Motor Control Firmware

Generated by Doxygen 1.9.6

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

Class Documentation

3.1 CommandType Class Reference

Represents the types of commands recognized by the firmware.

```
#include <Commands.hpp>
```

3.1.1 Detailed Description

Represents the types of commands recognized by the firmware.

Author

Terry Paul Ferguson
terry@terryferguson.us

Version

0.1

The documentation for this class was generated from the following file:

- Commands.hpp

3.2 Direction Class Reference

Direction values for the direction indicator for the motor controller and the motors themselves.

```
#include <Direction.hpp>
```

3.2.1 Detailed Description

Direction values for the direction indicator for the motor controller and the motors themselves.

Author

Terry Paul Ferguson
`terry@terryferguson.us`

Version

0.1

The documentation for this class was generated from the following file:

- Direction.hpp

3.3 Motor Class Reference

This class represents the motor controlled by the microcontroller.

```
#include <Motor.hpp>
```

Public Member Functions

- Motor ()
- Motor (const char ∗name, const MotorPin rpwm, const MotorPin lpwm, const MotorPin r_en, const MotorPin l_en, const MotorPin hall_1, const MotorPin hall_2, const MotorPin lIS_pin, const MotorPin rIS_pin, const int totalPulses, const int freq=PWM_FREQUENCY, const int defSpeed=70, const int pwmRes=8)

  The constructor for the motor controlled by the microcontroller.
- void **initialize** ()

  Initialize motor.
- void **drive** (const Direction motorDirection, const int specifiedSpeed=0)
- void **extend** ()

  Tell the motor to rotate in the direction of extension.
- void **retract** ()

  Tell the motor to rotate in the direction of retraction.
- void **stop** ()

  Tell the motor to stop.
- void **zero** ()

  Zero out position information for this motor.
- void **home** ()

  Perform the homing routine to calibrate the position sensor for this motor.
- void **update** (const int newSpeed=(MAX_SPEED+1))

  Update the position information for this motor and move it.
- void setPos (const int newPos)

  Set a new target position for this motor.
- void **readPos** ()

  Read rotary encoder value into position variable.
- float getNormalizedPos () const

  Get a normalized indicaton of the position of this motor based on its total range.
- void **displayInfo** ()
- int getCurrent () const

  Get an analog value proportional to the current used by the motor. The max of the two channels is returned.
- void **setSpeed** (int newSpeed)

Public Attributes

- int pos
- int lastPos
- int speed = 255
- int maxPulses = -1
- Direction **dir** = Direction::STOP

3.3.1 Detailed Description

This class represents the motor controlled by the microcontroller.

Author

Terry Paul Ferguson

Version

0.1

3.3.2 Constructor & Destructor Documentation

3.3.2.1 Motor() [1/2]

```
Motor::Motor ( )  [inline]
```

The direction of the motor rotation

3.3.2.2 Motor() [2/2]

```
Motor::Motor (
            const char * name,
            const MotorPin rpwm,
            const MotorPin lpwm,
            const MotorPin r_en,
            const MotorPin l_en,
            const MotorPin hall_1,
            const MotorPin hall_2,
            const MotorPin lIS_pin,
            const MotorPin rIS_pin,
            const int totalPulses,
            const int freq = PWM_FREQUENCY,
            const int defSpeed = 70,
            const int pwmRes = 8 )  [inline]
```

The constructor for the motor controlled by the microcontroller.

Parameters

name	The name of this motor for debug prints
rpwm	The right PWM signal pin
lpwm	The left PWM signal pin
r_en	The right PWM enable pin
l_en	The left PWM enable pin
hall_1	The pin for hall sensor 1
hall_2	The pin for hall sensor 2
lIS_pin	The pin for left current sensor
rIS_pin	The pin for right current sensor
totalPulses	The total number of pulses from full retraction to full extension
freq	The frequency of the PWM signal
defSpeed	The default motor speed
pwmRes	The PWM bitdepth resolution

Copy name of linear actuator into ID field

3.3.3 Member Function Documentation

3.3.3.1 getCurrent()

```
int Motor::getCurrent ( ) const  [inline]
```

Get an analog value proportional to the current used by the motor. The max of the two channels is returned.

Returns

The larger of the two current values used by the motor

3.3.3.2 getNormalizedPos()

```
float Motor::getNormalizedPos ( ) const  [inline]
```

Get a normalized indicaton of the position of this motor based on its total range.

Returns

A fraction that represents how much of total extension we are currently at

3.3.3.3 setPos()

```
void Motor::setPos (
             const int newPos )  [inline]
```

Set a new target position for this motor.

Parameters

newPos	The new target position to move the motor to

3.3.4 Member Data Documentation

3.3.4.1 lastPos

```
int Motor::lastPos
```

Initial value:

```
=
      0
```

The current position of the motor based on hall sensor pulses

3.3.4.2 maxPulses

```
int Motor::maxPulses = -1
```

The current speed of the motor. The duty cycle of the PWM signal is speed/(2^pwmResolution - 1)

3.3.4.3 pos

```
int Motor::pos
```

Initial value:

```
=
      0
```

The motor position encoder (quadrature signal from 2 hall sensors)

3.3.4.4 speed

```
int Motor::speed = 255
```

The last position of the motor based on hall sensor pulses

The documentation for this class was generated from the following file:

- Motor.hpp

3.4 MotorController Class Reference

This is the controller of the motors.

```
#include <MotorController.hpp>
```

Public Member Functions

- MotorController (const int pwmFrequency=PWM_FREQUENCY, const int pwmResolution=PWM_RESOLUTION_BITS, const int defaultSpeed=DEFAULT_MOTOR_SPEED, const int currentIncreaseLimit=DEFAULT_CURRENT_INCREASE_LIMIT)

  This class controls the motors connected to the microcontoller.
- void **initialize** ()

  Load the stored position preferences into RAM and initialize the motors.
- void **extend** ()

  Tell the motorized system to extend.
- void **retract** ()

  Tell the motorized system to retract.
- void **stop** ()

  Tell the motorized system to stop.
- void **home** ()

  Home the linear actuator to recalibrate the position sensor.
- void **zero** ()

  Clear all position information.
- void setSpeed (int newSpeed)

  Smoothly change to the newly requested speed.
- int getSpeed () const

  Get the system speed.
- bool countsAreUnequal (void) const

  Indicates whether the motor counts are unequal.
- void **report** ()

  Report debugging information to the serial console.
- void savePosition (const int slot, const int position_value)

  Save a position to the preferences slot.
- void setPos (const int newPos)

  Move the motors to the given position.
- bool isStopped () const

  Check whether the system is in a STOP state.
- void update (const float deltaT=0.0f)

  Perform one update interval for the motor system.

Public Attributes

- int K_p = 100000
- float K_i = 0.1f
- float eIntegral = 0.0f
- int defaultSpeed = DEFAULT_MOTOR_SPEED
- int speed = 0
- int currentIncreaseTolerance

3.4.1 Detailed Description

This is the controller of the motors.

Author

Terry Paul Ferguson

terry@terryferguson.us

Version

0.1

3.4.2 Constructor & Destructor Documentation

3.4.2.1 MotorController()

```
MotorController::MotorController (
            const int pwmFrequency = PWM_FREQUENCY,
            const int pwmResolution = PWM_RESOLUTION_BITS,
            const int defaultSpeed = DEFAULT_MOTOR_SPEED,
            const int currentIncreaseLimit = DEFAULT_CURRENT_INCREASE_LIMIT )  [inline]
```

This class controls the motors connected to the microcontoller.

Parameters

pwmFrequency	The frequency of the PWM signal to the motors
pwmResolution	The bitdepth resolution of the PWM signal to the motors
defaultSpeed	The default speed of the motors that the control program starts them off with

3.4.3 Member Function Documentation

3.4.3.1 countsAreUnequal()

```
bool MotorController::countsAreUnequal (
            void ) const  [inline]
```

Indicates whether the motor counts are unequal.

Returns

True if the motor counts are different, false otherwise

3.4.3.2 getSpeed()

```
int MotorController::getSpeed ( ) const  [inline]
```

Get the system speed.

Returns

The average speed of the system

3.4.3.3 isStopped()

```
bool MotorController::isStopped ( ) const  [inline]
```

Check whether the system is in a STOP state.

Returns

True if the system is in a STOP state, else false

3.4.3.4 savePosition()

```
void MotorController::savePosition (
             const int slot,
             const int position_value )  [inline]
```

Save a position to the preferences slot.

Parameters

slot	The selected slot to save the position information to
position_value	The position value in hall sensor pulses to save to the selected slot

3.4.3.5 setPos()

```
void MotorController::setPos (
             const int newPos )  [inline]
```

Move the motors to the given position.

Parameters

newPos	The new target position for the motors in hall sensor pulses

3.4.3.6 setSpeed()

```
void MotorController::setSpeed (
             int newSpeed )  [inline]
```

Smoothly change to the newly requested speed.

Parameters

newSpeed	The new speed to target

3.4.3.7 update()

```
void MotorController::update (
             const float deltaT = 0.0f )  [inline]
```

Perform one update interval for the motor system.

Parameters

deltaT	The amount of time that has passed since the last update

3.4.4 Member Data Documentation

3.4.4.1 currentIncreaseTolerance

```
int MotorController::currentIncreaseTolerance
```

Initial value:

```
=
     DEFAULT_CURRENT_INCREASE_LIMIT
```

Maxim1um current increase limit for motor cutoff

3.4.4.2 defaultSpeed

```
int MotorController::defaultSpeed = DEFAULT_MOTOR_SPEED
```

The default speed to operate the motors at on startup

3.4.4.3 eIntegral

```
float MotorController::eIntegral = 0.0f
```

The integral error coefficient for the PID controller

3.4.4.4 K_i

```
float MotorController::K_i = 0.1f
```

The intagral gain for the PID controller

3.4.4.5 K_p

```
int MotorController::K_p = 100000
```

The proprotional gain for the PID controller

3.4.4.6 speed

```
int MotorController::speed = 0
```

Current target speed

The documentation for this class was generated from the following file:

- MotorController.hpp

3.5 MotorPins Class Reference

Pin number definitions for the motor.

```
#include <MotorPins.hpp>
```

3.5.1 Detailed Description

Pin number definitions for the motor.

Pin number definitions for the potentiometer controlled parameters.

Author

Terry Paul Ferguson
terry@terryferguson.us

This has the pin numbering to wire to the microcontroller

Author

Terry Paul Ferguson
terry@terryferguson.us

Version

0.1

The documentation for this class was generated from the following file:

- MotorPins.hpp

3.6 PIDController Class Reference

Public Member Functions

- **PIDController** (float kp=0.1, float ti=0.002, float td=0.01, float uMax=255.0)
- void **setParams** (float kpIn, float kdIn, float kiIn, float uMaxIn=255.0)
- void **evaluate** (int value, int target, float deltaT, int &speed, Direction &dir)
- void **report** (const int value, const int target, const float deltaT, const int speed, const Direction dir) const

The documentation for this class was generated from the following file:

- PIDController.hpp

Chapter 4

File Documentation

4.1 Commands.hpp File Reference

```
#include <cstdint>
```

Enumerations

- enum class Command : std::uint32_t {
  RETRACT = 17 , EXTEND , REPORT , STOP ,
  SAVE_TILT_1 , SAVE_TILT_2 , SAVE_TILT_3 , SAVE_TILT_4 ,
  SAVE_TILT_5 , GET_TILT_1 , GET_TILT_2 , GET_TILT_3 ,
  GET_TILT_4 , GET_TILT_5 , ZERO , SYSTEM_RESET }

4.1.1 Enumeration Type Documentation

4.1.1.1 Command

```
enum class Command :  std::uint32_t  [strong]
```

Enumerator

RETRACT	Command to tell motors to retract - 17
EXTEND	Command to tell motors to extend - 18
REPORT	Command to tell tell the motor controller to report its state - 19
STOP	Command to tell the motor controller to stop - 20
SAVE_TILT_1	Save value to stored position slot 1 - 21
SAVE_TILT_2	Save value to stored position slot 2 - 22
SAVE_TILT_3	Save value to stored position slot 3 - 23
SAVE_TILT_4	Save value to stored position slot 4 - 24
SAVE_TILT_5	Save value to stored position slot 5 - 25
GET_TILT_1	Get value from stored position slot 1 - 26

Enumerator

GET_TILT_2	Get value from stored position slot 2 - 27
GET_TILT_3	Get value from stored position slot 3 - 28
GET_TILT_4	Get value from stored position slot 4 - 29
GET_TILT_5	Get value from stored position slot 5 - 30
ZERO	Command to tell tell the motor controller to reset position counters - 31
SYSTEM_RESET	Command to tell tell the microcontroller to reset - 32

4.2 Commands.hpp

Go to the documentation of this file.

```

#ifndef _COMMANDS_HPP_
#define _COMMANDS_HPP_

#include <cstdint>

enum class Command : std::uint32_t {
  RETRACT  = 17,

  EXTEND,

  REPORT,

  STOP,

  SAVE_TILT_1,

  SAVE_TILT_2,

  SAVE_TILT_3,

  SAVE_TILT_4,

  SAVE_TILT_5,

  GET_TILT_1,

  GET_TILT_2,

  GET_TILT_3,

  GET_TILT_4,

  GET_TILT_5,

  ZERO,

  SYSTEM_RESET
};

#endif // _COMMANDS_HPP_
```

4.3 CurrentSettings.hpp File Reference

Macros

- #define **ADC_BITS** 12
- #define **ADC_MAX** (2 << ADC_BITS)
- #define **LOGICAL_LEVEL_VOLTAGE** 3.3f
- #define **DEFAULT_CURRENT_INCREASE_LIMIT** ((int)((0.07 ∗ LOGICAL_LEVEL_VOLTAGE) ∗ ADC_MAX))
- #define **CURRENT_INCREASE_LIMIT_MAX** ((int)((0.15 ∗ LOGICAL_LEVEL_VOLTAGE) ∗ ADC_MAX))

Functions

- bool currentIncreseExceedsThreshold (const int currentSensePin, const int baseValue, const int threshold)

  Indicate whether the increase in current on the driver current sense pin has exceeded the threshold.

4.3.1 Function Documentation

4.3.1.1 currentIncreseExceedsThreshold()

```
bool currentIncreseExceedsThreshold (
             const int currentSensePin,
             const int baseValue,
             const int threshold )
```

Indicate whether the increase in current on the driver current sense pin has exceeded the threshold.

Parameters

currentSensePin	The current sense pin input from the motor driver
baseValue	The base value of the current of the motor (min load)
threshold	The threshold value to use

Returns

true if the increase exceeds threshold value, else false

4.4 CurrentSettings.hpp

Go to the documentation of this file.

```

#ifndef _CURRENT_SETTINGS_HPP_
#define _CURRENT_SETTINGS_HPP_

#define ADC_BITS 12

#define ADC_MAX (2 « ADC_BITS)

#define LOGICAL_LEVEL_VOLTAGE 3.3f

#define DEFAULT_CURRENT_INCREASE_LIMIT                                          \
  ((int)((0.07 * LOGICAL_LEVEL_VOLTAGE) * ADC_MAX))

#define CURRENT_INCREASE_LIMIT_MAX                                              \
  ((int)((0.15 * LOGICAL_LEVEL_VOLTAGE) * ADC_MAX))

bool currentIncreseExceedsThreshold(const int currentSensePin, const int baseValue,
                             const int threshold) {
  int currentValue = analogRead(currentSensePin);

  return (currentValue - baseValue) >= threshold;
}

#endif // _CURRENT_SETTINGS_HPP_
```

4.5 defs.hpp File Reference

```
#include "Commands.hpp"
#include "MotorPins.hpp"
#include "PotentiometerPins.hpp"
#include "Direction.hpp"
```

Macros

- #define **FORMAT_SPIFFS_IF_FAILED** true
- #define **NUM_POSITION_SLOTS** 5
- #define **PWM_FREQUENCY** 15000
- #define **PWM_RESOLUTION_BITS** 8
- #define **DEFAULT_MOTOR_SPEED** 100
- #define **MICROS_IN_MS** 1000
- #define **SOFT_MOVEMENT_TIME_MS** 2000
- #define **SOFT_MOVEMENT_MICROS** (SOFT_MOVEMENT_TIME_MS ∗ MICROS_IN_MS)
- #define **SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS** 20000
- #define **SOFT_MOVEMENT_UPDATE_STEPS** (SOFT_MOVEMENT_MICROS / SOFT_MOVEMENT_↩ PWM_UPDATE_INTERVAL_MICROS)
- #define **MAX_SPEED** (2 << (PWM_RESOLUTION_BITS)-1)
- #define **MIN_SPEED** (2 << (PWM_RESOLUTION_BITS)-1) ∗ -1

Variables

- const char ∗ **motor_roles** [2] = {"LEADER", "FOLLOWER"}
- const char ∗ save_position_slot_names [NUM_POSITION_SLOTS]
- int **savedPositions** [NUM_POSITION_SLOTS] = {0, 0, 0, 0, 0}
- bool **debugEnabled** = true

  Indicates whether debug messages should be sent to serial.

4.5.1 Variable Documentation

4.5.1.1 save_position_slot_names

```
const char* save_position_slot_names[NUM_POSITION_SLOTS]
```

Initial value:

```
= {
    "tilt-1", "tilt-2", "tilt-3", "tilt-4", "tilt-5",
}
```

4.6 defs.hpp

Go to the documentation of this file.

```

#ifndef _DEFS_HPP_
#define _DEFS_HPP_

#define FORMAT_SPIFFS_IF_FAILED true

#include "Commands.hpp"
#include "MotorPins.hpp"
#include "PotentiometerPins.hpp"
#include "Direction.hpp"

//@brief String representations of the motor roles at instantiation
const char *motor_roles[2] = {"LEADER", "FOLLOWER"};

#define NUM_POSITION_SLOTS 5
const char *save_position_slot_names[NUM_POSITION_SLOTS] = {
    "tilt-1", "tilt-2", "tilt-3", "tilt-4", "tilt-5",
};

//@brief Storage for position in hall sensor pusles relative to initial position
//when powered on
int savedPositions[NUM_POSITION_SLOTS] = {0, 0, 0, 0, 0};

bool debugEnabled = true;

#define PWM_FREQUENCY 15000

#define PWM_RESOLUTION_BITS 8

#define DEFAULT_MOTOR_SPEED 100

#define MICROS_IN_MS 1000

#define SOFT_MOVEMENT_TIME_MS 2000

#define SOFT_MOVEMENT_MICROS (SOFT_MOVEMENT_TIME_MS * MICROS_IN_MS)

#define SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS 20000

#define SOFT_MOVEMENT_UPDATE_STEPS                                                    \
  (SOFT_MOVEMENT_MICROS / SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS)

#define MAX_SPEED (2 « (PWM_RESOLUTION_BITS)-1)

#define MIN_SPEED (2 « (PWM_RESOLUTION_BITS)-1) * -1

#endif // _DEFS_HPP_

```

4.7 Direction.hpp File Reference

Enumerations

- enum class Direction { EXTEND = 0 , STOP , RETRACT }

Variables

- const char * **directions** [3] = {"EXTEND", "STOP", "RETRACT"}

  String representations of the directions.

4.7.1 Enumeration Type Documentation

4.7.1.1 Direction

```
enum class Direction  [strong]
```

Enumerator

EXTEND	Motor is turning for extensions
STOP	Motor is stopped
RETRACT	Motor is turning for retraction

4.8 Direction.hpp

Go to the documentation of this file.

```

#ifndef _DIRECTION_HPP_
#define _DIRECTION_HPP_

enum class Direction {
  EXTEND = 0,

  STOP,

  RETRACT
};

const char *directions[3] = {"EXTEND", "STOP", "RETRACT"};

#endif // _DIRECTION_HPP_
```

4.9 Motor.hpp File Reference

```
#include "defs.hpp"
#include "PinMacros.hpp"
#include <ESP32Encoder.h>
#include <cstring>
```

Classes

- class Motor

  This class represents the motor controlled by the microcontroller.

Macros

- #define **READ_POSITION_ENCODER**() this->pos = distanceSensor.getCount();
- #define MOVE_TO_POS(setpoint, min_delta, buffer)

Variables

- int **currentPWMChannel** = 0

4.9.1 Macro Definition Documentation

4.9.1.1 MOVE_TO_POS

```
#define MOVE_TO_POS(
              setpoint,
              min_delta,
              buffer )
```

Value:

```
  if (abs(pos - setpoint) > min_delta) {                                        \
    if (pos < setpoint) {                                                       \
      desiredPos = setpoint - buffer;                                           \
    } else if (pos > newPos) {                                                  \
      desiredPos = setpoint + buffer;                                           \
    }                                                                           \
  }
```

4.10 Motor.hpp

Go to the documentation of this file.

```

#ifndef _MOTOR_HPP_
#define _MOTOR_HPP_

#include "defs.hpp"
#include "PinMacros.hpp"
#include <ESP32Encoder.h>
#include <cstring>

#define READ_POSITION_ENCODER() this->pos = distanceSensor.getCount();
#define MOVE_TO_POS(setpoint, min_delta, buffer)                               \
  if (abs(pos - setpoint) > min_delta) {                                       \
    if (pos < setpoint) {                                                      \
      desiredPos = setpoint - buffer;                                          \
    } else if (pos > newPos) {                                                 \
      desiredPos = setpoint + buffer;                                          \
    }                                                                          \
  }

int currentPWMChannel = 0;

class Motor {
private:
  char id[16];
  int pwmRChannel = -1;
  int pwmLChannel = -1;
  MotorPin rPWM_Pin = MotorPin::UNASSIGNED;
  MotorPin lPWM_Pin = MotorPin::UNASSIGNED;
  MotorPin r_EN_Pin = MotorPin::UNASSIGNED;
  MotorPin l_EN_Pin = MotorPin::UNASSIGNED;
  MotorPin hall_1_Pin = MotorPin::UNASSIGNED;
  MotorPin hall_2_Pin = MotorPin::UNASSIGNED;
  MotorPin l_is_pin = MotorPin::UNASSIGNED;
  MotorPin r_is_pin = MotorPin::UNASSIGNED;
  int frequency = PWM_FREQUENCY;
  int pwmResolution = 8;
  int desiredPos =
      -1;
  int totalPulseCount = 0;
  ESP32Encoder distanceSensor;
public:
  int pos =
      0;
  int lastPos =
      0;
  int speed = 255;
  int maxPulses = -1;

  Direction dir =  Direction::STOP;
  Motor() {} // end default constructor

  Motor(const char *name,
        const MotorPin rpwm,
        const MotorPin lpwm,
        const MotorPin r_en,
        const MotorPin l_en,
        const MotorPin hall_1,
        const MotorPin hall_2,
        const MotorPin lIS_pin,
        const MotorPin rIS_pin,
        const int totalPulses,
        const int freq = PWM_FREQUENCY,
        const int defSpeed = 70,
        const int pwmRes = 8)
      : rPWM_Pin(rpwm), lPWM_Pin(lpwm), r_EN_Pin(r_en), l_EN_Pin(l_en),
        hall_1_Pin(hall_1), hall_2_Pin(hall_2), l_is_pin(lIS_pin),
        r_is_pin(rIS_pin), totalPulseCount(totalPulses), frequency(freq),
        speed(defSpeed), pwmResolution(pwmRes) {
    strncpy(id, name, sizeof(id) - 1);
    id[sizeof(id) - 1] = ’\0’;
  } // end constructor

  void initialize() {
    // At least two channels are needed for the linear actuator motor
    if (currentPWMChannel > -1 && currentPWMChannel < 14) {
      pwmRChannel = currentPWMChannel++;
      pwmLChannel = currentPWMChannel++;
    }

    ledcSetup(pwmRChannel, frequency, pwmResolution);
    ledcSetup(pwmLChannel, frequency, pwmResolution);

    motorAttachPin(rPWM_Pin, pwmRChannel);
    motorAttachPin(lPWM_Pin, pwmLChannel);

    motorPinMode(r_EN_Pin, OUTPUT);
    motorPinMode(l_EN_Pin, OUTPUT);

    motorPinWrite(r_EN_Pin, HIGH);
    motorPinWrite(l_EN_Pin, HIGH);

    ledcWrite(pwmRChannel, 0);
    ledcWrite(pwmLChannel, 0);

    distanceSensor.attachSingleEdge(
      static_cast<int>(hall_1_Pin),
      static_cast<int>(hall_2_Pin));
    distanceSensor.clearCount();
    READ_POSITION_ENCODER()

    if (debugEnabled) {
      Serial.printf("Motor: %s\n"
                  "--------------------\n"
                  "Frequency:    %5d\n"
                  "Resolution:   %5d\n"
                  "Speed:        %5d\n"
                  "Position:     %5d\n"
                  "RPWM Pin:     %5d\n"
                  "LPWM Pin:     %5d\n"
                  "Hall 1 Pin:   %5d\n"
                  "Hall 2 Pin:   %5d\n"
                  "Max Position: %5d\n\n",
                  id,
                  frequency,
                  pwmResolution,
                  speed,
                  pos,
                  rPWM_Pin,
                  lPWM_Pin,
                  hall_1_Pin,
                  hall_2_Pin,
                  totalPulseCount);
    }
  }

  void drive(const Direction motorDirection, const int specifiedSpeed = 0) {
    const int driveSpeed = specifiedSpeed > 0 ? specifiedSpeed : speed;

    switch (motorDirection) {
      case Direction::EXTEND:
        motorPinWrite(r_EN_Pin, HIGH);
        motorPinWrite(l_EN_Pin, HIGH);
        ledcWrite(pwmRChannel, driveSpeed);
        ledcWrite(pwmLChannel, 0);
        break;
      case Direction::STOP:
        motorPinWrite(r_EN_Pin, HIGH);
        motorPinWrite(l_EN_Pin, HIGH);
        ledcWrite(pwmRChannel, 0);
        ledcWrite(pwmLChannel, 0);
        motorPinWrite(r_EN_Pin, LOW);
        motorPinWrite(l_EN_Pin, LOW);
        break;
      case Direction::RETRACT:
        motorPinWrite(r_EN_Pin, HIGH);
```

```
        motorPinWrite(l_EN_Pin, HIGH);
        ledcWrite(pwmRChannel, 0);
        ledcWrite(pwmLChannel, driveSpeed);
        break;
      default:
        break;
    } // end direction handler

    lastPos = pos;
    READ_POSITION_ENCODER()
  } // end drive

  void extend() {
    // Works as a toggle
    dir = (dir != Direction::EXTEND) ? Direction::EXTEND : Direction::STOP;
  }

  void retract() {
    // Works as a toggle
    dir = (dir != Direction::RETRACT) ? Direction::RETRACT : Direction::STOP;
  }

  void stop() {
    // Works as a toggle
    dir = Direction::STOP;
  }

  void zero() {
    distanceSensor.clearCount();
    lastPos = pos = 0;
  }

  void home() {
    // First retract as much as possible

    int sameCount = 0;
    int firstSameTime = 0;
    dir = Direction::RETRACT;
    while (sameCount < 1000) {
      drive(dir, MAX_SPEED);
      if (lastPos == pos) {
        if (sameCount == 0) {
          firstSameTime = millis();
        } else {
          if (millis() - firstSameTime > 1000)
            break;
        }
        sameCount++;
      } else {
        sameCount = 0;
      }
      READ_POSITION_ENCODER()
    }

    Serial.println("Fully retracted");

    sameCount = 0;
    firstSameTime = 0;
    dir = Direction::EXTEND;
    while (sameCount < 1000) {
      drive(dir, MAX_SPEED);
      if (lastPos == pos) {
        if (sameCount == 0) {
          firstSameTime = millis();
        } else {
          if (millis() - firstSameTime > 1000)
            break;
        }
        sameCount++;
      } else {
        sameCount = 0;
      }
      READ_POSITION_ENCODER()
    }

    Serial.print("Fully extended. Max pulse: ");
    Serial.println(pos);
    maxPulses = pos;

    sameCount = 0;
    firstSameTime = 0;
    dir = Direction::RETRACT;
    while (sameCount < 1000) {
      drive(dir, MAX_SPEED);
      if (lastPos == pos) {
        if (sameCount == 0) {
          firstSameTime = millis();
        } else {
          if (millis() - firstSameTime > 1000)
            break;
        }
        sameCount++;
      } else {
        sameCount = 0;
      }
      READ_POSITION_ENCODER()
    }

    Serial.println("Fully retracted");
    dir = Direction::STOP;
  }

  void update(const int newSpeed = (MAX_SPEED + 1)) {
    if (desiredPos >= 0) {
      if (pos > desiredPos) {
        dir = Direction::RETRACT;
      } else if (pos < desiredPos) {
        dir = Direction::EXTEND;
      } else {
        dir = Direction::STOP;
        desiredPos = -1;
        displayInfo();
      }
    }

    // For lift column - Extension limit
    if (dir == Direction::EXTEND && (pos) > totalPulseCount) {
      dir = Direction::STOP;
      return;
    }

    // For lift column - Retraction limit
    if (dir == Direction::RETRACT && (pos) < 50) {
      dir = Direction::STOP;
      return;
    }

    if (newSpeed > MAX_SPEED || newSpeed < 0) {
      drive(dir, this->speed);
    } else {
      drive(dir, newSpeed);
    }
  }

  void setPos(const int newPos) {
    READ_POSITION_ENCODER()
    MOVE_TO_POS(newPos, 15, 40)
  }

  void readPos() { READ_POSITION_ENCODER() }

  float getNormalizedPos() const {
    return (float)this->pos / (float)this->totalPulseCount;
  }

  void displayInfo() {
    Serial.printf("Motor %s - Direction: %s, pos: %d\n", id, directions[static_cast<int>(dir)],
                  pos);
    Serial.printf("Motor %s - Speed: %d, desired pos: %d\n", id, speed,
                  desiredPos);
    Serial.printf("Motor %s - Max hall position: %d \n\n", id, totalPulseCount);
  }

  int getCurrent() const {
    const int leftCurrent = motorAnalogRead(l_is_pin);
    const int rightCurrent = motorAnalogRead(r_is_pin);

    return max(leftCurrent, rightCurrent);
  }

  void setSpeed(int newSpeed) { speed = newSpeed; }
}; // end class Motor

#endif // _MOTOR_HPP_
```

4.11 MotorController.hpp File Reference

```
#include <Preferences.h>
#include "Motor.hpp"
```

```
#include "PIDController.hpp"
#include "PinMacros.hpp"
#include "defs.hpp"
```

Classes

- class MotorController

  This is the controller of the motors.

Macros

- #define **NUMBER_OF_MOTORS** 2
- #define ALL_MOTORS(operation)
- #define **ALL_MOTORS_COMMAND**(command) ALL_MOTORS(motors[motor].command();)
- #define RESET_SOFT_MOVEMENT
- #define **RESTORE_POSITION**(slot) motor_controller.setPos(savedPositions[slot]);
- #define SERIAL_SAVE_POSITION(slot)

4.11.1 Macro Definition Documentation

4.11.1.1 ALL_MOTORS

```
#define ALL_MOTORS(
                operation )
```

Value:

```
  for (int motor = 0; motor < NUMBER_OF_MOTORS; motor++) {                        \
    operation                                                                      \
  }
```

4.11.1.2 RESET_SOFT_MOVEMENT

```
#define RESET_SOFT_MOVEMENT
```

Value:

```
  pwmUpdateAmount = 0;                                                            \
  lastPWMUpdate = -1;                                                             \
  softStart = -1;                                                                 \
  targetSpeed = -1;                                                               \
  eIntegral = 0.0f;
```

4.11.1.3 SERIAL_SAVE_POSITION

```
#define SERIAL_SAVE_POSITION(
                slot )
```

Value:

```
  if (Serial.available() > 0) {                                                \
    int new_pos = Serial.parseInt();                                           \
    motor_controller.savePosition(slot, new_pos);                              \
  }
```

4.12 MotorController.hpp

Go to the documentation of this file.

```

#ifndef _MOTOR_CONTROLLER_HPP_
#define _MOTOR_CONTROLLER_HPP_

#include <Preferences.h>

#include "Motor.hpp"
#include "PIDController.hpp"
#include "PinMacros.hpp"
#include "defs.hpp"

#define NUMBER_OF_MOTORS 2

#define ALL_MOTORS(operation)                                                  \
  for (int motor = 0; motor < NUMBER_OF_MOTORS; motor++) {                     \
    operation                                                                  \
  }

#define ALL_MOTORS_COMMAND(command) ALL_MOTORS(motors[motor].command();)

#define RESET_SOFT_MOVEMENT                                                    \
  pwmUpdateAmount = 0;                                                         \
  lastPWMUpdate = -1;                                                          \
  softStart = -1;                                                              \
  targetSpeed = -1;                                                            \
  eIntegral = 0.0f;

#define RESTORE_POSITION(slot) motor_controller.setPos(savedPositions[slot]);

#define SERIAL_SAVE_POSITION(slot)                                             \
  if (Serial.available() > 0) {                                                \
    int new_pos = Serial.parseInt();                                           \
    motor_controller.savePosition(slot, new_pos);                              \
  }


class MotorController {
private:
  Motor motors[NUMBER_OF_MOTORS];

  enum MotorRoles {
    LEADER,
    FOLLOWER
  };

  // const int motorPulseTotals[2] = {8060, 8057};

  const int motorPulseTotals[2] = {2055, 2050};

  int laggingIndex = 0;

  int leadingIndex = 0;

  int softStart = -1;

  int lastPWMUpdate = -1;

  int targetSpeed = -1;

  float pwmUpdateAmount =
      -1.0f;

  int lastPrintTime = -1;

  const int printDelta = 333000;

  // PIDController pidController;

  Direction systemDirection =
      Direction::STOP;

  Direction requestedDirection =
      Direction::STOP;

  int pwmFrequency =
      PWM_FREQUENCY;

  int pwmResolution = PWM_RESOLUTION_BITS;

  int initialCurrentReadings[NUMBER_OF_MOTORS] = {0, 0};

  Preferences positionStorage; //

  void loadPositions() {
    for (int slot = 0; slot < NUM_POSITION_SLOTS; slot++) {
      savedPositions[slot] =
          positionStorage.getInt(save_position_slot_names[slot]);
    }
  }

  void immediateHalt() {
    speed = targetSpeed = 0;
    systemDirection = Direction::STOP;
    requestedDirection = Direction::STOP;
    RESET_SOFT_MOVEMENT

    ALL_MOTORS(motors[motor].speed = 0;)
  }

  void initializeMotors() {
    ALL_MOTORS_COMMAND(initialize)
    immediateHalt();
  }

public:
  int K_p = 100000;

  float K_i = 0.1f;

  float eIntegral = 0.0f;

  int defaultSpeed = DEFAULT_MOTOR_SPEED;

  int speed = 0;

  int currentIncreaseTolerance =
      DEFAULT_CURRENT_INCREASE_LIMIT;

  MotorController(
      const int pwmFrequency = PWM_FREQUENCY,
      const int pwmResolution = PWM_RESOLUTION_BITS,
      const int defaultSpeed = DEFAULT_MOTOR_SPEED,
      const int currentIncreaseLimit = DEFAULT_CURRENT_INCREASE_LIMIT)
      : pwmFrequency(pwmFrequency),
        pwmResolution(pwmResolution),
        defaultSpeed(defaultSpeed),
        currentIncreaseTolerance(currentIncreaseLimit) {
    if (debugEnabled) {
      char buf[256];
      sprintf(
          buf,
          "Controller Params: Frequency: %d - Resolution: %d - Duty Cycle: %d\n",
          pwmFrequency, pwmResolution, defaultSpeed);
      Serial.println(buf);
    }
  }

  void initialize() {
    // Read in saved positions
    // Open in read-write mode
    motors[0] =
        Motor("Leader",
              MotorPin::MOTOR1_RPWM_PIN,
              MotorPin::MOTOR1_LPWM_PIN,
              MotorPin::MOTOR1_R_EN_PIN,
              MotorPin::MOTOR1_L_EN_PIN,
              MotorPin::MOTOR1_HALL1_PIN,
              MotorPin::MOTOR1_HALL2_PIN,
              MotorPin::MOTOR1_LIS_PIN,
              MotorPin::MOTOR1_RIS_PIN,
              motorPulseTotals[0],
              PWM_FREQUENCY,
              defaultSpeed,
              pwmResolution);

    motors[1] =
        Motor("Follower",
              MotorPin::MOTOR2_RPWM_PIN,
              MotorPin::MOTOR2_LPWM_PIN,
              MotorPin::MOTOR2_R_EN_PIN,
              MotorPin::MOTOR2_L_EN_PIN,
              MotorPin::MOTOR2_HALL1_PIN,
              MotorPin::MOTOR2_HALL2_PIN,
              MotorPin::MOTOR2_LIS_PIN,
              MotorPin::MOTOR2_RIS_PIN,
              motorPulseTotals[1],
              PWM_FREQUENCY,
              defaultSpeed,
              pwmResolution);

    positionStorage.begin("evox-tilt", false);
    loadPositions();
    initializeMotors();
    Serial.println("System initialized.");

    ALL_MOTORS(initialCurrentReadings[motor] = motors[motor].getCurrent();)
  }

  void extend() {
    // SET_TO_ANALOG_PIN_FUNC(SPEED_POT_PIN, this->setSpeed, 0, 2 «
    // PWM_RESOLUTION_BITS - 1);
    setSpeed(defaultSpeed);
    ALL_MOTORS_COMMAND(extend)
    systemDirection = Direction::EXTEND;
    requestedDirection = Direction::EXTEND;
  }

  void retract() {
    // SET_TO_ANALOG_PIN_FUNC(SPEED_POT_PIN, this->setSpeed, 0, 2 «
    // PWM_RESOLUTION_BITS - 1);
    setSpeed(defaultSpeed);
    ALL_MOTORS_COMMAND(retract)
    systemDirection = Direction::RETRACT;
    requestedDirection = Direction::RETRACT;
  }

  void stop() {
    RESET_SOFT_MOVEMENT

    setSpeed(0);
    requestedDirection = Direction::STOP;
  }

  void home() { ALL_MOTORS_COMMAND(home) }

  void zero() { ALL_MOTORS_COMMAND(zero) }

  void setSpeed(int newSpeed) {
    targetSpeed = newSpeed;
    softStart = lastPWMUpdate = micros();

    // Calculate the difference between the current speed and the requested
    // speed and divide that difference by the number of update steps to get
    // the PWM duty cycle increase/decrease per step.
    //
    // This will usually have a fractional part, so we make it a float value. We
    // handle the rounding and conversion to an integer in the update method.
    pwmUpdateAmount =
        ceil((float)abs(targetSpeed - speed) / SOFT_MOVEMENT_UPDATE_STEPS);

    // If the new speed is lower, make it negative, as we add the
    // pwmUpdateAmount to the speed
    if (targetSpeed < speed) {
      pwmUpdateAmount = -pwmUpdateAmount;
    }

    if (debugEnabled) {
      Serial.printf("MotorController\n"
                  "------------\n"
                  "setSpeed(%d)\n"
                  "speed: %3d\n"
                  "target speed: %3d\n"
                  "pwmUpdateAmount: %3.6f\n\n",
                  newSpeed,
                  speed,
                  targetSpeed,
                  pwmUpdateAmount);
    }
  }

  int getSpeed() const {
    // Return the average
    return (motors[0].speed + motors[1].speed) / 2;
  }

  bool countsAreUnequal(void) const {
    bool areUnequal = true;
    ALL_MOTORS(areUnequal &= motors[motor].pos == motors[motor].lastPos;)
    return areUnequal;
  }

  void report() {
    Serial.printf("MotorController\n--------------------\nSpeed: %d\nTarget "
                  "Speed: %d\nK_p: %d\nK_i: %f\neIntegral:%f\npwmUpdateAmont: %f \n",
                  speed, targetSpeed, K_p, K_i, eIntegral, pwmUpdateAmount);
    Serial.print("Leading motor: ");
    Serial.println(motor_roles[leadingIndex]);
    Serial.print("Lagging motor: ");
    Serial.println(motor_roles[laggingIndex]);
    Serial.printf("\n\n\n");

    ALL_MOTORS_COMMAND(displayInfo)
  }

  /*
  void pidReport(const float deltaT) const {
    int leaderPos = motors[0].pos;
    int followerPos = motors[1].pos;
    int followerSpeed = motors[1].speed;
    Direction dir = motors[1].dir;

    pidController.report(leaderPos, followerPos, deltaT, followerSpeed, dir);
  }

  */

  void savePosition(const int slot, const int position_value) {
    if (slot > 0 && slot < NUM_POSITION_SLOTS && position_value > -1) {
      ALL_MOTORS(motors[motor].setPos(position_value);)
      savedPositions[slot - 1] = position_value;
      positionStorage.putInt(save_position_slot_names[slot], position_value);
    }
  }

  void setPos(const int newPos) { ALL_MOTORS(motors[motor].setPos(newPos);) }

  bool isStopped() const { return systemDirection == Direction::STOP; }

  void update(const float deltaT = 0.0f) {
    // Only update PID if we're not stopped
    if (Direction::STOP != systemDirection) {
      ALL_MOTORS_COMMAND(readPos)

      if (Direction::EXTEND == systemDirection) {
        if (motors[LEADER].getNormalizedPos() <
            motors[FOLLOWER].getNormalizedPos()) {
          laggingIndex = LEADER;
          leadingIndex = FOLLOWER;
        } else {
          laggingIndex = FOLLOWER;
          leadingIndex = LEADER;
        }
      } else {
        if (motors[LEADER].getNormalizedPos() >
            motors[FOLLOWER].getNormalizedPos()) {
          laggingIndex = LEADER;
          leadingIndex = FOLLOWER;
        } else {
          laggingIndex = FOLLOWER;
          leadingIndex = LEADER;
        }
      }
    }

    // Difference between current speed and target speed
    const int speedDelta = abs(speed - targetSpeed);

    // Get current time
    const int currentTime = micros();

    // Time since soft move has started
    const int moveTimeDelta = currentTime - softStart;

    // Time since PWM duty cycle (speed) was last modified
    const int updateTimeDelta = currentTime - lastPWMUpdate;

    // If we have a target speed
    if (targetSpeed >= 0) {
      // Are we ready to update?
      if (updateTimeDelta >= SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS) {
        // If so, first check if the distance to target is less than our step
        // amount. Or if the time to ramp up has expired
        if (speedDelta < abs(pwmUpdateAmount) ||
            moveTimeDelta >= SOFT_MOVEMENT_MICROS) {
          // If so, then set the current speed to target speed and reset state
          speed = targetSpeed;
          RESET_SOFT_MOVEMENT

          // If the requested direction is to stop, the finally update the system
          // direction to stop
          if (requestedDirection == Direction::STOP) {
            systemDirection = Direction::STOP;
            if (debugEnabled) {
              Serial.println("System Direction: STOP");
              const double timeSinceSoftStart = (double) (micros() - softStart) / (double)
MICROS_IN_MS;

              Serial.printf("Soft Movement PWM Update - "
                      "speed: %d - "
                      "target speed: %d - "
                      "time since soft start: %f ms - "
                      "pwmUpdateAmount: %f\n",
                  speed,
                  targetSpeed,
                  timeSinceSoftStart,
                  pwmUpdateAmount);
            }

            // Tell all motors to stop
            immediateHalt();
          }
        }

        // Otherwise, add the PWM update ammount to the current duty cycle
        // (speed) Round the result and then convert to an integer
        const float newSpeed = (float)speed + pwmUpdateAmount;
        speed = (int)floorf(newSpeed);

        // We just updated, so update last update timestamp
        lastPWMUpdate = micros();

        const double timeSinceSoftStart = (double) (micros() - softStart) / (double) MICROS_IN_MS;

        if (debugEnabled) {
          Serial.printf("Soft Movement PWM Update - "
                      "speed: %d - "
                      "target speed: %d - "
                      "time since soft start: %f ms - "
                      "pwmUpdateAmount: %f\n",
          speed,
          targetSpeed,
          timeSinceSoftStart,
          pwmUpdateAmount);
        }
      }
    }

    /*
int maxCurrent = max(motors[leadingIndex].getCurrent(),
                     motors[laggingIndex].getCurrent());
*/

    // The error is the difference between the normalization of positions of
    // the two motors
    const float error = abs(motors[laggingIndex].getNormalizedPos() -
                            motors[leadingIndex].getNormalizedPos());

    // Add the error from this timestep to the integral term
    eIntegral += error * deltaT;

    // Calculate the adjusted speed to set to the faster motor
    //const int adjustedSpeed = speed - int((error * K_p) + (K_i * eIntegral));
    const int adjustedSpeed = speed - int((error * K_p));

    // Slow down the faster motor, keeping the value in the acceptable range
    motors[leadingIndex].speed =
        constrain(adjustedSpeed, 0, 2 « pwmResolution);
    motors[laggingIndex].speed = speed;

    // Update the motors with the adjustments
    ALL_MOTORS_COMMAND(update)
  }
```

```
};

#endif // _MOTOR_CONTROLLER_HPP_
```

4.13 MotorPins.hpp File Reference

```
#include <cstdint>
```

Enumerations

- enum class MotorPin : std::uint8_t {
UNASSIGNED = 0 , MOTOR1_RPWM_PIN = 25 , MOTOR1_LPWM_PIN = 19 , MOTOR1_R_EN_PIN = 26 , MOTOR1_L_EN_PIN = 18 , MOTOR1_HALL1_PIN = 22 , MOTOR1_HALL2_PIN = 23 , MOTOR1_LIS_PIN = 22 ,
MOTOR1_RIS_PIN = 23 , MOTOR2_RPWM_PIN = 5 , MOTOR2_LPWM_PIN = 17 , MOTOR2_R_EN_PIN = 16 ,
MOTOR2_L_EN_PIN = 15 , MOTOR2_HALL1_PIN = 14 , MOTOR2_HALL2_PIN = 13 , MOTOR2_LIS_PIN = 29 ,
MOTOR2_RIS_PIN = 30 }

4.13.1 Enumeration Type Documentation

4.13.1.1 MotorPin

```
enum class MotorPin : std::uint8_t [strong]
```

Enumerator

UNASSIGNED	NULL pin for unassigned
MOTOR1_RPWM_PIN	Motor RPWM Pin for extension square wave
MOTOR1_LPWM_PIN	Motor LPWM Pin for extension square wave
MOTOR1_R_EN_PIN	Enable pin for RPWM channel (extension)
MOTOR1_L_EN_PIN	Enable pin for LPWM channel (retraction)
MOTOR1_HALL1_PIN	Hall 1 sensor pin
MOTOR1_HALL2_PIN	Hall 2 sensor pin
MOTOR1_LIS_PIN	Left motor channel current sense pin
MOTOR1_RIS_PIN	Right motor channel current sense pin
MOTOR2_RPWM_PIN	Motor RPWM Pin for extension square wave
MOTOR2_LPWM_PIN	Motor LPWM Pin for retraction square wave
MOTOR2_R_EN_PIN	Enable pin for RPWM channel (extension)
MOTOR2_L_EN_PIN	Enable pin for LPWM channel (retraction)
MOTOR2_HALL1_PIN	Hall 1 sensor pin
MOTOR2_HALL2_PIN	Hall 2 sensor pin
MOTOR2_LIS_PIN	Left motor channel current sense pin
MOTOR2_RIS_PIN	Right motor channel current sense pin

4.14 MotorPins.hpp

Go to the documentation of this file.

```

#ifndef _MOTOR_PINS_HPP_
#define _MOTOR_PINS_HPP_

#include <cstdint>

enum class MotorPin : std::uint8_t {
  UNASSIGNED = 0,

  MOTOR1_RPWM_PIN = 25,

  MOTOR1_LPWM_PIN = 19,

  MOTOR1_R_EN_PIN = 26,

  MOTOR1_L_EN_PIN = 18,

  MOTOR1_HALL1_PIN = 22,

  MOTOR1_HALL2_PIN = 23,

  MOTOR1_LIS_PIN = 22,

  MOTOR1_RIS_PIN = 23,

  MOTOR2_RPWM_PIN = 5,

  MOTOR2_LPWM_PIN = 17,

  MOTOR2_R_EN_PIN = 16,

  MOTOR2_L_EN_PIN = 15,

  MOTOR2_HALL1_PIN = 14,

  MOTOR2_HALL2_PIN = 13,

  MOTOR2_LIS_PIN = 29,

  MOTOR2_RIS_PIN = 30,
};

#endif // _MOTOR_PINS_HPP_
```

4.15 PIDController.hpp File Reference

```
#include "defs.hpp"
#include <math.h>
#include <cstring>
```

Classes

- class PIDController

4.16 PIDController.hpp

Go to the documentation of this file.

```

#ifndef _PID_CONTROLLER_HPP__
#define _PID_CONTROLLER_HPP__

#include "defs.hpp"
#include <math.h>
```

```
#include <cstring>

class PIDController {
private:
  float kp;               // the controller path proportional gain
  float ti;               // the controller’s integrator time constant
  float td;               // the controller’s derivative time constant
  float uMax;             // Maximum magnitude of control signal
  float ePrev, eIntegral; // Storage

public:
  PIDController(float kp = 0.1,
                float ti = 0.002,
                float td = 0.01,
                float uMax = 255.0)
      : kp(kp), ti(ti), td(td), uMax(uMax), ePrev(0.0), eIntegral(0.0) {}

  // A function to set the parameters
  void setParams(float kpIn, float kdIn, float kiIn, float uMaxIn = 255.0) {
    kp = kpIn;
    td = kdIn;
    ti = kiIn;
    uMax = uMaxIn;
  }

  // A function to compute the control signal
  void evaluate(int value, int target, float deltaT, int& speed, Direction& dir) {
     // error
    int e = target - value;

    // derivative
    float dedt = (e - ePrev) / (deltaT);

    // integral
    eIntegral = eIntegral + e * deltaT;

    // control signal
    float u = kp * e + td * dedt + ti * eIntegral;

    // motor power
    speed = (int)fabs(u);
    if (speed > uMax) {
      speed = uMax;
    }

    // motor direction
    dir = Direction::EXTEND;
    if (u < 0) {
      dir = Direction::RETRACT;
    } else if (u == 0) {
        dir = Direction::STOP;
    }


    // store previous error
    ePrev = e;
  }

  void report(const int value, const int target, const float deltaT, const int speed, const Direction dir) const {
    char buf[256]; // Diagnostic messages
    sprintf(buf, "PID params: value:%d target: %d, deltaT: %f, speed: %d, dir: %s", value, target, deltaT, speed, directions[static_cast<int>(dir)]);
    Serial.println(buf);
  }
};

#endif // _PID_CONTROLLER_HPP__
```

4.17 PinMacros.hpp File Reference

Macros

- #define **FSET_TO_ANALOG_PIN**(pin, var_to_set, range_min, range_max)   var_to_set = fmap(analogRead(pin), 0, 4096, range_min, range_max)
- #define **SET_TO_ANALOG_PIN**(pin, var_to_set, range_min, range_max)   var_to_set = map(analogRead(pin), 0, 4096, range_min, range_max)

- #define **SET_TO_ANALOG_PIN_FUNC**(pin, func, range_min, range_max) func(map(analogRead(pin), 0, 4096, range_min, range_max))
- #define **motorPinWrite**(pin, value) digitalWrite(static_cast<std::uint8_t>(pin), value)
- #define **motorPinWrite**(pin, value) digitalWrite(static_cast<std::uint8_t>(pin), value)
- #define **motorPinMode**(pin, value) pinMode(static_cast<std::uint8_t>(pin), value)
- #define **motorAttachPin**(pin, channel) ledcAttachPin(static_cast<std::uint8_t>(pin), channel)
- #define **motorAnalogRead**(pin) analogRead(static_cast<std::uint8_t>(pin))

Functions

- float fmap (float x, float in_min, float in_max, float out_min, float out_max)

  Map 12-bit ADC value to a value within defined range.

4.17.1 Function Documentation

4.17.1.1 fmap()

```
float fmap (
             float x,
             float in_min,
             float in_max,
             float out_min,
             float out_max )
```

Map 12-bit ADC value to a value within defined range.

Parameters

x	12-bit ADC value
in_min	Minimum input value
in_max	Maximum input value
out_min	Minimum output value
out_max	Maximum output value

Returns

Mapped output value for input value

4.18 PinMacros.hpp

Go to the documentation of this file.

```

#ifndef _PIN_MACROS_HPP_
#define _PIN_MACROS_HPP_

float fmap(float x, float in_min, float in_max, float out_min, float out_max) {
  return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}
```

```

#define FSET_TO_ANALOG_PIN(pin, var_to_set, range_min, range_max) \
    var_to_set = fmap(analogRead(pin), 0, 4096, range_min, range_max)

#define SET_TO_ANALOG_PIN(pin, var_to_set, range_min, range_max) \
    var_to_set = map(analogRead(pin), 0, 4096, range_min, range_max)

#define SET_TO_ANALOG_PIN_FUNC(pin, func, range_min, range_max) \
func(map(analogRead(pin), 0, 4096, range_min, range_max))

#define motorPinWrite(pin, value) \
  digitalWrite(static_cast<std::uint8_t>(pin), value)

#define motorPinWrite(pin, value) \
  digitalWrite(static_cast<std::uint8_t>(pin), value)

#define motorPinMode(pin, value) \
  pinMode(static_cast<std::uint8_t>(pin), value)

#define motorAttachPin(pin, channel) \
  ledcAttachPin(static_cast<std::uint8_t>(pin), channel)

#define motorAnalogRead(pin) \
  analogRead(static_cast<std::uint8_t>(pin))

#endif // _PIN_MACROS_HPP_
```

4.19 PotentiometerPins.hpp File Reference

```
#include <cstdint>
```

Enumerations

- enum class PotentiometerPins : std::uint8_t { SPEED_POT_PIN = 35 , KP_POT_PIN = 32 }

4.19.1 Enumeration Type Documentation

4.19.1.1 PotentiometerPins

```
enum class PotentiometerPins : std::uint8_t [strong]
```

Enumerator

SPEED_POT_PIN	Speed potentiometer pin
KP_POT_PIN	PID gain potentiometer pin

4.20 PotentiometerPins.hpp

Go to the documentation of this file.

```

#ifndef _POTENTIOMTER_PINS_HPP_
#define _POTENTIOMTER_PINS_HPP_
```

```

#include <cstdint>

enum class PotentiometerPins : std::uint8_t {
  SPEED_POT_PIN = 35,

  KP_POT_PIN = 32,
};

#endif // _POTENTIOMTER_PINS_HPP_
```

4.21 RouteMacros.hpp File Reference

Macros

- #define SET_TILT(n)
- #define **DEF_HANDLER**(func) [ ](AsyncWebServerRequest ∗request) { func }
- #define LOAD_SAVED_POSITION(position, response_text)
- #define MOTOR_COMMAND(command, response_text)
- #define SET_POS_HANDLER(slot)
- #define **STATIC_FILE**(filename, file_type) request->send(SPIFFS, filename, file_type);

4.21.1 Macro Definition Documentation

4.21.1.1 LOAD_SAVED_POSITION

```
#define LOAD_SAVED_POSITION(
                position,
                response_text )
```

Value:

```
  motor_controller.setPos(savedPositions[position]);                         \
  request->send(200, "text/plain", response_text);
```

4.21.1.2 MOTOR_COMMAND

```
#define MOTOR_COMMAND(
                command,
                response_text )
```

Value:

```
  motor_controller.command();                                                \
  request->send(200, "text/plain", response_text);
```

4.21.1.3 SET_POS_HANDLER

```
#define SET_POS_HANDLER(
              slot )
```

Value:

```
String inputMessage1;                                                     \
SET_TILT(slot)                                                            \
request->send(200, "text/plain", inputMessage1);
```

4.21.1.4 SET_TILT

```
#define SET_TILT(
              n )
```

Value:

```
if (request->hasParam(PARAM_INPUT_1)) {                                   \
  inputMessage1 = request->getParam(PARAM_INPUT_1)->value();              \
  const int new_pos = inputMessage1.toInt();                                   \
  motor_controller.savePosition(n, new_pos);                              \
} else {                                                                  \
  inputMessage1 = "Error: No position sent.";                             \
}
```

4.22 RouteMacros.hpp

Go to the documentation of this file.

```

#ifndef _ROUTE_MACROS_HPP__
#define _ROUTE_MACROS_HPP__

#define SET_TILT(n)                                                               \
  if (request->hasParam(PARAM_INPUT_1)) {                                         \
    inputMessage1 = request->getParam(PARAM_INPUT_1)->value();                    \
    const int new_pos = inputMessage1.toInt();                                         \
    motor_controller.savePosition(n, new_pos);                                    \
  } else {                                                                        \
    inputMessage1 = "Error: No position sent.";                                   \
  }

#define DEF_HANDLER(func) [](AsyncWebServerRequest *request) { func }

#define LOAD_SAVED_POSITION(position, response_text)                              \
  motor_controller.setPos(savedPositions[position]);                              \
  request->send(200, "text/plain", response_text);

#define MOTOR_COMMAND(command, response_text)                                     \
  motor_controller.command();                                                     \
  request->send(200, "text/plain", response_text);

#define SET_POS_HANDLER(slot)                                                     \
  String inputMessage1;                                                           \
  SET_TILT(slot)                                                                  \
  request->send(200, "text/plain", inputMessage1);

#define STATIC_FILE(filename, file_type)                                          \
  request->send(SPIFFS, filename, file_type);

#endif // _ROUTE_MACROS_HPP__
```

Index