FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Department of Measurement

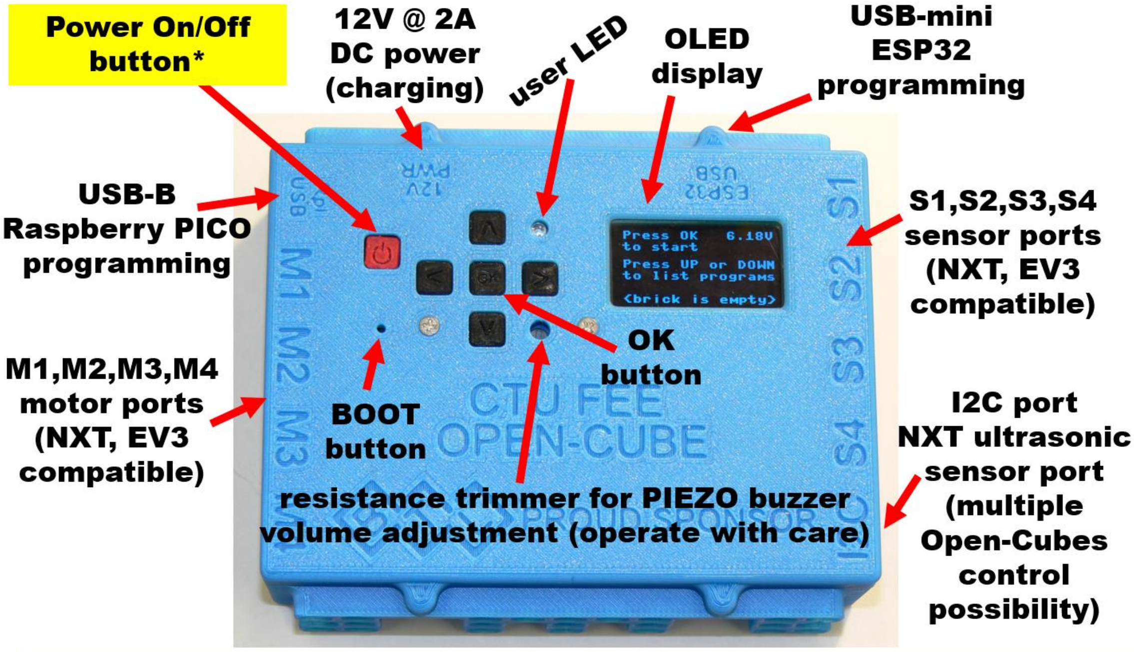
MAGLAB.cz
SENSORS AND MAGNETICS LABORATORY

PROUD SPONSOR

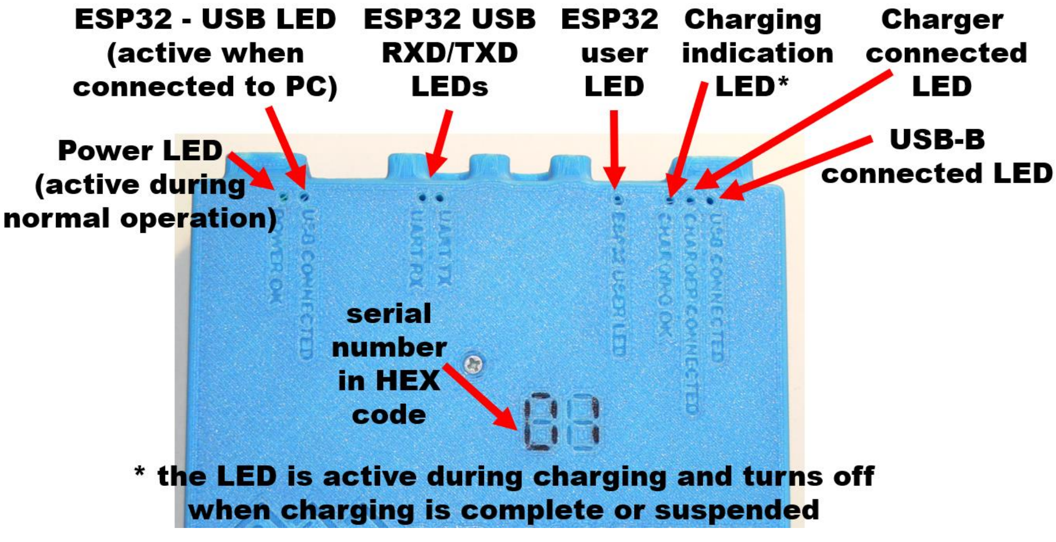
OPEN-CUBE QUICK START GUIDE

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

1. Switch ON the Open-Cube and connect it to the computer via USB-B (look for COMport number in the device manager)
2. 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…)
3. 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.



***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(4000,50)                      # 4000 Hz, 50 % duty
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), currently not implemented

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

2024

Contents

List of codes

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 also be downloaded at Open-Cube repository [https://gitlab.fel.cvut.cz/open-cube/firmware/] from the micropython folder. To upload firmware after an update, or to upload firmware to a new cube, follow these steps:

1. Download firmware [https://gitlab.fel.cvut.cz/open-cube/firmware/-/tree/main/micropython/] (uf2 binary file) with Open-Cube firmware.
2. Connect the cube to the computer using a USB cable.
3. 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, 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/], 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 in the editor, you can press `Run current script` to run the code currently displayed in the editor on the cube. It is recommended to run only the main program `main.py` in this way, which contains the cube framework with all the necessary functions for controlling the peripherals initialized. When the main program is run, a menu controlled by the buttons on the cube is displayed with the option to run the loaded user program. The `main.py` file runs automatically when the cube is turned on.

2.3 Run the program independently

To speed up testing, the user program can be run independently without starting the main program:

```
# Importing the Robot class
from lib.robot import *

# If the program is run from the menu, nothing happens
# If it is run independently, the robot variable is initialized
global robot
try:
  robot
except:
  robot = Robot()
.
.
.
# When the program exits, the cube can be reset to return to the menu
import machine
machine.reset()
```

Code 1: Run the program independently.

3 Robot

When the cube is turned on, main program `main.py` initialises a global object `robot`. This object contains all functions for initializing, deinitializing and accessing cube peripheral objects. The functions are described in the following sections. The structure of the variables of the `robot` object that can be used to access the peripheral objects is as follows:

```
robot
 ├─ battery
 ├─ buttons
 ├─ buzzer
 ├─ display
 ├─ esp
 ├─ 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 cube buttons, LED, buzzer and battery voltage measurement with undervoltage protection. Sensors, motors and ESP32 Bluetooth 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. No need to specify for sensor types Sensor.GYRO and Sensor.ULTRA.

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.

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.

```
# Import sleep function
from time import sleep
# Import cube button constants
from lib.robot_consts import Button

# Program definition
def main:
    # Access the global variable robot
    global robot

    # Loop waiting for program termination
    while True:
        # Change LED state
        robot.led.toggle()

        # Get buttons state
        buttons = robot.buttons.pressed()
        # Terminate program when left button is pressed
        if buttons[Button.LEFT]:
            break

        # Wait 1 second
        sleep(1)

# Start program
main()
```

Code 2: 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)

```
# Import sleep function
from time import sleep
# Import cube button constants
from lib.robot_consts import Button

def main:
  global robot
  while True:
    # Get buttons state
    buttons = robot.buttons.pressed()

    # Show buttons state in terminal
    print("Button pressed:",
          "power:", buttons[Button.POWER],
          "left:",  buttons[Button.LEFT],
          "right:", buttons[Button.RIGHT],
          "OK:",       buttons[Button.OK],
          "up:",   buttons[Button.UP],
          "down:",     buttons[Button.DOWN])
    # Terminate program when left button is pressed
    if buttons[Button.LEFT]:
      break
    # Wait 1 second
    sleep(1)

# Start program
main()
```

Code 3: 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.

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

Code 4: 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** (frekvence: Hz) – PWM frequency.
- **duty** (střída:%) – PWM duty cycle.

off()

Turn off buzzer.

```
# Turning on the buzzer at 4000 Hz and 50% duty cycle
robot.buzzer.set_freq_duty(4000, 50)
# Change buzzer frequency to 1000 Hz
robot.buzzer.set_freq_duty(1000, 50)
# Turn off buzzer
robot.buzzer.off()
```

Code 5: Buzzer usage.

3.4 Battery

class **Battery()**

Voltage measurement on the power supply batteries. During initialization, a timer is set with a 200 ms period to start the voltage measurement.

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 switches off the cube when voltage drops below 6.5 V. 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

deinit()

Disable periodic battery voltage measurement.

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

Code 6: 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()

Show current frame buffer on the display.

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=1111b*)

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_rect(*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, color=1*)

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

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

draw_bar_chart_v(*value, x, y, w, h, low_lim=0, high_lim=100, no_of_tics=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_tics** (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_tics=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_tics** (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, 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.
- **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.

```
# Delete frame buffer
robot.display.fill(0)
# Draw text to the upper left corner of the frame buffer
robot.display.text("test message", 0, 0, 1)

# Writing numbers with precision of 2 decimal places to the frame
    buffer on the next line
x, y = 1.23, 56.789
robot.display.text(f"x={x:.2f}, y: {y:.2f}", 0, 10, 1)
# Show frame buffer on the display
robot.display.show()

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

Code 7: Display usage.

3.6 Gyroscope and accelerometer

class **ICM42688**()

Integrated gyroscope and accelerometer ICM42688.

Measurement of acceleration and angular velocity in three axes. The interrupt pin of the sensor is connected to the GyroAcc.IRQ_PIN (GPIO 28) of the micro controller.

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

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

Access: `robot.sensors.gyro_acc`

read_value()

Read and return the measured values of angular velocity and acceleration. The GyroAcc constants can be used to choose a specific value.

Returns: Tuple of acceleration and angular velocity in x, y, z axes.

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

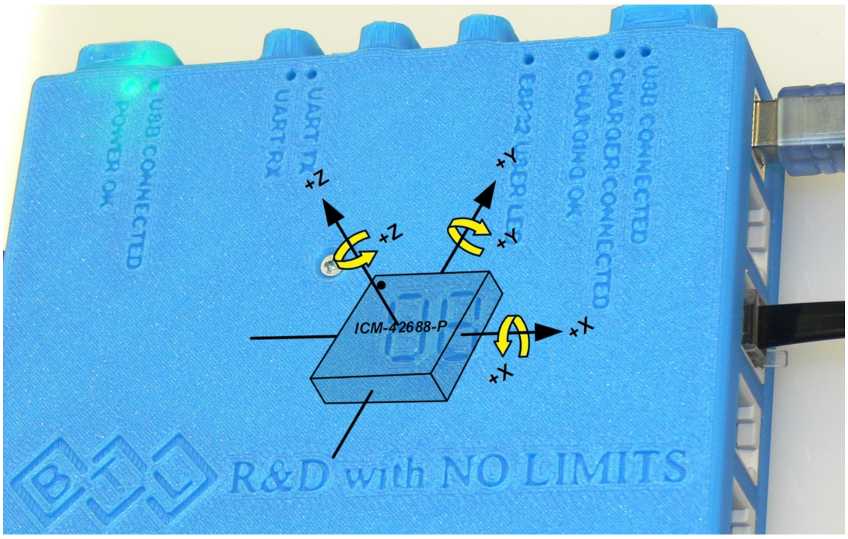


Figure 1: Orientation of integrated gyroscope and accelerometer axes.

Example of reading data from gyroscope and accelerometer using irq (interrupt pin):

```
# Import libraries
from time import sleep
from machine import Pin
# Import constants
from lib.robot_consts import Button, Sensor, GyroAcc

a_g_data = (0,0,0,0,0,0)

def main:
  global robot, a_g_data
  # Sensor initialization
  robot.init_sensor(sensor_type=Sensor.GYRO_ACC)
  # Setup of GPIO 28
  icm_irq_pin = Pin(GyroAcc.IRQ_PIN, Pin.IN)
  # Set the callback function called on the rising edge of irq when new data is available
  icm_irq_pin.irq(trigger=Pin.IRQ_RISING, handler=callback)

  # Loop waiting for program termination
  while True:
    # Print last acceleration and angular velocity values in terminal
    print("Acc:",  a_g_data[GyroAcc.AX],
                   a_g_data[GyroAcc.AY],
                   a_g_data[GyroAcc.AZ],
          "Gyro:", a_g_data[GyroAcc.GX],
                   a_g_data[GyroAcc.GY],
                   a_g_data[GyroAcc.GZ],)
    # Get buttons state
    buttons = robot.buttons.pressed()
    # Terminate program when left button is pressed
    if buttons[Button.LEFT]:
      break
    # Wait 1 second
    sleep(1)

  # Turn off callback before sensor deinitialization
  icm_irq_pin.irq(handler=None)
```

```
  # Sensor deinitialization
  robot.deinit_sensor(sensor_type=Sensor.GYRO_ACC)

def callback(p):
  global robot, a_g_data
  # Read data from sensor
  a_g_data = robot.sensors.gyro_acc.read_value()

# Start program
main()
```

Code 8: Reading data from a gyroscope and accelerometer using irq.

3.7 ESP32 communication

ESP32 communicates with the cube (RP2040) via serial link. It can communicate with other devices via Wi-Fi or Bluetooth. The ESP is programmable by connecting a USB cable on the top of the cube. The default firmware of the ESP (available from the Open-Cube repository [https://gitlab.fel.cvut.cz/open-cube/firmware/-/tree/main/ESP]) allows in Bluetooth mode to forward data from the cube via Bluetooth virtual COM port to a computer or mobile phone and in Wi-Fi mode acts as an access point with a web server with versatile controls (buttons, switches, indicators).

3.7.1 Bluetooth ESP32 connection with external device

Communication is serial via Bluetooth virtual COM port and is tested only for Windows and Android.

1. To pair, the cube must be turned on and in menu.
2. In the Windows Bluetooth settings, select Add new device, then Bluetooth, and pair your computer with the cube. See image 2.
3. Confirm pin on the cube and on the computer.
4. Continue to the advanced Bluetooth settings to find the COM port number. The desired COM port has a name containing ESP32SPP and direction Outgoing. See figure 3. This COM port is used for both receiving and sending data!

Figure 2: Pairing a computer with a cube.

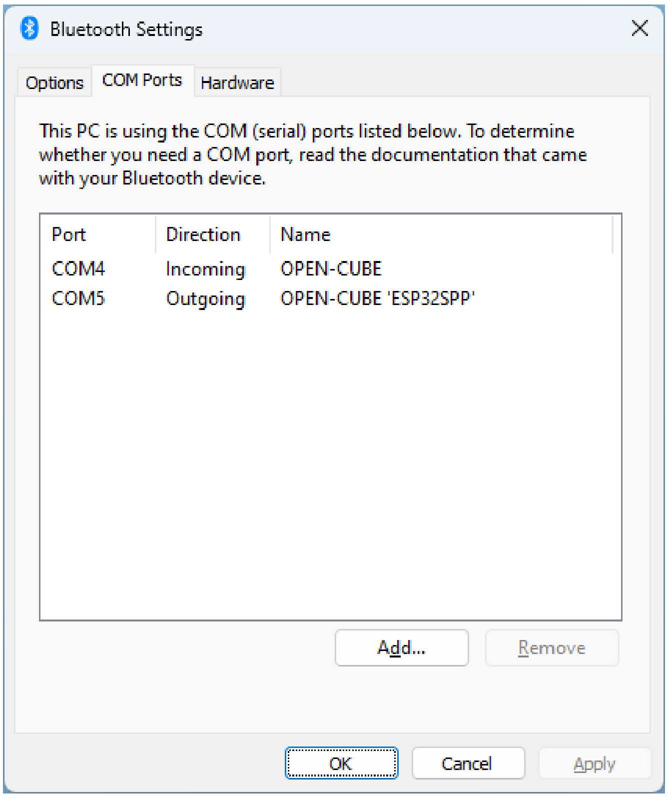


Figure 3: Determining the COM port number

3.7.2 Cube

class **ESP**()

Bidirectional data sending between microcontroller and ESP32 via UART interface. At initialization, a timer with a 10 ms period is started, during which the data sent by the ESP is received and stored.

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

1. In the case of Bluetooth mode, data is sent in messages. Sending and receiving data can be done in ASCII or in binary mode.

   ASCII mode: The message being sent is terminated by the LF (new line) terminator. When receiving data, the message is available after receiving the LF or CR terminator.

   Binary mode: For communication, it is necessary to specify the header of the data being sent and received to distinguish the beginning of the message. In order to correctly receive the message and determine the end of the message, the size of the message in bytes must be pre-specified. The data is of type bytes and its interpretation is up to the user.

2. Wi-Fi access point mode with web server provides versatile controls – buttons, switches and indicators.

Initialization:	Turning on the cube.
Deinitialization:	Turning off the cube.
Access:	`robot.esp`

reset()

Reset the ESP and set Bluetooth mode.

bt()

Set Bluetooth mode

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 ESP name for both Bluetooth and Wi-Fi. Default name is "OPEN-CUBE".

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

set_password(*password*)

Set ESP Wi-Fi passwords. Password must be at least 8 characters. 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(*buff*)

Send message in Bluetooth mode.

Parameters: **buff** (buffer: str(ASCII)/bytes(binary)) – Buffer of data to be sent.

bt_set_binary(*header, data_size*)

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.

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 9 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: **indicators** (tuple) – 6x float.

wifi_set_buttons_labels(*labels*)

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

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

wifi_set_switches_labels(*labels*)

Set label of 9 web server switches in Wi-Fi access point mode. String of each switch can have a maximum length of 9 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. String of each indicator can have a maximum length of 9 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. String of each number can have a maximum length of 9 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:

```
# Import library for working with binary data
import struct

# Send and receive a string message in Bluetooth ASCII mode
robot.esp.bt_write("test message")
received_message = robot.esp.bt_read()

# Send and receive a message of 8 bytes in Bluetooth binary mode
robot.esp.bt_set_binary((112, 241, 3, 62), 8)
# Create and send a message with two numbers of type single (4 bytes
    each)
write_message = struct.pack("<ff", 521.9, -11.821)
robot.esp.bt_write(write_message)
# Receive and decode a message containing two numbers of type single
received_message2 = robot.esp.bt_read()
(value1, value2) = struct.unpack("<ff", received_message2)

# Switch ESP to Wi-Fi access point mode
robot.esp.wifi()
# Set labels of indicators
robot.esp.set_indicators_labels(["1", "2", "3", "4", "5"])
# Get state of switches
switches = robot.esp.get_switches()
```

Code 9: Communication between the micro controller and ESP32.

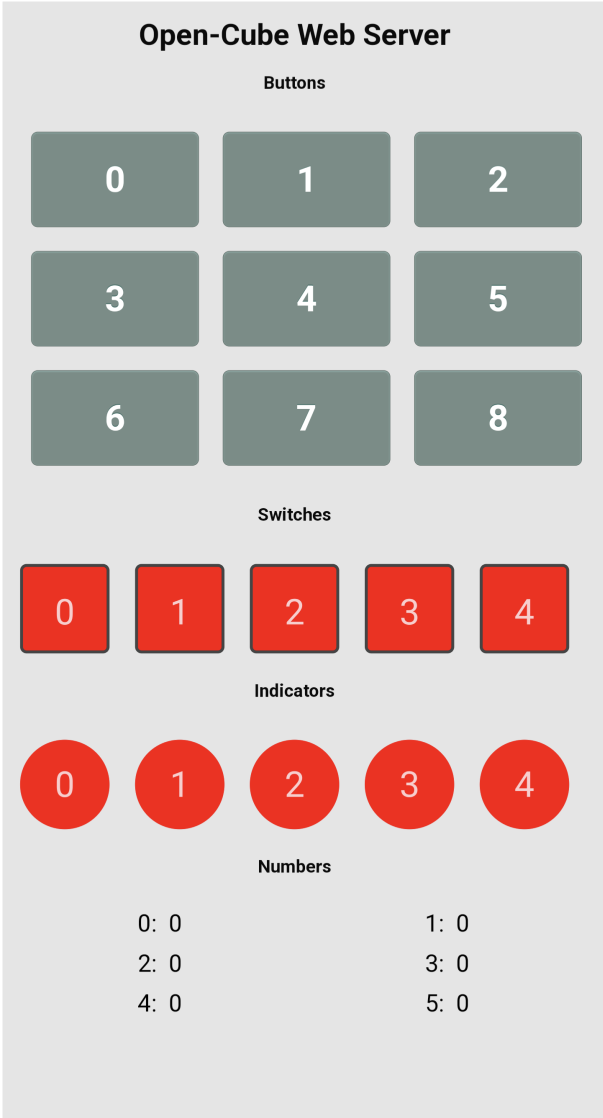


Figure 4: Indexing of ESP web server elements.

3.7.3 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]. This app allows you to send and receive messages from the cube in Bluetooth ASCII mode.

3.7.4 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 CF). 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/bluetooth-communication.html/].

```
% Connecting the cube with Matlab
cube = bluetooth('OPEN-CUBE')
% Set LF (new line) terminator for sending and receiving ASCII data
configureTerminator(device, 'LF')
%%
% Send a test message to the cube in Bluetooth ACII mode. The message is automatically terminated by the set terminator.
writeline(cube, 'test message')
% Receiving a message from the cube. The function receives data and waits for termination by previously set terminator.
cube_message = readline(cube)
%%
% Send a test message to the cube in Bluetooth binary mode. The message is interpreted as a string format and is not terminated by a terminator.
write(cube, 'test message', 'string')
% Receive a message from the cube. The message is 8 bytes long and is interpreted as float.
cube_message2 = read(cube, 8, 'float')
```

Code 10: Communication with the cube using Matlab.

3.7.5 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
textttFixed-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 3.7.1 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 v official documentation [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`.

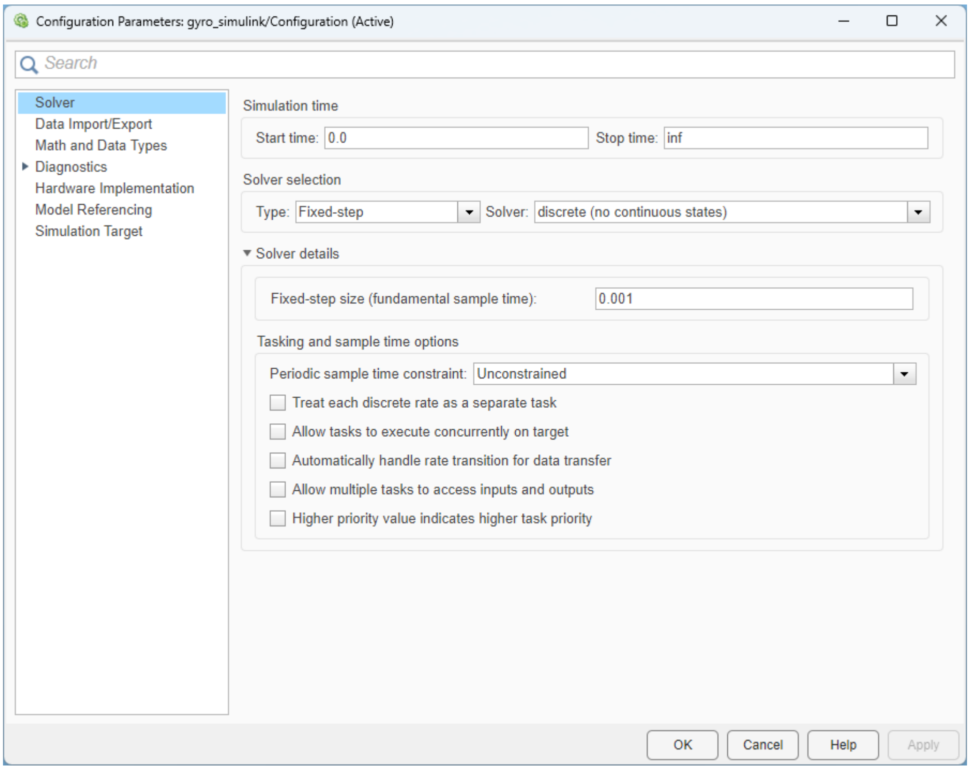


Figure 5: Simulink solver configuration.

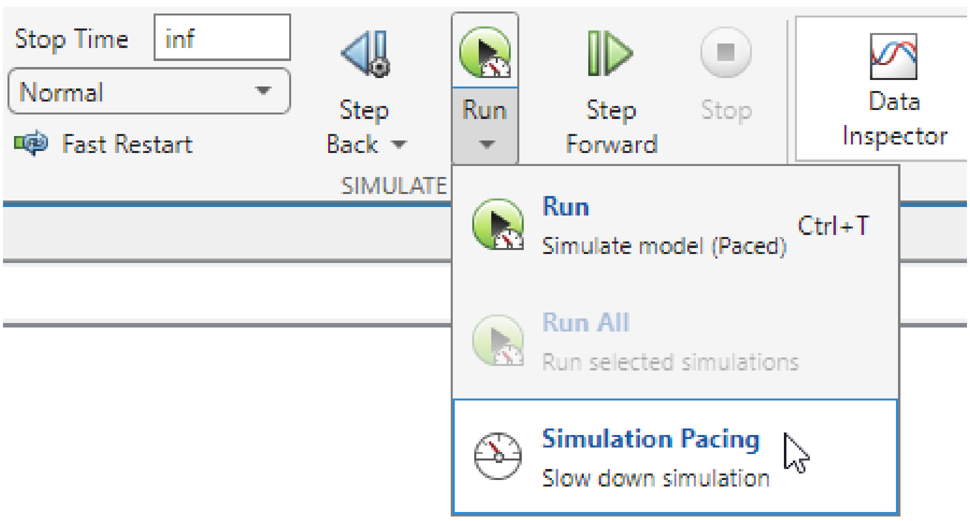


Figure 6: Location of Simulation pacing configuration in Simulink.

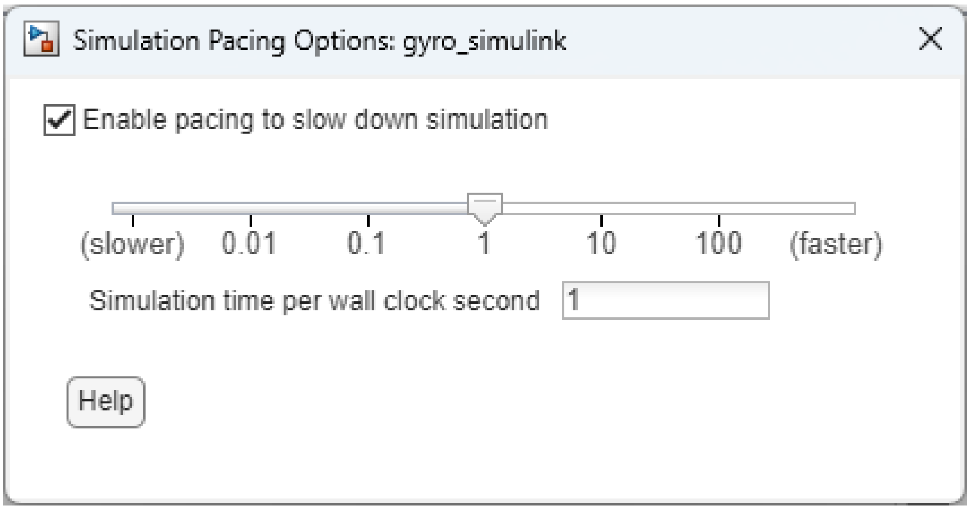


Figure 7: Simulink Simulation Pacing configuration.

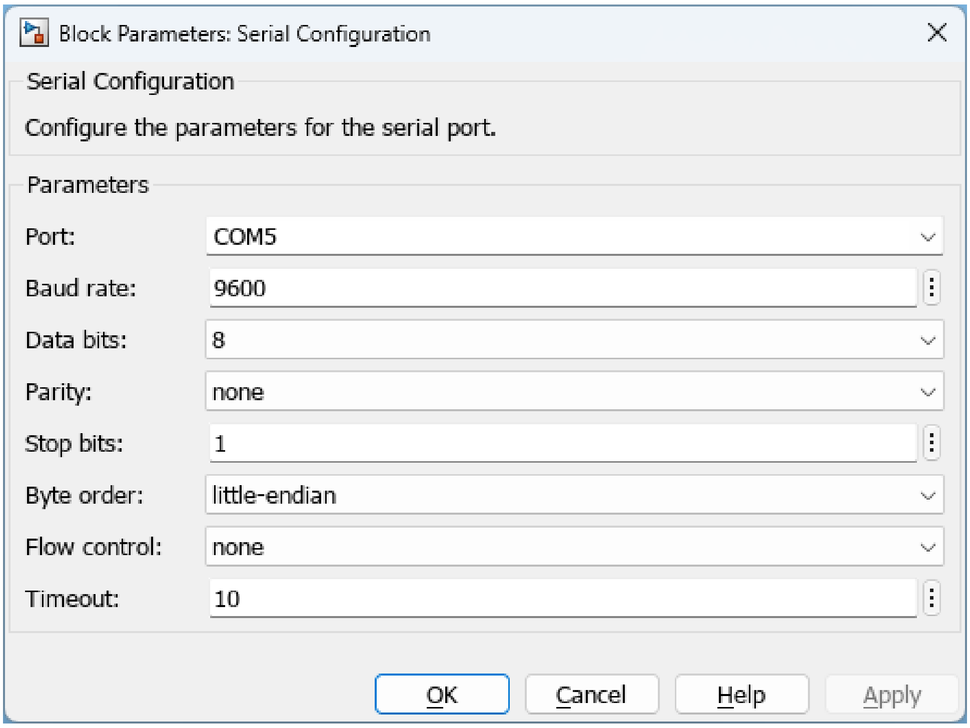


Figure 8: Configuration of Serial Configuration block in Simulink.

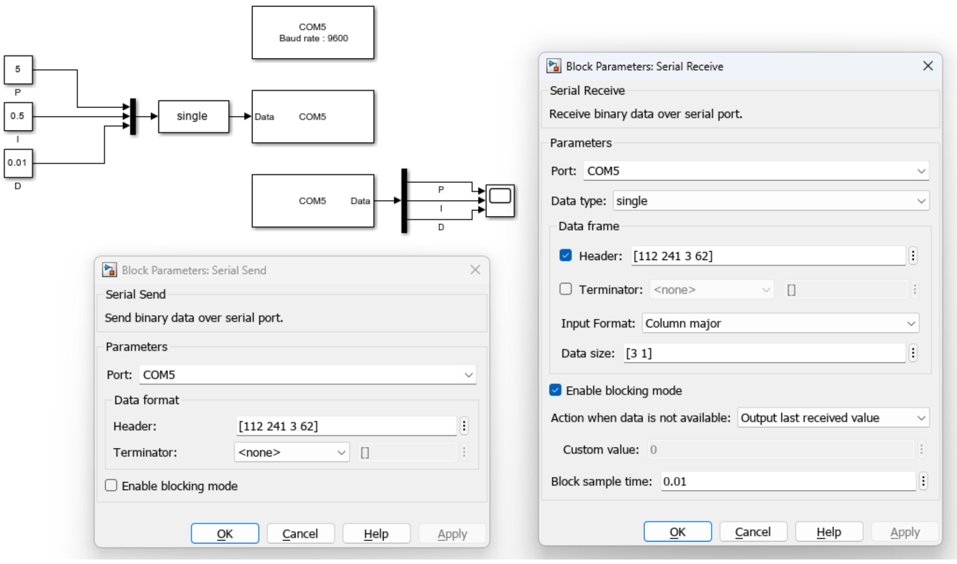


Figure 9: Example of using serial communication in Simulink.

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

```python
# Import sensor and port constants
from lib.robot_consts import Sensor, Port

# Initialize NXT Touch Sensor on sensor port 1
robot.init_sensor(sensor_type=Sensor.NXT_TOUCH, port=Port.S1)

# Determine the button state
touch_pressed = robot.sensors.touch[Port.S1].pressed()
```

Code 11: NXT Touch Sensor usage.

4.2 Light sensor

class **LightSensor**(*port*)

NXT Ligth 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.

togggle()

Change state of sensor LED.

intensity(*pin_on*)

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)

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.

```
# Import sensor and port constants
from lib.robot_consts import Sensor, Port

# Initialize NXT Light Sensor on sensor port 1
robot.init_sensor(sensor_type=Sensor.LIGHT, port=Port.S1)

# Turn on sensor LED
robot.sensors.light[Port.S1].on()
# Measure light intensity
light_intensity = robot.sensors.light[Port.S1].intensity()
```

Code 12: 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, multiple NXT Ultrasonic Sensor type sensors cannot be connected.

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)

```
# Import sensor constants
from lib.robot_consts import Sensor

# Initialize NXT Ultrasonic Sensor on the I2C port
robot.init_sensor(sensor_type=Sensor.NXT_ULTRASONIC)
# Measure distance
distance = robot.sensors.ultra_nxt.distance()
```

Code 13: 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)

```
# Import sensor and port constants
from lib.robot_consts import Sensor, Port

# Initialize NXT Sound Sensor on sensor port 1
robot.init_sensor(sensor_type=Sensor.NXT_SOUND, port=Port.S1)

# Measure sound intenisty
sound_intensity = robot.sensors.sound[Port.S1].intensity()
```

Code 14: NXT Sound Sensor usage.

5 EV3 sensors

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

```
# Import sensor and port constants
from lib.robot_consts import Sensor, Port

# Initialize EV3 Tocuh Sensor on sensor port 1
robot.init_sensor(sensor_type=Sensor.EV3_TOUCH, port=Port.S1)

# Get button state
touch_pressed = robot.sensors.touch[Port.S1].pressed()
```

Code 15: 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)`
Deinitialization: `robot.deinit_sensor(sensor_type=Sensor.EV3_COLOR)`
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. Light intensity is measured when the LED is on, from this value a measurement when the LED is off is subtracted.

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: (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 of 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 (1-7 – black, blue, green, yellow, red, white, brown)

```
# Import sensor constants
from lib.robot_consts import Sensor, Port

```

```python
# Initialize EV3 Color Sensor on sensor port 1
robot.init_sensor(sensor_type=Sensor.EV3_COLOR, port=Port.S1)

# Measure reflected light intensity
reflection = robot.sensors.light[Port.S1].reflection()
```

Code 16: 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)`
Deinitialization: `robot.deinit_sensor(sensor_type=Sensor.EV3_ULTRASONIC)`
Access: `robot.sensors.ultrasonic[Port]`

distance()

Return last measured distance.
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

```python
# Import sensor constants
from lib.robot_consts import Sensor, Port

# Initialize EV3 Ultrasonic Sensor on sensor port 1
robot.init_sensor(sensor_type=Sensor.EV3_ULTRASONIC, port=Port.S1)

# Measure distance
distance = robot.sensors.ultrasonic[Port.S1].distance()
```

Code 17: 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)`
Deinitialization: `robot.deinit_sensor(sensor_type=Sensor.EV3_INFRARED)`
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) – Channel to be checked.
Returns: Tuple of direction (-25 is leftmost, 25 rightmost) and distance (100% is approximately 200 cm, -128 if no driver is found) of the controller.
Type: (int, int) (-25-25, 0-100%)

```
# Import sensor constants
from lib.robot_consts import Sensor, Port

# Initialize EV3 Infrared Sensor on sensor port 1
robot.init_sensor(sensor_type=Sensor.EV3_INFRARED, port=Port.S1)
# Measure distance
distance = robot.sensors.infrared[Port.S1].distance()
```

Code 18: 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)`
Deinitialization: `robot.deinit_sensor(sensor_type=Sensor.EV3_GYRO)`
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)

```
# Import sensor constants
from lib.robot_consts import Sensor, Port

# Initialize EV3 Gyro Sensor on sensor port 1
robot.init_sensor(sensor_type=Sensor.EV3_GYRO, port=Port.S1)

# Reset angle of rotation
robot.sensors.gyro[Port.S1].reset_angle(0)

# Measure angle of rotation and angular velocity
(angle, speed) = robot.sensors.gyro[Port.S1].angle_and_speed()
```

Code 19: EV3 Gyro Sensor usage.

6 Open-Cube sensors

6.1 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)`
Deinitialization:	`robot.deinit_sensor(sensor_type=Sensor.OC_COLOR)`
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 reflected and reference light 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 reflected and reference light 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 reflected and reference light 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 (1-7 – black, blue, green, yellow, red, white, brown)

```
# Import sensor constants
from lib.robot_consts import Sensor, Port

# Initialize Open-Cube Color Sensor on sensor port 1
robot.init_sensor(sensor_type=Sensor.OC_COLOR, port=Port.S1)

# Measure reflected light intensity
reflection = robot.sensors.light[Port.S1].reflection()
```

Code 20: Open-Cube Color Sensor usage.

6.2 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)`
Deinitialization: `robot.deinit_sensor(sensor_type=Sensor.OC_LASER)`
Access: `robot.sensors.ultrasonic[Port]`

distance()

Return last measured distance.
Returns: Distance of the nearest object.
Type: int (0-4000)

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)

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 (0-4000)

set_led_distance(*distance*)

Set threshold distance at which the green LED of the sensor lights up.
Parameters: **distance** (int) – Threshold distance in mm.

```
# Import sensor constants
from lib.robot_consts import Sensor, Port

# Initialize Open-Cube LIDAR on sensor port 1
robot.init_sensor(sensor_type=Sensor.OC_LASER, port=Port.S1)

# Measure distance
distance = robot.sensors.laser[Port.S1].distance()
```
Code 21: Open-Cube LIDAR usage.

6.3 Ultrasonic sensor

class **UltrasonicSensor**(*port*)

Open-Cube Ultrasonic Sensor.

Measuring distance of an object from the sensor in the range 0–9999 mm.

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

distance()

Return last measured distance.

Returns:	Distance of the nearest object.
Type:	int (0-4000)

set_led_distance(*distance*)

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

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

```
# Import sensor constants
from lib.robot_consts import Sensor, Port

# Initialize Open-Cube Ultrasonic Sensor on sensor port 1
robot.init_sensor(sensor_type=Sensor.OC_ULTRASONIC, port=Port.S1)

# Measure distance
distance = robot.sensors.ultrasonic[Port.S1].distance()
```

Code 22: Open-Cube Ultrasonic sensor usage.

6.4 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)`
Deinitialization:	`robot.deinit_sensor(sensor_type=Sensor.OC_GYRO)`
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°)

euler_angles_mag()

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

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: (0–1, 0–1, 0–1, 0–1)

raw()

Returns the last measured angular velocity, acceleration and magnetic field intensity in the x, y, z axes.

Returns: Tuple of angular velocity, acceleration and magnetic field intensity in the x, y, z axes.
Type: (g_x: °/s, g_y: °/s, g_z: °/s, a_x: -, a_y: -, a_z: -, m_x: mg, m_y: mg, m_z: mg)

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 until the calibration is complete.

Returns: 1 if calibration is completed.
Type: int (0/1)

```python
# Import sensor constants
from lib.robot_consts import Sensor, Port

# Initialize the OC AHRS sensor on sensor port1
robot.init_sensor(sensor_type=Sensor.OC_GYRO, port=Port.S1)

# Gyroscope calibration
calibrated = 0
while not calibrated:
  calibrated = robot.sensors.gyro[Port.S1].calibrate_gyro()
```

```

# Get sensor orientation
euler_angles = robot.sensors.gyro[Port.S1].euler_angles()
yaw, pitch, roll = euler_angles[0], euler_angles[1], euler_angles[2]
```

Code 23: Open-Cube AHRS sensor usage.

7 Motor

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 o 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.motor[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.

position()

Return motor rotational position.

Returns: Motor rotational position.
Type: angle: °

speed()

Return motor speed o rotation.

Returns: Motor speed o rotation.
Type: angular velocity: °/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 modey** – position: °
- **speed control mode** – angular velocity: °/s

regulator_power()

Return current control action.

Returns: Control action.

Type: power (-100–100%)

```python
# Import port constants
from lib.robot_consts import Port

# Initialize motors on motor ports 1 and 2
robot.init_motor(Port.M1)
robot.init_motor(Port.M2)

# Set power on the motors to 50%, and 20% in the opposite direction of
    rotation
robot.motors[Port.M1].set_power(50)
robot.motors[Port.M2].set_power(-20)

# Initialize motors encoders
robot.motors[Port.M1].init_encoder()
robot.motors[Port.M2].init_encoder()

# Measure current position and speed of rotation of motors
pos1 = robot.motors[Port.M1].position()
pos2 = robot.motors[Port.M2].position()
speed1 = robot.motors[Port.M1].speed()
speed2 = robot.motors[Port.M2].speed()
```

Code 24: 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_ULTRA

NXT Ultrasonic Sensro.

NXT_TOUCH

NXT Touch Sensor.

NXT_SOUND

NXT Sound Sensor.

GYRO_ACC

ICM20608-G gyroscope and accelerometer.

EV3_COLOR

EV3 Color Sensor.

EV3_GYRO

EV3 Gyroscopic Sensor.

EV3_INFRARED

EV3 Infrarfed 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.

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.

S1

Sensor port 1.

S2

Sensor port 2.

S3

Sensor port 3.

S4

Sensor port 4.

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.

BLACK

BLUE

GREEN

YELLOW

RED

WHITE

BROWN

9 Useful MicroPython functions

9.1 Information printing

```
x, y = 64, 128.4096
y_string = "y"

# 1. option - separation by commas
# This option is suitable for simple and quick printing
# Individual elements are separated by a space when printed
print("x =", x, ",", y_string, "=", y)
# x = 64, y = 128.4096

# 2. option - use of f-string
# This option allows formatting float numbers
# d indicates integer type, s indicates string type and f indicates
    float type
# .2 specifies the number of displayed decimal places
print(f"x = {x:d}, {y_string:s} = {y:.2f}")
# x = 64, y = 128.41

# The use of f-string is also suitable for the cube display
robot.display.fill(0)
# first line of the display
robot.display.text(f"x = {x:d}", 0, 0, 1)
# second line of the display
robot.display.text(f"{y_string:s} = {y:.2f}", 0, 10, 1)
robot.display.show()

# Printing without line breaks
for i in range(10)
  print(i, end="")
print() # line break at the end
# 123456789
```

Code 25: 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]

```
# Import functions from the random library
from random import random, randint

random() # Random float number in interval [0.0, 1.0)

a, b = 0, 100
randint(a, b) # Random integer in the interval [a, b]
```

Code 26: 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]
- MicroPython documentation of Timer class [https://docs.micropython.org/en/latest/library/machine.Timer.html]

```
# Import functions from the sleep library
from time import sleep, sleep_ms, sleep_us, ticks_us, ticks_diff

start_time = ticks_us() # Get start time in microseconds

sleep(1)     # Sleep for 1 second
sleep_ms(2) # Sleep for 2 milliseconds
sleep_us(3) # Sleep for 3 microseconds

end_time = ticks_us() # Get end time in microseconds
print(ticks_diff(end_time, start_time), "us") # 1002003 us
```

Code 27: Time library usage.

```
# Import class Timer
from machine import Timer

counter = 0

# Definition of callback function for timers
def my_callback(t):
  global counter
  counter += 1
  print(counter)

# Initializing a timer that calls the callback function periodically
    with a frequency of 100 Hz
# -1 indicates a virtual timer (the rp2040 microcontroller does not
    allow the use of hardware timers)
timer1 = Timer(-1)
timer1.init(mode=Timer.PERIODIC, freq=100,  callback=my_callback)

# Initializing a timer that calls the callback function only once after
     1000 ms
timer2 = Timer(-1)
timer2.init(mode=Timer.ONE_SHOT, period=1000, callback=my_callback)

# Turn off the periodic timer
timer1.deinit()
```

Code 28: 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]

```
# Import struct library
import struct

# 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.
# 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).
binary_data = struct.pack("@ifd", 12, 1.234, 5)

# Backward Conversion of binary data
values = struct.unpack("@ifd", binary_data)

# Compute the memory size in bytes needed to store binary data
byte_size = struct.calcsize("@ifd")
print(byte_size) # 16
```

Code 29: Struct library usage.

9.5 Multi-core programming

The rp2040 micro controller has 2 cores, only one core is used by default. The use of a second core is possible, but is currently only experimental in MicroPython.

- Python _thread library documentation [https://docs.python.org/3.5/library/_thread.html]

```
# Import library _thread a function sleep
import _thread
from time import sleep

lock = _thread.allocate_lock()
counter = 0

# Second core function definition
def core2():
  global counter
  lock.acquire() # Lock for working with shared data
  counter += 1
  lock.release() # Release
  _thread.exit() # Termination of second core

# Second core activation
_thread.start_new_thread(core2, ())

lock.acquire() # Lock for working with shared data
print(counter)
lock.release() # Release
```

Code 30: _thread library usage.

10 Used Python terms

Datova 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.).