Programmierung der Roboter

Micropython Software-Bibliotheken für den Betrieb der Roboter.

- Die Roboterbibliothek "robotlibrary"
- Dokumentation
- Codebeispiele
- Startdatei für den SMARS-Roboter
- Startdatei für Crawly

Die Roboterbibliothek "robotlibrary"

Dieses Modul, das von Codeberg heruntergeladen werden kann, steuert die Roboter mit allen Peripheriegeräten (Motoren, Sensoren). Dazu muss das Paket heruntergeladen und entpackt werden. Das Verzeichnis „robotlibrary“ muss dann auf den Pico hochgeladen werden.

Lade hier das letzte Release herunter.

SMARS Roboter "Theo III"

Um das Modul zu benutzen, muss nur folgender Import gemacht werden: `from robotlibrary.robot import Robot`.

Ein kurzes Codebeispiel, wie der Roboter funktioniert, ist in der Quelldatei zu finden.

Oder hier ein Beispiel für die Benutzung des Servomotors.

```
from robotlibrary.robot import Robot
from time import sleep,sleep_ms
try:
    r = Robot(False)
    r.set_angle(0)
    sleep_ms(500)
    r.set_angle(180)
    sleep_ms(500)
    r.set_angle(90)
    r.set_speed(80)
    while True:
        while r.get_dist() > 15:
            pass
        r.emergency_stop()
        sleep_ms(400)
        r.set_speed_instantly(80)
        r.spin_before_obstacle(20)
        r.set_forward(True)
```

```
        r.set_speed(80)
except:
    r.emergency_stop()
    print("Robot stopped")
```

Eine ausführlichere Anleitung zum Benutzen der Bibliothek befindet sich auf der Seite Startdatei für den SMARS-Roboter

Crawly

Crawly ist ein Roboter, der, ähnlich wie eine Schildkröte auf vier Beinen kriechen kann. Um Crawly zu steuern, muss nur folgender Import gemacht werden: `from robotlibrary.crawly import Crawly`

Ein kurzes Codebeispiel, wie der Roboter funktioniert, ist in der Quelldatei zu finden.

Under construction:

Walky

Walky ist ein Roboter, der, ähnlich wie ein Hund, auf vier Beinen laufen kann. Um Walky zu steuern, muss nur folgender Import gemacht werden: `from robotlibrary.walky import Walky`

Ein kurzes Codebeispiel, wie der Roboter funktioniert, ist in der Quelldatei zu finden.

Die Dokumentation für die Bibliothek ist unter `docs` zu finden oder kann hier heruntergeladen werden: robotlibrary.pdf

Dokumentation

Documentation for robotlibrary/robot.py

Robot

This is the central class which manages and uses all the other components of the robot. The parameters are defined in config.py

_drive

This abstracted driving function is only called locally by the other functions with better names. It accelerates and decelerates to make driving more natural. Do not call directly!

_drive_instantly

This abstracted driving function is only called locally by the other functions with better names. It sets the speed immediately. Do not call directly!

set_speed_instantly

Sets the new speed immediately. Doesn't change the driving mode of the robot. :param s: the speed you want to set.

set_speed

Sets the new speed and accelerates and decelerates. Doesn't change the driving mode of the robot. :param s: the speed you want to set.

set_forward

Sets the direction of the robot. True means forward. :param f: True for forwards and False for backwards.

spin_right

Spin right indefinitely.

spin_left

Spin left indefinitely.

turn_right

This turns the robot to the right without it spinning on the spot. Each call makes the turn steeper.

turn_left

This turns the robot to the right without it spinning on the spot. Each call makes the turn steeper.

go_left

With Meccanum wheels the robot goes sideways to the left.

go_right

With Meccanum wheels the robot goes sideways to the right.

turn

This turns the robot right or left. Is mostly used by the remote control. :param turn: positive or negative value. Higher values mean steeper turn.

go_straight

Lets the robot go straight on. Usually called when a turn shall end.

spin_before_obstacle

This spins until the distance to an obstacle is greater than the given parameter *distance*. :param distance: The distance

toggle_spin

Toggle turn for the given duration. With each call the opposite direction(clockwise / anti-clockwise) is used. :param d: The duration for the turn in milliseconds.

random_spin

Randomly turn for the given duration. :param d: The duration for the turn in milliseconds.

stop

Stop the robot slowly by deceleration.

emergency_stop

Stop the robot immediately.

ir_detected

If implemented this method is called when the IR-sensor has detected a change. Fill in your code accordingly.

get_dist

Get the distance from the ultrasonic sensor.

set_angle

If implemented, turn the servo motor with the ultrasonic sensor to the given angle. :param a: The angle that is to be set.

get_smallest_distance

This returns the angle of the ultrasonic sensor where it measured the smallest distance

get_longest_distance

This returns the angle of the ultrasonic sensor where it measured the longest distance

Documentation for robotlibrary/config.py

Module

This defines the parameters for the motors.

MAX_DUTY: Set to lower than the maximum not to overload the motors. Absolute maximum is 65535. MIN_DUTY: Set this to the minimum duty cycle that the motor needs to start moving. MIN_SPEED: Only 0 is making sense here but if you want you can change that. Must be above 0 though. MAX_SPEED: If you want another scale than 0-100, set the maximum here.

DEBOUNCE_WAIT: This defines the waiting time for the debouncing of the buttons. Leave as it is if you don't know what it means.

WHITE_DETECTED: Use these constants to check for white or black with the IR-sensor. Don't change! BLACK_DETECTED: Use these constants to check for white or black with the IR-sensor. Don't change!

Motors and ultrasonic sensor must use consecutive pins. so, f. ex. the left motor uses pins 12 and 13. Use >None< if you don't use the device. MLF and LRF are for four wheel drive. ML: pin number

for left motor (or left rear motor for four wheel drive). MR: pin number for right motor (or right rear motor for four wheel drive). MLF: pin number for left motor (or left front motor for four wheel drive). Use None if not used. MRF: pin number for right motor (or right front motor for four wheel drive). Use None if not used. US: pin number for the ultrasonic sensor. Use None if not used. IR: pin number for the infrared sensor. Use None if not used. SERVO: pin number for the servo motor. Use None if not used.

JS_X_MEDIAN JS_Y_MEDIAN JS_MAX_DUTY JS_MIN_DUTY: These define the parameters for the joystick. You need to calibrate the numbers. Look at joystick.py for details.

ROBOT_NAME: You need to set a custom name if you use a remote control.

SERVO_MIN_DUTY: Only change if the servo doesn't move the required 180°. SERVO_MAX_DUTY: Only change if the servo doesn't move the required 180°.

Codebeispiele

Theo III

Best Practise

Ausschalten der Motoren bei Unterbrechung des Programms

Anfangs wird man sehr viel an dem Roboterprogramm testen müssen. Dabei wird das Programm dann häufig abstürzen. Damit die Motoren nicht weiter in dem Zustand laufen, in dem sie dabei geschaltet waren, kann man mit einem `try/except` arbeiten:

```
try:
  # Hier läuft das Programm
except Exception as err:
    print(err) # Nötig, um Fehlermeldungen angezeigt zu bekommen.
    r.emergency_stop() # Roboter anhalten, hier ein Beispiel mit der robotlibrary.
    print("Robot stopped") # Damit es ganz deutlich ist.
except KeyboardInterrupt:
    r.emergency_stop()
    print("Keyboard interrupt")
```

Effizienter Code

Auch wenn die Rechenleistung der Picos ausreichen sollte, ergibt es Sinn, sich über Effizienz Gedanken zu machen, da schwer zu erkennen ist, ob manche Probleme durch Überlastung des Prozessors hervorgerufen werden.

```
while True:
    robot.drive()
    if us.get_dist() > min_distance:
        # stop or turn
        robot.stop()
```

In diesem Beispiel wird in der Schleife der Befehl `drive()` mit jedem Durchlauf aufgerufen, was nicht sonderlich effizient ist, da die Motoren weiterfahren, auch wenn das Programm gerade andere Befehle ausführt. Eine bessere Variante wäre diese:

```
robot.drive()
while us.get_dist() > min_distance:
    pass
robot.turn()
```

Hier wird nur die Entfernung zum nächsten Hindernis überprüft. Sobald der Roboter zu nahe gekommen ist, wird die Schleife beendet und der Code wird weiter ausgeführt.

Fehlertoleranter Code

Die Sensoren, die wir benutzen, liefern nicht immer zuverlässige und korrekte Ergebnisse. Daher kann man sich nicht darauf verlassen, dass **eine** Messung ausreicht. Ist man auf genauere Ergebnisse angewiesen, kann es sinnvoll sein, die Ergebnisse von Sensormessungen (insbesondere des Ultraschallsensors) zu filtern. Dazu kann gehören, Extremwerte, die im vorliegenden Fall unwahrscheinlich sind, zu ignorieren oder Mittelwerte von mehreren Messungen zu bilden. Ein Beispiel für die Glättung der Entfernungswerte aus dem Ultraschallsensor:

```
us = Ultra(16)
dist_values = deque([0,0,0,0,0],5)
while True:
    d = us.get_dist()
    dist_values.append(d)
    d = sum(dist_values)/len(dist_values)
    print(f"Entfernung: {d} cm")
```

Eine andere Methode muss für das Auslesen des Infrarotsensors gefunden werden. Hier könnte man z.B. eine kurze Wartezeit einbauen und dann abfragen, ob der Sensor immer noch denselben Wert liefert wie bei Auslösung der Reaktion.

Überschreiben von Methoden aus der Bibliothek

Möchte man die Funktion einer Methode aus der Roboterbibliothek (robotlibrary) verändern, kann man die Methode überschreiben (Fachbegriff aus der objektorientierten Programmierung). Dafür wird die Klasse `Robot` vererbt, wie in dem Codebeispiel angegeben. Möchte man nun z.B. die Methode `set_speed()` verändern, dann definiert man sie einfach neu. Ist eine Methode nicht in der abgeleiteten Klasse (in diesem Fall `MyRobot` definiert, dann wird die Methode der Elternklasse (`Robot`) genommen. Probiere es aus, indem du das vorliegende Programm einmal mit und einmal ohne die Definition von `set_speed()` aufrufst.

```
from robotlibrary.robot import Robot
from time import sleep, sleep_ms


class MyRobot(Robot):
```

```
    def __init__(self):
        super().__init__(False)
        print("start")

    def set_speed(self,x):
        print(f"Child method set_speed. Value: {x}")

def main():
    try:
        robot = MyRobot()
        robot.set_speed(90)
        while True:
            sleep(1)
    except KeyboardInterrupt:
        print("The robot was stopped by the user.")
    finally:
        robot.emergency_stop()

if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Den Ultraschallsensor im Hintergrund laufen lassen

In komplexeren Programmen kann es lästig werden, immer wieder die Entfernung zum nächsten Hindernis zu überpüfen. Dieses Beispiel erläutert, wie man das Problem löst, indem man die Entfernungsmessung in den Hintergrund verlegt.

Dieses Beispiel vereint alle vorgestellten Techniken und sollte als Standard-Startdatei genutzt werden.

```
from time import sleep, sleep_ms
import uasyncio as asyncio
from robotlibrary.robot import Robot
################################## Your class definition
class MyRobot(Robot):
    def __init__(self):
        super().__init__(False) # Call the original constructor.
        print("Start MyRobot")
```

```
    # With this method defined here, the robot will not drive as the speed is not set in this
function.
    # This is to illustrate how overwriting works.
    def set_speed(self,x):
        print(f"Child method set_speed. Value: {x}")

################################## End of class definition
# Define functions for your program
async def monitor_dist():
    '''This checks the distance from the ultrasonic sensor continually.
    If the given distance is longer than the measured one, react_to_obstacle() will be called.
    '''
    global distance
    while True:
        if robot.get_dist() < distance:
            react_to_obstacle()
        await asyncio.sleep_ms(100)

def react_to_obstacle():
    '''Do whatever you want to do when an obstacle is detected.
    '''
    global distance
    robot.random_spin(300)
    robot.set_forward(True)
    robot.set_speed(80)

async def driving_program():
    robot.set_speed(90)
    while True:
        print("Driving program running.")
        await asyncio.sleep_ms(3000)

async def main():
    asyncio.create_task(monitor_dist())
    await driving_program()



################################## Initialize the robot and start the program.
robot = MyRobot()
```

```
distance = 20 # Define the distance you want to have.
if __name__ == "__main__":
    # execute only if run as a script
    try:
        asyncio.run(main())
    except KeyboardInterrupt:
        print("The robot was stopped by the user.")
    finally:
        robot.emergency_stop()
```

Beschleunigung mit Entfernungsmessung

Diese Methode ist nur nötig, wenn man nicht asyncio benutzt. Beschleunigt man den Roboter langsam mit der Methode `set_speed`, dann kann er in der Zeit bis zum Erreichen der Geschwindigkeit keine Entfernungsmessung durchführen. Dies ist ein Beispiel, wie man beides erreichen kann:

```
obstacle_detected = False
new_speed = 100
speed_now = 0
min_distance = 15
while speed_now <= new_speed and not obstacle_detected:
    #Set the speed for the motors, f. ex. motor.set_speed(speed_now)
    utime.sleep_ms(10+int(speed_now/2))
    speed_now += 1
    if us.get_dist() < min_distance: # Adjust the code to your needs.
        obstacle_detected = True
if obstacle_detected:
    # Stop or turn or whatever
    obstacle_detected = False
else:
    # keep going
    pass
```

Mit einem Timer arbeiten

Diese Programmiertechnik könnte man z. B. nutzen, um nach einer bestimmten Zeit in das Programm einzuschalten, falls der Roboter zu lange dieselbe Funktion, z. B. vorwärts fahren, ausführt. Dann würde der Zähler zu gegebener Zeit eine Änderung herbeiführen können.

```
from machine import Pin,Timer
import time

def do_something(t):
    print("do something!")

def main():
    timer = Timer()
    timer.init(period=1500, mode=Timer.PERIODIC, callback=do_something)
    while True:
        print("Running...")
        time.sleep_ms(700)

if __name__ == "__main__":
    # execute only if run as a script
    main()
```

Startdatei für den SMARS-Roboter

Kopiere den Code und speichere die Datei als `main.py` auf dem `Pico`.

```
from time import sleep, sleep_ms
import uasyncio as asyncio
from robotlibrary.robot import Robot
################################## Your class definition
class MyRobot(Robot):
    def __init__(self):
        super().__init__(False) # Call the original constructor.
        print("Start MyRobot")

    # With this method defined here, the robot will not drive as the speed is not set in this
function.
    # This is to illustrate how overwriting works.
    def set_speed(self,x):
        print(f"Child method set_speed. Value: {x}")

################################## End of class definition
# Define functions for your program
async def monitor_dist():
    '''This checks the distance from the ultrasonic sensor continually.
    If the given distance is longer than the measured one, react_to_obstacle() will be called.
    '''
    global distance
    while True:
        if robot.get_dist() < distance:
            react_to_obstacle()
        await asyncio.sleep_ms(100)

def react_to_obstacle():
    '''Do whatever you want to do when an obstacle is detected.
    '''
    global distance
```

```
    robot.random_spin(300)
    robot.set_forward(True)
    robot.set_speed(80)

async def driving_program():
    robot.set_speed(90)
    while True:
        print("Driving program running.")
        await asyncio.sleep_ms(3000)

async def main():
    asyncio.create_task(monitor_dist())
    await driving_program()



################################## Initialize the robot and start the program.
robot = MyRobot()
distance = 20 # Define the distance you want to have.
if __name__ == "__main__":
    # execute only if run as a script
    try:
        asyncio.run(main())
    except KeyboardInterrupt:
        print("The robot was stopped by the user.")
    finally:
        robot.emergency_stop()
```

Startdatei für Crawly

Um den Bewegungsablauf für Crawly anders als in der Bibliothek zu machen, können auch hier die entsprechenden Klassen überschrieben werden.

```
from time import sleep, sleep_ms
from robotlibrary.crawly import Crawly
from robotlibrary.crawly_leg import Leg
from robotlibrary.crawly_joint import Joint
from robotlibrary import config_crawly as conf
from robotlibrary.servo import Servo
################################## Your class definition
class MyCrawly(Crawly):
    def __init__(self):
        self.legs = {
            "front_right" : MyLeg(4, True, True, "front right"),
            "rear_right" : MyLeg(6, True, False, "rear right"),
            "rear_left" : MyLeg(0, False, False, "rear left"),
            "front_left" : MyLeg(2, False, True, "front left")
            }

    def galumph(self):
            for l in self.legs.values():
                l.leg_fully_up()
            sleep_ms(150)
            for l in self.legs.values():
                l.leg_fully_forward()
            sleep_ms(150)
            for l in self.legs.values():
                l.leg_fully_down()
            sleep_ms(150)
            for l in self.legs.values():
                l.leg_fully_backward()
            sleep_ms(150)

class MyLeg(Leg):
    def __init__(self, pin, right, front, name):
        if right and front:
```

```
            self.shoulder = MyJoint(conf.SHOULDER_FRONT, name, False, False, pin)
        if right and not front:
            self.shoulder = MyJoint(conf.SHOULDER_REAR, name, False, False, pin)
        if not right and front:
            self.shoulder = MyJoint(conf.SHOULDER_FRONT, name, True, False, pin)
        if not right and not front:
            self.shoulder = MyJoint(conf.SHOULDER_REAR, name, True, False, pin)
        self.knee = MyJoint(conf.KNEE, name, False, True, pin+1)

class MyJoint(Joint):
    def __init__(self, j_type, name, left_side, inverted, pin):
        self.name = name
        self.j_type = j_type
        min_duty = conf.SERVO_MIN_DUTY
        max_duty = conf.SERVO_MAX_DUTY
        self.left_side = left_side
        if j_type == conf.SHOULDER_FRONT:
            self.__min_angle = conf.SHOULDER_FRONT_MIN_ANGLE
            self.__max_angle = conf.SHOULDER_FRONT_MAX_ANGLE
        elif j_type == conf.SHOULDER_REAR:
            self.__min_angle = conf.SHOULDER_REAR_MIN_ANGLE
            self.__max_angle = conf.SHOULDER_REAR_MAX_ANGLE
        elif j_type == conf.KNEE:
            self.__min_angle = conf.KNEE_MIN_ANGLE
            self.__max_angle = conf.KNEE_MAX_ANGLE
            # min_duty = conf.SERVO_MIN_DUTY_TYPE2 # Comment out if the duty cycle is
different from the shoulder servo's duty cycle.
            # max_duty = conf.SERVO_MAX_DUTY_TYPE2 # Comment out if the duty cycle is
different from the shoulder servo's duty cycle.
        self.servo = Servo(pin, inverted, min_duty, max_duty)


################################## End of class definition

def move_program():
    crawly.move_to_start_pos()
    for i in range(10):
        crawly.galumph()

def main():
```

```
    move_program()



################################## Initialize the robot and start the program.
crawly = MyCrawly()
if __name__ == "__main__":
    # execute only if run as a script
    try:
        main()
    except KeyboardInterrupt:
        print("The robot was stopped by the user.")
    finally:
        crawly.park()
```

Hier wurden schon alle drei Hauptklassen des Crawly-Roboters überschrieben. Die anderen Methoden in den Klassen können bei Bedarf auch überschrieben werden.