the motor currently is busy')
    else:
        print('the motor currently is not busy')
```

motor_type

Read only property *motor_type* tells the motor type of the motor. The values may be 7 (ev3_dc.EV3_LARGE_MOTOR) or 8 (ev3_dc.EV3_MEDIUM_MOTOR).

Take an USB cable and connect your EV3 brick with your computer. Connect a motor (medium or large) with PORT B, then start this program.

```
import ev3_dc as ev3

with ev3.Motor(
    ev3.PORT_B,
    protocol=ev3.USB
) as my_motor:
    print(f'the motor type: {my_motor.motor_type}')
```

port

Read only property *port* tells the port to which this motor is connected. The values may be 1 (ev3_dc.PORT_A), 2 (ev3_dc.PORT_B), 3 (ev3_dc.PORT_C) or 4 (ev3_dc.PORT_D).

Take an USB cable and connect your EV3 brick with your computer. Connect a motor (medium or large) with PORT B, then start this program.

```
import ev3_dc as ev3

with ev3.Motor(
    ev3.PORT_B,
    protocol=ev3.USB
) as my_motor:
    print(f'the port, where this motor is connected to: {my_motor.port}')
```

position

Property `position` tells the current motor position [degree]. After creating a new object of class *Motor*, its *position* is *0*°. This is independent from the motor's history.

Take an USB cable and connect your EV3 brick with your computer. Connect a motor (medium or large) with PORT B, then start this program.

```
from time import sleep
import ev3_dc as ev3

with ev3.Motor(
    ev3.PORT_B,
    protocol=ev3.USB
) as my_motor:
    print('please move the motor manually (you have 5 sec. of time)')
    sleep(5)

    print(f'the current motor position is: {my_motor.position}°')
```

Property *position* allows to reset the motor's position. This means: the current position becomes the new zero position. As mentioned above, this also is done, whenever a new instance of class *Motor* is instantiated.

Take an USB cable and connect your EV3 brick with your computer. Connect a motor (medium or large) with PORT B, then start this program.

```
from time import sleep
import ev3_dc as ev3

with ev3.Motor(
    ev3.PORT_B,
    protocol=ev3.USB
) as my_motor:
    print('please move the motor manually (you have 5 sec. of time)')
    sleep(5)

    print(f'the current motor position is: {my_motor.position}°')

    my_motor.position = 0
    print(f'after resetting, the new motor position is: {my_motor.position}°')
```

delta_time

Property `delta_time` affects the data traffic and precision of controlled movements. Its default value depends on the connection type and is 0.05 sec. (ev3.USB), 0.10 sec. (ev3.WIFI) and 0.20 sec. (ev3.BLUETOOTH). You can set this value when creating a new *Motor* object, you can also change this value, whenever you need higher precision or whenever you need to reduce the data traffic.

Take an USB cable and connect your EV3 brick with your computer. Connect a motor (medium or large) with PORT B, then start this program.

```
from time import sleep
import ev3_dc as ev3

with ev3.Motor(
    ev3.PORT_B,
    protocol=ev3.USB
```

(continues on next page)

(continued from previous page)

```
) as my_motor:
    print(f'the default value of delta_time is: {my_motor.delta_time} sec.')
    sleep(5)

    my_motor.delta_time = 0.2
    print(f'we reduce data traffic and set delta_time to: {my_motor.delta_time} sec.')
```

speed

Property `speed` and measures in percent and sets the speed of this motor's movements.

Take an USB cable and connect your EV3 brick with your computer. Connect a motor (medium or large) with PORT B, then start this program.

```
from time import sleep
import ev3_dc as ev3

with ev3.Motor(
    ev3.PORT_B,
    protocol=ev3.USB
    speed=100
) as my_motor:
    print(f'speed: {my_motor.speed}%')
    sleep(5)

    my_motor.speed = 20
    print(f'new speed: {my_motor.speed}%')
```

ramp_up and ramp_down

Properties `ramp_up` and `ramp_down` measure in degrees and adjust the smoothness of precise movements. The higher the speed is, the higher these values should be. This relationship is a quadratic one. This says: if you double the speed, you should multiply ramp_up and ramp_down by a factor four.

Take an USB cable and connect your EV3 brick with your computer. Connect a motor (medium or large) with PORT B, then start this program.

```
import ev3_dc as ev3

with ev3.Motor(
    ev3.PORT_B,
    protocol=ev3.USB
) as my_motor:
    print(
        f'defaults of speed: {my_motor.speed}%, ' +
        f'ramp_up: {my_motor.ramp_up}° ' +
        f'and ramp_down: {my_motor.ramp_down}°'
    )
```

The output:

```
defaults of speed: 10%, ramp_up: 15° and ramp_down: 15°
```

There are three options to set *speed*, *ramp_up* and *ramp_down*:

- Set them as keyword arguments, when a new object of class `Motor` is created.
- Use properties to change these values for defined parts of your program.
- Set them as keyword arguments per movement. This option does not affect any of the following movements.

ramp_up_time and ramp_down_time

Properties `ramp_up_time` and `ramp_down_time` measure in seconds and adjust the smoothness of timed movements. As before, the higher the speed is, the higher these values should be. But here the relationship is linear. This says: if you double the speed, you should also double ramp_up_time and ramp_down_time.

Take an USB cable and connect your EV3 brick with your computer. Connect a motor (medium or large) with PORT B, then start this program.

```
import ev3_dc as ev3

with ev3.Motor(
    ev3.PORT_B,
    protocol=ev3.USB
) as my_motor:
    print(
        f'defaults of speed: {my_motor.speed} %, ' +
        f'ramp_up_time: {my_motor.ramp_up_time} sec. ' +
        f'and ramp_down_time: {my_motor.ramp_down_time} sec.'
    )
```

The output:

```
defaults of speed: 10%, ramp_up_time: 0.15 sec. and ramp_down_time: 0.15 sec.
```

2.10.2 Precise and Smooth Motor Movements

move_to

Method `move_to()` returns a thread_task.Task object, which can be started, stopped and continued. You can combine such *Task* objects with other *Task* objects just like you combine LEGO bricks.

Take an USB cable and connect your EV3 brick with your computer. Connect a motor (medium or large) with PORT B, then start this program.

```
from thread_task import Sleep
import ev3_dc as ev3

with ev3.Motor(
    ev3.PORT_B,
    protocol=ev3.USB
) as my_motor:
    movement_plan = (
        my_motor.move_to(360) +
        Sleep(5) +
        my_motor.move_to(0, speed=100, ramp_up=90, ramp_down=90, brake=True) +
        Sleep(0.5) +
        my_motor.stop_as_task(brake=False)
    )

```

(continues on next page)

(continued from previous page)

```
    movement_plan.start()
    print('movement has been started')

    movement_plan.join()
    print('movement has been finished')
```

Some remarks:

- operator + combines two *Task* objects. Here we combine multiple *Task* objects and the resulting *Task* object is named *movement_plan*.
- Starting the *Task* object happens in the blink of an eye even when the movement needs a number of seconds.
- The program joins the *movement_plan*, which says: it waits until the *movement_plan* has finished.
- *movement_plan* first moves the motor to position *360°*, then it sleeps for five sec., then it moves the motor back to its original position.
- The first movements ends with a free floating motor, the second one with activated brake, which is released 0.5 sec. later.
- Explicitly setting *brake=False* in method *stop_as_task* is not needed, this is the default.
- You can manually move the motor in the first sleeping timespan. Try that, it will not prevent the motor from moving back to its original position.
- The first movement moves with default speed of *10%*, the second one moves with maximum speed.
- Joining allows to do the second printing after *movement_plan's* end.

The output:

```
movement has been started
movement has been finished
```

We modify this program:

```
from thread_task import Sleep
import ev3_dc as ev3

with ev3.Motor(
    ev3.PORT_B,
    protocol=ev3.USB
) as my_motor:
    movement_plan = (
        my_motor.move_to(360) +
        Sleep(5) +
        my_motor.move_to(0, speed=100, ramp_up=90, ramp_down=90, brake=True) +
        Sleep(0.5) +
        my_motor.stop_as_task(brake=False)
    )

    print('movement starts now')
    movement_plan.start(thread=False)

    print('movement has been finished')
```

Starting *movement_plan* with keyword argument *thread=False* makes its execution more familiar. The program waits until the movement has finished, then it continues with its next statement. The creation of *movement_plan* with its two movements is not different from the version above.

move_by

Method `move_by()` moves a motor by a given angle. The API is very similar to method `move_to()`.

Take an USB cable and connect your EV3 brick with your computer. Connect a motor (medium or large) with PORT B, then start this program.

```
import ev3_dc as ev3

with ev3.Motor(
    ev3.PORT_B,
    protocol=ev3.USB
) as my_motor:
    (
        my_motor.move_by(360, brake=True) +
        my_motor.move_by(-360)
    ).start(thread=False)
```

Some remarks:

- Programs should never end with any motor's brake in active state. This permanently would cost power until the motor is used again or the LEGO brick shuts down. Therefore the default setting is *brake=False*.
- Here the *Task* has no name, it's a anonymous *Task* object.

The next program really does two things parallel. It plays the song *Frère Jacques* and it moves the motor at port *B* forwards and backwards.

```
import ev3_dc as ev3
from thread_task import Task, Repeated, Sleep
from time import sleep

my_motor = ev3.Motor(
    ev3.PORT_B,
    protocol=ev3.USB
)
my_jukebox = ev3.Jukebox(ev3_obj=my_motor)

t_song = my_jukebox.song(ev3.FRERE_JACQUES, volume=1)
t_movements = Repeated(
    my_motor.move_by(90) + my_motor.move_by(-90)
)
t = Task(t_movements.start) + t_song + Task(t_movements.stop)

t.start()

sleep(5)
t.stop()

sleep(2)
t.cont(thread=False)
print('all done')
```

Some remarks:

- *my_motor* and *my_jukebox* communicate with the same physical EV3 brick. This is, what *ev3_obj=my_motor* means.
- *t_song* is a thread_task.Task object.
- *t_movements* is a thread_task.Repeated object.

- *t*, which combines *t_song* and *t_movements* also is a thread_task.Task object, that can be started, stopped and continued.
- The timing is done by the song *Frère Jacques*. As long as it lasts, the motor moves forwards and backwards.
- The movements are precise and smooth and have a measure of 90 degrees.
- Stopping *t* stops the song and the movement and continuing *t* continues both.
- There is no setting of *speed*, *ramp_up* or *ramp_down*, this program uses the defaults.

start_move_to

Method `start_move_to()` moves a motor to a given position. But it does not control time. It's movement ends after undetermined time and the program can't subsequently follow with the next action.

Take an USB cable and connect your EV3 brick with your computer. Connect a motor (medium or large) with PORT B, then start this program.

```
import ev3_dc as ev3
from time import sleep

my_motor = ev3.Motor(
    ev3.PORT_B,
    protocol=ev3.USB
)

my_motor.start_move_to(90)
sleep(5)
my_motor.start_move_to(0)
```

Some remarks:

- The motor positions are relative to the position from where instance *my_motor* of class `Motor` was created. From then on class *Motor* remembers this position as its zero point.
- Again you can use the timespan between the two movements and move the motor by hand. Class Motor will realize the manual movement and will correctly move the motor back to its zero position.
- Modify the program and set *brake=True* in the first movement. This activates the brake and prevents manual movements.
- Method *start_move_to* does not return a *thread_task.Task* object. It is an ordinary method, it just starts the movement.
- The timing depends on the suggestion, that a movement of 90° needs less than 5 sec. of time. Method *start_move_to* is not time controlled, which makes it different from method *move_to*.

start_move_by

Method `start_move_by()` relates to method `move_by()` as method `start_move_to()` relates to method `move_to()`. It starts a movement without any time control. A program, which needs to know, if the movement still is in progress, can use property `busy`.

Take an USB cable and connect your EV3 brick with your computer. Connect a motor (medium or large) with PORT B, then start this program.

```
from time import sleep
import ev3_dc as ev3

with ev3.Motor(
    ev3.PORT_B,
    protocol=ev3.USB
) as my_motor:
    my_motor.start_move_by(360, brake=False)
    print('movement has been started')

    while my_motor.busy:
        sleep(.1)

    print(f'movement has finished at position {my_motor.position}°')
```

Some remarks:

- The motor does a movement by 360° without time control.
- The time control is done by the while loop.
- Instead of coding the time control this way, think about using method *move_by()*.

2.10.3 Timed and Smooth Motor Movements

move_for

Method *move_for()* returns a thread_task.Task object. It does not set the angle of a movement. Instead it sets its duration. The name is meant as: move for a defined duration.

Take an USB cable and connect your EV3 brick with your computer. Connect a motor (medium or large) with PORT B, then start this program.

```
from time import sleep
import ev3_dc as ev3

with ev3.Motor(
    ev3.PORT_B,
    protocol=ev3.USB
) as my_motor:
    t = my_motor.move_for(
        3,
        speed=20,
        ramp_up_time=0.3,
        ramp_down_time=0.3
    ) + my_motor.move_for(
        3,
        speed=20,
        direction=-1,
        ramp_up_time=0.3,
        ramp_down_time=0.3
    )
    t.start()
    print('movement has been started')

    sleep(2)
    t.stop()
```

(continues on next page)

(continued from previous page)

```
    sleep(3)
    t.cont(thread=False)

    print(f'movement has finished at position {my_motor.position}°')
```

Some remarks:

- As in some examples above, this program schedules two movements, forwards and backwards.
- After two seconds, during the first movement, task t is stopped and continued three seconds later. After the continuation it absolves the last second forwards and then the three seconds backwards.
- Compared with the examples above, here the duration of the task is precisely determined. It lasts exactly six seconds. If stopped and continued, the timespan of the interruption is added on top.

start_move_for

Method `start_move_for()` has the same argument signature as method *move_for*, but it directly starts the movement and does not return a thread_task.Task object.

Take an USB cable and connect your EV3 brick with your computer. Connect a motor (medium or large) with PORT B, then start this program.

```
from time import sleep
import ev3_dc as ev3

with ev3.Motor(
    ev3.PORT_B,
    protocol=ev3.USB
) as my_motor:
    my_motor.start_move_for(
        3,
        speed=20,
        ramp_up_time=0.4,
        ramp_down_time=0.4
    )
    sleep(3)
    print(f'movement has finished at position {my_motor.position}°')
```

2.10.4 Unlimited Motor Movements

Another operating mode of a motor may be to start and steadily run it until something interrupts or stops it. If you like to do so, use methods `start_move()` and `stop()`.

Connect your EV3 brick with your computer via USB, connect a motor (medium or large) with PORT B, then start this program.

```
import ev3_dc as ev3
from time import sleep

my_motor = ev3.Motor(
    ev3.PORT_B,
    protocol=ev3.USB
)
```

(continues on next page)

(continued from previous page)

```
my_motor.verbosity = 1
my_motor.sync_mode = ev3.STD

my_motor.start_move()
sleep(1)
my_motor.start_move(direction=-1)
sleep(1)
my_motor.stop()
```

Some remarks:

- No speed was set, therefore the default speed is used.
- Each movement would last for unlimited time, if not interrupted.
- Interrupting a movement by a next one with significant different speed means mechanical stress for the motor.
- We want analyze the communication, therefore we set *verbosity = 1*.
- *sync_mode = ev3.STD* prevents from needless replies because protocol *USB* would default to *sync_mode = ev3.SYNC*, which replies all requests.

The output:

```
08:30:44.141158 Sent␣
↪0x|15:00|2C:00|80|00:00|AF:00:02:0A:81:64:83:FF:FF:FF:7F:00:00:A6:00:02|
08:30:45.143264 Sent␣
↪0x|15:00|2D:00|80|00:00|AF:00:02:36:81:64:83:FF:FF:FF:7F:00:00:A6:00:02|
08:30:46.145253 Sent 0x|09:00|2E:00|80|00:00|A3:00:02:00|
```

Some remarks:

- The first direct command starts the motor. It consists from two operations: *opOutput_Time_Speed* and *opOutput_Start*.
- The second command interrupts the current motor movement and starts a new movement in opposite direction.
- The third command stops the motor movement.

2.11 Two Wheel Vehicle

TwoWheelVehicle is a subclass of *EV3*. You can use it for synchronized movements of two motors. You need no knowledge of direct command syntax. Class TwoWheelVehicle uses thread_task, which allows to move a vehicle parallel to other activities.

TwoWheelVehicle tracks the movements of the vehicle by tracking the motor movements of its two wheels. This allows to ask for the current position and the current orientation of the vehicle.

2.11.1 Calibration

Class *TwoWheelVehicle* does the tracking by frequently reading the current motor positions of both wheels and then updating the vehicle's position. This works fine if the steps between the recalculations are small (small deltas of angle) or if the motor movements inbetween are steady. This kind of calculation needs two precise informations:

- the wheel's radius and
- the wheel's tread, which is the track width of the two drived wheels.

Therefore we start with two small programs, which allow to determine first the radius, then the tread.

Determine the wheel's radius

Construct a vehicle with two drived wheels, connect your EV3 brick and your computer via WiFi, replace the MAC-address by the one of your EV3 brick, connect the left wheel motor (medium or large) with PORT A and the right wheel motor with PORT D. Measure the diameter of the drived wheels and take half of the diameter as value of radius_wheel (in meter). Then start this program.

```
import ev3_dc as ev3

with ev3.TwoWheelVehicle(
    0.0210,  # radius_wheel_measured
    0.1224,  # tread
    protocol=ev3.WIFI,
    host='00:16:53:42:2B:99',
) as my_vehicle:
    my_vehicle.drive_straight(2.).start(thread=False)
```

Some remarks:

- If you don't own a WiFi dongle, use protocol *BLUETOOTH* instead.
- If your vehicle circles clockwise on place, add `my_vehicle.polarity_right = -1` to your code.
- If your vehicle circles anticlockwise on place, add `my_vehicle.polarity_left = -1` to your code.
- If your vehicle moves backwards, add `my_vehicle.polarity_left = -1` and `my_vehicle.polarity_right = -1` to your code.
- Measure the real distance of your vehicle's movement, then do these steps:
  - Calculate $radius\_wheel_{effective} = radius\_wheel_{measured} \times \frac{real\_distance}{2\,m}$.
  - In the program code replace $radius\_wheel_{measured}$ by $radius\_wheel_{effective}$.
  - Restart the program and again measure the distance of the movement. Now it should be close to $2.00\,m$.
- The last code line looks a bit strange. First we call method *drive_straight()*, which returns an object. Then we call method `start()` of this object and set its keyword argument *thread* to value *False*.

Determine the wheel's tread

Now you know the effective radius of your vehicle's wheels but you need to know the effective width of the vehicle's tread too. Replace *radius_wheel* by your effective value, measure the track width of your vehicle and take it as the tread value, then start the following program and count the circles.

```
import ev3_dc as ev3

with ev3.TwoWheelVehicle(
    0.0210,  # radius_wheel
    0.1224,  # tread_measured
    protocol=ev3.WIFI,
    host='00:16:53:42:2B:99',
) as my_vehicle:
    my_vehicle.drive_turn(3600, 0.0).start(thread=False)
```

Some remarks:

- The vehicle circles anticlockwise because this is the positive direction of rotation.
- 3600 degrees means 10 full circles. You will measure something different. Multiply the number of full circles by 360 degrees and add the fraction of the last circle (in degrees). This is the $real\_angle$ of the rotation. Then do:
  - Calculate $tread_{effective} = tread_{measured} \times \frac{3600}{real\_angle}$.
  - In the program code replace the value $tread_{measured}$ by the value $tread_{effective}$.
  - Restart the program and again measure the total angle of the rotations. Now it should be close to 10 full circles or 3600.
- The precision depends on the tyres. If you use wheels with wide base tyres, then the calibration is less exact. From situation to situation it will be a different location of the contact face, where the grip occurs, which says: the tread width varies.

2.11.2 Precise Driving

Two methods `drive_straight()` and `drive_turn()` allow to specify a series of movements, which the vehicle will follow. Maybe you know turtle from the standard python library. Here is a robotic pendant.

Define a Parcours

Connect your EV3 brick and your computer via WiFi, connect the left wheel motor (medium or large) with PORT A and the right wheel motor with PORT D, replace the values of *radius_wheel* and *tread* with the values from your calibration, then start this program:

```python
import ev3_dc as ev3

with ev3.TwoWheelVehicle(
    0.01518,  # radius_wheel
    0.11495,  # tread
    protocol=ev3.WIFI
) as my_vehicle:
    parcours = (
        my_vehicle.drive_straight(0.5) +
        my_vehicle.drive_turn(120, 0.0) +
        my_vehicle.drive_straight(0.5) +
        my_vehicle.drive_turn(120, 0.0) +
        my_vehicle.drive_straight(0.5) +
        my_vehicle.drive_turn(120, 0.0)
    )
    parcours.start(thread=False)
```

Some remarks:

- The parcours builds an equilateral triangle with a side length of half a meter.
- The program does not start six single movements, it instead defines a parcours and then starts the driving by starting the parcours.
- Method *drive_turn* is called with two arguments, the first one sets the angle, the second one the radius. Here the radius is zero, therefore the vehicle turns on place. Please replace the radius with a positive value greater than zero and start the program again.

- Positive values of *drive_turn*'s angle mean turn to the left, negative values mean turn to the right. Please change the signs of the three angles and start the program again. Then the triangle will be drived clockwise.

Sensor controlled Driving

This example is a more demanding one. It demontrates how to control a thread task by calling its methods *stop* and *cont* and how to do this inside a thread task.

Modify your vehicle and place an infrared sensor on it, which directs forwards. Connect the infrared sensor with port 2, then connect your EV3 brick and your computer with the WiFi and start this program:

```
import ev3_dc as ev3
from thread_task import (
    Task,
    Repeated,
    Periodic,
    STATE_STARTED,
    STATE_FINISHED,
    STATE_STOPPED,
)

with ev3.TwoWheelVehicle(
    0.0210,  # radius_wheel
    0.1224,  # tread
    protocol=ev3.WIFI,
    speed=40
) as vehicle:
    infrared = ev3.Infrared(ev3.PORT_2, ev3_obj=vehicle)

    parcours = (
        Repeated(
            vehicle.drive_turn(360, 0.2) +
            vehicle.drive_turn(-360, 0.2),
            num=2
        )
    )

    def keep_care():
        curr_state = parcours.state
        if curr_state == STATE_FINISHED:
            return True  # all done

        dist = infrared.distance
        if (
            curr_state == STATE_STARTED and
            (dist is not None and dist < 0.1)
        ):
            parcours.stop()
        elif (
            curr_state == STATE_STOPPED and
            (dist is None or dist >= 0.1)
        ):
            parcours.cont()

        return False  # call me again

    (
```

(continues on next page)

(continued from previous page)

```
        Task(parcours.start) +
        Periodic(
            0.1,  # interval
            keep_care
        )
    ).start(thread=False)
```

Some remarks:

- the parcours is a lying eight, build from two circles and wrapped in a Repeated, which makes the vehicle to drive it two times. This says: the vehicle drives two times alongside a lying eight.
- function *keep_care* controls the vehicle's movement and it does three things:
  - it tests if the vehicle already has finished the parcours. If so, it ends the Periodic, which called it.
  - it tests if the vehicle currently is driving (STATE_STARTED) though there is a barrier close in front of the sensor. If so, it stops the driving (read stopping for the details).
  - it tests if the vehicle currently is stopped (STATE_STOPPED) though the infrared sensor does not see something closer than 0.1 m. If so, it lets the vehicle continue its movement (read continue for the details).
- to understand the details of function *keep_care*, you need to understand, how a Periodic works (read Periodic actions for the details).
- *Task(parcours.start)* starts the parcours in its own thread, which says: driving the parcours and reading the sensor happen parallel in different threads.
- the *Periodic* calls function *keep_care* ten times per second, which is often enough to stop the vehicle before it collides with a barrier.

Plotting the Energy Consumption

The option of doing multiple things parallel opens a lot of perspectives. As an example we track the energy consumption of a vehicle by repeatedly reading its battery state. We will realize, that the battery state is not precisely the current one, instead it shows medium values over some time, therefore the result will be not more than reasonable. This example uses module *matplotlib*, which you need to have installed.

Connect your EV3 brick and your computer via WiFi, connect the left wheel motor (medium or large) with PORT A and the right wheel motor with PORT D, replace the values of *radius_wheel* and *tread* with the values from your calibration, then start this program:

```
import matplotlib.pyplot as plt
import datetime
from thread_task import Periodic, Task, Sleep
import ev3_dc as ev3

times = []
powers = []

with ev3.TwoWheelVehicle(
    0.01518,  # radius_wheel
    0.11495,  # tread
    protocol=ev3.WIFI,
    speed=40
) as vehicle:

    def track_power():
```

(continues on next page)

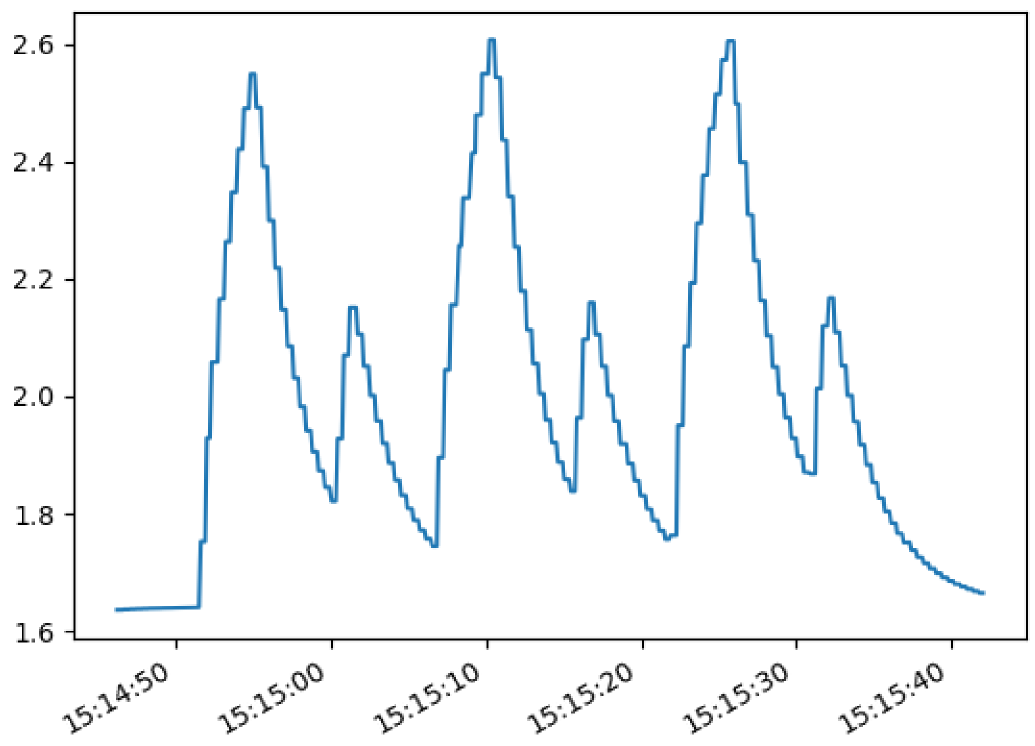
(continued from previous page)

```
        '''
        determine current power consumption
        '''
        battery = vehicle.battery
        times.append(datetime.datetime.now())
        powers.append(battery.voltage * battery.current)

    t_track = Periodic(0.1, track_power)

    t_parcours = (
        Sleep(5) +
        vehicle.drive_straight(0.5) +
        Sleep(5) +
        vehicle.drive_turn(120, 0.0) +
        Sleep(5) +
        vehicle.drive_straight(0.5) +
        Sleep(5) +
        vehicle.drive_turn(120, 0.0) +
        Sleep(5) +
        vehicle.drive_straight(0.5) +
        Sleep(5) +
        vehicle.drive_turn(120, 0.0) +
        Sleep(10)
    )

    (
        Task(t_track.start) +
        t_parcours +
        Task(t_track.stop)
    ).start(thread=False)

    # plot powers over times
    plt.plot(times, powers)
    plt.gcf().autofmt_xdate()
    plt.show()
```

Some remarks:

- The parcours includes timespans with no action. We want to see how the energy consumption differs between action and rest.
- Function *track_power* protocols a single datetime and power [W] in the corresponding lists *times* and *powers*.
- After started, *t_track* would run forever and protocol the current power consumption 10 times per second. Therefore *t_track* is stopped, when the vehicle finished the parcours.
- *Task(t_track.start)* starts *t_track* in its own thread. This says: *t_track* and *t_parcours* run parallel.
- This pattern is typical for executing two thread tasks parallel, when one of the thread tasks sets the timing by its duration.

My program plotted this figure:



Some remarks:

- We expect the highest energy consumtion at the beginning of the movements, when the vehicle accelerates and we expect an immediate fallback to its original value, when the movements end.
- Instead we see a flattened increase and decrease with a flattening over a few seconds. We see the six movements as peaks, but the form of the peaks does not show the real energy consumption.
- Our conclusion: the battery state shows a kind of medium values over a timespan of a few seconds.

2.11.3 Tracking the vehicle's Position and Orientation

Class *TwoWheelVehicle* tracks the vehicle's position and orientation. Property `position` tells the current values. Alternatively, you can use `tracking_callback` to handle the information about the current position and orientation.

Print Current Position

Connect your EV3 brick and your computer via WiFi, replace the MAC-address by the one of your EV3 brick, connect the left wheel motor (medium or large) with PORT A and the right wheel motor with PORT D, replace the values of *radius_wheel* and *tread* with the values from your calibration, then start this program:

```
import ev3_dc as ev3

def print_position(pos: ev3.VehiclePosition) -> None:
```

(continues on next page)

(continued from previous page)

```
    '''
    prints current position and orientation of the vehicle
    '''
    print(
        f'\rx: {pos.x:5.2f} m, y: {pos.y:5.2f} m, o: {pos.o:4.0f} °',
        end=''
    )

with ev3.TwoWheelVehicle(
    0.01518,  # radius_wheel
    0.11495,  # tread
    protocol=ev3.WIFI,
    host='00:16:53:42:2B:99',
    speed=20,
    ramp_up=60,
    ramp_down=60,
    tracking_callback=print_position
) as my_vehicle:
    parcours = my_vehicle.drive_turn(360, 0.2)
    parcours.start(thread=False)
    print('\n' + '-' * 14, 'done', '-' * 13)
    print(my_vehicle.position)
```

Some remarks:

- This parcours drives the vehicle a single cirle in anticlockwise direction.
- The vehicle's tracking uses the middle between the two drived wheels as point of reference and measures in meters.
- The x-axes points in direction of the vehicle's starting orientation. The y-axes points to the left of its starting orientation. The starting position is, as you may have expected, (0.0, 0.0).
- Function *print_position* prints the values of the x- and y-coordinates and the vehicle's orientation whenever it is called. It repeatedly prints the same line. This is done by printing carriage return ("\r") in front of the printed line and ending the line without a newline ("\n").
- After the parcours has been finished, property *position* is printed, which demonstrates the alternative way to get the vehicle's current position.
- This construction of the *TwoWheelVehicle* object uses some more keyword arguments than you have seen before. Beneath `tracking_callback` there also is set a higher `speed` and higher values for `ramp_up` and `ramp_down`.

Visualize the Movement

We use matplotlib to visualize the vehicle's movement. Most of the next program are details of this tool. Some of you already know motplotlib and will find some details to modify. For some of you this will be the first contact with the tool, then take is as it is. Some of you will know the standard python turtle module and we already mentioned it. This program comes even closer to this module and will give some of you a familiar warm feeling.

You need to have *matplotlib* installed. If so, connect your EV3 brick and your computer via WiFi, connect the left wheel motor (medium or large) with PORT A and the right wheel motor with PORT D, replace the values of *radius_wheel* and *tread* with the values from your calibration, then start this program:

```
import matplotlib.pyplot as plt
import ev3_dc as ev3
```

(continues on next page)

(continued from previous page)

```
plt.ion()
fig, ax = plt.subplots()
ax.figure.set_size_inches(5, 5)
ax.grid(True)

ax.set_xlim([-1.1, 1.1])
ax.set_xticks([-1, -0.5, 0.5, 1])
ax.spines['left'].set_position('zero')
ax.spines['right'].set_color('none')

ax.set_ylim([-1.1, 1.1])
ax.set_yticks([-1, -0.5, 0.5, 1])
ax.spines['bottom'].set_position('zero')
ax.spines['top'].set_color('none')

x_values = [0.0,]
y_values = [0.0,]
pos = ax.plot(0.0, 0.0, 'ro')[0]
line = ax.plot(x_values, y_values, 'r-')[0]
fig.canvas.draw()

def plot_curr_pos(curr_pos: ev3.VehiclePosition):
    '''
    updates pos in plot
    '''
    x_values.append(curr_pos.x)
    y_values.append(curr_pos.y)
    line.set_xdata(x_values)
    line.set_ydata(y_values)
    pos.set_xdata(curr_pos.x)
    pos.set_ydata(curr_pos.y)
    fig.canvas.flush_events()

with ev3.TwoWheelVehicle(
    0.01518,  # radius_wheel
    0.11495,  # tread
    protocol=ev3.WIFI,
    speed=50,
    tracking_callback=plot_curr_pos
) as vehicle:
    parcours = (
        vehicle.drive_straight(0.5) +
        vehicle.drive_turn(270, 0.5) +
        vehicle.drive_straight(1.0) +
        vehicle.drive_turn(-270, 0.5) +
        vehicle.drive_straight(0.5)
    )
    parcours.start(thread=False)
```

Some remarks:

- The program consists of three parts. The first does the setup of the plot, the second defines a function *plot_curr_pos*, which updates the plot and the last part lets the vehicle drive a parcours.
- Setting keyword argument *tracking_callback=plot_curr_pos* is an important detail. This tells the `TwoWheelVehicle` object to call function *plot_curr_pos* whenever it reads the current position of the vehicle's motors.

- The parcours lets the vehicle drive a lying eight and start from the figure's center. It starts driving alongside the x-axis. After half a meter it turns 270 ° to the left side until it reaches the y-axis, etc.
- Play around and modify the parcours. Let the vehicle drive your geometric favorites.

2.11.4 Regulated Movements

A parcours, which the vehicle follows, is one option for driving a vehicle. Another option are regulated movements, where sensors or a person take over the vehicle's control. In a car the instruments of regulation are the steering wheel, the gas pedal and others. Class `TwoWheelVehicle` provides method `move()` for this and method *move* knows only two arguments, speed and turn. The sign of argument *speed* sets the movement's direction (forwards or backwards). Argument *turn* is a bit more complicated. It may vary between -200 and 200. Here are explanations for some special values of *turn*:

- -200: circle right on place
- -100: turn right with unmoved right wheel
- 0: straight
- 100: turn left with unmoved left wheel
- 200: circle left on place

Now let's demonstrate it with a program. Connect your EV3 brick and your computer via WiFi, replace the MAC-address by the one of your EV3 brick, connect the left wheel motor (medium or large) with PORT A and the right wheel motor with PORT D, replace the values of *radius_wheel* and *tread* with the values from your calibration, then start this program in a terminal (not in an interactive python shell):

```python
import curses
import ev3_dc as ev3


def main(stdscr) -> None:
    '''
    controls terminal and keyboard events
    '''
    def react():
        '''
        reacts on keyboard arrow key events by modifying speed and turn
        '''
        nonlocal speed, turn
        if c == curses.KEY_LEFT:
            turn += 5
            turn = min(turn, 200)
        elif c == curses.KEY_RIGHT:
            turn -= 5
            turn = max(turn, -200)
        elif c == curses.KEY_UP:
            speed += 5
            speed = min(speed, 100)
        elif c == curses.KEY_DOWN:
            speed -= 5
            speed = max(speed, -100)

    # initialize terminal

    stdscr.clear()
```

(continues on next page)

(continued from previous page)

```
    stdscr.refresh()
    stdscr.addstr(0, 0, 'use Arrows to navigate your vehicle')
    stdscr.addstr(1, 0, 'pause your vehicle with key <p>')
    stdscr.addstr(2, 0, 'terminate with key <q>')

    # control vehicle movement and visualize it

    speed = 0
    turn = 0
    with ev3.TwoWheelVehicle(
        0.01518,  # radius_wheel
        0.11495,  # tread
        protocol=ev3.WIFI,
        host='00:16:53:42:2B:99'
    ) as my_vehicle:
        while True:
            c = stdscr.getch()  # catch keyboard event
            if c in (
                curses.KEY_RIGHT,
                curses.KEY_LEFT,
                curses.KEY_UP,
                curses.KEY_DOWN
            ):
                react()
                my_vehicle.move(speed, turn)  # modify movement
                stdscr.addstr(
                    4,
                    0,
                    f'speed: {speed:4d}, turn: {turn:4d}          '
                )
            elif c == ord('p'):
                speed = 0
                turn = 0
                my_vehicle.stop()  # stop movement
                pos = my_vehicle.position
                stdscr.addstr(
                    4,
                    0,
                    f'x: {pos.x:5.2f} m, y: {pos.y:5.2f} m, o: {pos.o:4.0f} °'
                )
            elif c in (ord('q'), 27):
                my_vehicle.stop()  # finally stop movement
                break

curses.wrapper(main)
```

Some remarks:

- This program is a simple remote control, that uses the arrow keys of the terminal to modify the vehicle's movement. Key <p> pauses the movement, key <q> quits it.
- Python standard module curses is kind of old-fashioned because it uses a terminal instead of a graphical interface.
- *curses* takes the control over the terminal and the keyboard. With *stdscr.getch()* it catches the keyboard events and reacts on the arrow keys.
- Function *react* does the real stuff. It modifies either *speed* or *turn*.

- This program uses two methods of class *TwoWheelVehicle*: *move* and *stop*.
- Method *move* is called whenever an array key event occurs. The next movement replaces (or interrupts) the last one.
- The movement seems to be smooth even when *speed* and *turn* change in steps of 5.
- Whenever the movement pauses, the program shows the vehicle's current position, which demonstrates, that the tracking works with regulated movements too.

2.12 File System

`FileSystem` is a subclass of *EV3*. It uses system commands and allows to operate on EV3's file system (read LEGO's Communication Developer Kit for the details). You can read, write and delete files and directories. Please take care, you can damage the software of your EV3 brick.

2.12.1 Method list_dir

Connect your EV3 brick and your computer with an USB cable. Replace the MAC-address by the one of your EV3 brick, then start this program:

```python
import ev3_dc as ev3

my_ev3 = ev3.FileSystem(
    protocol=ev3.USB,
    host='00:16:53:42:2B:99',
)


def print_header(type: str):
    print()
    if type == 'rsf':
        print('robot sound file                 size (bytes)   md5-checksum')
    elif type == 'rgf':
        print('robot graphics file              size (bytes)   md5-checksum')
    elif type == 'rbf':
        print('robot brick file                 size (bytes)   md5-checksum')
    print('-'*75)


def print_table(files: tuple, type: str):
    header_printed = False
    for file in files:
        if file[0].endswith('.' + type):
            if not header_printed:
                print_header(type)
                header_printed = True
            print(
                '{:27s}  {:12d}  {:12s}'.format(*file)
            )


folders, files = my_ev3.list_dir('/home/root/lms2012/sys/ui')
print_table(files, 'rsf')
print_table(files, 'rgf')
print_table(files, 'rbf')
```

Some remarks:

- This program reads files and subfolders in directory */home/root/lms2012/sys/ui*. Then it prints the sound-, graphics- and brick-files as tables.
- The operating system of the EV3 brick is Unix, therefore use slashes, not backslashes when writing the path of a directory.
- You can use absolute or relative paths, releative paths are relative to */home/root/lms2012/sys*, e.g. our path could also be written as *./ui*.
- files is a tuple of tuples. Per file, we get three data:
  - name of the file,
  - size of the file,
  - md5 checksum of the file's data.
- We ignore subfolders, because in this directory, there is none.

The output:

```
robot sound file             size (bytes)  md5-checksum
----------------------------------------------------------------------------
Startup.rsf                          3109  7BE0A201F57917BC0DDE508E63DD7AD8
PowerDown.rsf                        7939  2381EF46C5166BFF0B5852369E5A2CC7
OverpowerAlert.rsf                   8553  BE802DF67CBBC4E4A40C223BFFF4C14A
GeneralAlarm.rsf                     7300  A40C190AF86C8FA9A7FE9143B36B86EC
DownloadSucces.rsf                   6599  681C88B5930DE152C0BB096F890C492F
Click.rsf                             173  A16F9F1FDDACF56EDF81B4CD968826B4

robot graphics file          size (bytes)  md5-checksum
----------------------------------------------------------------------------
settings_screen.rgf                   600  55186477FDBAF838AEDA09BFDBFAABA2
screen.rgf                           2049  ACE80443D1FA8736231BA21D63260CA4
playrecent_screen.rgf                 600  CDBAE801B780484D80DA95538CF867C2
mindstorms.rgf                        302  BCED9CC85FCB72259F4901E836AED8DF
file_screen.rgf                       600  EFF6FAE6C487828734800AFB912DD700
apps_screen.rgf                       600  19EA377DAD1869512B3759E28B6DECCD
Ani1x.rgf                              42  AB225E46367E84D5FC23649EC4DE1CE9
144x82_POP4.rgf                      1478  7E255363590442E339F93CBDAF222CA1
144x65_POP3.rgf                      1172  2BED43A3D00A5842E4B91E136D232CEA

robot brick file             size (bytes)  md5-checksum
----------------------------------------------------------------------------
ui.rbf                               5030  6F46636743FBDE68B489071E590F0752
```

Now we come to directories. The following program demonstrates, how to recursively read a directory subtree.

```python
import ev3_dc as ev3

my_ev3 = ev3.FileSystem(
    protocol=ev3.USB,
    host='00:16:53:42:2B:99',
)


def dir_recursive(path: str):
    folders, files = my_ev3.list_dir(path)
    for folder in folders:
```

(continues on next page)

(continued from previous page)

```
        if folder in ('.', '..'):
            continue
        next_path = path + '/' + folder
        print(next_path)
        dir_recursive(next_path)


dir_recursive('/home')
```

This program recursively reads the */home* folder, where Unix systems hold the user-owned data. It prints all subfolders, but ignores files inside the folders.

The output:

```
/home/root
/home/root/lms2012
/home/root/lms2012/tools
/home/root/lms2012/tools/WiFi
/home/root/lms2012/tools/Volume
/home/root/lms2012/tools/Sleep
/home/root/lms2012/tools/Brick Info
/home/root/lms2012/tools/Bluetooth
/home/root/lms2012/sys
/home/root/lms2012/sys/ui
/home/root/lms2012/sys/settings
/home/root/lms2012/sys/mod
/home/root/lms2012/sys/lib
/home/root/lms2012/source
/home/root/lms2012/prjs
/home/root/lms2012/prjs/BrkProg_SAVE
/home/root/lms2012/prjs/BrkProg_SAVE/CVS
/home/root/lms2012/apps
/home/root/lms2012/apps/Brick Program
/home/root/lms2012/apps/Brick Program/CVS
/home/root/lms2012/apps/IR Control
/home/root/lms2012/apps/IR Control/CVS
/home/root/lms2012/apps/Port View
/home/root/lms2012/apps/Port View/CVS
/home/root/lms2012/apps/Motor Control
/home/root/lms2012/apps/Motor Control/CVS
```

Some remarks:

- *root* is the only user on this Unix system.
- If you already worked on some projects and did run them on your EV3 brick, you will find them in */home/root/lms2012/prjs*.
- The sequence of subfolders is backward-sorted by name as is the sequence of files.

2.12.2 Method create_dir

Method `create_dir()` allows to create directories in the filesystem of the EV3 brick.

Connect your EV3 brick and your computer with an USB cable. Replace the MAC-address by the one of your EV3 brick, then start this program:

```
import ev3_dc as ev3

my_ev3 = ev3.FileSystem(
    protocol=ev3.USB,
    host='00:16:53:42:2B:99',
)

dir = '/home/root/lms2012/prjs'
subdir = 'tmp'

# read sub-directories
folders, files = my_ev3.list_dir(dir)
print('*** old ***')
for folder in folders:
    print(folder)

# create directory
my_ev3.create_dir(dir + '/' + subdir)

# read sub-directories
folders, files = my_ev3.list_dir(dir)
print('*** new ***')
for folder in folders:
    print(folder)
```

This program first reads the sub-directories of */home/root/lms2012/prjs*, then it creates directory */home/root/lms2012/prjs/tmp* and finally it again reads the sub-directories of */home/root/lms2012/prjs*.

There are a lot of restrictions for user *root*'s filesystem. E.g. you are not allowed to create sub-directories in */home/root* or */home/root/lms2012*. If you try to do that, the EV3 brick answers with an error.

The output:

```
*** old ***
BrkProg_SAVE
..
.
*** new ***
BrkProg_SAVE
tmp
..
.
```

Indeed, after creating directory */home/root/lms2012/prjs/tmp* there is an additional sub-directory named *tmp* in */home/root/lms2012/prjs*.

If you start this program a second time, you will get an error because you can't create a directory that allready exists.

2.12.3 Method del_dir

Method `del_dir()` allows to delete directories in the filesystem of the EV3 brick.

Connect your EV3 brick and your computer with an USB cable and replace the MAC-address by the one of your EV3 brick. The following program is thought to be executed after the one above:

```
import ev3_dc as ev3

```

(continues on next page)

(continued from previous page)

```
my_ev3 = ev3.FileSystem(
    protocol=ev3.USB,
    host='00:16:53:42:2B:99',
)

dir = '/home/root/lms2012/prjs'
subdir = 'tmp'

# read sub-directories
folders, files = my_ev3.list_dir(dir)
print('*** old ***')
for folder in folders:
    print(folder)

# delete directory
my_ev3.del_dir(dir + '/' + subdir)

# read sub-directories
folders, files = my_ev3.list_dir(dir)
print('*** new ***')
for folder in folders:
    print(folder)
```

The program is very similar to the one above, but it deletes a directory instead of creating it.

The output:

```
*** old ***
BrkProg_SAVE
tmp
..
.
*** new ***
BrkProg_SAVE
..
.
```

Indeed, after deleting directory */home/root/lms2012/prjs/tmp* there is no more a sub-directory named *tmp* in */home/root/lms2012/prjs*.

And again, you can't run this program a second time. If you do so, you will get an error because you can't delete a directory that doesn't exist.

If you need to delete non-empty directories, setting keword argument *secure=False* allows to do so.

2.12.4 Method read_file

Connect your EV3 brick and your computer with an USB cable. Replace the MAC-address by the one of your EV3 brick, then start this program:

```
import ev3_dc as ev3
from hashlib import md5

my_ev3 = ev3.FileSystem(
    protocol=ev3.USB,
    host='00:16:53:42:2B:99',
```

(continues on next page)

(continued from previous page)

```
)

folder = '/bin'
filename = 'usb-devices'

# read data from EV3 brick, calculate md5 and write data to local file
data = my_ev3.read_file(folder + '/' + filename)
print('md5-checksum (copy):', md5(data).hexdigest().upper())
with open(filename, 'w') as f:
    f.write(data.decode('utf-8'))

# get md5 of the file from EV3 brick
subfolders, files = my_ev3.list_dir(folder)
for file in files:
    if file[0] == filename:
        print('md5-checksum (orig):', file[2])
```

This program reads file */bin/usb-devices* from the EV3 brick and writes a local copy. The file is part of the brick's operating system. It's human readable because it is a bash-script. The correctness of the reading is demonstrated by two md5-checksums, one from the original on the EV3 brick, the other from the read data.

The output:

```
md5-checksum (copy): 5E78E1B8C0E1E8CB73FDED5DE384C000
md5-checksum (orig): 5E78E1B8C0E1E8CB73FDED5DE384C000
```

2.12.5 Method write_file

Connect your EV3 brick and your computer with an USB cable. Replace the MAC-address by the one of your EV3 brick and start the following program, that creates sub-directory and a file on the EV3 brick. It writes some text into the file and it allows to test if the md5-checksum is the correct one.

```
import ev3_dc as ev3
from hashlib import md5

my_ev3 = ev3.FileSystem(
    protocol=ev3.USB,
    host='00:16:53:42:2B:99',
)

dir = '/home/root/lms2012/prjs'
subdir = 'tmp'
filename = 'some.txt'
txt = 'This is some text.'
txt_bytes = txt.encode('utf-8')

# md5-ckecksum of txt
print('md5-checksum (text):', md5(txt_bytes).hexdigest().upper())

# create directory
my_ev3.create_dir(dir + '/' + subdir)

# write txt into file
my_ev3.write_file(
    dir + '/' + subdir + '/' + filename,
```

(continues on next page)

(continued from previous page)

```
    txt_bytes
)

# md5-checksum of file
folders, files = my_ev3.list_dir(dir + '/' + subdir)
print('md5-checksum (file):', files[0][2])

# delete directory
my_ev3.del_dir(dir + '/' + subdir, secure=False)
```

Some remarks:

- Method *write-file* accepts *bytes* not *str*, therefore we need to encode the text.
- Setting *secure=False* allows to delete the subdirectory with its content. This is done at the end of the program.

The output:

```
md5-checksum (text): 5A42E1F277FBC664677C2D290742176B
md5-checksum (file): 5A42E1F277FBC664677C2D290742176B
```

2.12.6 Method copy_file

Connect your EV3 brick and your computer with an USB cable. Replace the MAC-address by the one of your EV3 brick and start the following program:

```
import ev3_dc as ev3

my_ev3 = ev3.FileSystem(
    protocol=ev3.USB,
    host='00:16:53:42:2B:99',
)

dir = '../prjs/tmp'
filename = dir + '/' + 'some.txt'
filename_copy = dir + '/' + 'copy.txt'
txt = 'This is some text.'

# create directory
my_ev3.create_dir(dir)

# write txt into file
my_ev3.write_file(filename, txt.encode('utf-8'))

# copy file
my_ev3.copy_file(filename, filename_copy)

# read directory's content
folders, files = my_ev3.list_dir(dir)
print('file                         size (bytes)   md5-checksum')
print('-'*75)
for file in files:
    print(
        '{:27s}  {:12d}  {:12s}'.format(*file)
    )
```

(continues on next page)

(continued from previous page)

```
# delete directory
my_ev3.del_dir(dir, secure=False)
```

Some remarks:

- This program works with relative paths.
- As above it creates a sub-directory */home/root/lms2012/prjs/tmp*.
- File */home/root/lms2012/prjs/tmp/some.txt* is created by method `write_file()`, file */home/root/lms2012/prjs/tmp/copy.txt* is created by method `copy_file()`.

The output:

```
file                             size (bytes)  md5-checksum
---------------------------------------------------------------------------
some.txt                                   18  5A42E1F277FBC664677C2D290742176B
copy.txt                                   18  5A42E1F277FBC664677C2D290742176B
```

As expected, both files have the same sizes and md5-checksums.

2.12.7 Method del_file

Method `del_file()` allows to delete single files in the file-system of an EV3 brick. Be careful, when using it, you can even delete files of the EV3 brick's operating system.

Connect your EV3 brick and your computer with an USB cable. Replace the MAC-address by the one of your EV3 brick and start the following program:

```
import ev3_dc as ev3

my_ev3 = ev3.FileSystem(
    protocol=ev3.USB,
    host='00:16:53:42:2B:99',
)

dir = '../prjs/tmp'
filename = dir + '/' + 'some.txt'
filename_copy = dir + '/' + 'copy.txt'
txt = 'This is some text.'

# create directory
my_ev3.create_dir(dir)

# write txt into file
my_ev3.write_file(filename, txt.encode('utf-8'))

# copy file
my_ev3.copy_file(filename, filename_copy)

# delete file
my_ev3.del_file(filename)

# read directory's content
folders, files = my_ev3.list_dir(dir)
print('file                         size (bytes)  md5-checksum')
print('-'*75)
```

(continues on next page)

(continued from previous page)

```
for file in files:
    print(
        '{:27s}  {:12d}  {:12s}'.format(*file)
    )

# delete directory
my_ev3.del_dir(dir, secure=False)
```

The program is very similar to the one above. It uses nearly all methods of class `FileSystem`.

```
file                          size (bytes)  md5-checksum
----------------------------------------------------------------------------
copy.txt                                18  5A42E1F277FBC664677C2D290742176B
```

File *some.txt* has been deleted, only the copy did exist, when `list_dir()` was called.

2.13 PID Controller

A PID controller implements a control loop mechanism, which applies a correction based on a (p)roportional, an (i)ntegrative and a (d)erivative term. *PID controllers* are widely used in industrial control systems.

2.13.1 Background

Let's think of a system which is controlled by a single control signal. E.g. the position of the gas pedal regulates the speed of a car. Any normal car driver is not able to describe the dependency of his car's velocity from the position of the gas pedal. The only thing he does: he changes the pedal's position, when he wants to accelerate or decelerate his car. If we try to analyze the dependency between the gas pedal's position and the car's velocity, we will find it quite complicated, the velocity depends on the position of the gas pedal and on multiple other factors, e.g.:

- the slope of the road,
- the load of the car
- the current acceleration, e.g. at the beginning of the acceleration phase, a car under full gas will have a low speed. The opposite in case of deceleration: high speed combined with low gas.
- the aerodynamic drag of the car, which depends on the car's surface form.
- the efficiency of the car's motor.
- the mechanical and electronical mechanism, which connects the gas pedal with the fuel injector and the motor electronics.

The parametrization of the controller does not need an analysis of these dependencies (as the human driver does not need it). It instead is based on practical experience and some simple rules of thumb.

Let's take a look on the mechanism: The controller modifies a control signal (e.g. the position of the gas pedal) in dependency of a measurement (e.g. the car's velocity). Using a controller needs a sensor, which does the measurement and this measurement becomes the controller's input. The output of the controller, the signal, regulates the process, which says it is used as an input argument of a regulating function or method.

Let's describe it as a formalism:

- In a setup step, a controller is parametrized.
- The parametrized controller is used inside a loop, where the following steps are executed repeatedly:

  - a measurement is done, which returns a value,
  - the controller takes the measurement value as its single input argument and returns a control signal.
  - the control signal is used as an input argument of a regulation.

With this formalism the PID controller is able to adjust the control signal and this adaption often is astonishing close to the reaction of an intelligent observer. We take a closer look on the setup step, which needs:

- A *setpoint*, which is the target measurement value (e.g. the target velocity of a car),
- A *gain*, which describes the (proportional) relation between the error (deviation of the measured value from the *setpoint*) and the control signal (e.g. the position of the gas pedal). High gains help for fast adaption to the *setpoint*, but produce fluctuations.
- An *integration time* (optional), which approx. is the time to eliminate deviations from the past. Be carefull, high integration times produce fluctuations.
- A *deviation time* (optional), which approx. is the forecast time. This forecast says: if the car already accelerates, the gas pedal may stay unchanged, even when the velocity still is lower than the *setpoint*. A high forecast time helps to prevent nervous changes of the gas but reacts sensible on measurement noise. If the velocity measurement shows random fluctuations, the forecast's result will change to the opposite and will itself produce a nervous driving style.

Determing the *setpoint* often is easy, the rest needs some experience and/or trial and error. Probably you will fastly learn to do it step by step with some simple rules of thumb.

2.13.2 Close but not too Close

A few lines of code are worth a thousand words. We start with a quite simple regulation. A vehicle drives towards a barrier and a controller has to regulate this process. The measurement value is the current distance from the barrier, the control signal ist the vehicle's speed. The *setpoint* is the target distance from the barrier and it is obvious, what the controller has to do. If the current distance is higher than the *setpoint*, then the speed has to be positive. If the distance is too small, then the velocity has to be negative. The controller needs not more than a *setpoint* and a *gain* and is a P controller (a proportional controller without any intergational or deviative term).

Construct a vehicle with two drived wheels, connect the left wheel motor (medium or large) with PORT A and the right wheel motor with PORT D. Place an ifrared sensor on your vehicle, which directs forwards. Connect the infrared sensor with port 2, then connect your EV3 brick and your computer via WiFi, replace the MAC-address by the one of your EV3 brick. If your vehicle is not calibrated, then measure the diameter of the drived wheels and take half of the diameter as value of radius_wheel (in meter). Then start this program.

```python
import ev3_dc as ev3
from time import sleep

# proportional controller for vehicle speed
speed_ctrl =  ev3.pid(0.2, 100)

with ev3.TwoWheelVehicle(
    0.01518,  # radius_wheel
    0.11495,  # tread
    protocol=ev3.WIFI,
    host='00:16:53:42:2B:99'
) as vehicle:
    infrared = ev3.Infrared(ev3.PORT_2, ev3_obj=vehicle)
    while True:
        dist = infrared.distance  # read distance
        if dist is None:
```

(continues on next page)

(continued from previous page)

```
            print('\n' + '**** seen nothing ****')
            vehicle.stop()
            break

        # get speed from speed controller
        speed = round(-speed_ctrl(dist))
        if speed > 100:
            speed = 100
        elif speed < -100:
            speed = -100

        vehicle.move(speed, 0)
        print(f'\rdistance: {dist:3.2f} m, speed: {speed:4d} %', end='')

        if speed == 0:
            break

        sleep(0.1)
```

Some remarks:

- Line `speed_ctrl = ev3.pid(0.2, 100)` does the setup by calling `pid()`. *setpoint* is set to *0.2 m*, *gain* is set to *100*. The rule of thumb for setting *gain*: the sensor's measurement accuracy is *1 cm*, therefore a deviation of *1 cm* will result in a speed setting to *1* (percent of maximum speed).
- A *setpoint* of *0.2 m* means: the vehicle adjusts to stand off this distance.
- Line `dist = infrared.distance` does te measurement.
- Line `speed = round(-speed_ctrl(dist))` calls the controller and gets *speed* as its signal setting. This programs inverts the signal because the controller regulates high values with small signals which in our situation is counterproductive.
- The controller returns float values, but speed must be an integer. This is why the program rounds the controller's signal. It also restricts the signal (speed) to the range, which method `move()` accepts.
- Line `print(f'\rdistance: {dist:3.2f} m, speed: {speed:4d} %', end='')` prints the measured *value* and the *signal* from the controller. This helps to quantify the visual impression.
- A *P controller* is a quite simple thing. If you replace `speed_ctrl(dist)` by `100 * (0.2 - dist)` (or `gain * (setpoint - value)`), you will see the very same behaviour of the vehicle.
- The controller is called inside a loop and this loop sleeps 0.1 sec. between each of its cycles. This time step is small enough to get the impression of a smooth adjustment.
- Make your own experience, vary the *gain* and vary the time steps of the loop. High values of both result in overshooting and you will see the vehicle oscillating around the *setpoint*.

2.13.3 Keep the Distance

We modify the program above. Now we add an integrative term to the controller, which makes it a PI controller. We want the vehicle to adjust to a dynamic situation. The vehicle has to follow the movements of the barrier (e.g. your hand) in a constant distance.

The preparation is the same as above. Place your hand in front of the infrared sensor, then start this program:

```
import ev3_dc as ev3
from time import sleep

# PI controller for vehicle speed
speed_ctrl =  ev3.pid(0.2, 500, time_int=5)

with ev3.TwoWheelVehicle(
    0.01518,  # radius_wheel
    0.11495,  # tread
    protocol=ev3.WIFI,
    host='00:16:53:42:2B:99'
) as vehicle:
    infrared = ev3.Infrared(ev3.PORT_2, ev3_obj=vehicle)
    while True:
        dist = infrared.distance  # read distance
        if dist is None:
            print('\n' + '**** seen nothing ****')
            vehicle.stop()
            break

        # get speed from speed controller
        speed = round(-speed_ctrl(dist))
        if speed > 100:
            speed = 100
        elif speed < -100:
            speed = -100

        vehicle.move(speed, 0)
        print(f'\rdistance: {dist:3.2f} m, speed: {speed:4d} %', end='')

        sleep(0.1)
```

Some remarks:

- A PI controller (here with an additional integrative term `time_int=5`) helps to keep the distance at `setpoint = 0.1 m`, even when the barrier moves steady.
- A P controller would not accomplish this. Let's say, the barrier moves with a speed of 50 (percent of the vehicles maximum speed). The P controller's balance distance will be larger than 0.1 m. When we solve equation `50 = -500 * (0.1 - dist)`, we get `dist = 0.2`. This says: balanced state (vehicle and barrier move with the same speed) is reached at a distance of *0.20 m* and not at setpoint distance 0.1 m.
- Again my advice: make your own experience, play around, vary the controller setup and compare the results.

2.13.4 Follow Me

We modify the program once again and add a second controller, which controls argument turn.

The preparation is the same as above. Additionally use a beacon, select its channel 1 and switch it on. Place the beacon in front of the infrared sensor, then start this program (switching off the beacon ends this program):

```
import ev3_dc as ev3
from time import sleep

speed_ctrl =  ev3.pid(0.1, 500, time_int=5)
turn_ctrl = ev3.pid(0, 10)

```

(continues on next page)

(continued from previous page)

```
with ev3.TwoWheelVehicle(
    0.01518,  # radius_wheel
    0.11495,  # tread
    protocol=ev3.WIFI,
    host='00:16:53:42:2B:99'
) as vehicle:
    infrared = ev3.Infrared(ev3.PORT_2, ev3_obj=vehicle, channel=1)
    while True:
        beacon = infrared.beacon  # read position of infrared beacon
        if beacon is None:
            print('\n' + '**** lost connection ****')
            vehicle.stop()
            break

        # get speed from speed controller
        speed = round(-speed_ctrl(beacon.distance))
        if speed > 100:
            speed = 100
        elif speed < -100:
            speed = -100

        # get turn from turn controller
        turn = round(turn_ctrl(beacon.heading))
        if turn > 200:
            turn = 200
        elif turn < -200:
            turn = -200

        vehicle.move(speed, turn)
        print(
            f'\rspeed: {speed:4d} %, turn: {turn:4d}',
            end=''
        )

        sleep(0.1)
```

Some remarks:

- This program uses two controllers, PI controller *speed_ctrl* regulates argument *speed*, P controller *turn_ctrl* regulates argument *turn*.
- The balanced state of argument *turn* is zero. This is a clear hint to use a simple P controller.
- As before, vary the setup, probably you will find a parametrization, which fits better than mine and results in a faster or smoother or even better adjustment.

CHAPTER 3

API documentation

LEGO EV3 direct commands

3.1 Static methods

3.1.1 LCX

Translates an integer into a direct command compatible number with identification byte. It is used for input arguments of operations, which are not read from global or local memory. Dependent from the value an LCX will be a byte string of one, two, three or 5 bytes length.

`ev3_dc.`**`LCX`**`(`*value: int*`)` → bytes

create a LC0, LC1, LC2, LC4, dependent from the value

Positional Argument

value integer value as argument of a direct command

3.1.2 LCS

Adds a leading identification byte and an ending zero terminator to an ascii string and returns a byte string.

`ev3_dc.`**`LCS`**`(`*value: str*`)` → bytes

pack a string into a LCS by adding a leading and a trailing byte

Positional Argument

value string as argument of a direct command

3.1.3 LVX

Translates a local memory adress into a direct command compatible format with identification byte. This can be used for input or output arguments of operations.

`ev3_dc.`**`LVX`**(*value: int*) → bytes

create a LV0, LV1, LV2, LV4, dependent from the value

Positional Argument

value position (bytes address) in the local memory

3.1.4 GVX

Translates a global memory adress into a direct command compatible format with identification byte. This can be used for input or output arguments of operations.

`ev3_dc.`**`GVX`**(*value: int*) → bytes

create a GV0, GV1, GV2, GV4, dependent from the value

Positional Argument

value position (bytes address) in the global memory

3.1.5 port_motor_input

Allows to use well known motor ports of output commands for input commands too.

`ev3_dc.`**`port_motor_input`**(*port_output: int*) → bytes

get corresponding input motor port (from output motor port)

Positional Argument

port_output motor port number

3.1.6 pid

pid is a PID controller. It continuously adjusts a system by periodically calculating a signal value from a measured variable. Function `pid()` does the setup and returns the controller, which is a function (with this signature: `signal(value:  float) -> float`).

`ev3_dc.`**`pid`**(*setpoint: float*, *gain: float*, *, *time_int: float = None*, *time_der: float = None*) → Callable

Parametrize a new PID controller (standard form)

A PID controller derives a control signal from a measurement value

Mandatory positional arguments

setpoint target value of the measurement

gain proportional gain, high values result in fast adaption, but too high values produce oscillations or instabilities

Optional keyword only arguments

time_int time of the integrative term [s] (approx. the time for elimination), compensates errors from the past (e.g. steady-state error) small values produce oscillations or instabilities and increase settling time

time_der time of the derivative term [s] (approx. the forecast time), damps oszillations, decreases overshooting and reduces settling time but reacts sensitive on noise

Returns function signal(value: float) -> float

3.2 Classes

3.2.1 EV3

EV3 establishes a connection between your computer and the EV3 device. It allows to send direct and system commands and receive their replies.

class ev3_dc.**EV3**(*, *protocol: str = None*, *host: str = None*, *ev3_obj: Optional[ev3_dc.ev3.EV3] = None*, *sync_mode: str = None*, *verbosity=0*)

communicates with a LEGO EV3 using direct or system commands

Establish a connection to a LEGO EV3 device

Keyword arguments (either protocol and host or ev3_obj)

protocol 'Bluetooth', 'USB' or 'WiFi'

host MAC-address of the LEGO EV3 (e.g. '00:16:53:42:2B:99')

ev3_obj existing EV3 object (its connections will be used)

sync mode (standard, asynchronous, synchronous) STD - if reply then use DIRECT_COMMAND_REPLY and wait for reply.

ASYNC - if reply then use DIRECT_COMMAND_REPLY, but never wait for reply (it's the task of the calling program).

SYNC - Always use DIRECT_COMMAND_REPLY and wait for reply, which may be empty.

verbosity level (0, 1, 2) of verbosity (prints on stdout).

send_direct_cmd(*ops: bytes*, *, *local_mem: int = 0*, *global_mem: int = 0*, *sync_mode: str = None*, *verbosity: int = None*) → bytes

Send a direct command to the LEGO EV3

Mandatory positional arguments

ops holds netto data only (operations), these fields are added:

length: 2 bytes, little endian

msg_cnt: 2 bytes, little endian

type: 1 byte, DIRECT_COMMAND_REPLY or DIRECT_COMMAND_NO_REPLY

header: 2 bytes, holds sizes of local and global memory

Optional keyword only arguments

local_mem size of the local memory

global_mem size of the global memory

sync_mode synchronization mode (STD, SYNC, ASYNC)

verbosity level (0, 1, 2) of verbosity (prints on stdout).

Returns

STD (sync_mode) if global_mem > 0 then reply else message counter

ASYNC (sync_mode) message counter

SYNC (sync_mode) reply of the LEGO EV3

`send_system_cmd`(*cmd: bytes*, ***, *reply: bool = True*, *verbosity: int = None*) → bytes

Send a system command to the LEGO EV3

Mandatory positional arguments

cmd holds netto data only (cmd and arguments), these fields are added:

length: 2 bytes, little endian

msg_cnt: 2 bytes, little endian

type: 1 byte, SYSTEM_COMMAND_REPLY or SYSTEM_COMMAND_NO_REPLY

Optional keyword only arguments

reply flag if with reply

verbosity level (0, 1, 2) of verbosity (prints on stdout).

Returns

reply (in case of SYSTEM_COMMAND_NO_REPLY: msg_cnt)

`wait_for_reply`(*msg_cnt: bytes*, ***, *verbosity: int = None*) → bytes

Ask the LEGO EV3 for a reply and wait until it is received

Mandatory positional arguments

msg_cnt is the message counter of the corresponding send_direct_cmd

Optional keyword only arguments

verbosity level (0, 1, 2) of verbosity (prints on stdout).

Returns

reply to the direct command (without len, msg_cnt and return status)

`battery`

battery voltage [V], current [A] and percentage (as named tuple)

`host`

mac address of EV3 device

`memory`

total and free memory [kB] (as named tuple)

`name`

name of EV3 device

`network`

name, ip_adr and mac_adr of the EV3 device (as named tuple)

available only for WiFi connected devices, mac_adr is the address of the WiFi dongle

`protocol`

connection type

`sensors`

all connected sensors and motors at all ports (as named tuple Sensors)

You can address a single one by e.g.: ev3_dc.EV3.sensors.Port_3 or ev3_dc.EV3.sensors.Port_C

`sensors_as_dict`

all connected sensors and motors at all ports (as dict)

You can address a single one by e.g.: ev3_dc.EV3.sensors_as_dict[ev3_dc.PORT_1] or ev3_dc.EV3.sensors_as_dict[ev3_dc.PORT_A_SENSOR]

sleep
idle minutes until EV3 shuts down, values from 0 to 120

value 0 says: never shut down

sync_mode
sync mode (standard, asynchronous, synchronous)

STD use DIRECT_COMMAND_REPLY only if global_mem > 0, wait for reply if there is one.

ASYNC use DIRECT_COMMAND_REPLY only if global_mem > 0, never wait for reply (it's the task of the calling program).

SYNC always use DIRECT_COMMAND_REPLY and wait for reply.

The idea is

ASYNC - if there is a reply, it must explicitly be asked for. Control directly comes back.

SYNC - EV3 device is blocked and control comes back, when direct command is finished, which means synchronicity of program and EV3 device.

STD - direct commands with no reply are handled like ASYNC, direct commands with reply are handled like SYNC.

system
system versions and build numbers (as named tuple System)

os_version operating system version

os_build operating system build number

fw_version firmware version

fw_build firmware build number

hw_version hardware version

verbosity
level of verbosity (prints on stdout), values 0, 1 or 2

volume
sound volume [%], values from 0 to 100

3.2.2 Touch

Touch is a subclass of EV3 and allows to read data from a touch sensor, which may be an EV3-Touch or a NXT-Touch.

class ev3_dc.**Touch**(*port: bytes*, ***, *protocol: str = None*, *host: str = None*, *ev3_obj: ev3_dc.ev3.EV3 = None*, *sync_mode: str = None*, *verbosity=0*)
EV3 touch, controls a single touch sensor

Positional Arguments

port port of touch sensor (PORT_1, PORT_2, PORT_3 or PORT_4)

Keyword only Arguments (either protocol and host or ev3_obj)

protocol either ev3_dc.BLUETOOTH, ev3_dc.USB or ev3_dc.WIFI

host mac-address of the LEGO EV3 (e.g. '00:16:53:42:2B:99')

ev3_obj an existing EV3 object (its connections will be used)

sync mode (standard, asynchronous, synchronous) STD - if reply then use DIRECT_COMMAND_REPLY and wait for reply.

ASYNC - if reply then use DIRECT_COMMAND_REPLY, but never wait for reply (it's the task of the calling program).

SYNC - Always use DIRECT_COMMAND_REPLY and wait for reply, which may be empty.

verbosity level (0, 1, 2) of verbosity (prints on stdout).

`clear()` → None
clears bump counter of touch sensor

`bumps`
number of bumps since last clearing of bump counter

`port`
port, where touch sensor is connected (PORT_1, PORT_2, PORT_3 or PORT_4)

`sensor_type`
type of sensor

`touched`
flag, that tells if sensor is currently touched

3.2.3 Infrared

Infrared is a subclass of EV3 and allows to read data from an infrared sensor. It uses three modes of the infrared sensor:

- *proximity* mode, which measures the distance between the sensor an a surface in front of the sensor.
- *seeker* mode, which measures the position (heading and distance) of up to four beacons.
- *remode* mode, which reads the currently pressed buttons of up to four beacons.

class ev3_dc.**Infrared**(*port: bytes*, *, *protocol: str = None*, *host: str = None*, *ev3_obj: ev3_dc.ev3.EV3 = None*, *channel: int = None*, *verbosity=0*)
controls a single infrared sensor

Positional Arguments

port port of infrared sensor (PORT_1, PORT_2, PORT_3 or PORT_4)

Keyword only Arguments (either protocol and host or ev3_obj)

protocol either ev3_dc.BLUETOOTH, ev3_dc.USB or ev3_dc.WIFI

host mac-address of the LEGO EV3 (e.g. '00:16:53:42:2B:99')

ev3_obj an existing EV3 object (its already established connection will be used)

channel beacon sends on this channel (1, 2, 3, 4)

verbosity level (0, 1, 2) of verbosity (prints on stdout).

`beacon`
heading and distance [m] of detected the beacon (seeker mode). returned heading is between -25 and 25.

-25 stands for far left

0 stands for straight forward

25 stands for far right

returned distance is between 0.01 and 1.00 m.

returns either None or namedtuple Beacon with heading and distance

beacons
headings and distances [m] of detected beacons (seeker mode). returned headings are between -25 and 25:

-25 stands for far left

0 stands for straight forward

25 stands for far right

returned distances are between 0.01 and 1.00 m.

returns a tuple of four items, one per channel. Each of them is either None or a namedtuple Beacon with heading and distance

channel
selected channel, on which the beacon sends

distance
distance [m], where the sensor detected something (proximity mode). returned distances are between 0.01 and 1.00 m. None stands for 'seen nothing'.

port
port, where sensor is connected (PORT_1, PORT_2, PORT_3 or PORT_4)

remote
heading and distance [m] of detected beacon (remote mode) returned heading is between -25 and 25:

-25 stands for far left

0 stands for straight forward

25 stands for far right

returned distance is between 0.01 and 1.00 m.

returns either None or namedtuple Beacon with heading and distance

remotes
headings and distances [m] of detected beacons (remote mode). returned headings are between -25 and 25:

-25 stands for far left

0 stands for straight forward

25 stands for far right

returned distances are between 0.01 and 1.00 m.

returns a tuple of four items, each of them is either None or a namedtuple Beacon with heading and distance

3.2.4 Ultrasonic

Ultrasonic is a subclass of EV3 and allows to read data from an ultrasonic sensor, which may be an EV3-Ultrasonic or a NXT-Ultrasonic. It uses mode *EV3-Ultrasonic-Cm* (resp. NXT-Ultrasonic-Cm).

class ev3_dc.**Ultrasonic**(*port: bytes*, ***, *protocol: str = None*, *host: str = None*, *ev3_obj: ev3_dc.ev3.EV3 = None*, *verbosity=0*)
controls a single ultrasonic sensor in cm mode

Positional Arguments

port port of ultrasonic sensor (PORT_1, PORT_2, PORT_3 or PORT_4)

Keyword only Arguments (either protocol and host or ev3_obj)

protocol either ev3_dc.BLUETOOTH, ev3_dc.USB or ev3_dc.WIFI

host mac-address of the LEGO EV3 (e.g. '00:16:53:42:2B:99')

ev3_obj an existing EV3 object (its already established connection will be used)

verbosity level (0, 1, 2) of verbosity (prints on stdout).

distance
distance [m] ahead, where the sensor detected something

distances are between 0.01 and 2.55 m. None stands for 'seen nothing'

port
port, where sensor is connected (PORT_1, PORT_2, PORT_3 or PORT_4)

sensor_type
type of sensor

3.2.5 Color

Color is a subclass of EV3 and allows to read data from a color sensor, which may be an EV3-Color or a NXT-Color. It uses modes *EV3-Color-Reflected*, (resp. *NXT-Color-Reflected*).

class ev3_dc.**Color**(*port: bytes*, *, *protocol: str = None*, *host: str = None*, *ev3_obj: ev3_dc.ev3.EV3 = None*, *channel: int = None*, *verbosity=0*)
controls a single color sensor

Mandatory positional Arguments

port port of color sensor (PORT_1, PORT_2, PORT_3 or PORT_4)

Keyword only Arguments (either protocol and host or ev3_obj)

protocol either ev3_dc.BLUETOOTH, ev3_dc.USB or ev3_dc.WIFI

host mac-address of the LEGO EV3 (e.g. '00:16:53:42:2B:99')

ev3_obj an existing EV3 object (its already established connection will be used)

verbosity level (0, 1, 2) of verbosity (prints on stdout).

ambient
intensity of ambient light in percent [0 - 100]

uses modes EV3-Color-Ambient or NXT-Color-Ambient

color
surface color in front of the sensor

0: none, 1: black, 2: blue, 3: green, 4: yellow, 5: red, 6: white, 7: brown

uses modes EV3-Color-Color or NXT-Color-Color

port
port, where sensor is connected (PORT_1, PORT_2, PORT_3 or PORT_4)

reflected
intensity of the reflected (red) light in percent [0 - 100]

uses modes EV3-Color-Reflected or NXT-Color-Reflected

rgb
surface color in front of the sensor as red, green, blue intensities

intensities are white balanced reflected light intensities [0 - 255]

uses mode EV3-Color-Color, does not work with NXT-Color-Color

rgb_raw
surface color in front of the sensor as red, green, blue intensities

intensities are reflected light intensities [0 - 1024]

uses mode EV3-Color-Color, does not work with NXT-Color-Color

rgb_white_balance
perfect white surface in front of the sensor for calibration

returned intensities are raw reflected light intensities [0 - 1024]

uses mode EV3-Color-Color, does not work with NXT-Color-Color

3.2.6 Gyro

Gyro is a subclass of EV3 and allows to read data from a gyro sensor (EV3-Gyro). It uses mode *EV3-Gyro-Rate & Angle*.

class `ev3_dc.`**Gyro**(*port: bytes*, ***, *protocol: str = None*, *host: str = None*, *ev3_obj: ev3_dc.ev3.EV3 = None*, *verbosity=0*)
controls a single gyro sensor

Positional Arguments

port port of gyro sensor (PORT_1, PORT_2, PORT_3 or PORT_4)

Keyword only Arguments (either protocol and host or ev3_obj)

protocol either ev3_dc.BLUETOOTH, ev3_dc.USB or ev3_dc.WIFI

host mac-address of the LEGO EV3 (e.g. '00:16:53:42:2B:99')

ev3_obj an existing EV3 object (its already established connection will be used)

verbosity level (0, 1, 2) of verbosity (prints on stdout).

reset(*angle=0*) → int
define current angle to be angle 0 (or another given value)

Optional keyword only arguments

angle sets the current angle to this value

Returns

current angle in previous normalization

angle
angle [degree] measured by gyro sensor

port
port, where sensor is connected (PORT_1, PORT_2, PORT_3 or PORT_4)

rate
rate [degree/second] measured by gyro sensor

sensor_type
type of sensor

state
angle [degree] and rate [degree/second] measured by gyro sensor

3.2.7 Sound

Sound is a subclass of EV3 and provides higher order methods for the EV3 sound.

class ev3_dc.**Sound**(**, protocol: str = None, host: str = None, ev3_obj: ev3_dc.ev3.EV3 = None, verbosity: int = 0, volume: int = None*)
basic sound functionality

Keyword only arguments (either protocol and host or ev3_obj)

protocol either ev3_dc.BLUETOOTH, ev3_dc.USB or ev3_dc.WIFI

host mac-address of the LEGO EV3 (e.g. '00:16:53:42:2B:99')

ev3_obj an existing EV3 object (its already established connection will be used)

verbosity level (0, 1, 2) of verbosity (prints on stdout).

volume sound volume [%], values from 0 to 100

duration(*path: str, *, local: bool = False*) → float
detemines duration of a sound file by reading its header

Mandatory positional arguments

path name of the sound file (may be without extension ".rsf") as absolute path, or relative to /home/root/lms2012/sys/

Optional keyword only arguments

local flag, if path is a location on the local host

play_sound(*path: str, *, volume: int = None, repeat: bool = False, local: bool = False*) → None
plays a sound file

Mandatory positional arguments

path name of the sound file (may be without extension ".rsf") as absolute path, or relative to /home/root/lms2012/sys/

Keyword only arguments

volume volume [0 - 100] of tone (defaults to attribute volume)

repeat flag, if repeatedly playing

local flag, if path is a location on the local host (PC)

sound(*path: str, *, volume: int = None, duration: float = None, repeat: bool = False, local: bool = False*) → thread_task.task.Task
returns a Task object, which plays a sound file

Mandatory positional arguments

path name of the sound file (may be without extension ".rsf") as absolute path, or relative to /home/root/lms2012/sys/

Optional keyword only arguments

volume volume [0 - 100] of tone (defaults to attribute volume)

duration total duration in sec., in combination with repeat, this means interruption,

repeat flag, if repeatedly playing (unlimited if no duration is set)

local flag, if path is a location on the local host

stop_sound() → None
stops the sound

tone(*freq: int*, *, *duration: numbers.Number = None*, *volume: int = None*) → None
plays a tone

Mandatory positional arguments

freq frequency of tone, range [250 - 10000]

Optional keyword only arguments

duration duration (sec.) of the tone (no duration means forever)

volume volume [0 - 100] of tone

3.2.8 Jukebox

Jukebox is a subclass of Sound and provides higher order methods for tones and LEDs.

class ev3_dc.**Jukebox**(*, *protocol: str = None*, *host: str = None*, *ev3_obj: ev3_dc.ev3.EV3 = None*, *volume: int = None*, *temperament: int = 440*, *verbosity: int = 0*)
plays songs and uses LEDs

Keyword only arguments (either protocol and host or ev3_obj)

protocol either ev3_dc.BLUETOOTH, ev3_dc.USB or ev3_dc.WIFI

host mac-address of the LEGO EV3 (e.g. '00:16:53:42:2B:99')

ev3_obj an existing EV3 object (its already established connection will be used)

volume sound volume [%], values from 0 to 100

temperament temperament of the tones (default: 440 Hz)

verbosity level (0, 1, 2) of verbosity (prints on stdout).

change_color(*led_pattern: bytes*) → None
changes LED color

Positional arguments:

led_pattern: color of LEDs, f.i. ev3.LED_RED

play_tone(*tone: str*, *, *duration: numbers.Number = None*, *volume: int = None*) → None
plays a tone

Mandatory positional arguments

tone name of tone f.i. "c'", "cb''", "c#"

Optional keyword only arguments

duration length (sec.) of the tone (no duration means forever)

volume volume [0 - 100] of tone (defaults to attribute volume)

song(*song: dict*, *, *volume: int = None*) → thread_task.task.Task
returns a Task object, that plays a song

Mandatory positional arguments

song dict with song data (e.g. ev3.HAPPY_BIRTHDAY)

Keyword only arguments

volume volume [0 - 100] of tone (defaults to attribute volume)

temperament
temperament of the tones (default: 440 Hz)

3.2.9 Voice

Voice is a subclass of Sound and provides higher order methods for speaking. It supports text to speech and calls gTTS, which needs an internet connection. *Voice* allows to select the language, the top level domain and a slower reading speed by supporting *gTTS*'s attributes *lang*, *tld* and *slow*.

gTTS answers with mp3 data, therefore *Voice* calls ffmpeg to convert mp3 to pcm. If *ffmpeg* is not installed on your system, *Voice* will not work.

class ev3_dc.**Voice**(**, protocol: str = None, host: str = None, ev3_obj: ev3_dc.ev3.EV3 = None, volume: int = None, lang: str = 'en', tld: str = 'com', slow: bool = False, verbosity: int = 0*)
lets the EV3 device speak tts (text to sound)

Keyword only arguments (either protocol and host or ev3_obj)

protocol either ev3_dc.BLUETOOTH, ev3_dc.USB or ev3_dc.WIFI

host mac-address of the LEGO EV3 (e.g. '00:16:53:42:2B:99')

ev3_obj an existing EV3 object (its already established connection will be used)

filesystem already existing FileSystem object

volume sound volume [%], values from 0 to 100

lang language, e.g. 'it', 'fr, 'de', 'en' (default)

verbosity level (0, 1, 2) of verbosity (prints on stdout).

speak(*txt: str, *, lang: str = None, tld: str = None, slow: bool = None, duration: numbers.Number = None, volume: int = None*) → thread_task.task.Task
let the EV3 device speak some text

Mandatory positional arguments

txt text to speak

Optional keyword only arguments

lang language, e.g. 'it', 'fr, 'de', 'en' (default)

tld top level domain of google server, e.g. 'de', 'co.jp', 'com' (default)

slow reads text more slowly. Defaults to False

duration length (sec.) of the tone (no duration means forever)

volume volume [0 - 100] of tone (defaults to attribute volume)

3.2.10 Motor

Motor is a subclass of EV3 and provides higher order methods for motor movements.

class ev3_dc.**Motor**(*port: int*, ***, *protocol: str = None*, *host: str = None*, *ev3_obj: ev3_dc.ev3.EV3 = None*, *speed: int = 10*, *ramp_up: int = 15*, *ramp_up_time: float = 0.15*, *ramp_down: int = 15*, *ramp_down_time: float = 0.15*, *delta_time: numbers.Number = None*, *verbosity: int = 0*)

EV3 motor, moves a single motor

Mandatory positional arguments

port port of motor (PORT_A, PORT_B, PORT_C or PORT_D)

Keyword only arguments (either protocol and host or ev3_obj)

protocol BLUETOOTH == 'Bluetooth' USB == 'Usb' WIFI == 'WiFi'

host mac-address of the LEGO EV3 (f.i. '00:16:53:42:2B:99')

ev3_obj an existing EV3 object (its connections will be used)

speed percentage of maximum speed [1 - 100] (default is 10)

ramp_up degrees for ramp-up (default is 15)

ramp_up_time duration of ramp-up (used by move_for, default is 0.1 sec.)

ramp_down degrees for ramp-down (default is 15)

ramp_down_time duration of ramp-down (used by move_for, default is 0.1 sec.)

delta_time timespan between introspections [s] (default depends on protocol, USB: 0.2 sec., WIFI: 0.1 sec., USB: 0.05 sec.)

verbosity level (0, 1, 2) of verbosity (prints on stdout).

cont() → None

continues a stopped movement

cont_as_task() → thread_task.task.Task

continues a stopped movement

Returns

thread_task.Task object, which does the continuing

move_by(*degrees: int*, ***, *speed: int = None*, *ramp_up: int = None*, *ramp_down: int = None*, *brake: bool = False*, *duration: numbers.Number = None*) → thread_task.task.Task

exact and smooth movement of the motor by a given angle.

Positional Arguments

degrees direction (sign) and angle (degrees) of movement

Keyword Arguments

speed percentage of maximum speed [1 - 100]

ramp_up degrees for ramp-up

ramp_down degrees for ramp-down

brake Flag if ending with floating motor (False) or active brake (True).

duration duration of Task execution [s] (waits if movement lasts shorter)

Returns

Task object, that can be started, stopped and continued.

move_for(*duration: float*, ***, *speed: int = None*, *direction: int = 1*, *ramp_up_time: float = None*, *ramp_down_time: float = None*, *brake: bool = False*) → thread_task.task.Task
start moving the motor for a given duration.

Mandatory positional arguments

duration duration of the movement [sec.]

Optional keyword only arguments

speed percentage of maximum speed [1 - 100]

direction direction of movement (-1 or 1)

ramp_up_time duration time for ramp-up [sec.]

ramp_down_time duration time for ramp-down [sec.]

brake flag if ending with floating motor (False) or active brake (True).

Returns

Task object, which can be started, stopped and continued.

move_to(*position: int*, ***, *speed: int = None*, *ramp_up: int = None*, *ramp_down: int = None*, *brake: bool = False*, *duration: numbers.Number = None*) → thread_task.task.Task
move the motor to a given position.

Mandatory positional arguments

position target position (degrees)

Optional keyword only arguments

speed percentage of maximum speed [1 - 100]

ramp_up degrees for ramp-up

ramp_down degrees for ramp-down

brake flag if ending with floating motor (False) or active brake (True).

duration duration of Task execution [s] (waits if movement lasts shorter)

Returns

Task object, which can be started, stopped and continued.

start_move(***, *speed: int = None*, *direction: int = 1*, *ramp_up_time: float = None*) → None
starts unlimited movement of the motor.

Optional keyword only arguments

speed percentage of maximum speed [1 - 100]

direction direction of movement (-1 or 1)

ramp_up_time duration time for ramp-up [sec.]

start_move_by(*degrees: int*, ***, *speed: int = None*, *ramp_up: int = None*, *ramp_down: int = None*, *brake: bool = False*, *_control: bool = False*) → None
starts moving the motor by a given angle (without time control).

Positional Arguments

degrees direction (sign) and angle (degrees) of movement

Keyword Arguments

speed percentage of maximum speed [1 - 100]

ramp_up degrees for ramp-up

ramp_down degrees for ramp-down

brake Flag if ending with floating motor (False) or active brake (True).

start_move_for(*duration: float*, ***, *speed: int = None*, *direction: int = 1*, *ramp_up_time: float = None*, *ramp_down_time: float = None*, *brake: bool = False*, *_control: bool = False*) → None

start moving the motor for a given duration.

Mandatory positional arguments

duration duration of the movement [sec.]

Optional keyword only arguments

speed percentage of maximum speed [1 - 100]

direction direction of movement (-1 or 1)

ramp_up_time duration time for ramp-up [sec.]

ramp_down_time duration time for ramp-down [sec.]

brake flag if ending with floating motor (False) or active brake (True).

start_move_to(*position: int*, ***, *speed: int = None*, *ramp_up: int = None*, *ramp_down: int = None*, *brake: bool = False*, *_control: bool = False*)

start moving the motor to a given position (without time control).

Mandatory positional arguments

position target position (degrees)

Optional keyword only arguments

speed percentage of maximum speed [1 - 100]

ramp_up degrees for ramp-up

ramp_down degrees for ramp-down

brake flag if ending with floating motor (False) or active brake (True).

stop(***, *brake: bool = False*) → None

stops the current motor movement, sets or releases brake

Keyword Arguments

brake flag if stopping with active brake

stop_as_task(***, *brake: bool = False*) → thread_task.task.Task

stops the current motor movement, with or without brake (can be used to release brake)

Optional keyword only arguments

brake flag if stopping with active brake

Returns

thread_task.Task object, which does the stopping

busy

Flag if motor is currently busy

delta_time

timespan between introspections [s]

motor_type
type of motor (7: EV3-Large, 8: EV3-Medium,)

port
port of motor (default: PORT_A)

position
current position of motor [degree]

ramp_down
degrees for ramp-down (default is 15)

ramp_down_time
seconds for ramp-down of timed movements (default is 0.1)

ramp_up
degrees for ramp-up (default is 15)

ramp_up_time
seconds for ramp-up of timed movements (default is 0.1)

speed
speed of movements in percentage of maximum speed [1 - 100] (default is 10)

3.2.11 TwoWheelVehicle

TwoWheelVehicle is a subclass of EV3 and provides higher order methods for moving or driving a vehicle with two drived wheels. It tracks the position of the vehicle.

class ev3_dc.**TwoWheelVehicle**(*radius_wheel: numbers.Number*, *tread: numbers.Number*, ***, *protocol: str = None*, *host: str = None*, *ev3_obj: ev3_dc.ev3.EV3 = None*, *speed: int = 10*, *ramp_up: int = 30*, *ramp_down: int = 30*, *delta_time: numbers.Number = None*, *port_left: bytes = 1*, *port_right: bytes = 8*, *polarity_left: int = 1*, *polarity_right: int = 1*, *tracking_callback: Callable = None*, *verbosity: int = 0*)

EV3 vehicle with two drived wheels

Establishes a connection to a LEGO EV3 device

Mandatory positional arguments

radius_wheel radius of the wheels [m]

tread: the vehicles tread [m]

Keyword only arguments (either protocol and host or ev3_obj)

protocol BLUETOOTH == ‘Bluetooth’, USB == ‘Usb’, WIFI == ‘WiFi’

host mac-address of the LEGO EV3 (e.g. ‘00:16:53:42:2B:99’)

ev3_obj an existing EV3 object (its connections will be used)

speed percentage of maximum speed [1 - 100] (default is 10)

ramp_up degrees for ramp-up (default is 30)

ramp_down degrees for ramp-down (default is 30)

delta_time timespan between introspections [s] (default depends on protocol, USB: 0.2 sec., WIFI: 0.1 sec., USB: 0.05 sec.)

port_left port of left motor (PORT_A, PORT_B, PORT_C or PORT_D)

port_right port of right motor (PORT_A, PORT_B, PORT_C or PORT_D)

polarity_left polarity of left motor rotation, values: -1, 1 (default)

polarity_right polarity of right motor rotation, values: -1, 1 (default)

tracking_callback callable, which frequently tells current position, its single argument must be of type VehiclePosition

verbosity level (0, 1, 2) of verbosity (prints on stdout)

`cont`() → None
continues stopped movement

`drive_straight`(*distance: numbers.Number*, ***, *speed: int = None*, *ramp_up: int = None*, *ramp_down: int = None*, *brake: bool = False*) → thread_task.task.Task
drives the vehicle straight by a given distance

Mandatory positional arguments

distance direction (sign) and distance (meters) of straight movement

Optional keyword only arguments

speed percentage of maximum speed [1 - 100]

ramp_up degrees for ramp-up

ramp_down degrees for ramp-down

brake flag if ending with floating motors (False) or active brake (True).

Returns

Task object, which can be started, stopped and continued.

`drive_turn`(*angle: numbers.Number*, *radius: numbers.Number*, ***, *speed: int = None*, *back: bool = False*, *ramp_up: int = None*, *ramp_down: int = None*, *brake: bool = False*) → thread_task.task.Task
starts driving the vehicle a turn by given angle and radius

Mandatory positional arguments

angle angle of turn (degrees), positive sign: to the left, negative sign: to the right

radius radius of turn (meters)

Optional keyword only arguments

speed percentage of maximum speed [1 - 100]

back flag if backwards

ramp_up degrees for ramp-up

ramp_down degrees for ramp-down

brake Flag if ending with floating motors (False) or active brake (True).

Returns

Task object, which can be started, stopped and continued.

`move`(*speed: int*, *turn: int*) → None
Starts unlimited synchronized movement of the vehicle

Mandatory positional arguments

speed direction (sign) and speed of movement as percentage of maximum speed [-100 - 100]

turn type of turn [-200 - 200]

-200: circle right on place

-100: turn right with unmoved right wheel

0 : straight

100: turn left with unmoved left wheel

200: circle left on place

`stop`(*brake: bool = False*) → None
stops the current motor movements, sets or releases brake

Keyword Arguments

brake flag if stopping with active brake

`busy`
Flag if motors are currently busy

`motor_pos`
current positions of left and right motor [degree] (as named tuple)

`polarity_left`
polarity of left motor rotation (values: -1, 1, default: 1)

`polarity_right`
polarity of left motor rotation (values: -1, 1, default: 1)

`port_left`
port of left wheel (default: PORT_D)

`port_right`
port of right wheel (default: PORT_A)

`position`
current vehicle position (as named tuple)

x and x-coordinates are in meter, orientation is in degree [-180 - 180]

`ramp_down`
degrees for ramp-down

`ramp_up`
degrees for ramp-up

`speed`
speed as percentage of maximum speed [1 - 100]

`tracking_callback`
callable, which frequently tells current vehicle position, its single argument must be of type VehiclePosition

3.2.12 FileSystem

FileSystem is a subclass of EV3 and provides higher order methods for the filesystem of an EV3 device. It allows to read and write EV3's files or directories.

class `ev3_dc.`**`FileSystem`**(**, protocol: str = None, host: str = None, ev3_obj: Optional[ev3_dc.ev3.EV3] = None, sync_mode: str = None, verbosity=0*)
Access to EV3's filesystem

Establish a connection to a LEGO EV3 device

Keyword arguments (either protocol and host or ev3_obj)

protocol 'Bluetooth', 'USB' or 'WiFi'

host MAC-address of the LEGO EV3 (e.g. '00:16:53:42:2B:99')

ev3_obj existing EV3 object (its connections will be used)

sync mode (standard, asynchronous, synchronous) STD - if reply then use DIRECT_COMMAND_REPLY and wait for reply.

ASYNC - if reply then use DIRECT_COMMAND_REPLY, but never wait for reply (it's the task of the calling program).

SYNC - Always use DIRECT_COMMAND_REPLY and wait for reply, which may be empty.

verbosity level (0, 1, 2) of verbosity (prints on stdout).

`copy_file`(*path_source: str*, *path_dest: str*) → None

Copies a file in EV3's file system from its old location to a new one (no error if the file doesn't exist)

Mandatory positional arguments

path_source absolute or relative path (from "/home/root/lms2012/sys/") of the existing file

path_dest absolute or relative path of the new file

`create_dir`(*path: str*) → None

Create a directory in EV3's file system

Mandatory positional arguments

path absolute or relative path (from "/home/root/lms2012/sys/")

`del_dir`(*path: str*, ***, *secure: bool = True*) → None

Delete a directory in EV3's file system

Mandatory positional arguments

path absolute or relative path (from "/home/root/lms2012/sys/")

Optional keyword only arguments

secure flag, if the directory must be empty

`del_file`(*path: str*) → None

Delete a file in EV3's file system

Mandatory positional arguments

path absolute or relative path (from "/home/root/lms2012/sys/") of the file

`list_dir`(*path: str*) → dict

Read one EV3 directory's content

Mandatory positional arguments

path absolute or relative path (from "/home/root/lms2012/sys/") to the directory (f.i. "/bin")

Returns

subfolders tuple of strings (names)

files tuple of tuples (name:str, size:int, md5:str)

`load_file`(*path_source: str*, *path_dest: str*, ***, *check: bool = True*) → None

Copy a local file to EV3's file system

Mandatory positional arguments

path_source absolute or relative path of the existing file in the local file system

path_dest absolute or relative path (from "/home/root/lms2012/sys/") in EV3's file system

Optional keyword only aguments

check flag for check if file already exists with identical MD5 checksum

read_file(*path: str*) → bytes
Read one of EV3's files

Mandatory positional arguments

path absolute or relative path to file (f.i. "/bin/sh")

write_file(*path: str*, *data: bytes*, *, *check: bool = True*) → None
Create a file in EV3's file system and write data into it

Mandatory positional arguments

path absolute or relative path (from "/home/root/lms2012/sys/") of the file

data data to write into the file

Optional keyword only arguments

check flag for check if file already exists with identical MD5 checksum

CHAPTER 4

Index

Python Module Index

Index