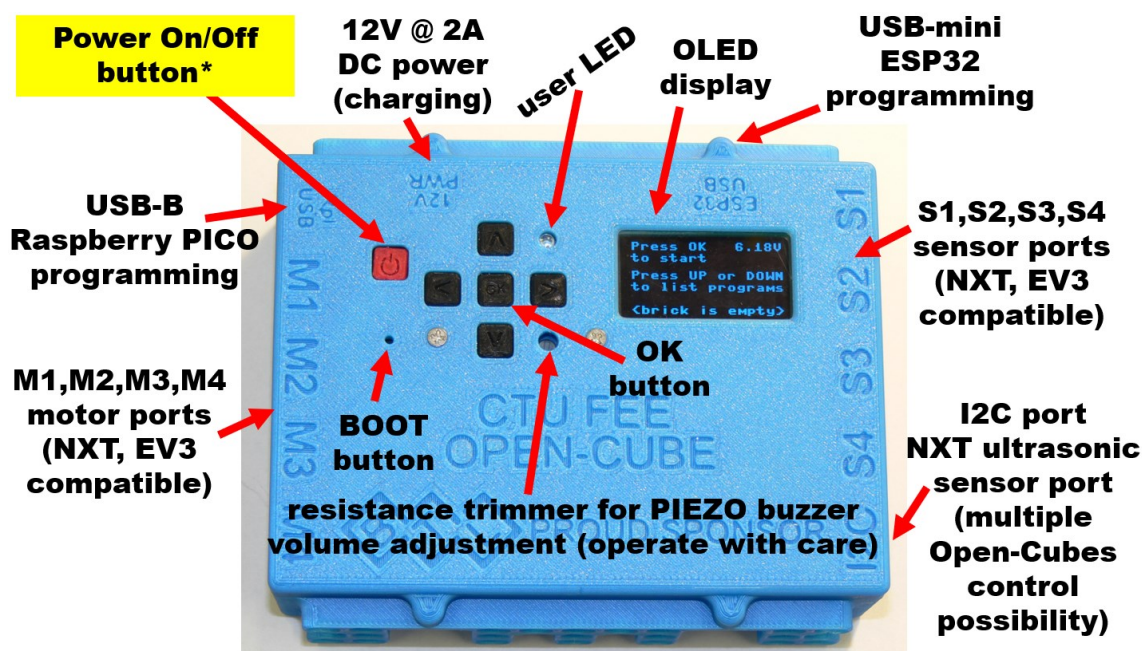


OPEN-CUBE QUICK START GUIDE

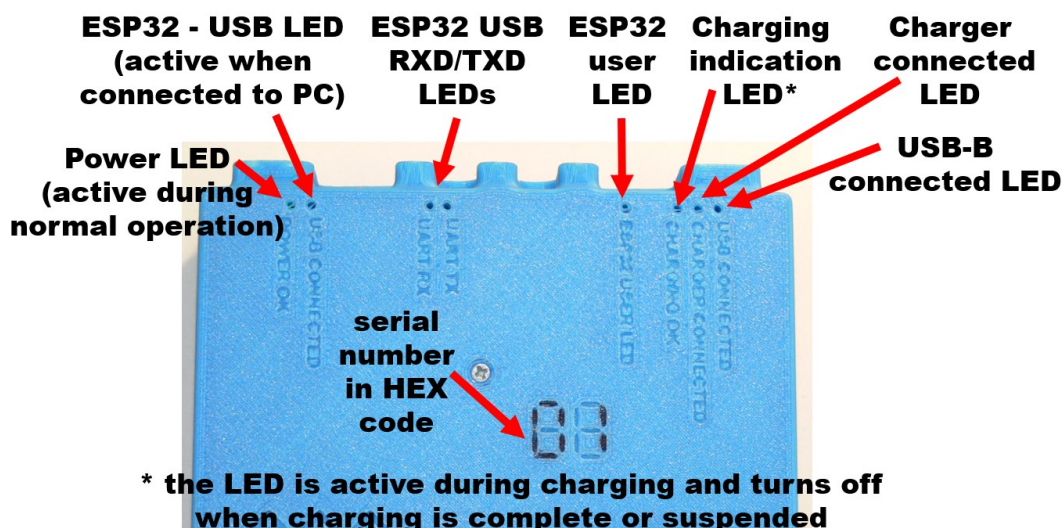
For more information please visit: open-cube.fel.cvut.cz

Switch ON the Open-Cube and connect it to the computer via USB-B (look for COMport number in the device manager)

1. Use Thonny software to create and debug Micropython programs (select Raspberry Pi PICO as a platform and the right COMport, go to Tools/Options/Interpreter...). Install Open-Cube plug-in using plug-in manager.
2. See the next page for a list of most common commands, check the full description of available functions on following pages for more detailed information and useful tips.
3. Alternatively, pair the Open-Cube via Bluetooth serial with your computer, use the plug-in to the Thonny program and upload programs wirelessly, see section 3.9 in the more detailed instructions below.



***a long press (>5s) of the Power button turns the Open-Cube OFF (secured by hardware circuit, will work even if firmware is corrupted)**



LIST OF MOST USEFUL MICROPYTHON O-C COMMANDS

TIMING

```
from time import sleep , sleep_ms , sleep_us
sleep(1)           # Sleep program for 1 second
sleep_ms(2)        # Sleep program for 2 miliseconds
sleep_us(3)        # Sleep program for 3 microseconds look also for "Timer"
```

MOTORS (the same for NXT and EV3), more in chapter 7

```
from lib. robot__consts import Port      # import constants
robot.init__motor(Port.M1)               # initialize the motor
robot.init__motor(Port.M2)               # initialize the motor
robot.motors[Port.M1].set__power(50)     # set power to 50%, clock-wise
robot.motors[Port.M2].set__power(-20)    # set power to 20%, counter-clock-wise
```

SENSORS

NXT TOUCH

```
from lib. robot__consts import Sensor, Port      # import constants, sensor types...
robot.init__sensor(sensor__type = Sensor.NXT__TOUCH, port = Port.S1) # init the sensor on port S1
touch__pressed = robot.sensors.touch[Port.S1].pressed() #read the state of the button
```

NXT OPTICAL see chapter 4.2 for more functions!

```
from lib. robot__consts import Sensor, Port      # import constants, sensor types...
robot.init__sensor(sensor__type = Sensor.LIGHT, port = Port.S1)    # init optical sensor at port S1
robot.sensors.light[Port.S1].on() #switch on the LED
light__intensity = robot.sensors.light[Port.S1].intensity() # measure the light intensity
```

NXT MICROPHONE

```
from lib. robot__consts import Sensor, Port      # import constants, sensor types...
robot.init__sensor(sensor__type = Sensor.NXT__SOUND, port = Port.S1) # init the sensor on port S1
sound__intensity = robot.sensors.sound[Port.S1].intensity() # read the sound intensity
```

NXT ULTRASONIC

```
from lib. robot__consts import Sensor      # import constants, sensor types...
robot.init__sensor(sensor__type = Sensor.NXT__ULTRASONIC) # initialize the sensor
distance = robot.sensors.ultra__nxt.distance() # measure distance
```

NXT GYROSCOPE - currently not implemented (I2C interface, manufactured by HiTechnic)

EV3 TOUCH

```
from lib. robot__consts import Sensor, Port      # import constants, sensor types...
robot.init__sensor(sensor__type = Sensor.EV3__TOUCH, port = Port.S1)    # initialize the sensor
touch__pressed = robot.sensors.touch[Port.S1].pressed() #read the state of the button
```

EV3 OPTICAL see chapter 5.2 for more functions (color sensing for example)

```
from lib. robot__consts import Sensor, Port      # import constants, sensor types...
robot.init__sensor(sensor__type = Sensor.EV3__COLOR, port = Port.S1)    # initialize the sensor
reflection = robot.sensors.light[Port.S1].reflection() # measure the light intensity
```

EV3 ULTRASONIC

```
from lib. robot__consts import Sensor, Port      # import constants, sensor types...
robot.init__sensor(sensor__type = Sensor.EV3__ULTRA, port = Port.S1)    # initialize the sensor
distance = robot.sensors.ultrasonic[Port.S1].distance() # measure the distance
```

EV3 GYROSCOPE *(single axis)*

```
from lib.robot__consts import Sensor, Port          # import constants, sensor types...
robot.init__sensor(sensor__type = Sensor.EV3__GYRO, port = Port.S1)      # initialize the sensor
robot.sensors.gyro[Port.S1].reset__angle(0)        # reset angle
(angle, speed) = robot.sensors.gyro[Port.S1].angle__and__speed()  # measure the angular velocity
in one axis and position
```

O-C OPTICAL *see chapter 6.1 for other modes (color sensing, blue and green light - make POLICE car beacon effect...)*

```
from lib.robot__consts import Sensor, Port          # import constants, sensor types...
robot.init__sensor(sensor__type = Sensor.OC__COLOR, port = Port.S1)      # initialize the sensor
reflection = robot.sensors.light[Port.S1].reflection()  # measure the reflected red light intensity
```

O-C LASER *see chapter 6.2*

```
from lib.robot__consts import Sensor, Port          # import constants, sensor types...
robot.init__sensor(sensor__type = Sensor.OC__LASER, port = Port.S1)      # initialize the sensor
distance = robot.sensors.laser[Port.S1].distance()    # measure the distance
```

O-C ULTRASONIC *see chapter 6.3*

```
from lib.robot__consts import Sensor, Port          # import constants, sensor types...
robot.init__sensor(sensor__type = Sensor.OC__ULTRASONIC, port = Port.S1) # initialize the sensor
distance = robot.sensors.ultrasonic[Port.S1].distance()  # measure the distance
```

O-C AHRS *see chapter 6.4*

```
from lib.robot__consts import Sensor, Port          # import constants, sensor types...
robot.init__sensor(sensor__type = Sensor.OC__GYRO, port = Port.S1)      # initialize the sensor
euler__angles = robot.sensors.gyro[Port.S1].euler__angles()  # read the data (more and calibration
in chapter 6.4)
```

INTERNAL GYROSCOPE *see chapter 3.6 for example of sensor readout using interrupts*

```
robot.init__sensor(sensor__type = Sensor.GYRO__ACC)
a__g__data = (0, 0, 0, 0, 0, 0)          # ax:-, ay:-, az:-, gx:°/s, gy:°/s, gz:°/s
a__g__data = robot.sensors.gyro__acc.read__value()  # measure the accelerations and angular rates
```

MISCELLANEOUS

DISPLAY *see chapter 3.5 for complete list of functions*

```
robot.display.fill(0)          # clear display
robot.display.text("test message",0,0,1)      # print text
x, y = 1.23, 56.789            # set some variables
robot.display.text(f"x={x:.2 f}, y: {y:.2f}", 0, 10, 1)  # print two float numbers
robot.display.show()           # show the framebuffer
```

SOUND - PIEZO

```
robot.buzzer.set__freq__duty(4000,0.5)        # 4000 Hz, 50 % duty (0.0-1.0)
robot.buzzer.off()                            # stop the PWM generation
```

BUTTONS

```
from lib. robot__consts import Button          # import constants, sensor types...
buttons = robot.buttons.pressed()              # read the state
if buttons[Button.LEFT]: break                 # use it in an if condition
# other buttons: Button.LEFT, Button.RIGHT, Button.OK, Button.UP, Button.DOWN
```

LED *(red, on the Open-Cube front panel)*

```
robot.led.on()          # switch LED on
robot.led.off()          # switch LED off
```

BATTERY

voltage = robot.battery.voltage() *#read the internal O-C Li-ion battery voltage*

I2C MASTER, I2C SLAVE (Open-Cube control, or general usage), see chapter 3.7 and 3.8

BLUETOOTH serial communication & **Wi-Fi** access point – see chapter 3.7

MicroPython also supports:

```
if x < 9:    while x < 9:    for y in range(0, 9):
print("debug message!")    # print debug message into Thonny console
def add(number1, number2):    return number1 + number2    # function definition
add(1, 2)    # function usage

send__data = 0.1    #create some variable
buffer = "Values: " + str(send__data) + "," + str(-send__data) + ";"    # Values: 0.1,-0.1;
print(buffer)    #Print the string to PC
or
robot.esp.bt__write(buffer)    # or send it via Bluetooth
```

Want to see **time-plots** of some variables? Look for "*DataPlotter*"

<https://github.com/jirimaier/DataPlotter> nice program created by CTU-FEE student Jiri Maier

```
buffer = "$$P"+str(send__data)+"," +str(-send__data)+";"
```

```
robot.esp.bt__write(buffer)    #more info in bt_serial.py in your Open-Cube or at gitlab
```

"\$\$P" is a specific header to add one point into a graph, see *DataPlotter* documentation

Open-Cube

CTU FEE

2026

Contents

1	Firmware installation	4
2	Thonny IDE	4
2.1	Recommended settings	4
2.2	Uploading code to the cube	4
2.3	Wireless program upload and launch	5
2.4	Cube menu	6
3	Robot	7
3.1	Buttons	10
3.2	LED	11
3.3	Buzzer	11
3.4	Battery	13
3.5	Display	13
3.6	Gyroscope and accelerometer	18
3.7	I2C master	20
3.8	I2C slave	22
3.9	ESP32 communication	23
3.9.1	Bluetooth ESP32 connection with external device	24
3.9.2	Cube	24
3.9.3	Remote control of the cube	29
3.9.4	Serial Bluetooth Terminal	30
3.9.5	Matlab	30
3.9.6	Simulink	31
4	NXT sensors	35
4.1	Touch sensor	35
4.2	Light sensor	35
4.3	Ultrasonic sensor	37
4.4	Sound sensor	38

5	EV3 sensors	39
5.1	Touch sensor	39
5.2	Color sensor	39
5.3	Ultrasonic sensor	41
5.4	Infrared sensor	42
5.5	Gyroscopic sensor	42
6	Open-Cube sensors	44
6.1	Touch sensor	44
6.2	Color sensor	44
6.3	Laser distance sensor	46
6.4	Ultrasonic sensor	47
6.5	AHRS sensor	48
7	Motors	50
8	Parameters and constants	53
8.1	Sensor	53
8.2	Port	54
8.3	Button	55
8.4	Light	55
8.5	GyroAcc	55
8.6	Color	56
8.7	Note	56
8.8	Melody	56
9	Useful MicroPython functions	58
9.1	Information printing	58
9.2	Random number generation	58
9.3	Time	59
9.4	Binary data manipulation	59
9.5	Multi-core programming	60
10	Used Python terms	61

List of codes

1	Using the global robot variable.	9
2	Checking the status of the buttons.	10
3	LED usage.	11
4	Buzzer usage.	12
5	Determining battery voltage.	13
6	Display usage.	17
7	Getting raw data and cube orientation from gyroscope and accelerometer.	19
8	Using the cube for multi-cube communication as an I2C master.	21
9	Using the cube for multi-cube communication as an I2C slave.	23
10	Communication between the micro controller and ESP32.	28
11	Communication with the cube using Matlab.	30
12	NXT Touch Sensor usage.	35
13	NXT Light Sensor usage.	37
14	NXT Ultrasonic Sensor usage.	38
15	NXT Sound Sensor usage.	38
16	EV3 Touch Sensor usage.	39
17	EV3 Color Sensor usage.	41
18	EV3 Ultrasonic sensor usage.	41
19	EV3 Infrared Sensor usage.	42
20	EV3 Gyro Sensor usage.	43
21	Open-Cube Color Sensor usage.	46
22	Open-Cube LIDAR usage.	47
23	Open-Cube Ultrasonic sensor usage.	47
24	Open-Cube AHRS sensor usage.	49
25	Motor usage.	52
26	Printing information.	58
27	Random library usage.	58
28	Time library usage.	59
29	Timer class usage.	59
30	Struct library usage.	60
31	_thread library usage.	60

1 Firmware installation

The MicroPython firmware, along with all the Open-Cube libraries described in this document, is loaded in the cube by default for students. The current firmware can be downloaded from the [Open-Cube releases page](https://gitlab.fel.cvut.cz/open-cube/firmware/-/releases) [<https://gitlab.fel.cvut.cz/open-cube/firmware/-/releases>]. For normal use, download `opencube_micropython_with_pylibs.uf2`, which contains the native modules and the frozen Open-Cube Python libraries. The `without_pylibs` image contains only the native modules.

To upload firmware after an update, or to upload firmware to a new cube, follow these steps:

1. Download the UF2 firmware file from the releases page.
2. Connect the cube to the computer using a USB cable.
3. Open the About page in the cube menu and hold the OK button for approximately one second. Alternatively, hold the boot select button and press the power button.
4. Open the RPI-RP2 directory on your computer and copy the downloaded firmware into it.
5. After uploading the firmware, the cube restarts and the display shows the menu.

2 Thonny IDE

For editing, debugging and uploading code to the cube, we recommend using the [Thonny IDE](https://thonny.org/) [<https://thonny.org/>], which is available for Windows, Mac and Linux.

2.1 Recommended settings

When the editor is started for the first time with the cube connected and running, click the button in the bottom right corner, then click on **Configure interpreter...**, choose **Interpreter** and set the interpreter to **MicroPython (Raspberry Pi Pico)**. It is also recommended to choose **General** and set the option **UI mode** to **regular** or **expert**. In the main Thonny window click on **View** and choose **Files** and **Shell**. The **Files** window is used to browse directories on the computer and the **Shell** window is used to communicate with the cube, display information and error messages.

2.2 Uploading code to the cube

After connecting the switched on cube with a USB cable to the computer, press the **Stop/Restart backend** button in the editor to connect to the cube. The **Shell** box will display the connection information and the **Files** box will display the files loaded in the cube. You can copy files and entire directories between the cube and the computer by right-clicking a file or a directory. Similarly, you can delete or create new files and directories.

You can edit files in both the computer directory and the cube directory. If you open a file in the cube and save it after editing, it will automatically be loaded into the cube.

Upload user programs to the `programs` directory in the cube. A program can be a single file with the `.py` extension, or a directory containing a user file `main.py`. Programs uploaded in this way will appear in the cube menu. The displayed program name is determined by the name of the file without the `.py` extension or the name of the directory containing `main.py` file.

After editing the code, press **Run current script** to run the displayed script on the cube. The firmware startup code creates the global `robot` object before a script is started, therefore the script can use the documented peripherals without creating another `Robot` object. The firmware `main.py` starts the cube menu automatically after boot and should not normally be run manually.

2.3 Wireless program upload and launch

The Open-Cube Thonny plugin allows programs to be transferred and run through the Bluetooth serial connection:

1. Install the plugin from PyPI using `pip install thonny-open-cube`, or install a `.whl` file from the [Open-Cube releases page](https://gitlab.fel.cvut.cz/open-cube/firmware/-/releases) | <https://gitlab.fel.cvut.cz/open-cube/firmware/-/releases> | using **Tools -> Manage plug-ins...** -> **Install from local file**.
2. Pair the cube with your computer (see [3.9.1](#)).
3. Open **Tools -> Options -> Open-Cube** and select the outgoing Bluetooth serial port. The defaults are COM3 on Windows and `/dev/rfcomm0` on Linux.
4. Open a Python file and press **OC Run** or **Shift+F5**. A modified file is saved automatically before upload.
5. Press **OC Stop** or **Shift+F6** to abort an upload. When **Take control** is enabled, the same command stops a running program.

The plugin options include running after upload, taking control of the running program, automatic connection, debug messages, code autocompletion and Bluetooth port. Run after upload, Take control, automatic connection and code autocompletion are enabled by default.

The Open-Cube log displays connection state, cube name, battery voltage and percentage, and firmware information. With Take control enabled, battery state is updated approximately once per second and messages sent using `robot.esp.bt_write()` are shown while the program runs. If automatic connection is enabled, the plugin reconnects after a lost connection.

The **Tools -> Open-Cube File Manager** command lists programs stored on the cube and can delete a selected file or directory after confirmation.

OC Stop first requests a graceful program interruption. If the program cannot process it within approximately three seconds, for example because it is blocked in native code, the plugin resets the cube. Without Take control, the plugin does not monitor the running program and **OC Stop** has no effect after the upload completes.

2.3.0.1 Program transfer

Program transfer uses a 20-packet window, CRC-32 protection for every packet and SHA-256 verification of the complete file. The default payload size is 512 bytes and the receiver accepts payloads up to 1024 bytes. Files are limited to 100 KiB and are accepted only if sufficient flash and RAM are available. The cube writes received data to a temporary file and renames it to its final destination in `/programs` only after successful verification.

2.4 Cube menu

The cube menu is controlled using the LEFT, RIGHT, UP, DOWN and OK buttons. LEFT and RIGHT select the Programs, Motor test, Sensor test, Cube test, BT pair, Settings and About pages. UP and DOWN select an item and OK activates it.

The Settings page contains the following options:

- **Brightness** – display brightness from 10–100% in 10% steps,
- **Invert** – normal or inverted display colours,
- **BT name** – Bluetooth and Wi-Fi name `Open-Cube01` to `Open-Cube99`,
- **Bat calib** – battery measurement multiplier from 0.50–1.50,
- **OC doublecore** – reserve Core 1 for the Open-Cube real-time hardware loop,
- **Reset** – restore default settings.

Changing **OC doublecore** restarts the cube because Core 1 ownership cannot be changed while the hardware loop is running. Settings are stored in `/config.json`.

The About page displays the firmware version, commit and the last four hexadecimal digits of the Pico serial number. Holding OK for approximately one second enters the USB bootloader.

3 Robot

During firmware startup, `boot.py` creates the global object `robot` before starting the menu. The same object is available to programs started from the menu and to scripts executed through Thonny. It contains functions for initializing, deinitializing and accessing cube peripherals. The structure of its peripheral variables is as follows:

```
robot
├── battery
├── buttons
├── buzzer
├── display
├── esp
├── i2c_master
├── i2c_slave
├── led
├── motors[4]
├── sensors
│   ├── ultra_nxt
│   ├── gyro_acc
│   ├── touch[4]
│   ├── light[4]
│   ├── sound[4]
│   ├── ultrasonic[4]
│   ├── gyro[4]
│   ├── infrared[4]
│   └── laser[4]
```

```
class Robot()
```

Initialization, deinitialization and management of motor, sensor and cube peripheral objects. Automatically initializes the buttons, LED, buzzer, battery voltage measurement, display and ESP32 communication. Sensors, motors and optional multi-cube I2C communication can be initialized by the user.

Initialization: Turning on the cube.

Deinitialization: Turning off the cube.

Access: `robot`

```
init_sensor(sensor_type=None, port=None)
```

Sensor initialization.

Parameters: `sensor_type` ([Sensor](#)) – Sensor type.

`port` ([Port](#)) – Sensor port. It is not used for [Sensor.GYRO_ACC](#) and [Sensor.NXT_ULTRASONIC](#); all other sensor types require a port.

```
deinit_sensor(sensor_type=None, port=None)
```

Sensor deinitialization.

Parameters: `sensor_type` ([Sensor](#)) – Sensor type. If not specified, deinitializes any sensor on that port.

`port` ([Port](#)) – Sensor port. If not specified, deinitializes all sensors of given type.

init_motor(*port=None*)

Motor initialization.

Parameters: **port** ([Port](#)) – Motor port.

deinit_motor(*port=None*)

Motor deinitialization. Shuts down the controller, encoder capture, and stops the motor.

Parameters: **port** ([Port](#)) – Motor port.

run(*program, name=None*)

Run a Python function or a .py file using the managed program environment. The display and Bluetooth control are restored and initialized user peripherals are deinitialized when the program exits.

Parameters: **program** (callable/str) – Function or path of the program to run.
name (str) – Optional name displayed while the program runs.

deinit_all()

Turn off the LED and buzzer and deinitialize user I2C, motors and sensors.

reset()

Reset the cube.

bootloader()

Enter the RP2040 USB bootloader.

turn_off()

Turn off the cube.

fw_version()

Return the firmware build date.

storage()

Print the total and free filesystem space.

`init_i2c_master()`

Initialize multi-cube I2C master communication.

`deinit_i2c_master()`

Deinitialize multi-cube I2C master communication.

`init_i2c_slave(address)`

Initialize multi-cube I2C slave communication with an address from 8–119.

`deinit_i2c_slave()`

Deinitialize multi-cube I2C slave communication.

The following code demonstrates the use of the robot class on a simple program for flashing the cube LED. When this program is run from the menu, the LED changes state periodically every second. When the left button is pressed, the program is terminated, the display shows the menu again, and you can start another program.

```
1 # Import sleep function
2 from time import sleep
3 # Import cube button constants
4 from lib.robot_consts import Button
5
6 # Program definition
7 def main():
8     # Access the global variable robot
9     global robot
10
11     # Loop waiting for program termination
12     while True:
13         # Change LED state
14         robot.led.toggle()
15
16         # Get buttons state
17         buttons = robot.buttons.pressed_since()
18         # Terminate program when left button is pressed
19         if buttons[Button.LEFT]:
20             break
21
22         # Wait 1 second
23         sleep(1)
24
25 # Start program
26 main()
```

Code 1: Using the global robot variable.

3.1 Buttons

`class Buttons()`

Information about the state of the buttons on the front panel of the cube. If the cube does not turn itself off after briefly pressing the POWER button, it can be turned off by long pressing this button to disconnect the cube from the power supply.

Initialization: Turning on the cube.

Deinitialization: Turning off the cube.

Access: `robot.buttons`

`pressed()`

Returns current state of the buttons. To find out the status of a specific button, you can use [buttons constants](#).

Returns: Tuple of states of the buttons (power, left, right, ok, up, down). True if the button is pressed, False if not.

Type: (bool, bool, bool, bool, bool, bool)

`pressed_since()`

Returns buttons pressed since the previous call. A navigation button is also reported while it remains held, therefore this method is suitable for loops in which a short press must not be missed. The POWER value is always False because power-button shutdown is handled by the firmware.

Returns: Tuple of button states (power, left, right, ok, up, down).

Type: (bool, bool, bool, bool, bool, bool)

```
1 # Import sleep function
2 from time import sleep
3 # Import cube button constants
4 from lib.robot_consts import Button
5
6 def main():
7     global robot
8     while True:
9         # Get buttons state
10         buttons = robot.buttons.pressed_since()
11
12         # Show buttons state in terminal
13         print("Button pressed:",
14               "power:", buttons[Button.POWER],
15               "left:",  buttons[Button.LEFT],
16               "right:", buttons[Button.RIGHT],
17               "OK:",     buttons[Button.OK],
18               "up:",     buttons[Button.UP],
19               "down:",   buttons[Button.DOWN])
20         # Terminate program when left button is pressed
21         if buttons[Button.LEFT]:
22             break
23         # Wait 1 second
24         sleep(1)
```

```

25
26 # Start program
27 main()

```

Code 2: Checking the status of the buttons.

3.2 LED

```
class Led()
```

Switching on and off the red LED on the front panel of the cube.

Initialization: Turning on the cube.

Deinitialization: Turning off the cube.

Access: robot.led

```
on()
```

Turn on LED.

```
off()
```

Turn off LED.

```
toggle()
```

Change LED state.

```

1 # Turn on LED
2 robot.led.on()
3 # Turn off LED
4 robot.led.off()

```

Code 3: LED usage.

3.3 Buzzer

```
class Buzzer()
```

Controlling the cube buzzer. The buzzer has the highest volume at a frequency of approximately 4500 Hz due to the uneven frequency characteristic of the piezo transducer. The volume can be further adjusted by turning the resistive trimmer on the front panel of the cube.

Initialization: Turning on the cube.

Deinitialization: Turning off the cube.

Access: robot.buzzer

```
set_freq_duty(freq, duty)
```

Set the PWM controlling the buzzer to the desired frequency and duty cycle.

Parameters:

- **freq** (Hz) – PWM frequency. Zero or a negative value produces silence.
- **duty** (float) – PWM duty cycle from 0.0–1.0.

off()

Stop the current note, melody or WAV file and turn off the buzzer.

play_note(*frequency, duration_ms, duty=0.5*)

Start a note for the given duration and return immediately.

Parameters:

- **frequency** (int: Hz) – Note frequency; zero produces a rest.
- **duration_ms** (int: ms) – Note duration.
- **duty** (float) – Duty cycle from 0.0–1.0.

play_sequence(*notes*)

Play a sequence of (**frequency**, **duration_ms**) pairs without blocking the program.

play_melody(*melody*)

Play a [Melody](#) constant, a note sequence or a case-insensitive melody name.

play_wav(*path, loop=False*)

Start non-blocking playback of a WAV file. The supported format is mono unsigned 8-bit PCM with a sample rate from 4–32 kHz.

Parameters:

- **path** (str) – Path to the WAV file.
- **loop** (bool) – Repeat the file until **off()** is called.

is_playing_wav()

Return True while WAV playback is active.

```
1 # Turning on the buzzer at 4000 Hz and 50% duty cycle
2 robot.buzzer.set_freq_duty(4000, 0.5)
3 # Change buzzer frequency to 1000 Hz
4 robot.buzzer.set_freq_duty(1000, 0.5)
5 # Turn off buzzer
6 robot.buzzer.off()
```

Code 4: Buzzer usage.

The Cube test page contains a Melodies program for the predefined [Melody](#) sequences and a Music (WAV) program for files stored in `/music`. The music program creates this directory when necessary, lists its `.wav` files and uses OK to select normal playback, looping and stop.

3.4 Battery

```
class Battery()
```

Voltage measurement on the power supply batteries. An averaged voltage is measured during initialization and refreshed every 200 ms by the native hardware runtime.

The Open-Cube is powered by two Li-ion batteries (nominal voltage of one cell is 3.7 V, maximum charging voltage is 4.2 V, discharging below 3 V already shortens the cell lifetime). In case of full charge, 8.2–8.4 V can be measured. Batteries should not be discharged below 6 V. The firmware initiates shutdown when the measured voltage remains below 6.3 V for two consecutive one-second checks. Hardware undervoltage protection disconnects batteries when voltage drops below 5.8 V.

Initialization: Turning on the cube.

Deinitialization: Turning off the cube.

Access: `robot.battery`

```
voltage()
```

Returns the last measured battery voltage.

Returns: Last measured battery voltage.

Type: float: V

```
read_voltage()
```

Measure battery voltage.

Returns: Measured battery voltage.

Type: float: V

```
set_multiplier(multiplier)
```

Set the battery calibration multiplier and immediately rescale the stored measurement.

Parameters: **multiplier** (float) – Battery calibration multiplier.

```
deinit()
```

Disable periodic battery voltage measurement.

```
1 # Determine battery voltage in volts
2 voltage = robot.battery.voltage()
```

Code 5: Determining battery voltage.

3.5 Display

```
class SH1106_I2C()
```

Show text, geometric shapes and graphs on the display. The activation of individual pixels is written to the frame buffer, which is transferred to the display using the `show` command. The display is black and white and the resolution is 128x64 pixels.

Initialization: Turning on the cube.

Deinitialization: Turning off the cube.

Access: `robot.display`.

`show(full_update=False)`

Submit modified frame-buffer pages to the display. The I2C transfer runs asynchronously and pending changes are merged automatically. Set `full_update` to True to submit all pages.

`fill(color)`

Fill entire frame buffer with specified color.

Parameters: `color` (0/1) – Color – 0 black, 1 white.

`pixel(x, y [, color])`

Set a pixel in the frame buffer to the specified color. If color is not specified, returns the color of pixel.

Parameters: • `x, y` – Pixel coordinates.
• `color` (0/1) – Color – 0 black, 1 white.

Returns: Pixel color.

Type: int (0/1)

`hline(x, y, w, color)`

Draw horizontal line to the frame buffer.

Parameters: • `x, y` – Coordinates of left origin of the line
• `w` – Line width.
• `color` (0/1) – Color – 0 black, 1 white.

`vline(x, y, h, color)`

Draw vertical line to the frame buffer.

Parameters: • `x, y` – Coordinates of upper origin of the line.
• `h` – Line height.
• `color` (0/1) – Color – 0 black, 1 white.

`line(x1, y1, x2, y2, color)`

Draw line to the frame buffer.

Parameters: • `x1, y1` – Coordinates of first origin of the line.
• `x2, y2` – Coordinates of second origin of the line.
• `color` (0/1) – Color – 0 black, 1 white.

rect(*x, y, w, h, color*)

Draw rectangle to the frame buffer.

- Parameters:**
- **x, y** – Coordinates of the upper left corner of the rectangle.
 - **w, h** – Rectangle width and height.
 - **color** (0/1) – Color – 0 black, 1 white.

fill_rect(*x, y, w, h, color*)

Draw filled rectangle to the frame buffer.

- Parameters:**
- **x, y** – Coordinates of the upper left corner of the rectangle.
 - **w, h** – Rectangle width and height.
 - **color** (0/1) – Color – 0 black, 1 white.

ellipse(*x, y, xr, yr, color, f=False, m=0b1111*)

Draw ellipse to the frame buffer.

- Parameters:**
- **x, y** – Coordinates of center of the ellipse.
 - **xr, yr** – Ellipse axes length.
 - **color** (0/1) – Color – 0 black, 1 white.
 - **f** (bool) – Fill the ellipse if the parameter is specified and is True.
 - **m** (4 bits) – Restriction of ellipse drawing to specified quadrants. Bit 0 specifies Q1, b1 Q2, b2 Q3 and b3 Q4. Quadrants are numbered counterclockwise and Q1 is the upper right quadrant.

fill_triangle(*x1, y1, x2, y2, x3, y3, color*)

Draw filled triangle to the frame buffer.

- Parameters:**
- **x1, y1, x2, y2, x3, y3** – Coordinates of triangle corners.
 - **color** (0/1) – Color – 0 black, 1 white.

text(*s, x, y, color=1*)

Draw text to the frame buffer. Characters are 8x8 pixels in size. Only one font is available.

- Parameters:**
- **s** (str) – Text.
 - **x, y** – Coordinates of the upper left corner of the text beginning.
 - **color** (0/1) – Color – 0 black, 1 white.

centered_text(*s, y, c=1*)

Draw centered text to the frame buffer. Characters are 8x8 pixels in size. Only one font is available.

- Parameters:**
- **s** (str) – Text.
 - **y** – Coordinate of the upper border of the text.
 - **c** (0/1) – Color – 0 black, 1 white.

```
draw_bar_chart_v(value, x, y, w, h, low_lim=0, high_lim=100,  
no_of_ticks=5, label=None, redraw=False)
```

Draw vertical bar chart to the frame buffer.

Returns: Redraw chart to reduce flickering.

Type: bool

Parameters:

- **value** (float) – Displayed value.
- **x, y** (int) – Coordinates of the lower left corner of the beginning of the bar chart.
- **w, h** (int) – Bar chart width and height.
- **low_lim, high_lim** (int) – Lower and upper constraints of the bar chart.
- **no_of_ticks** (int) – Number of chart divisions for better readability. Recommended maximum value is 5
- **label** (str) – Chart label.
- **redraw** (bool) – Chart redraw.

```
draw_bar_chart_h(value, x, y, w, h, low_lim=0, high_lim=100,  
no_of_ticks=5, label=None, redraw=False)
```

Draw horizontal bar chart to the frame buffer.

Returns: Redraw chart to reduce flickering.

Type: bool

Parameters:

- **value** (float) – Displayed value.
- **x, y** (int) – Coordinates of the lower left corner of the beginning of the bar chart.
- **w, h** (int) – Bar chart width and height.
- **low_lim, high_lim** (int) – Lower and upper constraints of the bar chart.
- **no_of_ticks** (int) – Number of chart divisions for better readability. Recommended maximum value is 5
- **label** (str) – Chart label.
- **redraw** (bool) – Chart redraw.

```
draw_dial(value, x, y, r, loval, hival, no_of_steps, sa, label, redraw)
```

Draw dial to the frame buffer.

Returns: Redraw dial to reduce flickering.

Type: bool

Parameters:

- **value** (float) – Displayed value.
- **x, y** (int) – Dial center coordinates.
- **r** (int) – Dial radius.
- **loval, hival** (int) – Lower and upper constraints of the dial.
- **no_of_steps** (int) – Number of dial divisions. Recommended maximum value is 10.
- **sa** (int) – Dial sweep angle in degrees.
- **label** (str) – Dial label.
- **redraw** (bool) – Dial redraw.

continuous_graph(*x, y, gx, gy, w, h, xlo, xhi, ylo, yhi, label, redraw*)

Draw continuous graph to the frame buffer. The function internally maintains graph data points. When the width of the graph is filled, the previous data points are removed and the graph is redrawn from the beginning.

Returns: Redraw chart to reduce flickering.

Type: bool

Parameters:

- **x, y** (float) – Data point to be shown.
- **gx, gy** (int) – Coordinates of the lower left corner of the graph.
- **w, h** (int) – Graph width and height.
- **xlo, xhi** (int) – Lower and upper constraints of the x axis.
- **ylo, yhi** (int) – Lower and upper constraints of the y axis.
- **label** (str) – Graph label.
- **redraw** (bool) – Graph redraw.

contrast(*contrast*)

Set display contrast from 0–255.

invert(*invert*)

Enable or disable inverted display colours.

flip(*flag=None, update=True*)

Select the normal or 180-degree display orientation.

draw_loading_bar(*progress, y=0, bar_width=80, bar_height=10, label=None*)

Draw a loading bar in the frame buffer.

draw_battery_icon(*battery_level, y=0, x=0*)

Draw a battery icon in the frame buffer.

```
1 # Delete frame buffer
2 robot.display.fill(0)
3 # Draw text to the upper left corner of the frame buffer
4 robot.display.text("test message", 0, 0, 1)
5
6 # Writing numbers with precision of 2 decimal places to the frame
  buffer on the next line
7 x, y = 1.23, 56.789
8 robot.display.text(f"x={x:.2f}, y: {y:.2f}", 0, 10, 1)
9 # Show frame buffer on the display
10 robot.display.show()
```

```

11
12 # Functions that return redraw value to reduce graph flicker should be
    called in the loop as follows
13 redraw = True
14 while(True):
15     redraw = robot.display.draw_dial(50,128/2,40,20,0,100,10,270,
        "Speedometer",redraw)

```

Code 6: Display usage.

3.6 Gyroscope and accelerometer

`class ICM42688()`

Integrated gyroscope and accelerometer ICM42688.

Measurement of acceleration and angular velocity in three axes. Sensor data fusion to obtain cube orientation – Euler angles and quaternions.

Initialization: `robot.init_sensor(sensor_type=Sensor.GYRO_ACC)`

Deinitialization: `robot.deinit_sensor(sensor_type=Sensor.GYRO_ACC)`

Access: `robot.sensors.gyro_acc`

`raw()`

Returns the last measured angular velocity and acceleration values. The `GyroAcc` constants of the gyroscope and accelerometer can be used to find a specific value.

Returns: Tuple of accelerations and angular velocities in the x, y, z axes.

Type: (a_x: g, a_y: g, a_z: g, g_x: °/s, g_y: °/s, g_z: °/s)

`euler_angles()`

Returns the orientation of the cube in Euler angles. The fusion algorithm must be on.

Returns: Tuple of Euler angles yaw, pitch roll.

Type: (yaw: °, pitch: °, roll: °)

`quaternions()`

Returns the orientation of the cube in quaternions. The fusion algorithm must be on.

Returns: Tuple of quaternions.

Type: (w, x, y, z)

`start_fusion()`

Turns on the accelerometer and gyroscope data fusion algorithm to obtain Euler angles and quaternions.

`stop_fusion()`

Turns off the accelerometer and gyroscope data fusion algorithm.

`reset_fusion()`

Reset heading to zero and derive pitch and roll from gravity in the selected coordinate frame.

`calibrate_gyro()`

Start gyroscope-offset calibration. Keep the cube stationary and call the method until it returns True.

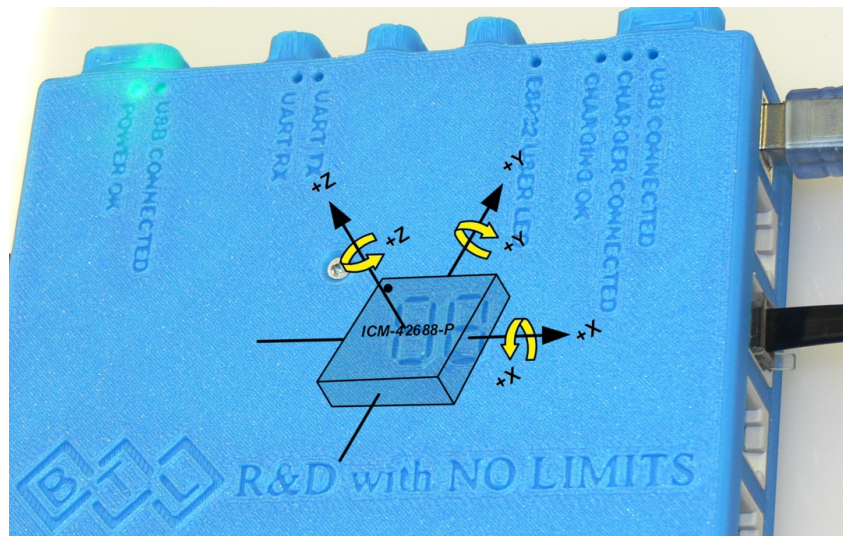


Figure 1: Orientation of integrated gyroscope and accelerometer axes.

Example of getting raw data and cube orientation from gyroscope and accelerometer:

```
1 # Importing libraries
2 from time import sleep
3 # Importing constants
4 from lib.robot_consts import Button, Sensor, GyroAcc
5
6 # Initializing sensor and data fusion algorithm
7 robot.init_sensor(sensor_type=Sensor.GYRO_ACC)
8 robot.sensors.gyro_acc.start_fusion()
9
10 # Loop waiting for the program to exit
11 while True:
12     # Display the last acceleration and angular velocity values
13     a_g_data = robot.sensors.gyro_acc.raw()
14     print("Acc:",
15           a_g_data[GyroAcc.AX],
16           a_g_data[GyroAcc.AY],
17           a_g_data[GyroAcc.AZ],
18           "Gyro:",
19           a_g_data[GyroAcc.GX],
20           a_g_data[GyroAcc.GY],
21           a_g_data[GyroAcc.GZ])
22
```

```

23 # Display the orientation of the cube
24 euler_angles = robot.sensors.gyro_acc.euler_angles()
25 print("Yaw:", euler_angles[0],
26       "Pitch:", euler_angles[1],
27       "Roll:", euler_angles[2])
28
29 # Get button state
30 buttons = robot.buttons.pressed()
31 # Exit the program if the left button is pressed
32 if buttons[Button.LEFT]:
33     break
34 # Put the program to sleep for 1 second
35 sleep(1)
36
37 # Deinitialize sensor
38 robot.deinit_sensor(sensor_type=Sensor.GYRO_ACC)

```

Code 7: Getting raw data and cube orientation from gyroscope and accelerometer.

3.7 I2C master

```
class I2C_master()
```

Set the cube as I2C master device for multi-cube communication. Cubes can be connected using the I2C port (NXT UTZ port) and a modified cable. For proper operation one cube must be set as I2C master and the others as I2C slave. The I2C master reads or writes data to the 252 I2C slave registers.

Initialization: `robot.init_i2c_master()`
Deinitialization: `robot.deinit_i2c_master()`
Access: `robot.i2c_master`

```
read(slave_add, len=1, mem_add=None)
```

Read data from I2C slave device.

Parameters: **slave_add** (8-119) – Address of the I2C slave device.
len (int) – Number of registers (bytes) to read.
mem_add (0-251) – Register address to read. If None, it is read from the last address read or written.
Returns: Data from I2C slave device.
Type: bytes

```
write(slave_add, message, mem_add=None)
```

Writes data to the I2C slave device.

Parameters: **slave_add** (8-119) – Address of the I2C slave device.
message (bytes) – Data to be written.
mem_add (0-251) – Address of the register to write. If None, writes from the last read or written address.
Returns: True if writing is successful, False if not.
Type: bool

trigger(*time_us*, *slave_add*, *callback=None*)

Sets the wake-up time for the I2C slave device. The I2C slave must have a callback function set in advance.

Parameters: **time_us** (int) – Time (us) after which the set callback function will be called.

slave_add (8-119) – Address of the I2C slave device.

callback (callable) – Callback function that is called in the I2C master device after the specified time has elapsed.

Returns: True if the setting is successful, False if not.

Type: bool

trigger_all(*time_us*, *slave_adds=None*, *callback=None*)

Sets the wake-up of several I2C slave devices. I2C slaves must have callback functions set in advance.

Parameters: **time_us** (int) – Time (us) after which the set callback function will be called.

slave_adds (list) – List of I2C slave device addresses. If None, performs an I2C bus scan and sets the trigger for all devices found.

callback (callable) – Callback function that is called in the I2C master device after the specified time has elapsed.

Returns: True if the setting is successful, False if not (for example, if the selected time for calling the callback function is too short).

Type: bool

scan()

Performs a scan of the I2C bus.

Returns: A list of I2C slave device addresses available on the I2C bus.

Type: list

```
1 # Initialize the cube as an I2C master
2 robot.init_i2c_master()
3
4 # Scan the I2C bus to obtain all I2C slave device addresses
5 all_adds = robot.i2c_master.scan()
6
7 # Write and read the character "a" (97) to the I2C slave device with
  address 10 in register 1
8 robot.i2c_master.write(10, (97).to_bytes(1, 'little'), mem_add=1)
9 data_read = robot.i2c_master.read(10, 1, mem_add=1)
10 print(chr(int.from_bytes(data_read, 'little'))) # "a"
11
12 # Callback function for synchronization between cubes
13 def master_trigger_callback(timer):
14     robot.led.on()
15
16 # Set trigger in all found I2C slave devices in 1 s (1000000 us)
17 robot.i2c_master.trigger_all(1000000, slave_adds=all_adds, callback=
    master_trigger_callback)
```

```

18
19 # Deinitialize I2C master
20 robot.deinit_i2c_master()

```

Code 8: Using the cube for multi-cube communication as an I2C master.

3.8 I2C slave

```
class I2C_slave()
```

Set the cube as an I2C slave device for multi-cube communication. Cubes can be connected using the I2C port (NXT UTZ port) and a modified cable. For proper operation one cube must be set as I2C master and the others as I2C slave. The I2C master reads or writes data to the 252 I2C slave registers.

The address of the I2C slave device can be selected in the range 8–119. Reserved I2C addresses are rejected.

Initialization: `robot.init_i2c_slave(address)`
Deinitialization: `robot.deinit_i2c_slave()`
Access: `robot.i2c_slave`

```
read(mem_add)
```

Reads the register.

Parameters: `mem_add` (0-251) – Address of the register to read.
Returns: Data from the register.
Type: int

```
read_all()
```

Reads addresses 0–254. Addresses 0–251 are user registers and addresses 252–254 belong to the time-trigger area.

Returns: Data from all available registers.
Type: bytes

```
write(mem_add, data)
```

Writes to the register.

Parameters: `mem_add` (0-251) – Address of the register to write to.
 `data` (0-255) – Data to write.

```
read_flags()
```

Return four bit fields indicating which registers were read by the I2C master.

```
written_flags()
```

Return four bit fields indicating which registers were written by the I2C master.

```
irq(handler=None, trigger=0, hard=False)
```

Sets the interrupt handler function.

Parameters: **handler** (callable) – Callback function, must have one input argument for the I2C_slave instance.

trigger (0-7) – Event mask at which the handler function is called.

robot.i2c_slave.RECEIVED – I2C master wrote data to the register.

robot.i2c_slave.READ – I2C master read data from the register.

robot.i2c_slave.TRIGGER – I2C master trigger.

```
1 # Initialize the cube as an I2C slave with address 10
2 robot.init_i2c_slave(10)
3
4 # Callback function for responding to events
5 def slave_callback(i2c_slave):
6     flags = i2c_slave.irq().flags()
7     if (flags & i2c_slave.RECEIVED):
8         print("new data received")
9     if (flags & i2c_slave.READ):
10        print("master read data")
11    if (flags & i2c_slave.TRIGGER):
12        robot.led.on()
13
14 # Set interrupt handler for all three possible events
15 flags = robot.i2c_slave.RECEIVED | robot.i2c_slave.READ | robot.
16        i2c_slave.TRIGGER
17 robot.i2c_slave.irq(handler=slave_callback, trigger=flags)
18
19 # Write and read the character "b" (98) to register 2
20 robot.i2c_slave.write(2, 98)
21 data_received = robot.i2c_slave.read(2)
22 print(chr(data_received)) # "b"
23
24 # Deinitialize I2C slave
25 robot.deinit_i2c_slave()
```

Code 9: Using the cube for multi-cube communication as an I2C slave.

3.9 ESP32 communication

The ESP32 communicates with the cube RP2040 over an internal serial link. It operates in one of two mutually exclusive modes: Bluetooth SPP mode exposes a virtual serial connection to a computer or mobile phone, while Wi-Fi access point mode provides a captive-portal web interface with buttons, switches, indicators and numeric values. Entering Wi-Fi mode stops Bluetooth SPP; returning to Bluetooth mode stops the web server and disconnects the access point. The ESP32 can be programmed through the USB connector on top of the cube. Its default firmware is available from the [Open-Cube releases page](https://gitlab.fel.cvut.cz/open-cube/firmware/-/releases) [<https://gitlab.fel.cvut.cz/open-cube/firmware/-/releases>].

3.9.1 Bluetooth ESP32 connection with external device

Communication is serial via Bluetooth virtual COM port and has been tested for Windows, Ubuntu, and Android. To pair, the cube must be turned on in the BT pair menu. The device name will appear on the display.

- Windows:
 1. In the Bluetooth settings, select Add new device, then Bluetooth, and pair the computer with the cube. See figure 2.
 2. Confirm the PIN on the cube and on the computer.
 3. To find the COM port number, go to the advanced Bluetooth settings. The required COM port has a name containing ESP32SPP and the direction Outgoing. See figure 3. This COM port is used for both receiving and sending data.
- Ubuntu:
 1. In the Bluetooth settings, select the Open-Cube device and pair the computer with the cube.
 2. Confirm the PIN on the cube and on the computer.
 3. Display the device details and copy its MAC address.
 4. Create a serial port link: `sudo rfcomm bind /dev/rfcomm0 XX:XX:XX:XX:XX:XX`.
 5. Open the serial port, for example, with the command `screen /dev/rfcomm0`
- Android:
 1. In the Bluetooth settings, select the Open-Cube device and pair your phone with the cube.
 2. Confirm the PIN on the cube and on your phone.

3.9.2 Cube

```
class Esp()
```

Bidirectional communication between the RP2040 and ESP32.

ESP can operate in two modes – Bluetooth and Wi-Fi access point.

1. In Bluetooth mode, data are sent as ASCII or binary messages.

ASCII mode: An incoming message is completed by an LF or CR terminator. The terminator is removed from the value returned by `bt_read()`. The `bt_write()` method sends exactly the supplied string and does not add a terminator.

Binary mode: The receiver searches for a configured header and reads the specified number of payload bytes. Variable-size binary mode is selected with `data_size=0`; an external sender must then transmit the header, a two-byte little-endian payload length and the payload. A variable header contains 1–8

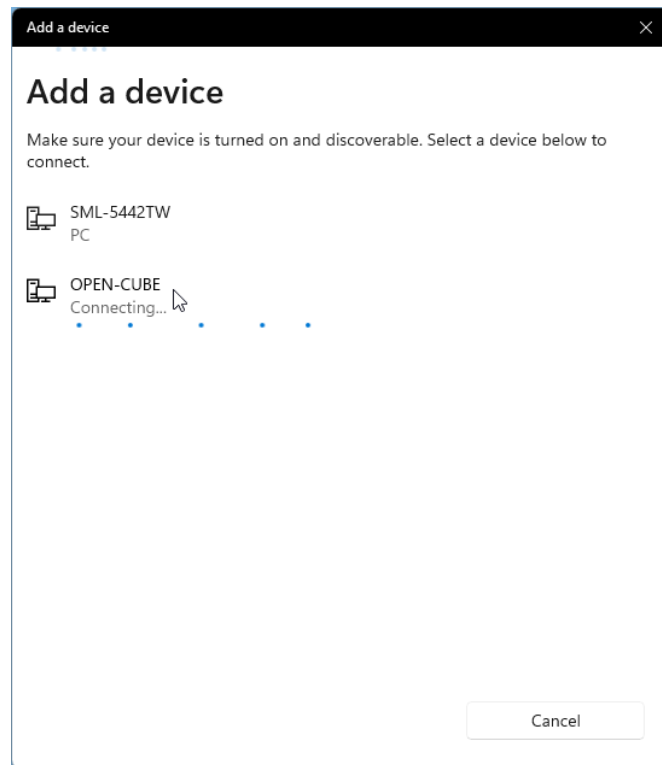


Figure 2: Pairing with cube on Windows.

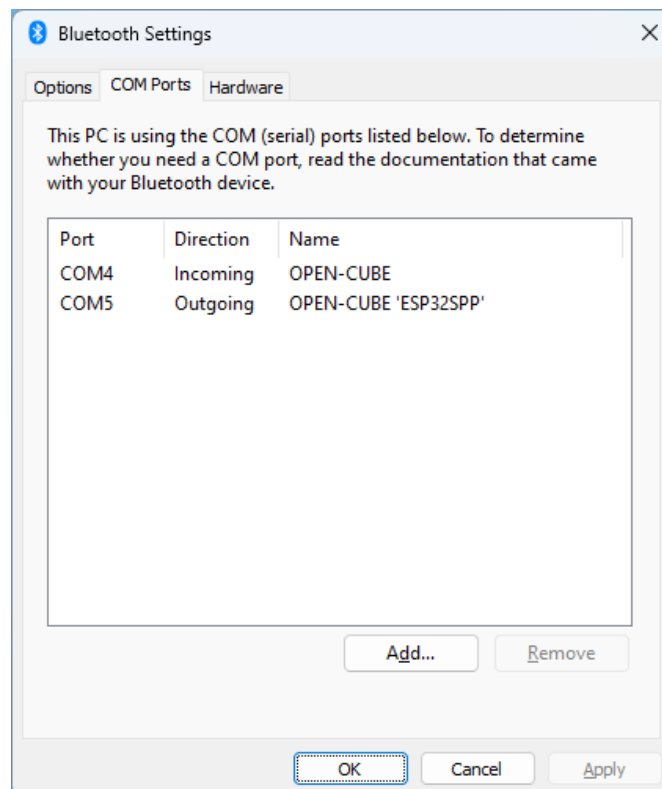


Figure 3: Determining the COM port number

bytes and a payload contains at most 1024 bytes. The cube's `bt_write()` method prepends the configured header but does not add the variable-size

length field.

2. Wi-Fi access point mode provides 9 buttons, 5 switches, 5 indicators and 6 numeric displays. After connecting to its network, the control page normally opens as a captive portal. It can also be opened manually at <http://192.168.4.1/>.

Initialization: Turning on the cube.

Deinitialization: Turning off the cube.

Access: `robot.esp`

reset(*timeout=2000*)

Reset the ESP, set Bluetooth mode and return True after acknowledgement or False after timeout.

reset_non_blocking()

Start an ESP reset without waiting for acknowledgement.

bt()

Switch from Wi-Fi access point mode to Bluetooth SPP mode. If Bluetooth mode is already active, the method performs no change.

wifi()

Set Wi-Fi access point mode. The default ESP address is 192.168.4.1. Indexing of configurable web server controls is shown in image 4.

set_name(*name*)

Set the name used for Bluetooth and the Wi-Fi SSID. Normal cube startup applies the name selected in Settings, initially "Open-Cube01".

Parameters: **name** (str) – ESP name.

set_password(*password*)

Set the ESP Wi-Fi password. Use 8–63 ASCII characters. The default password is "12345678".

Parameters: **password** (str) – ESP password.

bt_read()

Return last received message in Bluetooth mode.

Returns: Received message. If no new message available: None.

Type:

- ASCII mode: str
- Binary mode: bytes

bt_write(*message*)

Send message in Bluetooth mode.

Parameters: **message** (str/bytes) – Text or binary data to send.

bt_message_ready()

Return True if a complete Bluetooth message is available.

bt_set_binary(*header=None, data_size=0*)

Set Bluetooth mode to binary mode if header is specified and to ASCII mode if header is None.

Parameters: **header** (tuple) – Header containing numbers 0–255, or None.

data_size (int) – Received message size in bytes.

bt_set_callback(*callback=None*)

Set a function called as **callback(esp)** when a complete Bluetooth message is received.

bt_reset()

Clear received Bluetooth data and return to ASCII mode.

wifi_get_buttons()

Return state of 9 web server buttons in Wi-Fi access point mode.

Returns: Tuple of buttons state..

Type: 9x bool

wifi_get_switches()

Return state of 5 web server switches in Wi-Fi access point mode.

Returns: Tuple of switches state.

Type: 5x bool

wifi_set_indicators(*indicators*)

Set state of 5 web server indicators in Wi-Fi access point mode.

Parameters: **indicators** (tuple) – 5x bool of indicators state.

wifi_set_numbers(*numbers*)

Set state of 6 web server numbers in Wi-Fi access point mode.

Parameters: **numbers** (tuple) – 6x float.

wifi_set_buttons_labels(*labels*)

Set label of 9 web server buttons in Wi-Fi access point mode. Each label can have a maximum length of 9 ASCII characters and an empty string hides the button.

Parameters: `labels` (tuple) – 9x str buttons label.

`wifi_set_switches_labels(labels)`

Set label of 5 web server switches in Wi-Fi access point mode. Each label can have a maximum length of 9 ASCII characters and an empty string hides the switch.

Parameters: `labels` (tuple) – 5x str switches label.

`wifi_set_indicators_labels(labels)`

Set label of 5 web server indicators in Wi-Fi access point mode. Each label can have a maximum length of 9 ASCII characters and an empty string hides the indicator.

Parameters: `labels` (tuple) – 5x str indicators label.

`wifi_set_numbers_labels(labels)`

Set label of 6 web server numbers in Wi-Fi access point mode. Each label can have a maximum length of 9 ASCII characters and an empty string hides the number.

Parameters: `labels` (tuple) – 6x str numbers label.

Example of using communication between the micro controller and ESP32:

```
1 import struct
2 from time import sleep_ms
3
4 # Send text to an external Bluetooth device
5 robot.esp.bt_write("test message\n")
6
7 # Wait for the external device to send a response
8 while not robot.esp.bt_message_ready():
9     sleep_ms(10)
10 received_message = robot.esp.bt_read()
11
12 # Send an 8-byte binary payload to the external device
13 robot.esp.bt_set_binary((112, 241, 3, 62), 8)
14 write_message = struct.pack("<ff", 521.9, -11.821)
15 robot.esp.bt_write(write_message)
16
17 # Wait for a framed binary response from the external device
18 while not robot.esp.bt_message_ready():
19     sleep_ms(10)
20 received_message2 = robot.esp.bt_read()
21 value1, value2 = struct.unpack("<ff", received_message2)
22
23 # Switch ESP to Wi-Fi access point mode
24 robot.esp.wifi()
25 # Set labels of indicators
26 robot.esp.wifi_set_indicators_labels(("1", "2", "3", "4", "5"))
27 # Get state of switches
28 switches = robot.esp.wifi_get_switches()
```

Code 10: Communication between the micro controller and ESP32.

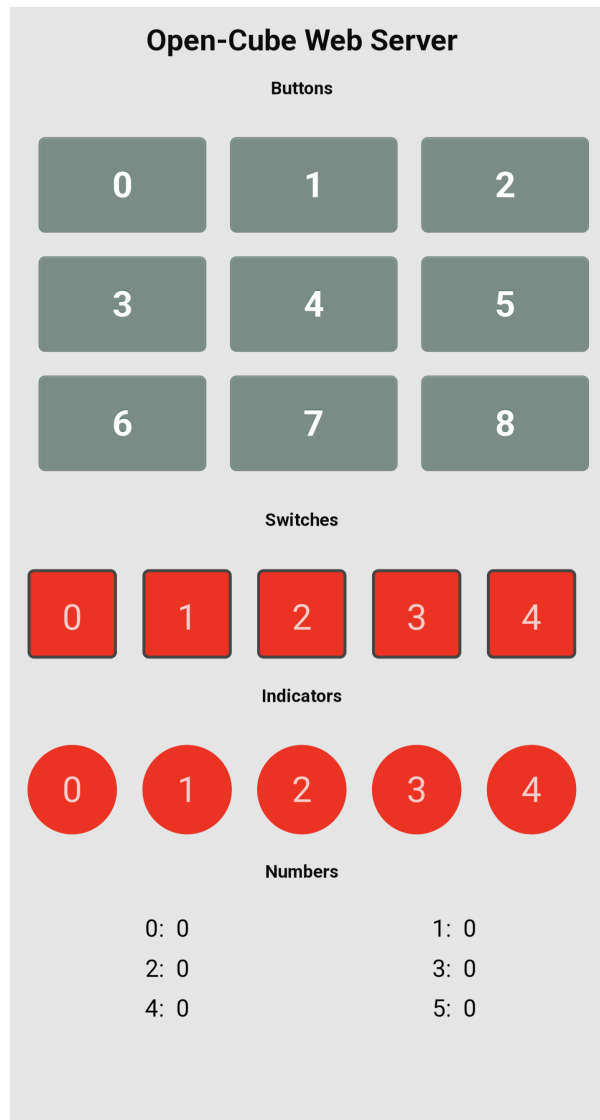


Figure 4: Indexing of ESP web server elements.

3.9.3 Remote control of the cube

When the cube framework is running in Bluetooth mode, line-oriented commands terminated by LF or CR can control the menu, inspect the cube and manage programs.

Control messages:

- `oc u`, `oc d`, `oc l`, `oc r`, `oc k` – emulate the UP, DOWN, LEFT, RIGHT and OK buttons,
- `oc ko` – emulate the power button and turn the cube off,
- `oc ping` – respond with `oc pong`,
- `oc fw` – enter the RP2040 USB bootloader,
- `oc rs` – reset the RP2040,
- `oc vfw` – return the firmware date and commit,

- `oc bat` – return battery voltage and percentage,
- `oc nam` – return the configured cube name,
- `oc mst` – return the current menu page, selected item and program state,
- `oc ls` – return a comma-separated list of programs in `/programs`,
- `oc del <name>` – delete a program file or directory and return `oc del ok` or `oc del ko`,
- `oc run <name>` – start a program without monitoring its output,
- `oc rtc <name>` – start a program in take-control mode so its output and completion can be monitored,
- `oc stp` – stop a managed program by raising `KeyboardInterrupt`,
- `oc prg <filename>` – start the binary upload protocol used by the Thonny plugin and PC utility.

The cube sends `oc menu` when Bluetooth menu control starts or a managed program returns to the menu. Program names are restricted to a final file or directory name under `/programs`; empty names, `.`, `..` and names containing a colon are rejected.

3.9.4 Serial Bluetooth Terminal

For a simple Bluetooth communication from your mobile phone you can use the app [Serial Bluetooth Terminal](https://play.google.com/store/apps/details?id=de.kai_morich.serial_bluetooth_terminal) [https://play.google.com/store/apps/details?id=de.kai_morich.serial_bluetooth_terminal]. This app allows you to send and receive messages from the cube in Bluetooth ASCII mode. Line-oriented messages and commands must end with LF or CR.

3.9.5 Matlab

To communicate with the cube using Matlab, it is necessary to have the Communications Toolbox installed. This interface uses Bluetooth SPP (Serial Port Profile) and allows you to receive and send ASCII and binary data. When sending ASCII data, the message is terminated with a terminator (LF or CR). In the case of binary data, the message is not terminated with a terminator and both the cube and the Matlab program must know the length of the message in advance in order to interpret messages correctly.

An example of using the interface is described in the following code. For more information on how to use the interface, see the official [Bluetooth Communication documentation](https://www.mathworks.com/help/matlab/ref/bluetooth.html) [<https://www.mathworks.com/help/matlab/ref/bluetooth.html>].

```
1 % Connecting the cube with Matlab
2 cube = bluetooth('Open-Cube01')
3 % Set LF (new line) terminator for sending and receiving
  ASCII data
4 configureTerminator(cube, 'LF')
5 %%
```

```

6 % Send a test message to the cube in Bluetooth ASCII mode.
  The message is automatically terminated by the set
  terminator.
7 writeline(cube, 'test message')
8 % Receiving a message from the cube. The function receives
  data and waits for termination by previously set
  terminator.
9 cube_message = readline(cube)
10 %%
11 % The cube is configured with the same four-byte header and
  an eight-byte payload.
12 header = uint8([112 241 3 62]);
13 payload = typecast(single([521.9 -11.821]), 'uint8');
14 write(cube, [header payload], 'uint8')
15
16 % Read a framed response and convert its payload to two
  single values.
17 frame = read(cube, 12, 'uint8');
18 if isequal(frame(1:4), header)
19     cube_values = typecast(uint8(frame(5:12)), 'single');
20 end

```

Code 11: Communication with the cube using Matlab.

3.9.6 Simulink

For communication using Simulink it is necessary to have the Instrument Control Toolbox installed. This toolbox provides blocks **Serial Configuration**, **Serial Send** and **Serial Receive** for serial communication. When creating a new model, you need to modify the simulation solver settings and set the serial communication parameters as follows:

1. Click the button in the bottom right corner to set the solver to **Fixed-Step**, **discrete** and **Fixed-step size** to a value according to the frequency of sending and receiving data (e.g. 0.001). See image 5.
2. Click on the arrow below the **Run** button to enable **Simulation Pacing**. See images 6 and 7.
3. Create the **Serial Configuration** block in the model, in which set the COM port found in 3.9.1 and other parameters according to image 8.

After this configuration it is possible to use the blocks **Serial Send** and **Serial Receive** for sending and receiving data. For additional information on how to use the blocks, see the [official documentation](https://www.mathworks.com/help/instrument/direct-interface-communication-in-simulink.html) [<https://www.mathworks.com/help/instrument/direct-interface-communication-in-simulink.html>].

An example of using serial communication to set the PID constants of a controller running in the cube and display the current PID components is shown in image 9. In this example, the values are sent and received in Bluetooth binary mode, the header for sending and receiving is set, and used data type is single (4 bytes). Messages are received

every 0.01 seconds, which is set in the `Serial Send` block in the `Block sample time` cell. Messages are sent every second, which is set in the `Block sample time` cell of the three `Constant` blocks. Set `Fixed-step size` of the solver should be smaller than the `Block sample time`.

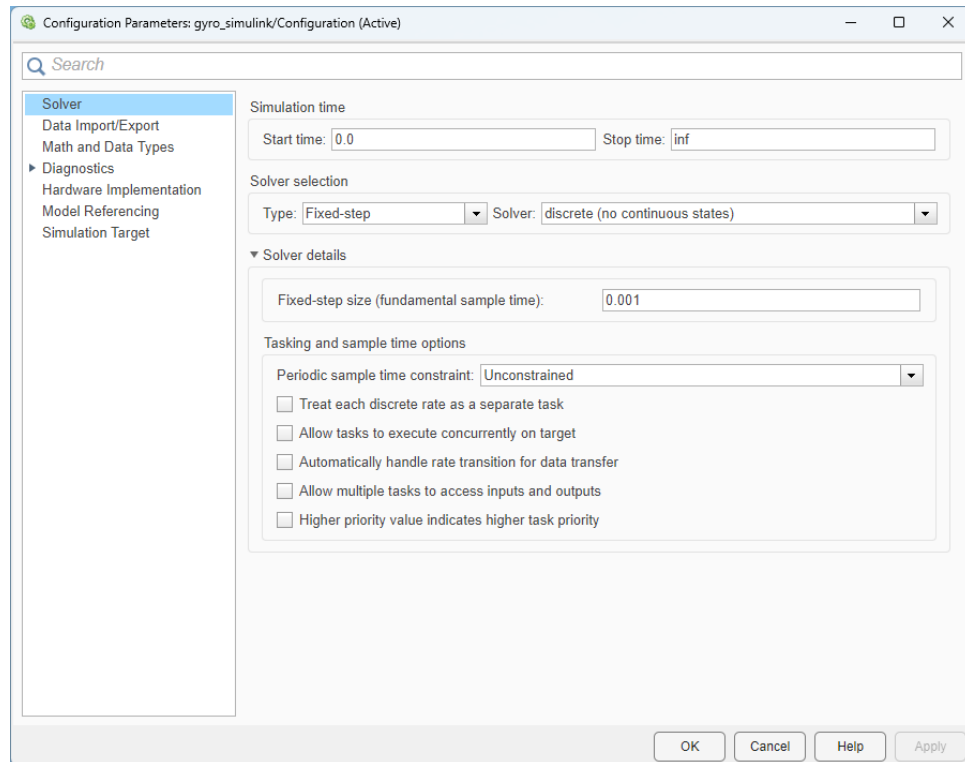


Figure 5: Simulink solver configuration.

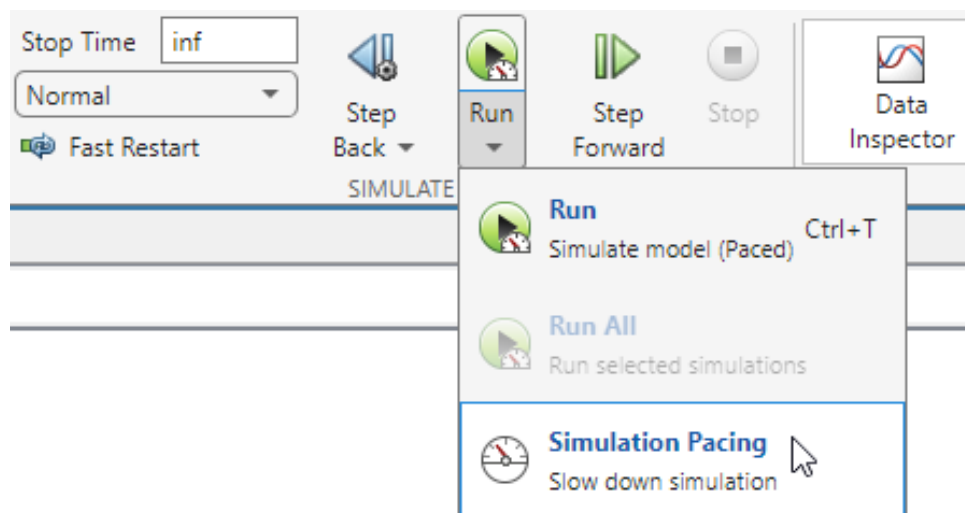


Figure 6: Location of Simulation pacing configuration in Simulink.

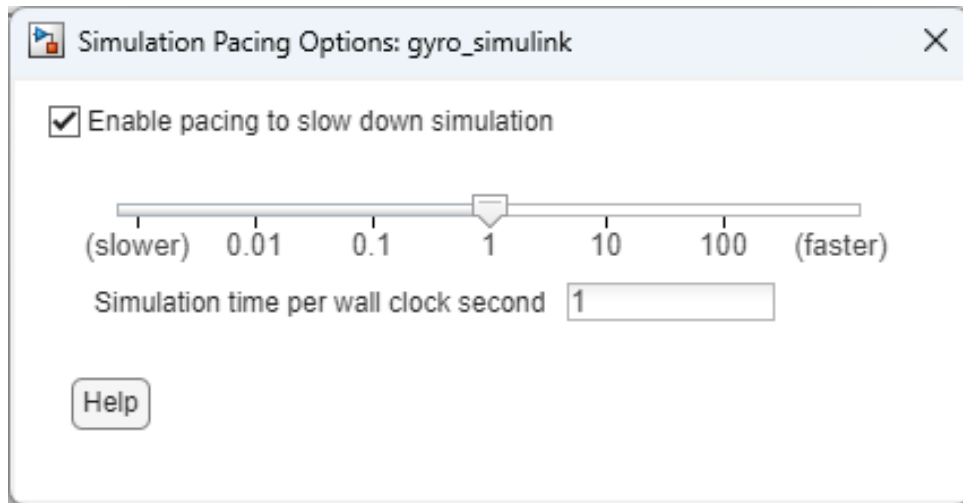


Figure 7: Simulink Simulation Pacing configuration.

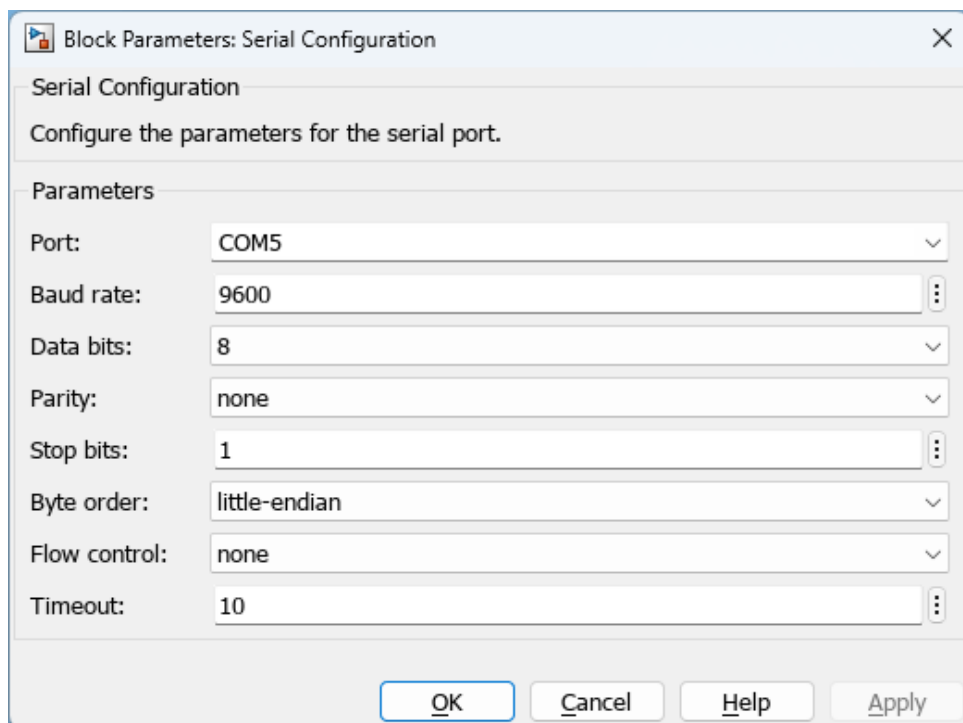


Figure 8: Configuration of Serial Configuration block in Simulink.

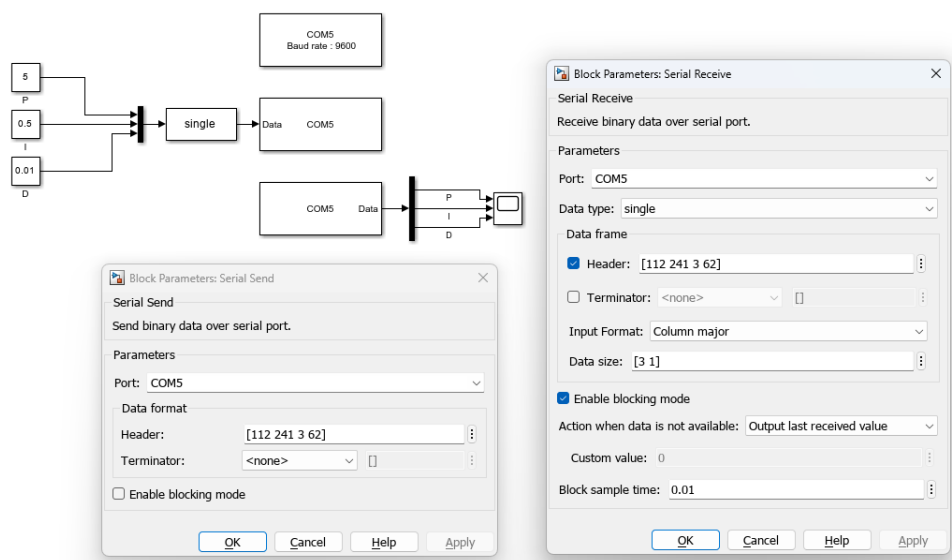


Figure 9: Example of using serial communication in Simulink.

4 NXT sensors

4.1 Touch sensor

```
class TouchSensor(port)
```

NXT Touch Sensor.

Information about the button state.

Parameters: **port** (**Port**) – The port to which the sensor is connected.
Initialization: `robot.init_sensor(sensor_type=Sensor.NXT_TOUCH, -
port=Port).`
Deinitialization: `robot.deinit_sensor(sensor_type=Sensor.NXT_TOUCH, -
port=Port).`
Access: `robot.sensors.touch[Port].`

```
pressed()
```

Return button state.

Returns: True if the button is pressed, False if not.

Type: bool

```
1 # Import sensor and port constants
2 from lib.robot_consts import Sensor, Port
3
4 # Initialize NXT Touch Sensor on sensor port 1
5 robot.init_sensor(sensor_type=Sensor.NXT_TOUCH, port=Port.S1)
6
7 # Determine the button state
8 touch_pressed = robot.sensors.touch[Port.S1].pressed()
```

Code 12: NXT Touch Sensor usage.

4.2 Light sensor

```
class LightSensor(port)
```

NXT Light Sensor.

Measuring light intensity.

Measurement is possible in two modes – manual, in which the intensity is measured on request, and continuous, in which the intensity is measured periodically and on request the last measured value is returned. The continuous mode also provides non-flashing and flashing modes. In the non-flashing mode, the intensity is measured in the LED state previously set by the user. In the flashing mode, the intensity is measured in both the off and on LED states. Changing the LED state is done automatically and the last measured intensity for both LED states is available to the user on request.

The ADC samples the signal with a period of 1 ms. Light intensity is obtained by measuring the voltage on the sensor. When the LED state changes, this voltage will stabilize in approximately 10 ms.

Parameters: `port` (`Port`) – The port to which the sensor is connected.
Initialization: `robot.init_sensor(sensor_type=Sensor.NXT_LIGHT,port=Port)`
Deinitialization: `robot.deinit_sensor(sensor_type=Sensor.NXT_LIGHT,port=Port)`
Access: `robot.sensors.light[Port]`

`on()`

Turn on sensor LED.

`off()`

Turn off sensor LED.

`toggle()`

Change state of sensor LED.

`intensity(pin_on=True)`

Measure and return light intensity in manual mode. Return last measured light intensity in continuous mode.

Parameters: `pin_on` (bool) – Selection of the returned measured light intensity. The value does not matter in manual mode and in continuous non-flashing mode. In continuous flashing mode, return the measured intensity when the LED is on for True and when the LED is off for False.

Returns: Measured light intensity. 0 for the highest intensity, 32768 for the lowest.

Type: int (0–32768)

`get_value(pin_on)`

Return the last measured value for the selected LED state.

`new_data(led_on)`

Return 1 if a new value is available for the selected LED state, otherwise 0.

`set_continuous(period_us, wait_us)`

Set sensor to continuous mode.

- Parameters:**
- **period_us** (int) – Time in μs between the start of the analog value conversion and the reading of the converted digital value from the ADC. Recommended minimum value is 1100.
 - **wait_us** (int) – Waiting time in μs before starting a measurement after LED state change. Recommended minimum value is 10000.

stop_continuous()

Set sensor to manual mode.

set_switching()

Set sensor in continuous mode to continuous flashing mode.

stop_switching()

Set sensor in continuous mode to continuous non-flashing mode.

```

1 # Import sensor and port constants
2 from lib.robot_consts import Sensor, Port
3
4 # Initialize NXT Light Sensor on sensor port 1
5 robot.init_sensor(sensor_type=Sensor.NXT_LIGHT, port=Port.S1)
6
7 # Turn on sensor LED
8 robot.sensors.light[Port.S1].on()
9 # Measure light intensity
10 light_intensity = robot.sensors.light[Port.S1].intensity()
```

Code 13: NXT Light Sensor usage.

4.3 Ultrasonic sensor

class UltrasonicSensor()

NXT Ultrasonic Sensor.

Measuring distance of an object from the sensor in the range 0–255 cm with an accuracy of ± 3 cm. The sensor must be connected to the cube port marked I2C and only one NXT Ultrasonic Sensor can be connected. This connector cannot be used for multi-cube I2C master or slave communication at the same time.

Initialization: `robot.init_sensor(sensor_type=Sensor.NXT_ULTRASONIC)`

Deinitialization: `robot.deinit_sensor(sensor_type=Sensor.NXT_ULTRASONIC)`

Access: `robot.sensors.ultra_nxt`

distance()

Measure and return distance.

Returns: Distance of the nearest object.
Type: int (0–255: cm)

```
1 # Import sensor constants
2 from lib.robot_consts import Sensor
3
4 # Initialize NXT Ultrasonic Sensor on the I2C port
5 robot.init_sensor(sensor_type=Sensor.NXT_ULTRASONIC)
6 # Measure distance
7 distance = robot.sensors.ultra_nxt.distance()
```

Code 14: NXT Ultrasonic Sensor usage.

4.4 Sound sensor

```
class SoundSensor(port)
```

NXT Sound Sensor.

Measuring ambient sound intensity.

Parameters: port (Port) – The port to which the sensor is connected.
Initialization: robot.init_sensor(sensor_type=Sensor.NXT_SOUND, port=Port)
Deinitialization: robot.deinit_sensor(sensor_type=Sensor.NXT_SOUND, port=Port)
Access: robot.sensors.sound[Port]

```
intensity()
```

Measure and return sound intensity.

Returns: Sound intensity. 0 for the highest intensity, 32768 for the lowest.
Type: int (0–32768)

```
1 # Import sensor and port constants
2 from lib.robot_consts import Sensor, Port
3
4 # Initialize NXT Sound Sensor on sensor port 1
5 robot.init_sensor(sensor_type=Sensor.NXT_SOUND, port=Port.S1)
6
7 # Measure sound intenisty
8 sound_intensity = robot.sensors.sound[Port.S1].intensity()
```

Code 15: NXT Sound Sensor usage.

5 EV3 sensors

EV3 UART sensors are detected in the background while the cube menu is open. Initializing a matching sensor reuses the detected connection and avoids repeating the complete handshake. A wrong sensor type, unavailable sensor or port already in use can raise a connection error.

5.1 Touch sensor

class TouchSensor(*port*)

EV3 Touch Sensor.

Information about the button state.

Parameters: **port** (**Port**) – The port to which the sensor is connected.
Initialization: `robot.init_sensor(sensor_type=Sensor.EV3_TOUCH, -
port=Port)`
Deinitialization: `robot.deinit_sensor(sensor_type=Sensor.EV3_TOUCH, -
port=Port)`
Access: `robot.sensors.touch[Port].`

pressed()

Return button state.

Returns: True if the button is pressed, False if not.
Type: bool

```
1 # Import sensor and port constants
2 from lib.robot_consts import Sensor, Port
3
4 # Initialize EV3 Touch Sensor on sensor port 1
5 robot.init_sensor(sensor_type=Sensor.EV3_TOUCH, port=Port.S1)
6
7 # Get button state
8 touch_pressed = robot.sensors.touch[Port.S1].pressed()
```

Code 16: EV3 Touch Sensor usage.

5.2 Color sensor

class ColorSensor(*port*)

EV3 Color Sensor.

Measuring light intensity and color detection.

Parameters: **port** (**Port**) – The port to which the sensor is connected.
Initialization: `robot.init_sensor(sensor_type=Sensor.EV3_COLOR, port=Port)`
Deinitialization: `robot.deinit_sensor(sensor_type=Sensor.EV3_COLOR, port=Port)`
Access: `robot.sensors.light[Port]`

reflection()

Return last measured reflected light intensity. Sensor red LED periodically switches on and off. The raw calculation subtracts the LED-on value from the LED-off value. The resulting value is adjusted using sensor internal calibration and remapped to the 0–100% interval.

Returns: Reflected light intensity.
Type: int (0-100%)

reflection_raw()

Return last measured reflected light intensity in raw format. Sensor red LED periodically switches on and off. The LED-on value is subtracted from the LED-off value and the result is limited to zero or greater.

Returns: Reflected light intensity.
Type: int

ambient()

Return last measured ambient light intensity. The resulting value is adjusted using sensor internal calibration and remapped to the 0-100% interval.

Returns: Ambient light intensity.
Type: int (0-100%)

rgb_raw()

Return last measured reflected light intensity for red, blue and green LEDs turned on and measured separately.

Returns: Tuple of reflected light intensity (red, green, blue).
Type: (int, int, int)

rgb()

Return last measured reflected light intensity for the red, blue and green LEDs turned on and measured separately. The resulting values are adjusted using Open-Cube calibration and remapped to the 0-100% interval.

Returns: Tuple of reflected light intensity (red, green, blue).
Type: (float, float, float) (0–100%)

hsv()

Return HSV values for the last measured reflected light intensity with the red, blue and green LEDs turned on and measured separately.

Returns: Tuple of HSV.
Type: (hue: °, saturation: %, value: %)

color()

Return a constant of sensor detected color. Color constants can be used [Color](#).

Returns: Detected color constant, or None when no color is recognized.

Type: None or int (1–7 – black, blue, green, yellow, red, white, brown)

```

1 # Import sensor constants
2 from lib.robot_consts import Sensor, Port
3
4 # Initialize EV3 Color Sensor on sensor port 1
5 robot.init_sensor(sensor_type=Sensor.EV3_COLOR, port=Port.S1)
6
7 # Measure reflected light intensity
8 reflection = robot.sensors.light[Port.S1].reflection()

```

Code 17: EV3 Color Sensor usage.

5.3 Ultrasonic sensor

class UltrasonicSensor(*port*)

EV3 Ultrasonic Sensor.

Measuring distance of an object from the sensor in range 0–2550 mm with an accuracy of ± 30 mm.

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

Initialization: `robot.init_sensor(sensor_type=Sensor.EV3_ULTRASONIC, port=Port)`

Deinitialization: `robot.deinit_sensor(sensor_type=Sensor.EV3_ULTRASONIC, port=Port)`

Access: `robot.sensors.ultrasonic[Port]`

distance(*silent=False*)

Return last measured distance. If *silent* is True, perform a single measurement and stop continuous ultrasonic transmission afterward.

Returns: Distance of the nearest object. in mm

Type: int (0–2550)

presence()

Detect presence of another ultrasonic sensor.

Returns: True if an ultrasonic sensor detected, False otherwise.

Type: bool

```

1 # Import sensor constants
2 from lib.robot_consts import Sensor, Port
3
4 # Initialize EV3 Ultrasonic Sensor on sensor port 1
5 robot.init_sensor(sensor_type=Sensor.EV3_ULTRASONIC, port=Port.S1)
6
7 # Measure distance
8 distance = robot.sensors.ultrasonic[Port.S1].distance()

```

Code 18: EV3 Ultrasonic sensor usage.

5.4 Infrared sensor

```
class InfraredSensor(port)
```

EV3 Infrared Sensor.

Measuring the distance to the nearest surface and control by remote controller.

Parameters: **port** (**Port**) – The port to which the sensor is connected.
Initialization: `robot.init_sensor(sensor_type=Sensor.EV3_INFRARED, port=Port)`
Deinitialization: `robot.deinit_sensor(sensor_type=Sensor.EV3_INFRARED, port=Port)`
Access: `robot.sensors.infrared[Port]`

```
distance()
```

Measure relative distance to the nearest surface. 100% is approximately 70 cm.

Returns: Relative distance to the nearest surface.
Type: int (0-100%)

```
seeker(channel)
```

Detect presence of a remote controller on given channel.

Parameters: **channel** (int: 1–4) – Remote-controller channel to be checked.
Returns: Tuple of direction (-25 is leftmost, 25 rightmost) and relative distance (100% is approximately 200 cm). Distance is -128 if no beacon is detected.
Type: (int, int) (direction: -25–25, distance: -128 or 0–100%)

```
1 # Import sensor constants
2 from lib.robot_consts import Sensor, Port
3
4 # Initialize EV3 Infrared Sensor on sensor port 1
5 robot.init_sensor(sensor_type=Sensor.EV3_INFRARED, port=Port.S1)
6 # Measure distance
7 distance = robot.sensors.infrared[Port.S1].distance()
```

Code 19: EV3 Infrared Sensor usage.

5.5 Gyroscopic sensor

```
class GyroSensor(port)
```

EV3 Gyro Sensor.

Measuring angle of rotation and angular velocity in one axis.

Parameters: **port** (**Port**) – The port to which the sensor is connected.
Initialization: `robot.init_sensor(sensor_type=Sensor.EV3_GYRO, port=Port)`
Deinitialization: `robot.deinit_sensor(sensor_type=Sensor.EV3_GYRO, port=Port)`
Access: `robot.sensors.gyro[Port]`

angle()

Return angle of rotation of the sensor.

Returns: Angle of rotation.

Type: int (°)

speed()

Return angular velocity of the sensor.

Returns: Angular velocity.

Type: int (°/s)

reset_angle(*angle*)

Reset angle measurement or set a new initial angle value.

Parameters: **angle** (int) – Value that the angle method will return if the sensor does not move. 0 to reset angle.

angle_and_speed()

Return angle of rotation and angular velocity of the sensor.

Returns: Tuple of angle of rotation and angular velocity.

Type: (int, int) (°, °/s)

coarse_speed()

Return angular velocity of the sensor with lower resolution and wider range.

Returns: Angular velocity.

Type: int (°/s)

tilt_speed()

Return angular velocity around the secondary sensor axis in °/s.

tilt_angle()

Return rotation around the secondary sensor axis in degrees. Secondary-axis measurements are not supported by every EV3 gyro revision.

```
1 # Import sensor constants
2 from lib.robot_consts import Sensor, Port
3
4 # Initialize EV3 Gyro Sensor on sensor port 1
5 robot.init_sensor(sensor_type=Sensor.EV3_GYR0, port=Port.S1)
6
7 # Reset angle of rotation
8 robot.sensors.gyro[Port.S1].reset_angle(0)
9
10 # Measure angle of rotation and angular velocity
11 (angle, speed) = robot.sensors.gyro[Port.S1].angle_and_speed()
```

Code 20: EV3 Gyro Sensor usage.

6 Open-Cube sensors

Open-Cube color, laser, ultrasonic and AHRS sensors use the same UART background detection and connection reuse as EV3 sensors. Opening one can adopt a matching detected connection. Managed program cleanup deinitializes sensors and restarts detection. Open-Cube Touch uses the non-UART touch interface.

6.1 Touch sensor

```
class TouchSensor(port)
```

Open-Cube Touch Sensor. The sensor uses the same button interface as the EV3 Touch Sensor.

Parameters: `port` (`Port`) – The port to which the sensor is connected.
Initialization: `robot.init_sensor(sensor_type=Sensor.OC_TOUCH,port=Port)`
Deinitialization: `robot.deinit_sensor(sensor_type=Sensor.OC_TOUCH,port=Port)`
Access: `robot.sensors.touch[Port]`

```
pressed()
```

Return True if the sensor is pressed, otherwise False.

6.2 Color sensor

```
class ColorSensor(port)
```

Open-Cube RGB optical reflective and color sensor.

Measuring light intensity and color detection.

Parameters: `port` (`Port`) – The port to which the sensor is connected.
Initialization: `robot.init_sensor(sensor_type=Sensor.OC_COLOR,port=Port)`
Deinitialization: `robot.deinit_sensor(sensor_type=Sensor.OC_COLOR,port=Port)`
Access: `robot.sensors.light[Port]`

```
reflection()
```

Return last measured reflected light intensity. Sensor red LED periodically switches on and off. Light intensity is measured when the LED is on, from this value a measurement when the LED is off is subtracted. The resulting value is adjusted using sensor internal calibration and remapped to the 0-100% interval.

Returns: Reflected light intensity.
Type: int (0-100%)

```
reflection_raw()
```

Return last measured reflected light intensity in raw format. Sensor red LED periodically switches on and off and measures light intensity for both LED states.

Returns: Tuple of red-LED and LED-off reference intensity.
Type: (int, int)

reflection_raw_green()

Return last measured reflected light intensity in raw format. Sensor green LED periodically switches on and off and measures light intensity for both LED states.

Returns: Tuple of green-LED and LED-off reference intensity.
Type: (int, int)

reflection_raw_blue()

Return last measured reflected light intensity in raw format. Sensor blue LED periodically switches on and off and measures light intensity for both LED states.

Returns: Tuple of blue-LED and LED-off reference intensity.
Type: (int, int)

ambient()

Return last measured ambient light intensity. The resulting value is adjusted using sensor internal calibration and remapped to the 0-100% interval.

Returns: Ambient light intensity.
Type: int (0-100%)

rgb_raw()

Return last measured reflected light intensity for red, blue and green LEDs turned on and measured separately and reference light intensity for all LEDs turned off.

Returns: Tuple of reflected light intensity (red, green, blue) and reference light intensity.
Type: (int, int, int, int)

rgb()

Return last measured reflected light intensity for the red, blue and green LEDs turned on and measured separately. The resulting values are adjusted using Open-Cube calibration and remapped to the 0–100% interval.

Returns: Tuple of reflected light intensity (red, green, blue).
Type: (int, int, int) (0–100%)

hsv()

Return HSV values for the last measured reflected light intensity with the red, blue and green LEDs turned on and measured separately.

Returns: Tuple HSV.
Type: (hue, saturation, value)

color()

Return a constant of sensor detected color. Color constants can be used [Color](#).

Returns: Detected color constant.

Type: int (0–7 – no color, black, blue, green, yellow, red, white, brown)

```
1 # Import sensor constants
2 from lib.robot_consts import Sensor, Port
3
4 # Initialize Open-Cube Color Sensor on sensor port 1
5 robot.init_sensor(sensor_type=Sensor.OC_COLOR, port=Port.S1)
6
7 # Measure reflected light intensity
8 reflection = robot.sensors.light[Port.S1].reflection()
```

Code 21: Open-Cube Color Sensor usage.

6.3 Laser distance sensor

class LaserSensor(*port*)

Open-Cube LIDAR distance sensor.

Measuring distance of an object from the sensor in the range 0–4000 mm. The minimum guaranteed measurement distance is 4 cm. If the object is closer than this distance, the sensor will measure an inaccurate value.

Parameters: **port** ([Port](#)) – The port to which the sensor is connected.

Initialization: `robot.init_sensor(sensor_type=Sensor.OC\_LASER, port=Port)`

Deinitialization: `robot.deinit_sensor(sensor_type=Sensor.OC\_LASER, port=Port)`

Access: `robot.sensors.laser[Port]`

distance()

Return last measured distance.

Returns: Distance of the nearest object.

Type: int (0–4000: mm)

distance_fov()

Return last measured distance. This mode uses a larger sensor field of view.

Returns: Distance of the nearest object.

Type: int (0–4000: mm)

distance_short()

Return last measured distance. This mode is more accurate for shorter distances (up to 1300 mm) and strong ambient light.

Returns: Distance of the nearest object.

Type: int (0–1300: mm)

set_led_distance(*distance*)

Set threshold distance at which the green LED of the sensor lights up.

Parameters: `distance` (int) – Threshold distance in mm.

```
1 # Import sensor constants
2 from lib.robot_consts import Sensor, Port
3
4 # Initialize Open-Cube LIDAR on sensor port 1
5 robot.init_sensor(sensor_type=Sensor.OC_LASER, port=Port.S1)
6
7 # Measure distance
8 distance = robot.sensors.laser[Port.S1].distance()
```

Code 22: Open-Cube LIDAR usage.

6.4 Ultrasonic sensor

`class UltrasonicSensor(port)`

Open-Cube Ultrasonic Sensor.

Measuring distance of an object from the sensor in the range approximately 50–9999 mm.

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

Initialization: `robot.init_sensor(sensor_type=Sensor.OC_ULTRASONIC, port=Port)`

Deinitialization: `robot.deinit_sensor(sensor_type=Sensor.OC_ULTRASONIC, port=Port)`

Access: `robot.sensors.ultrasonic[Port]`

`distance()`

Return last measured distance.

Returns: Distance of the nearest object.

Type: int (approximately 50–9999: mm)

`set_led_distance(distance)`

Set threshold distance at which the green LED of the sensor lights up.

Parameters: `distance` (int) – Threshold distance in mm.

```
1 # Import sensor constants
2 from lib.robot_consts import Sensor, Port
3
4 # Initialize Open-Cube Ultrasonic Sensor on sensor port 1
5 robot.init_sensor(sensor_type=Sensor.OC_ULTRASONIC, port=Port.S1)
6
7 # Measure distance
8 distance = robot.sensors.ultrasonic[Port.S1].distance()
```

Code 23: Open-Cube Ultrasonic sensor usage.

6.5 AHRS sensor

```
class GyroSensor(port)
```

Open-Cube AHRS (Attitude and Heading Reference System) Sensor.

Measurement of angular velocity, acceleration and magnetic field intensity in three axes.

Parameters: `port` (`Port`) – The port to which the sensor is connected.
Initialization: `robot.init_sensor(sensor_type=Sensor.OC_GYRO,port=Port)`
Deinitialization: `robot.deinit_sensor(sensor_type=Sensor.OC_GYRO,port=Port)`
Access: `robot.sensors.gyro[Port]`

```
euler_angles()
```

Returns the current sensor orientation in Euler angles calculated from the gyroscope and accelerometer.

Returns: Tuple of Euler angles yaw, pitch, roll.
Type: (yaw: -180–180°, pitch: -90–90°, roll: -180–180°)

```
reset_angles()
```

Reset the current Euler-angle or quaternion orientation.

```
euler_angles_mag()
```

Returns the current sensor orientation in Euler angles calculated from the gyroscope, accelerometer and magnetometer. If the optional magnetometer is unavailable, the sensor remains usable and this mode falls back to gyroscope and accelerometer fusion.

Returns: Tuple of Euler angles yaw, pitch, roll.
Type: (yaw: -180–180°, pitch: -90–90°, roll: -180–180°)

```
quaternions()
```

Returns the current sensor orientation in quaternions calculated from the gyroscope and accelerometer.

Returns: Tuple of quaternions.
Type: (w, x, y, z), components from -1.0–1.0

```
raw()
```

Return three nested tuples containing angular velocity, acceleration and magnetic field values in the x, y, z axes.

Returns: Tuple containing gyroscope, accelerometer and magnetometer tuples.
Type: ((g_x, g_y, g_z): °/s, (a_x, a_y, a_z): g, (m_x, m_y, m_z): raw units)

`calibrate_gyro()`

Initiates gyro offset calibration. The sensor must not move during the entire calibration period.

Returns: 1 if calibration is completed.

Type: int (0/1)

`calibrate_mag()`

Initiates a hard iron calibration of the magnetometer. Rotate the sensor through all orientations until calibration is complete. Calibration cannot complete if the magnetometer is unavailable.

Returns: 1 if calibration is completed.

Type: int (0/1)

```
1 # Import sensor constants
2 from lib.robot_consts import Sensor, Port
3
4 # Initialize the OC AHRS sensor on sensor port1
5 robot.init_sensor(sensor_type=Sensor.OC_GYRO, port=Port.S1)
6
7 # Gyroscope calibration
8 calibrated = 0
9 while not calibrated:
10     calibrated = robot.sensors.gyro[Port.S1].calibrate_gyro()
11
12 # Get sensor orientation
13 euler_angles = robot.sensors.gyro[Port.S1].euler_angles()
14 yaw, pitch, roll = euler_angles[0], euler_angles[1], euler_angles[2]
```

Code 24: Open-Cube AHRS sensor usage.

7 Motors

The motors from the NXT and EV3 sets are identical internally, differing only in appearance and plastic construction.

```
class Motor(port)
```

Rotary encoders are located on the motors, which allow detecting the motor rotational position and approximating the speed of rotation.

Provides three modes of control – power, position, speed.

In the power control mode, the user directly sets the desired power on the motor in the range of -100–100%.

In the position control mode, the user selects the desired motor rotational position in °. The control is mediated by a PID controller with adjustable constants.

In the speed control mode, the user selects the desired motor speed of rotation in °/s. The maximum speed when the motor is unloaded is approximately 950 °/s. The control is mediated by a PID controller with adjustable constants.

Parameters: **port** (*Port*) – The port to which the motor is connected.

Initialization: `robot.init_motor(port=Port)`

Deinitialization: `robot.deinit_motor(port=Port)`

Access: `robot.motors[Port]`

Parameters: **port** (*Port*) – The port to which the motor is connected.

```
set_power(power)
```

Set power on the motor.

Parameters: **power** (float) – Desired power on the motor in the range of -100–100%. If a value is entered outside this range, the limit value is set.

```
init_encoder()
```

Initialize encoders with a measurement of the motor position and speed on all motors.

```
deinit_encoder()
```

Deinitialize encoders with a measurement of the motor position and speed on all motors.

```
reset_encoder()
```

Reset the selected motor encoder position to zero.

position()

Return motor rotational position.

Returns: Motor rotational position.

Type: int: °

speed()

Return motor speed of rotation.

Returns: Motor speed of rotation.

Type: int: °/s

init_regulator()

Initialize motor PID controller.

deinit_regulator()

Deinitialize motor PID controller.

set_regulator_position(*position*)

Set motor to position control mode.

Parameters: **position** (int) – Required motor position in °.

regulator_pos_set_consts(*p, i, d*)

Set PID constants of the position controller. The default values are 0.9, 0.1 and 0.

Parameters:

- **p** (float) – Constant of the proportional component.
- **i** (float) – Constant of the integration component.
- **d** (float) – Constant of the derivative component.

set_regulator_speed(*speed*)

Set motor to speed control mode.

Parameters: **speed** (int) – Required motor speed of rotation in °/s.

regulator_speed_set_consts(*p, i, d*)

Set PID constants of the speed controller. The default values are 0.1, 0.8 and 0.

Parameters:

- **p** (float) – Constant of the proportional component.
- **i** (float) – Constant of the integration component.
- **d** (float) – Constant of the derivative component.

regulator_error()

Return current control error.

Returns: Control error.

Type:

- **position control mode** – position: °
- **speed control mode** – angular velocity: °/s

regulator_power()

Return current control action.

Returns: Control action.

Type: power (-100–100%)

regulator_speed()

Return the filtered speed used by the speed regulator in °/s.

```
1 # Import port constants
2 from lib.robot_consts import Port
3
4 # Initialize motors on motor ports 1 and 2
5 robot.init_motor(Port.M1)
6 robot.init_motor(Port.M2)
7
8 # Set power on the motors to 50%, and 20% in the opposite direction of
  rotation
9 robot.motors[Port.M1].set_power(50)
10 robot.motors[Port.M2].set_power(-20)
11
12 # Initialize motors encoders
13 robot.motors[Port.M1].init_encoder()
14 robot.motors[Port.M2].init_encoder()
15
16 # Measure current position and speed of rotation of motors
17 pos1 = robot.motors[Port.M1].position()
18 pos2 = robot.motors[Port.M2].position()
19 speed1 = robot.motors[Port.M1].speed()
20 speed2 = robot.motors[Port.M2].speed()
```

Code 25: Motor usage.

8 Parameters and constants

Parameters and constants importable from `lib.robot_consts`.

8.1 Sensor

class **Sensor**

Sensor types.

NO_SENSOR

No sensor.

NXT_LIGHT

NXT Light Sensor.

NXT_ULTRASONIC

NXT Ultrasonic Sensor.

NXT_TOUCH

NXT Touch Sensor.

NXT_SOUND

NXT Sound Sensor.

GYRO_ACC

Integrated ICM42688 gyroscope and accelerometer.

EV3_COLOR

EV3 Color Sensor.

EV3_GYRO

EV3 Gyroscopic Sensor.

EV3_INFRARED

EV3 Infrared Sensor.

EV3_TOUCH

EV3 Touch Sensor.

EV3_ULTRASONIC

EV3 Ultrasonic Sensor.

OC_LASER

Open-Cube LIDAR.

OC_COLOR

Open-Cube Color Sensor.

OC_ULTRASONIC

Open-Cube Ultrasonic Sensor.

OC_GYRO

Open-Cube AHRS Sensor.

OC_TOUCH

Open-Cube Touch Sensor.

8.2 Port

class **Port**

Motor and sensor port of the cube.

M1

Motor port 1.

M2

Motor port 2.

M3

Motor port 3.

M4

Motor port 4.

M_COUNT

Number of motor ports.

S1

Sensor port 1.

S2

Sensor port 2.

S3

Sensor port 3.

S4

Sensor port 4.

S_COUNT

Number of sensor ports.

8.3 Button

class Button

Cube buttons.

POWER

LEFT

RIGHT

OK

UP

DOWN

8.4 Light

class Light

Measured value when the NXT light sensor LED is off and on.

OFF

Measured value when the NXT light sensor LED is off.

ON

Measured value when the NXT light sensor LED is on.

8.5 GyroAcc

class GyroAcc

Measured value of built-in gyroscope and accelerometer ICM42688 in x, y, z axes and sensor interrupt pin.

AX

Acceleration in the x-axis

AY

Acceleration in the y-axis

AZ

Acceleration in the z-axis

GX

Angular velocity in the x-axis.

GY

Angular velocity in the y-axis.

GZ

Angular velocity in the z-axis.

IRQ_PIN

Pin of the micro controller to which the ICM42688 is connected.

8.6 Color

class **Color**

Colors detected by the EV3 and Open-Cube Color Sensors. Open-Cube Color can return `NO_COLOR`; the EV3 Color high-level method returns `None` when no color is recognized.

NO_COLOR

No color detected.

BLACK

BLUE

GREEN

YELLOW

RED

WHITE

BROWN

8.7 Note

class **Note**

Frequencies used for buzzer melodies. The constants `C4-B4`, `C5-B5` and `C6-B6` contain note frequencies in Hz. `REST` has a frequency of zero.

8.8 Melody

class **Melody**

Predefined sequences accepted by `robot.buzzer.play_melody()`.

ASCENDING
DESCENDING
TWINKLE
SUCCESS
FAILURE
ALERT
STARTUP
SHUTDOWN
HAPPY_BIRTHDAY
SUPER_MARIO
ODE_TO_JOY
IMPERIAL_MARCH

9 Useful MicroPython functions

9.1 Information printing

```
1 x, y = 64, 128.4096
2 y_string = "y"
3
4 # 1. option - separation by commas
5 # This option is suitable for simple and quick printing
6 # Individual elements are separated by a space when printed
7 print("x =", x, ",", y_string, "=", y)
8 # x = 64, y = 128.4096
9
10 # 2. option - use of f-string
11 # This option allows formatting float numbers
12 # d indicates integer type, s indicates string type and f indicates
    float type
13 # .2 specifies the number of displayed decimal places
14 print(f"x = {x:d}, {y_string:s} = {y:.2f}")
15 # x = 64, y = 128.41
16
17 # The use of f-string is also suitable for the cube display
18 robot.display.fill(0)
19 # first line of the display
20 robot.display.text(f"x = {x:d}", 0, 0, 1)
21 # second line of the display
22 robot.display.text(f"{y_string:s} = {y:.2f}", 0, 10, 1)
23 robot.display.show()
24
25 # Printing without line breaks
26 for i in range(10):
27     print(i, end="")
28 print() # line break at the end
29 # 123456789
```

Code 26: Printing information.

9.2 Random number generation

Example of random number generation.

- [MicroPython random library documentation](https://docs.micropython.org/en/latest/library/random.html) [<https://docs.micropython.org/en/latest/library/random.html>]

```
1 # Import functions from the random library
2 from random import random, randint
3
4 random() # Random float number in interval [0.0, 1.0)
5
6 a, b = 0, 100
7 randint(a, b) # Random integer in the interval [a, b]
```

Code 27: Random library usage.

9.3 Time

Example of functions for putting the program to sleep, timing, and periodic function calls using the timer.

- [MicroPython time library documentation](https://docs.micropython.org/en/latest/library/time.html) [<https://docs.micropython.org/en/latest/library/time.html>]
- [MicroPython documentation of Timer class](https://docs.micropython.org/en/latest/library/machine.Timer.html) [<https://docs.micropython.org/en/latest/library/machine.Timer.html>]

```
1 # Import functions from the sleep library
2 from time import sleep, sleep_ms, sleep_us, ticks_us, ticks_diff
3
4 start_time = ticks_us() # Get start time in microseconds
5
6 sleep(1)      # Sleep for 1 second
7 sleep_ms(2)   # Sleep for 2 milliseconds
8 sleep_us(3)   # Sleep for 3 microseconds
9
10 end_time = ticks_us() # Get end time in microseconds
11 print(ticks_diff(end_time, start_time), "us") # 1002003 us
```

Code 28: Time library usage.

```
1 # Import class Timer
2 from machine import Timer
3
4 counter = 0
5
6 # Definition of callback function for timers
7 def my_callback(t):
8     global counter
9     counter += 1
10    print(counter)
11
12 # Initializing a timer that calls the callback function periodically
13 # with a frequency of 100 Hz
14 # -1 indicates a virtual timer (the rp2040 microcontroller does not
15 # allow the use of hardware timers)
16 timer1 = Timer(-1)
17 timer1.init(mode=Timer.PERIODIC, freq=100, callback=my_callback)
18
19 # Initializing a timer that calls the callback function only once after
20 # 1000 ms
21 timer2 = Timer(-1)
22 timer2.init(mode=Timer.ONE_SHOT, period=1000, callback=my_callback)
23
24 # Turn off the periodic timer
25 timer1.deinit()
```

Code 29: Timer class usage.

9.4 Binary data manipulation

These functions are useful when working with binary data e.g. when communicating wirelessly via [ESP32](#).

- [MicroPython struct library documentation \[https://docs.micropython.org/en/latest/library/struct.html \]](https://docs.micropython.org/en/latest/library/struct.html)

```

1 # Import struct library
2 import struct
3
4 # Creating binary data from three numbers. The format of the numbers is
  # determined by the string of the first function argument. The string
  # format is described in detail in the struct library documentation.
5 # In this case, the first number is stored as int (4 bytes), the second
  # as float (4 bytes) and the third as double (8 bytes).
6 binary_data = struct.pack("@ifd", 12, 1.234, 5)
7
8 # Backward Conversion of binary data
9 values = struct.unpack("@ifd", binary_data)
10
11 # Compute the memory size in bytes needed to store binary data
12 byte_size = struct.calcsize("@ifd")
13 print(byte_size) # 16

```

Code 30: Struct library usage.

9.5 Multi-core programming

The RP2040 microcontroller has two cores. Open-Cube reserves Core 1 by default for its native real-time hardware loop. This loop processes motor regulation, encoders, the integrated IMU, sensor polling, buttons and native timers while MicroPython and the user program run on Core 0.

While `Settings -> OC doublecore` is enabled, `_thread.start_new_thread()` raises `RuntimeError` because Core 1 is reserved. Lock and identity functions remain available. To run a Python function on Core 1, disable `OC doublecore` in Settings. The cube restarts automatically and hardware processing uses its Core 0 fallback paths.

- [Python _thread library documentation \[https://docs.python.org/3/library/_thread.html \]](https://docs.python.org/3/library/_thread.html)

```

1 # First disable Settings -> OC doublecore
2 import _thread
3
4 lock = _thread.allocate_lock()
5 counter = 0
6
7 # Core 1 function definition
8 def core1_task():
9     global counter
10    with lock:
11        counter += 1
12
13 # Core 1 activation
14 _thread.start_new_thread(core1_task, ())
15
16 with lock:
17    print(counter)

```

Code 31: _thread library usage.

10 Used Python terms

Data types:

- **int** (integer) – (12, -51),
- **float** – decimal number (120.32, -0.1248),
- **bool** – logical value (True, False),
- **str** (string) – text string ("abc", "Open-Cube").

Data structures:

- **list** – sequence of data that can contain elements of different data types. Individual elements can be changed, added and removed. It is written in square brackets and the elements are separated by commas. [1, "Open-Cube", True, 0.23]
- **tuple** – A sequence of data that can contain elements of different data types. Individual elements cannot be changed, added or removed. It is written in round brackets and the elements are separated by commas. (1, "Open-Cube", True, 0.23)

Key words:

- **None** – no value (sensor not working, failed to measure value, etc.).