

Basicmicro API Reference

This document provides detailed information about the Basicmicro library API.

Table of Contents

- Main Controller Class
- Connection Management
- Duty Cycle Control
- Speed Control
- Position Control
- PID Configuration
- Encoder Functions
- Status and Diagnostic Functions
- Configuration Functions
- CAN Bus Functions
- Advanced Functions
- Legacy Motor Control

Controller Compatibility

Functions in this API work with either: - **Roboclaw** - Original Basicmicro motor controller - **MCP** - Motor Control Protocol controllers - (**Roboclaw**, **MCP**) - Functions that work with both controller types

Main Controller Class

Basicmicro

The main interface class for controlling Basicmicro motor controllers.

```
controller = Basicmicro(comport, rate, timeout=0.01, retries=2, verbose=False)
```

Parameters: - **comport** (str): The COM port to use (e.g., 'COM3', '/dev/ttyACM0') - **rate** (int): The baud rate for the serial communication - **timeout** (float, optional): The timeout for serial communication in seconds. Default is 0.01. - **retries** (int, optional): The number of retries for communication. Default is 2. - **verbose** (bool, optional): Enable detailed debug logging. Default is False.

Connection Management

Open() (Roboclaw, MCP)

Opens and configures the serial connection to the controller. This method attempts to establish communication with the controller and verify it by reading the firmware version.

```
success = controller.Open()
```

Returns: - bool: True if connection successful, False otherwise

Raises: - `serial.SerialException`: If there are issues with the serial port -
`ValueError`: If port parameters are invalid

close() (Roboclaw, MCP)

Closes the serial connection to the controller. This should be called when finished using the controller to free up system resources.

```
controller.close()
```

Returns: - None

Context Manager Support (Roboclaw, MCP)

The `Basicmicro` class supports the Python context manager pattern:

```
with Basicmicro("/dev/ttyACM0", 38400) as controller:
    # Work with controller
    # Connection is automatically closed when leaving the block
```

Duty Cycle Control

Modern control using 16-bit duty cycle values (-32767 to +32767):

DutyM1(address, val) (Roboclaw, MCP)

Sets the duty cycle for motor 1. This directly controls the PWM output to the motor.

```
success = controller.DutyM1(address, 16384) # 50% forward
```

Parameters: - `address` (int): The address of the controller (0x80-0x87) - `val` (int): The duty cycle value (-32767 to +32767) - Positive values: Forward direction - Negative values: Reverse direction - Magnitude: Power level (32767 = 100%)

Returns: - bool: True if successful, False otherwise

DutyM2(address, val) (Roboclaw, MCP)

Sets the duty cycle for motor 2. This directly controls the PWM output to the motor.

```
success = controller.DutyM2(address, -8192) # 25% backward
```

Parameters: - `address` (int): The address of the controller (0x80-0x87) - `val` (int): The duty cycle value (-32767 to +32767) - Positive values: Forward direction - Negative values: Reverse direction - Magnitude: Power level (32767 = 100%)

Returns: - bool: True if successful, False otherwise

DutyM1M2(address, m1, m2) (Roboclaw, MCP)

Sets the duty cycle for both motors simultaneously. This allows coordinated movement and ensures both motors start at the same time.

```
success = controller.DutyM1M2(address, 16384, -8192) # M1 forward, M2 backward
```

Parameters: - address (int): The address of the controller (0x80-0x87) - m1 (int): The duty cycle value for motor 1 (-32767 to +32767) - m2 (int): The duty cycle value for motor 2 (-32767 to +32767)

Returns: - bool: True if successful, False otherwise

DutyAccelM1(address, accel, duty) (Roboclaw, MCP)

Sets acceleration and duty cycle for motor 1.

```
success = controller.DutyAccelM1(address, 500, 16384) # Accelerate to 50%
```

Parameters: - address (int): The address of the controller (0x80-0x87) - accel (int): The acceleration value (change in duty cycle per second) - duty (int): The target duty cycle value (-32767 to +32767)

Returns: - bool: True if successful, False otherwise

DutyAccelM2(address, accel, duty) (Roboclaw, MCP)

Sets acceleration and duty cycle for motor 2.

Parameters: - address (int): The address of the controller (0x80-0x87) - accel (int): The acceleration value (change in duty cycle per second) - duty (int): The target duty cycle value (-32767 to +32767)

Returns: - bool: True if successful, False otherwise

DutyAccelM1M2(address, accel1, duty1, accel2, duty2) (Roboclaw, MCP)

Sets acceleration and duty cycle for both motors.

Parameters: - address (int): The address of the controller (0x80-0x87) - accel1 (int): The acceleration value for motor 1 - duty1 (int): The target duty cycle value for motor 1 (-32767 to +32767) - accel2 (int): The acceleration value for motor 2 - duty2 (int): The target duty cycle value for motor 2 (-32767 to +32767)

Returns: - bool: True if successful, False otherwise

Speed Control

Commands for velocity control (requires encoders):

SpeedM1(address, val) (Roboclaw, MCP)

Sets the speed for motor 1 in encoder counts per second.

```
success = controller.SpeedM1(address, 1000) # 1000 counts/sec forward
```

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **val** (int): The speed value in encoder counts per second - Positive values: Forward direction - Negative values: Reverse direction

Returns: - **bool**: True if successful, False otherwise

SpeedM2(address, val) (Roboclaw, MCP)

Sets the speed for motor 2 in encoder counts per second.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **val** (int): The speed value in encoder counts per second - Positive values: Forward direction - Negative values: Reverse direction

Returns: - **bool**: True if successful, False otherwise

SpeedM1M2(address, m1, m2) (Roboclaw, MCP)

Sets the speed for both motors.

```
success = controller.SpeedM1M2(address, 1000, -800) # Different speeds
```

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **m1** (int): The speed value for motor 1 in encoder counts per second - **m2** (int): The speed value for motor 2 in encoder counts per second

Returns: - **bool**: True if successful, False otherwise

With Acceleration Control (Roboclaw, MCP)

SpeedAccelM1(address, accel, speed)

Commands motor 1 to move at the specified speed using acceleration control.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **accel** (int): The acceleration value in encoder counts per second per second - **speed** (int): The target speed in encoder counts per second (positive or negative)

Returns: - **bool**: True if successful, False otherwise

SpeedAccelM2(address, accel, speed)

Commands motor 2 to move at the specified speed using acceleration control.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **accel** (int): The acceleration value in encoder counts per second per second - **speed** (int): The target speed in encoder counts per second (positive or negative)

Returns: - **bool**: True if successful, False otherwise

SpeedAccelM1M2(address, accel, speed1, speed2)

Commands both motors to move at the specified speeds using the same acceleration.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **accel** (int): The acceleration value for both motors in encoder counts per second per second - **speed1** (int): The target speed for motor 1 in encoder counts per second - **speed2** (int): The target speed for motor 2 in encoder counts per second

Returns: - **bool**: True if successful, False otherwise

SpeedAccelM1M2_2(address, accel1, speed1, accel2, speed2)

Commands both motors to move at the specified speeds using different acceleration values for each motor.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **accel1** (int): The acceleration value for motor 1 in encoder counts per second per second - **speed1** (int): The target speed for motor 1 in encoder counts per second - **accel2** (int): The acceleration value for motor 2 in encoder counts per second per second - **speed2** (int): The target speed for motor 2 in encoder counts per second

Returns: - **bool**: True if successful, False otherwise

Example:

```
# Accelerate Motor 1 to 1000 counts/sec at rate of 500 counts/sec2  
success = controller.SpeedAccelM1(address, 500, 1000)
```

With Distance Control (Roboclaw, MCP)

SpeedDistanceM1(address, speed, distance, buffer)

Commands motor 1 to move a specific distance at a specified speed.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **speed** (int): The target speed in encoder counts per second - **distance** (int): The distance to travel in encoder counts - **buffer** (int): Buffer option - 0: Add to buffer, executes after previous command completes - 1: Immediate execution, cancels any running command

Returns: - `bool`: True if successful, False otherwise

SpeedDistanceM2(address, speed, distance, buffer)

Commands motor 2 to move a specific distance at a specified speed.

Parameters: - `address` (int): The address of the controller (0x80-0x87) - `speed` (int): The target speed in encoder counts per second - `distance` (int): The distance to travel in encoder counts - `buffer` (int): Buffer option - 0: Add to buffer, executes after previous command completes - 1: Immediate execution, cancels any running command

Returns: - `bool`: True if successful, False otherwise

SpeedDistanceM1M2(address, speed1, distance1, speed2, distance2, buffer)

Commands both motors to move specific distances at specified speeds.

Parameters: - `address` (int): The address of the controller (0x80-0x87) - `speed1` (int): The target speed for motor 1 in encoder counts per second - `distance1` (int): The distance for motor 1 to travel in encoder counts - `speed2` (int): The target speed for motor 2 in encoder counts per second - `distance2` (int): The distance for motor 2 to travel in encoder counts - `buffer` (int): Buffer option - 0: Add to buffer, executes after previous command completes - 1: Immediate execution, cancels any running command

Returns: - `bool`: True if successful, False otherwise

Example:

```
# Move Motor 1 for 1000 counts at 500 counts/sec, immediate execution
success = controller.SpeedDistanceM1(address, 500, 1000, 0)
```

With Acceleration and Distance (Roboclaw, MCP)

SpeedAccelDistanceM1(address, accel, speed, distance, buffer)

Commands motor 1 to move a specific distance at a specified speed using acceleration control.

Parameters: - `address` (int): The address of the controller (0x80-0x87) - `accel` (int): The acceleration value in encoder counts per second per second - `speed` (int): The target speed in encoder counts per second - `distance` (int): The distance to travel in encoder counts - `buffer` (int): Buffer option - 0: Add to buffer, executes after previous command completes - 1: Immediate execution, cancels any running command

Returns: - `bool`: True if successful, False otherwise

SpeedAccelDistanceM2(address, accel, speed, distance, buffer)

Commands motor 2 to move a specific distance at a specified speed using acceleration control.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **accel** (int): The acceleration value in encoder counts per second per second - **speed** (int): The target speed in encoder counts per second - **distance** (int): The distance to travel in encoder counts - **buffer** (int): Buffer option - 0: Add to buffer, executes after previous command completes - 1: Immediate execution, cancels any running command

Returns: - **bool**: True if successful, False otherwise

SpeedAccelDistanceM1M2(address, accel, speed1, distance1, speed2, distance2, buffer)

Commands both motors to move specific distances at specified speeds with the same acceleration.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **accel** (int): The acceleration value for both motors in encoder counts per second per second - **speed1** (int): The target speed for motor 1 in encoder counts per second - **distance1** (int): The distance for motor 1 to travel in encoder counts - **speed2** (int): The target speed for motor 2 in encoder counts per second - **distance2** (int): The distance for motor 2 to travel in encoder counts - **buffer** (int): Buffer option - 0: Add to buffer, executes after previous command completes - 1: Immediate execution, cancels any running command

Returns: - **bool**: True if successful, False otherwise

SpeedAccelDistanceM1M2_2(address, accel1, speed1, distance1, accel2, speed2, distance2, buffer)

Commands both motors to move specific distances at specified speeds using different acceleration values for each motor. This provides a complete motion profile with independent control of both motors.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **accel1** (int): The acceleration value for motor 1 in encoder counts per second per second - **speed1** (int): The target speed for motor 1 in encoder counts per second - **distance1** (int): The distance for motor 1 to travel in encoder counts - **accel2** (int): The acceleration value for motor 2 in encoder counts per second per second - **speed2** (int): The target speed for motor 2 in encoder counts per second - **distance2** (int): The distance for motor 2 to travel in encoder counts - **buffer** (int): Buffer option - 0: Add to buffer, executes after previous command completes - 1: Immediate execution, cancels any running command

Returns: - **bool**: True if successful, False otherwise

Position Control

Commands for position control (requires encoders):

PositionM1(address, position, buffer)

Commands motor 1 to move to absolute position.

Parameters: - **address** (int): The address of the controller (0x80-0x87) -
- **position** (int): Target absolute position in encoder counts - **buffer** (int):
Buffer option - 0: Add to buffer, executes after previous command completes -
1: Immediate execution, cancels any running command

Returns: - **bool**: True if successful, False otherwise

PositionM2(address, position, buffer)

Commands motor 2 to move to absolute position.

Parameters: - **address** (int): The address of the controller (0x80-0x87) -
- **position** (int): Target absolute position in encoder counts - **buffer** (int):
Buffer option - 0: Add to buffer, executes after previous command completes -
1: Immediate execution, cancels any running command

Returns: - **bool**: True if successful, False otherwise

PositionM1M2(address, position1, position2, buffer)

Commands both motors to specific absolute positions.

Parameters: - **address** (int): The address of the controller (0x80-0x87) -
- **position1** (int): Target absolute position for motor 1 in encoder counts -
- **position2** (int): Target absolute position for motor 2 in encoder counts -
- **buffer** (int): Buffer option - 0: Add to buffer, executes after previous command completes - 1: Immediate execution, cancels any running command

Returns: - **bool**: True if successful, False otherwise

Example:

```
# Move Motor 1 to position 5000, immediate execution
```

```
success = controller.M1Position(address, 5000, 0)
```

```
# Move both motors to positions, immediate execution
```

```
success = controller.MixedPosition(address, 5000, 3000, 0)
```

Speed-Controlled Position Commands (Roboclaw, MCP)

These commands allow position control with specified speeds.

SpeedPositionM1(address, speed, position, buffer)

Commands motor 1 to move to absolute position with specified maximum speed.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **speed** (int): Maximum speed in encoder counts per second (positive value only) - **position** (int): Target absolute position in encoder counts - **buffer** (int): Buffer option - 0: Add to buffer, executes after previous command completes - 1: Immediate execution, cancels any running command

Returns: - **bool**: True if successful, False otherwise

SpeedPositionM2(address, speed, position, buffer)

Commands motor 2 to move to absolute position with specified maximum speed.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **speed** (int): Maximum speed in encoder counts per second (positive value only) - **position** (int): Target absolute position in encoder counts - **buffer** (int): Buffer option - 0: Add to buffer, executes after previous command completes - 1: Immediate execution, cancels any running command

Returns: - **bool**: True if successful, False otherwise

SpeedPositionM1M2(address, speed1, position1, speed2, position2, buffer)

Commands both motors to move to absolute positions with specified maximum speeds.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **speed1** (int): Maximum speed for motor 1 in encoder counts per second (positive value only) - **position1** (int): Target absolute position for motor 1 in encoder counts - **speed2** (int): Maximum speed for motor 2 in encoder counts per second (positive value only) - **position2** (int): Target absolute position for motor 2 in encoder counts - **buffer** (int): Buffer option - 0: Add to buffer, executes after previous command completes - 1: Immediate execution, cancels any running command

Returns: - **bool**: True if successful, False otherwise

Example:

```
# Move M1 to position 5000 at speed 1000, immediate execution
success = controller.M1SpeedPosition(address, 1000, 5000, 0)
```

Percent Position Commands (Roboclaw, MCP)

These commands allow position control using percentage values, which is useful for applications like servos or limited-range movements.

PercentPositionM1(address, position, buffer)

Commands motor 1 to move to a position specified as a percentage of its range.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **position** (int): Target position as percentage (-32767 to +32767) - -32767: Minimum position (specified in SetM1PositionPID) - 0: Mid-position - +32767: Maximum position (specified in SetM1PositionPID) - **buffer** (int): Buffer option - 0: Add to buffer, executes after previous command completes - 1: Immediate execution, cancels any running command

Returns: - **bool**: True if successful, False otherwise

PercentPositionM2(address, position, buffer)

Commands motor 2 to move to a position specified as a percentage of its range.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **position** (int): Target position as percentage (-32767 to +32767) - -32767: Minimum position (specified in SetM2PositionPID) - 0: Mid-position - +32767: Maximum position (specified in SetM2PositionPID) - **buffer** (int): Buffer option - 0: Add to buffer, executes after previous command completes - 1: Immediate execution, cancels any running command

Returns: - **bool**: True if successful, False otherwise

PercentPositionM1M2(address, position1, position2, buffer)

Commands both motors to move to positions specified as percentages of their ranges.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **position1** (int): Target position for motor 1 as percentage (-32767 to +32767) - **position2** (int): Target position for motor 2 as percentage (-32767 to +32767) - **buffer** (int): Buffer option - 0: Add to buffer, executes after previous command completes - 1: Immediate execution, cancels any running command

Returns: - **bool**: True if successful, False otherwise

Example:

```
# Move motor 1 to 50% of its range (mid-position)
success = controller.PercentPositionM1(address, 0, 0)

# Move motor 1 to 75% of its range (3/4 of the way to maximum)
success = controller.PercentPositionM1(address, 16384, 0)
```

Advanced Position Control (Roboclaw, MCP)

These commands provide complete motion profiles with acceleration, constant velocity, and deceleration phases for precise position control.

SpeedAccelDeccelPositionM1(address, accel, speed, deccel, position, buffer)

Commands motor 1 to move to absolute position with specified acceleration, speed, and deceleration.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **accel** (int): Acceleration value in encoder counts per second per second - **speed** (int): Maximum speed in encoder counts per second (positive value only) - **deccel** (int): Deceleration value in encoder counts per second per second - **position** (int): Target absolute position in encoder counts - **buffer** (int): Buffer option - 0: Add to buffer, executes after previous command completes - 1: Immediate execution, cancels any running command

Returns: - **bool**: True if successful, False otherwise

SpeedAccelDeccelPositionM2(address, accel, speed, deccel, position, buffer)

Commands motor 2 to move to absolute position with specified acceleration, speed, and deceleration.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **accel** (int): Acceleration value in encoder counts per second per second - **speed** (int): Maximum speed in encoder counts per second (positive value only) - **deccel** (int): Deceleration value in encoder counts per second per second - **position** (int): Target absolute position in encoder counts - **buffer** (int): Buffer option - 0: Add to buffer, executes after previous command completes - 1: Immediate execution, cancels any running command

Returns: - **bool**: True if successful, False otherwise

SpeedAccelDeccelPositionM1M2(address, accel1, speed1, deccel1, position1, accel2, speed2, deccel2, position2, buffer)

Commands both motors to move to absolute positions with separate motion profiles.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **accel1** (int): Acceleration value for motor 1 in encoder counts per second per second - **speed1** (int): Maximum speed for motor 1 in encoder counts per second (positive value only) - **deccel1** (int): Deceleration value for motor 1 in encoder counts per second per second - **position1** (int): Target absolute position for motor 1 in encoder counts - **accel2** (int): Acceleration value for motor 2 in encoder counts per second per second - **speed2** (int): Maximum speed for motor 2 in encoder counts per second (positive value only) - **deccel2** (int): Deceleration value for motor 2 in encoder counts per second per second - **position2** (int): Target absolute position for motor 2 in encoder counts - **buffer** (int):

Buffer option - 0: Add to buffer, executes after previous command completes -
1: Immediate execution, cancels any running command

Returns: - bool: True if successful, False otherwise

Example:

```
# Move both motors to different positions with different motion profiles  
# Motor 1: position 10000, accel 500, speed 2000, decel 1000  
# Motor 2: position -5000, accel 300, speed 1500, decel 800  
success = controller.SpeedAccelDecelPositionM1M2(address, 500, 2000, 1000, 10000, 300, 1500, 800)
```

PID Configuration

Velocity PID (Roboclaw, MCP)

SetM1VelocityPID(address, p, i, d, qpps) Sets the velocity PID constants for motor 1. These control how the motor responds to speed commands.

Parameters: - address (int): The address of the controller (0x80-0x87) - p (float): Proportional constant (0.0-1024.0) - affects response time - i (float): Integral constant (0.0-1024.0) - affects steady-state error - d (float): Derivative constant (0.0-1024.0) - affects stability and overshoot - qpps (int): Maximum speed in quadrature pulses per second

Returns: - bool: True if successful, False otherwise

SetM2VelocityPID(address, p, i, d, qpps) Sets the velocity PID constants for motor 2. These control how the motor responds to speed commands.

Parameters: - address (int): The address of the controller (0x80-0x87) - p (float): Proportional constant (0.0-1024.0) - affects response time - i (float): Integral constant (0.0-1024.0) - affects steady-state error - d (float): Derivative constant (0.0-1024.0) - affects stability and overshoot - qpps (int): Maximum speed in quadrature pulses per second

Returns: - bool: True if successful, False otherwise

ReadM1VelocityPID(address) Reads the velocity PID constants for motor 1.

Parameters: - address (int): The address of the controller (0x80-0x87)

Returns: - tuple: (success, p, i, d, qpps) - success (bool): True if read successful - p (float): Proportional constant - i (float): Integral constant - d (float): Derivative constant - qpps (int): Maximum speed in quadrature pulses per second

ReadM2VelocityPID(address) Reads the velocity PID constants for motor 2.

Parameters: - address (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, p, i, d, qpps) - **success** (bool): True if read successful - **p** (float): Proportional constant - **i** (float): Integral constant - **d** (float): Derivative constant - **qpps** (int): Maximum speed in quadrature pulses per second

Example:

```
# Set velocity PID for motor 1
success = controller.SetM1VelocityPID(address, 1.0, 0.5, 0.25, 44000)

# Read back the settings
success, p, i, d, qpps = controller.ReadM1VelocityPID(address)
```

Position PID (Roboclaw, MCP)

SetM1PositionPID(address, kp, ki, kd, kimax, deadzone, min_pos, max_pos) Sets the position PID constants for motor 1.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **kp** (float): Proportional constant (0.0-1024.0) - affects response time - **ki** (float): Integral constant (0.0-1024.0) - affects steady-state error - **kd** (float): Derivative constant (0.0-1024.0) - affects stability and overshoot - **kimax** (int): Maximum integral windup limit - **deadzone** (int): Encoder count deadzone (error less than this is treated as zero) - **min_pos** (int): Minimum position limit in encoder counts - **max_pos** (int): Maximum position limit in encoder counts

Returns: - **bool:** True if successful, False otherwise

SetM2PositionPID(address, kp, ki, kd, kimax, deadzone, min_pos, max_pos) Sets the position PID constants for motor 2.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **kp** (float): Proportional constant (0.0-1024.0) - affects response time - **ki** (float): Integral constant (0.0-1024.0) - affects steady-state error - **kd** (float): Derivative constant (0.0-1024.0) - affects stability and overshoot - **kimax** (int): Maximum integral windup limit - **deadzone** (int): Encoder count deadzone (error less than this is treated as zero) - **min_pos** (int): Minimum position limit in encoder counts - **max_pos** (int): Maximum position limit in encoder counts

Returns: - **bool:** True if successful, False otherwise

ReadM1PositionPID(address) Reads the position PID constants for motor 1.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, kp, ki, kd, kimax, deadzone, min, max) - **success** (bool): True if read successful - **kp** (float): Proportional constant - **ki** (float): Integral constant - **kd** (float): Derivative constant - **kimax** (int): Maximum integral windup limit - **deadzone** (int): Encoder count deadzone - **min** (int):

Minimum position limit in encoder counts - **max** (int): Maximum position limit in encoder counts

ReadM2PositionPID(address) Reads the position PID constants for motor 2.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, kp, ki, kd, kimax, deadzone, min, max) - **success** (bool): True if read successful - **kp** (float): Proportional constant - **ki** (float): Integral constant - **kd** (float): Derivative constant - **kimax** (int): Maximum integral windup limit - **deadzone** (int): Encoder count deadzone - **min** (int): Minimum position limit in encoder counts - **max** (int): Maximum position limit in encoder counts

Example:

```
# Set position PID for motor 1
```

```
success = controller.SetM1PositionPID(address, 10.0, 0.5, 1.0, 50, 10, -1000000, 1000000)
```

```
# Read position PID for motor 1
```

```
success, kp, ki, kd, kimax, deadzone, min_pos, max_pos = controller.ReadM1PositionPID(address)
```

Motor Parameters (MCP only)

SetM1LR(address, L, R)

Sets the inductance and resistance values for motor 1. These parameters are used for advanced current control.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **L** (float): Inductance value in Henries (H) - **R** (float): Resistance value in Ohms (Ω)

Returns: - **bool:** True if successful, False otherwise

SetM2LR(address, L, R)

Sets the inductance and resistance values for motor 2. These parameters are used for advanced current control.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **L** (float): Inductance value in Henries (H) - **R** (float): Resistance value in Ohms (Ω)

Returns: - **bool:** True if successful, False otherwise

GetM1LR(address)

Reads the inductance and resistance values for motor 1.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, L, R) - **success** (bool): True if read successful - L (float): Inductance value in Henries (H) - R (float): Resistance value in Ohms (Ω)

GetM2LR(address)

Reads the inductance and resistance values for motor 2.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, L, R) - **success** (bool): True if read successful - L (float): Inductance value in Henries (H) - R (float): Resistance value in Ohms (Ω)

Example:

```
# Set motor 1 parameters (typical values for a small DC motor)
success = controller.SetM1LR(address, 0.0015, 0.5) # 1.5mH inductance, 0.5 $\Omega$  resistance

# Read motor 1 parameters
success, L, R = controller.GetM1LR(address)
print(f"Motor 1: L={L*1000:.2f}mH, R={R:.2f} $\Omega$ ")
```

Encoder Functions

Reading Encoders (Roboclaw, MCP)

ReadEncM1(address) Reads the encoder count for motor 1.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, count, status) - **success** (bool): True if read successful - **count** (int): The encoder count value - **status** (int): The status byte - Bit 0: Counter underflow (1 = underflow occurred) - Bit 1: Direction (0 = forward, 1 = backward) - Bit 2: Counter overflow (1 = overflow occurred)

ReadEncM2(address) Reads the encoder count for motor 2.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, count, status) - **success** (bool): True if read successful - **count** (int): The encoder count value - **status** (int): The status byte - Bit 0: Counter underflow (1 = underflow occurred) - Bit 1: Direction (0 = forward, 1 = backward) - Bit 2: Counter overflow (1 = overflow occurred)

GetEncoders(address) Reads the encoder values for both motors simultaneously.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, enc1, enc2) - **success** (bool): True if read successful - **enc1** (int): The encoder count for motor 1 - **enc2** (int): The encoder count for motor 2

Example:

```
# Read Motor 1 encoder
success, count, status = controller.ReadEncM1(address)

# Read both encoders
success, enc1, enc2 = controller.GetEncoders(address)
```

Encoder Management (Roboclaw, MCP)

ResetEncoders(address) Resets the encoders for both motors to zero.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **bool:** True if successful, False otherwise

SetEncM1(address, cnt) Sets the encoder count for motor 1 to a specific value.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **cnt** (int): The encoder count value to set

Returns: - **bool:** True if successful, False otherwise

SetEncM2(address, cnt) Sets the encoder count for motor 2 to a specific value.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **cnt** (int): The encoder count value to set

Returns: - **bool:** True if successful, False otherwise

Example:

```
# Reset encoders to zero
success = controller.ResetEncoders(address)

# Set encoder 1 value to 1000
success = controller.SetEncM1(address, 1000)
```

Encoder Configuration (Roboclaw, MCP)

ReadEncoderModes(address) Reads the encoder modes for both motors.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, mode1, mode2) - **success** (bool): True if read successful - **mode1** (int): Encoder mode for motor 1 - **mode2** (int): Encoder mode for motor 2

****(Roboclaw)****

- bits 0: 0 = Quadrature Encoder, 1 = Absolute Encoder
- bits 5: Reverse Motor
- bits 6: Reverse Encoder
- bits 7: Enable Encoder in RC Mode

****(MCP)****

- bits(0:7): DIN pin mapping
- bits(8): Reverse Motor

SetM1EncoderMode(address, mode) Sets the encoder mode for motor 1.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **mode** (int): Encoder mode to set

****(Roboclaw)****

- bits 0: 0 = Quadrature Encoder, 1 = Absolute Encoder
- bits 5: Reverse Motor
- bits 6: Reverse Encoder
- bits 7: Enable Encoder in RC Mode

****(MCP)****

- bits(0:7): DIN pin mapping
- bits(8): Reverse Motor

Returns: - **bool:** True if successful, False otherwise

SetM2EncoderMode(address, mode) Sets the encoder mode for motor 2.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **mode** (int): Encoder mode to set

****(Roboclaw)****

- bits 0: 0 = Quadrature Encoder, 1 = Absolute Encoder
- bits 5: Reverse Motor
- bits 6: Reverse Encoder
- bits 7: Enable Encoder in RC Mode

****(MCP)****

- bits(0:7): DIN pin mapping
- bits(8): Reverse Motor

Returns: - **bool:** True if successful, False otherwise

Example:

```

# Set motor 1 to use single-ended encoder
success = controller.SetM1EncoderMode(address, 1)

# Read encoder modes
success, mode1, mode2 = controller.ReadEncoderModes(address)
print(f"Encoder modes: Motor 1 = {mode1}, Motor 2 = {mode2}")

```

Encoder Status (Roboclaw, MCP)

GetEncStatus(address) Gets the encoder error statuses.

Parameters: - address (int): The address of the controller (0x80-0x87)

Returns: - tuple: (success, enc1status, enc2status) - **success** (bool): True if read successful - **enc1status** (int): Status byte for encoder 1 - Bit 0: Counter underflow (1 = underflow occurred) - Bit 1: Direction (0 = forward, 1 = backward) - Bit 2: Counter overflow (1 = overflow occurred) - **enc2status** (int): Status byte for encoder 2 - Bit 0: Counter underflow (1 = underflow occurred) - Bit 1: Direction (0 = forward, 1 = backward) - Bit 2: Counter overflow (1 = overflow occurred)

Example:

```

# Read encoder status
success, status1, status2 = controller.GetEncStatus(address)

# Check if encoder 1 is not connected
if status1 & 0x04: # Check bit 2
    print("Warning: Encoder 1 may not be connected")

```

Status and Diagnostic Functions

Basic Status Information (Roboclaw, MCP)

ReadVersion(address) Reads the firmware version of the controller.

Parameters: - address (int): The address of the controller (0x80-0x87)

Returns: - tuple: (success, version) - **success** (bool): True if read successful - **version** (str): The firmware version string

ReadMainBatteryVoltage(address) Reads the main battery voltage.

Parameters: - address (int): The address of the controller (0x80-0x87)

Returns: - tuple: (success, voltage) - **success** (bool): True if read successful - **voltage** (int): The main battery voltage in tenths of a volt (e.g., 124 = 12.4V)

ReadLogicBatteryVoltage(address) Reads the logic battery voltage.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, voltage) - **success** (bool): True if read successful
- **voltage** (int): The logic battery voltage in tenths of a volt (e.g., 50 = 5.0V)

GetVolts(address) Reads both main and logic battery voltages.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, mbat, lbat) - **success** (bool): True if read successful
- **mbat** (int): The main battery voltage in tenths of a volt - **lbat** (int): The logic battery voltage in tenths of a volt

ReadTemp(address) Reads the temperature from the first sensor.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, temp) - **success** (bool): True if read successful -
temp (int): The temperature in tenths of a degree Celsius (e.g., 255 = 25.5°C)

ReadTemp2(address) Reads the temperature from the second sensor(on supported units).

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, temp) - **success** (bool): True if read successful -
temp (int): The temperature in tenths of a degree Celsius (e.g., 255 = 25.5°C)

GetTemps(address) Reads both temperature sensors.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, temp1, temp2) - **success** (bool): True if read successful
- **temp1** (int): The temperature from sensor 1 in tenths of a degree Celsius
- **temp2** (int): The temperature from sensor 2 in tenths of a degree Celsius(on supported units)

ReadError(address) Reads the error status.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, error) - **success** (bool): True if read successful
- **error** (int): The error status (bitmask) - Bit 0: E-Stop - Bit 1: Temperature Error - Bit 2: Temperature2 Error - Bit 3: Main Battery High Error - Bit 4: Logic Battery High Error - Bit 5: Logic Battery Low Error

- Bit 8: Speed Error Limit M1
- Bit 9: Speed Error Limit M2
- Bit 10: Position Error Limit M1

- Bit 11: Position Error Limit M2
- Bit 12: Over Current Error M1
- Bit 13: Over Current Error M2
-
- Bit 16: Over Current Warning M1
- Bit 17: Over Current Warning M2
- Bit 18: Main Battery High Warning
- Bit 19: Main Battery Low Warning
- Bit 20: Temperature Warning
- Bit 21: Temperature2 Warning
- Bit 22: Limit Signal Triggered M1
- Bit 23: Limit Signal Triggered M2
-
- Bit 29: Booting Warning
- Bit 30: Over Regen Warning M1
- Bit 31: Over Regen Warning M2

Example:

```
# Read battery voltage
success, voltage = controller.ReadMainBatteryVoltage(address)
voltage_volts = voltage / 10.0 # Convert to volts

# Read temperature
success, temp = controller.ReadTemp(address)
temp_celsius = temp / 10.0 # Convert to degrees Celsius
```

Motor Speed and Current (Roboclaw, MCP)

Functions for reading motor speed and current information.

ReadISpeedM1(address) Reads the instantaneous speed for motor 1. This provides the most recent encoder reading without filtering.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, speed, status) - **success** (bool): True if read successful - **speed** (int): Instantaneous speed in encoder counts per second - **status** (int): Status byte - Bit 0: Counter underflow (1 = underflow occurred) - Bit 1: Direction (0 = forward, 1 = backward) - Bit 2: Counter overflow (1 = overflow occurred)

ReadISpeedM2(address) Reads the instantaneous speed for motor 2. This provides the most recent encoder reading without filtering.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, speed, status) - **success** (bool): True if read successful - **speed** (int): Instantaneous speed in encoder counts per second -

status (int): Status byte - Bit 0: Counter underflow (1 = underflow occurred)
- Bit 1: Direction (0 = forward, 1 = backward) - Bit 2: Counter overflow (1 = overflow occurred)

GetISpeeds(address) Reads the instantaneous speeds for both motors.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, speed1, speed2) - **success** (bool): True if read successful - **speed1** (int): Instantaneous speed for motor 1 in encoder counts per second - **speed2** (int): Instantaneous speed for motor 2 in encoder counts per second

ReadSpeedM1(address) Reads the speed for motor 1. This provides a filtered speed reading.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, speed, status) - **success** (bool): True if read successful - **speed** (int): Speed in encoder counts per second - **status** (int): Status byte - Bit 0: Counter underflow (1 = underflow occurred) - Bit 1: Direction (0 = forward, 1 = backward) - Bit 2: Counter overflow (1 = overflow occurred)

ReadSpeedM2(address) Reads the speed for motor 2. This provides a filtered speed reading.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, speed, status) - **success** (bool): True if read successful - **speed** (int): Speed in encoder counts per second - **status** (int): Status byte - Bit 0: Counter underflow (1 = underflow occurred) - Bit 1: Direction (0 = forward, 1 = backward) - Bit 2: Counter overflow (1 = overflow occurred)

GetSpeeds(address) Reads the speeds for both motors.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, speed1, speed2) - **success** (bool): True if read successful - **speed1** (int): Speed for motor 1 in encoder counts per second - **speed2** (int): Speed for motor 2 in encoder counts per second

ReadCurrents(address) Reads the current values for both motors.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, current1, current2) - **success** (bool): True if read successful - **current1** (int): Current for motor 1 in 10mA units (e.g., 150 = 1.5A) - **current2** (int): Current for motor 2 in 10mA units (e.g., 150 = 1.5A)

Example:

```

# Read instantaneous speed for motor 1
success, speed, status = controller.ReadISpeedM1(address)

# Read currents for both motors
success, current1, current2 = controller.ReadCurrents(address)
print(f"Motor currents: M1 = {current1/100:.2f}A, M2 = {current2/100:.2f}A")

```

Current Limits (Roboclaw, MCP)

Functions for setting and reading current limits for the motors.

SetM1MaxCurrent(address, maxi, mini)

Sets the maximum and minimum current limits for motor 1.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **maxi** (int): Maximum current limit in 10mA units (e.g., 250 = 2.5A) - **mini** (int): Minimum current limit in 10mA units (e.g., 0 = 0A)

Returns: - **bool**: True if successful, False otherwise

SetM2MaxCurrent(address, maxi, mini)

Sets the maximum and minimum current limits for motor 2.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **maxi** (int): Maximum current limit in 10mA units (e.g., 250 = 2.5A) - **mini** (int): Minimum current limit in 10mA units (e.g., 0 = 0A)

Returns: - **bool**: True if successful, False otherwise

ReadM1MaxCurrent(address)

Reads the maximum and minimum current limits for motor 1.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple**: (success, maxi, mini) - **success** (bool): True if read successful - **maxi** (int): Maximum current limit in 10mA units - **mini** (int): Minimum current limit in 10mA units

ReadM2MaxCurrent(address)

Reads the maximum and minimum current limits for motor 2.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple**: (success, maxi, mini) - **success** (bool): True if read successful - **maxi** (int): Maximum current limit in 10mA units - **mini** (int): Minimum current limit in 10mA units

Example:

```

# Set motor 1 current limits: maximum 2.5A, minimum 0A
success = controller.SetM1MaxCurrent(address, 250, 0)

# Read motor 1 current limits
success, maxi, mini = controller.ReadM1MaxCurrent(address)
print(f"Motor 1 current limits: Max = {maxi/100:.1f}A, Min = {mini/100:.1f}A")

```

Detailed Status (Roboclaw, MCP)

Functions for reading detailed status information from the controller.

GetStatus(address)

Reads comprehensive status information from the controller.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** A complex tuple containing multiple status values: - **success** (bool): True if read successful - **tick** (int): Internal tick counter - **state** (int): Controller state - **temp1** (int): Temperature 1 in tenths of a degree Celsius - **temp2** (int): Temperature 2 in tenths of a degree Celsius - **mbat** (int): Main battery voltage in tenths of a volt - **lbat** (int): Logic battery voltage in tenths of a volt - **pwm1** (int): PWM duty cycle for motor 1 (-32767 to +32767) - **pwm2** (int): PWM duty cycle for motor 2 (-32767 to +32767) - **cur1** (int): Current for motor 1 in 10mA units - **cur2** (int): Current for motor 2 in 10mA units - **enc1** (int): Encoder count for motor 1 - **enc2** (int): Encoder count for motor 2 - **speed1** (int): Speed for motor 1 - **speed2** (int): Speed for motor 2 - **ispeed1** (int): Instantaneous speed for motor 1 - **ispeed2** (int): Instantaneous speed for motor 2 - **speederror1** (int): Speed error for motor 1 - **speederror2** (int): Speed error for motor 2 - **poserror1** (int): Position error for motor 1 - **poserror2** (int): Position error for motor 2

ReadPWMS(address)

Reads the PWM duty cycle values for both motors.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, pwm1, pwm2) - **success** (bool): True if read successful - **pwm1** (int): PWM duty cycle for motor 1 (-32767 to +32767) - **pwm2** (int): PWM duty cycle for motor 2 (-32767 to +32767)

ReadBuffers(address)

Reads the command buffer status.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, buffer1, buffer2) - **success** (bool): True if read successful - **buffer1** (int): Command buffer status for motor 1 (0 = empty, 1 =

has command) - **buffer2** (int): Command buffer status for motor 2 (0 = empty, 1 = has command)

GetSpeedErrors(address)

Reads the speed error values for both motors.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, error1, error2) - **success** (bool): True if read successful - **error1** (int): Speed error for motor 1 - **error2** (int): Speed error for motor 2

GetPosErrors(address)

Reads the position error values for both motors.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, error1, error2) - **success** (bool): True if read successful - **error1** (int): Position error for motor 1 - **error2** (int): Position error for motor 2

Example:

```
# Read PWM values
success, pwm1, pwm2 = controller.ReadPWMS(address)
print(f"PWM duty cycles: M1 = {pwm1/327.67:.1f}%, M2 = {pwm2/327.67:.1f}%")

# Read buffer status
success, buffer1, buffer2 = controller.ReadBuffers(address)
if buffer1 == 0 and buffer2 == 0:
    print("Both motor command buffers are empty")
```

Configuration Functions

Controller Configuration (Roboclaw, MCP)

Functions for managing controller settings and configuration.

RestoreDefaults(address) (Roboclaw, MCP) Restores factory default settings.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **bool:** True if successful, False otherwise

Note: This command requires a special value as part of the packet to prevent accidental reset.

WriteNVM(address) (Roboclaw, MCP) Saves current settings to non-volatile memory (NVM). Settings will be loaded on next power-up.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **bool**: True if successful, False otherwise

Note: This command requires a special value as part of the packet to prevent accidental write.

ReadNVM(address) (Roboclaw, MCP) Loads settings from non-volatile memory (NVM) to current active settings.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **bool**: True if successful, False otherwise

SetSerialNumber(address, serial_number) (Roboclaw, MCP) Sets the controller serial number (36 bytes max).

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **serial_number** (str): Serial number string (36 characters max)

Returns: - **bool**: True if successful, False otherwise

Note: Serial number will be padded with nulls if less than 36 bytes.

Raises: - **ValueError**: If **serial_number** is not a string

GetSerialNumber(address) (Roboclaw, MCP) Reads the controller serial number.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple**: (success, serial_number) - **success** (bool): True if read successful - **serial_number** (str): Serial number string

ReadEeprom(address, ee_address) (Roboclaw, MCP) Reads a word from the EEPROM.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **ee_address** (int): The EEPROM address to read from (0-255)

Returns: - **tuple**: (success, value) - **success** (bool): True if read successful - **value** (int): The word value read from EEPROM

WriteEeprom(address, ee_address, ee_word) (Roboclaw, MCP) Writes a word to the EEPROM.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **ee_address** (int): The EEPROM address to write to (0-255) - **ee_word** (int): The word value to write to EEPROM (0-65535)

Returns: - bool: True if successful, False otherwise

Example:

```
# Save current settings to non-volatile memory
success = controller.WriteNVM(address)

# Read serial number
success, serial = controller.GetSerialNumber(address)
print(f"Controller serial number: {serial}")
```

General Configuration

SetConfig(address, config) (Roboclaw only) Sets the controller configuration.

Parameters: - address (int): The address of the controller (0x80-0x87) - config (int): Configuration value - Bit 0:1 Mode (0=RC, 1=Analog, 2=SimpleSerial, 3=PacketSerial) - Bit 2:4 Battery(0=User, 1=Auto, 2=3 Cell, 3=4 Cell, 4=5 Cell, 5=6 Cell, 6=7 Cell, 7=8 Cell) - Bit 13 Swap Encoders

Serial modes - Bit 5:7 Baudrate(0=2400, 1=9600, 2=19200, 3=38400, 4=57600, 5=115200, 6=230400, 7=460800) - Bit 8:10 Packet Address - Bit 12 Slave Select(SimpleSerial Only) - Bit 13 Relay mode - Bit 15 Open Drain(PacketSerial, S2 pin only)

RC/Analog modes - Bit 5 Mixing - Bit 6 Exponential - Bit 7 AutoCalibrate - Bit 8 FlipSwitch - Bit 9 RC Signal Timeout

Returns: - bool: True if successful, False otherwise

Warning: Baudrate and packet address are not changed until a WriteNVM is executed. **Warning:** If control mode is changed from packet serial mode, communications will be lost!

GetConfig(address) (Roboclaw only) Reads the controller configuration.

Parameters: - address (int): The address of the controller (0x80-0x87)

Returns: - tuple: (success, config) - success (bool): True if read successful - config (int): Configuration value (see SetConfig for bit definitions)

SetTimeout(address, timeout) (Roboclaw only) Sets the communications timeout. If no valid commands are received within this time, the motors will be stopped.

Parameters: - address (int): The address of the controller (0x80-0x87) - timeout (float): Timeout value in seconds (0.0-655.35)

Returns: - bool: True if successful, False otherwise

GetTimeout(address) (Roboclaw only) Reads the communications timeout.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, timeout) - **success** (bool): True if read successful
- **timeout** (float): Timeout value in seconds

Example:

```
# Set a 1.5 second communications timeout
success = controller.SetTimeout(address, 1.5)

# Read configuration
success, config = controller.GetConfig(address)
```

Motor Configuration (Roboclaw, MCP)

Functions for configuring motor-specific settings.

SetM1DefaultAccel(address, accel) Sets the default acceleration for motor 1. This will be used when no acceleration is specified in movement commands or if a 0 accel is used

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **accel** (int): Default acceleration value in binary percentage(32768 = 100%, 65536 = 2000%) - **decel** (int): Default deceleration value in binary percentage(32768 = 100%, 65536 = 2000%)

Returns: - **bool:** True if successful, False otherwise

SetM2DefaultAccel(address, accel) Sets the default acceleration for motor 2. This will be used when no acceleration is specified.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **accel** (int): Default acceleration value in binary percentage(32768 = 100%, 65536 = 2000%) - **decel** (int): Default deceleration value in binary percentage(32768 = 100%, 65536 = 2000%)

Returns: - **bool:** True if successful, False otherwise

GetDefaultAccels(address) Reads the default accelerations for both motors.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, accel1, decel1, accel2, decel2) - **success** (bool): True if read successful - **accel1** (int): Default acceleration for motor 1 - **decel1** (int): Default deceleration for motor 1 - **accel2** (int): Default acceleration for motor 2 (for some versions) - **decel2** (int): Default deceleration for motor 2 (for some versions)

SetMainVoltages(address, min_voltage, max_voltage, auto_offset)

Sets the main battery voltage limits.

Parameters: - **address** (int): The address of the controller (0x80-0x87) -
min_voltage (int): Minimum voltage in tenths of a volt (e.g., 95 = 9.5V) -
max_voltage (int): Maximum voltage in tenths of a volt (e.g., 140 = 14.0V)
- **auto_offset** (int): Auto offset option 0 = disabled, 1+ = offset voltage in
tenths of a volt

Returns: - **bool**: True if successful, False otherwise

SetLogicVoltages(address, min_voltage, max_voltage) Sets the logic
battery voltage limits.

Parameters: - **address** (int): The address of the controller (0x80-0x87) -
min_voltage (int): Minimum voltage in tenths of a volt - **max_voltage** (int):
Maximum voltage in tenths of a volt

Returns: - **bool**: True if successful, False otherwise

ReadMinMaxMainVoltages(address) Reads the main battery voltage limits.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple**: (success, min, max, auto_offset) - **success** (bool): True
if read successful - **min** (int): Minimum voltage in tenths of a volt - **max** (int):
Maximum voltage in tenths of a volt - **auto_offset** (int): Auto offset option (0
= disabled, 1+ = offset voltage in tenths of a volt)

ReadMinMaxLogicVoltages(address) Reads the logic battery voltage limits.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple**: (success, min, max) - **success** (bool): True if read successful
- **min** (int): Minimum voltage in tenths of a volt - **max** (int): Maximum voltage
in tenths of a volt

SetOffsets(address, offset1, offset2) Sets voltage offsets.

Parameters: - **address** (int): The address of the controller (0x80-0x87) -
offset1 (int): Offset voltage for Main Battery (0-255) in tenths of a volt -
offset2 (int): Offset voltage for Logic Battery (0-255) in tenths of a volt

Returns: - **bool**: True if successful, False otherwise

GetOffsets(address) Reads voltage offsets.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, offset1, offset2) - **success** (bool): True if read successful - **offset1** (int): Offset voltage for Main Battery in tenths of a volt - **offset2** (int): Offset voltage for Logic Battery in tenths of a volt

Example:

```
# Set default acceleration for motor 1 to 500 counts/sec2
success = controller.SetM1DefaultAccel(address, 500)

# Set main battery limits (min: 10.0V, max: 14.0V)
success = controller.SetMainVoltages(address, 100, 140, 0)
```

PWM Configuration (Roboclaw, MCP)

Functions for configuring PWM behavior.

SetPWMMode(address, mode) Sets the PWM mode for the controller.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **mode** (int): PWM mode - 0: Inductive Mode - 1: Resistive, Mode 5% Blanking - 2: Resistive, Mode 10% Blanking - 3: Resistive, Mode 15% Blanking - 4: Resistive, Mode 20% Blanking

Returns: - **bool:** True if successful, False otherwise

ReadPWMMode(address) Reads the PWM mode from the controller.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, mode) - **success** (bool): True if read successful - **mode** (int): PWM mode - 0: Inductive Mode - 1: Resistive, Mode 5% Blanking - 2: Resistive, Mode 10% Blanking - 3: Resistive, Mode 15% Blanking - 4: Resistive, Mode 20% Blanking

SetPWMIdle(address, idledelay1, idlemode1, idledelay2, idlemode2)
Sets the PWM idle parameters that control motor behavior when no commands are given.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **idledelay1** (float): Idle delay for motor 1 in seconds (0 to 12.7) - **idlemode1** (bool): Idle mode for motor 1 (True = enabled, False = disabled) - **idledelay2** (float): Idle delay for motor 2 in seconds (0 to 12.7) - **idlemode2** (bool): Idle mode for motor 2 (True = enabled, False = disabled)

Returns: - **bool:** True if successful, False otherwise

GetPWMIdle(address) Reads the PWM idle parameters.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, idledelay1, idlemode1, idledelay2, idlemode2) - **success** (bool): True if read successful - **idledelay1** (float): Idle delay for motor 1 in seconds - **idlemode1** (bool): Idle mode for motor 1 (True = enabled, False = disabled) - **idledelay2** (float): Idle delay for motor 2 in seconds - **idlemode2** (bool): Idle mode for motor 2 (True = enabled, False = disabled)

Example:

```
# Set complementary PWM mode
success = controller.SetPWMMode(address, 1)

# Set motor idle parameters
# Motor 1: 5 second delay, idle mode enabled
# Motor 2: 10 second delay, idle mode enabled
success = controller.SetPWMIdle(address, 5.0, True, 10.0, True)
```

Error Limit Configuration (Roboclaw, MCP)

Functions for configuring error limits that can trigger automatic responses like motor shutdown.

SetSpeedErrorLimit(address, limit1, limit2) Sets the speed error limits for both motors. If the difference between commanded speed and actual speed exceeds these limits, an error will be triggered.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **limit1** (int): Speed error limit for motor 1 in encoder counts per second - **limit2** (int): Speed error limit for motor 2 in encoder counts per second

Returns: - **bool:** True if successful, False otherwise

GetSpeedErrorLimit(address) Reads the speed error limits for both motors.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, limit1, limit2) - **success** (bool): True if read successful - **limit1** (int): Speed error limit for motor 1 - **limit2** (int): Speed error limit for motor 2

SetPosErrorLimit(address, limit1, limit2) Sets the position error limits for both motors. If the difference between commanded position and actual position exceeds these limits, an error will be triggered.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **limit1** (int): Position error limit for motor 1 in encoder counts - **limit2** (int): Position error limit for motor 2 in encoder counts

Returns: - **bool:** True if successful, False otherwise

GetPosErrorLimit(address) Reads the position error limits for both motors.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, limit1, limit2) - **success** (bool): True if read successful - **limit1** (int): Position error limit for motor 1 - **limit2** (int): Position error limit for motor 2

Example:

```
# Set speed error limits
# Motor 1: 500 counts/sec, Motor 2: 500 counts/sec
success = controller.SetSpeedErrorLimit(address, 500, 500)

# Set position error limits
# Motor 1: 100 counts, Motor 2: 100 counts
success = controller.SetPosErrorLimit(address, 100, 100)
```

Pin Configuration (Roboclaw only)

Functions for configuring digital I/O pins on Roboclaw controllers.

SetPinFunctions(address, S3mode, S4mode, S5mode) Sets the functions of pins S3, S4, and S5.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **S3mode** (int): Mode for pin S3 - **S4mode** (int): Mode for pin S4 - **S5mode** (int): Mode for pin S5 - **D1mode** (int): Mode for pin CTRL1 - **D2mode** (int): Mode for pin CTRL2

Returns: - **bool:** True if successful, False otherwise

ReadPinFunctions(address) Reads the functions of pins S3, S4, and S5.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, S3mode, S4mode, S5mode) - **success** (bool): True if read successful - **S3mode** (int): Mode for pin S3 - **S4mode** (int): Mode for pin S4 - **S5mode** (int): Mode for pin S5 - **D1mode** (int): Mode for pin S5 - **D2mode** (int): Mode for pin S5

SetCtrlSettings(address, S1revdeadband, S1fwddeadband, S1revlimit, S1fwdlimit, S1rangecenter, S1rangemin, S1rangemax, S2revdeadband, S2fwddeadband, S2revlimit, S2fwdlimit, S2rangecenter, S2rangemin, S2rangemax) Sets RC/Analog control settings.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **S1revdeadband** (int): Reverse deadband value (0-255) - **S1fwddeadband** (int): Forward deadband value (0-255) - **S1revlimit** (int): Reverse Limit value,

RC:0-3000,Analog(0-2047) - **S1fwdlimit** (int): Forward Limit value, RC:0-3000,Analog(0-2047) - **S1rangecenter** (int): Input Center - **S1rangemin** (int): Input Minimum - **S1rangemax** (int): Input Maximum - **S2revdeadband** (int): Reverse deadband value (0-255) - **S2fwddeadband** (int): Forward deadband value (0-255) - **S2revlimit** (int): Reverse Limit value, RC:0-3000,Analog(0-2047) - **S2fwdlimit** (int): Forward Limit value, RC:0-3000,Analog(0-2047) - **S2rangecenter** (int): Input Center - **S2rangemin** (int): Input Minimum - **S2rangemax** (int): Input Maximum

Returns: - bool: True if successful, False otherwise

GetCtrlSettings(address) Reads RC/Analog control settings.

Parameters: - address (int): The address of the controller (0x80-0x87)

Returns: - tuple: (success, min, max) - **success** (bool): True if read successful - **S1revdeadband** (int): Reverse deadband value (0-255) - **S1fwddeadband** (int): Forward deadband value (0-255) - **S1revlimit** (int): Reverse Limit value, RC:0-3000,Analog(0-2047) - **S1fwdlimit** (int): Forward Limit value, RC:0-3000,Analog(0-2047) - **S1rangecenter** (int): Input Center - **S1rangemin** (int): Input Minimum - **S1rangemax** (int): Input Maximum - **S2revdeadband** (int): Reverse deadband value (0-255) - **S2fwddeadband** (int): Forward deadband value (0-255) - **S2revlimit** (int): Reverse Limit value, RC:0-3000,Analog(0-2047) - **S2fwdlimit** (int): Forward Limit value, RC:0-3000,Analog(0-2047) - **S2rangecenter** (int): Input Center - **S2rangemin** (int): Input Minimum - **S2rangemax** (int): Input Maximum

Example:

```
# Set pin functions
# S3: Default, S4: E-Stop, S5: Disabled
success = controller.SetPinFunctions(address, 0, 1, 0)
```

SetAuxDutys(address, S3duty, S4duty, S5duty, D1duty, D2duty) Sets auxiliary PWM duty cycles for peripheral devices.

Parameters: - address (int): The address of the controller (0x80-0x87) - **S3duty** (int): Duty cycle for S3 (0 to 32767) - **S4duty** (int): Duty cycle for S4 (0 to 32767) - **S5duty** (int): Duty cycle for S5 (0 to 32767) - **D1duty** (int): Duty cycle for Ctrl1 (0 to 32767) - **D2duty** (int): Duty cycle for Ctrl2 (0 to 32767)

Returns: - bool: True if successful, False otherwise

GetAuxDutys(address) Gets auxiliary PWM duty cycles.

Parameters: - address (int): The address of the controller (0x80-0x87)

Returns: - tuple: (success, duty1, duty2, duty3, duty4, duty5) - **success** (bool): True if read successful - **S3duty** (int): Duty cycle for S3 (0 to 32767) -

S4duty (int): Duty cycle for S4 (0 to 32767) - **S5duty** (int): Duty cycle for S5 (0 to 32767) - **D1duty** (int): Duty cycle for Ctrl1 (0 to 32767) - **D2duty** (int): Duty cycle for Ctrl2 (0 to 32767)

SetAuto1(address, value) Sets M1 homing timeout value.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **value** (int): Auto mode configuration

Returns: - **bool**: True if successful, False otherwise

SetAuto2(address, value) Sets M2 homing timeout value.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **value** (int): Auto mode configuration

Returns: - **bool**: True if successful, False otherwise

GetAutos(address) Gets homing timeout values.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple**: (success, auto1, auto2) - **success** (bool): True if read successful - **auto1** (int): Auto mode 1 value - **auto2** (int): Auto mode 2 value

Example:

```
# Set a new address and enable mixing
success = controller.SetAddressMixed(address, 0x81, 1)

# Set auxiliary duty cycle for output 1 to 50% (16384)
success = controller.SetAuxDutys(address, 16384, 0, 0, 0, 0)
```

Digital Output Configuration (MCP only)

Functions for configuring digital outputs on MCP controllers.

SetDOUT(address, index, action) Sets a digital output pin action.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **index** (int): DOUT pin index - **action** (int): Action to perform

Returns: - **bool**: True if successful, False otherwise

GetDOUTS(address) Gets the digital outputs status.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple**: (success, count, actions) - **success** (bool): True if read successful - **count** (int): Number of digital outputs - **actions** (list): List of output actions

Advanced Configuration (MCP only)

Functions for advanced configuration options in MCP controllers.

SetPriority(address, priority1, priority2, priority3) Sets the priority levels for different operations.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **priority1** (int): Priority level 1 (0-255) - **priority2** (int): Priority level 2 (0-255) - **priority3** (int): Priority level 3 (0-255)

Returns: - **bool**: True if successful, False otherwise

GetPriority(address) Gets the priority levels.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple**: (success, priority1, priority2, priority3) - **success** (bool): True if read successful - **priority1** (int): Priority level 1 - **priority2** (int): Priority level 2 - **priority3** (int): Priority level 3

SetAddressMixed(address, new_address, enable_mixing) Sets a new address and mixing mode.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **new_address** (int): New address (0x80-0x87) - **enable_mixing** (int): Enable mixing (0 = disabled, 1 = enabled)

Returns: - **bool**: True if successful, False otherwise

GetAddressMixed(address) Gets the address and mixing mode.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple**: (success, new_address, mixed) - **success** (bool): True if read successful - **address** (int): Current address - **mixed** (int): Mixing mode (0 = disabled, 1 = enabled)

SetNodeID(address, nodeid) Sets the node ID for CAN networking.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **nodeid** (int): Node ID (0-255)

Returns: - **bool**: True if successful, False otherwise

GetNodeID(address) Gets the node ID.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple**: (success, nodeid) - **success** (bool): True if read successful - **nodeid** (int): Node ID

Signal Configuration (MCP only)

Functions for configuring signal processing in MCP controllers.

SetSignal(address, index, signal_type, mode, target, min_action, max_action, lowpass, timeout, loadhome, min_val, max_val, center, deadband, powerexp, minout, maxout, powermin, potentiometer) Sets complex signal parameters for input processing.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **index** (int): Signal index (0-7) - **signal_type** (int): Signal type (0-255) - **mode** (int): Signal mode (0-255) - **target** (int): Target handler (0-255) - **min_action** (int): Minimum action (0-65535) - **max_action** (int): Maximum action (0-65535) - **lowpass** (int): Lowpass filter (0-255) - **timeout** (int): Timeout in milliseconds - **loadhome** (int): Load home position - **min_val** (int): Minimum value - **max_val** (int): Maximum value - **center** (int): Center value - **deadband** (int): Deadband - **powerexp** (int): Power exponent - **minout** (int): Minimum output - **maxout** (int): Maximum output - **powermin** (int): Minimum power - **potentiometer** (int): Potentiometer

Returns: - **bool**: True if successful, False otherwise

GetSignals(address) Gets the signal parameters.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple**: (success, count, signals) - **success** (bool): True if read successful - **count** (int): Number of signals - **signals** (list): List of signal parameter dictionaries

GetSignalsData(address) Gets the signals data (current values).

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple**: (success, count, signals_data) - **success** (bool): True if read successful - **count** (int): Number of signals data - **signals_data** (list): List of signals data dictionaries

SetStream(address, index, stream_type, baudrate, timeout) Sets the stream parameters for data streaming.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **index** (int): Stream index (0-7) - **stream_type** (int): Stream type (0-255) - **baudrate** (int): Baudrate in bps - **timeout** (int): Timeout in milliseconds

Returns: - **bool**: True if successful, False otherwise

GetStreams(address) Gets the stream parameters.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, count, streams) - **success** (bool): True if read successful - **count** (int): Number of streams - **streams** (list): List of stream parameter dictionaries

Example:

```
# Configure a stream for UART communication at 115200 bps
success = controller.SetStream(address, 0, 1, 115200, 1000)
```

CAN Bus Functions (MCP only)

These functions enable communication with CAN bus devices through the MCP controller.

CANBufferState(address) Gets the count of available CAN packets in the receive buffer.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, count) - **success** (bool): True if read successful - **count** (int): Number of available CAN packets

CANPutPacket(address, cob_id, RTR, data) Sends a CAN packet.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **cob_id** (int): CAN object identifier (0 to 2047) - **RTR** (int): Remote Transmission Request (0 or 1) - **data** (list): List of data bytes (length must be <= 8 bytes)

Returns: - **bool:** True if successful, False otherwise

Raises: - **ValueError:** If data length is more than 8 bytes

CANGetPacket(address) Reads a CAN packet from the buffer.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, cob_id, RTR, length, data) - **success** (bool): True if read successful - **cob_id** (int): CAN object identifier - **RTR** (int): Remote Transmission Request - **length** (int): Length of the data - **data** (list): List of data bytes

CANOpenWriteLocalDict(address, wIndex, bSubindex, lValue, bSize)
Writes to the local CANopen dictionary.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **wIndex** (int): Index in the dictionary - **bSubindex** (int): Subindex in the dictionary - **lValue** (int): Value to write - **bSize** (int): Size of the value in bytes (1, 2, or 4)

Returns: - **tuple**: (success, lResult) - **success** (bool): True if successful - **lResult** (int): Result of the write operation

CANOpenReadLocalDict(address, wIndex, bSubindex) Reads from the local CANOpen dictionary.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **wIndex** (int): Index in the dictionary - **bSubindex** (int): Subindex in the dictionary

Returns: - **tuple**: (success, lValue, bSize, bType, lResult) - **success** (bool): True if read successful - **lValue** (int): Value read - **bSize** (int): Size of the value in bytes - **bType** (int): Type of the value - **lResult** (int): Result of the read operation

Example:

```
# Send a CAN packet
data = [0x01, 0x02, 0x03, 0x04]
success = controller.CANPutPacket(address, 0x123, 0, data)

# Read a CAN packet
success, cob_id, rtr, length, data = controller.CANGetPacket(address)
```

Advanced Functions

Script Control (MCP only)

Functions for controlling onboard scripts in MCP controllers.

StartScript(address) Starts the onboard script.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **bool**: True if successful, False otherwise

StopScript(address) Stops the onboard script.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **bool**: True if successful, False otherwise

GetScriptAutoRun(address) Gets the script auto run setting.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, scriptauto_time) - **success** (bool): True if read successful - **scriptauto_time** (int): Auto run time in milliseconds - 0: Script does not auto run - >0: Delay in milliseconds before auto run

SetScriptAutoRun(address, scriptauto_time) Sets the script auto run time.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **scriptauto_time** (int): Auto run time in milliseconds - 0: Disable auto run - 100: Delay in milliseconds before auto run

Returns: - **bool:** True if successful, False otherwise

Raises: - **ValueError:** If scriptauto_time is less than 100 and not 0

Example:

```
# Start the script
success = controller.StartScript(address)

# Get script autorun setting
success, autorun_time = controller.GetScriptAutoRun(address)
```

Emergency Stop (MCP only)

Functions for controlling emergency stop features in MCP controllers.

ResetEStop(address) Resets the emergency stop condition.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **bool:** True if successful, False otherwise

SetEStopLock(address, state) Sets the emergency stop lock state.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **state** (int): State value - 0x55: Automatic reset (reset E-Stop automatically when condition clears) - 0xAA: Software reset (requires ResetEStop() to be called) - 0: Hardware reset (requires physical reset)

Returns: - **bool:** True if successful, False otherwise

Raises: - **ValueError:** If state value is invalid

GetEStopLock(address) Gets the emergency stop lock state.

Parameters: - **address** (int): The address of the controller (0x80-0x87)

Returns: - **tuple:** (success, state) - **success** (bool): True if read successful - **state** (int): State value - 0x55: Automatic reset - 0xAA: Software reset - 0: Hardware reset

Example:

```
# Reset emergency stop
success = controller.ResetEStop(address)

# Set E-Stop to software reset mode
success = controller.SetEStopLock(address, 0xAA)
```

Legacy Motor Control

Legacy-Style Commands (0-127 values) (Roboclaw, MCP)

These commands use a 0-127 value range for compatibility with older versions:

ForwardM1(address, val) Sets the power for motor 1 to move forward.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **val** (int): The power value to set (0-127)

Returns: - **bool**: True if successful, False otherwise

BackwardM1(address, val) Sets the power for motor 1 to move backward.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **val** (int): The power value to set (0-127)

Returns: - **bool**: True if successful, False otherwise

ForwardM2(address, val) Sets the power for motor 2 to move forward.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **val** (int): The power value to set (0-127)

Returns: - **bool**: True if successful, False otherwise

BackwardM2(address, val) Sets the power for motor 2 to move backward.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **val** (int): The power value to set (0-127)

Returns: - **bool**: True if successful, False otherwise

ForwardBackwardM1(address, val) Sets the power for motor 1 using 7-bit mode (bidirectional from center point).

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **val** (int): The power value to set (0-127)

Returns: - **bool**: True if successful, False otherwise

ForwardBackwardM2(address, val) Sets the power for motor 2 using 7-bit mode (bidirectional from center point).

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **val** (int): The power value to set (0-127)

Returns: - **bool**: True if successful, False otherwise

For mixed mode (differential drive):

ForwardMixed(address, val) Sets the power for both motors to move forward in mixed mode.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **val** (int): The power value to set (0-127)

Returns: - **bool**: True if successful, False otherwise

BackwardMixed(address, val) Sets the power for both motors to move backward in mixed mode.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **val** (int): The power value to set (0-127)

Returns: - **bool**: True if successful, False otherwise

TurnRightMixed(address, val) Sets the power for motors to turn right in mixed mode.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **val** (int): The power value to set (0-127)

Returns: - **bool**: True if successful, False otherwise

TurnLeftMixed(address, val) Sets the power for motors to turn left in mixed mode.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **val** (int): The power value to set (0-127)

Returns: - **bool**: True if successful, False otherwise

ForwardBackwardMixed(address, val) Sets the forward/backward power in 7-bit mixed mode.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **val** (int): The power value to set (0-127)

Returns: - **bool**: True if successful, False otherwise

LeftRightMixed(address, val) Sets the left/right turning power in 7-bit mixed mode.

Parameters: - **address** (int): The address of the controller (0x80-0x87) - **val** (int): The power value to set (0-127)

Returns: - **bool**: True if successful, False otherwise

For complete details on each function, refer to the docstrings in the source code or consult the Basicmicro controller documentation.