RoboClaw

Dec 10, 2019

Contents

Roboclaw driver library and examples adapted for python3 and circuitpython and micropython

Optimizations applied to the original code include UART Serial I/O & CRC checking. These optimizations are meant to allow your application to run faster than the vanilla python library offered by BasicMicro.

CHAPTER 1

Installation

The best way to make sure you have the latest version of this library is by cloning the repository, and running the python setup script.

```
git clone htpps://github.com/2bndy5/python-roboclaw.git
cd python-roboclaw
python setup.py install
```

CHAPTER 2

Usage

Once you have installed the library, you can import it into your python application. Please note that the roboclaw requires a USB serial connection as well as the main power source (or battery) for the motors connected to the "+" & "-" terminals for proper communication. In your applications code, you need only import the `Roboclaw` driver class.

```
from python_roboclaw import Roboclaw
from serial import Serial

serial_obj = Serial('dev/ttyUSB0', 38400) # default baudrate is 38400
rclaw = RoboClaw(serial_obj)
rclaw.forward_backward_mixed(64) # stops both motors
```

2.1 Table of Contents

2.1.1 Vanilla (examples from original library)

Bare Minimum

Listing 1: examples/roboclaw_bareminimum.py

```
"""bare minimum example shows you """
from roboclaw import Roboclaw
try:  # if on win32 or linux
    from serial import Serial as UART
except ImportError:
    try:  # try CircuitPython
        from board import UART
    except ImportError:  # try MicroPythom
        from roboclaw.usart_serial_ctx import SerialUART as UART

# Windows comport name
```

(continues on next page)

(continued from previous page)

```
rc = Roboclaw(UART("COM10", 38400))
# Linux comport name
# rc = Roboclaw(UART("/dev/ttyACM0", 38400))
# if CircuitPython or MicroPythom
# rc = Roboclaw(UART(rate=38400))
```

Simple PWM

Listing 2: examples/roboclaw_simplepwm.py

```
import time
from roboclaw import Roboclaw
try:  # if on win32 or linux
    from serial import SerialException, Serial as UART
except ImportError:
    try:  # try CircuitPython
        from board import UART
    except ImportError:
        try:  # try MicroPythom
            from roboclaw.usart_serial_ctx import SerialUART as UART

# Windows comport name
# rc = Roboclaw(UART("COM3", 115200))
# Linux comport name
# rc = Roboclaw(UART("/dev/ttyACM0", 115200))
# if CircuitPython or MicroPythom
rc = Roboclaw(UART(), address=0x80)

while 1:
    rc.forward_m1(32)  # 1/4 power forward
    rc.backward_m2(32)  # 1/4 power backward
    time.sleep(2)

    rc.backward_m1(32)  # 1/4 power backward
    rc.forward_m2(32)  # 1/4 power forward
    time.sleep(2)

    rc.backward_m1(0)  # Stopped
    rc.forward_m2(0)  # Stopped
    time.sleep(2)

    m1duty = 16
    m2duty = -16
    rc.forward_backward_m1(64+m1duty)  # 1/4 power forward
    rc.forward_backward_m2(64+m2duty)  # 1/4 power backward
    time.sleep(2)

    m1duty = -16
    m2duty = 16
    rc.forward_backward_m1(64+m1duty)  # 1/4 power backward
    rc.forward_backward_m2(64+m2duty)  # 1/4 power forward
    time.sleep(2)

    rc.forward_backward_m1(64)  # Stopped
    rc.forward_backward_m2(64)  # Stopped
```

(continues on next page)

(continued from previous page)

```
    time.sleep(2)
```

Mixed PWM

Listing 3: examples/roboclaw_mixedpwm.py

```
import time
from roboclaw import Roboclaw
try:  # if on win32 or linux
    from serial import SerialException, Serial as UART
except ImportError:
    try:  # try CircuitPython
        from board import UART
    except ImportError:
        try:  # try MicroPythom
            from roboclaw.usart_serial_ctx import SerialUART as UART

# Windows comport name
# rc = Roboclaw(UART("COM3", 115200))
# Linux comport name
# rc = Roboclaw(UART("/dev/ttyACM0", 115200))
# if CircuitPython or MicroPythom
rc = Roboclaw(UART(), address=0x80)

rc.forward_mixed(0)
rc.turn_right_mixed(0)


def test(loop=2):
    while loop:
        rc.forward_mixed(32)
        time.sleep(2)
        rc.backward_mixed(32)
        time.sleep(2)
        rc.turn_right_mixed(32)
        time.sleep(2)
        rc.turn_left_mixed(32)
        time.sleep(2)
        rc.forward_mixed(0)
        rc.turn_right_mixed(32)
        time.sleep(2)
        rc.turn_left_mixed(32)
        time.sleep(2)
        rc.turn_right_mixed(0)
        time.sleep(2)
        loop -= 1
```

Speed

Listing 4: examples/roboclaw_speed.py

```
# ***Before using this example the motor/controller combination must be
# ***tuned and the settings saved to the Roboclaw using IonMotion.
```

(continues on next page)

(continued from previous page)

```
# ***The Min and Max Positions must be at least 0 and 50000

import time
from roboclaw import Roboclaw
try:  # if on win32 or linux
    from serial import SerialException, Serial as UART
except ImportError:
    try:  # try CircuitPython
        from board import UART
    except ImportError:
        try:  # try MicroPythom
            from roboclaw.usart_serial_ctx import SerialUART as UART

# Windows comport name
# rc = Roboclaw(UART("COM3", 115200))
# Linux comport name
# rc = Roboclaw(UART("/dev/ttyACM0", 115200))
# if CircuitPython or MicroPythom
rc = Roboclaw(UART(), address=0x80)


def displayspeed():
        enc1 = rc.read_encoder_m1()
        enc2 = rc.read_encoder_m2()
        speed1 = rc.read_speed_m1()
        speed2 = rc.read_speed_m2()

        print("Encoder1:"),
        if(enc1[0] == 1):
                print enc1[1],
                print format(enc1[2], '02x'),
        else:
                print "failed",
        print "Encoder2:",
        if(enc2[0] == 1):
                print enc2[1],
                print format(enc2[2], '02x'),
        else:
                print "failed ",
        print "Speed1:",
        if(speed1[0]):
                print speed1[1],
        else:
                print "failed",
        print("Speed2:"),
        if(speed2[0]):
                print speed2[1]
        else:
                print "failed "


version = rc.read_version(
if version[0] == False:
        print "GETVERSION Failed"
else:
        print repr(version[1])

```

(continues on next page)

(continued from previous page)

```
while 1:
        rc.speed_m1(12000)
        rc.speed_m2(-12000)
        for i in range(0, 200):
                displayspeed()
                time.sleep(0.01)

        rc.speed_m1(-12000)
        rc.speed_m2(12000)
        for i in range(0, 200):
                displayspeed()
                time.sleep(0.01)
```

Speed and Acceleration

Listing 5: examples/roboclaw_speedaccel.py

```
# ***Before using this example the motor/controller combination must be
# ***tuned and the settings saved to the Roboclaw using IonMotion.
# ***The Min and Max Positions must be at least 0 and 50000

import time
from roboclaw import Roboclaw
try:  # if on win32 or linux
    from serial import SerialException, Serial as UART
except ImportError:
    try:  # try CircuitPython
        from board import UART
    except ImportError:
        try:  # try MicroPythom
            from roboclaw.usart_serial_ctx import SerialUART as UART

# Windows comport name
# rc = Roboclaw(UART("COM3", 115200))
# Linux comport name
# rc = Roboclaw(UART("/dev/ttyACM0", 115200))
# if CircuitPython or MicroPythom
rc = Roboclaw(UART(), address=0x80)


def displayspeed():
    enc1 = rc.read_encoder_m1()
    enc2 = rc.read_encoder_m2()
    speed1 = rc.read_speed_m1()
    speed2 = rc.read_speed_m2()
    print("Encoder1:")
    if enc1[0] == 1:
        print(enc1[1])
        print(format(enc1[2], '02x'))
    else:
        print("failed")
    print("Encoder2:")
    if enc2[0] == 1:
        print(enc2[1])
        print(format(enc2[2], '02x'))
```

(continues on next page)

(continued from previous page)

```
    else:
        print("failed ")
    print("Speed1:")
    if speed1[0]:
        print(speed1[1])
    else:
        print("failed")
    print("Speed2:")
    if speed2[0]:
        print(speed2[1])
    else:
        print("failed ")


version = rc.ReadVersion()
if version[0] == False:
    print("GETVERSION Failed")
else:
    print(repr(version[1]))

while 1:
    rc.speed_accel_m1(12000, 12000)
    rc.speed_accel_m2(12000, -12000)
    for i in range(0, 200):
        displayspeed()
        time.sleep(0.01)

    rc.speed_accel_m1(12000, -12000)
    rc.speed_accel_m2(12000, 12000)
    for i in range(0, 200):
        displayspeed()
        time.sleep(0.01)
```

Speed, Acceleration, and Distance

Listing 6: examples/roboclaw_speedacceldistance.py

```
# ***Before using this example the motor/controller combination must be
# ***tuned and the settings saved to the Roboclaw using IonMotion.
# ***The Min and Max Positions must be at least 0 and 50000

import time
from roboclaw import Roboclaw
try:  # if on win32 or linux
    from serial import SerialException, Serial as UART
except ImportError:
    try:  # try CircuitPython
        from board import UART
    except ImportError:
        try:  # try MicroPythom
            from roboclaw.usart_serial_ctx import SerialUART as UART

# Windows comport name
# rc = Roboclaw(UART("COM3", 115200))
# Linux comport name
```

(continues on next page)

(continued from previous page)

```
# rc = Roboclaw(UART("/dev/ttyACM0", 115200))
# if CircuitPython or MicroPythom
rc = Roboclaw(UART(), address=0x80)


def displayspeed():
    enc1 = rc.read_encoder_m1()
    enc2 = rc.read_encoder_m2()
    speed1 = rc.read_speed_m1()
    speed2 = rc.read_speed_m2()

    print("Encoder1:")
    if enc1[0] == 1:
        print(enc1[1])
        print(format(enc1[2], '02x'))
    else:
        print("failed")
    print("Encoder2:")
    if enc2[0] == 1:
        print(enc2[1])
        print(format(enc2[2], '02x'))
    else:
        print("failed ")
    print("Speed1:")
    if speed1[0]:
        print(speed1[1])
    else:
        print("failed")
    print("Speed2:")
    if speed2[0]:
        print(speed2[1])
    else:
        print("failed ")


version = rc.read_version()
if version[0] == False:
    print("GETVERSION Failed")
else:
    print(repr(version[1]))

while 1:
    rc.speed_accel_distance_m1(12000, 12000, 42000, 1)
    rc.speed_accel_distance_m2(12000, -12000, 42000, 1)

    # distance travelled is v*v/2a = 12000*12000/2*48000 = 1500
    rc.speed_accel_distance_m1(12000, 0, 0, 0)

    # that makes the total move in one direction 48000
    rc.speed_accel_distance_m2(12000, 0, 0, 0)

    buffers = (0, 0, 0)

    # Loop until distance command has completed
    while(buffers[1] != 0x80 and buffers[2] != 0x80):
        print("Buffers: ")
        print(buffers[1])
```

(continues on next page)

(continued from previous page)

```
        print(" ")
        print(buffers[2])
        displayspeed()
        buffers = rc.ReadBuffers()

    time.sleep(1)

    rc.speed_accel_distance_m1(48000, -12000, 46500, 1)
    rc.speed_accel_distance_m2(48000, 12000, 46500, 1)

    # distance travelled is v*v/2a = 12000*12000/2*48000 = 1500
    rc.speed_accel_distance_m1(48000, 0, 0, 0)
    # that makes the total move in one direction 48000
    rc.speed_accel_distance_m2(48000, 0, 0, 0)

    buffers = (0, 0, 0)
    # Loop until distance command has completed
    while(buffers[1] != 0x80 and buffers[2] != 0x80):
        print("Buffers: ")
        print(buffers[1])
        print(" ")
        print(buffers[2])
        displayspeed()
        buffers = rc.read_buffer_length()

    # When no second command is given the motors will automatically slow down to 0␣
↪which takes 1 second
    time.sleep(1)
```

Speed and Distance

Listing 7: examples/roboclaw_speeddistance.py

```
# ***Before using this example the motor/controller combination must be
# ***tuned and the settings saved to the Roboclaw using IonMotion.
# ***The Min and Max Positions must be at least 0 and 50000

import time
from roboclaw import Roboclaw
try:  # if on win32 or linux
    from serial import SerialException, Serial as UART
except ImportError:
    try:  # try CircuitPython
        from board import UART
    except ImportError:
        try:  # try MicroPythom
            from roboclaw.usart_serial_ctx import SerialUART as UART

# Windows comport name
# rc = Roboclaw(UART("COM3", 115200))
# Linux comport name
# rc = Roboclaw(UART("/dev/ttyACM0", 115200))
# if CircuitPython or MicroPythom
rc = Roboclaw(UART(), address=0x80)

```

(continues on next page)

(continued from previous page)

```
def displayspeed():
    enc1 = rc.read_encoder_m1()
    enc2 = rc.read_encoder_m2()
    speed1 = rc.read_speed_m1()
    speed2 = rc.read_speed_m2()

    print("Encoder1:")
    if enc1[0] == 1:
        print(enc1[1])
        print(format(enc1[2], '02x'))
    else:
        print("failed")
    print("Encoder2:")
    if enc2[0] == 1:
        print(enc2[1])
        print(format(enc2[2], '02x'))
    else:
        print("failed ")
    print("Speed1:")
    if speed1[0]:
        print(speed1[1])
    else:
        print("failed")
    print("Speed2:")
    if speed2[0]:
        print(speed2[1])
    else:
        print("failed ")


version = rc.read_version()
if version[0] == False:
    print("GETVERSION Failed")
else:
    print(repr(version[1]))

while 1:
    rc.speed_distance_m1(12000, 48000, 1)
    rc.speed_distance_m2(-12000, 48000, 1)
    buffers = (0, 0, 0)
    # Loop until distance command has completed
    while(buffers[1] != 0x80 and buffers[2] != 0x80):
        displayspeed()
        buffers = rc.read_buffer_length()

    time.sleep(2)

    rc.speed_distance_m1(-12000, 48000, 1)
    rc.speed_distance_m2(12000, 48000, 1)
    buffers = (0, 0, 0)
    # Loop until distance command has completed
    while(buffers[1] != 0x80 and buffers[2] != 0x80):
        displayspeed()
        buffers = rc.read_buffer_length()

    # When no second command is given the motors will automatically slow down to 0␣
↪which takes 1 second
```

(continues on next page)

(continued from previous page)

```
    time.sleep(2)

    rc.speed_distance_m1(12000, 48000, 1)
    rc.speed_distance_m2(-12000, 48000, 1)
    rc.speed_distance_m1(-12000, 48000, 0)
    rc.speed_distance_m2(12000, 48000, 0)
    rc.speed_distance_m1(0, 48000, 0)
    rc.speed_distance_m2(0, 48000, 0)
    buffers = (0, 0, 0)
    # Loop until distance command has completed
    while(buffers[1] != 0x80 and buffers[2] != 0x80):
        displayspeed()
        buffers = rc.read_buffer_length()

    time.sleep(1)
```

Mixed Speed and Acceleration

Listing 8: examples/roboclaw_mixedspeedaccel.py

```
# ***Before using this example the motor/controller combination must be
# ***tuned and the settings saved to the Roboclaw using IonMotion.
# ***The Min and Max Positions must be at least 0 and 50000

import time
from roboclaw import Roboclaw
try:  # if on win32 or linux
    from serial import SerialException, Serial as UART
except ImportError:
    try:  # try CircuitPython
        from board import UART
    except ImportError:
        try:  # try MicroPythom
            from roboclaw.usart_serial_ctx import SerialUART as UART

# Windows comport name
# rc = Roboclaw(UART("COM3", 115200))
# Linux comport name
# rc = Roboclaw(UART("/dev/ttyACM0", 115200))
# if CircuitPython or MicroPythom
rc = Roboclaw(UART(), address=0x80)


def displayspeed():
    enc1 = rc.read_encoder_m1()
    enc2 = rc.read_encoder_m2()
    speed1 = rc.read_speed_m1()
    speed2 = rc.read_speed_m2()

     print("Encoder1:")
       if enc1[0] == 1:
            print(enc1[1])
            print(format(enc1[2], '02x'))
        else:
            print("failed")
```

(continues on next page)

(continued from previous page)

```
        print("Encoder2:")
        if enc2[0] == 1:
            print(enc2[1])
            print(format(enc2[2], '02x'))
        else:
            print("failed ")
        print("Speed1:")
        if speed1[0]:
            print(speed1[1])
        else:
            print("failed")
        print("Speed2:")
        if(speed2[0]):
            print(speed2[1])
        else:
            print("failed ")


version = rc.ReadVersion()
if version[0] == False:
    print("GETVERSION Failed")
else:
    print(repr(version[1]))

while 1:
    rc.speed_accel_m1_m2(12000, 12000, -12000)
    for i in range(0, 200):
        displayspeed()
        time.sleep(0.01)

    rc.speed_accel_m1_m2(12000, -12000, 12000)
    for i in range(0, 200):
        displayspeed()
        time.sleep(0.01)
```

Position

Listing 9: examples/roboclaw_position.py

```
# ***Before using this example the motor/controller combination must be
# ***tuned and the settings saved to the Roboclaw using IonMotion.
# ***The Min and Max Positions must be at least 0 and 50000

import time
from roboclaw import Roboclaw
try:  # if on win32 or linux
    from serial import SerialException, Serial as UART
except ImportError:
    try:  # try CircuitPython
        from board import UART
    except ImportError:
        try:  # try MicroPythom
            from roboclaw.usart_serial_ctx import SerialUART as UART

# Windows comport name
```

(continues on next page)

(continued from previous page)

```
# rc = Roboclaw(UART("COM3", 115200))
# Linux comport name
# rc = Roboclaw(UART("/dev/ttyACM0", 115200))
# if CircuitPython or MicroPythom
rc = Roboclaw(UART(), address=0x80)


def displayspeed():
    enc1 = rc.read_encoder_m1()
    enc2 = rc.read_encoder_m2()
    speed1 = rc.read_speed_m1()
    speed2 = rc.read_speed_m2()

    print("Encoder1:")
    if enc1[0] == 1:
        print(enc1[1])
        print(format(enc1[2], '02x'))
    else:
        print("failed")
    print("Encoder2:")
    if enc2[0] == 1:
        print(enc2[1])
        print(format(enc2[2], '02x'))
    else:
        print("failed ")
    print("Speed1:")
    if speed1[0]:
        print(speed1[1])
    else:
        print("failed")
    print("Speed2:")
    if speed2[0]:
        print(speed2[1])
    else:
        print("failed ")


while 1:
    print("Pos 50000")
    rc.speed_accel_deccel_position_m1(32000, 12000, 32000, 50000, 0)
    for i in range(0, 80):
        displayspeed()
        time.sleep(0.1)

    time.sleep(2)

    print("Pos 0")
    rc.speed_accel_deccel_position_m1(32000, 12000, 32000, 0, 0)
    for i in range(0, 80):
        displayspeed()
        time.sleep(0.1)

    time.sleep(2)
```

RC mode Pulses

Listing 10: examples/roboclaw_rcpulses.py

```
""" On Roboclaw set switch 1 and 6 on. """
import time
from board import SCL, SDA
import busio

# Import the PCA9685 module. Available in the bundle and here:
#   https://github.com/adafruit/Adafruit_CircuitPython_PCA9685
from adafruit_pca9685 import PCA9685

from adafruit_motor import servo

i2c = busio.I2C(SCL, SDA)

# Create a simple PCA9685 class instance.
pca = PCA9685(i2c)
# You can optionally provide a finer tuned reference clock speed to improve the␣
↪accuracy of the
# timing pulses. This calibration will be specific to each board and its environment.␣
↪See the
# calibration.py example in the PCA9685 driver.
# pca = PCA9685(i2c, reference_clock_speed=25630710)
pca.frequency = 50

# The pulse range is [1250 (full forward), 1750 (full reverse)].
pulses[
    servo.ContinuousServo(pca.channels[7], min_pulse=1250, max_pulse=1750),
    servo.ContinuousServo(pca.channels[8], min_pulse=1250, max_pulse=1750)
]

pulses[0].throttle = 1
pulses[1].throttle = -1
time.sleep(2)

# stop
pulses[0].throttle = 0
pulses[1].throttle = 0

pca.deinit()
```

RC mode Pulses Mixed

Listing 11: examples/roboclaw_rcpulsemixed.py

```
# On Roboclaw set switch 1 and 6 on. <-- what does this refer to?
# mode 2 option 4 <-- my note based on user manual pg 26
import time
from board import SCL, SDA
import busio

# Import the PCA9685 module. Available in the bundle and here:
#   https://github.com/adafruit/Adafruit_CircuitPython_PCA9685
from adafruit_pca9685 import PCA9685

```

(continues on next page)

(continued from previous page)

```
from adafruit_motor import servo

i2c = busio.I2C(SCL, SDA)

# Create a simple PCA9685 class instance.
pca = PCA9685(i2c)
# You can optionally provide a finer tuned reference clock speed to improve the accuracy of the
# timing pulses. This calibration will be specific to each board and its environment. See the
# calibration.py example in the PCA9685 driver.
# pca = PCA9685(i2c, reference_clock_speed=25630710)
pca.frequency = 50

# The pulse range is [1250 (full forward), 1750 (full reverse)].
pulses[
    servo.ContinuousServo(pca.channels[7], min_pulse=1250, max_pulse=1750),
    servo.ContinuousServo(pca.channels[8], min_pulse=1250, max_pulse=1750)
]

# TODO MATH OF PULSE INTO FLOAT RANGE [-1, 1]
while 1:
  //forward
  pulses[0].throttle = 1 # writeMicroseconds(1600);
  pulses[1].throttle = 1 # writeMicroseconds(1500);
  time.sleep(2)

  //backward
  pulses[0].throttle = 1 # writeMicroseconds(1400);
  pulses[1].throttle = 1 # writeMicroseconds(1500);
  time.sleep(2)

  //left
  pulses[0].throttle = 1 # writeMicroseconds(1500);
  pulses[1].throttle = 1 # writeMicroseconds(1600);
  time.sleep(2)

  //right
  pulses[0].throttle = 1 # writeMicroseconds(1500);
  pulses[1].throttle = 1 # writeMicroseconds(1400);
  time.sleep(2)

  //mixed forward/left
  pulses[0].throttle = 1 # writeMicroseconds(1600);
  pulses[1].throttle = 1 # writeMicroseconds(1600);
  time.sleep(2)

  //mixed forward/right
  pulses[0].throttle = 1 # writeMicroseconds(1600);
  pulses[1].throttle = 1 # writeMicroseconds(1400);
  time.sleep(2)

  //mixed backward/left
  pulses[0].throttle = 1 # writeMicroseconds(1400);
  pulses[1].throttle = 1 # writeMicroseconds(1600);
  time.sleep(2)

```

(continues on next page)

(continued from previous page)

```
  //mixed backward/right
  pulses[0].throttle = 1 # writeMicroseconds(1400);
  pulses[1].throttle = 1 # writeMicroseconds(1400);
  time.sleep(2)

}
```

Read Various Data

Listing 12: examples/roboclaw_read.py

```python
import time
from roboclaw import Roboclaw
try:  # if on win32 or linux
    from serial import SerialException, Serial as UART
except ImportError:
    try:  # try CircuitPython
        from board import UART
    except ImportError:
        try:  # try MicroPythom
            from roboclaw.usart_serial_ctx import SerialUART as UART

# Windows comport name
# rc = Roboclaw(UART("COM3", 115200))
# Linux comport name
# rc = Roboclaw(UART("/dev/ttyACM0", 115200))
# if CircuitPython or MicroPythom
rc = Roboclaw(UART(), address=0x80)


def displayspeed():
    enc1 = rc.read_encoder_m1()
    enc2 = rc.read_encoder_m2()
    speed1 = rc.read_speed_m1()
    speed2 = rc.read_speed_m2()

     print("Encoder1:")
       if enc1[0] == 1:
            print(enc1[1])
            print(format(enc1[2], '02x'))
        else:
            print("failed")
        print("Encoder2:")
        if enc2[0] == 1:
            print(enc2[1])
            print(format(enc2[2], '02x'))
        else:
            print("failed ")
        print("Speed1:")
        if speed1[0]:
            print(speed1[1])
        else:
            print("failed")
        print("Speed2:")
        if speed2[0]:
```

(continues on next page)

(continued from previous page)

```
            print(speed2[1])
        else:
            print("failed ")


version = rc.ReadVersion()
if version[0] == False:
    print("GETVERSION Failed")
else:
    print(repr(version[1]))


def test(loop=2):
    while loop:
        displayspeed()
        loop -= 1
```

Read Version

Listing 13: examples/roboclaw_readversion.py

```
import time
from roboclaw import Roboclaw
try:  # if on win32 or linux
    from serial import SerialException, Serial as UART
except ImportError:
    try:  # try CircuitPython
        from board import UART
    except ImportError:
        try:  # try MicroPythom
            from roboclaw.usart_serial_ctx import SerialUART as UART

# Windows comport name
# rc = Roboclaw(UART("COM3", 115200))
# Linux comport name
# rc = Roboclaw(UART("/dev/ttyACM0", 115200))
# if CircuitPython or MicroPythom
rc = Roboclaw(UART(), address=0x80)

while 1:
        # Get version string
    version = rc.read_version()
    if version[0] == False:
        print("GETVERSION Failed")
    else:
        print(repr(version[1]))
    time.sleep(1)
```

Read EEPROM

Listing 14: examples/roboclaw_readeeprom.py

```
import time
from roboclaw import Roboclaw
try:  # if on win32 or linux
    from serial import SerialException, Serial as UART
except ImportError:
    try:  # try CircuitPython
        from board import UART
    except ImportError:
        try:  # try MicroPythom
            from roboclaw.usart_serial_ctx import SerialUART as UART

# Windows comport name
# rc = Roboclaw(UART("COM3", 115200))
# Linux comport name
# rc = Roboclaw(UART("/dev/ttyACM0", 115200))
# if CircuitPython or MicroPythom
rc = Roboclaw(UART(), address=0x80)

# Get version string
for x in range(0, 255):
    value = rc.read_eeprom(x)
    print("EEPROM:")
    print(x)
    print(" ")
    if value[0] == False:
        print("Failed")
    else:
        print(value[1])
```

Write EEPROM

Listing 15: examples/roboclaw_writeeeprom.py

```
from roboclaw import Roboclaw
try:  # if on win32 or linux
    from serial import SerialException, Serial as UART
except ImportError:
    try:  # try CircuitPython
        from board import UART
    except ImportError:
        try:  # try MicroPythom
            from roboclaw.usart_serial_ctx import SerialUART as UART

# Windows comport name
# rc = Roboclaw(UART("COM3", 115200))
# Linux comport name
# rc = Roboclaw(UART("/dev/ttyACM0", 115200))
# if CircuitPython or MicroPythom
rc = Roboclaw(UART(), address=0x80)

# Get version string
for x in range(0, 255):
    value = rc.write_eeprom(x, x * 2)
```

(continues on next page)

(continued from previous page)

```
    print("EEPROM:")
    print(x)
    print(" ")
    if not value:
        print("Failed")
    else:
        print("Written")
```

2.1.2 Roboclaw

Roboclaw driver class

roboclaw driver module contains the roboclaw driver class that controls the roboclaw via a UART serial

class roboclaw.roboclaw.**Roboclaw**(*serial_obj*, *address=128*, *retries=3*, *packet_serial=True*)

A driver class for the RoboClaw Motor Controller device.

Parameters

- **serial_obj** (*Serial*) – The serial obj associated with the serial port that is connected to the RoboClaw.
- **address** (*int*) – The unique address assigned to the particular RoboClaw. Valid addresses range [0x80, 0x87].
- **retries** (*int*) – The amount of attempts to read/write data over the serial port. Defaults to 3.

packet_serial = None

this bool represents if using packet serial mode.

address

The Address of the specific Roboclaw device on the object's serial port Must be in range [0x80, 0x87]

send_random_data(*cnt*, *address=None*)

Send some randomly generated data of of a certain length. Don't know what this would be used for, but it was in the original driver code…

Parameters **cnt** (*int*) – the number of bytes to randomly generate.

forward_m1(*val*, *address=None*)

Drive motor 1 forward.

Parameters **val** (*int*) – Valid data range is 0 - 127. A value of 127 = full speed forward, 64 = about half speed forward and 0 = full stop.

backward_m1(*val*, *address=None*)

Drive motor 1 backwards.

Parameters **val** (*int*) – Valid data range is 0 - 127. A value of 127 full speed backwards, 64 = about half speed backward and 0 = full stop.

set_min_voltage_main_battery(*val*, *address=None*)

Sets main battery (B- / B+) minimum voltage level. If the battery voltages drops below the set voltage level, RoboClaw will stop driving the motors. The voltage is set in .2 volt increments. The minimum value allowed which is 6V.

Parameters **val** (*float*) – The valid data range is [6, 34] Volts.

set_max_voltage_main_battery(*val*, *address=None*)
Sets main battery (B- / B+) maximum voltage level. During regenerative breaking a back voltage is applied to charge the battery. When using a power supply, by setting the maximum voltage level, RoboClaw will, before exceeding it, go into hard braking mode until the voltage drops below the maximum value set. This will prevent overvoltage conditions when using power supplies.

Parameters **val** (*float*) – The valid data range is [6, 34] Volts.

forward_m2(*val*, *address=None*)
Drive motor 2 forward.

Parameters **val** (*int*) – Valid data range is [0, 127]. A value of 127 full speed forward, 64 = about half speed forward and 0 = full stop.

backward_m2(*val*, *address=None*)
Drive motor 2 backwards.

Parameters **val** (*int*) – Valid data range is [0, 127]. A value of 127 full speed backwards, 64 = about half speed backward and 0 = full stop.

forward_backward_m1(*val*, *address=None*)
Drive motor 1 forward or reverse.

Parameters **val** (*int*) – Valid data range is [0, 127]. A value of 0 = full speed reverse, 64 = stop and 127 = full speed forward.

forward_backward_m2(*val*, *address=None*)
Drive motor 2 forward or reverse.

Parameters **val** (*int*) – Valid data range is [0, 127]. A value of 0 = full speed reverse, 64 = stop and 127 = full speed forward.

forward_mixed(*val*, *address=None*)
Drive forward in mix mode.

Parameters **val** (*int*) – Valid data range is [0, 127]. A value of 0 = full stop and 127 = full forward.

backward_mixed(*val*, *address=None*)
Drive backwards in mix mode.

Parameters **val** (*int*) – Valid data range is [0, 127]. A value of 0 = full stop and 127 = full reverse.

turn_right_mixed(*val*, *address=None*)
Turn right in mix mode.

Parameters **val** (*int*) – Valid data range is [0, 127]. A value of 0 = stop turn and 127 = full speed turn.

turn_left_mixed(*val*, *address=None*)
Turn left in mix mode.

Parameters **val** (*int*) – Valid data range is [0, 127]. A value of 0 = stop turn and 127 = full speed turn.

forward_backward_mixed(*val*, *address=None*)
Drive forward or backwards.

Parameters **val** (*int*) – Valid data range is [0, 127]. A value of 0 = full backward, 64 = stop and 127 = full forward.

left_right_mixed(*val*, *address=None*)
Turn left or right.

Parameters **`val`** (*int*) – Valid data range is [0, 127]. A value of 0 = full left, 64 = stop turn and 127 = full right.

`read_encoder_m1`(*address=None*)
Read M1 encoder count/position.

Returns [Enc1(4 bytes), Status, crc16(2 bytes)]

Quadrature encoders have a range of 0 to 4,294,967,295. Absolute encoder values are converted from an analog voltage into a value from 0 to 4095 for the full 5.1v range.

The status byte tracks counter underflow, direction and overflow. The byte value represents:

- Bit0 - Counter Underflow (1= Underflow Occurred, Clear After Reading)
- Bit1 - Direction (0 = Forward, 1 = Backwards)
- Bit2 - Counter Overflow (1= Underflow Occurred, Clear After Reading)
- Bit3 through Bit7 - Reserved

`read_encoder_m2`(*address=None*)
Read M2 encoder count/position.

Returns [EncoderCount(4 bytes), Status]

Quadrature encoders have a range of 0 to 4,294,967,295. Absolute encoder values are converted from an analog voltage into a value from 0 to 4095 for the full 5.1v range.

The Status byte tracks counter underflow, direction and overflow. The byte value represents:

- Bit0 - Counter Underflow (1= Underflow Occurred, Cleared After Reading)
- Bit1 - Direction (0 = Forward, 1 = Backwards)
- Bit2 - Counter Overflow (1= Underflow Occurred, Cleared After Reading)
- Bit3 through Bit7 - Reserved

`read_speed_m1`(*address=None*)
Read M1 counter speed. Returned value is in pulses per second. MCP keeps track of how many pulses received per second for both encoder channels.

Returns [Speed(4 bytes), Status]

Status indicates the direction (0 – forward, 1 - backward).

`read_speed_m2`(*address=None*)
Read M2 counter speed. Returned value is in pulses per second. MCP keeps track of how many pulses received per second for both encoder channels.

Returns [Speed(4 bytes), Status]

Status indicates the direction (0 – forward, 1 - backward).

`reset_encoders`(*address=None*)
Will reset both quadrature decoder counters to zero. This command applies to quadrature encoders only.

`read_version`(*address=None*)
Read RoboClaw firmware version. Returns up to 48 bytes(depending on the Roboclaw model) and is terminated by a line feed character and a null character.

Returns ["MCP266 2x60A v1.0.0",10,0]

The command will return up to 48 bytes. The return string includes the product name and firmware version. The return string is terminated with a line feed (10) and null (0) character.

set_enc_m1 (*cnt*, *address=None*)
Set the value of the Encoder 1 register. Useful when homing motor 1. This command applies to quadrature encoders only.

set_enc_m2 (*cnt*, *address=None*)
Set the value of the Encoder 2 register. Useful when homing motor 2. This command applies to quadrature encoders only.

read_main_battery_voltage (*address=None*)
Read the main battery voltage level connected to B+ and B- terminals.

Returns The voltage is returned in 10ths of a volt (eg 30.0).

read_logic_battery_voltage (*address=None*)
Read a logic battery voltage level connected to LB+ and LB- terminals. The voltage is returned in 10ths of a volt(eg 50 = 5v).

Returns [Value.Byte1, Value.Byte0]

set_min_voltage_logic_battery (*val*, *address=None*)
Sets logic input (LB- / LB+) minimum voltage level. RoboClaw will shut down with an error if the voltage is below this level. The voltage is set in .2 volt increments. The minimum value allowed which is 6V.

Parameters **val** (*float*) – The valid data range is [6, 34].

Note: This command is included for backwards compatibility. We recommend you use `set_logic_voltages()` instead.

set_max_voltage_logic_battery (*val*, *address=None*)
Sets logic input (LB- / LB+) maximum voltage level. RoboClaw will shutdown with an error if the voltage is above this level.

Parameters **val** (*float*) – The valid data range is [6, 34].

Note: This command is included for backwards compatibility. We recommend you use `set_main_voltages()` instead.

set_m1_velocity_pid (*p*, *i*, *d*, *qpps*, *address=None*)
Several motor and quadrature combinations can be used with RoboClaw. In some cases the default PID values will need to be tuned for the systems being driven. This gives greater flexibility in what motor and encoder combinations can be used. The RoboClaw PID system consist of four constants starting with QPPS, P = Proportional, I= Integral and D= Derivative.

Parameters

- **p** (*int*) – The default P is 0x00010000.
- **i** (*int*) – The default I is 0x00008000.
- **d** (*int*) – The default D is 0x00004000.
- **qpps** (*int*) – The default QPPS is 44000.

QPPS is the speed of the encoder when the motor is at 100% power. P, I, D are the default values used after a reset.

set_m2_velocity_pid (*p*, *i*, *d*, *qpps*, *address=None*)
Several motor and quadrature combinations can be used with RoboClaw. In some cases the default PID values will need to be tuned for the systems being driven. This gives greater flexibility in what motor

and encoder combinations can be used. The RoboClaw PID system consist of four constants starting with QPPS, P = Proportional, I= Integral and D= Derivative.

Parameters

- **p** (*int*) – The default P is 0x00010000.
- **i** (*int*) – The default I is 0x00008000.
- **d** (*int*) – The default D is 0x00004000.
- **qpps** (*int*) – The default QPPS is 44000.

QPPS is the speed of the encoder when the motor is at 100% power. P, I, D are the default values used after a reset.

read_raw_speed_m1(*address=None*)

Read the pulses counted in that last 300th of a second. This is an unfiltered version of *read_speed_m1()*. This function can be used to make a independent PID routine. Value returned is in encoder counts per second.

Returns [Speed(4 bytes), Status]

The Status byte is direction (0 – forward, 1 - backward).

read_raw_speed_m2(*address=None*)

Read the pulses counted in that last 300th of a second. This is an unfiltered version of *read_speed_m2()*. This function can be used to make a independent PID routine. Value returned is in encoder counts per second.

Returns [Speed(4 bytes), Status]

The Status byte is direction (0 – forward, 1 - backward).

duty_m1(*val*, *address=None*)

Drive M1 using a duty cycle value. The duty cycle is used to control the speed of the motor without a quadrature encoder.

Parameters **val** (*int*) – The duty value is signed and the range [-32767, 32767] (eg. +-100% duty).

duty_m2(*val*, *address=None*)

Drive M2 using a duty cycle value. The duty cycle is used to control the speed of the motor without a quadrature encoder.

Parameters **val** (*int*) – The duty value is signed and the range [-32767, 32767] (eg. +-100% duty).

duty_m1_m2(*m1*, *m2*, *address=None*)

Drive both M1 and M2 using a duty cycle value. The duty cycle is used to control the speed of the motor without a quadrature encoder.

Parameters

- **m1** (*int*) – The duty value is signed and the range [-32767, 32767] (eg. +-100% duty).
- **m2** (*int*) – The duty value is signed and the range [-32767, 32767] (eg. +-100% duty).

speed_m1(*val*, *address=None*)

Drive M1 using a speed value. The sign indicates which direction the motor will turn. This command is used to drive the motor by quad pulses per second. Different quadrature encoders will have different rates at which they generate the incoming pulses. The values used will differ from one encoder to another. Once a value is sent the motor will begin to accelerate as fast as possible until the defined rate is reached.

Parameters **val** (*int*) – Valid input ranges [-2147483647, 2147483647].

speed_m2 (*val*, *address=None*)
Drive M2 with a speed value. The sign indicates which direction the motor will turn. This command is used to drive the motor by quad pulses per second. Different quadrature encoders will have different rates at which they generate the incoming pulses. The values used will differ from one encoder to another. Once a value is sent, the motor will begin to accelerate as fast as possible until the rate defined is reached.

Parameters **val** (*int*) – Valid input ranges [-2147483647, 2147483647].

speed_m1_m2 (*m1*, *m2*, *address=None*)
Drive M1 and M2 in the same command using a signed speed value. The sign indicates which direction the motor will turn. This command is used to drive the motor by quad pulses per second. Different quadrature encoders will have different rates at which they generate the incoming pulses. The values used will differ from one encoder to another. Once a value is sent the motor will begin to accelerate as fast as possible until the rate defined is reached.

Parameters

- **m1** (*int*) – Valid input ranges [-2147483647, 2147483647].
- **m2** (*int*) – Valid input ranges [-2147483647, 2147483647].

speed_accel_m1 (*accel*, *speed*, *address=None*)
Drive M1 with a signed speed and acceleration value. The sign indicates which direction the motor will run. The acceleration values are not signed. This command is used to drive the motor by quad pulses per second and using an acceleration value for ramping. Different quadrature encoders will have different rates at which they generate the incoming pulses. The values used will differ from one encoder to another. Once a value is sent the motor will begin to accelerate incrementally until the rate defined is reached.

The acceleration is measured in speed increase per second. An acceleration value of 12,000 QPPS with a speed of 12,000 QPPS would accelerate a motor from 0 to 12,000 QPPS in 1 second. Another example would be an acceleration value of 24,000 QPPS and a speed value of 12,000 QPPS would accelerate the motor to 12,000 QPPS in 0.5 seconds.

speed_accel_m2 (*accel*, *speed*, *address=None*)
Drive M2 with a signed speed and acceleration value. The sign indicates which direction the motor will run. The acceleration value is not signed. This command is used to drive the motor by quad pulses per second and using an acceleration value for ramping. Different quadrature encoders will have different rates at which they generate the incoming pulses. The values used will differ from one encoder to another. Once a value is sent the motor will begin to accelerate incrementally until the rate defined is reached.

The acceleration is measured in speed increase per second. An acceleration value of 12,000 QPPS with a speed of 12,000 QPPS would accelerate a motor from 0 to 12,000 QPPS in 1 second. Another example would be an acceleration value of 24,000 QPPS and a speed value of 12,000 QPPS would accelerate the motor to 12,000 QPPS in 0.5 seconds.

speed_accel_m1_m2 (*accel*, *speed1*, *speed2*, *address=None*)
Drive M1 and M2 in the same command using one value for acceleration and two signed speed values for each motor. The sign indicates which direction the motor will run. The acceleration value is not signed. The motors are sync during acceleration. This command is used to drive the motor by quad pulses per second and using an acceleration value for ramping. Different quadrature encoders will have different rates at which they generate the incoming pulses. The values used will differ from one encoder to another. Once a value is sent the motor will begin to accelerate incrementally until the rate defined is reached.

The acceleration is measured in speed increase per second. An acceleration value of 12,000 QPPS with a speed of 12,000 QPPS would accelerate a motor from 0 to 12,000 QPPS in 1 second. Another example would be an acceleration value of 24,000 QPPS and a speed value of 12,000 QPPS would accelerate the motor to 12,000 QPPS in 0.5 seconds.

speed_distance_m1 (*speed*, *distance*, *buffer*, *address=None*)
Drive M1 with a signed speed and distance value. The sign indicates which direction the motor will run.

The distance value is not signed. This command is buffered. This command is used to control the top speed and total distance traveled by the motor. Each motor channel M1 and M2 have separate buffers. This command will execute immediately if no other command for that channel is executing, otherwise the command will be buffered in the order it was sent. Any buffered or executing command can be stopped when a new command is issued by setting the Buffer argument. All values used are in quad pulses per second.

The Buffer argument can be set to a 1 or 0. If a value of 0 is used the command will be buffered and executed in the order sent. If a value of 1 is used the current running command is stopped, any other commands in the buffer are deleted and the new command is executed.

speed_distance_m2(*speed*, *distance*, *buffer*, *address=None*)

Drive M2 with a speed and distance value. The sign indicates which direction the motor will run. The distance value is not signed. This command is buffered. Each motor channel M1 and M2 have separate buffers. This command will execute immediately if no other command for that channel is executing, otherwise the command will be buffered in the order it was sent. Any buffered or executing command can be stopped when a new command is issued by setting the Buffer argument. All values used are in quad pulses per second.

The Buffer argument can be set to a 1 or 0. If a value of 0 is used the command will be buffered and executed in the order sent. If a value of 1 is used the current running command is stopped, any other commands in the buffer are deleted and the new command is executed.

speed_distance_m1_m2(*speed1*, *distance1*, *speed2*, *distance2*, *buffer*, *address=None*)

Drive M1 and M2 with a speed and distance value. The sign indicates which direction the motor will run. The distance value is not signed. This command is buffered. Each motor channel M1 and M2 have separate buffers. This command will execute immediately if no other command for that channel is executing, otherwise the command will be buffered in the order it was sent. Any buffered or executing command can be stopped when a new command is issued by setting the Buffer argument. All values used are in quad pulses per second.

The Buffer argument can be set to a 1 or 0. If a value of 0 is used the command will be buffered and executed in the order sent. If a value of 1 is used the current running command is stopped, any other commands in the buffer are deleted and the new command is executed.

speed_accel_distance_m1(*accel*, *speed*, *distance*, *buffer*, *address=None*)

Drive M1 with a speed, acceleration and distance value. The sign indicates which direction the motor will run. The acceleration and distance values are not signed. This command is used to control the motors top speed, total distanced traveled and at what incremental acceleration value to use until the top speed is reached. Each motor channel M1 and M2 have separate buffers. This command will execute immediately if no other command for that channel is executing, otherwise the command will be buffered in the order it was sent. Any buffered or executing command can be stopped when a new command is issued by setting the Buffer argument. All values used are in quad pulses per second.

The Buffer argument can be set to a 1 or 0. If a value of 0 is used the command will be buffered and executed in the order sent. If a value of 1 is used the current running command is stopped, any other commands in the buffer are deleted and the new command is executed.

speed_accel_distance_m2(*accel*, *speed*, *distance*, *buffer*, *address=None*)

Drive M2 with a speed, acceleration and distance value. The sign indicates which direction the motor will run. The acceleration and distance values are not signed. This command is used to control the motors top speed, total distanced traveled and at what incremental acceleration value to use until the top speed is reached. Each motor channel M1 and M2 have separate buffers. This command will execute immediately if no other command for that channel is executing, otherwise the command will be buffered in the order it was sent. Any buffered or executing command can be stopped when a new command is issued by setting the Buffer argument. All values used are in quad pulses per second.

The Buffer argument can be set to a 1 or 0. If a value of 0 is used the command will be buffered and executed in the order sent. If a value of 1 is used the current running command is stopped, any other commands in the buffer are deleted and the new command is executed.

speed_accel_distance_m1_m2 (*accel*, *speed1*, *distance1*, *speed2*, *distance2*, *buffer*, *address=None*)

Drive M1 and M2 with a speed, acceleration and distance value. The sign indicates which direction the motor will run. The acceleration and distance values are not signed. This command is used to control both motors top speed, total distanced traveled and at what incremental acceleration value to use until the top speed is reached. Each motor channel M1 and M2 have separate buffers. This command will execute immediately if no other command for that channel is executing, otherwise the command will be buffered in the order it was sent. Any buffered or executing command can be stopped when a new command is issued by setting the Buffer argument. All values used are in quad pulses per second.

The Buffer argument can be set to a 1 or 0. If a value of 0 is used the command will be buffered and executed in the order sent. If a value of 1 is used the current running command is stopped, any other commands in the buffer are deleted and the new command is executed.

read_buffer_length (*address=None*)

Read both motor M1 and M2 buffer lengths. This command can be used to determine how many commands are waiting to execute.

Returns [BufferM1, BufferM2]

The return values represent how many commands per buffer are waiting to be executed. The maximum buffer size per motor is 64 commands(0x3F). A return value of 0x80(128) indicates the buffer is empty. A return value of 0 indiciates the last command sent is executing. A value of 0x80 indicates the last command buffered has finished.

read_pwms (*address=None*)

Read the current PWM output values for the motor channels. The values returned are +/-32767. The duty cycle percent is calculated by dividing the Value by 327.67.

Returns [M1 PWM(2 bytes), M2 PWM(2 bytes)]

read_currents (*address=None*)

Read the current draw from each motor in 10ma increments. The amps value is calculated by dividing the value by 100.

Returns [M1 Current(2 bytes), M2 Currrent(2 bytes)]

speed_accel_m1_m2_2 (*accel1*, *speed1*, *accel2*, *speed2*, *address=None*)

Drive M1 and M2 in the same command using one value for acceleration and two signed speed values for each motor. The sign indicates which direction the motor will run. The acceleration value is not signed. The motors are sync during acceleration. This command is used to drive the motor by quad pulses per second and using an acceleration value for ramping. Different quadrature encoders will have different rates at which they generate the incoming pulses. The values used will differ from one encoder to another. Once a value is sent the motor will begin to accelerate incrementally until the rate defined is reached.

The acceleration is measured in speed increase per second. An acceleration value of 12,000 QPPS with a speed of 12,000 QPPS would accelerate a motor from 0 to 12,000 QPPS in 1 second. Another example would be an acceleration value of 24,000 QPPS and a speed value of 12,000 QPPS would accelerate the motor to 12,000 QPPS in 0.5 seconds.

speed_accel_distance_m1_m2_2 (*accel1*, *speed1*, *distance1*, *accel2*, *speed2*, *distance2*, *buffer*, *address=None*)

Drive M1 and M2 in the same command using one value for acceleration and two signed speed values for each motor. The sign indicates which direction the motor will run. The acceleration value is not signed. The motors are sync during acceleration. This command is used to drive the motor by quad pulses per second and using an acceleration value for ramping. Different quadrature encoders will have different rates at which they generate the incoming pulses. The values used will differ from one encoder to another. Once a value is sent the motor will begin to accelerate incrementally until the rate defined is reached.

The acceleration is measured in speed increase per second. An acceleration value of 12,000 QPPS with a speed of 12,000 QPPS would accelerate a motor from 0 to 12,000 QPPS in 1 second. Another example would be an acceleration value of 24,000 QPPS and a speed value of 12,000 QPPS would accelerate the motor to 12,000 QPPS in 0.5 seconds.

duty_accel_m1(*accel*, *duty*, *address=None*)
Drive M1 with a signed duty and acceleration value. The sign indicates which direction the motor will run. The acceleration values are not signed. This command is used to drive the motor by PWM and using an acceleration value for ramping. Accel is the rate per second at which the duty changes from the current duty to the specified duty.

The duty value is signed and the range is -32768 to +32767(eg. +-100% duty). The accel value range is 0 to 655359(eg maximum acceleration rate is -100% to 100% in 100ms).

duty_accel_m2(*accel*, *duty*, *address=None*)
Drive M2 with a signed duty and acceleration value. The sign indicates which direction the motor will run. The acceleration values are not signed. This command is used to drive the motor by PWM and using an acceleration value for ramping. Accel is the rate at which the duty changes from the current duty to the specified dury.

The duty value is signed and the range is -32768 to +32767 (eg. +-100% duty). The accel value range is 0 to 655359 (eg maximum acceleration rate is -100% to 100% in 100ms).

duty_accel_m1_m2(*accel1*, *duty1*, *accel2*, *duty2*, *address=None*)
Drive M1 and M2 in the same command using acceleration and duty values for each motor. The sign indicates which direction the motor will run. The acceleration value is not signed. This command is used to drive the motor by PWM using an acceleration value for ramping.

The duty value is signed and the range is -32768 to +32767 (eg. +-100% duty). The accel value range is 0 to 655359 (eg maximum acceleration rate is -100% to 100% in 100ms).

read_m1_velocity_pid(*address=None*)
Read the PID and QPPS Settings.

Returns [P(4 bytes), I(4 bytes), D(4 bytes), QPPS(4 byte)]

read_m2_velocity_pid(*address=None*)
Read the PID and QPPS Settings.

Returns [P(4 bytes), I(4 bytes), D(4 bytes), QPPS(4 byte)]

set_main_voltages(*minimum*, *maximum*, *address=None*)
Set the Main Battery Voltage cutoffs, Min and Max. Min and Max voltages are in 10th of a volt increments. Multiply the voltage to set by 10.

set_logic_voltages(*minimum*, *maximum*, *address=None*)
Set the Logic Battery Voltages cutoffs, Min and Max. Min and Max voltages are in 10th of a volt increments. Multiply the voltage to set by 10.

read_min_max_main_voltages(*address=None*)
Read the Main Battery Voltage Settings. The voltage is calculated by dividing the value by 10

Returns [Min(2 bytes), Max(2 bytes)]

read_min_max_logic_voltages(*address=None*)
Read the Logic Battery Voltage Settings. The voltage is calculated by dividing the value by 10

Returns [Min(2 bytes), Max(2 bytes)]

set_m1_position_pid(*kp*, *ki*, *kd*, *kimax*, *deadzone*, *minimum*, *maximum*, *address=None*)
The RoboClaw Position PID system consist of seven constants starting with P = Proportional, I= Integral and D= Derivative, MaxI = Maximum Integral windup, Deadzone in encoder counts, MinPos = Minimum Position and MaxPos = Maximum Position. The defaults values are all zero.

Position constants are used only with the Position commands, 65,66 and 67 or when encoders are enabled in RC/Analog modes.

set_m2_position_pid(*kp*, *ki*, *kd*, *kimax*, *deadzone*, *minimum*, *maximum*, *address=None*)
The RoboClaw Position PID system consist of seven constants starting with P = Proportional, I= Integral and D= Derivative, MaxI = Maximum Integral windup, Deadzone in encoder counts, MinPos = Minimum Position and MaxPos = Maximum Position. The defaults values are all zero.

Position constants are used only with the Position commands, 65,66 and 67 or when encoders are enabled in RC/Analog modes.

read_m1_position_pid(*address=None*)
Read the Position PID Settings.

Returns [P(4 bytes), I(4 bytes), D(4 bytes), MaxI(4 byte), Deadzone(4 byte), MinPos(4 byte), MaxPos(4 byte)]

read_m2_position_pid(*address=None*)
Read the Position PID Settings.

Returns [P(4 bytes), I(4 bytes), D(4 bytes), MaxI(4 byte), Deadzone(4 byte), MinPos(4 byte), MaxPos(4 byte)]

speed_accel_deccel_position_m1(*accel*, *speed*, *deccel*, *position*, *buffer*, *address=None*)
Move M1 position from the current position to the specified new position and hold the new position. Accel sets the acceleration value and deccel the decceleration value. QSpeed sets the speed in quadrature pulses the motor will run at after acceleration and before decceleration.

speed_accel_deccel_position_m2(*accel*, *speed*, *deccel*, *position*, *buffer*, *address=None*)
Move M2 position from the current position to the specified new position and hold the new position. Accel sets the acceleration value and deccel the decceleration value. QSpeed sets the speed in quadrature pulses the motor will run at after acceleration and before decceleration.

speed_accel_deccel_position_m1_m2(*accel1*, *speed1*, *deccel1*, *position1*, *accel2*, *speed2*, *deccel2*, *position2*, *buffer*, *address=None*)
Move M1 & M2 positions from their current positions to the specified new positions and hold the new positions. Accel sets the acceleration value and deccel the decceleration value. QSpeed sets the speed in quadrature pulses the motor will run at after acceleration and before decceleration.

set_m1_default_accel(*accel*, *address=None*)
Set the default acceleration for M1 when using duty cycle commands (`duty_m1()` and `duty_m1_m2()`) or when using Standard Serial, RC and Analog PWM modes.

set_m2_default_accel(*accel*, *address=None*)
Set the default acceleration for M2 when using duty cycle commands (`duty_m2()` and `duty_m1_m2()`) or when using Standard Serial, RC and Analog PWM modes.

set_pin_functions(*s3mode*, *s4mode*, *s5mode*, *address=None*)
Set modes for S3,S4 and S5.

Mode	S3mode	S4mode	S5mode
0	Default	Disabled	Disabled
1	E-Stop(latching)	E-Stop(latching)	E-Stop(latching)
2	E-Stop	E-Stop	E-Stop
3	Voltage Clamp	Voltage Clamp	Voltage Clamp
4		M1 Home	M2 Home

Mode Description

- Disabled: pin is inactive.
- Default: Flip switch if in RC/Analog mode or E-Stop(latching) in Serial modes.
- E-Stop(Latching): causes the Roboclaw to shutdown until the unit is power cycled.
- E-Stop: Holds the Roboclaw in shutdown until the E-Stop signal is cleared.
- Voltage Clamp: Sets the signal pin as an output to drive an external voltage clamp circuit.
- Home(M1 & M2): will trigger the specific motor to stop and the encoder count to reset to 0.

read_pin_functions(*address=None*)

Read mode settings for S3,S4 and S5. See `set_pin_functions()` for mode descriptions

Returns [S3mode, S4mode, S5mode]

set_deadband(*minimum*, *maximum*, *address=None*)

Set RC/Analog mode control deadband percentage in 10ths of a percent. Default value is 25(2.5%). Minimum value is 0(no DeadBand), Maximum value is 250(25%).

get_deadband(*address=None*)

Read DeadBand settings in 10ths of a percent.

Returns [Reverse, SForward]

restore_defaults(*address=None*)

Reset Settings to factory defaults.

read_temp(*address=None*)

Read the board temperature. Value returned is in 10ths of degrees.

Returns [Temperature(2 bytes)]

read_temp2(*address=None*)

Read the second board temperature(only on supported units). Value returned is in 10ths of degrees.

Returns [Temperature(2 bytes)]

read_error(*address=None*)

Read the current unit status.

Returns [Status]

Function	Status Bit Mask
Normal	0x0000
M1 OverCurrent Warning	0x0001
M2 OverCurrent Warning	0x0002
E-Stop	0x0004
Temperature Error	0x0008
Temperature2 Error	0x0010
Main Battery High Error	0x0020
Logic Battery High Error	0x0040
Logic Battery Low Error	0x0080
Main Battery High Warning	0x0400
Main Battery Low Warning	0x0800
Termperature Warning	0x1000
Temperature2 Warning	0x2000

read_encoder_modes(*address=None*)
Read the encoder pins assigned for both motors.

Returns [Enc1Mode, Enc2Mode]

set_m1_encoder_mode(*mode*, *address=None*)
Set the Encoder Pin for motor 1. See `read_encoder_modes()`.

set_m2_encoder_mode(*mode*, *address=None*)
Set the Encoder Pin for motor 2. See `read_encoder_modes()`.

write_nvm(*address=None*)
Writes all settings to non-volatile memory. Values will be loaded after each power up.

read_nvm(*address=None*)
Read all settings from non-volatile memory.

Returns [Enc1Mode, Enc2Mode]

> **Warning:** Concerning TTL Serial: If baudrate changes or the control mode changes communications will be lost.

set_config(*config*, *address=None*)
Set config bits for standard settings.

Function	Config Bit Mask
RC Mode	0x0000
Analog Mode	0x0001
Simple Serial Mode	0x0002
Packet Serial Mode	0x0003
Battery Mode Off	0x0000
Battery Mode Auto	0x0004
Battery Mode 2 Cell	0x0008
Battery Mode 3 Cell	0x000C
Battery Mode 4 Cell	0x0010
Battery Mode 5 Cell	0x0014
Battery Mode 6 Cell	0x0018
Battery Mode 7 Cell	0x001C

Continued on next page

Table 1 – continued from previous page

Function	Config Bit Mask
Mixing	0x0020
Exponential	0x0040
MCU	0x0080
BaudRate 2400	0x0000
BaudRate 9600	0x0020
BaudRate 19200	0x0040
BaudRate 38400	0x0060
BaudRate 57600	0x0080
BaudRate 115200	0x00A0
BaudRate 230400	0x00C0
BaudRate 460800	0x00E0
FlipSwitch	0x0100
Packet Address 0x80	0x0000
Packet Address 0x81	0x0100
Packet Address 0x82	0x0200
Packet Address 0x83	0x0300
Packet Address 0x84	0x0400
Packet Address 0x85	0x0500
Packet Address 0x86	0x0600
Packet Address 0x87	0x0700
Slave Mode	0x0800
Relay Mode	0x1000
Swap Encoders	0x2000
Swap Buttons	0x4000
Multi-Unit Mode	0x8000

Warning: Concerning TTL Serial: * If control mode is changed from packet serial mode when setting config communications will be lost! * If baudrate of packet serial mode is changed communications will be lost!

get_config(*address=None*)

Read config bits for standard settings See `set_config()`.

Returns [Config(2 bytes)]

set_m1_max_current(*maximum*, *address=None*)

Set Motor 1 Maximum Current Limit. Current value is in 10ma units. To calculate multiply current limit by 100.

set_m2_max_current(*maximum*, *address=None*)

Set Motor 2 Maximum Current Limit. Current value is in 10ma units. To calculate multiply current limit by 100.

read_m1_max_current(*address=None*)

Read Motor 1 Maximum Current Limit. Current value is in 10ma units. To calculate divide value by 100. MinCurrent is always 0.

Returns [MaxCurrent(4 bytes), MinCurrent(4 bytes)]

read_m2_max_current(*address=None*)

Read Motor 2 Maximum Current Limit. Current value is in 10ma units. To calculate divide value by 100. MinCurrent is always 0.

Returns [MaxCurrent(4 bytes), MinCurrent(4 bytes)]

set_pwm_mode(*mode*, *address=None*)
Set PWM Drive mode. Locked Antiphase(0) or Sign Magnitude(1).

read_pwm_mode(*address=None*)
Read PWM Drive mode. See `set_pwm_mode()`.

Returns [PWMMode]

read_eeprom(*ee_address*, *address=None*)
Read a value from the User EEProm memory(256 bytes).

Returns [Value(2 bytes)]

write_eeprom(*ee_address*, *ee_word*, *address=None*)
Get Priority Levels.

Roboclaw serial commands

Serial Command Enums

class roboclaw.serial_commands.**Cmd**
the domain of key/value pairs used for serial commands to the roboclaw. Each command represents a specfic function of the `Roboclaw` driver.

M1FORWARD = 0
The `forward_m1` command byte

M1BACKWARD = 1
The `backward_m1` command byte

SETMINMB = 2
The `set_min_voltage_main_battery` command byte

SETMAXMB = 3
The `set_max_voltage_main_battery` command byte

M2FORWARD = 4
The `forward_m2` command byte

M2BACKWARD = 5
The `backward_m2` command byte

M17BIT = 6
The `forward_backward_m1` command byte

M27BIT = 7
The `forward_backward_m2` command byte

MIXEDFORWARD = 8
The `forward_mixed` command byte

MIXEDBACKWARD = 9
The `backward_mixed` command byte

MIXEDRIGHT = 10
The `turn_right_mixed` command byte

MIXEDLEFT = 11
The `turn_left_mixed` command byte

MIXEDFB = 12
The *forward_backward_mixed* command byte

MIXEDLR = 13
The *left_right_mixed* command byte

GETM1ENC = 16
The *read_encoder_m1* command byte

GETM2ENC = 17
The *read_encoder_m2* command byte

GETM1SPEED = 18
The *read_speed_m1* command byte

GETM2SPEED = 19
The *read_speed_m2* command byte

RESETENC = 20
The *reset_encoders* command byte

GETVERSION = 21
The *read_version* command byte

SETM1ENCCOUNT = 22
The *set_enc_m1* command byte

SETM2ENCCOUNT = 23
The *set_enc_m2* command byte

GETMBATT = 24
The *read_main_battery_voltage* command byte

GETLBATT = 25
The *read_logic_battery_voltage* command byte

SETMINLB = 26
The *set_min_voltage_logic_battery* command byte

SETMAXLB = 27
The *set_max_voltage_logic_battery* command byte

SETM1PID = 28
The *set_m1_velocity_pid* command byte

SETM2PID = 29
The *set_m2_velocity_pid* command byte

GETM1ISPEED = 30
The *read_raw_speed_m1* command byte

GETM2ISPEED = 31
The *read_raw_speed_m2* command byte

M1DUTY = 32
The *duty_m1* command byte

M2DUTY = 33
The *duty_m2* command byte

MIXEDDUTY = 34
The *duty_m1_m2* command byte

M1SPEED = 35
The *speed_m1* command byte

M2SPEED = 36
The *speed_m2* command byte

MIXEDSPEED = 37
The *speed_m1_m2* command byte

M1SPEEDACCEL = 38
The *speed_accel_m1* command byte

M2SPEEDACCEL = 39
The *speed_accel_m2* command byte

MIXEDSPEEDACCEL = 40
The *speed_accel_m1_m2* command byte

M1SPEEDDIST = 41
The *speed_distance_m1* command byte

M2SPEEDDIST = 42
The *speed_distance_m2* command byte

MIXEDSPEEDDIST = 43
The *speed_distance_m1_m2* command byte

M1SPEEDACCELDIST = 44
The *speed_accel_distance_m1* command byte

M2SPEEDACCELDIST = 45
The *speed_accel_distance_m2* command byte

MIXEDSPEEDACCELDIST = 46
The *speed_accel_distance_m1_m2* command byte

GETBUFFERS = 47
The *read_buffer_length* command byte

GETPWMS = 48
The *read_pwms* command byte

GETCURRENTS = 49
The *read_currents* command byte

MIXEDSPEED2ACCEL = 50
The *speed_accel_m1_m2_2* command byte

MIXEDSPEED2ACCELDIST = 51
The *speed_accel_distance_m1_m2_2* command byte

M1DUTYACCEL = 52
The *duty_accel_m1* command byte

M2DUTYACCEL = 53
The *duty_accel_m2* command byte

MIXEDDUTYACCEL = 54
The *duty_accel_m1_m2* command byte

READM1PID = 55
The *read_m1_velocity_pid* command byte

READM2PID = 56
The *read_m2_velocity_pid* command byte

SETMAINVOLTAGES = 57
The *set_main_voltages* command byte

SETLOGICVOLTAGES = 58
The *set_logic_voltages* command byte

GETMINMAXMAINVOLTAGES = 59
The *read_min_max_main_voltages* command byte

GETMINMAXLOGICVOLTAGES = 60
The *read_min_max_logic_voltages* command byte

SETM1POSPID = 61
The *set_m1_position_pid* command byte

SETM2POSPID = 62
The *set_m2_position_pid* command byte

READM1POSPID = 63
The *read_m1_position_pid* command byte

READM2POSPID = 64
The *read_m2_position_pid* command byte

M1SPEEDACCELDECCELPOS = 65
The *speed_accel_deccel_position_m1* command byte

M2SPEEDACCELDECCELPOS = 66
The *speed_accel_deccel_position_m2* command byte

MIXEDSPEEDACCELDECCELPOS = 67
The *speed_accel_deccel_position_m1_m2* command byte

SETM1DEFAULTACCEL = 68
The *set_m1_default_accel* command byte

SETM2DEFAULTACCEL = 69
The *set_m2_default_accel* command byte

SETPINFUNCTIONS = 74
The *set_pin_functions* command byte

GETPINFUNCTIONS = 75
The *read_pin_functions* command byte

SETDEADBAND = 76
The *set_deadband* command byte

GETDEADBAND = 77
The *get_deadband* command byte

RESTOREDEFAULTS = 80
The *restore_defaults* command byte

GETTEMP = 82
The *read_temp* command byte

GETTEMP2 = 83
The *read_temp2* command byte

GETERROR = 90
The `read_error` command byte

GETENCODERMODE = 91
The `read_encoder_modes` command byte

SETM1ENCODERMODE = 92
The `set_m1_encoder_mode` command byte

SETM2ENCODERMODE = 93
The `set_m2_encoder_mode` command byte

WRITENVM = 94
The `write_nvm` command byte

READNVM = 95
The `read_nvm` command byte

SETCONFIG = 98
The `set_config` command byte

GETCONFIG = 99
The `get_config` command byte

SETM1MAXCURRENT = 133
The `set_m1_max_current` command byte

SETM2MAXCURRENT = 134
The `set_m2_max_current` command byte

GETM1MAXCURRENT = 135
The `read_m1_max_current` command byte

GETM2MAXCURRENT = 136
The `read_m2_max_current` command byte

SETPWMMODE = 148
The `set_pwm_mode` command byte

GETPWMMODE = 149
The `read_pwm_mode` command byte

READEEPROM = 252
The `read_eeprom` command byte

WRITEEEPROM = 253
The `write_eeprom` command byte

FLAGBOOTLOADER = 255
The command byte

2.1.3 Helpers

CRC16 data manipulation

A module for manipulating dat including generating CRC values and datatype constraints. For more information on how CRC algorithms work: https://www.zlib.net/crc_v3.txt

`roboclaw.data_manip.`**`make_poly`**(*bit_length*, *msb=False*)
Make `int` "degree polynomial" in which each bit represents a degree who's coefficient is 1

Parameters

- **bit_length** (*int*) – The amount of bits to play with
- **msb** (*bool*) – `True` make only the MSBit 1 and the rest a 0. `False` makes all bits 1.

`roboclaw.data_manip.`**`crc16`**(*data*, *deg_poly=4129*, *init_value=0*)
Calculates a checksum of 16-bit length

`roboclaw.data_manip.`**`crc32`**(*data*, *deg_poly=23302*, *init_value=5592405*)
Calculates a checksum of 32-bit length. Default `deg_poly` and `init_value` values are BLE compliant.

`roboclaw.data_manip.`**`crc_bits`**(*data*, *bit_length*, *deg_poly*, *init_value*)
Calculates a checksum of various sized buffers

Parameters

- **data** (*bytearray*) – This `bytearray` of data to be uncorrupted.
- **bit_length** (*int*) – The length of bits that will represent the checksum.
- **deg_poly** (*int*) – A preset "degree polynomial" in which each bit represents a degree who's coefficient is 1.
- **init_value** (*int*) – This will be the value that the checksum will use while shifting in the buffer data.

`roboclaw.data_manip.`**`validate16`**(*data*, *deg_poly=4129*, *init_value=0*)
Validates a received data by comparing the calculated 16-bit checksum with the checksum included at the end of the data

`roboclaw.data_manip.`**`validate`**(*data*, *bit_length*, *deg_poly*, *init_value*)
Validates a received checksum of various sized buffers

Parameters

- **data** (*bytearray*) – This `bytearray` of data to be uncorrupted.
- **bit_length** (*int*) – The length of bits that will represent the checksum.
- **deg_poly** (*int*) – A preset "degree polynomial" (in which each bit represents a degree who's coefficient is 1) as a quotient.
- **init_value** (*int*) – This will be the value that the checksum will use while shifting in the buffer data.

Returns `True` if data was uncorrupted. `False` if something went wrong. (either checksum didn't match or payload is altered).

UART Serial with context manager For MicroPython

This module contains a wrapper class for MicroPython's `UART` or CircuitPython's `UART` class to work as a drop-in replacement to `Serial` object.

Note: This helper class does not expose all the pySerial API. It's tailored to this library only. That said, to use this:

```
from roboclaw.usart_serial_ctx import SerialUART as UART
serial_bus = UART()
with serial_bus:
    serial_bus.read_until() # readline() with timeout
    serial_bus.in_waiting() # how many bytes in the RX buffer
    serial_bus.close() # same as UART.deinit()
# exit ``with`` also calls machine.UART.deinit()
```

2.2 Indices and tables

- genindex
- modindex
- search

Python Module Index

r

Index

L

M

P

R

S

T

V

W