CAN Motor Controller Library Documentation

Overview

The CAN Motor Controller Library provides a high-level, user-friendly Python interface for controlling CAN-based motors using a custom communication protocol. This library abstracts the low-level CAN protocol details and provides intuitive methods for motor control, monitoring, and configuration.

Installation

Requirements

- Python 3.7+
- pyserial package

Installation

```
pip install pyserial
```

Quick Start

Basic Usage

```python
from can_motor_controller import CANMotorController

# Create motor controller instance
motor = CANMotorController("/dev/tty.usbserial-2130", device_id=1)

# Connect to motor
motor.connect()

# Move to 90 degrees
motor.move_to(90)

# Disconnect
```

```
motor.disconnect()
```

Multiple Motors

```
# Control multiple motors on the same bus
motor1 = CANMotorController("COM3", device_id=1)
motor2 = CANMotorController("COM3", device_id=2)

motor1.connect()
motor2.connect()

motor1.move_to(45)
motor2.move_to(90)
```

API Reference

CANMotorController Class

Constructor

```
CANMotorController(port: str, device_id: int = 1, baudrate: int = 2000000)
```

Parameters:

- `port` (str): Serial port name (e.g., "/dev/ttyUSB0", "COM3", "/dev/tty.usbserial-2130")
- `device_id` (int): CAN device address (1-254). Default: 1
- `baudrate` (int): Serial communication baudrate. Default: 2000000

Raises:

- `ValueError`: If device_id is not between 1-254

Connection Management

`connect() -> bool`

Establish connection to the motor controller.

Returns:

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

Example:

```
if motor.connect():
    print("Connected successfully")
else:
    print("Connection failed")
```

`disconnect()`

Disconnect from the motor controller and disable the motor.

Example:

```
motor.disconnect()
```

Motor Control Methods

`move_to(angle_degrees: float, wait: bool = False, timeout: float = 5.0)`

Move to an absolute angle.

Parameters:

- `angle_degrees` (float): Target angle in degrees
- `wait` (bool): Wait for movement completion. Default: False
- `timeout` (float): Maximum wait time in seconds. Default: 5.0

Example:

```
# Move to 90 degrees and wait for completion
motor.move_to(90, wait=True, timeout=10.0)
```

`move_relative(angle_degrees: float, wait: bool = False, timeout: float = 5.0)`

Move relative to current position.

Parameters:

- `angle_degrees` (float): Relative angle in degrees
- `wait` (bool): Wait for movement completion. Default: False
- `timeout` (float): Maximum wait time in seconds. Default: 5.0

Example:

```
# Move 45 degrees clockwise from current position
motor.move_relative(45, wait=True)
```

`set_speed(speed_rpm: float)`

Set motor speed (RPM).

Parameters:

- `speed_rpm` (float): Speed in RPM (positive or negative)

Example:

```
motor.set_speed(100)  # 100 RPM clockwise
motor.set_speed(-50)  # 50 RPM counterclockwise
```

`set_current(current_amps: float)`

Set motor current for torque control.

Parameters:

- `current_amps` (float): Current in Amps (positive or negative)

Example:

```
motor.set_current(0.5)  # 0.5A torque
```

`stop()`

Stop the motor (set speed to 0).

Example:

```
motor.stop()
```

Home Position Methods

`set_home()`

Set current position as home position.

Example:

```
motor.set_home()
```

`go_home(wait: bool = False, timeout: float = 5.0)`

Return to home position via shortest path.

Parameters:

- `wait` (bool): Wait for movement completion. Default: False
- `timeout` (float): Maximum wait time in seconds. Default: 5.0

Example:

```
motor.go_home(wait=True)
```

`wait_for_move(timeout: float = 5.0, tolerance: float = 1.0)`

Wait until motor reaches target position.

Parameters:

- `timeout` (float): Maximum wait time in seconds. Default: 5.0
- `tolerance` (float): Position tolerance in degrees. Default: 1.0

Example:

```
motor.move_to(180)
motor.wait_for_move(timeout=10.0, tolerance=0.5)
```

System Status Methods

`get_status() -> Dict[str, Any]`

Get comprehensive motor status.

Returns:

- `Dict` containing:
  - `device_id`: CAN device address
  - `voltage`: Bus voltage in volts
  - `bus_current`: Bus current in amps
  - `temperature`: Temperature in °C
  - `operating_mode`: Current operating mode
  - `faults`: List of active faults
  - `state`: Motor state
  - `control_mode`: Current control mode
  - `current_angle`: Current position in degrees
  - `current_speed`: Current speed in RPM
  - `target_angle`: Target position in degrees

Example:

```
status = motor.get_status()
print(f"Position: {status['current_angle']}°")
print(f"Temperature: {status['temperature']}°C")
```

`update_position()`

Update the current position reading.

Example:

```
motor.update_position()
print(f"Current angle: {motor.current_angle}°")
```

`get_versions() -> Dict[str, Any]`

Get firmware and hardware versions.

Returns:

- `Dict` containing:
  - `device_id`: CAN device address
  - `bootloader`: Bootloader version
  - `firmware`: Firmware version
  - `hardware`: Hardware version
  - `can_protocol`: CAN protocol version

Example:

```
versions = motor.get_versions()
print(f"Firmware: {versions['firmware']}")
```

Safety Methods

`enable()`

Enable the motor (wake from disabled state).

Example:

```
motor.enable()
```

`disable()`

Disable the motor (free state, no torque).

Example:

```
motor.disable()
```

`emergency_stop()`

Immediately stop the motor and disable output.

Example:

```
motor.emergency_stop()
```

`clear_faults()`

Clear all fault conditions.

Example:

```
motor.clear_faults()
```

Brake Control Methods

`brake_on() -> bool`

Engage the brake.

Returns:

- `bool`: True if successful, False otherwise

Example:

```
if motor.brake_on():
    print("Brake engaged")
```

`brake_off() -> bool`

Release the brake.

Returns:

- `bool`: True if successful, False otherwise

Example:

```python
if motor.brake_off():
    print("Brake released")
```

`get_brake_status() -> bool`

Get brake status.

Returns:

- `bool`: True if brake engaged, False if released

Example:

```python
if motor.get_brake_status():
    print("Brake is engaged")
else:
    print("Brake is released")
```

Configuration Methods

`set_max_speed(max_rpm: float)`

Set maximum speed for position mode.

Parameters:

- `max_rpm` (float): Maximum speed in RPM

Example:

```python
motor.set_max_speed(200)  # 200 RPM maximum
```

`set_max_current(max_amps: float)`

Set maximum current for position/speed mode.

Parameters:

- `max_amps` (float): Maximum current in Amps

Example:

```python
motor.set_max_current(2.0)  # 2A maximum
```

Callback Management

`add_callback(event_type: str, callback: Callable)`

Add callback function for specific events.

Parameters:

- `event_type` (str): One of:
  - `'state_change'`: Motor state changes
  - `'fault'`: Fault detected
  - `'position_update'`: Position updated
  - `'status_update'`: Status updated
- `callback` (Callable): Function to call when event occurs

Callback Signatures:

- State change: `callback(state: MotorState, device_id: int)`
- Fault: `callback(faults: List[FaultType], device_id: int)`
- Position update: `callback(angle: float, device_id: int)`
- Status update: `callback(status: Dict[str, Any])`

Example:

```python
def on_state_change(state, device_id):
    print(f"Motor {device_id} state: {state.value}")

def on_fault(faults, device_id):
    print(f"Motor {device_id} faults: {[f.value for f in faults]}")

motor.add_callback('state_change', on_state_change)
motor.add_callback('fault', on_fault)
```

`remove_callback(event_type: str, callback: Callable)`

Remove callback function.

Parameters:

- `event_type` (str): Event type
- `callback` (Callable): Callback function to remove

Example:

```
motor.remove_callback('state_change', on_state_change)
```

Convenience Functions

`create_motor_controller(port: str, device_id: int = 1) -> CANMotorController`

Create and connect a motor controller.

Parameters:

- `port` (str): Serial port name
- `device_id` (int): CAN device address (1-254)

Returns:

- `CANMotorController`: Connected motor controller instance

Raises:

- `ConnectionError`: If connection fails
- `ValueError`: If device_id is invalid

Example:

```
from can_motor_controller import create_motor_controller

motor = create_motor_controller("/dev/ttyUSB0", device_id=1)
```

`create_multiple_motors(port: str, device_ids: List[int]) -> List[CANMotorController]`

Create and connect multiple motor controllers.

Parameters:

- `port` (str): Serial port name
- `device_ids` (List[int]): List of CAN device addresses

Returns:

- `List[CANMotorController]`: List of connected motor controllers

Example:

```python
from can_motor_controller import create_multiple_motors

motors = create_multiple_motors("/dev/ttyUSB0", [1, 2, 3])
```

Enumerations

`MotorState`

- `DISABLED`: Motor is disabled
- `ENABLED`: Motor is enabled
- `FAULT`: Motor has a fault
- `HOMING`: Motor is homing
- `MOVING`: Motor is moving
- `HOLDING`: Motor is holding position

`ControlMode`

- `CURRENT`: Current/torque control mode
- `SPEED`: Speed control mode
- `POSITION`: Position control mode
- `HOMING`: Homing mode
- `MIT`: MIT control mode

`FaultType`

- `NONE`: No faults
- `VOLTAGE`: Voltage fault
- `CURRENT`: Current fault
- `TEMPERATURE`: Temperature fault
- `ENCODER`: Encoder fault
- `HARDWARE`: Hardware fault

- `SOFTWARE`: Software fault

Usage Examples

Basic Position Control

```python
from can_motor_controller import CANMotorController

motor = CANMotorController("/dev/tty.usbserial-2130", device_id=1)
motor.connect()

# Move to specific angles
motor.move_to(0)      # Move to 0°
motor.move_to(90)     # Move to 90°
motor.move_to(180)    # Move to 180°

# Relative movements
motor.move_relative(45)   # Move 45° clockwise
motor.move_relative(-90)  # Move 90° counterclockwise

motor.disconnect()
```

Speed Control

```python
motor = CANMotorController("COM3", device_id=2)
motor.connect()

# Set different speeds
motor.set_speed(100)   # 100 RPM clockwise
time.sleep(2)
motor.set_speed(-50)   # 50 RPM counterclockwise
time.sleep(2)
motor.stop()          # Stop the motor

motor.disconnect()
```

Advanced Usage with Callbacks

```python
def on_state_change(state, device_id):
    print(f"Motor {device_id}: {state.value}")

def on_fault(faults, device_id):
    if any(fault != FaultType.NONE for fault in faults):
        print(f"Motor {device_id} fault! Clearing...")
        motor.clear_faults()

def on_position_update(angle, device_id):
    print(f"Motor {device_id} position: {angle:.1f}°")

motor = CANMotorController("/dev/ttyUSB0", device_id=1)
motor.add_callback('state_change', on_state_change)
motor.add_callback('fault', on_fault)
motor.add_callback('position_update', on_position_update)

motor.connect()
motor.move_to(90, wait=True)
motor.disconnect()
```

Multiple Motor Coordination

```python
# Control multiple motors simultaneously
motors = []
for i in range(3):
    motor = CANMotorController("/dev/ttyUSB0", device_id=i+1)
    motor.connect()
    motors.append(motor)

# Move all motors to different positions
targets = [0, 120, 240]  # 0°, 120°, 240°
for motor, target in zip(motors, targets):
    motor.move_to(target, wait=False)

# Wait for all to complete
for motor in motors:
    motor.wait_for_move()

# Return all to home
for motor in motors:
    motor.go_home(wait=False)

# Cleanup
for motor in motors:
    motor.disconnect()
```

Error Handling

Connection Errors

```python
try:
    motor = CANMotorController("/dev/ttyUSB0", device_id=1)
    if not motor.connect():
        print("Failed to connect - check port and power")
except Exception as e:
    print(f"Connection error: {e}")
```

Fault Handling

```python
motor.connect()
status = motor.get_status()

if status['faults']:
    print(f"Faults detected: {status['faults']}")
    motor.clear_faults()
```

Communication Errors

The library automatically handles most communication errors and will attempt to reconnect if possible. Critical errors will be raised as exceptions.

Troubleshooting

Common Issues

1. **Connection Failed**
   - Check serial port name
   - Verify motor power is on
   - Check cable connections

2. **Motor Not Responding**
   - Verify CAN device address
   - Check baudrate settings
   - Ensure motor is enabled

3. **Position Errors**
   - Check encoder connections
   - Verify home position is set
   - Check for mechanical obstructions

Debug Mode

For troubleshooting, enable debug output by monitoring the callback events:

```
def debug_callback(event, *args):
    print(f"Debug: {event} - {args}")

motor.add_callback('state_change', debug_callback)
motor.add_callback('fault', debug_callback)
```

Protocol Details

- **Baudrate**: 2000000 (default), configurable
- **CAN Address Range**: 1-254
- **Position Resolution**: 16384 counts/revolution (0.022° per count)
- **Speed Resolution**: 0.01 RPM
- **Current Resolution**: 0.001 A

License

This library is provided as-is. Please refer to your motor controller documentation for specific protocol details and limitations.