Motor Control Firmware

Generated by Doxygen 1.9.8

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 ControlPin Class Reference

Pin number definitions for the controlled parameters.

`#include <ControlPins.hpp>`

3.2.1 Detailed Description

Pin number definitions for the controlled parameters.

Author

Terry Paul Ferguson
terry@terryferguson.us

Version

0.1

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

- ControlPins.hpp

3.3 CurrentSense Class Reference

This implements the current sense functionality for the motors.

`#include <CurrentSense.hpp>`

Public Member Functions

- **CurrentSense** (const ControlPin pCurrentSensePin=ControlPin::CURRENT_SENSE_PIN, const double p↩ LogicVoltage=5.0, const int32_t pMaxAdcValue=MAX_ADC_VALUE)
- void **initialize** ()
- int getCurrent () const

3.3.1 Detailed Description

This implements the current sense functionality for the motors.

Author

Terry Paul Ferguson
terry@terryferguson.us

Version

0.1

3.3.2 Member Function Documentation

3.3.2.1 getCurrent()

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

Calculates the average current.

Returns

the average current of the sampling

Exceptions

None

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

- CurrentSense.hpp

3.4 Direction Class Reference

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

`#include <Direction.hpp>`

3.4.1 Detailed Description

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

Author

Terry Paul Ferguson
terry@terryferguson.us

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

- Direction.hpp

3.5 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
- void **setSpeed** (int newSpeed)

Public Attributes

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

3.5.1 Detailed Description

This class represents the motor controlled by the microcontroller.

Author

Terry Paul Ferguson

Version

0.1

3.5.2 Constructor & Destructor Documentation

3.5.2.1 Motor() [1/2]

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

The direction of the motor rotation

3.5.2.2 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

3.5.3 Member Function Documentation

3.5.3.1 displayInfo()

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

Displays information about the motor.

Parameters

None	

Returns

None

Exceptions

None	

3.5.3.2 drive()

```
void Motor::drive (
            const Direction motorDirection,
            const int specifiedSpeed = 0 )  [inline]
```

Drives the motor in the specified direction at the specified speed.

Parameters

motorDirection	the direction in which the motor should be driven
specifiedSpeed	the specified speed at which the motor should be driven (default: 0)

3.5.3.3 getCurrent()

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

Returns

The larger of the two current values used by the motor

3.5.3.4 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

3.5.3.5 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

3.5.4 Member Data Documentation

3.5.4.1 lastPos

```
int Motor::lastPos
```

Initial value:

```
=
      0
```

The current position of the motor based on hall sensor pulses

3.5.4.2 maxPulses

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

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

3.5.4.3 pos

```
int Motor::pos
```

Initial value:

```
=
      0
```

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

3.5.4.4 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

3.6 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 = 50000
- float K_i = 0.1f
- float eIntegral = 0.0f
- int defaultSpeed = DEFAULT_MOTOR_SPEED
- int speed = 0
- int leftCurrent = 0
- int rightCurrent = 0
- int currentIncreaseTolerance = DEFAULT_CURRENT_INCREASE_LIMIT
- Motor **motors** [NUMBER_OF_MOTORS]

  The motors controlled by this motor controller instance.
- Direction **systemDirection** = Direction::STOP

  The current system level direction indicator.

3.6.1 Detailed Description

This is the controller of the motors.

Author

Terry Paul Ferguson
terry@terryferguson.us

Version

0.1

3.6.2 Constructor & Destructor Documentation

3.6.2.1 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

3.6.3 Member Function Documentation

3.6.3.1 countsAreUnequal()

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

Indicates whether the motor counts are unequal.

Returns

True if the motor counts are different, false otherwise

3.6.3.2 getSpeed()

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

Get the system speed.

Returns

The average speed of the system

3.6.3.3 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

3.6.3.4 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

3.6.3.5 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

3.6.3.6 setSpeed()

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

Smoothly change to the newly requested speed.

Parameters

newSpeed	The new speed to target

3.6.3.7 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

3.6.4 Member Data Documentation

3.6.4.1 currentIncreaseTolerance

```
int MotorController::currentIncreaseTolerance = DEFAULT_CURRENT_INCREASE_LIMIT
```

Maxim1um current increase limit for motor cutoff

3.6.4.2 defaultSpeed

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

The default speed to operate the motors at on startup

3.6.4.3 eIntegral

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

The integral error coefficient for the PID controller

3.6.4.4 K_i

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

The intagral gain for the PID controller

3.6.4.5 K_p

```
int MotorController::K_p = 50000
```

The proprotional gain for the PID controller

3.6.4.6 leftCurrent

```
int MotorController::leftCurrent = 0
```

Left motor current

3.6.4.7 rightCurrent

```
int MotorController::rightCurrent = 0
```

Right motor current

3.6.4.8 speed

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

Current target speed

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

- MotorController.hpp

3.7 MotorPins Class Reference

Pin number definitions for the motor.

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

3.7.1 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

3.8 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

3.8.1 Member Function Documentation

3.8.1.1 evaluate()

```
void PIDController::evaluate (
            int value,
            int target,
            float deltaT,
            int & speed,
            Direction & dir )  [inline]
```

A function to compute the control signal

Parameters

value	The current value
target	The target value
deltaT	The time step
speed	The reference to the speed variable
dir	The reference to the direction variable

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 ,
  TOGGLE_PID }

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
GET_TILT_2	Get value from stored position slot 2 - 27
GET_TILT_3	Get value from stored position slot 3 - 28

Enumerator

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
TOGGLE_PID	Command to tell tell the microcontroller to turn off PID control - 33

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
00020 enum class Command : std::uint32_t {
00022   RETRACT  = 17,
00023
00025   EXTEND,
00026
00028   REPORT,
00029
00031   STOP,
00032
00034   SAVE_TILT_1,
00035
00037   SAVE_TILT_2,
00038
00040   SAVE_TILT_3,
00041
00043   SAVE_TILT_4,
00044
00046   SAVE_TILT_5,
00047
00049   GET_TILT_1,
00050
00052   GET_TILT_2,
00053
00055   GET_TILT_3,
00056
00058   GET_TILT_4,
00059
00061   GET_TILT_5,
00062
00064   ZERO,
00065
00067   SYSTEM_RESET,
00068
00070   TOGGLE_PID,
00071 };
00072
00073 #endif // _COMMANDS_HPP_
```

4.3 ControlPins.hpp File Reference

```
#include <cstdint>
```

Enumerations

- enum class ControlPin : std::uint8_t { UNASSIGNED = 255 , LEFT_CURRENT_SENSE_PIN = 35 , RIGHT_CURRENT_SENSE_PIN = 34 }

4.3.1 Enumeration Type Documentation

4.3.1.1 ControlPin

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

Enumerator

UNASSIGNED	NULL pin for unassigned
LEFT_CURRENT_SENSE_PIN	Left current sense pin
RIGHT_CURRENT_SENSE_PIN	Right current sense pin

4.4 ControlPins.hpp

Go to the documentation of this file.

```
00001
00003 #ifndef _CONTROL_PINS_HPP_
00004 #define _CONTROL_PINS_HPP_
00005
00006 #include <cstdint>
00007
00019 enum class ControlPin : std::uint8_t {
00021   UNASSIGNED = 255,
00022
00024   LEFT_CURRENT_SENSE_PIN = 35,
00025
00027   RIGHT_CURRENT_SENSE_PIN = 34,
00028 };
00029
00030 #endif // _CONTROL_PINS_HPP_
```

4.5 CurrentSense.hpp File Reference

```
#include <stdint.h>
#include <driver/adc.h>
#include "defs.hpp"
#include "ControlPins.h"
```

Classes

- class CurrentSense

  This implements the current sense functionality for the motors.

4.6 CurrentSense.hpp

Go to the documentation of this file.

```
00001
00003 #ifndef _CURRENT_SENSE_HPP_
00004 #define _CURRENT_SENSE_HPP_
00005
00006 #include <stdint.h>
00007 #include <driver/adc.h>
```

```
00008
00009 #include "defs.hpp"
00010 #include "ControlPins.h"
00011
00012
00023 class CurrentSense
00024 {
00025 private:
00026   const int_fast32_t CALIBRATE_ITERATIONS_SHIFT = 15;
00027   const int_fast32_t SAMPLE_CURRENT_ITERATIONS_SHIFT = 7;
00028
00029   const int32_t MV_PER_AMP = static_cast<int32_t>(185 * 1.132);
00031
00032   int32_t ACS_OFFSET = 1885;
00033
00036   // negitive current flow.
00037
00038   ControlPin currentSensePin;
00039   double logicVoltage = 5.0;
00040   int32_t maxAdcValue = 4096;
00041
00042 public:
00043   CurrentSense(const ControlPin pCurrentSensePin = ControlPin::CURRENT_SENSE_PIN,
00044                const double pLogicVoltage = 5.0,
00045                const int32_t pMaxAdcValue = MAX_ADC_VALUE)
00046       : currentSensePin(pCurrentSensePin), logicVoltage(pLogicVoltage), maxAdcValue(pMaxAdcValue)
00047   {
00048   }
00049
00050   void initialize()
00051   {
00052     adc1_config_width(ADC_WIDTH_12Bit);
00053     adc1_config_channel_atten(ADC1_CHANNEL_7, ADC_ATTEN_DB_11); // using GPIO 34 wind direction
00054
00055     Serial.print("Pin: ");
00056     Serial.println(static_cast<uint8_t>(currentSensePin));
00057     Serial.print("Logic Voltage: ");
00058     Serial.println(logicVoltage);
00059     Serial.print("Max ADC Value: ");
00060     Serial.println(maxAdcValue);
00061     Serial.print("mV per A: ");
00062     Serial.println(MV_PER_AMP);
00063
00064     const int iterations = 1 « CALIBRATE_ITERATIONS_SHIFT;
00065
00066     int32_t adcSum = 0;
00067     double currentSum = 0;
00068
00069     for (int32_t i = 0; i < iterations; i++)
00070     {
00071       const int adcValue = adc1_get_raw(ADC1_CHANNEL_7);
00072       adcSum += adcValue;
00073     }
00074
00075     ACS_OFFSET = adcSum » CALIBRATE_ITERATIONS_SHIFT;
00076
00077     Serial.printf("ACS Offset: %d\n", ACS_OFFSET);
00078   }
00079
00088   int getCurrent() const
00089   {
00090     const int32_t iterations = 1 « SAMPLE_CURRENT_ITERATIONS_SHIFT;
00091
00092     int32_t currentSum = 0;
00093     for (int i = 0; i < iterations; i++)
00094     {
00095       const int adcOffset = adc1_get_raw(ADC1_CHANNEL_7) - ACS_OFFSET;
00096       // const int adcOffset = analogRead(static_cast<uint8_t>(currentSensePin)) - ACS_OFFSET;
00097       const double voltageDelta = (adcOffset * (logicVoltage / maxAdcValue));
00098       const int current = static_cast<int>(voltageDelta * 1000000.0 / MV_PER_AMP);
00099       currentSum += current;
00100     }
00101
00102     const double averageCurrent = static_cast<double>(currentSum » SAMPLE_CURRENT_ITERATIONS_SHIFT);
00103
00104     return static_cast<int>(averageCurrent);
00105   } // end method getCurrent
00106 };  // end class CurrentSense
00107
00108 #endif // _CURRENT_SENSE_HPP_
```

4.7 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.

4.7.1 Function Documentation

4.7.1.1 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

4.8 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_
```

4.9 defs.hpp File Reference

```
#include "Commands.hpp"
#include "MotorPins.hpp"
#include "ControlPins.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 **ADC_RESOLUTION_BITS** 12
- #define **DEFAULT_MOTOR_SPEED** 192
- #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** (1 << (PWM_RESOLUTION_BITS)-1)
- #define **MIN_SPEED** (1 << (PWM_RESOLUTION_BITS)-1) ∗ -1
- #define **MAX_ADC_VALUE** (1 << (ADC_RESOLUTION_BITS))

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.
- bool **pid_on** = true

4.9.1 Variable Documentation

4.9.1.1 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",
}
```

4.10 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 "ControlPins.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 ADC_RESOLUTION_BITS 12
00033
00034 #define DEFAULT_MOTOR_SPEED 192
00035
00036 #define MICROS_IN_MS 1000
00037
00038 #define SOFT_MOVEMENT_TIME_MS 2000
00039
00040 #define SOFT_MOVEMENT_MICROS (SOFT_MOVEMENT_TIME_MS * MICROS_IN_MS)
00041
00042 #define SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS 20000
00043
00044 #define SOFT_MOVEMENT_UPDATE_STEPS                                          \
00045   (SOFT_MOVEMENT_MICROS / SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS)
00046
00047 #define MAX_SPEED (1 << (PWM_RESOLUTION_BITS)-1)
00048
00049 #define MIN_SPEED (1 << (PWM_RESOLUTION_BITS)-1) * -1
00050
00051 #define MAX_ADC_VALUE (1 << (ADC_RESOLUTION_BITS))
00052
00053 bool pid_on = true;
00054
00055 #endif // _DEFS_HPP_
```

4.11 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.

4.11.1 Enumeration Type Documentation

4.11.1.1 Direction

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

Enumerator

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

4.12 Direction.hpp

Go to the documentation of this file.

```
00001
00003 #ifndef _DIRECTION_HPP_
00004 #define _DIRECTION_HPP_
00005
00017 enum class Direction {
00019   EXTEND = 0,
00020
00022   STOP,
00023
00025   RETRACT
00026 };
00027
00029 const char *directions[3] = {"EXTEND", "STOP", "RETRACT"};
00030
00031 #endif // _DIRECTION_HPP_
```

4.13 Motor.hpp File Reference

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

4.13.1 Macro Definition Documentation

4.13.1.1 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;                                       \
    }                                                                       \
  }
```

4.14 Motor.hpp

Go to the documentation of this file.

```
00001
00003 #ifndef _MOTOR_HPP_
00004 #define _MOTOR_HPP_
00005
00006 #include "PinMacros.hpp"
00007 #include "defs.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 =
00042       MotorPin::UNASSIGNED;
00043   MotorPin r_is_pin =
00044       MotorPin::UNASSIGNED;
00045   int frequency = PWM_FREQUENCY;
00046   int pwmResolution = 8;
00047   int desiredPos =
00048       -1;
00049   int totalPulseCount = 0;
00052   ESP32Encoder distanceSensor;
00055 public:
00056   int pos =
00057       0;
00058   int lastPos =
00059       0;
00060   int speed = 255;
00062   int maxPulses = -1;
00063
00064   Direction dir = Direction::STOP;
00066   Motor() {} // end default constructor
00067
00083   Motor(const char *name, const MotorPin rpwm, const MotorPin lpwm,
```

```
00084         const MotorPin r_en, const MotorPin l_en, const MotorPin hall_1,
00085         const MotorPin hall_2, const MotorPin lIS_pin, const MotorPin rIS_pin,
00086         const int totalPulses, const int freq = PWM_FREQUENCY,
00087         const int defSpeed = 70, const int pwmRes = 8)
00088       : rPWM_Pin(rpwm), lPWM_Pin(lpwm), r_EN_Pin(r_en), l_EN_Pin(l_en),
00089         hall_1_Pin(hall_1), hall_2_Pin(hall_2), l_is_pin(lIS_pin),
00090         r_is_pin(rIS_pin), totalPulseCount(totalPulses), frequency(freq),
00091         speed(defSpeed), pwmResolution(pwmRes) {
00093     strncpy(id, name, sizeof(id) - 1);
00094     id[sizeof(id) - 1] = '\0';
00095   } // end constructor
00096
00098   void initialize() {
00099     // At least two channels are needed for the linear actuator motor
00100     if (currentPWMChannel > -1 && currentPWMChannel < 14) {
00101       pwmRChannel = currentPWMChannel++;
00102       pwmLChannel = currentPWMChannel++;
00103     }
00104
00105     ledcSetup(pwmRChannel, frequency, pwmResolution);
00106     ledcSetup(pwmLChannel, frequency, pwmResolution);
00107
00108     motorAttachPin(rPWM_Pin, pwmRChannel);
00109     Serial.printf("Attaching pin %d to RPWM Channel %d\n", rPWM_Pin,
00110                   pwmRChannel);
00111     motorAttachPin(lPWM_Pin, pwmLChannel);
00112     Serial.printf("Attaching pin %d to LPWM Channel %d\n\n", lPWM_Pin,
00113                   pwmLChannel);
00114
00115     motorPinMode(r_EN_Pin, OUTPUT);
00116     motorPinMode(l_EN_Pin, OUTPUT);
00117
00118     motorPinWrite(r_EN_Pin, HIGH);
00119     motorPinWrite(l_EN_Pin, HIGH);
00120
00121     ledcWrite(pwmRChannel, 0);
00122     ledcWrite(pwmLChannel, 0);
00123
00124     distanceSensor.attachSingleEdge(static_cast<int>(hall_1_Pin),
00125                                     static_cast<int>(hall_2_Pin));
00126     distanceSensor.clearCount();
00127     READ_POSITION_ENCODER()
00128
00129     if (debugEnabled) {
00130       Serial.printf("Motor: %s\n"
00131                     "--------------------\n"
00132                     "Frequency:    %5d\n"
00133                     "Resolution:   %5d\n"
00134                     "Speed:        %5d\n"
00135                     "Position:     %5d\n"
00136                     "RPWM Pin:     %5d\n"
00137                     "LPWM Pin:     %5d\n"
00138                     "Hall 1 Pin:   %5d\n"
00139                     "Hall 2 Pin:   %5d\n"
00140                     "Max Position: %5d\n\n",
00141                     id, frequency, pwmResolution, speed, pos, rPWM_Pin,
00142                     lPWM_Pin, hall_1_Pin, hall_2_Pin, totalPulseCount);
00143       Serial.printf("RPWM Channel %d - LPWM Channel: %d\n\n", pwmRChannel,
00144                     pwmLChannel);
00145     }
00146   }
00147
00156   void drive(const Direction motorDirection, const int specifiedSpeed = 0) {
00157     const int driveSpeed = specifiedSpeed > 0 ? specifiedSpeed : speed;
00158
00159     motorPinWrite(r_EN_Pin, HIGH);
00160     motorPinWrite(l_EN_Pin, HIGH);
00161
00162     switch (motorDirection) {
00163     case Direction::EXTEND:
00164       ledcWrite(pwmRChannel, driveSpeed);
00165       ledcWrite(pwmLChannel, 0);
00166       break;
00167     case Direction::STOP:
00168       ledcWrite(pwmRChannel, 0);
00169       ledcWrite(pwmLChannel, 0);
00170       motorPinWrite(r_EN_Pin, LOW);
00171       motorPinWrite(l_EN_Pin, LOW);
00172       break;
00173     case Direction::RETRACT:
00174       ledcWrite(pwmRChannel, 0);
00175       ledcWrite(pwmLChannel, driveSpeed);
00176       break;
00177     default:
00178       break;
00179     } // end direction handler
00180
```

```
00181     lastPos = pos;
00182     READ_POSITION_ENCODER()
00183   } // end drive
00184
00186   void extend() {
00187     // Works as a toggle
00188     dir = (dir != Direction::EXTEND) ? Direction::EXTEND : Direction::STOP;
00189   }
00190
00192   void retract() {
00193     // Works as a toggle
00194     dir = (dir != Direction::RETRACT) ? Direction::RETRACT : Direction::STOP;
00195   }
00196
00198   void stop() {
00199     // Works as a toggle
00200     dir = Direction::STOP;
00201   }
00202
00204   void zero() {
00205     distanceSensor.clearCount();
00206     lastPos = pos = 0;
00207   }
00208
00211   void home() {
00212     // First retract as much as possible
00213
00214     int sameCount = 0;
00215     int firstSameTime = 0;
00216     dir = Direction::RETRACT;
00217     while (sameCount < 1000) {
00218       drive(dir, MAX_SPEED);
00219       if (lastPos == pos) {
00220         if (sameCount == 0) {
00221           firstSameTime = millis();
00222         } else {
00223           if (millis() - firstSameTime > 1000)
00224             break;
00225         }
00226         sameCount++;
00227       } else {
00228         sameCount = 0;
00229       }
00230       READ_POSITION_ENCODER()
00231     }
00232
00233     Serial.println("Fully retracted");
00234
00235     sameCount = 0;
00236     firstSameTime = 0;
00237     dir = Direction::EXTEND;
00238     while (sameCount < 1000) {
00239       drive(dir, MAX_SPEED);
00240       if (lastPos == pos) {
00241         if (sameCount == 0) {
00242           firstSameTime = millis();
00243         } else {
00244           if (millis() - firstSameTime > 1000)
00245             break;
00246         }
00247         sameCount++;
00248       } else {
00249         sameCount = 0;
00250       }
00251       READ_POSITION_ENCODER()
00252     }
00253
00254     Serial.print("Fully extended. Max pulse: ");
00255     Serial.println(pos);
00256     maxPulses = pos;
00257
00258     sameCount = 0;
00259     firstSameTime = 0;
00260     dir = Direction::RETRACT;
00261     while (sameCount < 1000) {
00262       drive(dir, MAX_SPEED);
00263       if (lastPos == pos) {
00264         if (sameCount == 0) {
00265           firstSameTime = millis();
00266         } else {
00267           if (millis() - firstSameTime > 1000)
00268             break;
00269         }
00270         sameCount++;
00271       } else {
00272         sameCount = 0;
00273       }
```

```
00274       READ_POSITION_ENCODER()
00275     }
00276
00277     Serial.println("Fully retracted");
00278     dir = Direction::STOP;
00279   }
00280
00282   void update(const int newSpeed = (MAX_SPEED + 1)) {
00283     if (desiredPos >= 0) {
00284       if (pos > desiredPos) {
00285         dir = Direction::RETRACT;
00286       } else if (pos < desiredPos) {
00287         dir = Direction::EXTEND;
00288       } else {
00289         dir = Direction::STOP;
00290         desiredPos = -1;
00291         displayInfo();
00292       }
00293     }
00294
00295     /*
00296       // For lift column - Extension limit
00297       if (dir == Direction::EXTEND && (pos) > totalPulseCount) {
00298         dir = Direction::STOP;
00299         return;
00300       }
00301
00302       // For lift column - Retraction limit
00303       if (dir == Direction::RETRACT && (pos) < 50) {
00304         dir = Direction::STOP;
00305         return;
00306       }
00307
00308       */
00309
00310     if (newSpeed > MAX_SPEED || newSpeed < 0) {
00311       drive(dir, this->speed);
00312     } else {
00313       drive(dir, newSpeed);
00314     }
00315   }
00316
00319   void setPos(const int newPos) {
00320     READ_POSITION_ENCODER()
00321     MOVE_TO_POS(newPos, 15, 40)
00322   }
00323
00325   void readPos() { READ_POSITION_ENCODER() }
00326
00331   float getNormalizedPos() const {
00332     if (totalPulseCount == 0)
00333       return 0.0f;
00334     return static_cast<float>(pos) / static_cast<float>(totalPulseCount);
00335   }
00336
00346    void displayInfo() {
00347     Serial.printf("Motor %s - Direction: %s, pos: %d\n", id, directions[static_cast<int>(dir)],
00348                   pos);
00349     Serial.printf("Motor %s - Speed: %d, desired pos: %d\n", id, speed,
00350                   desiredPos);
00351     Serial.printf("Motor %s - Max hall position: %d \n\n", id, totalPulseCount);
00352   }
00353
00355   int getCurrent() const {
00356     const int leftCurrent = motorAnalogRead(l_is_pin);
00357     const int rightCurrent = motorAnalogRead(r_is_pin);
00358
00359     return max(leftCurrent, rightCurrent);
00360   }
00361
00362   void setSpeed(int newSpeed) { speed = newSpeed; }
00363 }; // end class Motor
00364
00365 #endif // _MOTOR_HPP_
```

4.15 MotorController.hpp File Reference

```
#include <Preferences.h>
#include "CurrentSettings.hpp"
#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)

4.15.1 Macro Definition Documentation

4.15.1.1 ALL_MOTORS

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

Value:

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

4.15.1.2 RESET_SOFT_MOVEMENT

```
#define RESET_SOFT_MOVEMENT
```

Value:

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

4.15.1.3 SERIAL_SAVE_POSITION

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

Value:

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

4.16 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 "CurrentSettings.hpp"
00009 #include "Motor.hpp"
00010 #include "PIDController.hpp"
00011 #include "PinMacros.hpp"
00012 #include "defs.hpp"
00013
00014 #define NUMBER_OF_MOTORS 2
00015
00016 #define ALL_MOTORS(operation)                                                  \
00017   for (int motor = 0; motor < NUMBER_OF_MOTORS; motor++) {                     \
00018     operation                                                                  \
00019   }
00020
00021 #define ALL_MOTORS_COMMAND(command) ALL_MOTORS(motors[motor].command();)
00022
00023 #define RESET_SOFT_MOVEMENT                                                    \
00024   pwmUpdateAmount = 0;                                                         \
00025   lastPWMUpdate = -1;                                                          \
00026   softStart = -1;                                                              \
00027   targetSpeed = -1;                                                            \
00028   eIntegral = 0.0f;
00029
00030 #define RESTORE_POSITION(slot) motor_controller.setPos(savedPositions[slot]);
00031
00032 #define SERIAL_SAVE_POSITION(slot)                                             \
00033   if (Serial.available() > 0) {                                                \
00034     int new_pos = Serial.parseInt();                                           \
00035     motor_controller.savePosition(slot, new_pos);                              \
00036   }
00037
00047 class MotorController {
00048 private:
00051   enum MotorRoles {
00052     LEADER,
00053     FOLLOWER
00054   };
00055
00056   const int motorPulseTotals[2] = {8080, 8080};
00057
00059   // const int motorPulseTotals[2] = {2055, 2050};
00060
00063   int laggingIndex = 0;
00064
00067   int leadingIndex = 0;
00068
00070   int softStart = -1;
00071
00073   int lastPWMUpdate = -1;
00074
00076   int targetSpeed = -1;
00077
00079   float pwmUpdateAmount = -1.0f;
00080
00082   int lastPrintTime = -1;
00083
00085   const int printDelta = 500000;
00086
00087   // PIDController pidController;
00088
00090   Direction requestedDirection = Direction::STOP;
00091
00093   int pwmFrequency = PWM_FREQUENCY;
00094
00096   int pwmResolution = PWM_RESOLUTION_BITS;
00097
00098   int initialCurrentReadings[NUMBER_OF_MOTORS] = {0, 0};
00099
00100   Preferences positionStorage; //
00101
00102   void immediateHalt() {
00103     speed = targetSpeed = 0;
00104     systemDirection = Direction::STOP;
00105     requestedDirection = Direction::STOP;
00106     RESET_SOFT_MOVEMENT
00107
00108     ALL_MOTORS(motors[motor].speed = 0;)
```

```
00109   }
00110
00112   void loadPositions() {
00113     for (int slot = 0; slot < NUM_POSITION_SLOTS; slot++) {
00114       savedPositions[slot] =
00115           positionStorage.getInt(save_position_slot_names[slot]);
00116     }
00117   }
00118
00120   void initializeMotors() {
00121     RESET_SOFT_MOVEMENT
00122   ALL_MOTORS_COMMAND(initialize)
00123   immediateHalt();
00124 }
00125
00126 public:
00128 int K_p = 50000;
00129
00131 float K_i = 0.1f;
00132
00134 float eIntegral = 0.0f;
00135
00138 int defaultSpeed = DEFAULT_MOTOR_SPEED;
00139
00141 int speed = 0;
00142
00144 int leftCurrent = 0;
00145
00147 int rightCurrent = 0;
00148
00151 int currentIncreaseTolerance = DEFAULT_CURRENT_INCREASE_LIMIT;
00152
00154 Motor motors[NUMBER_OF_MOTORS];
00155
00157 Direction systemDirection = Direction::STOP;
00158
00165 MotorController(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), pwmResolution(pwmResolution),
00170       defaultSpeed(defaultSpeed),
00171       currentIncreaseTolerance(currentIncreaseLimit) {
00172   char buf[256];
00173   sprintf(
00174       buf,
00175       "Controller Params: Frequency: %d - Resolution: %d - Duty Cycle: %d\n",
00176       pwmFrequency, pwmResolution, defaultSpeed);
00177   Serial.println(buf);
00178   speed = targetSpeed = 0;
00179   Direction systemDirection = Direction::STOP;
00180   ALL_MOTORS(motors[motor].speed = 0;)
00181 }
00182
00185 void initialize() {
00186   // Read in saved positions
00187   // Open in read-write mode
00188   motors[0] =
00189       Motor("Leader", MotorPin::MOTOR1_RPWM_PIN, MotorPin::MOTOR1_LPWM_PIN,
00190             MotorPin::MOTOR1_R_EN_PIN, MotorPin::MOTOR1_L_EN_PIN,
00191             MotorPin::MOTOR1_HALL1_PIN, MotorPin::MOTOR1_HALL2_PIN,
00192             MotorPin::MOTOR1_LIS_PIN, MotorPin::MOTOR1_RIS_PIN,
00193             motorPulseTotals[0], PWM_FREQUENCY, defaultSpeed, pwmResolution);
00194
00195   motors[1] =
00196       Motor("Follower", MotorPin::MOTOR2_RPWM_PIN, MotorPin::MOTOR2_LPWM_PIN,
00197             MotorPin::MOTOR2_R_EN_PIN, MotorPin::MOTOR2_L_EN_PIN,
00198             MotorPin::MOTOR2_HALL1_PIN, MotorPin::MOTOR2_HALL2_PIN,
00199             MotorPin::MOTOR2_LIS_PIN, MotorPin::MOTOR2_RIS_PIN,
00200             motorPulseTotals[1], PWM_FREQUENCY, defaultSpeed, pwmResolution);
00201
00202   positionStorage.begin("evox-tilt", false);
00203   loadPositions();
00204   initializeMotors();
00205   Serial.println("System initialized.");
00206
00207   ALL_MOTORS(initialCurrentReadings[motor] = motors[motor].getCurrent();)
00208 }
00209
00211 void extend() {
00212   // SET_TO_ANALOG_PIN_FUNC(SPEED_POT_PIN, this->setSpeed, 0, 2 «
00213   // PWM_RESOLUTION_BITS - 1);
00214   setSpeed(defaultSpeed);
00215   ALL_MOTORS_COMMAND(extend)
00216   systemDirection = Direction::EXTEND;
00217   requestedDirection = Direction::EXTEND;
00218 }
```

```
00219
00221 void retract() {
00222   // SET_TO_ANALOG_PIN_FUNC(SPEED_POT_PIN, this->setSpeed, 0, 2 «
00223   // PWM_RESOLUTION_BITS - 1);
00224   setSpeed(defaultSpeed);
00225   ALL_MOTORS_COMMAND(retract)
00226   systemDirection = Direction::RETRACT;
00227   requestedDirection = Direction::RETRACT;
00228 }
00229
00231 void stop() {
00232   RESET_SOFT_MOVEMENT
00233
00234   setSpeed(0);
00235   requestedDirection = Direction::STOP;
00236 }
00237
00239 void home() { ALL_MOTORS_COMMAND(home) }
00240
00242 void zero() { ALL_MOTORS_COMMAND(zero) }
00243
00246 void setSpeed(int newSpeed) {
00247   targetSpeed = newSpeed;
00248   softStart = lastPWMUpdate = micros();
00249
00250   // Calculate the difference between the current speed and the requested
00251   // speed and divide that difference by the number of update steps to get
00252   // the PWM duty cycle increase/decrease per step.
00253   //
00254   // This will usually have a fractional part, so we make it a float value. We
00255   // handle the rounding and conversion to an integer in the update method.
00256   pwmUpdateAmount =
00257       ceil((float)abs(targetSpeed - speed) / SOFT_MOVEMENT_UPDATE_STEPS);
00258
00259   // If the new speed is lower, make it negative, as we add the
00260   // pwmUpdateAmount to the speed
00261   if (targetSpeed < speed) {
00262     pwmUpdateAmount = -pwmUpdateAmount;
00263   }
00264
00265   if (debugEnabled) {
00266     Serial.printf("MotorController\n"
00267                   "------------\n"
00268                   "setSpeed(%d)\n"
00269                   "speed: %3d\n"
00270                   "target speed: %3d\n"
00271                   "pwmUpdateAmount: %3.6f\n\n",
00272                   newSpeed, speed, targetSpeed, pwmUpdateAmount);
00273   }
00274 }
00275
00278 int getSpeed() const {
00279   // Return the average
00280   return (motors[0].speed + motors[1].speed) / 2;
00281 }
00282
00285 bool countsAreUnequal(void) const {
00286   bool areUnequal = true;
00287   ALL_MOTORS(areUnequal &= motors[motor].pos == motors[motor].lastPos;)
00288   return areUnequal;
00289 }
00290
00292 void report() {
00293   Serial.printf(
00294       "MotorController\n--------------------\nSpeed: %d\nTarget "
00295       "Speed: %d\nK_p: %d\nK_i: %f\neIntegral:%f\npwmUpdateAmont: %f \n",
00296       speed, targetSpeed, K_p, K_i, eIntegral, pwmUpdateAmount);
00297   Serial.print("Leading motor: ");
00298   Serial.println(motor_roles[leadingIndex]);
00299   Serial.print("Lagging motor: ");
00300   Serial.println(motor_roles[laggingIndex]);
00301   Serial.printf("\n\n\n");
00302
00303   ALL_MOTORS_COMMAND(displayInfo)
00304 }
00305
00306 /*
00307 void pidReport(const float deltaT) const {
00308   int leaderPos = motors[0].pos;
00309   int followerPos = motors[1].pos;
00310   int followerSpeed = motors[1].speed;
00311   Direction dir = motors[1].dir;
00312
00313   pidController.report(leaderPos, followerPos, deltaT, followerSpeed, dir);
00314 }
00315
00316 */
```

```
00317
00322 void savePosition(const int slot, const int position_value) {
00323   if (slot > 0 && slot < NUM_POSITION_SLOTS && position_value > -1) {
00324     ALL_MOTORS(motors[motor].setPos(position_value);)
00325     savedPositions[slot - 1] = position_value;
00326     positionStorage.putInt(save_position_slot_names[slot], position_value);
00327   }
00328 }
00329
00332 void setPos(const int newPos) { ALL_MOTORS(motors[motor].setPos(newPos);) }
00333
00336 bool isStopped() const { return systemDirection == Direction::STOP; }
00337
00340 void update(const float deltaT = 0.0f) {
00341   if (Direction::STOP == systemDirection) {
00342     return;
00343   }
00344
00345   ALL_MOTORS_COMMAND(readPos)
00346
00347   if (Direction::EXTEND == systemDirection) {
00348     if (motors[LEADER].getNormalizedPos() <
00349         motors[FOLLOWER].getNormalizedPos()) {
00350       laggingIndex = MotorRoles::LEADER;
00351       leadingIndex = MotorRoles::FOLLOWER;
00352     } else {
00353       laggingIndex = MotorRoles::FOLLOWER;
00354       leadingIndex = MotorRoles::LEADER;
00355     }
00356   } else {
00357     if (motors[LEADER].getNormalizedPos() >
00358         motors[FOLLOWER].getNormalizedPos()) {
00359       laggingIndex = MotorRoles::LEADER;
00360       leadingIndex = MotorRoles::FOLLOWER;
00361     } else {
00362       laggingIndex = MotorRoles::FOLLOWER;
00363       leadingIndex = MotorRoles::LEADER;
00364     }
00365   }
00366
00367   if (!pid_on) {
00368     motors[leadingIndex].speed = 255;
00369     motors[laggingIndex].speed = 255;
00370     ALL_MOTORS_COMMAND(update);
00371     return;
00372   }
00373
00374   const int speedDelta = abs(speed - targetSpeed);
00375   const int currentTime = micros();
00376   const int moveTimeDelta = currentTime - softStart;
00377   const int updateTimeDelta = currentTime - lastPWMUpdate;
00378
00379   if (targetSpeed < 0) {
00380     return;
00381   }
00382
00383   if (updateTimeDelta < SOFT_MOVEMENT_PWM_UPDATE_INTERVAL_MICROS) {
00384     return;
00385   }
00386
00387   if (speedDelta >= abs(pwmUpdateAmount) &&
00388       moveTimeDelta < SOFT_MOVEMENT_MICROS) {
00389     const float newSpeed = (float)speed + pwmUpdateAmount;
00390     speed = (int)floorf(newSpeed);
00391     lastPWMUpdate = micros();
00392
00393     if (!debugEnabled) {
00394       return;
00395     }
00396
00397     const double timeSinceSoftStart =
00398         (double)(micros() - softStart) / (double)MICROS_IN_MS;
00399
00400     Serial.printf("Soft Movement PWM Update - "
00401                   "speed: %d - "
00402                   "target speed: %d - "
00403                   "time since soft start: %f ms - "
00404                   "pwmUpdateAmount: %f\n",
00405                   speed, targetSpeed, timeSinceSoftStart, pwmUpdateAmount);
00406   } else {
00407     speed = targetSpeed;
00408     RESET_SOFT_MOVEMENT
00409
00410     if (requestedDirection == Direction::STOP) {
00411       systemDirection = Direction::STOP;
00412
00413       if (debugEnabled) {
```

```
00414         Serial.println("System Direction: STOP");
00415         const double timeSinceSoftStart =
00416             (double)(micros() - softStart) / (double)MICROS_IN_MS;
00417
00418         Serial.printf("Soft Movement PWM Update - "
00419                       "speed: %d - "
00420                       "target speed: %d - "
00421                       "time since soft start: %f ms - "
00422                       "pwmUpdateAmount: %f\n",
00423                       speed, targetSpeed, timeSinceSoftStart, pwmUpdateAmount);
00424       }
00425
00426       immediateHalt();
00427     }
00428   }
00429
00430   const float error = abs(motors[laggingIndex].getNormalizedPos() -
00431                           motors[leadingIndex].getNormalizedPos());
00432
00433   eIntegral += error * deltaT;
00434
00435   const int adjustedSpeed = speed - int((error * K_p));
00436
00437   motors[leadingIndex].speed = constrain(adjustedSpeed, 0, 2 << pwmResolution);
00438   motors[laggingIndex].speed = speed;
00439
00440   ALL_MOTORS_COMMAND(update);
00441 }
00442 }
00443 ;
00444
00445 #endif // _MOTOR_CONTROLLER_HPP_
```

4.17 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 = 31 ,
  MOTOR1_RIS_PIN = 32 , MOTOR2_RPWM_PIN = 5 , MOTOR2_LPWM_PIN = 27 , MOTOR2_R_EN_PIN = 4 ,
  MOTOR2_L_EN_PIN = 12 , MOTOR2_HALL1_PIN = 14 , MOTOR2_HALL2_PIN = 13 , MOTOR2_LIS_PIN = 16 ,
  MOTOR2_RIS_PIN = 11 }

4.17.1 Enumeration Type Documentation

4.17.1.1 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)

Enumerator

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

4.18 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 = 31,
00044
00046   MOTOR1_RIS_PIN = 32,
00047
00049   MOTOR2_RPWM_PIN = 5,
00050
00052   MOTOR2_LPWM_PIN = 27,
00053
00055   MOTOR2_R_EN_PIN = 4,
00056
00058   MOTOR2_L_EN_PIN = 12,
00059
00061   MOTOR2_HALL1_PIN = 14,
00062
00064   MOTOR2_HALL2_PIN = 13,
00065
00067   MOTOR2_LIS_PIN = 16,
00068
00070   MOTOR2_RIS_PIN = 11,
00071 };
00072
00073 #endif // _MOTOR_PINS_HPP_
```

4.19 PIDController.hpp

```
00001
00003 #ifndef _PID_CONTROLLER_HPP__
00004 #define _PID_CONTROLLER_HPP__
00005
00006 #include <stdio.h>
00007 #include "defs.hpp"
00008 #include <math.h>
00009 #include <cstring>
00010
00011 class PIDController {
00012 private:
00013   float kp;               // the controller path proportional gain
00014   float ti;               // the controller's integrator time constant
00015   float td;               // the controller's derivative time constant
00016   float uMax;             // Maximum magnitude of control signal
00017   float ePrev, eIntegral; // Storage
00018
00019 public:
00020   PIDController(float kp = 0.1,
00021                 float ti = 0.002,
00022                 float td = 0.01,
00023                 float uMax = 255.0)
00024       : kp(kp), ti(ti), td(td), uMax(uMax), ePrev(0.0), eIntegral(0.0) {}
00025
00026   // A function to set the parameters
00027   void setParams(float kpIn, float kdIn, float kiIn, float uMaxIn = 255.0) {
00028     kp = kpIn;
00029     td = kdIn;
00030     ti = kiIn;
00031     uMax = uMaxIn;
00032   }
00033
00043 void evaluate(int value, int target, float deltaT, int& speed, Direction& dir) {
00044   // error
00045   int e = target - value;
00046
00047   // derivative
00048   float dedt = static_cast<float>(e - ePrev) / deltaT;
00049
00050   // integral
00051   eIntegral += e * deltaT;
00052
00053   // control signal
00054   float u = kp * e + td * dedt + ti * eIntegral;
00055
00056   // motor power
00057   speed = static_cast<int>(fabs(u));
00058   speed = speed > uMax ? uMax : speed;
00059
00060   // motor direction
00061   dir = u < 0 ? Direction::RETRACT : u > 0 ? Direction::EXTEND : Direction::STOP;
00062
00063   // store previous error
00064   ePrev = e;
00065 }
00066
00067   void report(const int value, const int target, const float deltaT, const int speed, const Direction
      dir) const {
00068     char buf[256]; // Diagnostic messages
00069     sprintf(buf, "PID params: value:%d target: %d, deltaT: %f, speed: %d, dir: %s", value, target,
      deltaT, speed, directions[static_cast<int>(dir)]);
00070     Serial.println(buf);
00071   }
00072 };
00073
00074 #endif // _PID_CONTROLLER_HPP__
```

4.20 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.

4.20.1 Function Documentation

4.20.1.1 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

4.21 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_
```

4.22 PotentiometerPins.hpp File Reference

```
#include <cstdint>
```

Enumerations

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

4.22.1 Enumeration Type Documentation

4.22.1.1 PotentiometerPins

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

Enumerator

SPEED_POT_PIN	Speed potentiometer pin
KP_POT_PIN	PID gain potentiometer pin

4.23 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_
```

4.24 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);

4.24.1 Macro Definition Documentation

4.24.1.1 LOAD_SAVED_POSITION

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

Value:

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

4.24.1.2 MOTOR_COMMAND

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

Value:

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

4.24.1.3 SET_POS_HANDLER

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

Value:

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

4.24.1.4 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.";                         \
  }
```

4.25 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