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 currentIncrease↩ Limit=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.

```
00001
00003 #ifndef _COMMANDS_HPP_
00004 #define _COMMANDS_HPP_
00005
00006 #include <cstdint>
00007
00019 enum class Command : std::uint32_t {
00021   RETRACT  = 17,
00022
00024   EXTEND,
00025
00027   REPORT,
00028
00030   STOP,
00031
00033   SAVE_TILT_1,
00034
00036   SAVE_TILT_2,
00037
00039   SAVE_TILT_3,
00040
00042   SAVE_TILT_4,
00043
00045   SAVE_TILT_5,
00046
00048   GET_TILT_1,
00049
00051   GET_TILT_2,
00052
00054   GET_TILT_3,
00055
00057   GET_TILT_4,
00058
00060   GET_TILT_5,
00061
00063   ZERO,
00064
00066   SYSTEM_RESET
00067 };
00068
00069 #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.

```
00001
00003 #ifndef _CURRENT_SETTINGS_HPP_
00004 #define _CURRENT_SETTINGS_HPP_
00005
00006 #define ADC_BITS 12
00007
00008 #define ADC_MAX (2 << ADC_BITS)
00009
00010 #define LOGICAL_LEVEL_VOLTAGE 3.3f
00011
00012 #define DEFAULT_CURRENT_INCREASE_LIMIT                                    \
00013   ((int)((0.07 * LOGICAL_LEVEL_VOLTAGE) * ADC_MAX))
00014
00015 #define CURRENT_INCREASE_LIMIT_MAX                                        \
00016   ((int)((0.15 * LOGICAL_LEVEL_VOLTAGE) * ADC_MAX))
00017
00023 bool currentIncreseExceedsThreshold(const int currentSensePin, const int baseValue,
00024                                     const int threshold) {
00025   int currentValue = analogRead(currentSensePin);
00026
00027   return (currentValue - baseValue) >= threshold;
00028 }
00029
00030 #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.

```
00001
00003 #ifndef _DEFS_HPP_
00004 #define _DEFS_HPP_
00005
00006 #define FORMAT_SPIFFS_IF_FAILED true
00007
00008 #include "Commands.hpp"
00009 #include "MotorPins.hpp"
00010 #include "PotentiometerPins.hpp"
00011 #include "Direction.hpp"
00012
00013 //@brief String representations of the motor roles at instantiation
00014 const char *motor_roles[2] = {"LEADER", "FOLLOWER"};
00015
00016 #define NUM_POSITION_SLOTS 5
00017 const char *save_position_slot_names[NUM_POSITION_SLOTS] = {
00018     "tilt-1", "tilt-2", "tilt-3", "tilt-4", "tilt-5",
00019 };
00020
00021 //@brief Storage for position in hall sensor pusles relative to initial position
00022 //when powered on
00023 int savedPositions[NUM_POSITION_SLOTS] = {0, 0, 0, 0, 0};
00024
00026 bool debugEnabled = true;
00027
00028 #define PWM_FREQUENCY 15000
00029
00030 #define PWM_RESOLUTION_BITS 8
00031
00032 #define DEFAULT_MOTOR_SPEED 100
00033
00034 #define MICROS_IN_MS 1000
00035
00036 #define SOFT_MOVEMENT_TIME_MS 2000
00037
00038 #define SOFT_MOVEMENT_MICROS (SOFT_MOVEMENT_TIME_MS * MICROS_IN_MS)
00039
00040 #define SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS 20000
00041
00042 #define SOFT_MOVEMENT_UPDATE_STEPS                                          \
00043   (SOFT_MOVEMENT_MICROS / SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS)
00044
00045 #define MAX_SPEED (2 << (PWM_RESOLUTION_BITS)-1)
00046
00047 #define MIN_SPEED (2 << (PWM_RESOLUTION_BITS)-1) * -1
00048
00049 #endif // _DEFS_HPP_
00050
```

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.

```
00001
00003 #ifndef _DIRECTION_HPP_
00004 #define _DIRECTION_HPP_
00005
00018 enum class Direction {
00020   EXTEND = 0,
00021
00023   STOP,
00024
00026   RETRACT
00027 };
00028
00030 const char *directions[3] = {"EXTEND", "STOP", "RETRACT"};
00031
00032 #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.

```
00001
00003 #ifndef _MOTOR_HPP_
00004 #define _MOTOR_HPP_
00005
00006 #include "defs.hpp"
00007 #include "PinMacros.hpp"
00008 #include <ESP32Encoder.h>
00009 #include <cstring>
00010
00011 #define READ_POSITION_ENCODER() this->pos = distanceSensor.getCount();
00012 #define MOVE_TO_POS(setpoint, min_delta, buffer)                        \
00013   if (abs(pos - setpoint) > min_delta) {                                \
00014     if (pos < setpoint) {                                               \
00015       desiredPos = setpoint - buffer;                                   \
00016     } else if (pos > newPos) {                                          \
00017       desiredPos = setpoint + buffer;                                   \
00018     }                                                                   \
00019   }
00020
00021 int currentPWMChannel = 0;
00022
00030 class Motor {
00031 private:
00032   char id[16];
00033   int pwmRChannel = -1;
00034   int pwmLChannel = -1;
00035   MotorPin rPWM_Pin = MotorPin::UNASSIGNED;
00036   MotorPin lPWM_Pin = MotorPin::UNASSIGNED;
00037   MotorPin r_EN_Pin = MotorPin::UNASSIGNED;
00038   MotorPin l_EN_Pin = MotorPin::UNASSIGNED;
00039   MotorPin hall_1_Pin = MotorPin::UNASSIGNED;
00040   MotorPin hall_2_Pin = MotorPin::UNASSIGNED;
00041   MotorPin l_is_pin = MotorPin::UNASSIGNED;
00042   MotorPin r_is_pin = MotorPin::UNASSIGNED;
00043   int frequency = PWM_FREQUENCY;
00044   int pwmResolution = 8;
00045   int desiredPos =
00046       -1;
00047   int totalPulseCount = 0;
00050   ESP32Encoder distanceSensor;
00053 public:
00054   int pos =
00055       0;
00056   int lastPos =
00057       0;
00058   int speed = 255;
00060   int maxPulses = -1;
00061
00062   Direction dir =  Direction::STOP;
00064   Motor() {} // end default constructor
00065
00081   Motor(const char *name,
00082         const MotorPin rpwm,
00083         const MotorPin lpwm,
00084         const MotorPin r_en,
00085         const MotorPin l_en,
00086         const MotorPin hall_1,
00087         const MotorPin hall_2,
```

```
00088         const MotorPin lIS_pin,
00089         const MotorPin rIS_pin,
00090         const int totalPulses,
00091         const int freq = PWM_FREQUENCY,
00092         const int defSpeed = 70,
00093         const int pwmRes = 8)
00094       : rPWM_Pin(rpwm), lPWM_Pin(lpwm), r_EN_Pin(r_en), l_EN_Pin(l_en),
00095         hall_1_Pin(hall_1), hall_2_Pin(hall_2), l_is_pin(lIS_pin),
00096         r_is_pin(rIS_pin), totalPulseCount(totalPulses), frequency(freq),
00097         speed(defSpeed), pwmResolution(pwmRes) {
00099     strncpy(id, name, sizeof(id) - 1);
00100     id[sizeof(id) - 1] = '\0';
00101   } // end constructor
00102
00104   void initialize() {
00105     // At least two channels are needed for the linear actuator motor
00106     if (currentPWMChannel > -1 && currentPWMChannel < 14) {
00107       pwmRChannel = currentPWMChannel++;
00108       pwmLChannel = currentPWMChannel++;
00109     }
00110
00111     ledcSetup(pwmRChannel, frequency, pwmResolution);
00112     ledcSetup(pwmLChannel, frequency, pwmResolution);
00113
00114     motorAttachPin(rPWM_Pin, pwmRChannel);
00115     motorAttachPin(lPWM_Pin, pwmLChannel);
00116
00117     motorPinMode(r_EN_Pin, OUTPUT);
00118     motorPinMode(l_EN_Pin, OUTPUT);
00119
00120     motorPinWrite(r_EN_Pin, HIGH);
00121     motorPinWrite(l_EN_Pin, HIGH);
00122
00123     ledcWrite(pwmRChannel, 0);
00124     ledcWrite(pwmLChannel, 0);
00125
00126     distanceSensor.attachSingleEdge(
00127       static_cast<int>(hall_1_Pin),
00128       static_cast<int>(hall_2_Pin));
00129     distanceSensor.clearCount();
00130     READ_POSITION_ENCODER()
00131
00132     if (debugEnabled) {
00133       Serial.printf("Motor: %s\n"
00134                     "--------------------\n"
00135                     "Frequency:    %5d\n"
00136                     "Resolution:   %5d\n"
00137                     "Speed:        %5d\n"
00138                     "Position:     %5d\n"
00139                     "RPWM Pin:     %5d\n"
00140                     "LPWM Pin:     %5d\n"
00141                     "Hall 1 Pin:   %5d\n"
00142                     "Hall 2 Pin:   %5d\n"
00143                     "Max Position: %5d\n\n",
00144                     id,
00145                     frequency,
00146                     pwmResolution,
00147                     speed,
00148                     pos,
00149                     rPWM_Pin,
00150                     lPWM_Pin,
00151                     hall_1_Pin,
00152                     hall_2_Pin,
00153                     totalPulseCount);
00154     }
00155   }
00156
00157   void drive(const Direction motorDirection, const int specifiedSpeed = 0) {
00158     const int driveSpeed = specifiedSpeed > 0 ? specifiedSpeed : speed;
00159
00160     switch (motorDirection) {
00161       case Direction::EXTEND:
00162         motorPinWrite(r_EN_Pin, HIGH);
00163         motorPinWrite(l_EN_Pin, HIGH);
00164         ledcWrite(pwmRChannel, driveSpeed);
00165         ledcWrite(pwmLChannel, 0);
00166         break;
00167       case Direction::STOP:
00168         motorPinWrite(r_EN_Pin, HIGH);
00169         motorPinWrite(l_EN_Pin, HIGH);
00170         ledcWrite(pwmRChannel, 0);
00171         ledcWrite(pwmLChannel, 0);
00172         motorPinWrite(r_EN_Pin, LOW);
00173         motorPinWrite(l_EN_Pin, LOW);
00174         break;
00175       case Direction::RETRACT:
00176         motorPinWrite(r_EN_Pin, HIGH);
```

```
00177         motorPinWrite(l_EN_Pin, HIGH);
00178         ledcWrite(pwmRChannel, 0);
00179         ledcWrite(pwmLChannel, driveSpeed);
00180         break;
00181       default:
00182         break;
00183     } // end direction handler
00184 
00185     lastPos = pos;
00186     READ_POSITION_ENCODER()
00187   } // end drive
00188 
00190   void extend() {
00191     // Works as a toggle
00192     dir = (dir != Direction::EXTEND) ? Direction::EXTEND : Direction::STOP;
00193   }
00194 
00196   void retract() {
00197     // Works as a toggle
00198     dir = (dir != Direction::RETRACT) ? Direction::RETRACT : Direction::STOP;
00199   }
00200 
00202   void stop() {
00203     // Works as a toggle
00204     dir = Direction::STOP;
00205   }
00206 
00208   void zero() {
00209     distanceSensor.clearCount();
00210     lastPos = pos = 0;
00211   }
00212 
00215   void home() {
00216     // First retract as much as possible
00217 
00218     int sameCount = 0;
00219     int firstSameTime = 0;
00220     dir = Direction::RETRACT;
00221     while (sameCount < 1000) {
00222       drive(dir, MAX_SPEED);
00223       if (lastPos == pos) {
00224         if (sameCount == 0) {
00225           firstSameTime = millis();
00226         } else {
00227           if (millis() - firstSameTime > 1000)
00228             break;
00229         }
00230         sameCount++;
00231       } else {
00232         sameCount = 0;
00233       }
00234       READ_POSITION_ENCODER()
00235     }
00236 
00237     Serial.println("Fully retracted");
00238 
00239     sameCount = 0;
00240     firstSameTime = 0;
00241     dir = Direction::EXTEND;
00242     while (sameCount < 1000) {
00243       drive(dir, MAX_SPEED);
00244       if (lastPos == pos) {
00245         if (sameCount == 0) {
00246           firstSameTime = millis();
00247         } else {
00248           if (millis() - firstSameTime > 1000)
00249             break;
00250         }
00251         sameCount++;
00252       } else {
00253         sameCount = 0;
00254       }
00255       READ_POSITION_ENCODER()
00256     }
00257 
00258     Serial.print("Fully extended. Max pulse: ");
00259     Serial.println(pos);
00260     maxPulses = pos;
00261 
00262     sameCount = 0;
00263     firstSameTime = 0;
00264     dir = Direction::RETRACT;
00265     while (sameCount < 1000) {
00266       drive(dir, MAX_SPEED);
00267       if (lastPos == pos) {
00268         if (sameCount == 0) {
00269           firstSameTime = millis();
```

```
00270         } else {
00271           if (millis() - firstSameTime > 1000)
00272             break;
00273         }
00274         sameCount++;
00275       } else {
00276         sameCount = 0;
00277       }
00278       READ_POSITION_ENCODER()
00279     }
00280
00281     Serial.println("Fully retracted");
00282     dir = Direction::STOP;
00283   }
00284
00286   void update(const int newSpeed = (MAX_SPEED + 1)) {
00287     if (desiredPos >= 0) {
00288       if (pos > desiredPos) {
00289         dir = Direction::RETRACT;
00290       } else if (pos < desiredPos) {
00291         dir = Direction::EXTEND;
00292       } else {
00293         dir = Direction::STOP;
00294         desiredPos = -1;
00295         displayInfo();
00296       }
00297     }
00298
00299     // For lift column - Extension limit
00300     if (dir == Direction::EXTEND && (pos) > totalPulseCount) {
00301       dir = Direction::STOP;
00302       return;
00303     }
00304
00305     // For lift column - Retraction limit
00306     if (dir == Direction::RETRACT && (pos) < 50) {
00307       dir = Direction::STOP;
00308       return;
00309     }
00310
00311     if (newSpeed > MAX_SPEED || newSpeed < 0) {
00312       drive(dir, this->speed);
00313     } else {
00314       drive(dir, newSpeed);
00315     }
00316   }
00317
00320   void setPos(const int newPos) {
00321     READ_POSITION_ENCODER()
00322     MOVE_TO_POS(newPos, 15, 40)
00323   }
00324
00326   void readPos() { READ_POSITION_ENCODER() }
00327
00332   float getNormalizedPos() const {
00333     return (float)this->pos / (float)this->totalPulseCount;
00334   }
00335
00336   void displayInfo() {
00337     Serial.printf("Motor %s - Direction: %s, pos: %d\n", id, directions[static_cast<int>(dir)],
00338                   pos);
00339     Serial.printf("Motor %s - Speed: %d, desired pos: %d\n", id, speed,
00340                   desiredPos);
00341     Serial.printf("Motor %s - Max hall position: %d \n\n", id, totalPulseCount);
00342   }
00343
00347   int getCurrent() const {
00348     const int leftCurrent = motorAnalogRead(l_is_pin);
00349     const int rightCurrent = motorAnalogRead(r_is_pin);
00350
00351     return max(leftCurrent, rightCurrent);
00352   }
00353
00354   void setSpeed(int newSpeed) { speed = newSpeed; }
00355 }; // end class Motor
00356
00357 #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.

```
00001
00003 #ifndef _MOTOR_CONTROLLER_HPP_
00004 #define _MOTOR_CONTROLLER_HPP_
00005
00006 #include <Preferences.h>
00007
00008 #include "Motor.hpp"
00009 #include "PIDController.hpp"
00010 #include "PinMacros.hpp"
00011 #include "defs.hpp"
00012
00013 #define NUMBER_OF_MOTORS 2
00014
00015 #define ALL_MOTORS(operation)                                              \
00016   for (int motor = 0; motor < NUMBER_OF_MOTORS; motor++) {                 \
00017     operation                                                              \
00018   }
00019
00020 #define ALL_MOTORS_COMMAND(command) ALL_MOTORS(motors[motor].command();)
00021
00022 #define RESET_SOFT_MOVEMENT                                                \
00023   pwmUpdateAmount = 0;                                                     \
00024   lastPWMUpdate = -1;                                                      \
00025   softStart = -1;                                                          \
00026   targetSpeed = -1;                                                        \
00027   eIntegral = 0.0f;
00028
00029 #define RESTORE_POSITION(slot) motor_controller.setPos(savedPositions[slot]);
00030
00031 #define SERIAL_SAVE_POSITION(slot)                                         \
00032   if (Serial.available() > 0) {                                            \
00033     int new_pos = Serial.parseInt();                                       \
00034     motor_controller.savePosition(slot, new_pos);                          \
00035   }
00036
00037
00047 class MotorController {
00048 private:
00050   Motor motors[NUMBER_OF_MOTORS];
00051
00054   enum MotorRoles {
00055     LEADER,
00056     FOLLOWER
00057   };
00058
00059   // const int motorPulseTotals[2] = {8060, 8057};
00060
00062   const int motorPulseTotals[2] = {2055, 2050};
00063
00066   int laggingIndex = 0;
00067
00070   int leadingIndex = 0;
00071
00073   int softStart = -1;
00074
00076   int lastPWMUpdate = -1;
00077
00079   int targetSpeed = -1;
00080
00082   float pwmUpdateAmount =
00083       -1.0f;
00084
00086   int lastPrintTime = -1;
00087
```

```
00089   const int printDelta = 333000;
00090
00091   // PIDController pidController;
00092
00094   Direction systemDirection =
00095       Direction::STOP;
00096
00098   Direction requestedDirection =
00099       Direction::STOP;
00100
00102   int pwmFrequency =
00103       PWM_FREQUENCY;
00104
00106   int pwmResolution = PWM_RESOLUTION_BITS;
00107
00108   int initialCurrentReadings[NUMBER_OF_MOTORS] = {0, 0};
00109
00110   Preferences positionStorage; //
00111
00113   void loadPositions() {
00114     for (int slot = 0; slot < NUM_POSITION_SLOTS; slot++) {
00115       savedPositions[slot] =
00116           positionStorage.getInt(save_position_slot_names[slot]);
00117     }
00118   }
00119
00121   void immediateHalt() {
00122     speed = targetSpeed = 0;
00123     systemDirection = Direction::STOP;
00124     requestedDirection = Direction::STOP;
00125     RESET_SOFT_MOVEMENT
00126
00127     ALL_MOTORS(motors[motor].speed = 0;)
00128   }
00129
00131   void initializeMotors() {
00132     ALL_MOTORS_COMMAND(initialize)
00133     immediateHalt();
00134   }
00135
00136 public:
00138   int K_p = 100000;
00139
00141   float K_i = 0.1f;
00142
00144   float eIntegral = 0.0f;
00145
00148   int defaultSpeed = DEFAULT_MOTOR_SPEED;
00149
00151   int speed = 0;
00152
00155   int currentIncreaseTolerance =
00156       DEFAULT_CURRENT_INCREASE_LIMIT;
00157
00164   MotorController(
00165       const int pwmFrequency = PWM_FREQUENCY,
00166       const int pwmResolution = PWM_RESOLUTION_BITS,
00167       const int defaultSpeed = DEFAULT_MOTOR_SPEED,
00168       const int currentIncreaseLimit = DEFAULT_CURRENT_INCREASE_LIMIT)
00169       : pwmFrequency(pwmFrequency),
00170         pwmResolution(pwmResolution),
00171         defaultSpeed(defaultSpeed),
00172         currentIncreaseTolerance(currentIncreaseLimit) {
00173     if (debugEnabled) {
00174       char buf[256];
00175       sprintf(
00176           buf,
00177           "Controller Params: Frequency: %d - Resolution: %d - Duty Cycle: %d\n",
00178           pwmFrequency, pwmResolution, defaultSpeed);
00179       Serial.println(buf);
00180     }
00181   }
00182
00185   void initialize() {
00186     // Read in saved positions
00187     // Open in read-write mode
00188     motors[0] =
00189         Motor("Leader",
00190               MotorPin::MOTOR1_RPWM_PIN,
00191               MotorPin::MOTOR1_LPWM_PIN,
00192               MotorPin::MOTOR1_R_EN_PIN,
00193               MotorPin::MOTOR1_L_EN_PIN,
00194               MotorPin::MOTOR1_HALL1_PIN,
00195               MotorPin::MOTOR1_HALL2_PIN,
00196               MotorPin::MOTOR1_LIS_PIN,
00197               MotorPin::MOTOR1_RIS_PIN,
00198               motorPulseTotals[0],
```

```
00199               PWM_FREQUENCY,
00200               defaultSpeed,
00201               pwmResolution);
00202
00203     motors[1] =
00204         Motor("Follower",
00205               MotorPin::MOTOR2_RPWM_PIN,
00206               MotorPin::MOTOR2_LPWM_PIN,
00207               MotorPin::MOTOR2_R_EN_PIN,
00208               MotorPin::MOTOR2_L_EN_PIN,
00209               MotorPin::MOTOR2_HALL1_PIN,
00210               MotorPin::MOTOR2_HALL2_PIN,
00211               MotorPin::MOTOR2_LIS_PIN,
00212               MotorPin::MOTOR2_RIS_PIN,
00213               motorPulseTotals[1],
00214               PWM_FREQUENCY,
00215               defaultSpeed,
00216               pwmResolution);
00217
00218     positionStorage.begin("evox-tilt", false);
00219     loadPositions();
00220     initializeMotors();
00221     Serial.println("System initialized.");
00222
00223     ALL_MOTORS(initialCurrentReadings[motor] = motors[motor].getCurrent();)
00224   }
00225
00227   void extend() {
00228     // SET_TO_ANALOG_PIN_FUNC(SPEED_POT_PIN, this->setSpeed, 0, 2 «
00229     // PWM_RESOLUTION_BITS - 1);
00230     setSpeed(defaultSpeed);
00231     ALL_MOTORS_COMMAND(extend)
00232     systemDirection = Direction::EXTEND;
00233     requestedDirection = Direction::EXTEND;
00234   }
00235
00237   void retract() {
00238     // SET_TO_ANALOG_PIN_FUNC(SPEED_POT_PIN, this->setSpeed, 0, 2 «
00239     // PWM_RESOLUTION_BITS - 1);
00240     setSpeed(defaultSpeed);
00241     ALL_MOTORS_COMMAND(retract)
00242     systemDirection = Direction::RETRACT;
00243     requestedDirection = Direction::RETRACT;
00244   }
00245
00247   void stop() {
00248     RESET_SOFT_MOVEMENT
00249
00250     setSpeed(0);
00251     requestedDirection = Direction::STOP;
00252   }
00253
00255   void home() { ALL_MOTORS_COMMAND(home) }
00256
00258   void zero() { ALL_MOTORS_COMMAND(zero) }
00259
00262   void setSpeed(int newSpeed) {
00263     targetSpeed = newSpeed;
00264     softStart = lastPWMUpdate = micros();
00265
00266     // Calculate the difference between the current speed and the requested
00267     // speed and divide that difference by the number of update steps to get
00268     // the PWM duty cycle increase/decrease per step.
00269     //
00270     // This will usually have a fractional part, so we make it a float value. We
00271     // handle the rounding and conversion to an integer in the update method.
00272     pwmUpdateAmount =
00273         ceil((float)abs(targetSpeed - speed) / SOFT_MOVEMENT_UPDATE_STEPS);
00274
00275     // If the new speed is lower, make it negative, as we add the
00276     // pwmUpdateAmount to the speed
00277     if (targetSpeed < speed) {
00278       pwmUpdateAmount = -pwmUpdateAmount;
00279     }
00280
00281     if (debugEnabled) {
00282       Serial.printf("MotorController\n"
00283                     "------------\n"
00284                     "setSpeed(%d)\n"
00285                     "speed: %3d\n"
00286                     "target speed: %3d\n"
00287                     "pwmUpdateAmount: %3.6f\n\n",
00288                     newSpeed,
00289                     speed,
00290                     targetSpeed,
00291                     pwmUpdateAmount);
00292     }
```

```
00293   }
00294
00297   int getSpeed() const {
00298     // Return the average
00299     return (motors[0].speed + motors[1].speed) / 2;
00300   }
00301
00304   bool countsAreUnequal(void) const {
00305     bool areUnequal = true;
00306     ALL_MOTORS(areUnequal &= motors[motor].pos == motors[motor].lastPos;)
00307     return areUnequal;
00308   }
00309
00311   void report() {
00312     Serial.printf("MotorController\n--------------------\nSpeed: %d\nTarget "
00313                   "Speed: %d\nK_p: %d\nK_i: %f\neIntegral:%f\npwmUpdateAmont: %f \n",
00314                   speed, targetSpeed, K_p, K_i, eIntegral, pwmUpdateAmount);
00315     Serial.print("Leading motor: ");
00316     Serial.println(motor_roles[leadingIndex]);
00317     Serial.print("Lagging motor: ");
00318     Serial.println(motor_roles[laggingIndex]);
00319     Serial.printf("\n\n\n");
00320
00321     ALL_MOTORS_COMMAND(displayInfo)
00322   }
00323
00324   /*
00325   void pidReport(const float deltaT) const {
00326     int leaderPos = motors[0].pos;
00327     int followerPos = motors[1].pos;
00328     int followerSpeed = motors[1].speed;
00329     Direction dir = motors[1].dir;
00330
00331     pidController.report(leaderPos, followerPos, deltaT, followerSpeed, dir);
00332   }
00333
00334   */
00335
00340   void savePosition(const int slot, const int position_value) {
00341     if (slot > 0 && slot < NUM_POSITION_SLOTS && position_value > -1) {
00342       ALL_MOTORS(motors[motor].setPos(position_value);)
00343       savedPositions[slot - 1] = position_value;
00344       positionStorage.putInt(save_position_slot_names[slot], position_value);
00345     }
00346   }
00347
00350   void setPos(const int newPos) { ALL_MOTORS(motors[motor].setPos(newPos);) }
00351
00354   bool isStopped() const { return systemDirection == Direction::STOP; }
00355
00358   void update(const float deltaT = 0.0f) {
00359     // Only update PID if we're not stopped
00360     if (Direction::STOP != systemDirection) {
00361       ALL_MOTORS_COMMAND(readPos)
00362
00363       if (Direction::EXTEND == systemDirection) {
00364         if (motors[LEADER].getNormalizedPos() <
00365             motors[FOLLOWER].getNormalizedPos()) {
00366           laggingIndex = LEADER;
00367           leadingIndex = FOLLOWER;
00368         } else {
00369           laggingIndex = FOLLOWER;
00370           leadingIndex = LEADER;
00371         }
00372       } else {
00373         if (motors[LEADER].getNormalizedPos() >
00374             motors[FOLLOWER].getNormalizedPos()) {
00375           laggingIndex = LEADER;
00376           leadingIndex = FOLLOWER;
00377         } else {
00378           laggingIndex = FOLLOWER;
00379           leadingIndex = LEADER;
00380         }
00381       }
00382     }
00383
00384     // Difference between current speed and target speed
00385     const int speedDelta = abs(speed - targetSpeed);
00386
00387     // Get current time
00388     const int currentTime = micros();
00389
00390     // Time since soft move has started
00391     const int moveTimeDelta = currentTime - softStart;
00392
00393     // Time since PWM duty cycle (speed) was last modified
00394     const int updateTimeDelta = currentTime - lastPWMUpdate;
```

```
00395
00396     // If we have a target speed
00397     if (targetSpeed >= 0) {
00398       // Are we ready to update?
00399       if (updateTimeDelta >= SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS) {
00400         // If so, first check if the distance to target is less than our step
00401         // amount. Or if the time to ramp up has expired
00402         if (speedDelta < abs(pwmUpdateAmount) ||
00403             moveTimeDelta >= SOFT_MOVEMENT_MICROS) {
00404           // If so, then set the current speed to target speed and reset state
00405           speed = targetSpeed;
00406           RESET_SOFT_MOVEMENT
00407
00408           // If the requested direction is to stop, the finally update the system
00409           // direction to stop
00410           if (requestedDirection == Direction::STOP) {
00411             systemDirection = Direction::STOP;
00412             if (debugEnabled) {
00413               Serial.println("System Direction: STOP");
00414               const double timeSinceSoftStart = (double) (micros() - softStart) / (double)
      MICROS_IN_MS;
00415
00416               Serial.printf("Soft Movement PWM Update - "
00417                       "speed: %d - "
00418                       "target speed: %d - "
00419                       "time since soft start: %f ms - "
00420                       "pwmUpdateAmount: %f\n",
00421                   speed,
00422                   targetSpeed,
00423                   timeSinceSoftStart,
00424                   pwmUpdateAmount);
00425             }
00426
00427             // Tell all motors to stop
00428             immediateHalt();
00429           }
00430         }
00431
00432         // Otherwise, add the PWM update ammount to the current duty cycle
00433         // (speed) Round the result and then convert to an integer
00434         const float newSpeed = (float)speed + pwmUpdateAmount;
00435         speed = (int)floorf(newSpeed);
00436
00437         // We just updated, so update last update timestamp
00438         lastPWMUpdate = micros();
00439
00440         const double timeSinceSoftStart = (double) (micros() - softStart) / (double) MICROS_IN_MS;
00441
00442         if (debugEnabled) {
00443           Serial.printf("Soft Movement PWM Update - "
00444                       "speed: %d - "
00445                       "target speed: %d - "
00446                       "time since soft start: %f ms - "
00447                       "pwmUpdateAmount: %f\n",
00448           speed,
00449           targetSpeed,
00450           timeSinceSoftStart,
00451           pwmUpdateAmount);
00452         }
00453       }
00454     }
00455
00456     /*
00457 int maxCurrent = max(motors[leadingIndex].getCurrent(),
00458                     motors[laggingIndex].getCurrent());
00459 */
00460
00461     // The error is the difference between the normalization of positions of
00462     // the two motors
00463     const float error = abs(motors[laggingIndex].getNormalizedPos() -
00464                             motors[leadingIndex].getNormalizedPos());
00465
00466     // Add the error from this timestep to the integral term
00467     eIntegral += error * deltaT;
00468
00469     // Calculate the adjusted speed to set to the faster motor
00470     //const int adjustedSpeed = speed - int((error * K_p) + (K_i * eIntegral));
00471     const int adjustedSpeed = speed - int((error * K_p));
00472
00473     // Slow down the faster motor, keeping the value in the acceptable range
00474     motors[leadingIndex].speed =
00475         constrain(adjustedSpeed, 0, 2 << pwmResolution);
00476     motors[laggingIndex].speed = speed;
00477
00478     // Update the motors with the adjustments
00479     ALL_MOTORS_COMMAND(update)
00480   }
```

```
00481 };
00482
00483 #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.

```
00001
00003 #ifndef _MOTOR_PINS_HPP_
00004 #define _MOTOR_PINS_HPP_
00005
00006 #include <cstdint>
00007
00020 enum class MotorPin : std::uint8_t {
00022   UNASSIGNED = 0,
00023
00025   MOTOR1_RPWM_PIN = 25,
00026
00028   MOTOR1_LPWM_PIN = 19,
00029
00031   MOTOR1_R_EN_PIN = 26,
00032
00034   MOTOR1_L_EN_PIN = 18,
00035
00037   MOTOR1_HALL1_PIN = 22,
00038
00040   MOTOR1_HALL2_PIN = 23,
00041
00043   MOTOR1_LIS_PIN = 22,
00044
00046   MOTOR1_RIS_PIN = 23,
00047
00049   MOTOR2_RPWM_PIN = 5,
00050
00052   MOTOR2_LPWM_PIN = 17,
00053
00055   MOTOR2_R_EN_PIN = 16,
00056
00058   MOTOR2_L_EN_PIN = 15,
00059
00061   MOTOR2_HALL1_PIN = 14,
00062
00064   MOTOR2_HALL2_PIN = 13,
00065
00067   MOTOR2_LIS_PIN = 29,
00068
00070   MOTOR2_RIS_PIN = 30,
00071 };
00072
00073 #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.

```
00001
00003 #ifndef _PID_CONTROLLER_HPP__
00004 #define _PID_CONTROLLER_HPP__
00005
00006 #include "defs.hpp"
00007 #include <math.h>
```

```
00008 #include <cstring>
00009
00010 class PIDController {
00011 private:
00012   float kp;               // the controller path proportional gain
00013   float ti;               // the controller's integrator time constant
00014   float td;               // the controller's derivative time constant
00015   float uMax;             // Maximum magnitude of control signal
00016   float ePrev, eIntegral; // Storage
00017
00018 public:
00019   PIDController(float kp = 0.1,
00020                 float ti = 0.002,
00021                 float td = 0.01,
00022                 float uMax = 255.0)
00023       : kp(kp), ti(ti), td(td), uMax(uMax), ePrev(0.0), eIntegral(0.0) {}
00024
00025   // A function to set the parameters
00026   void setParams(float kpIn, float kdIn, float kiIn, float uMaxIn = 255.0) {
00027     kp = kpIn;
00028     td = kdIn;
00029     ti = kiIn;
00030     uMax = uMaxIn;
00031   }
00032
00033   // A function to compute the control signal
00034   void evaluate(int value, int target, float deltaT, int& speed, Direction& dir) {
00035      // error
00036     int e = target - value;
00037
00038     // derivative
00039     float dedt = (e - ePrev) / (deltaT);
00040
00041     // integral
00042     eIntegral = eIntegral + e * deltaT;
00043
00044     // control signal
00045     float u = kp * e + td * dedt + ti * eIntegral;
00046
00047     // motor power
00048     speed = (int)fabs(u);
00049     if (speed > uMax) {
00050       speed = uMax;
00051     }
00052
00053     // motor direction
00054     dir = Direction::EXTEND;
00055     if (u < 0) {
00056       dir = Direction::RETRACT;
00057     } else if (u == 0) {
00058        dir = Direction::STOP;
00059     }
00060
00061
00062     // store previous error
00063     ePrev = e;
00064   }
00065
00066   void report(const int value, const int target, const float deltaT, const int speed, const Direction
      dir) const {
00067     char buf[256]; // Diagnostic messages
00068     sprintf(buf, "PID params: value:%d target: %d, deltaT: %f, speed: %d, dir: %s", value, target,
      deltaT, speed, directions[static_cast<int>(dir)]);
00069     Serial.println(buf);
00070   }
00071 };
00072
00073 #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(analog↩ Read(pin), 0, 4096, range_min, range_max)
- #define **SET_TO_ANALOG_PIN**(pin, var_to_set, range_min, range_max) var_to_set = map(analog↩ Read(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.

```
00001
00002 #ifndef _PIN_MACROS_HPP_
00003 #define _PIN_MACROS_HPP_
00004
00012 float fmap(float x, float in_min, float in_max, float out_min, float out_max) {
00013   return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
00014 }
```

```
00015
00016 #define FSET_TO_ANALOG_PIN(pin, var_to_set, range_min, range_max) \
00017     var_to_set = fmap(analogRead(pin), 0, 4096, range_min, range_max)
00018
00019 #define SET_TO_ANALOG_PIN(pin, var_to_set, range_min, range_max) \
00020     var_to_set = map(analogRead(pin), 0, 4096, range_min, range_max)
00021
00022 #define SET_TO_ANALOG_PIN_FUNC(pin, func, range_min, range_max) \
00023 func(map(analogRead(pin), 0, 4096, range_min, range_max))
00024
00025 #define motorPinWrite(pin, value) \
00026   digitalWrite(static_cast<std::uint8_t>(pin), value)
00027
00028 #define motorPinWrite(pin, value) \
00029   digitalWrite(static_cast<std::uint8_t>(pin), value)
00030
00031 #define motorPinMode(pin, value) \
00032   pinMode(static_cast<std::uint8_t>(pin), value)
00033
00034 #define motorAttachPin(pin, channel) \
00035   ledcAttachPin(static_cast<std::uint8_t>(pin), channel)
00036
00037 #define motorAnalogRead(pin) \
00038   analogRead(static_cast<std::uint8_t>(pin))
00039
00040 #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.

```
00001
00003 #ifndef _POTENTIOMTER_PINS_HPP_
00004 #define _POTENTIOMTER_PINS_HPP_
```

```
00005
00006 #include <cstdint>
00007
00019 enum class PotentiometerPins : std::uint8_t {
00021   SPEED_POT_PIN = 35,
00022
00024   KP_POT_PIN = 32,
00025 };
00026
00027 #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.

```
00001
00003 #ifndef _ROUTE_MACROS_HPP__
00004 #define _ROUTE_MACROS_HPP__
00005
00006 #define SET_TILT(n)                                                 \
00007   if (request->hasParam(PARAM_INPUT_1)) {                           \
00008     inputMessage1 = request->getParam(PARAM_INPUT_1)->value();      \
00009     const int new_pos = inputMessage1.toInt();                            \
00010     motor_controller.savePosition(n, new_pos);                      \
00011   } else {                                                          \
00012     inputMessage1 = "Error: No position sent.";                     \
00013   }
00014
00015 #define DEF_HANDLER(func) [](AsyncWebServerRequest *request) { func }
00016
00017 #define LOAD_SAVED_POSITION(position, response_text)                \
00018   motor_controller.setPos(savedPositions[position]);                \
00019   request->send(200, "text/plain", response_text);
00020
00021 #define MOTOR_COMMAND(command, response_text)                       \
00022   motor_controller.command();                                       \
00023   request->send(200, "text/plain", response_text);
00024
00025 #define SET_POS_HANDLER(slot)                                       \
00026   String inputMessage1;                                             \
00027   SET_TILT(slot)                                                    \
00028   request->send(200, "text/plain", inputMessage1);
00029
00030 #define STATIC_FILE(filename, file_type)                            \
00031   request->send(SPIFFS, filename, file_type);
00032
00033 #endif // _ROUTE_MACROS_HPP__
```

Index